Client To Authenticator Protocol

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-client-to-authenticator-protocol-v2.0-ps-20170927.html
Previous version:
https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-client-to-authenticator-protocol-v2.0-rd-20161004.html
Editors:
Rolf Lindemann, Nok Nok Labs
Vijay Bharadwaj, Microsoft
Alexei Czeskis, Google
Michael B. Jones, Microsoft
Jeff Hodges, PayPal
Akshay Kumar, Microsoft
Christiaan Brand, Google
Johan Verrept, VASCO Data Security
Jakob Ehrensvärd, Yubico
Contributors:
Mirko J. Ploch, SurePassID
Matthieu Antoine, Gemalto

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This specification 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. The application layer protocol defines requirements for such transport protocols. Each transport binding defines the details of how such transport layer connections should be set up, in a manner that meets the requirements of the application layer protocol.

Status of This Document

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

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

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

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Overview

This section is non-normative.

This protocol is intended to be used in scenarios where a user interacts with a relying party (a website or native app) on some platform (e.g., a PC) which prompts the user to interact with 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: as a consent button, password, a PIN, a biometric or a combination of these.

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

2. Conformance

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

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

3. Protocol Structure

This section is non-normative.

This protocol is specified in three parts:

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

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

4. Protocol Overview

This section is non-normative.

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

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

5. Authenticator API

Each operation in the authenticator API can be performed independently of the others, and all operations are asynchronous. The authenticator may enforce a limit on outstanding operations to limit resource usage - in this case, the authenticator is expected to return a busy status and the host is expected to retry the operation later. Additionally, this protocol does not enforce in-order or reliable delivery of requests and responses; if these properties are desired, they must be provided by the underlying transport protocol or implemented at a higher layer by applications.

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

The authenticator API has the following methods and data structures.

5.1 authenticatorMakeCredential(0x01)

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

Parameter name	Data type	Required?	Definition
clientDataHash	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 a RP icon image. The RP name is to be used by the authenticator when displaying the credential to the user for selection and usage authorization.
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 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 [IANA-COSE-ALGS-REG]. 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 in the table below.
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 following values are defined for use in the options parameter. All options are booleans.

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

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 can't 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 pinAuth parameter is present and pinProtocol is 1, verify it by matching it against first 16 bytes of HMAC-SHA-256 of clientDataHash parameter using pinToken: `HMAC-SHA-256(pinToken, clientDataHash)`.
    - If the verification succeeds, set the "uv" bit to 1 in the response.
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID error.

   If pinAuth parameter is not present and clientPin been 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 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 options parameter is set to true:
    - If a credential for the same RP ID and account ID already exists on the authenticator, overwrite that credential.
    - Store the user parameter along the newly-created key pair.
    - If authenticator does not have enough internal storage to persist the new credential, return CTAP2_ERR_KEY_STORE_FULL.
9. Generate an attestation statement for the newly-created key using clientDataHash.

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

Member name	Data type	Required?	Definition
authData	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.

5.2 authenticatorGetAssertion(0x02)

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

Parameter name	Data type	Required?	Definition
rpId	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 the table below.
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 selected by Client.

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

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

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 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 device doesn't 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 when they were created. The first credential is the most recent credential that was created.
8. If 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 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].

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

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].
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. 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.

5.3 authenticatorGetNextAssertion(0x08)

The client calls this method when the authenticatorGetAssertion response contains the numberOfCredentials member and the number of credentials exceeds 1. This method is used to obtain the next per-credential signature for a given authenticatorGetAssertion request.

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

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

1. If authenticator does not remember any authenticatorGetAssertion parameters, return CTAP2_ERR_NOT_ALLOWED.
2. If the 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 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.

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

Client Logic

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

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

5.4 authenticatorCancel(0x03)

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

5.5 authenticatorGetInfo(0x04)

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

On success, the authenticator returns:

Member name	Data type	Required?	Definition
versions	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 per table below. The following is a list of supported options:

Option ID	Definition	Default
plat	platform device: Indicates that the device is attached to the client and therefore can't be removed and used on another client.	false
rk	resident key: Indicates that the device is capable of storing keys on the device itself and therefore can satisfy the authenticatorGetAssertion request with allowList parameter not specified or empty.	false
clientPin	Client PIN: If present and set to true, it indicates that the device is capable of accepting a PIN from the client and PIN has been set. If present and set to false, it indicates that the device is capable of accepting a PIN from the client and PIN has not been set yet. If absent, it indicates that the device is not capable of accepting a PIN from the client.	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

5.6 authenticatorClientPIN(0x06)

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

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

It takes the following input parameters:

Parameter name	Data type	Required?	Definition
pinProtocol	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	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 usingsharedSecret.
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 is:

Subcommand Name	Subcommand Number
Get Retries	1
Get Key Agreement	2
Set PIN	3
Change PIN	4
Get PIN token	5

On success, Authenticator returns the following structure in its response.

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.

5.6.1 Client PIN support requirements

- Platform has to fulfill following PIN support requirements while gathering input from the user:
  - Minimum PIN Length: 4 Unicode characters
  - Maximum PIN Length: UTF-8 representation must not exceed 255 bytes
- Authenticator has to fulfill 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 with requires 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.

5.6.2 Authenticator Configuration Operations Upon Power Up

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

Following are the operations Authenticator performs on each powerup:

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

5.6.3 Getting sharedSecret from Authenticator

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

Platform performs the following operations to arrive at the sharedSecret:

- Platform sends authenticatorClientPIN command by setting getKeyAgreement parameter to true.
  - 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 device is near authenticator lockout stage, platform can optionally warn the user to be careful while entering PIN.
- Authenticator responds back with public key of authenticatorKeyAgreementKey, "aG".
  - Authenticator optionally also sends retires count if getRetries parameter is set to true.
- Platform generates **"platformKeyAgreementKey"**:
  - Platform generates ECDH P-256 key pair called "platformKeyAgreementKey" denoted by `(b, bG)` where `"b"` denotes the private key and `"bG"` denotes the public key.
- Platform generates **"sharedSecret"**
  - Platform generates "sharedSecret" using SHA-256 over ECDH key agreement protocol using private key of platformKeyAgreementKey, "b" and public key of authenticatorKeyAgreementKey, "aG": `SHA-256((baG).x)`.
    - SHA-256 is done over only "x" curve point of baG.
    - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.

5.6.4 Setting a New PIN

Following operations are performed to set up a new PIN:

- Platform gets sharedSecret from the authenticator.
- Platform collects new PIN ("newPinUnicode") from the user in Unicode format.
  - Platform checks the Unicode character length of "newPinUnicode" against the minimum 4 Unicode character requirement and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
  - Let "newPin" be the UTF-8 representation of "newPinUnicode".

        - Platform checks the byte length of "newPin" against the max UTF-8 representation limit of 255 bytes and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
- Platform sends authenticatorClientPIN command with 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 of minimum 64 bytes length. This is to prevent leak of PIN length while communicating to the authenticator. There is no PKCS #7 padding used in this scheme.
    - pinAuth: `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc), 16)`.
        - The platform sends the first 16 bytes of the HMAC-SHA-256 result.
- Authenticator performs following operations upon receiving the request:
    - Authenticator generates "sharedSecret": `SHA-256((abG).x)` using private key of authenticatorKeyAgreementKey, "a" and public key of platformKeyAgreementKey, "bG".
        - SHA-256 is done over only `"x"` curve point of `"abG"`
        - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.
    - Authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc), 16)` and matching against input pinAuth parameter.
        - If pinAuth verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
    - Authenticator decrypts newPinEnc using above "sharedSecret" producing newPin and checks newPin length against minimum PIN length of 4 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, authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION error.
        - Authenticator may have additional constraints for PIN policy. The current spec only enforces minimum length of 4 characters.
    - Authenticator stores `LEFT(SHA-256(newPin), 16)` on the device and returns CTAP2_OK.

5.6.5 Changing existing PIN

Following operations are performed to change an existing PIN:

- Platform gets sharedSecret from the authenticator.
- Platform collects current PIN ("curPinUnicode") and new PIN ("newPinUnicode") from the user.
    - Platform checks the Unicode character length of "newPinUnicode" against the minimum 4 Unicode character requirement and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
    - Let "curPin" be the UTF-8 representation of "curPinUnicode" and "newPin" be the UTF-8 representation of "newPinUnicode"
        - Platform checks the byte length of "curPin" and "newPin" against the max UTF-8 representation limit of 255 bytes and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
- Platform sends authenticatorClientPIN command with 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 of minimum 64 bytes length. This is to prevent leak of PIN length while communicating to the authenticator. There is no PKCS #7 padding used in this scheme.
    - pinAuth: `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc || pinHashEnc), 16)`.
        - The platform sends the first 16 bytes of the HMAC-SHA-256 result.
- Authenticator performs following operations upon receiving the request:
    - Authenticator generates "sharedSecret": SHA-256((abG).x) using private key of authenticatorKeyAgreementKey, "a" and public key of platformKeyAgreementKey, "bG".
        - SHA-256 is done over only `"x"` curve point of `"abG"`
        - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.
    - Authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc || pinHashEnc), 16)` and matching against input pinAuth parameter.
        - If pinAuth verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
    - Authenticator decrypts pinHashEnc and verifies against its internal stored `LEFT(SHA-256(curPin), 16)`.
        - If a mismatch is detected, 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 [RFC6090] Section 4.1 and [SP800-56A] for more ECDH key agreement protocol details.
    - Authenticator decrypts newPinEnc using above "sharedSecret" producing newPin and checks newPin length against 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, authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION error.
        - Authenticator may have additional constraints for PIN policy. The current spec only enforces minimum length of 4 characters.
    - Authenticator stores `LEFT(SHA-256(newPin), 16)` on the device and returns CTAP2_OK.

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

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

- Platform gets sharedSecret from the authenticator.
- Platform collects PIN from the user.
- Platform sends authenticatorClientPIN command with following parameters to the authenticator:
  - keyAgreement: public key of platformKeyAgreementKey, "bG".
  - pinHashEnc: `AES256-CBC(sharedSecret, IV=0, LEFT(SHA-256(PIN),16))`.
- Authenticator performs following operations upon receiving the request:
  - Authenticator generates "sharedSecret": `SHA-256((abG).x)` using private key of authenticatorKeyAgreementKey, "a" and public key of platformKeyAgreementKey, "bG".
    - SHA-256 is done over only `"x"` curve point of `"abG"`
    - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.
  - Authenticator decrypts pinHashEnc and verifies against its internal stored `LEFT(SHA-256(curPin), 16)`.
    - If a mismatch is detected, 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 [RFC6090] Section 4.1 and [SP800-56A] for more ECDH key agreement protocol details.
  - Authenticator returns encrypted pinToken using "sharedSecret": `AES256-CBC(sharedSecret, IV=0, pinToken)`.
    - pinToken should be a multiple of 16 bytes (AES block length) without any padding or IV. There is no PKCS #7 padding used in this scheme.

5.6.7 Using pinToken

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

5.6.7.1 Using pinToken in authenticatorMakeCredential

Following operations are performed to use pinToken in authenticatorMakeCredential API:

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

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

5.6.7.2 Using pinToken in authenticatorGetAssertion

Following operations are performed to use pinToken in authenticatorGetAssertion API:

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

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

5.6.7.3 Without pinToken in authenticatorGetAssertion

Following operations are performed without using pinToken in authenticatorGetAssertion API:

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

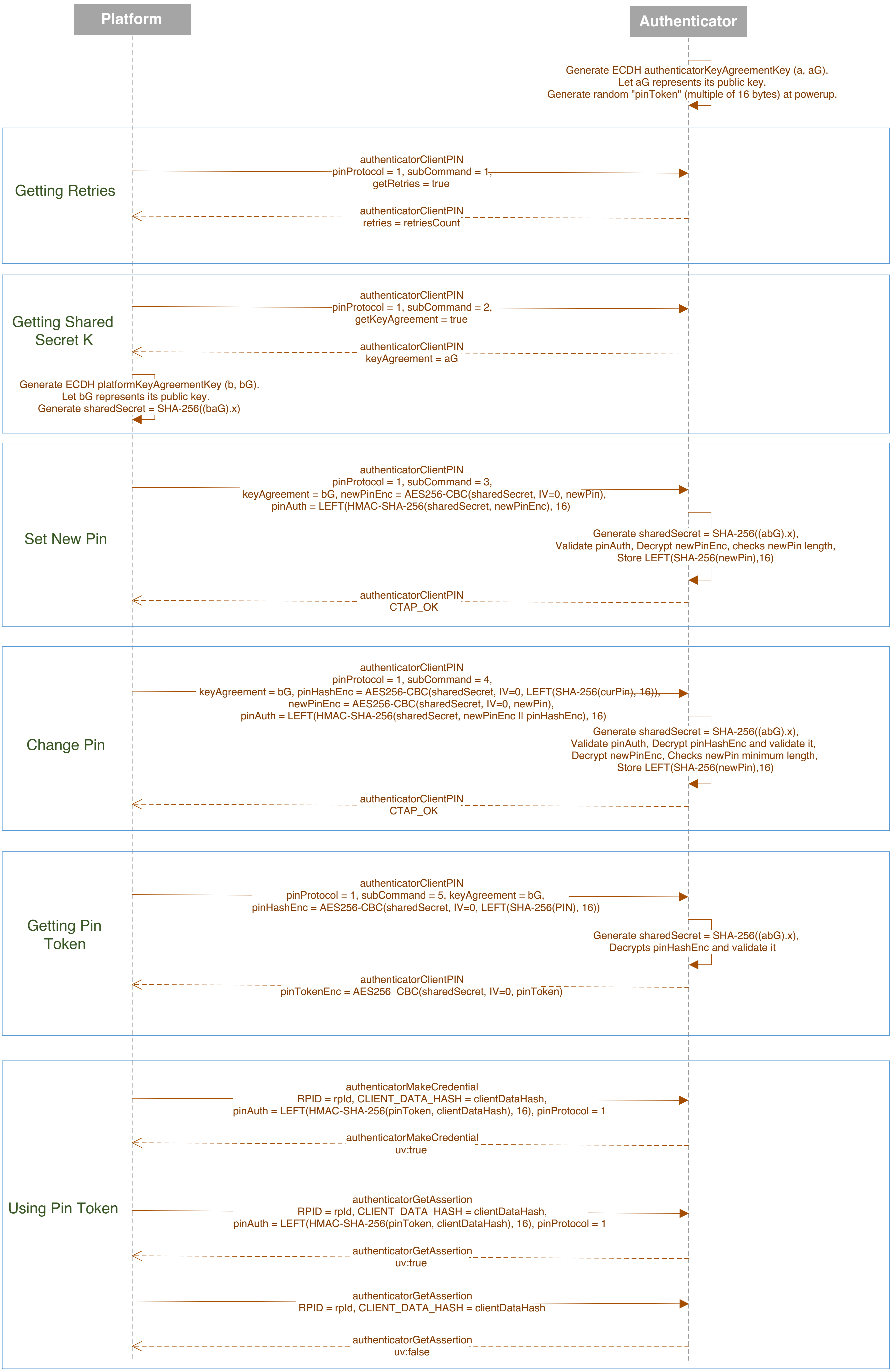
Platform
Authenticator
Generate ECDH authenticatorKeyAgreementKey (a, aG).
Let aG represents its public key.
Generate random "pinToken" (multiple of 16 bytes) at powerup.
Getting Retries
authenticatorClientPIN
pinProtocol = 1, subCommand = 1,
getRetries = true
authenticatorClientPIN
retries = retriesCount
Getting Shared Secret K
authenticatorClientPIN
pinProtocol = 1, subCommand = 2,
getKeyAgreement = true
authenticatorClientPIN
keyAgreement = aG
Generate ECDH platformKeyAgreementKey (b, bG).
Let bG represents its public key.
Generate sharedSecret = SHA-256((baG).x)
Set New Pin
authenticatorClientPIN
pinProtocol = 1, subCommand = 3,
keyAgreement = bG, newPinEnc = AES256-CBC(sharedSecret, IV=0, newPin),
pinAuth = LEFT(HMAC-SHA-256(sharedSecret, newPinEnc), 16)
Generate sharedSecret = SHA-256((abG).x),
Validate pinAuth, Decrypt newPinEnc, checks newPin length,
Store LEFT(SHA-256(newPin),16)
authenticatorClientPIN
CTAP_OK
Change Pin
authenticatorClientPIN
pinProtocol = 1, subCommand = 4,
keyAgreement = bG, pinHashEnc = AES256-CBC(sharedSecret, IV=0, LEFT(SHA-256(curPin), 16)),
newPinEnc = AES256-CBC(sharedSecret, IV=0, newPin),
pinAuth = LEFT(HMAC-SHA-256(sharedSecret, newPinEnc || pinHashEnc), 16)
Generate sharedSecret = SHA-256((abG).x),
Validate pinAuth, Decrypt pinHashEnc and validate it,
Decrypt newPinEnc, Checks newPin minimum length,
Store LEFT(SHA-256(newPin),16)
authenticatorClientPIN
CTAP_OK
Getting Pin Token
authenticatorClientPIN
pinProtocol = 1, subCommand = 5, keyAgreement = bG,
pinHashEnc = AES256-CBC(sharedSecret, IV=0, LEFT(SHA-256(PIN), 16))
Generate sharedSecret = SHA-256((abG).x),
Decrypts pinHashEnc and validate it
authenticatorClientPIN
pinTokenEnc = AES256_CBC(sharedSecret, IV=0, pinToken)
Using Pin Token
authenticatorMakeCredential
RPID = rpId, CLIENT_DATA_HASH = clientDataHash,
pinAuth = LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16), pinProtocol = 1
authenticatorMakeCredential
uv:true
authenticatorGetAssertion
RPID = rpId, CLIENT_DATA_HASH = clientDataHash,
pinAuth = LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16), pinProtocol = 1
authenticatorGetAssertion
uv:true
authenticatorGetAssertion
RPID = rpId, CLIENT_DATA_HASH = clientDataHash
authenticatorGetAssertion
uv:false

Fig. 1 Client Pin

5.7 authenticatorReset(0x07)

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

6. Message encoding

Many transports (e.g., Bluetooth Smart) are bandwidth constrained, and serialization formats such as JSON are too heavy-weight for such environments. For this reason, all encoding is done using the concise binary encoding CBOR [RFC7049].

To reduce the complexity of the messages and the resources required to parse and validate them, all messages must use 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 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 through 5 must be as short as possible. The rules for these lengths follow the above rule for integers.
- Indefinite-length items must be made into definite-length items.
- The keys in every map must be sorted lowest value to highest. 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.

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

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

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

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

6.1 Commands

All commands are structured as:

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

The assigned values for commands and their descriptions are:

Command Name	Command Value	Has parameters?
authenticatorMakeCredential	0x01	yes
authenticatorGetAssertion	0x02	yes
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 [RFC7049]. Specifically, parameters that are represented as DOM objects in the *Authenticator API* layers (formally defined in the Web API [WebAuthN]) are converted first to JSON and subsequently to CBOR.

EXAMPLE 1

A PublicKeyCredentialRpEntity DOM object defined as follows:

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

would be CBOR encoded as follows:

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

EXAMPLE 2

A PublicKeyCredentialUserEntity DOM object defined as follows:

```
var user = {
      id: Uint8Array.from(window.atob("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMII="), c=>c.charCodeAt(0)),
      icon: "https://pics.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 below:

Command	Parameter Name	Key	Value type
authenticatorMakeCredential	clientDataHash	0x01	byte string (CBOR major type 2).
	rp	0x02	CBOR definite length map (CBOR major type 5).
	user	0x03	CBOR definite length map (CBOR major type 5).

	pubKeyCredParams	0x04	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	excludeList	0x05	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	extensions	0x06	CBOR definite length map (CBOR major type 5).
	options	0x07	CBOR definite length map (CBOR major type 5).
	pinAuth	0x08	byte string (CBOR major type 2).
	pinProtocol	0x09	PIN protocol version chosen by the Client. For this version of the spec, this shall be the number 1.
authenticatorGetAssertion	rpId	0x01	UTF-8 encoded text string (CBOR major type 3).
	clientDataHash	0x02	byte string (CBOR major type 2).
	allowList	0x03	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	extensions	0x04	CBOR definite length map (CBOR major type 5).
	options	0x05	CBOR definite length map (CBOR major type 5).
	pinAuth	0x06	byte string (CBOR major type 2).
	pinProtocol	0x07	PIN protocol version chosen by the Client. For this version of the spec, this shall be the number 1.
authenticatorClientPIN	pinProtocol	0x01	Unsigned Integer. (CBOR major type 0)
	subCommand	0x02	Unsigned Integer. (CBOR major type 0)
	keyAgreement	0x03	COSE_KEY
	pinAuth	0x04	byte string (CBOR major type 2).
	newPinEnc	0x05	byte string (CBOR major type 2). It is UTF-8 representation of encrypted input PIN value.
	pinHashEnc	0x06	byte string (CBOR major type 2).
	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 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                            # text(4)
         6e616d65                   # "name"
      64                            # text(4)
         41636d65                   # "Acme"
   03                               # unsigned(3) - user
   a4                               # map(4)
      62                            # text(2)
         6964                       # "id"
      58 20                         # bytes(32)
         3082019330820138a003020102 # userid
         3082019330820138a003020102 # ...
         308201933082               # ...
      64                            # text(4)
         69636f6e                   # "icon"
      78 28                         # text(40)
         6874747073a2f2f706963732e616  # "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  # ...
                  3034313135353030305a170d3236  # ...
                  3132303231313535303305a3047  # ...
                  310b300906035504061302553  # ...
                  3114301206035504a0c0b5975  # ...
                  6269636f20546573743122302020  # ...
                  060355040b0c194175746865 6e  # ...
                  7469636174 6f722041747465 73  # ...
                  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)
                  03030303030303030303030303  # credential ID
                  03030303030303030303030303  # ...
```

```
                    030303030303030303030303  # ...
                    0303030303030303030303    # ...
                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)
```

6.2 Responses

All responses are structured as:

Name	Length	Required?	Definition
Status	1 byte	Required	The status of the response. 0x00 means success; all other values are errors. See the table in the next section for 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 below:

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).
	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].
	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.

6.3 Error Responses

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

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

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

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

Code	Name	Description
0x00	CTAP1_ERR_SUCCESS	Indicates successful response.
0x01	CTAP1_ERR_INVALID_COMMAND	The command is not a valid CTAP command.
0x02	CTAP1_ERR_INVALID_PARAMETER	The command included an invalid parameter.
0x03	CTAP1_ERR_INVALID_LENGTH	Invalid message or item length.
0x04	CTAP1_ERR_INVALID_SEQ	Invalid message sequencing.
0x05	CTAP1_ERR_TIMEOUT	Message timed out.
0x06	CTAP1_ERR_CHANNEL_BUSY	Channel busy.
0x0A	CTAP1_ERR_LOCK_REQUIRED	Command requires channel lock.
0x0B	CTAP1_ERR_INVALID_CHANNEL	Command not allowed on this cid.
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 exludeList.
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	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	Authenticator cannot cancel as it is not busy.
0x2A	CTAP2_ERR_NO_OPERATION_PENDING	No outstanding operations.

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	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.

7. Interoperating with CTAP1/U2F authenticators

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

7.1 Using the CTAP2 authenticatorMakeCredential Command with CTAP1/U2F authenticators

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

1. Platform tries to get information about the authenticator by sending authenticatorGetInfo command as specified in CTAP2 protocol overview.
    - CTAP1/U2F authenticator returns a command error or improperly formatted CBOR response. For any failure, platform may fall back to CTAP1/U2F protocol.
2. Map CTAP2 authenticatorMakeCredential request to U2F_REGISTER request.
    - Platform verifies that CTAP2 request does not have any parameters that CTAP1/U2F authenticators cannot fulfill.
        - All of the below conditions must be true for the platform to proceed to next step. If any of the below conditions is not true, platform errors out with CTAP2_ERR_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 makes it so the user still needs to touch the CTAP1/U2F authenticator before the RP gets told that the token is already registered.
    - Use clientDataHash parameter of CTAP2 request as CTAP1/U2F challenge parameter (32 bytes).
    - Let `rpIdHash` be a byte array of size 32 initialized with SHA-256 hash of `rp.id` parameter as CTAP1/U2F application parameter (32 bytes).
3. Send the U2F_REGISTER request to the authenticator as specified in [U2FRawMsgs] spec.
4. Map the U2F registration response message (see 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 length of the Credential ID initialized with CTAP1/U2F response key handle length.
            - Let `credentialID` be a `credentialIdLength` byte array initialized with CTAP1/U2F response key handle bytes.
            - Let `x9encodedUserPublicKey`be the `user public key` returned in the U2F registration response message [U2FRawMsgs]. Let `coseEncodedCredentialPublicKey` be the result of converting `x9encodedUserPublicKey`'s value from ANS X9.62 / Sec-1 v2 uncompressed curve point representation [SEC1V2] to COSE_Key representation ([RFC8152] Section 7).
            - Let `attestationData` be a byte array with following structure:

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

`credentialIdLength`	Credential ID.	Initialized with `credentialID` bytes.
77	The credential public key.	Initialized with `coseEncodedCredentialPublicKey` bytes.

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

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

5. 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

```
687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141    # clientdatahash
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE    # rpidhash
```

Sample CTAP1/U2F Response from the device

```
05                                                                  # Reserved Byte (1 Byte)
04E87625896EE4E46DC032766E8087962F36DF9DFE8B567F3763015B1990A60E    # User Public Key (65 Bytes)
1427DE612D66418BDA1950581EBC5C8C1DAD710CB14C22F8C97045F4612FB20C    # ...
91                                                                  # ...
40                                                                  # Key Handle Length (1 Byte)
3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6    # Key Handle (Key Handle Length Bytes)
54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038    # ...
3082024A30820132A0030201020204046C8822300D06092A864886F70D01010B    # X.509 Cert (Variable length Cert)
0500302E312C302A0603550403132359756269636F2055324620526F6F742043    # ...
412053657269616C203435373230303633313020170D313430383031303030    # ...
30305A180F32303530303930343030303030305A302C312A302806035504030C    # ...
2159756269636F20553246204545205365726961C20323439313832333323437    # ...
3730305930130607 2A8648CE3D020106082A8648CE3D030107034200043CCAB9    # ...
2CCB97287EE8E639437E21FCD6B6F165B2D5A3F3DB131D31C16B742BB476D8D1    # ...
E99080EB546C9BBDF556E6210FD42785899E78CC589EBE310F6CDB9FF4A33B30    # ...
39302206092B0601040182C40A020415312E332E362E312E342E312E34313438    # ...
322E312E323013060B2B0601040182E51C020101040403020430300D06092A86    # ...
4886F70D01010B050003820101009F9B052248BC4CF42CC5991FCAABAC9B651B    # ...
BE5BDCDC8EF0AD2C1C1FFB36D18715D42E78B249224F92C7E6E7A05C49F0E7E4    # ...
C881BF2E94F45E4A21833D7456851D0F6C145A29540C874F3092C934B43D222B    # ...
8962C0F410CEF1DB75892AF116B44A96F5D35ADEA3822FC7146F6004385BCB69    # ...
B65C99E7EB6919786703C0D8CD41E8F75CCA44AA8AB725AD8E799FF3A8696A6F    # ...
1B2656E631B1E40183C08FDA53FA4A8F85A05693944AE179A1339D002D15CABD    # ...
810090EC722EF5DEF9965A371D415D624B68A2707CAD97BCDD1785AF97E258F3    # ...
3DF56A031AA0356D8E8D5EBCADC74E071636C6B110ACE5CC9B90DFEACAE640FF    # ...
1BB0F1FE5DB4EFF7A95F060733F5                                        # ...
30450220324779C68F3380288A1197B6095F7A6EB9B1B1C127F66AE12A99FE85    # Signature (variable Length)
32EC23B9022100E39516AC4D61EE64044D50B415A6A4D4D84BA6D895CB5AB7A1    # ...
AA7D081DE341FA                                                      # ...
```

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)


```
{"fmt": "fido-u2f",
 "authData": h'1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE
              41000000000000000000000000000000000000000000403EBD89BF77EC509755
              EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B654D7FF945F50B5CC4E
              78055BDD396B64F78DA2C5F96200CCD415CD08FE420038A50102032620012158
              20E87625896EE4E46DC032766E8087962F36DF9DFE8B567F3763015B1990A60E
              1422582027DE612D66418BDA1950581EBC5C8C1DAD710CB14C22F8C97045F461
              2FB20C91',
 "attStmt": {"sig": h'30450220324779C68F3380288A1197B6095F7A6EB9B1B1C127F66AE12A99FE85
             32EC23B9022100E39516AC4D61EE64044D50B415A6A4D4D84BA6D895CB5AB7A1
             AA7D081DE341FA',
            "x5c": [h'3082024A30820132A0030201020204046C8822300D06092A864886F70D01010B
                    0500302E312C302A0603550403132359756269636F2055324620526F6F742043
                    412053657269616C203435373230303633313020170D3134303830313030303030
                    30305A180F32303530303930343030303030305A302C312A302806035504030C
                    2159756269636F20553246204545205365726961C20323439313832333323437
                    3730305930130607 2A8648CE3D020106082A8648CE3D030107034200043CCAB9
                    2CCB97287EE8E639437E21FCD6B6F165B2D5A3F3DB131D31C16B742BB476D8D1
                    E99080EB546C9BBDF556E6210FD42785899E78CC589EBE310F6CDB9FF4A33B30
                    39302206092B0601040182C40A020415312E332E362E312E342E312E34313438
                    322E312E323013060B2B0601040182E51C020101040403020430300D06092A86
                    4886F70D01010B050003820101009F9B052248BC4CF42CC5991FCAABAC9B651B
                    BE5BDCDC8EF0AD2C1C1FFB36D18715D42E78B249224F92C7E6E7A05C49F0E7E4
                    C881BF2E94F45E4A21833D7456851D0F6C145A29540C874F3092C934B43D222B
                    8962C0F410CEF1DB75892AF116B44A96F5D35ADEA3822FC7146F6004385BCB69
                    B65C99E7EB6919786703C0D8CD41E8F75CCA44AA8AB725AD8E799FF3A8696A6F
                    1B2656E631B1E40183C08FDA53FA4A8F85A05693944AE179A1339D002D15CABD
                    810090EC722EF5DEF9965A371D415D624B68A2707CAD97BCDD1785AF97E258F3
                    3DF56A031AA0356D8E8D5EBCADC74E071636C6B110ACE5CC9B90DFEACAE640FF
                    1BB0F1FE5DB4EFF7A95F060733F5']}}
```

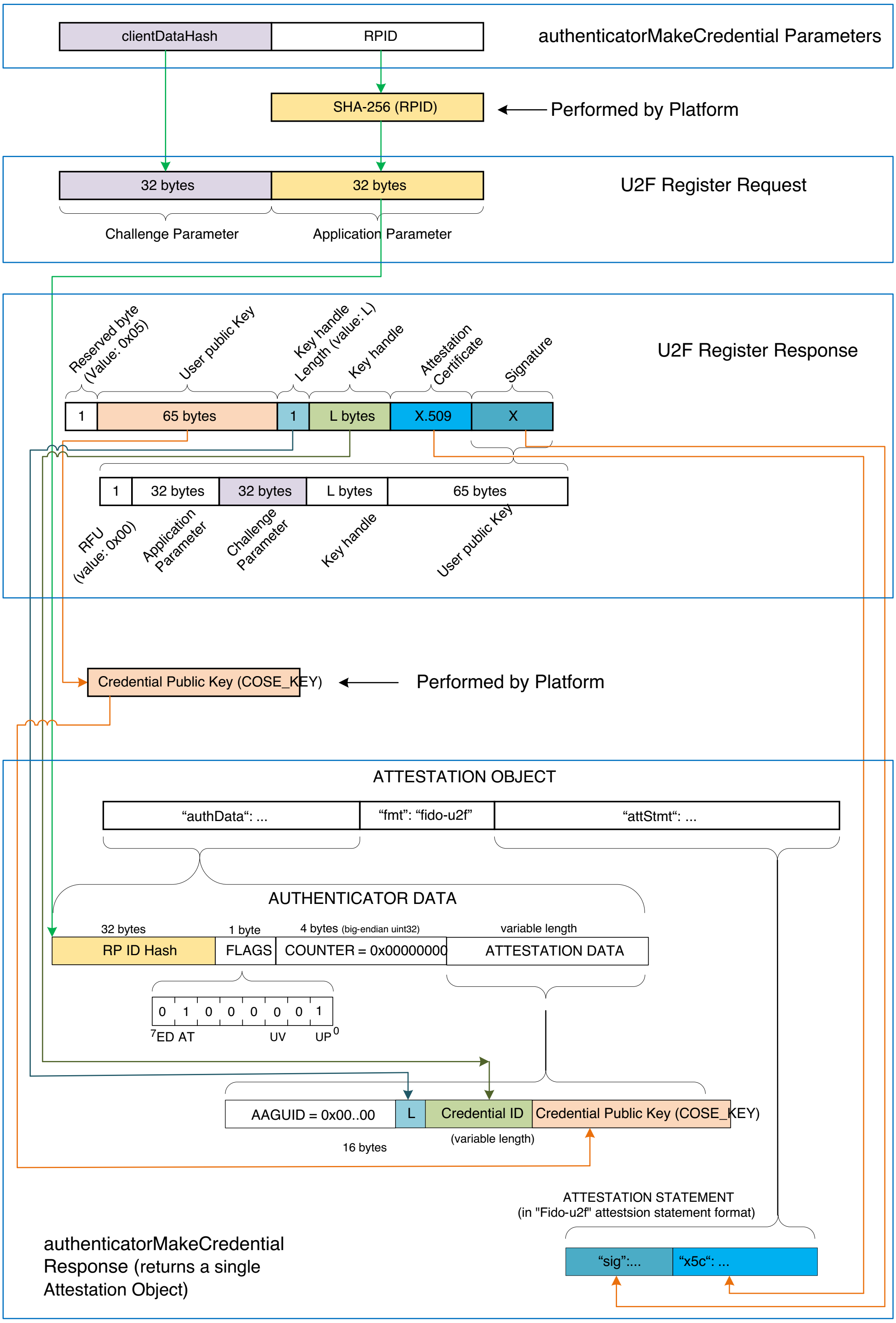

Fig. 2 Mapping: WebAuthn authenticatorMakeCredential to and from CTAP1/U2F Registration Messages.

7.2 Using the CTAP2 authenticatorGetAssertion Command with CTAP1/U2F authenticators

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

1. Platform tries to get information about the authenticator by sending authenticatorGetInfo command as specified in CTAP2 protocol overview.
   - CTAP1/U2F authenticator returns a command error or improperly formatted CBOR response. For any failure, platform may fall back to CTAP1/U2F protocol.
2. Map CTAP2 authenticatorGetAssertion request to U2F_AUTHENTICATE request:
   - Platform verifies that CTAP2 request does not have any parameters that CTAP1/U2F authenticators cannot fulfill:
     - All of the below conditions must be true for the platform to proceed to next step. If any of the below conditions is not true, platform errors out with CTAP2_ERR_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, platform has to loop over the list and send individual different U2F_AUTHENTICATE commands to the authenticator. For each credential in credential list, map CTAP2

authenticatorGetAssertion request to U2F_AUTHENTICATE as below:

- Let controlByte be a byte initialized as follows:
  - For USB, set it to 0x07 (check-only). This should prevent call getting blocked on waiting for user input. If response returns success, then call again setting the enforce-user-presence-and-sign.
  - For NFC, set it to 0x03 (enforce-user-presence-and-sign). The tap has already provided the presence and won't block.
- Use clientDataHash parameter of CTAP2 request as CTAP1/U2F challenge parameter (32 bytes).
- Let rpIdHash be a byte array of size 32 initialized with SHA-256 hash of rp.id parameter as CTAP1/U2F application parameter (32 bytes).
- Let credentialID is the byte array initialized with the id for this PublicKeyCredentialDescriptor.
- Let keyHandleLength be a byte initialized with length of credentialID byte array.
- Let u2fAuthenticateRequest be a byte array with the following structure:

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

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 Authenticator Data section of [WebAuthN].
     - Let signCount be a 4-byte unsigned integer initialized with CTAP1/U2F response counter field.
     - Let authenticatorData is a byte array of following structure:

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

   - Let authenticatorGetAssertionResponse be a CBOR map with the following keys whose values are as follows:
     - Set 0x01 with the credential from allowList that whose response succeeded.
     - Set 0x02 with authenticatorData bytes.
     - Set 0x03 with signature field from CTAP1/U2F authentication response message.

EXAMPLE 7

Sample CTAP2 authenticatorGetAssertion Request (CBOR):

```
{1: "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'}
```

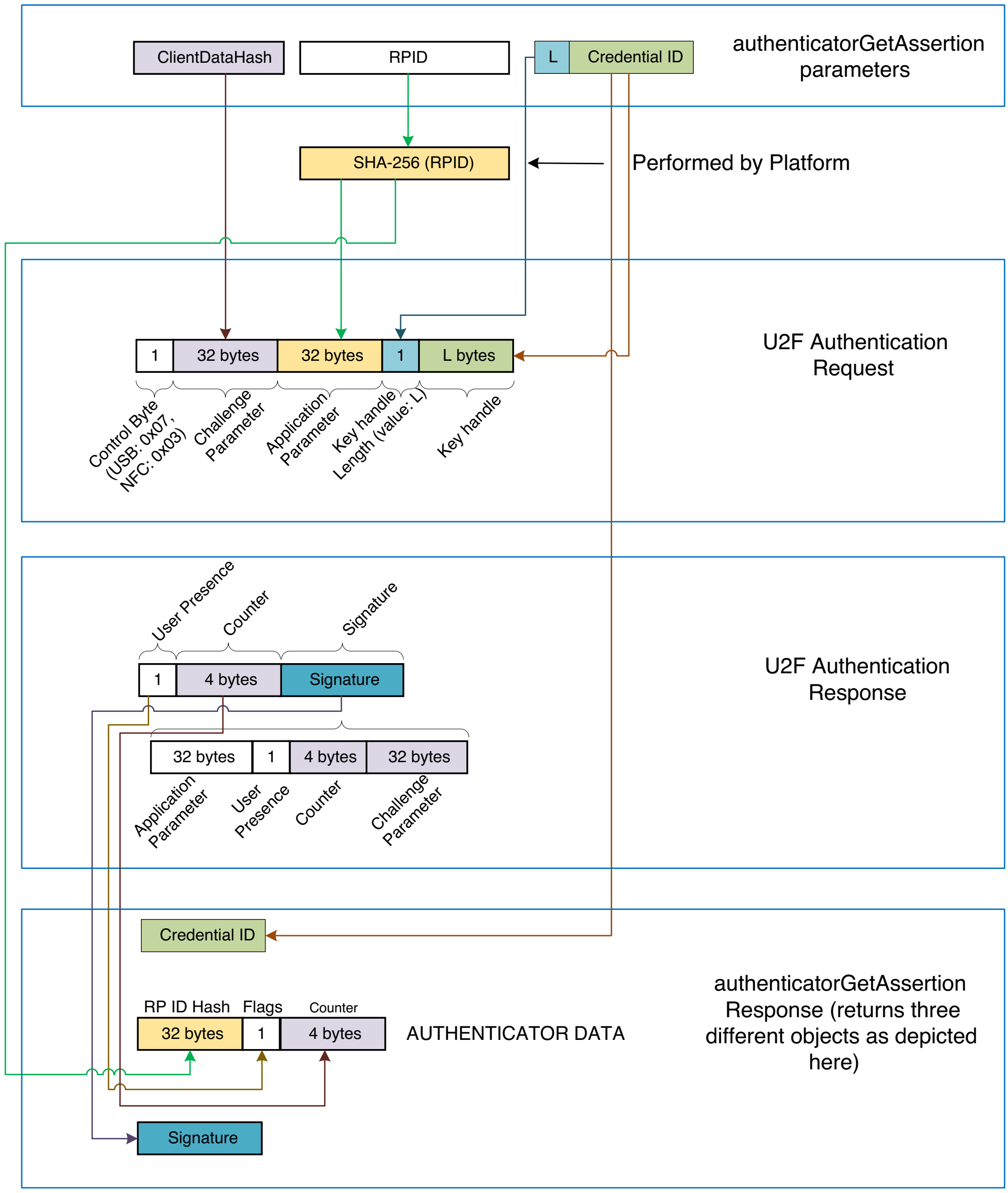

Fig. 3 Mapping: WebAuthn authenticatorGetAssertion to and from CTAP1/U2F Authentication Messages.

8. Transport-specific Bindings

8.1 USB

8.1.1 Design rationale

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

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

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

8.1.2 Protocol structure and data framing

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

- Transaction
- Message
- Packet

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

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

8.1.3 Concurrency and channels

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

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

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

8.1.4 Message and packet structure

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

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

The request and response messages have an identical structure. A transaction is started with the initialization packet of the request message and ends with the last packet of the response message.

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

An initialization packet is defined as

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

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

A continuation packet is defined as

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

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

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

8.1.5 Arbitration

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

8.1.5.1 Transaction atomicity, idle and busy states.

A transaction always consists of three stages:

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

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

The application channel that manages to get through the first initialization packet when the device is in idle state will keep the device locked for other channels until the last packet of the response message has been received. The device then returns to idle state, ready to perform another transaction for the same or a different channel. Between two transactions, no state is maintained in the device and a host application must assume that any other process may execute other transactions at any time.

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

8.1.5.2 Transaction timeout

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

8.1.5.3 Transaction abort and re-synchronization

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

8.1.5.4 Packet sequencing

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

8.1.6 Channel locking

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

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

8.1.7 Protocol version and compatibility

The CTAPHID protocol is designed to be extensible, yet 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.

8.1.8 HID device implementation

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

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

8.1.8.1 Interface and endpoint descriptors

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

Interface Descriptor

Mnemonic	Value	Description
bNumEndpoints	2	One IN and one OUT endpoint
bInterfaceClass	0x03	HID
bInterfaceSubClass	0x00	No interface subclass
bInterfaceProtocol	0x00	No interface protocol

Endpoint 1 descriptor

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

Endpoint 2 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdresss	0x81	1, IN
bMaxPacketSize	64	64-byte packet max

bInterval	5	Poll every 5 millisecond

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

8.1.8.2 HID report descriptor and device discovery

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

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

EXAMPLE 8

```
// HID report descriptor

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

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

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

8.1.9 CTAPHID commands

The CTAPHID protocol implements the following commands.

8.1.9.1 Mandatory commands

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

8.1.9.1.1 CTAPHID_MSG (0x03)

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

8.1.9.1.2 CTAPHID_CBOR (0x10)

This command sends an encapsulated CTAP CBOR encoded message. The semantics of the data message is defined in the CTAP Message encoding specification.

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

8.1.9.1.3 CTAPHID_INIT (0x06)

This command has two functions.

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

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

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

Request

CMD	CTAPHID_INIT
BCNT	8
DATA	8-byte nonce

Response at success

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

The protocol version identifies the protocol version implemented by the device. An 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 capabilities 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

8.1.9.1.4 CTAPHID_PING (0x01)

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

Request

CMD	CTAPHID_PING
BCNT	0..n
DATA	n bytes

Response at success

CMD	CTAPHID_PING
BCNT	n
DATA	N bytes

8.1.9.1.5 CTAPHID_CANCEL (0x11)

Cancel any outstanding requests on this CID.

Request

CMD	CTAPHID_CANCEL
BCNT	0

Response at success

CMD	CTAPHID_CANCEL
BCNT	0

8.1.9.1.6 CTAPHID_ERROR (0x3F)

This command code is used in response messages only.

CMD	CTAPHID_ERROR
BCNT	1
DATA	Error code

The following error codes are defined

ERR_INVALID_CMD	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

8.1.9.1.7 CTAPHID_KEEPALIVE (0x3B)

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

CMD	CTAPHID_KEEPALIVE
BCNT	1
DATA	Status code

The following status codes are defined

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

8.1.9.2 Optional commands

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

8.1.9.2.1 CTAPHID_WINK (0x08)

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

Request

CMD	CTAPHID_WINK
BCNT	0
DATA	N/A

Response at success

CMD	CTAPHID_WINK
BCNT	0
DATA	N/A

8.1.9.2.2 CTAPHID_LOCK (0x04)

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

Request

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

Response at success

CMD	CTAPHID_LOCK
BCNT	0
DATA	N/A

8.1.9.3 Vendor specific commands

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

8.2 ISO7816, ISO14443 and Near Field Communication (NFC)

8.2.1 Conformance

Please refer to [ISOIEC-7816-4-2013] for APDU definition.

8.2.2 Protocol

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

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

8.2.3 Applet selection

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

The FIDO2 AID consists of the following fields:

Field	Value
RID	0xA000000647
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 deviceshall 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.

8.2.4 Framing

Conceptually, framing defines an encapsulation of FIDO2 commands. In NFC, this encapsulation is done in an APDU following [ISOIEC-7816-4-2013]. Fragmentation, if needed, is discussed in the following paragraph.

8.2.4.1 Commands

Commands shall have the following format:

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

8.2.4.2 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 will immediately issue an NFCCTAP_GETREPONSE command unless a cancel was issued. CTAP will provide the status data to the higher layers.
Errors		See [ISOIEC-7816-4-2013]

8.2.5 Fragmentation

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

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

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

Some responses may not fit into a short APDU response. For this reason, FIDO2 authenticators must respond in the following way:

- If the request was encoded using *extended length* APDU encoding, the authenticator must respond using the extended length APDU response format.
- If the request was encoded using *short* APDU encoding, the authenticator must respond using ISO 7816-4 APDU chaining.

8.2.6 Commands

8.2.6.1 NFCCTAP_MSG (0x10)

The NFCCTAP_MSG command send a CTAP message to the authenticator. This commandshall 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.

NFCCTAP_GETRESPONSE (0x11)

The NFCCTAP_GETRESPONSE command is issued up to receiving 0x9100 unless a cancel was issued. This command shall return a 0x9100 result with a status indication if it has a status update, the reply to the request with a 0x9000 result code to indicate success or an error value.

All values for P1 and P2 are RFU and must be set to 0x00.

8.2.7 Bluetooth Smart / Bluetooth Low Energy Technology

8.2.7.1 Conformance

Authenticator and Client devices using Bluetooth Low Energy Technology shall conform to Bluetooth Core Specification 4.0 or later [BTCORE]

Bluetooth SIG specified UUID values shall be found on the Assigned Numbers website [BTASSNUM]

8.2.7.2 Pairing

Bluetooth Low Energy Technology is a long-range wireless protocol and thus has several implications for privacy, security, and overall user-experience. Because it is wireless, Bluetooth Low Energy Technology may be subject to monitoring, injection, and other network-level attacks.

For these reasons, Clients and Authenticators must create and use a long-term link key (LTK) and shall encrypt all communications. Authenticator must never use short term keys.

Because Bluetooth Low Energy Technology has poor ranging (*i.e.,* there is no good indication of proximity), it may not be clear to a FIDO Client with which Bluetooth Low Energy Technology Authenticator it should communicate. Pairing is the only mechanism defined in this protocol to ensure that FIDO Clients are interacting with the expected Bluetooth Low Energy Technology Authenticator. As a result, Authenticator manufacturers should instruct users to avoid performing Bluetooth pairing in a public space such as a cafe, shop or train station.

One disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This issue is discussed further in Implementation Considerations.

8.2.7.3 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.

8.2.7.4 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 discussed next for each type of transport layer.

8.2.7.4.1 Request from Client to Authenticator

Request frames must have the following format

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

Supported commands are `PING`, `MSG` and `CANCEL`. The constant values for them are described below.

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

The data format for the `MSG` command is defined in the Message Encoding section of this document.

8.2.7.4.2 Response from Authenticator to Client

Response frames must have the following format, which share a similar format to the request frames:

Offset	Length	Mnemonic	Description
0	1	`STAT`	Response status
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

When the status byte in the response is the same as the command byte in the request, the response is a successful response. The value `ERROR` indicates an error, and the response data contains an error code as a variable-length, big-endian integer. The constant value for `ERROR` is described below.

Note that the errors sent in this response are errors at the encapsulation layer, *e.g.,* indicating an incorrectly formatted request, or possibly an error communicating with the Authenticator's FIDO message processing layer. Errors reported by the FIDO message processing layer itself are considered a success from the encapsulation layer's point of view, and are reported as a complete `MSG` response.

Data format is defined in the Message Encoding section of this document.

8.2.7.4.3 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
`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

8.2.7.5 GATT Service Description

This profile defines two roles: FIDO Authenticator and FIDO Client.

- The FIDO Client shall be a GATT Client
- The FIDO Authenticator shall be a GATT Server

The following figure illustrates the mandatory services and characteristics that shall be offered by a FIDO Authenticator as part of its GATT server:

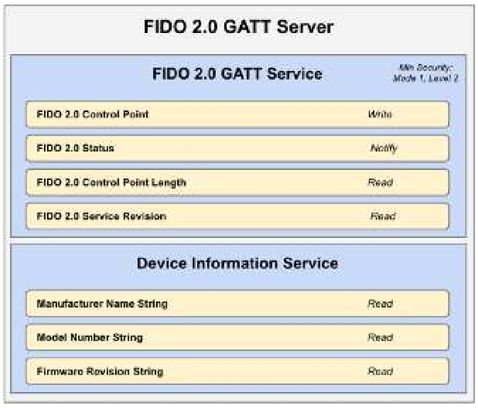

Fig. 4 Mandatory GATT services and characteristics that must be offered by a FIDO Authenticator. Note that the Generic

Access Service ([BTGAS] is not present as it is already mandatory for any Bluetooth Low Energy Technology compliant device.

The table below summarizes additional GATT sub-procedure requirements for a FIDO Authenticator (GATT Server) beyond those required by all GATT Servers.

GATT Sub-Procedure	Requirements
Write Characteristic Value	Mandatory
Notifications	Mandatory
Read Characteristic Descriptors	Mandatory
Write Characteristic Descriptors	Mandatory

The table below summarizes additional GATT sub-procedure requirements for a FIDO Client (GATT Client) beyond those required by all GATT Clients.

GATT Sub-Procedure	Requirements
Discover All Primary Services	(*)
Discover Primary Services by Service UUID	(*)
Discover All Characteristics of a Service	(**)
Discover Characteristics by UUID	(**)
Discover All Characteristic Descriptors	Mandatory
Read Characteristic Value	Mandatory
Write Characteristic Value	Mandatory
Notification	Mandatory
Read Characteristic Descriptors	Mandatory
Write Characteristic Descriptors	Mandatory

(*): Mandatory to support at least one of these sub-procedures.

(**): Mandatory to support at least one of these sub-procedures.

Other GATT sub-procedures may be used if supported by both client and server.

Specifics of each service are explained below. In the following descriptions: all values are big-endian coded, all strings are in UTF-8 encoding, and any characteristics not mentioned explicitly are optional.

8.2.7.5.1 FIDO Service

An Authenticator shall implement the FIDO Service described below. The UUID for the FIDO GATT service is `0xFFFD`, it shall be declared as a Primary Service. The service contains the following characteristics:

Characteristic Name	Mnemonic	Property	Length	UUID
FIDO Control Point	`fidoControlPoint`	Write	Defined by Vendor (20-512 bytes)	F1D0FFF1-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Status	`fidoStatus`	Notify	N/A	F1D0FFF2-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Control Point Length	`fidoControlPointLength`	Read	2 bytes	F1D0FFF3-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Service Revision Bitfield	`fidoServiceRevisionBitfield`	Read/Write	Defined by Vendor (1+ bytes)	F1D0FFF4-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Service Revision	`fidoServiceRevision`	Read	Defined by Vendor (20-512 bytes)	0x2A28

`fidoControlPoint` is a write-only command buffer.

`fidoStatus` is a notify-only response attribute. The Authenticator will send a series of notifications on this attribute with a maximum length of (ATT_MTU-3) using the response frames defined above. This mechanism is used because this results in a faster transfer speed compared to a notify-read combination.

`fidoControlPointLength` defines the maximum size in bytes of a single write request to `fidoControlPoint`. This value shall be between 20 and 512.

`fidoServiceRevision` is a deprecated field that is only relevant to U2F 1.0 support. It defines the revision of the U2F Service. The value is a UTF-8 string. For version 1.0 of the specification, the value `fidoServiceRevision` shall be `1.0` or in raw bytes: `0x312e30`. This field shall be omitted if protocol version 1.0 is not supported.

The `fidoServiceRevision` Characteristic may include a Characteristic Presentation Format descriptor with format value 0x19, `UTF-8 String`.

`fidoServiceRevisionBitfield` defines the revision of the FIDO Service. The value is a bit field which each bit representing a version. For each version bit the value is 1 if the version is supported, 0 if it is not. The length of the bitfield is 1 or more bytes. All bytes that are 0 are omitted if all the following bytes are 0 too. The byte order is big endian. The client shall write a value to this characteristic with exactly 1 bit set before sending any FIDO commands unless u2fServiceRevision is present and U2F 1.0 compatibility is desired. If only U2F version 1.0 is supported, this characteristic shall be omitted.

Byte (left to right)	Bit	Version
0	7	U2F 1.1

0	6	U2F 1.2
0	5	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 value 0x20.

8.2.7.5.2 Device Information Service

An Authenticator shall implement the Device Information Service [BTDIS] with the following characteristics:

- Manufacturer Name String
- Model Number String
- Firmware Revision String

All values for the Device Information Service are left to the vendors. However, vendors should not create uniquely identifiable values so that Authenticators do not become a method of tracking users.

8.2.7.5.3 Generic Access Profile Service

Every Authenticator shall implement the Generic Access Profile Service [BTGAS] with the following characteristics:

- Device Name
- Appearance

8.2.7.6 Protocol Overview

The general overview of the communication protocol follows:

1. Authenticator advertises the FIDO Service.
2. Client scans for Authenticator advertising the FIDO Service.
3. Client performs characteristic discovery on the Authenticator.
4. If not already paired, the Client and Authenticatorshall perform BLE pairing and create a LTK. Authenticator shall only allow connections from previously bonded Clients without user intervention.
5. Client checks if the `fidoServiceRevisionBitfield` characteristic is present. If so, the client selects a supported version by writing a value with a single bit set.
6. Client reads the `fidoControlPointLength` characteristic.
7. Client registers for notifications on the `fidoStatus` characteristic.
8. Client writes a request (*e.g.*, an enroll request) into the `fidoControlPoint` characteristic.
9. 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.

8.2.7.7 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 [BTXPLAD]).

When advertising in pairing mode, the Authenticatorshall either: (1) set the LE Limited Mode bit to zero and the LE General Discoverable bit to one OR (2) set the LE Limited Mode bit to one and the LE General Discoverable bit to zero. When advertising in non-pairing mode, the Authenticator shall set both the LE Limited Mode bit and the LE General Discoverable Mode bit to zero in the Advertising Data Flags.

The advertisement may also carry a device name which is distinctive and user-identifiable. For example, "ACME Key" would be an appropriate name, while "XJS4" would not be.

The Authenticator shall also implement the Generic Access Profile [BTGAP] and Device Information Service [BTDIS], both of which also provide a user-friendly name for the device that could be used by the Client.

It is not specified when or how often an Authenticator should advertise, instead that flexibility is left to manufacturers.

8.2.7.8 Requests

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

8.2.7.9 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 processing the commands. While the Authenticator is processing the request the KEEPALIVE command will contain status `PROCESSING`. If the Authenticator is waiting to complete the Test of User Presence, the KEEPALIVE command will contains status `UP_NEEDED`. While waiting to complete the Test of User Presence, the Authenticator may alert the user (e.g., by flashing) in order to prompt the user to complete the test of user presence. As soon the Authenticator has completed processing and confirmed user presence, it shall stop sending KEEPALIVE commands and send the reply.

Upon receiving a KEEPALIVE command, the Clientshall assume the Authenticator is still processing the command; the Client shall not resend the command. The Authenticator shall continue sending KEEPALIVE messages at least every `kKeepAliveMillis` to indicate that it is still handling the request. Until a client-defined timeout occurs, the Client shall not move on to other devices when it receives a KEEPALIVE with `UP_NEEDED` status, as it knows this is a device that can satisfy its request.

8.2.7.10 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 not currently meant for large messages. Frames are fragmented in the following way:

A frame is divided into an *initialization fragment* and one or more *continuation fragments*.

An initialization fragment is defined as:

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	0 to (maxLen - 3)	`DATA`	Data

where `maxLen` is the maximum packet size supported by the characteristic or notification.

In other words, the start of an initialization fragment is indicated by setting the high bit in the first byte. The subsequent two bytes indicate the total length of the frame, in big-endian order. The first `maxLen` - 3 bytes of data follow.

Continuation fragments are defined as:

Offset	Length	Mnemonic	Description
0	1	`SEQ`	Packet sequence 0x00..0x7f (high bit always cleared)
1	0 to (maxLen - 1)	`DATA`	Data

where `maxLen` is the maximum packet size supported by the characteristic or notification.

In other words, continuation fragments begin with a sequence number, beginning at 0, implicitly with the high bit cleared. The sequence number must wrap around to 0 after reaching the maximum sequence number of 0x7f.

Example for sending a `PING` command with 40 bytes of data with a `maxLen` of 20 bytes:

Frame	Bytes
0	`[810028] [17 bytes of data]`
1	`[00] [19 bytes of data]`
2	`[01] [4 bytes of data]`

Example for sending a ping command with 400 bytes of data with a `maxLen` of 512 bytes:

Frame	Bytes
0	`[810190] [400 bytes of data]`

8.2.7.11 Notifications

A client needs to register for notifications before it can receive them. Bluetooth Core Specification 4.0 or later [BTCORE] forces a device to remember the notification registration status over different connections [BTCCC]. Unless a client explicitly unregisters for notifications, the registration will be automatically restored when reconnecting. A client may therefor check the notification status upon connection and only register if notifications aren't already registered. Please note that some clients may disable notifications from a power management point of view (see below) and the notification registration is remembered per bond, not per client. A client must not remember the notification status in its own data storage.

8.2.7.12 Implementation Considerations

8.2.7.12.1 Bluetooth pairing: Client considerations

As noted in the Pairing section, 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 Client operating system security is partly out of FIDO's scope, further revisions of this specification may propose mitigations for this issue.

8.2.7.12.2 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 accidentally **especially** if the pairing method is Just Works. For example, the action could be pressing a physically recessed button or pressing multiple buttons. A visible or audible cue that the Authenticator is in Pairing Mode should be considered. As a counter example, a silent, long press of a single non-recessed button is not advised as some users naturally hold buttons down during regular operation.

Note that at times, Authenticators may legitimately receive communication from an unpaired device. For example, a user attempts to use an Authenticator for the first time with a new Client: he turns it on, but forgets to put the Authenticator into pairing mode. In this situation, after connecting to the Authenticator, the Client will notify the user that he needs to pair his Authenticator. The Authenticator should make it easy for the user to do so, e.g., by not requiring the user to wait for a timeout before being able to enable pairing mode.

Some Client platforms (most notably iOS) do not expose the AD Flag LE Limited and General Discoverable Mode bits to applications. For this reason, Authenticators are also strongly recommended to include the Service Data field [BTSD] in the Scan Response. The Service Data field is 3 or more octets long. This allows the Flags field to be extended while using the minimum number of octets within the data packet. All octets that are 0x00 are not transmitted as long as all other octets after that octet are also 0x00 and it is not the first octet after the service UUID. The first 2 bytes contain the FIDO Service UUID, the following bytes are flag bytes.

To help Clients show the correct UX, Authenticators can use the Service Data field to specify whether or not Authenticators will require a Passkey (PIN) during pairing.

Service Data Bit	Meaning (if set)
7	Device is in pairing mode.
6	Device requires Passkey Entry [BTPESTK].

8.2.7.13 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 anticipating more commands.

A further consideration is to ensure that a user is not confused about which command she is confirming by completing the test of user presence. That is, if a user performs the test of user presence, that action should perform exactly one operation.

We combine these considerations into the following series of recommendations:

- Upon initial connection to an Authenticator, and upon receipt of a response from an Authenticator, if a Client has more commands to issue, the Client must transmit the next command or fragment within `kMaxCommandTransmitDelayMillis` milliseconds.
- Upon final response from an Authenticator, if the Client decides it has no more commands to send it should indicate this by disabling notifications on the `fidoStatus` characteristic. When the notifications are disabled the Authenticator may enter a low power state or disconnect and shut down.
- Any time the Client wishes to send a FIDO message, it must have first enabled notifications on the `fidoStatus` characteristic and wait for the ATT acknowledgement to be sure the Authenticator is ready to process messages.
- Upon successful completion of a command which required a test of user presence, e.g. upon a successful authentication or registration command, the Authenticator can assume the Client is satisfied, and may reset its state or power down.
- Upon sending a command response that did not consume a test of user presence, the Authenticator must assume that the Client may wish to initiate another command, and leave the connection open until the Client closes it or until a timeout of at least `kErrorWaitMillis` elapses. Examples of command responses that do not consume user presence include failed authenticate or register commands, as well as get version responses, whether successful or not. After `kErrorWaitMillis` milliseconds have elapsed without further commands from a Client, an Authenticator may reset its state or power down.

Constant	Value
`kMaxCommandTransmitDelayMillis`	1500 milliseconds
`kErrorWaitMillis`	2000 milliseconds
`kKeepAliveMillis`	500 milliseconds

8.2.7.14 Data throughput

Bluetooth Low Energy Technology does not have particularly high throughput, this can cause noticeable latency to the user if request/responses are large. Some ways that implementers can reduce latency are:

- Support the maximum MTU size allowable by hardware (up to the 512-byte max from the BLE specifications).
- Make the attestation certificate as small as possible; do not include unnecessary extensions.

8.2.7.15 Advertising

Though the standard does not appear to mandate it (in any way that we've found thus far), advertising and device discovery seems to work better when the Authenticators advertise on all 3 advertising channels and not just one.

8.2.7.16 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.

A. References

A.1 Normative references

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

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

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

[U2FRawMsgs]
D. Balfanz. *FIDO U2F Raw Message Formats v1.0* Draft. URL: 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. *Web Authentication: An API for accessing Public Key Credentials Level 1*. 20 March 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/webauthn/

A.2 Informative references

[BTASSNUM]
Bluetooth Assigned Numbers. URL: https://www.bluetooth.org/en-us/specification/assigned-numbers

[BTCCC]
Client Characteristic Configuration. Bluetooth Core Specification 4.0, Volume 3, Part G, Section 3.3.3.3 URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTCORE]
Bluetooth Core Specification 4.0. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTDIS]
Device Information Service v1.1. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTGAP]
Generic Access Profile. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12 URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTGAS]
Generic Access Profile service. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12 URL: https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.generic_access.xml

[BTPESTK]
Passkey Entry. Bluetooth Core Specification 4.0, Volume 3, Part H, Section 2.3.5.3 URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTSD]
Bluetooth Service Data AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11 URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTXPLAD]
Bluetooth TX Power AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11 URL: https://www.bluetooth.com/specifications/adopted-specifications

[IANA-COSE-ALGS-REG]
Jim Schaad; Göran Selander; Derek Atkins; Sean Turner. *IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry* URL: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

[ISOIEC-7816-4-2013]
ISO 7816-4: Identification cards – Integrated circuit cards; Part 4 : Organization, security and commands for interchange URL:

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

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

[SP800-56A]
NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised). March 2007 URL: https://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf

FIDO AppID and Facet Specification

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-appid-and-facets-v2.0-ps-20170927.html
Previous version:
https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-appid-and-facets-v2.0-rd-20161004.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Brad Hill, PayPal, Inc.
Dirk Balfanz, Google, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The FIDO family of protocols introduce a new security concept, *Application Facets*, to describe the scope of user credentials and how a trusted computing base which supports application isolation may make access control decisions about which keys can be used by which applications and web origins.

This document describes the motivations for and requirements for implementing the Application Facet concept and how it applies to the FIDO protocols.

Status of This Document

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

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

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as`code`.

String literals are enclosed in “”, e.g. “UAF-TLV”.

In formulas we use “|” to denote byte wise concatenation operations.

This document applies to both the U2F protocol and the UAF protocol. UAF specific terminology used in this document is defined in [FIDOGlossary].

All diagrams, examples, notes in this specification are non-normative.

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

Modern networked applications typically present several ways that a user can interact with them. This document introduces the concept of an *Application Facet* to describe the identities of a single logical application across various platforms. For example, the application MyBank may have an Android app, an iOS app, and a Web app accessible from a browser. These are all facets of the MyBank application.

The FIDO architecture provides for simpler and stronger authentication than traditional username and password approaches while avoiding many of the shortfalls of alternative authentication schemes. At the core of the FIDO protocols are challenge and response operations performed with a public/private keypair that serves as a user's credential.

To minimize frequently-encountered issues around privacy, entanglements with concepts of "identity", and the necessity for trusted third parties, keys in FIDO are tightly scoped and dynamically provisioned between the user and each Relying Party and only optionally associated with a server-assigned username. This approach contrasts with, for example, traditional PKIX client certificates as used in TLS, which introduce a trusted third party, mix in their implementation details identity assertions with holder-of-key cryptographic proofs, lack audience restrictions, and may even be sent in the cleartext portion of a protocol handshake without the user's notification or consent.

While the FIDO approach is preferable for many reasons, it introduces several challenges.

- What set of Web origins and native applications (facets) make up a single logical application and how can they be reliably identified?
- How can we avoid making the user register a new key for each web browser or application on their device that accesses services controlled by the same target entity?
- How can access to registered keys be shared without violating the security guarantees around application isolation and protection from malicious code that users expect on their devices?
- How can a user roam credentials between multiple devices, each with a user-friendly Trusted Computing Base for FIDO?

This document describes how FIDO addresses these goals (where adequate platform mechanisms exist for enforcement) by allowing an application to declare a credential scope that crosses all the various facets it presents to the user.

2.1 Motivation

FIDO conceptually sets a scope for registered keys to the tuple of (Username, Authenticator, Relying Party). But what constitutes a Relying Party? It is quite common for a user to access the same set of services from a Relying Party, on the same device, in one or more web browsers as well as one or more dedicated apps. As the Relying Party may require the user to perform a costly ceremony in order to prove her identity and register a new FIDO key, it is undesirable that the user should have to repeat this ceremony multiple times on the same device, once for each browser or app.

2.2 Avoiding App-Phishing

FIDO provides for user-friendly verification ceremonies to allow access to registered keys, such as entering a simple PIN code and touching a device, or scanning a finger. It should not matter for security purposes if the user re-uses the same verification inputs across Relying Parties, and in the case of a biometric, she may have no choice.

Modern operating systems that use an "app store" distribution model often make a promise to the user that it is "safe to try" any app. They do this by providing strong isolation between applications, so that they may not read each others' data or mutually interfere, and by requiring explicit user permission to access shared system resources.

If a user were to download a maliciously constructed game that instructs her to activate her FIDO authenticator in order to "save your progress" but actually unlocks her banking credential and takes over her account, FIDO has failed, because the risk of phishing has only been moved from the password to an app download. FIDO must not violate a platform's promise that any app is "safe to try" by keeping good custody of the high-value shared state that a registered key represents.

2.3 Comparison to OAuth and OAuth2

The OAuth and OAuth2 of protocols were designed for a server-to-server security model with the assumption that each application instance can be issued, and keep, an "application secret". This approach is ill-suited to the "app store" security model. Although it is common for services to provision an OAuth-style application secret into their apps in an attempt to allow only authorized/official apps to connect, any such "secret" is in fact shared among everyone with access to the app store and can be trivially recovered thorough basic reverse engineering.

In contrast, FIDO's facet concept is designed for the "app store" model from the start. It relies on client-side platform isolation features to make sure that a key registered by a user with a member of a well-behaved "trusted club" stays within that trusted club, even if the user later installs a malicious app, and does not require any secrets hard-coded into a shared package to do so. The user must, however, still make good decisions about which apps and browsers they are willing to preform a registration ceremony with. App store policing can assist here by removing applications which solicit users to register FIDO keys to for Relying Parties in order to make illegitmate or fraudulent use of them.

2.4 Non-Goals

The *Application Facet* concept does not attempt to strongly identify the calling application to a service across a network. Remote attestation of an application identity is an explicit non-goal.

If an unauthorized app can convince a user to provide all the information to it required to register a new FIDO key, the Relying Party cannot use FIDO protocols or the Facet concept to recognize as unauthorized, or deny such an application from performing FIDO operations, and an application that a user has chosen to trust in such a manner can also share access to a key outside of the mechanisms described in this document.

The facet mechanism provides a way for registered keys to maintain their proper scope when created and accessed from a *Trusted Computing Base* (TCB) that provides isolation of malicious apps. A user can also roam their credentials between multiple devices with user-friendly TCBs and credentials will retain their proper scope if this mechanism is correctly implemented by each. However, no guarantees can be made in environments where the TCB is user-hostile, such as a device with malicious code operating with "root" level permissions. On environments that do not provide application isolation but run all code with the privileges of the user, (e.g. traditional desktop operating systems) an intact TCB, including web browsers, may successfully enforce scoping of credentials for web origins only, but cannot meaningfully enforce application scoping.

3. The AppID and FacetID Assertions

When a user performs a Registration operation [UAFArchOverview] a new private key is created by their authenticator, and the public key is sent to the Relying Party. As part of this process, each key is associated with an `AppID`. The `AppID` is a URL carried as part of the protocol message sent by the server and indicates the target for this credential. By default, the audience of the credential is restricted to the *Same Origin* of the `AppID`. In some circumstances, a Relying Party may desire to apply a larger scope to a key. If that `AppID` URL has the `https` scheme, a FIDO client may be able to dereference and process it as a **`TrustedFacetList`** that designates a scope or audience restriction that includes multiple facets, such as other web origins within the same DNS zone of control of the AppID's origin, or URLs indicating the identity of other types of trusted facets such as mobile apps.

> NOTE
>
> Users may also register multiple keys on a single authenticator for an `AppID`, such as for cases where they have multiple accounts. Such registrations may have a Relying Party assigned username or local nicknames associated to allow them to be distinguished by the user, or they may not (e.g. for 2nd factor use cases, the user account associated with a key may be communicated out-of-band to what is specified by FIDO protocols). All registrations that share an `AppID`, also share these same audience restriction.

3.1 Processing Rules for AppID and FacetID Assertions

3.1.1 Determining the FacetID of a Calling Application

In the Web case, the FacetID must be the Web Origin [RFC6454] of the web page triggering the FIDO operation, written as a URI with an empty path. Default ports are omitted and any path component is ignored.

An example FacetID is shown below:

```
https://login.mycorp.com/
```

In the Android [ANDROID] case, the FacetID must be a URI derived from the Base64 encoded SHA-256 (or SHA-1) hash of the APK signing certificate [APK-Signing]:

```
android:apk-key-hash-sha256:<base64_encoded_sha256_hash-of-apk-signing-cert>
```

```
android:apk-key-hash:<base64_encoded_sha1_hash-of-apk-signing-cert>
```

The SHA-1 hash can be computed as follows:

EXAMPLE 1: Computing an APK signing certificate SHA256 hash

```
# Export the signing certificate in DER format, hash, base64 encode and trim '='

keytool -exportcert \
        -alias <alias-of-entry> \
        -keystore <path-to-apk-signing-keystore> &>2 /dev/null | \
        openssl sha256 -binary | \
        openssl base64 | \
        sed 's/=//g'
```

EXAMPLE 2: Computing an APK signing certificate SHA1 hash

```
# Export the signing certificate in DER format, hash, base64 encode and trim '='

keytool -exportcert \
        -alias <alias-of-entry> \
        -keystore <path-to-apk-signing-keystore> &>2 /dev/null | \
        openssl sha1 -binary | \
        openssl base64 | \
        sed 's/=//g'
```

The Base64 encoding is the the "Base 64 Encoding" from Section 4 in [RFC4648], with padding characters removed.

> NOTE
>
> If compatibility with older versions of FIDO Clients (i.e. the ones not yet supporting SHA-256 for FacetIDs) is required, both entries should be specified.

In the iOS [iOS] case, the FacetID must be the BundleID [BundleID] URI of the application:

```
ios:bundle-id:<ios-bundle-id-of-app>
```

3.1.2 Determining if a Caller's FacetID is Authorized for an AppID

1. If the AppID is not an HTTPS URL, and matches the FacetID of the caller, no additional processing is necessary and the operation may proceed.
2. If the AppID is null or empty, the client must set the AppID to be the FacetID of the caller, and the operation may proceed without additional processing.
3. If the caller's FacetID is an `https://` Origin sharing the same host as the AppID, (e.g. if an application hosted at `https://fido.example.com/myApp` set an AppID of `https://fido.example.com/myAppId`), no additional processing is necessary and the operation may proceed. This algorithm may be continued asynchronously for purposes of caching the **`TrustedFacetList`**, if desired.
4. Begin to fetch the **`TrustedFacetList`** using the HTTP GET method. The location must be identified with an HTTPS URL.
5. The URL must be dereferenced with an anonymous fetch. That is, the HTTP GET must include no cookies, authentication, Origin or Referer headers, and present no TLS certificates or other forms of credentials.
6. The response must set a MIME Content-Type of "application/fido.trusted-apps+json".
7. The caching related HTTP header fields in the HTTP response (e.g. “Expires”) should be respected when fetching a **`TrustedFacetList`**.
8. The server hosting the **`TrustedFacetList`** must respond uniformly to all clients. That is, it must not vary the contents of the response body based on any credential material, including ambient authority such as originating IP address, supplied with the request.
9. If the server returns an HTTP redirect (status code 3xx) the server must also send the HTTP header `FIDO-AppID-Redirect-Authorized: true` and the client must verify the presence of such a header before following the redirect. This protects against abuse of open redirectors within the target domain by unauthorized parties. If this check has passed, restart this algorithm from step 4.
10. A **`TrustedFacetList`** may contain an unlimited number of entries, but clients may truncate or decline to process large responses.
11. From among the objects in the `trustedFacet` array, select the one with the `version` matching that of the protocol message version. With "matching" we mean: the highest version that appears in the TrustedFacetList that is smaller or equal to the actual protocol version being used.
12. The scheme of URLs in `ids` must identify either an application identity (e.g. using the `apk:`, `ios:` or similar scheme) or an `https:` Web Origin [RFC6454].
13. Entries in `ids` using the `https://` scheme must contain only scheme, host and port components, with an optional trailing /. Any path, query string, username/password, or fragment information must be discarded.
14. All Web Origins listed must have host names under the scope of the same least-specific private label in the DNS, using the following algorithm:
    1. Obtain the list of public DNS suffixes from https://publicsuffix.org/list/effective_tld_names.dat (the client may cache such data), or equivalent functionality as available on the platform.
    2. Extract the host portion of the original AppID URL, before following any redirects.
    3. The least-specific private label is the portion of the host portion of the AppID URL that matches a most-specific public suffix plus one additional label to the left (also known as 'effective top-level domain'+1 or eTLD+1).
    4. For each Web Origin in the **`TrustedFacetList`**, the calculation of the least-specific private label in the DNS must be a case-insensitive match of that of the AppID URL itself. Entries that do not match must be discarded.
15. If the **`TrustedFacetList`** cannot be retrieved and successfully parsed according to these rules, the client must abort processing of the requested FIDO operation.
16. After processing the `trustedFacets` entry of the correct `version` and removing any invalid entries, if the caller's FacetID matches one listed in `ids`, the operation is allowed.

3.1.3 TrustedFacet List and Structure

The Trusted Facets JSON resource is a serialized **`TrustedFacetList`** hosted at the AppID URL. It consists of a dictionary containing a single member, `trustedFacets` which is an array of `TrustedFacets` dictionaries.

WebIDL

```
dictionary TrustedFacetList {
    TrustedFacets[] trustedFacets;
};
```

*3.1.3.1 Dictionary **`TrustedFacetList`** Members*

`trustedFacets` of type array of *TrustedFacets*
: An array of TrustedFacets.

WebIDL

```
dictionary TrustedFacets {
    Version     version;
    DOMString[] ids;
};
```

*3.1.3.2 Dictionary **`TrustedFacets`** Members*

`version` of type Version
: The protocol version to which this set of trusted facets applies. See [UAFProtocol] for the definition of the `version` structure.

`ids` of type array of DOMString
: An array of URLs identifying authorized facets for this AppID.

3.1.4 AppID Example 1

".com" is a public suffix. "https://www.example.com/appID" is provided as an AppID. The body of the resource at this location contains:

EXAMPLE 3

```
{
  "trustedFacets" : [{
    "version": { "major": 1, "minor" : 0 },
    "ids": [
```

```
    "https://register.example.com", // VALID, shares "example.com" label
    "https://fido.example.com",     // VALID, shares "example.com" label
    "http://www.example.com",       // DISCARD, scheme is not https:
    "http://www.example-test.com",  // DISCARD, "example-test.com" does not match
    "https://www.example.com:444"   // VALID, port is not significant
    ]
  }]
}
```

For this policy, "https://www.example.com" and "https://register.example.com" would have access to the keys registered for this AppID, and "https://user1.example.com" would not.

3.1.5 AppID Example 2

"hosting.example.com" is a public suffix, operated under "example.com" and used to provide hosted cloud services for many companies. "https://companyA.hosting.example.com/appID" is provided as an AppID. The body of the resource at this location contains:

EXAMPLE 4

```
{
  "trustedFacets" : [{
    "version": { "major": 1, "minor" : 0 },
    "ids": [
        "https://register.example.com",              // DISCARD, does not share  "companyA.hosting.example.com" label
        "https://fido.companyA.hosting.example.com", // VALID, shares "companyA.hosting.example.com" label
        "https://xyz.companyA.hosting.example.com",  // VALID, shares "companyA.hosting.example.com" label
        "https://companyB.hosting.example.com"       // DISCARD, "companyB.hosting.example.com" does not match
     ]
  }]
}
```

For this policy, "https://fido.companyA.hosting.example.com" would have access to the keys registered for this AppID, and "https://register.example.com" and "https://companyB.hosting.example.com" would not as a public-suffix exists between these DNS names and the AppID's.

3.1.6 Obtaining FacetID of Android Native App

This section is non-normative.

The following code demonstrates how a FIDO Client can obtain and construct the FacetID of a calling Android native application.

EXAMPLE 5: AndroidFacetID SHA256

```
private String getFacetID(Context aContext, int callingUid) {

    String packageNames[] = aContext.getPackageManager().getPackagesForUid(callingUid);

    if (packageNames == null) {
        return null;
    }

    try {
        PackageInfo info = aContext.getPackageManager().getPackageInfo(packageNames[0], PackageManager.GET_SIGNATURES);

        byte[] cert = info.signatures[0].toByteArray();
        InputStream input = new ByteArrayInputStream(cert);

        CertificateFactory cf = CertificateFactory.getInstance("X509");
        X509Certificate c = (X509Certificate) cf.generateCertificate(input);

        MessageDigest md = MessageDigest.getInstance("SHA256");

        return "android:apk-key-hash-sha256:" +
                  Base64.encodeToString(md.digest(c.getEncoded()), Base64.DEFAULT | Base64.NO_WRAP | Base64.NO_PADDING);
    }
    catch (PackageManager.NameNotFoundException e) {
        e.printStackTrace();
    }
    catch (CertificateException e) {
        e.printStackTrace();
    }
    catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    catch (CertificateEncodingException e) {
        e.printStackTrace();
    }

    return null;
}
```

EXAMPLE 6: AndroidFacetID SHA1

```
private String getFacetID(Context aContext, int callingUid) {

    String packageNames[] = aContext.getPackageManager().getPackagesForUid(callingUid);

    if (packageNames == null) {
        return null;
    }

    try {
        PackageInfo info = aContext.getPackageManager().getPackageInfo(packageNames[0], PackageManager.GET_SIGNATURES);

        byte[] cert = info.signatures[0].toByteArray();
        InputStream input = new ByteArrayInputStream(cert);

        CertificateFactory cf = CertificateFactory.getInstance("X509");
        X509Certificate c = (X509Certificate) cf.generateCertificate(input);

        MessageDigest md = MessageDigest.getInstance("SHA1");

        return "android:apk-key-hash:" +
                  Base64.encodeToString(md.digest(c.getEncoded()), Base64.DEFAULT | Base64.NO_WRAP | Base64.NO_PADDING);
    }
```

```
        catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
        }
        catch (CertificateException e) {
            e.printStackTrace();
        }
        catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        catch (CertificateEncodingException e) {
            e.printStackTrace();
        }

        return null;
    }
```

3.1.7 Additional Security Considerations

The UAF protocol supports passing FacetID to the FIDO Server and including the FacetID in the computation of the authentication response.

Trusting a web origin facet implicitly trusts all subdomains under the named entity because web user agents do not provide a security barrier between such origins. So, in AppID Example 1, although not explicitly listed, "https://foobar.register.example.com" would still have effective access to credentials registered for the AppID "https://www.example.com/appID" because it can effectively act as "https://register.example.com".

The component implementing the controls described here must reliably identify callers to securely enforce the mechanisms. Platform inter-process communication mechanisms which allow such identification should be used when available.

It is unlikely that the component implementing the controls described here can verify the integrity and intent of the entries on a `TrustedFacetList`. If a trusted facet can be compromised or enlisted as a *confused deputy* [FIDOGlossary] by a malicious party, it may be possible to trick a user into completing an authentication ceremony under the control of that malicious party.

3.1.7.1 Wildcards in TrustedFacet identifiers

This section is non-normative.

Wildcards are not supported in TrustedFacet identifiers. This follows the advice of RFC6125 [RFC6125], section 7.2.

FacetIDs are URIs that uniquely identify specific security principals that are trusted to interact with a given registered credential. Wildcards introduce undesirable ambiguitiy in the defintion of the principal, as there is no consensus syntax for what wildcards mean, how they are expanded and where they can occur across different applications and protocols in common use. For schemes indicating application identities, it is not clear that wildcarding is appropriate in any fashion. For Web Origins, it broadly increases the scope of the credential to potentially include rogue or buggy hosts.

Taken together, these ambiguities might introduce exploitable differences in identity checking behavior among client implementations and would necessitate overly complex and inefficient identity checking algorithms.

A. References

A.1 Normative references

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html
[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119
[RFC4648]
S. Josefsson. *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*. October 2006. URL: http://www.ietf.org/rfc/rfc4648.txt
[RFC6125]
P. Saint-Andre; J. Hodges. *Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS) (RFC 6125)*. March 2011. URL: http://www.ietf.org/rfc/rfc6125.txt
[RFC6454]
A. Barth. *The Web Origin Concept (RFC 6454)*. June 2011. URL: http://www.ietf.org/rfc/rfc6454.txt
[UAFProtocol]
R. Lindemann; D. Baghdasaryan; E. Tiffany; D. Balfanz; B. Hill; J. Hodges. *FIDO UAF Protocol Specification v1.0*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html

A.2 Informative references

[ANDROID]
The Android™ Operating System. URL: http://developer.android.com/
[APK-Signing]
Signing Your Applications. URL: http://developer.android.com/tools/publishing/app-signing.html
[BundleID]
Configuring your Xcode Project for Distribution. URL: https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/ConfiguringYourApp/ConfiguringYourApp.html
[UAFArchOverview]
S. Machani; R. Philpott; S. Srinivas; J. Kemp; J. Hodges. *FIDO UAF Architectural Overview*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-overview-v1.2-rd-20171128.html
[iOS]
iOS Dev Center. URL: https://developer.apple.com/devcenter/ios/index.action

FIDO Metadata Statements

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-metadata-statement-v2.0-ps-20170927.html
Previous version:
https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-metadata-statement-v2.0-rd-20161004.html
Editors:
Rolf Lindemann, Nok Nok Labs, Inc.
John Kemp, FIDO Alliance
Contributors:
Brad Hill, PayPal, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

FIDO authenticators may have many different form factors, characteristics and capabilities. This document defines a standard means to describe the relevant pieces of information about an authenticator in order to interoperate with it, or to make risk-based policy decisions about transactions involving a particular authenticator.

Status of This Document

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

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

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

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as`code`.

String literals are enclosed in “”, e.g. “UAF-TLV”.

In formulas we use “|” to denote byte wise concatenation operations.

DOM APIs are described using the ECMAScript [ECMA-262] bindings for WebIDL [WebIDL-ED].

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as required.

WebIDL dictionary members must not have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

All diagrams, examples, notes in this specification are non-normative.

> NOTE
>
> Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Conformance

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

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

The FIDO family of protocols enable simpler and more secure online authentication utilizing a wide variety of different devices in a competitive marketplace. Much of the complexity behind this variety is hidden from Relying Party applications, but in order to accomplish the goals of FIDO, Relying Parties must have some means of discovering and verifying various characteristics of authenticators. Relying Parties can learn a subset of verifiable information for authenticators certified by the FIDO Alliance with an Authenticator Metadata statement. The URL to access that Metadata statement is provided by the Metadata TOC file accessible through the Metadata Service [FIDOMetadataService].

For definitions of terms, please refer to the FIDO Glossary [FIDOGlossary].

2.1 Scope

This document describes the format of and information contained in *Authenticator Metadata* statements. For a definitive list of possible values for the various types of information, refer to the FIDO Registry of Predefined Values [FIDORegistry].

The description of the processes and methods by which authenticator metadata statements are distributed and the methods how these statements can be verified are described in the Metadata Service Specification [FIDOMetadataService].

2.2 Audience

The intended audience for this document includes:

- FIDO authenticator vendors who wish to produce metadata statements for their products.
- FIDO server implementers who need to consume metadata statements to verify characteristics of authenticators and attestation statements, make proper algorithm choices for protocol messages, create policy statements or tailor various other modes of operation to authenticator-specific characteristics.
- FIDO relying parties who wish to
    - create custom policy statements about which authenticators they will accept
    - risk score authenticators based on their characteristics
    - verify attested authenticator IDs for cross-referencing with third party metadata

2.3 Architecture

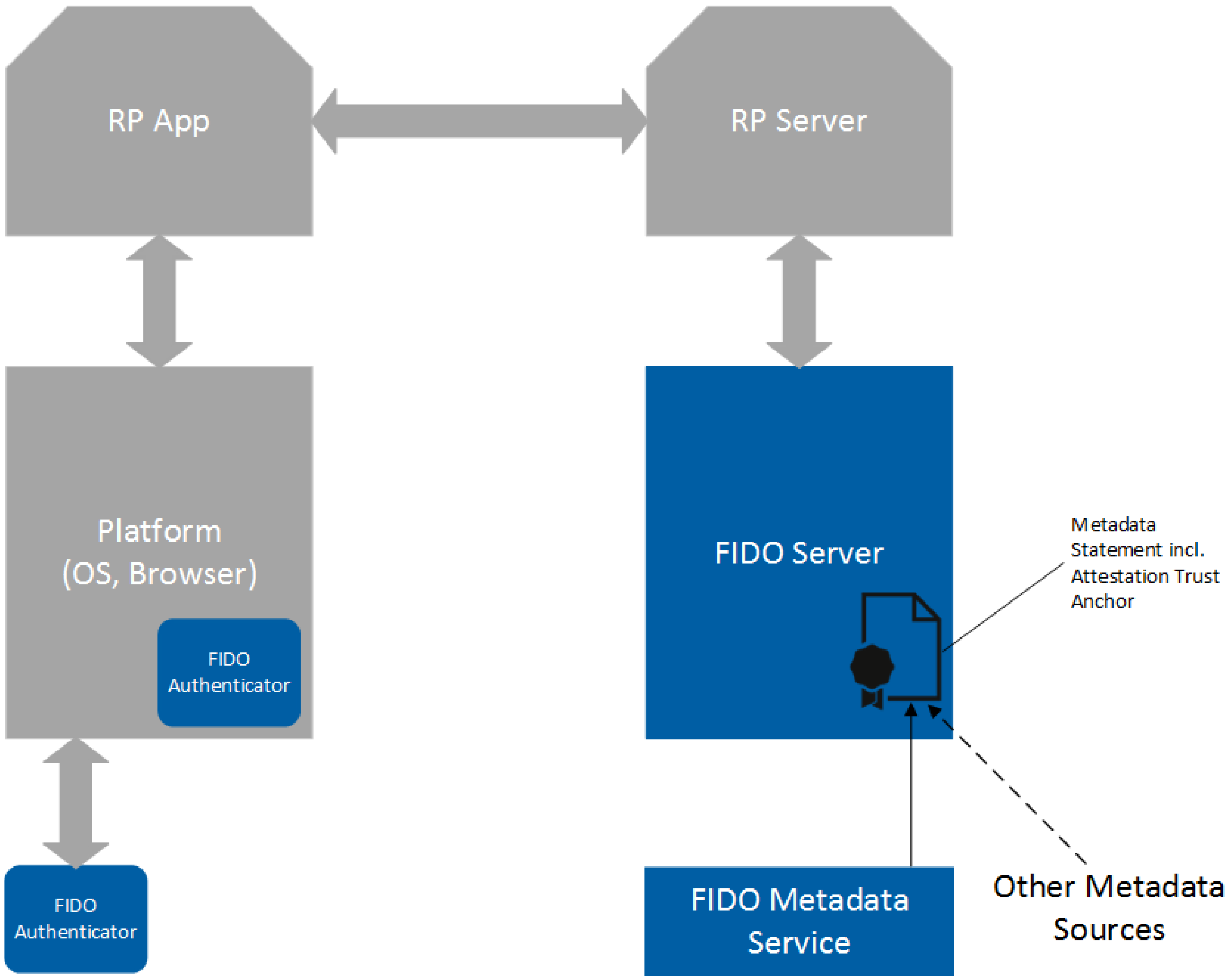

Fig. 1 The FIDO Architecture

Authenticator metadata statements are used directly by the FIDO server at a relying party, but the information contained in the authoritative statement is used in several other places. How a server obtains these metadata statements is described in [FIDOMetadataService].

The workflow around an authenticator metadata statement is as follows:

1. The authenticator vendor produces a metadata statement describing the characteristics of an authenticator.
2. The metadata statement is submitted to the FIDO Alliance as part of the FIDO certification process. The FIDO Alliance distributes the metadata as described in [FIDOMetadataService].
3. A FIDO relying party configures its registration policy to allow authenticators matching certain characteristics to be registered.
4. The FIDO server sends a registration challenge message. This message can contain such policy statement.
5. Depending on the FIDO protocol being used, either the relying party application or the FIDO UAF Client receives the policy statement as part of the challenge message and processes it. It queries available authenticators for their self-reported characteristics and (with the user's input) selects an authenticator that matches the policy, to be registered.
6. The client processes and sends a registration response message to the server. This message contains a reference to the authenticator model and, optionally, a signature made with the private key corresponding to the public key in the authenticator's attestation certificate.
7. The FIDO Server looks up the metadata statement for the particular authenticator model. If the metadata statement lists an attestation certificate(s), it verifies that an attestation signature is present, and made with the private key corresponding to either (a) one of the certificates listed in this metadata statement or (b) corrsponding to the public key in a certificate that *chains* to one of the issuer certificates listed in the authenticator's metadata statement.
8. The FIDO Server next verifies that the authenticator meets the originally supplied registration policy based on its authoritative metadata statement. This prevents the registration of unexpected authenticator models.
9. *Optionally*, a FIDO Server may, with input from the Relying Party, assign a risk or trust score to the authenticator, based on its metadata, including elements not selected for by the stated policy.
10. *Optionally*, a FIDO Server may cross-reference the attested authenticator model with other metadata databases published by third parties. Such third-party metadata might, for example, inform the FIDO Server if an authenticator has achieved certifications relevant to certain markets or industry verticals, or whether it meets application-specific regulatory requirements.

3. Types

This section is normative.

3.1 Authenticator Attestation GUID (AAGUID) typedef

WebIDL

```
typedef DOMString AAGUID;
```

`string[36]`

Some authenticators have an AAGUID, which is a 128-bit identifier that indicates the type (e.g. make and model) of the authenticator. The AAGUID must be chosen by the manufacturer to be identical across all substantially identical authenticators made by that manufacturer, and different (with probability $1-2^{-128}$ or greater) from the AAGUIDs of all other types of authenticators.

The AAGUID is represented as a string (e.g. "7a98c250-6808-11cf-b73b-00aa00b677a7") consisting of 5 hex strings separated by a dash ("-"), see [RFC4122].

3.2 CodeAccuracyDescriptor dictionary

The `CodeAccuracyDescriptor` describes the relevant accuracy/complexity aspects of passcode user verification methods.

> NOTE
>
> One example of such a method is the use of 4 digit PIN codes for mobile phone SIM card unlock.
>
> We are using the numeral system `base` (radix) and `minLen`, instead of the number of potential combinations since there is sufficient evidence [iPhonePasscodes] [MoreTopWorstPasswords] that users don't select their code evenly distributed at random. So software might take into account the various probability distributions for different bases. This essentially means that in practice, passcodes are not as secure as they could be if randomly chosen.

WebIDL

```
dictionary CodeAccuracyDescriptor {
    required unsigned short base;
    required unsigned short minLength;
    unsigned short          maxRetries;
    unsigned short          blockSlowdown;
};
```

3.2.1 Dictionary `CodeAccuracyDescriptor` Members

`base` of type required unsigned short
: The numeric system base (radix) of the code, e.g. 10 in the case of decimal digits.

`minLength` of type required unsigned short
: The minimum number of digits of the given base required for that code, e.g. 4 in the case of 4 digits.

`maxRetries` of type unsigned short
: Maximum number of false attempts before the authenticator will block this method (at least for some time). 0 means it will never block.

`blockSlowdown` of type unsigned short
: Enforced minimum number of seconds wait time after blocking (e.g. due to forced reboot or similar). 0 means this user verification method will be blocked, either permanently or until an alternative user verification method method succeeded. All alternative user verification methods must be specified appropriately in the Metadata in `userVerificationDetails`.

3.3 BiometricAccuracyDescriptor dictionary

The `BiometricAccuracyDescriptor` describes relevant accuracy/complexity aspects in the case of a biometric user verification method.

> NOTE
>
> The *False Acceptance Rate* (FAR) and *False Rejection Rate* (FRR) values typically are interdependent via the *Receiver Operator Characteristic* (ROC) curve.
>
> The *False Artefact Acceptance Rate* (FAAR) value reflects the capability of detecting presentation attacks, such as the detection of rubber finger presentation.
>
> The FAR, FRR, and FAAR values given here must reflect the actual configuration of the authenticators (as opposed to being theoretical best case values).

At least one of the values must be set. If the vendor doesn't want to specify such values, then `VerificationMethodDescriptor.baDesc` must be omitted.

> NOTE
>
> Typical fingerprint sensor characteristics can be found in Google Android 6.0 Compatibility Definition and Apple iOS Security Guide.

WebIDL

```
dictionary BiometricAccuracyDescriptor {
    double          FAR;
    double          FRR;
    double          EER;
    double          FAAR;
    unsigned short  maxReferenceDataSets;
    unsigned short  maxRetries;
    unsigned short  blockSlowdown;
};
```

3.3.1 Dictionary `BiometricAccuracyDescriptor` Members

`FAR` of type double
: The false acceptance rate [ISO19795-1] for a single reference data set, i.e. the percentage of non-matching data sets that are accepted as valid ones. For example a FAR of `0.002%` would be encoded as `0.00002`.

> NOTE
>
> The resulting FAR when all reference data sets are used is `maxReferenceDataSets * FAR`.
>
> The false acceptance rate is relevant for the security. Lower false acceptance rates mean better security.
>
> Only the live captured subjects are covered by this value - not the presentation of artefacts.

`FRR` of type double
: The false rejection rate for a single reference data set, i.e. the percentage of presented valid data sets that lead to a (false) non-acceptance. For example a FRR of `10%` would be encoded as `0.1`.

> NOTE

> The false rejection rate is relevant for the convenience. Lower false acceptance rates mean better convenience.

`EER` of type double
: The equal error rate for a single reference data set.

`FAAR` of type double
: The false artefact acceptance rate [ISO30107-1], i.e. the percentage of artefacts that are incorrectly accepted by the system. For example a FAAR of `0.1%` would be encoded as `0.001`.

> NOTE
>
> The false artefact acceptance rate is relevant for the security of the system. Lower false artefact acceptance rates imply better security.

`maxReferenceDataSets` of type unsigned short
: Maximum number of alternative reference data sets, e.g. 3 if the user is allowed to enroll 3 different fingers to a fingerprint based authenticator.

`maxRetries` of type unsigned short
: Maximum number of false attempts before the authenticator will block this method (at least for some time). 0 means it will never block.

`blockSlowdown` of type unsigned short
: Enforced minimum number of seconds wait time after blocking (e.g. due to forced reboot or similar). 0 means that this user verification method will be blocked either permanently or until an alternative user verification method succeeded. All alternative user verification methods must be specified appropriately in the metadata in `userVerificationDetails`.

3.4 PatternAccuracyDescriptor dictionary

The `PatternAccuracyDescriptor` describes relevant accuracy/complexity aspects in the case that a pattern is used as the user verification method.

> NOTE
>
> One example of such a pattern is the 3x3 dot matrix as used in Android [AndroidUnlockPattern] screen unlock. The `minComplexity` would be 1624 in that case, based on the user choosing a 4-digit PIN, the minimum allowed for this mechanism.

WebIDL

```
dictionary PatternAccuracyDescriptor {
    required unsigned long minComplexity;
    unsigned short         maxRetries;
    unsigned short         blockSlowdown;
};
```

3.4.1 Dictionary `PatternAccuracyDescriptor` Members

`minComplexity` of type required unsigned long
: Number of possible patterns (having the minimum length) out of which exactly one would be the right one, i.e. 1/probability in the case of equal distribution.

`maxRetries` of type unsigned short
: Maximum number of false attempts before the authenticator will block authentication using this method (at least temporarily). 0 means it will never block.

`blockSlowdown` of type unsigned short
: Enforced minimum number of seconds wait time after blocking (due to forced reboot or similar mechanism). 0 means this user verification method will be blocked, either permanently or until an alternative user verification method method succeeded. All alternative user verification methods must be specified appropriately in the metadata under `userVerificationDetails`.

3.5 VerificationMethodDescriptor dictionary

A descriptor for a specific *base user verification method* as implemented by the authenticator.

A base user verification method must be chosen from the list of those described in [FIDORegistry]

> NOTE
>
> In reality, several of the methods described above might be combined. For example, a fingerprint based user

> verification can be combined with an alternative password.

The specification of the related AccuracyDescriptor is optional, but recommended.

WebIDL

```
dictionary VerificationMethodDescriptor {
    required unsigned long      userVerification;
    CodeAccuracyDescriptor      caDesc;
    BiometricAccuracyDescriptor baDesc;
    PatternAccuracyDescriptor   paDesc;
};
```

3.5.1 Dictionary `VerificationMethodDescriptor` Members

`userVerification` of type required unsigned long
: a *single* `USER_VERIFY` constant (see [FIDORegistry]), **not a bit flag combination**. This value must be non-zero.

`caDesc` of type *CodeAccuracyDescriptor*
: May optionally be used in the case of method`USER_VERIFY_PASSCODE`.

`baDesc` of type *BiometricAccuracyDescriptor*
: May optionally be used in the case of method `USER_VERIFY_FINGERPRINT`, `USER_VERIFY_VOICEPRINT`, `USER_VERIFY_FACEPRINT`, `USER_VERIFY_EYEPRINT`, or `USER_VERIFY_HANDPRINT`.

`paDesc` of type *PatternAccuracyDescriptor*
: May optionally be used in case of method`USER_VERIFY_PATTERN`.

3.6 verificationMethodANDCombinations typedef

WebIDL

```
typedef VerificationMethodDescriptor[] VerificationMethodANDCombinations;
```

`VerificationMethodANDCombinations` must be non-empty. It is a list containing the base user verification methods which must be passed as part of a successful user verification.

This list will contain only a single entry if using a single user verification method is sufficient.

If this list contains multiple entries, then all of the listed user verification methods must be passed as part of the user verification process.

3.7 rgbPaletteEntry dictionary

The `rgbPaletteEntry` is an RGB three-sample tuple palette entry

WebIDL

```
dictionary rgbPaletteEntry {
    required unsigned short r;
    required unsigned short g;
    required unsigned short b;
};
```

3.7.1 Dictionary `rgbPaletteEntry` Members

`r` of type required unsigned short
: Red channel sample value

`g` of type required unsigned short
: Green channel sample value

`b` of type required unsigned short
: Blue channel sample value

3.8 DisplayPNGCharacteristicsDescriptor dictionary

The DisplayPNGCharacteristicsDescriptor describes a PNG image characteristics as defined in the PNG [PNG] spec for IHDR (image header) and PLTE (palette table)

WebIDL

```
dictionary DisplayPNGCharacteristicsDescriptor {
    required unsigned long width;
    required unsigned long height;
    required octet         bitDepth;
    required octet         colorType;
```

```
    required octet          compression;
    required octet          filter;
    required octet          interlace;
    rgbPaletteEntry[]       plte;
};
```

3.8.1 Dictionary DisplayPNGCharacteristicsDescriptor Members

width of type required unsigned long
: image width

height of type required unsigned long
: image height

bitDepth of type required octet
: Bit depth - bits per sample or per palette index.

colorType of type required octet
: Color type defines the PNG image type.

compression of type required octet
: Compression method used to compress the image data.

filter of type required octet
: Filter method is the preprocessing method applied to the image data before compression.

interlace of type required octet
: Interlace method is the transmission order of the image data.

plte of type array of *rgbPaletteEntry*
: 1 to 256 palette entries

3.9 EcdaaTrustAnchor dictionary

In the case of ECDAA attestation, the ECDAA-Issuer's trust anchor must be specified in this field.

WebIDL

```
dictionary EcdaaTrustAnchor {
    required DOMString X;
    required DOMString Y;
    required DOMString c;
    required DOMString sx;
    required DOMString sy;
    required DOMString G1Curve;
};
```

3.9.1 Dictionary EcdaaTrustAnchor Members

X of type required DOMString
: base64url encoding of the result of ECPoint2ToB of the ECPoint2 X = P_2^x. See [FIDOEcdaaAlgorithm] for the definition of ECPoint2ToB.

Y of type required DOMString
: base64url encoding of the result of ECPoint2ToB of the ECPoint2 Y = P_2^y. See [FIDOEcdaaAlgorithm] for the definition of ECPoint2ToB.

c of type required DOMString
: base64url encoding of the result of BigNumberToB(c). See section "Issuer Specific ECDAA Parameters" in [FIDOEcdaaAlgorithm] for an explanation of c. See [FIDOEcdaaAlgorithm] for the definition of BigNumberToB.

sx of type required DOMString
: base64url encoding of the result of BigNumberToB(sx). See section "Issuer Specific ECDAA Parameters" in [FIDOEcdaaAlgorithm] for an explanation of sx. See [FIDOEcdaaAlgorithm] for the definition of BigNumberToB.

sy of type required DOMString
: base64url encoding of the result of BigNumberToB(sy). See section "Issuer Specific ECDAA Parameters" in [FIDOEcdaaAlgorithm] for an explanation of sy. See [FIDOEcdaaAlgorithm] for the definition of BigNumberToB.

G1Curve of type required DOMString
: Name of the Barreto-Naehrig elliptic curve for G1. "BN_P256", "BN_P638", "BN_ISOP256", and "BN_ISOP512" are supported. See section "Supported Curves for ECDAA" in [FIDOEcdaaAlgorithm] for details.

> NOTE
>
> Whenever a party uses this trust anchor for the first time, it must first verify that it was correctly generated by verifying s, sx, sy. See [FIDOEcdaaAlgorithm] for details.

3.10 ExtensionDescriptor dictionary

This descriptor contains an extension supported by the authenticator.

WebIDL

```
dictionary ExtensionDescriptor {
    required DOMString id;
    unsigned short     tag;
    DOMString          data;
    required boolean   fail_if_unknown;
};
```

3.10.1 Dictionary `ExtensionDescriptor` Members

`id` of type required DOMString

Identifies the extension.

`tag` of type unsigned short

The TAG of the extension if this was assigned. TAGs are assigned to extensions if they could appear in an assertion.

`data` of type DOMString

Contains arbitrary data further describing the extension and/or data needed to correctly process the extension.

This field may be missing or it may be empty.

`fail_if_unknown` of type required boolean

Indicates whether unknown extensions must be ignored (`false`) or must lead to an error (`true`) when the extension is to be processed by the FIDO Server, FIDO Client, ASM, or FIDO Authenticator.

- A value of `false` indicates that unknown extensions must be ignored
- A value of `true` indicates that unknown extensions must result in an error.

4. Metadata Keys

This section is normative.

WebIDL

```
dictionary MetadataStatement {
    DOMString                                    legalHeader;
    AAID                                         aaid;
    AAGUID                                       aaguid;
    DOMString[]                                  attestationCertificateKeyIdentifiers;
    required DOMString                           description;
    required unsigned short                      authenticatorVersion;
    DOMString                                    protocolFamily;
    required Version[]                           upv;
    required DOMString                           assertionScheme;
    required unsigned short                      authenticationAlgorithm;
    unsigned short[]                             authenticationAlgorithms;
    required unsigned short                      publicKeyAlgAndEncoding;
    unsigned short[]                             publicKeyAlgAndEncodings;
    required unsigned short[]                    attestationTypes;
    required VerificationMethodANDCombinations[] userVerificationDetails;
    required unsigned short                      keyProtection;
    boolean                                      isKeyRestricted;
    boolean                                      isFreshUserVerificationRequired;
    required unsigned short                      matcherProtection;
    unsigned short                               cryptoStrength;
    DOMString                                    operatingEnv;
    required unsigned long                       attachmentHint;
    required boolean                             isSecondFactorOnly;
    required unsigned short                      tcDisplay;
    DOMString                                    tcDisplayContentType;
    DisplayPNGCharacteristicsDescriptor[]        tcDisplayPNGCharacteristics;
    required DOMString[]                         attestationRootCertificates;
    EcdaaTrustAnchor[]                           ecdaaTrustAnchors;
    DOMString                                    icon;
    ExtensionDescriptor                          supportedExtensions[];
};
```

4.1 Dictionary `MetadataStatement` Members

`legalHeader` of type DOMString

The legalHeader, if present, contains a legal guide for accessing and using metadata, which itself may contain URL(s) pointing to further information, such as a full Terms and Conditions statement.

`aaid` of type AAID
: The Authenticator Attestation ID. See [UAFProtocol] for the definition of the AAID structure. This field must be set if the authenticator implements FIDO UAF.

> NOTE
>
> FIDO UAF Authenticators support AAID, but they don't support AAGUID.

`aaguid` of type *AAGUID*
: The Authenticator Attestation GUID. See [FIDOKeyAttestation] for the definition of the AAGUID structure. This field must be set if the authenticator implements FIDO 2.

> NOTE
>
> FIDO 2 Authenticators support AAGUID, but they don't support AAID.

`attestationCertificateKeyIdentifiers` of type array of DOMString
: A list of the attestation certificate public key identifiers encoded as hex string. This value must be calculated according to method 1 for computing the keyIdentifier as defined in [RFC5280] section 4.2.1.2. The hex string must not contain any non-hex characters (e.g. spaces). All hex letters must be lower case. This field must be set if neither `aaid` nor `aaguid` are set. Setting this field implies that the attestation certificate(s) are dedicated to a single authenticator model.

All attestationCertificateKeyIdentifier values should be unique within the scope of the Metadata Service.

> NOTE
>
> FIDO U2F Authenticators typically do not support AAID nor AAGUID, but they use attestation certificates dedicated to a single authenticator model.

`description` of type required DOMString
: A human-readable short description of the authenticator.

> NOTE
>
> This description should help an administrator configuring authenticator policies. This description might deviate from the description returned by the ASM for that authenticator.
>
> This description should contain the public authenticator trade name and the publicly known vendor name.

`authenticatorVersion` of type required unsigned short
: Earliest (i.e. lowest) trustworthy `authenticatorVersion` meeting the requirements specified in this metadata statement.

Adding new `StatusReport` entries with status `UPDATE_AVAILABLE` to the metadata `TOC` object [FIDOMetadataService] must also change this `authenticatorVersion` if the update fixes severe security issues, e.g. the ones reported by preceding `StatusReport` entries with status code `USER_VERIFICATION_BYPASS`, `ATTESTATION_KEY_COMPROMISE`, `USER_KEY_REMOTE_COMPROMISE`, `USER_KEY_PHYSICAL_COMPROMISE`, `REVOKED`.

It is recommended to assume increased risk if this version is higher (newer) than the firmware version present in an authenticator. For example, if a `StatusReport` entry with status `USER_VERIFICATION_BYPASS` or `USER_KEY_REMOTE_COMPROMISE` precedes the `UPDATE_AVAILABLE` entry, than any firmware version lower (older) than the one specified in the metadata statement is assumed to be vulnerable.

`protocolFamily` of type DOMString
: The FIDO protocol family. The values "uaf", "u2f", and "fido2" are supported. If this field is missing, the assumed protocol family is "uaf". Metadata Statements for U2F authenticators must set the value of protocolFamily to "u2f" and FIDO 2.0 Authenticators implementations must set the value of protocolFamily to "fido2".

`upv` of type array of required Version
: The FIDO unified protocol version(s) (related to the specific protocol family) supported by this authenticator. See [UAFProtocol] for the definition of the `Version` structure.

`assertionScheme` of type required DOMString
: The assertion scheme supported by the authenticator. Must be set to one of the enumerated strings defined in the FIDO UAF Registry of Predefined Values [UAFRegistry] or to "FIDOV2" in the case of the FIDO 2 assertion scheme.

`authenticationAlgorithm` of type required unsigned short
: The preferred authentication algorithm supported by the authenticator. Must be set to one of the `ALG_` constants defined in the FIDO Registry of Predefined Values [FIDORegistry]. This value must be non-zero.

`authenticationAlgorithms` of type array of unsigned short

The list of authentication algorithms supported by the authenticator. Must be set to the *complete list* of the supported `ALG_` constants defined in the FIDO Registry of Predefined Values [FIDORegistry] if the authenticator supports multiple algorithms. Each value must be non-zero.

> NOTE
>
> FIDO UAF Authenticators
> For verification purposes, the field `SignatureAlgAndEncoding` in the FIDO UAF authentication assertion [UAFAuthnrCommands] should be used to determine the actual signature algorithm and encoding.
>
> FIDO U2F Authenticators
> FIDO U2F only supports one signature algorithm and encoding: `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW` [FIDORegistry].

`publicKeyAlgAndEncoding` of type required unsigned short
The preferred public key format used by the authenticator during registration operations. Must be set to one of the `ALG_KEY` constants defined in the FIDO Registry of Predefined Values [FIDORegistry]. Because this information is not present in APIs related to authenticator discovery or policy, a FIDO server must be prepared to accept and process any and all key representations defined for any public key algorithm it supports. This value must be non-zero.

`publicKeyAlgAndEncodings` of type array of unsigned short
The list of public key formats supported by the authenticator during registration operations. Must be set to the *complete list* of the supported `ALG_KEY` constants defined in the FIDO Registry of Predefined Values [FIDORegistry] if the authenticator model supports multiple encodings. Because this information is not present in APIs related to authenticator discovery or policy, a FIDO server must be prepared to accept and process any and all key representations defined for any public key algorithm it supports. Each value must be non-zero.

> NOTE
>
> FIDO UAF Authenticators
> For verification purposes, the field `PublicKeyAlgAndEncoding` in the FIDO UAF registration assertion [UAFAuthnrCommands] should be used to determine the actual encoding of the public key.
>
> FIDO U2F Authenticators
> FIDO U2F only supports one public key encoding: `ALG_KEY_ECC_X962_RAW` [FIDORegistry].

`attestationTypes` of type array of required unsigned short
The supported attestation type(s). (e.g. `TAG_ATTESTATION_BASIC_FULL(0x3E07)`, `TAG_ATTESTATION_BASIC_SURROGATE(0x3E08)`).
See section 4.1 of FIDO UAF Registry [UAFRegistry], section 5.2.1 of FIDO UAF Authenticator Commands specification [UAFAuthnrCommands], and section 4.1.2 of FIDO UAF Protocol specification [[UAFProtocol] for details.

> NOTE
>
> Even though these tags are defined in FIDO UAF protocol specifications, the attestation types apply to authenticators of all protocol families (e.g. UAF, U2F, ...).

`userVerificationDetails` of type array of required VerificationMethodANDCombinations
A list of *alternative* VerificationMethodANDCombinations. Each of these entries is one alternative user verification method. Each of these alternative user verification methods might itself be an "AND" combination of multiple modalities.

All effectively available alternative user verification methods must be properly specified here. A user verification method is considered effectively available if this method can be used to either:

- enroll new verification reference data to one of the user verification methods

  or

- unlock the UAuth key directly after successful user verification

`keyProtection` of type required unsigned short
A 16-bit number representing the bit fields defined by the `KEY_PROTECTION` constants in the FIDO Registry of Predefined Values [FIDORegistry].

This value must be non-zero.

> NOTE
>
> The keyProtection specified here denotes the effective security of the attestation key and Uauth private key and the effective trustworthiness of the attested attributes in the “sign assertion”. Effective security

> means that key extraction or injecting malicious attested attributes is only possible if the specified protection method is compromised. For example, if keyProtection=TEE is stated, it shall be impossible to extract the attestation key or the Uauth private key or to inject any malicious attested attributes *without breaking the TEE*.

`isKeyRestricted` of type boolean

This entry is set to `true`, if the Uauth private key is restricted by the *authenticator* to only sign valid FIDO signature assertions.

This entry is set to `false`, if the authenticator doesn't restrict the Uauth key to only sign valid FIDO signature assertions. In this case, the calling application could potentially get any hash value signed by the authenticator.

If this field is missing, the assumed value is isKeyRestricted=`true`

.

> NOTE
>
> Note that only in the case of isKeyRestricted=`true`, the FIDO server can trust a signature counter or transaction text to have been correctly processed/controlled by the authenticator.

`isFreshUserVerificationRequired` of type boolean

This entry is set to `true`, if Uauth key usage *always* requires a fresh user verification.

If this field is missing, the assumed value is isFreshUserVerificationRequired=`true`.

This entry is set to `false`, if the Uauth key can be used without requiring a fresh user verification, e.g. without any additional user interaction, if the user was verified a (potentially configurable) caching time ago.

In the case of isFreshUserVerificationRequired=`false`, the FIDO server must verify the registration response and/or authentication response and verify that the (maximum) caching time (sometimes also called "authTimeout") is acceptable.

This entry solely refers to the user verification. In the case of transaction confirmation, the authenticator must always ask the user to authorize the specific transaction.

> NOTE
>
> Note that in the case of isFreshUserVerificationRequired=`false`, the calling App could trigger use of the key without user involvement. In this case it is the responsibility of the App to ask for user consent.

`matcherProtection` of type required unsigned short

A 16-bit number representing the bit fields defined by the `MATCHER_PROTECTION` constants in the FIDO Registry of Predefined Values [FIDORegistry].

This value must be non-zero.

> NOTE
>
> If multiple matchers are implemented, then this value must reflect the *weakest* implementation of all matchers.
>
> The matcherProtection specified here denotes the effective security of the FIDO authenticator's user verification. This means that a false positive user verification implies breach of the stated method. For example, if matcherProtection=TEE is stated, it shall be impossible to trigger use of the Uauth private key when bypassing the user verification *without breaking the TEE*.

`cryptoStrength` of type unsigned short

The authenticator's **overall claimed cryptographic strength** in bits (sometimes also called security strength or security level). This is the minimum of the cryptographic strength of all involved cryptographic methods (e.g. RNG, underlying hash, key wrapping algorithm, signing algorithm, attestation algorithm), e.g. see [FIPS180-4], [FIPS186-4], [FIPS198-1], [SP800-38B], [SP800-38C], [SP800-38D], [SP800-38F], [SP800-90C], [SP800-90ar1], [FIPS140-2] etc.

If this value is absent, the cryptographic strength is unknown. If the cryptographic strength of one of the involved cryptographic methods is unknown the overall claimed cryptographic strength is also unknown.

`operatingEnv` of type DOMString

Description of the particular operating environment that is used for the Authenticator. These are specified in [FIDORestrictedOperatingEnv].

`attachmentHint` of type required unsigned long

A 32-bit number representing the bit fields defined by the `ATTACHMENT_HINT` constants in the FIDO Registry of Predefined Values [FIDORegistry].

> NOTE
>
> The connection state and topology of an authenticator may be transient and cannot be relied on as authoritative by a relying party, but the metadata field should have all the bit flags set for the topologies possible for the authenticator. For example, an authenticator instantiated as a single-purpose hardware token that can communicate over bluetooth should set `ATTACHMENT_HINT_EXTERNAL` but not `ATTACHMENT_HINT_INTERNAL`.

`isSecondFactorOnly` of type required boolean
Indicates if the authenticator is designed to be used only as a second factor, i.e. requiring some other authentication method as a first factor (e.g. username+password).

`tcDisplay` of type required unsigned short
A 16-bit number representing a combination of the bit flags defined by the `TRANSACTION_CONFIRMATION_DISPLAY` constants in the FIDO Registry of Predefined Values [FIDORegistry].

This value must be 0, if transaction confirmation is not supported by the authenticator.

> NOTE
>
> The tcDisplay specified here denotes the effective security of the authenticator's transaction confirmation display. This means that only a breach of the stated method allows an attacker to inject transaction text to be included in the signature assertion which hasn't been displayed and confirmed by the user.

`tcDisplayContentType` of type DOMString
Supported MIME content type [RFC2049] for the transaction confirmation display, such as `text/plain` or `image/png`.

This value must be present if transaction confirmation is supported, i.e. `tcDisplay` is non-zero.

`tcDisplayPNGCharacteristics` of type array of *DisplayPNGCharacteristicsDescriptor*
A list of *alternative* DisplayPNGCharacteristicsDescriptor. Each of these entries is one alternative of supported image characteristics for displaying a PNG image.

This list must be present if PNG-image based transaction confirmation is supported, i.e. `tcDisplay` is non-zero and `tcDisplayContentType` is `image/png`.

`attestationRootCertificates` of type array of required DOMString
Each element of this array represents a PKIX [RFC5280] X.509 certificate that is a valid trust anchor for this authenticator model. Multiple certificates might be used for different batches of the same model. The array does not represent a certificate chain, but only the trust anchor of that chain. A trust anchor can be a root certificate, an intermediate CA certificate or even the attestation certificate itself.

Each array element is a base64-encoded (section 4 of [RFC4648]), DER-encoded [ITU-X690-2008] PKIX certificate value. Each element must be dedicated for authenticator attestation.

> NOTE
>
> A certificate listed here is a trust anchor. It might be the actual certificate presented by the authenticator, or it might be an issuing authority certificate from the vendor that the actual certificate in the authenticator chains to.
>
> In the case of "uaf" protocol family, the attestation certificate itself and the ordered certificate chain are included in the registration assertion (see [UAFAuthnrCommands]).

Either

1. the manufacturer attestation trust anchor

   or

2. the trust anchor dedicated to a specific authenticator model

must be specified.

In the case (1), the trust anchor certificate might cover multiple authenticator models. In this case, it must be possible to uniquely derive the authenticator model from the Attestation Certificate. When using AAID or AAGUID, this can be achieved by either specifying the AAID or AAGUID in the attestation certificate using the extension id-fido-gen-ce-aaid { 1 3 6 1 4 1 45724 1 1 1 } or id-fido-gen-ce-aaguid { 1 3 6 1 4 1 45724 1 1 4 } or - when neither AAID nor AAGUID are defined - by using the `attestationCertificateKeyIdentifier` method.

In the case (2) this is not required as the trust anchor only covers a single authenticator model.

When supporting surrogate basic attestation only (see [UAFProtocol], section "Surrogate Basic Attestation"), no attestation trust anchor is required/used. So this array must be empty in that case.

ecdaaTrustAnchors of type array of *EcdaaTrustAnchor*
A list of trust anchors used for ECDAA attestation. This entry must be present if and only if attestationType includes TAG_ATTESTATION_ECDAA. The entries in `attestationRootCertificates` have no relevance for ECDAA attestation. Each ecdaaTrustAnchor must be dedicated to a single authenticator model (e.g as identified by its AAID/AAGUID).

icon of type DOMString
A `data:` url [RFC2397] encoded PNG [PNG] icon for the Authenticator.

supportedExtensions[] of type *ExtensionDescriptor*
List of extensions supported by the authenticator.

5. Metadata Statement Format

This section is non-normative.

NORMATIVE

A FIDO Authenticator Metadata Statement is a document containing a JSON encoded dictionary MetadataStatement.

5.1 UAF Example

Example of the metadata statement for an UAF authenticator with:

- authenticatorVersion 2.
- Fingerprint based user verification allowing up to 5 registered fingers, with false acceptance rate of 0.002% and rate limiting attempts for 30 seconds after 5 false trials.
- Authenticator is embedded with the FIDO User device.
- The authentication keys are protected by TEE and are restricted to sign valid FIDO sign assertions only.
- The (fingerprint) matcher is implemented in TEE.
- The Transaction Confirmation Display is implemented in a TEE.
- The Transaction Confirmation Display supports display of "image/png" objects only.
- Display has a width of 320 and a height of 480 pixel. A bit depth of 16 bits per pixel offering True Color (=Color Type 2). The zlib compression method (0). It doesn't support filtering (i.e. filter type of=0) and no interlacing support (interlace method=0).
- The Authentiator can act as first factor or as second factor, i.e. isSecondFactorOnly = false.
- It supports the "UAFV1TLV" assertion scheme.
- It uses the `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW` authentication algorithm.
- It uses the `ALG_KEY_ECC_X962_RAW` public key format (0x100=256 decimal).
- It only implements the `TAG_ATTESTATION_BASIC_FULL` method (0x3E07=15879 decimal).
- It implements UAF protocol version (upv) 1.0 and 1.1.

EXAMPLE 1: MetadataStatement for UAF Authenticator


```
{
  "aaid": "1234#5678",
  "description": "FIDO Alliance Sample UAF Authenticator",
  "authenticatorVersion": 2,
  "upv": [
    { "major": 1, "minor": 0 },
    { "major": 1, "minor": 1 }
  ],
  "assertionScheme": "UAFV1TLV",
  "authenticationAlgorithm": 1,
  "publicKeyAlgAndEncoding": 256,
  "attestationTypes": [15879],
  "userVerificationDetails": [
    [{
      "userVerification": 2,
      "baDesc": {
        "FAR": 0.00002,
        "maxRetries": 5,
        "blockSlowdown": 30,
        "maxReferenceDataSets": 5
      }
    }]
  ],
  "keyProtection": 6,
  "isKeyRestricted": true,
  "matcherProtection": 2,
  "cryptoStrength": 128,
  "operatingEnv": "TEEs based on ARM TrustZone HW",
  "attachmentHint": 1,
  "isSecondFactorOnly": "false",
  "tcDisplay": 5,
  "tcDisplayContentType": "image/png",
  "tcDisplayPNGCharacteristics": [{
    "width": 320,
    "height": 480,
```

```
      "bitDepth": 16,
      "colorType": 2,
      "compression": 0,
      "filter": 0,
      "interlace": 0
    }],
    "attestationRootCertificates": [
      "MIICPTCCAeOgAwIBAgIJAOuexvU3Oy2wMAoGCCqGSM49BAMCMHsxIDAeBgNVBAMM
      F1NhbXBsZSBBdHRlc3RhdGlvbiBSb290MRYwFAYDVQQKDA1GSURPIEFsbGlhbmNl
      MREwDwYDVQQLDAhVQUYgVFdHLDESMBAGA1UEBwwJUGFsbyBBbHRvMQswCQYDVQQI
      DAJDQTELMAkGA1UEBhMCVVMwHhcNMTQwNjE4MTMzMzMyWhcNNDExMTAzMTMzMzMy
      WjB7MSAwHgYDVQQDDBdTYW1wbGUgQXR0ZXN0YXRpb24gUm9vdDEWMBQGA1UECgwN
      RklETyBBbGxpYW5jZTERMA8GA1UECwwIVUFGIFRXRywxEjAQBgNVBAcMCVBhbG8g
      QWx0bzELMAkGA1UECAwCQ0ExCzAJBgNVBAYTAlVTMFkwEwYHKoZIzj0CAQYIKoZI
      zj0DAQcDQgAEH8hv2D0HXa59/BmpQ7RZehL/FMGzFd1QBg9vAUpOZ3ajnuQ94PR7
      aMzH33nUSBr8fHYDrqOBb58pxGqHJRyX/6NQME4wHQYDVR0OBBYEFPoHA3CLhxFb
      C0It7zE4w8hk5EJ/MB8GA1UdIwQYMBaAFPoHA3CLhxFbC0It7zE4w8hk5EJ/MAwG
      A1UdEwQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIhAJ06QSXt9ihIbEKYKIjsPkri
      VdLIgtfsbDSu7ErJfzr4AiBqoYCZf0+zI55aQeAHjIzA9Xm63rruAxBZ9ps9z2XN
      lQ=="
    ],
    "icon": "data:image/png;base64,
      iVBORw0KGgoAAAANSUhEUgAAAE8AAAAvCAYAAACiwJfcAAAAAXNSR0IArs4c6QAAAARnQU1BAACx
      jwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAAAahSURBVGhD7Zr5bxRlGMf9KzTB8AM/YEhE2W7p
      QZcWKKBclSpHATlELARE7kNECCA3FkWK0CKKSCFIsKBcgVCDWGNESdAYidwgggJBiRiMhFc/4wy8
      884zu9NdlnGTfZJP2n3nO++88933fveBBx+PqCzJkTUvBbLmpUDWvBTImpcCSZvXLCdX9R05Sk19
      bb5atf599fG+/erA541q47aP1LLVa9SIyVNUi8Ii8d5kGTsi30NFv7ai9n7QZPMwbdys2erU2XMq
      Udy8+ZcaNmGimE8yXN3RUd3a18nF0fUlovZ+0CTzWpd2Vj+eOm1bEyy6Dx4i5pUMGWveo506q227
      dtuWBIuffr6oWpV0FPNLhow1751Nm21LvPH3rVtWjfz66Lfql8tX7FRl9YFSXsmSseb9ceOGbYk7
      MNUcGPg8ZsbMe9rfQUaaV/JMX9sqdzDCSvp0kZHmTZg9x7bLHcMnThb16eJ+mVfQq8yaUZQNG64i
      XZ+0/kq6uOZFO0QtatdWKfXnRQ99Bj91R5OIFnk54jN0mkUiqlO3XDW+Ml+98mKB6tW7rWpZcPc+
      0zg4tLrYlUc86E6eGDjIMubVpcusearfgIYGRk6brhZVr/JcHzooL7550jedLExopWcApi2ZUqhu
      7JLvrVsQU81zkzOPeemMRYvVuQsX7PbiDQY5JvZonftK+1VY8H9utx530h0ob+jmRYqj6ouaYvEe
      nW/WlYjp8cwbMm682tPwqW1R4tj/2SH13IRJYl4moZvXpiSqDr7dXtQHxa/PK3/+BWsK1dTgHu6V
      8tQJ3bwFkwpFrUOQ50s1r3levm8zZcq17+BBaw7K8lEK5qzkYeark9A8p7P3GzDK+nd3DQow+6UC
      8SVN82iuv38im7NtaXtV1CVq6Rgw4pksmbdi3bu2De7YfaBBxcqfvqPrUjFQNTQ22lfdUVVT68rT
      JKF5DnSmUjgdqg4mSS9pmsfDJR3G6ToH0iW9aV7LWLHYXKllTDt0LTAtkYIaamp1QjVv++uyGUxV
      dJ0DNVXSm+b1qRxpl84ddfX1Lp1O/d69tsod0vs5hGre9xu8o+fpLR1cGhNTD6Z57C9KMWXefJdO
      Z94bb9oqd1ROnS7qITTzHimMqivbO3g0DdVyk3WQBhBztK35YKNdOnc8O3acS6fDZFgKaXLsEJp5
      rdrliBqp89cJcs/m7Tvs0rkjGfN4b0kPoZn3UJuIOrnZ22yP1fmvUx+O5gSqebV1m+zSuYNVhq7T
      WbDiLVvljplLlop6CLXP+2qtvGLIL/1vimISdMBgzSoFZyu6Tqd+jzxgsPaV9BCqee/NjYk6v6lK
      9cwiUc/STtf1HDpM3b592y7h3Thx5ozK69HLpYWuAwaqS5cv26q7ceb8efVYaReP3iFU8zj1knSw
      ZXHMmnCjY0Ogalo7UQfSCM3qQQr2H/XFP7ssXx45Yl91ByeCep4moZoH+1fG3xD4tT7x8kwyj8nw
      b9ev26V0B6d+7H4zKvudAH537FjqyzOHdJnHEuzmXq/WjxObvNMbv7nhywsX2aVsWtC8+48aLeap
      E7p5wKZi0A2AQRV5nvR4E+uJc+b61kApqInxBgmd/4V5QP/mt18HDC7sRHftmeu5lmhV0rn/ALX2
      32bqd4BFnDx7Vi1cWS2uff0IbB47qexxmUj9QutYjupd3tYD6abWBBMrh+apNbOKrNF1+ugCa4ri
      XGfwMPPtViavhU3YMOAAnuUb/R07L0yOSeOadE88ApsXFGff30ynhlJgM51CU6vN9EzgnpvHBFUy
      iVraePiwJ53DF5ZTZnomENg85kNUd2oJi2Wpr4OmmkfN4x4zHfiVFc8Dv8NzuhNqOidilGvA6DGu
      eZwO78AAQn6ciEk6+rw5VcvjvqNDYPOoIUwaKShrxAuXLlkH4aYuGfMYDc10WF5Ta31hPJOfcUhr
      U/JlINi6c6elRYdBpo6++Yfjx61lGNfRm4MD5rJ1j3FoGHnjDSBNarYUgMLyMszKpb7tXpoHfPs8
      h3Wp1LzNfNk54XxC1wDGUmYzXYefh6z/cKtVm4EBxa9VQGDzYr3LrUMRjHEKkk7zaFKYQA2hGQU1
      z+85NFWpXDrkz3vx10GqxQ6BzeNboBk5n8k4nebRh+k1hWfxTF0D1EyWUs5nv+dgQqKaxzuCdE0i
      sHl02NQ8ah0mXr12La3m0f9wik9+wLNTMY/86MPo8yi31OfxmT6PWoqG9+DZukYna56mSZt5WWSy
      5qVA1rwUyJqXAlnzkiai/gHSD7RkTyihogAAAABJRU5ErkJggg=="
  }
```


Example of an *User Verification Methods* entry for an authenticator with:

- Fingerprint based user verification method, with:
  - the ability for the user to enroll up to 5 fingers (reference data sets) with
    - a false acceptance rate of 1 in 50000 (0.002%) per finger. This results in a FAR of 0.01% (0.0001).
    - The fingerprint verification will be blocked after 5 unsuccessful attempts.
- A PIN code with a minimum length of 4 decimal digits has to be set-up as alternative verification method. Entering the PIN will be required to re-activate fingerprint based user verification after it has been blocked.

EXAMPLE 2: User Verification Methods Entry


```
[
  [ { "userVerification": 2, "baDesc": { "FAR": 0.00002, "maxReferenceDataSets": 5,
                             "maxRetries": 5, "blockSlowdown": 0} }],
  [ { "userVerification": 4, "caDesc": { "base": 10, "minLength": 4 } } ]
]
```


5.2 U2F Example

Example of the metadata statement for an U2F authenticator with:

- authenticatorVersion 2.
- Touch based user presence check.
- Authenticator is a USB pluggable hardware token.
- The authentication keys are protected by a secure element.
- The user presence check is implemented in the chip.
- The Authenticator is a pure second factor authenticator.
- It supports the "U2FV1BIN" assertion scheme.
- It uses the `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW` authentication algorithm.
- It uses the `ALG_KEY_ECC_X962_RAW` public key format (0x100=256 decimal).

- It only implements the `TAG_ATTESTATION_BASIC_FULL` method (0x3E07=15879 decimal).
- It implements U2F protocol version 1.0 only.

EXAMPLE 3: MetadataStatement for U2F Authenticator


```
{
  "description": "FIDO Alliance Sample U2F Authenticator",
  "attestationCertificateKeyIdentifiers": ["7c0903708b87115b0b422def3138c3c864e44573"],
  "protocolFamily": "u2f",
  "authenticatorVersion": 2,
  "upv": [
    { "major": 1, "minor": 0 }
  ],
  "assertionScheme": "U2FV1BIN",
  "authenticationAlgorithm": 1,
  "publicKeyAlgAndEncoding": 256,
  "attestationTypes": [15879],
  "userVerificationDetails": [
    [{ "userVerification": 1 }]
  ],
  "keyProtection": 10,
  "matcherProtection": 4,
  "cryptoStrength": 128,
  "operatingEnv": "Secure Element (SE)",
  "attachmentHint": 2,
  "isSecondFactorOnly": "true",
  "tcDisplay": 0,
  "attestationRootCertificates": [
    "MIICPTCCAeOgAwIBAgIJAOuexvU3Oy2wMAoGCCqGSM49BAMCMHsxIDAeBgNVBAMM
    F1NhbXBsZSBBdHRlc3RhdGlvbiBSb290MRYwFAYDVQQKDA1GSURPIEFsbGlhbmNl
    MREwDwYDVQQLDAhVQUYgVFdHLDESMBAGA1UEBwwJUGFsbyBBbHRvMQswCQYDVQQI
    DAJDQTELMAkGA1UEBhMCVVMwHhcNMTQwNjE4MTMzMzMyWhcNNDExMTAzMTMzMzMy
    WjB7MSAwHgYDVQQDDBdTYW1wbGUgQXR0ZXN0YXRpb24gUm9vdDEWMBQGA1UECgwN
    RklETyBBbGxpYW5jZTERMA8GA1UECwwIVUFGIFRXRywxEjAQBgNVBAcMCVBhbG8g
    QWx0bzELMAkGA1UECAwCQ0ExCzAJBgNVBAYTAlVTMFkwEwYHKoZIzj0CAQYIKoZI
    zj0DAQcDQgAEH8hv2D0HXa59/BmpQ7RZehL/FMGzFd1QBg9vAUpOZ3ajnuQ94PR7
    aMzH33nUSBr8fHYDrqOBb58pxGqHJRyX/6NQME4wHQYDVR0OBBYEFPoHA3CLhxFb
    C0It7zE4w8hk5EJ/MB8GA1UdIwQYMBaAFPoHA3CLhxFbC0It7zE4w8hk5EJ/MAwG
    A1UdEwQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIhAJ06QSXt9ihIbEKYKIjsPkri
    VdLIgtfsbDSu7ErJfzr4AiBqoYCZf0+zI55aQeAHjIzA9Xm63rruAxBZ9ps9z2XN
    lQ=="
  ],
  "icon": "data:image/png;base64,
    iVBORw0KGgoAAAANSUhEUgAAAE8AAAAvCAYAAACiwJfcAAAAAXNSR0IArs4c6QAAAARnQU1BAACx
    jwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAAAahSURBVGhD7Zr5bxRlGMf9KzTB8AM/YEhE2W7p
    QZcWKKBclSpHATlELARE7kNECCA3FkWK0CKKSCFIsKBcgVCDWGNESdAYidwgggJBiRiMhFc/4wy8
    884zu9NdlnGTfZJP2n3nO++88933fveBBx+PqCzJkTUvBbLmpUDWvBTImpcCSZvXLCdX9R05Sk19
    bb5atf599fG+/erA541q47aP1LLVa9SIyVNUi8Ii8d5kGTsi30NFv7ai9n7QZPMwbdys2erU2XMq
    Udy8+ZcaNmGimE8yXN3RUd3a18nF0fUlovZ+0CTzWpd2Vj+eOm1bEyy6Dx4i5pUMGWveo506q227
    dtuWBIuffr6oWpV0FPNLhow1751Nm21LvPH3rVtWjfz66Lfql8tX7FRl9YFSXsmSseb9ceOGbYk7
    MNUcGPg8ZsbMe9rfQUaaV/JMX9sqdzDCSvp0kZHmTZg9x7bLHcMnThb16eJ+mVfQq8yaUZQNG64i
    XZ+0/kq6uOZFO0QtatdWKfXnRQ99Bj91R5OIFnk54jN0mkUiqlO3XDW+Ml+98mKB6tW7rWpZcPc+
    0zg4tLrYlUc86E6eGDjIMubVpcusearfgIYGRk6brhZVr/JcHzooL7550jedLExopWcApi2ZUqhu
    7JLvrVsQU81zkzOPeemMRYvVuQsX7PbiDQY5JvZonftK+1VY8H9utx530h0ob+jmRYqj6ouaYvEe
    nW/WlYjp8cwbMm682tPwqW1R4tj/2SH13IRJYl4moZvXpiSqDr7dXtQHxa/PK3/+BWsK1dTgHu6V
    8tQJ3bwFkwpFrUOQ50s1r3levm8zZcq17+BBaw7K8lEK5qzkYeark9A8p7P3GzDK+nd3DQow+6UC
    8SVN82iuv38im7NtaXtV1CVq6Rgw4pksmbdi3bu2De7YfaBBxcqfvqPrUjFQNTQ22lfdUVVT68rT
    JKF5DnSmUjgdqg4mSS9pmsfDJR3G6ToH0iW9aV7LWLHYXKllTDt0LTAtkYIaamp1QjVv++uyGUxV
    dJ0DNVXSm+b1qRxpl84ddfX1Lp1O/d69tsod0vs5hGre9xu8o+fpLR1cGhNTD6Z57C9KMWXefJdO
    Z94bb9oqd1ROnS7qITTzHimMqivbO3g0DdVyk3WQBhBztK35YKNdOnc8O3acS6fDZFgKaXLsEJp5
    rdrliBqp89cJcs/m7Tvs0rkjGfN4b0kPoZn3UJuIOrnZ22yP1fmvUx+O5gSqebV1m+zSuYNVhq7T
    WbDiLVvljplLlop6CLXP+2qtvGLIL/1vimISdMBgzSoFZyu6Tqd+jzxgsPaV9BCqee/NjYk6v6lK
    9cwiUc/STtf1HDpM3b592y7h3Thx5ozK69HLpYWuAwaqS5cv26q7ceb8efVYaReP3iFU8zj1knSw
    ZXHMmnCjY0Ogalo7UQfSCM3qQQr2H/XFP7ssXx45Yl91ByeCep4moZoH+1fG3xD4tT7x8kwyj8nw
    b9ev26V0B6d+7H4zKvudAH537FjqyzOHdJnHEuzmXq/WjxObvNMbv7nhywsX2aVsWtC8+48aLeap
    E7p5wKZi0A2AQRV5nvR4E+uJc+b61kApqInxBgmd/4V5QP/mt18HDC7sRHftmeu5lmhV0rn/ALX2
    32bqd4BFnDx7Vi1cWS2uff0IbB47qexxmUj9QutYjupd3tYD6abWBBMrh+apNbOKrNF1+ugCa4ri
    XGfwMPPtViavhU3YMOAAnuUb/R07L0yOSeOadE88ApsXFGff30ynhlJgM51CU6vN9EzgnpvHBFUy
    iVraePiwJ53DF5ZTZnomENg85kNUd2oJi2Wpr4OmmkfN4x4zHfiVFc8Dv8NzuhNqOidilGvA6DGu
    eZwO78AAQn6ciEk6+rw5VcvjvqNDYPOoIUwaKShrxAuXLlkH4aYuGfMYDc10WF5Ta31hPJOfcUhr
    U/JlINi6c6elRYdBpo6++Yfjx61lGNfRm4MD5rJ1j3FoGHnjDSBNarYUgMLyMszKpb7tXpoHfPs8
    h3Wp1LzNfNk54XxC1wDGUmYzXYefh6z/cKtVm4EBxa9VQGDzYr3LrUMRjHEKkk7zaFKYQA2hGQU1
    z+85NFWpXDrkz3vx10GqxQ6BzeNboBk5n8k4nebRh+k1hWfxTF0D1EyWUs5nv+dgQqKaxzuCdE0i
    sHl02NQ8ah0mXr12La3m0f9wik9+wLNTMY/86MPo8yi31OfxmT6PWoqG9+DZukYna56mSZt5WWSy
    5qVA1rwUyJqXAlnzkiai/gHSD7RkTyihogAAAABJRU5ErkJggg=="
}
```


6. Additional Considerations

This section is non-normative.

6.1 Field updates and metadata

Metadata statements are intended to be stable once they have been published. When authenticators are updated in the field, such updates are expected to improve the authenticator security (for example, improve FRR or FAR). The `authenticatorVersion` must be updated if firmware updates fixing severe security issues (e.g. as reported previously) are available.

NOTE

The metadata statement is assumed to relate to all authenticators having the same AAID.

NOTE

The FIDO Server is recommended to assume increased risk if the `authenticatorVersion` specified in the metadata statement is newer (higher) than the one present in the authenticator.

NORMATIVE

Significant changes in authenticator functionality are not anticipated in firmware updates. For example, if an authenticator vendor wants to modify a PIN-based authenticator to use "Speaker Recognition" as a user verification method, the vendor must assign a new AAID to this authenticator.

NORMATIVE

A single authenticator implementation could report itself as two "virtual" authenticators using different AAIDs. Such implementations must properly (i.e. according to the security characteristics claimed in the metadata) protect `UAuth` keys and other sensitive data from the other "virtual" authenticator - just as a normal authenticator would do.

NOTE

Authentication keys (`UAuth.pub`) registered for one AAID cannot be used by authenticators reporting a different AAID - even when running on the same hardware (see section "Authentication Response Processing Rules for FIDO Server" in [UAFProtocol]).

A. References

A.1 Normative references

[FIDORestrictedOperatingEnv]
Laurence Lundblade; Meagan Karlsson. *FIDO Authenticator Allowed Restricted Operating Environments List* August 2016. Draft. URL: https://fidoalliance.org/specs/fido-security-requirements-v1.1-fd-20171108/fido-authenticator-allowed-restricted-operating-environments-list-v1.1-fd-20171108.html

[ISO19795-1]
ISO/IEC JTC 1/SC 37 Information Technology - Biometric peformance testing and reporting - Part 1: Principles and framework. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=41447

[ISO30107-1]
ISO/IEC JTC 1/SC 37 Information Technology - Biometrics - Presentation attack detection - Part 1: Framework. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=53227

[RFC2049]
N. Freed; N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples (RFC 2049)*. November 1996. URL: http://www.ietf.org/rfc/rfc2049.txt

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

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

[RFC4122]
P. Leach. *A Universally Unique IDentifier (UUID) URN Namespace*. July 2005. URL: https://tools.ietf.org/html/rfc4122

[UAFProtocol]
R. Lindemann; D. Baghdasaryan; E. Tiffany; D. Balfanz; B. Hill; J. Hodges. *FIDO UAF Protocol Specification v1.0*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html

[UAFRegistry]
R. Lindemann; D. Baghdasaryan; B. Hill. *FIDO UAF Registry of Predefined Values*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-reg-v1.2-rd-20171128.html

[WebIDL-ED]
Cameron McCormack. *Web IDL*. 13 November 2014. Editor's Draft. URL: http://heycam.github.io/webidl/'

A.2 Informative references

[AndroidUnlockPattern]
Android Unlock Pattern Security Analysis. Published. URL: http://www.sinustrom.info/2012/05/21/android-unlock-pattern-security-analysis/

[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAA Algorithm*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html

[FIDOMetadataService]

R. Lindemann; B. Hill; D. Baghdasaryan. *FIDO Metadata Service v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-metadata-service-v2.0-ps-20170927.html

[FIDORegistry]
R. Lindemann; D. Baghdasaryan; B. Hill. *FIDO Registry of Predefined Values*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-registry-v2.0-ps-20170927.html

[FIPS140-2]
FIPS PUB 140-2: Security Requirements for Cryptographic Modules. May 2001. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

[FIPS180-4]
FIPS PUB 180-4: Secure Hash Standard (SHS). March 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[FIPS186-4]
FIPS PUB 186-4: Digital Signature Standard (DSS). July 2013. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[FIPS198-1]
FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC). July 2008. URL: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf

[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). November 2008. URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en

[MoreTopWorstPasswords]
Mark Burnett. *10000 Top Passwords*. URL: https://xato.net/passwords/more-top-worst-passwords/

[PNG]
Tom Lane. *Portable Network Graphics (PNG) Specification (Second Edition)*. 10 November 2003. W3C Recommendation. URL: https://www.w3.org/TR/PNG/

[RFC4648]
S. Josefsson. *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*. October 2006. URL: http://www.ietf.org/rfc/rfc4648.txt

[RFC5280]
D. Cooper; S. Santesson; S. Farrell; S.Boeyen; R. Housley; W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. URL: http://www.ietf.org/rfc/rfc5280.txt

[SP800-38B]
M. Dworkin. *NIST Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. May 2005. URL: http://dx.doi.org/10.6028/NIST.SP.800-38B

[SP800-38C]
M. Dworkin. *NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. July 2007. URL: http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf

[SP800-38D]
M. Dworkin. *NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. November 2007 URL: https://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf

[SP800-38F]
M. Dworkin. *NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*. December 2012. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf

[SP800-90C]
Elaine Barker; John Kelsey. *NIST Special Publication 800-90C: Recommendation for Random Bit Generator (RBG) Constructions*. August 2012. URL: http://csrc.nist.gov/publications/drafts/800-90/sp800_90c_second_draft.pdf

[SP800-90ar1]
Elaine Barker; John Kelsey. *NIST Special Publication 800-90a: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. August 2012. URL: http://dx.doi.org/10.6028/NIST.SP.800-90Ar1

[UAFAuthnrCommands]
D. Baghdasaryan; J. Kemp; R. Lindemann; R. Sasson; B. Hill. *FIDO UAF Authenticator Commands v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-authnr-cmds-v1.2-rd-20171128.html

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

[iPhonePasscodes]
Daniel Amitay. *Most Common iPhone Passcodes*. URL: http://danielamitay.com/blog/2011/6/13/most-common-iphone-passcodes

FIDO Metadata Service

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-metadata-service-v2.0-ps-20170927.html
Previous version:
https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-metadata-service-v2.0-rd-20161004.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Brad Hill, PayPal, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The FIDO Authenticator Metadata Specification defines so-called "Authenticator Metadata" statements. The metadata statements contain the "Trust Anchor" required to validate the attestation object, and they also describe several other important characteristics of the authenticator.

The metadata service described in this document defines a baseline method for relying parties to access the latest metadata statements.

Status of This Document

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

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

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

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as`code`.

String literals are enclosed in “”, e.g. “UAF-TLV”.

In formulas we use “|” to denote byte wise concatenation operations.

The notation `base64url(byte[8..64])` reads as 8-64 bytes of data encoded in base64url, "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as `required`.

WebIDL dictionary members must not have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

UAF specific terminology used in this document is defined in [FIDOGlossary].

All diagrams, examples, notes in this specification are non-normative.

> NOTE
>
> Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

[FIDOMetadataStatement] defines authenticator metadata statements.

These metadata statements contain the trust anchor required to verify the attestation object (more specifically the

`KeyRegistrationData` object), and they also describe several other important characteristics of the authenticator, including supported authentication and registration assertion schemes, and key protection flags.

These characteristics can be used when defining policies about which authenticators are acceptable for registration or authentication.

The metadata service described in this document defines a baseline method for relying parties to access the latest metadata statements.

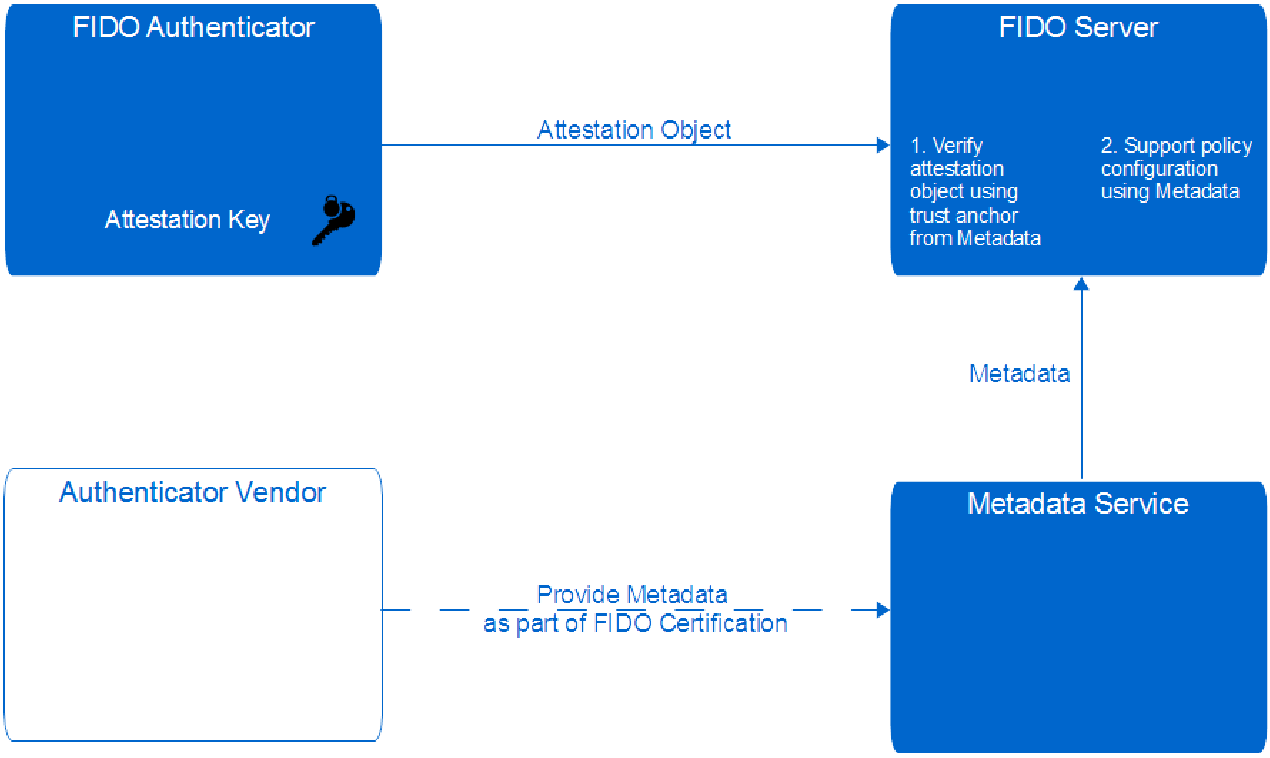

Fig. 1 FIDO Metadata Service Architecture Overview

2.1 Scope

This document describes the FIDO Metadata Service architecture in detail and it defines the structure and interface to access this service. It also defines the flow of the metadata related messages and presents the rationale behind the design choices.

2.2 Detailed Architecture

The metadata "table-of-contents" (TOC) file contains a list of metadata statements related to the authenticators known to the FIDO Alliance (FIDO Authenticators).

The FIDO Server downloads the metadata TOC file from a well-known FIDO URL and caches it locally.

The FIDO Server verifies the integrity and authenticity of this metadata TOC file using the digital signature. It then iterates through the individual entries and loads the metadata statements related to authenticator AAIDs relevant to the relying party.

Individual metadata statements will be downloaded from the URL specified in the entry of the metadata TOC file, and may be cached by the FIDO Server as required.

The integrity of the metadata statements will be verified by the FIDO Server using the hash value included in the related entry of the metadata TOC file.

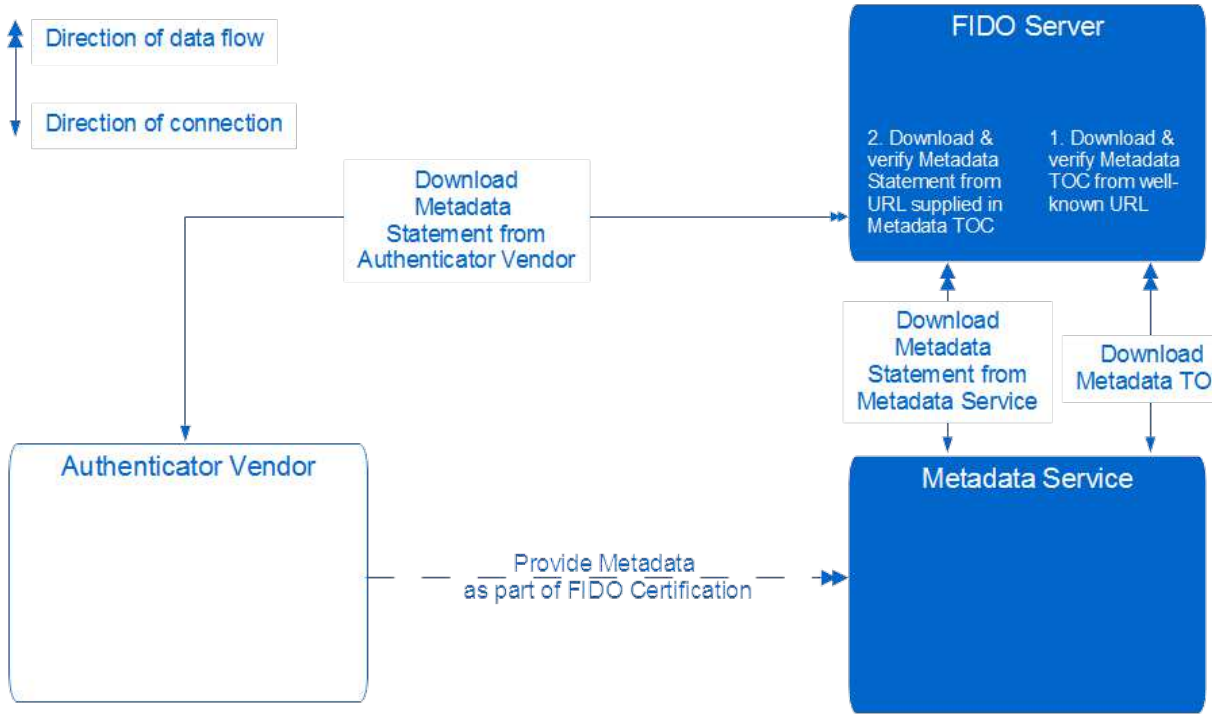

Fig. 2 FIDO Metadata Service Architecture

NOTE

The single arrow indicates the direction of the network connection, the double arrow indicates the direction of the data flow.

NOTE

The metadata TOC file is accessible at a well-known URL published by the FIDO Alliance.

NOTE

The relying party decides how frequently the metadata service is accessed to check for metadata TOC updates.

3. Metadata Service Details

This section is normative.

NOTE

The relying party can decide whether it wants to use the metadata service and whether or not it wants to accept certain authenticators for registration or authentication.

The relying party could also obtain metadata directly from authenticator vendors or other trusted sources.

3.1 Metadata TOC Format

NOTE

The metadata service makes the metadata TOC object (see Metadata TOC) accessible to FIDO Servers.

This object is a "table-of-contents" for metadata, as it includes the AAID, the download URL and the hash value of the individual metadata statements. The TOC object contains one signature.

3.1.1 Metadata TOC Payload Entry dictionary

Represents the MetadataTOCPayloadEntry

WebIDL

```
dictionary MetadataTOCPayloadEntry {
    AAID                      aaid;
    AAGUID                    aaguid;
    DOMString[]               attestationCertificateKeyIdentifiers;
    DOMString                 hash;
    DOMString                 url;
    required StatusReport[]   statusReports;
    required DOMString        timeOfLastStatusChange;
    DOMString                 rogueListURL;
    DOMString                 rogueListHash;
};
```

3.1.1.1 Dictionary `MetadataTOCPayloadEntry` Members

`aaid` of type AAID
: The AAID of the authenticator this metadata TOC payload entry relates to. See [UAFProtocol] for the definition of the AAID structure. This field must be set if the authenticator implements FIDO UAF.

> NOTE
>
> FIDO UAF authenticators support AAID, but they don't support AAGUID.

`aaguid` of type AAGUID
: The Authenticator Attestation GUID. See [FIDOKeyAttestation] for the definition of the AAGUID structure. This field must be set if the authenticator implements FIDO 2.

> NOTE
>
> FIDO 2 authenticators support AAGUID, but they don't support AAID.

`attestationCertificateKeyIdentifiers` of type array of DOMString
: A list of the attestation certificate public key identifiers encoded as hex string. This value must be calculated according to method 1 for computing the keyIdentifier as defined in [RFC5280] section 4.2.1.2. The hex string must not contain any non-hex characters (e.g. spaces). All hex letters must be lower case. This field must be set if neither `aaid` nor `aaguid` are set. Setting this field implies that the attestation certificate(s) are dedicated to a single authenticator model.

> NOTE
>
> FIDO U2F authenticators do not support AAID nor AAGUID, but they use attestation certificates dedicated to a single authenticator model.

`hash` of type DOMString
: `base64url(string[1..512])`

The hash value computed over the base64url encoding of the UTF-8 representation of the JSON encoded metadata statement available at `url` and as defined in [FIDOMetadataStatement]. The hash algorithm related to the signature algorithm specified in the JWTHeader (see Metadata TOC) must be used.

If this field is missing, the metadata statement has not been published.

> NOTE
>
> This method of base64url encoding the UTF-8 representation is also used by JWT [JWT] to avoid encoding ambiguities.

`url` of type DOMString
: Uniform resource locator (URL) of the encoded metadata statement for this authenticator model (identified by its AAID, AAGUID or attestationCertificateKeyIdentifier). This URL must point to the base64url encoding of the UTF-8 representation of the JSON encoded metadata statement as defined in [FIDOMetadataStatement].

If this field is missing, the metadata statement has not been published.

`encodedMetadataStatement = base64url(utf8(JSONMetadataStatement))`

> NOTE
>
> This method of the base64url encoding the UTF-8 representation is also used by JWT [JWT] to avoid encoding ambiguities.

statusReports of type array of required StatusReport
An array of status reports applicable to this authenticator.

timeOfLastStatusChange of type required DOMString
ISO-8601 formatted date since when the status report array was set to the current value.

rogueListURL of type DOMString
URL of a list of rogue (i.e. untrusted) individual authenticators.

rogueListHash of type DOMString
`base64url(string[1..512])`

The hash value computed over the Base64url encoding of the UTF-8 representation of the JSON encoded rogueList available at `rogueListURL` (with type rogueListEntry[]). The hash algorithm related to the signature algorithm specified in the JWTHeader (see Metadata TOC) must be used.

This hash value must be present and non-empty whenever `rogueListURL` is present.

> NOTE
>
> This method of base64url-encoding the UTF-8 representation is also used by JWT [JWT] to avoid encoding ambiguities.

EXAMPLE 1: UAF Metadata TOC Payload


```
{ "no": 1234, "nextUpdate": "2014-03-31",
  "entries": [
   { "aaid": "1234#5678",
     "hash": "90da8da6de23248abb34da0d4861f4b30a793e198a8d5baa7f98f260db71acd4",
     "url": "https://fidoalliance.org/metadata/1234%x23abcd",
     "rogueListHash": "b5079cf40fd7ed174c645cc04df1e72b7f1229590585d16df62dd20b9541c6b5",
     "rogueListURL": "https://fidoalliance.org/metadata/1234%x23abcd.rl",
     "statusReports": [
                         { status: "FIDO_CERTIFIED", effectiveDate: "2014-01-04"}
                     ],
     "timeOfLastStatusChange": "2014-01-04"
     },
   { "attestationCertificateKeyIdentifiers": ["7c0903708b87115b0b422def3138c3c864e44573"],
     "hash": "785d16df640fd7b50ed174cb5645cc0f1e72b7f19cf22959052dd20b9541c64d",
     "url": "https://authnr-vendor-a.com/metadata/9876%x234321",
     "statusReports": [
                         { status: "FIDO_CERTIFIED", effectiveDate: "2014-01-07"},
                         { status: "UPDATE_AVAILABLE", effectiveDate: "2014-02-19",
                           url: "https://example.com/update1234" }
                     ],
     "timeOfLastStatusChange": "2014-02-19"
     }
  ]
}
```


> NOTE
>
> The character `#` is a reserved character and not allowed in URLs [RFC3986]. As a consequence it has been replaced by its hex value `%x23`.
>
> The authenticator vendors can decide to let the metadata service publish its metadata statements or to publish metadata statements themselves. Authenticator vendors can restrict access to the metadata statements they publish themselves.

3.1.2 StatusReport dictionary

> NOTE
>
> Contains an `AuthenticatorStatus` and additional data associated with it, if any.

New StatusReport entries will be added to report known issues present in firmware updates.

The latest StatusReport entry must reflect the "current" status. For example, if the latest entry has status USER_VERIFICATION_BYPASS, then it is recommended assuming an increased risk associated with all authenticators of this AAID; if the latest entry has status UPDATE_AVAILABLE, then the update is intended to address at least all previous issues *reported* in this StatusReport dictionary.

WebIDL

```
dictionary StatusReport {
    required AuthenticatorStatus status;
    DOMString                    effectiveDate;
    DOMString                    certificate;
    DOMString                    url;
    DOMString                    certificationDescriptor;
    DOMString                    certificateNumber;
    DOMString                    certificationPolicyVersion;
    DOMString                    certificationRequirementsVersion;
};
```

*3.1.2.1 Dictionary **StatusReport** Members*

status of type required AuthenticatorStatus
Status of the authenticator. Additional fields may be set depending on this value.

effectiveDate of type DOMString
ISO-8601 formatted date since when the status code was set, if applicable. If no date is given, the status is assumed to be effective while present.

certificate of type DOMString
Base64-encoded [RFC4648] (not base64url!) DER [ITU-X690-2008] PKIX certificate value related to the current status, if applicable.

> NOTE
>
> As an example, this could be an Attestation Root Certificate (see [FIDOMetadataStatement]) related to a set of compromised authenticators (ATTESTATION_KEY_COMPROMISE).

url of type DOMString
HTTPS URL where additional information may be found related to the current status, if applicable.

> NOTE
>
> For example a link to a web page describing an available firmware update in the case of status UPDATE_AVAILABLE, or a link to a description of an identified issue in the case of status USER_VERIFICATION_BYPASS.

certificationDescriptor of type DOMString
Describes the externally visible aspects of the Authenticator Certification evaluation.

certificateNumber of type DOMString
The unique identifier for the issued Certification

certificationPolicyVersion of type DOMString
The version of the Authenticator Certification Policy the implementation is Certified to, e.g. "1.0.0".

certificationRequirementsVersion of type DOMString
The version of the Authenticator Security Requirements the implementation is Certified to, e.g. "1.0.0".

3.1.3 AuthenticatorStatus enum

This enumeration describes the status of an authenticator model as identified by its AAID and potentially some additional information (such as a specific attestation key).

WebIDL

```
enum AuthenticatorStatus {
    "NOT_FIDO_CERTIFIED",
    "FIDO_CERTIFIED",
    "USER_VERIFICATION_BYPASS",
```

```
        "ATTESTATION_KEY_COMPROMISE",
        "USER_KEY_REMOTE_COMPROMISE",
        "USER_KEY_PHYSICAL_COMPROMISE",
        "UPDATE_AVAILABLE",
        "REVOKED",
        "SELF_ASSERTION_SUBMITTED",
        "FIDO_CERTIFIED_L1",
        "FIDO_CERTIFIED_L2",
        "FIDO_CERTIFIED_L3",
        "FIDO_CERTIFIED_L4",
        "FIDO_CERTIFIED_L5"
};
```

Enumeration description	
`NOT_FIDO_CERTIFIED`	This authenticator is not FIDO certified.
`FIDO_CERTIFIED`	This authenticator has passed FIDO functional certification. This certification scheme is phased out and will be replaced by FIDO_CERTIFIED_L1.
`USER_VERIFICATION_BYPASS`	Indicates that malware is able to bypass the user verification. This means that the authenticator could be used without the user's consent and potentially even without the user's knowledge.
`ATTESTATION_KEY_COMPROMISE`	Indicates that an attestation key for this authenticator is known to be compromised. Additional data should be supplied, including the key identifier and the date of compromise, if known.
`USER_KEY_REMOTE_COMPROMISE`	This authenticator has identified weaknesses that allow registered keys to be compromised and should not be trusted. This would include both, e.g. weak entropy that causes predictable keys to be generated or side channels that allow keys or signatures to be forged, guessed or extracted.
`USER_KEY_PHYSICAL_COMPROMISE`	This authenticator has known weaknesses in its key protection mechanism(s) that allow user keys to be extracted by an adversary in physical possession of the device.
`UPDATE_AVAILABLE`	A software or firmware update is available for the device. Additional data should be supplied including a URL where users can obtain an update and the date the update was published. When this code is used, then the field `authenticatorVersion` in the metadata Statement [FIDOMetadataStatement] must be updated, if the update fixes severe security issues, e.g. the ones reported by preceding StatusReport entries with status code `USER_VERIFICATION_BYPASS`, `ATTESTATION_KEY_COMPROMISE`, `USER_KEY_REMOTE_COMPROMISE`, `USER_KEY_PHYSICAL_COMPROMISE`, `REVOKED`. NOTE Relying parties might want to inform users about available firmware updates.
`REVOKED`	The FIDO Alliance has determined that this authenticator should not be trusted for any reason, for example if it is known to be a fraudulent product or contain a deliberate backdoor.
`SELF_ASSERTION_SUBMITTED`	The authenticator vendor has completed and submitted the self-certification checklist to the FIDO Alliance. If this completed checklist is publicly available, the URL will be specified in `StatusReport.url`.
`FIDO_CERTIFIED_L1`	The authenticator has passed FIDO Authenticator certification at level 1. This level is the more strict successor of FIDO_CERTIFIED.
`FIDO_CERTIFIED_L2`	The authenticator has passed FIDO Authenticator certification at level 2. This level is more strict than level 1.
`FIDO_CERTIFIED_L3`	The authenticator has passed FIDO Authenticator certification at level 3. This level is more strict than level 2.
`FIDO_CERTIFIED_L4`	The authenticator has passed FIDO Authenticator certification at level 4. This level is more strict than level 3.
`FIDO_CERTIFIED_L5`	The authenticator has passed FIDO Authenticator certification at level 5. This level is more strict than level 4.

More values might be added in the future. FIDO Servers must silently ignore all unknown AuthenticatorStatus values.

3.1.4 RogueListEntry dictionary

NOTE

Contains a list of individual authenticators known to be rogue.

New `RogueListEntry` entries will be added to report new individual authenticators known to be rogue.

Old `RogueListEntry` entries will be removed if the individual authenticator is known to not be rogue any longer.

WebIDL

```
dictionary RogueListEntry {
    required DOMString sk;
    required DOMString date;
};
```

3.1.4.1 Dictionary `RogueListEntry` Members

`sk` of type required DOMString
: Base64url encoding of the rogue authenticator's secret key (sk value, see [FIDOEcdaaAlgorithm], section ECDAA Attestation).

> NOTE
>
> In order to revoke an individual authenticator, its secret key (sk) must be known.

`date` of type required DOMString
: ISO-8601 formatted date since when this entry is effective.

EXAMPLE 2: RogueListEntry[] example


```
[
  { "sk": "30efa86aa6de25249acb35da0d4861f4b30a793e198a8d5baa7e96f240da51f3",
    "date": "2016-06-07"},
  { "sk": "93de8da6de23248abb34da0d4861f4b30a793e153a8d5bb27f98f260db71acd4",
    "date": "2016-06-09"},
]
```


3.1.5 Metadata TOC Payload dictionary

Represents the MetadataTOCPayload

WebIDL

```
dictionary MetadataTOCPayload {
    DOMString                           legalHeader;
    required Number                     no;
    required DOMString                  nextUpdate;
    required MetadataTOCPayloadEntry[]  entries;
};
```

3.1.5.1 Dictionary `MetadataTOCPayload` Members

`legalHeader` of type DOMString
: The legalHeader, if present, contains a legal guide for accessing and using metadata, which itself may contain URL(s) pointing to further information, such as a full Terms and Conditions statement.

`no` of type required Number
: The serial number of this UAF Metadata TOC Payload. Serial numbers must be consecutive and strictly monotonic, i.e. the successor TOC will have a `no` value exactly incremented by one.

`nextUpdate` of type required DOMString
: ISO-8601 formatted date when the next update will be provided at latest.

`entries` of type array ofrequired MetadataTOCPayloadEntry
: List of zero or more MetadataTOCPayloadEntry objects.

3.1.6 Metadata TOC

The metadata table of contents (TOC) is a JSON Web Token (see [JWT] and [JWS]).

It consists of three elements:

- The base64url encoding, without padding, of the UTF-8 encoded JWT Header (see example below),
- the base64url encoding, without padding, of the UTF-8 encoded UAF Metadata TOC Payload (see example at the beginning of section Metadata TOC Format),
- and the base64url-encoded, also without padding, JWS Signature [JWS] computed over the to-be-signed payload using the Metadata TOC signing key, i.e.

  ```
  tbsPayload = EncodedJWTHeader | "." | EncodedMetadataTOCPayload
  ```

All three elements of the TOC are concatenated by a period ("."):

```
MetadataTOC = EncodedJWTHeader | "." | EncodedMetadataTOCPayload | "." | EncodedJWSSignature
```

The hash algorithm related to the signing algorithm specified in the JWT Header (e.g. SHA256 in the case of "ES256") must also be used to compute the hash of the metadata statements (see section Metadata TOC Payload Entry Dictionary).

3.1.6.1 Examples

This section is non-normative.

EXAMPLE 3: Encoded Metadata Statement


```
eyAiQUFJRCI6ICIxMjM0IzU2NzgiLA0KICAiQXR0ZXN0YXRpb25Sb290Q2VydGlmaWNhdGUiOiAi
TUlJQ1BUQ0NBZU9nQXdJQkFnSUpBT3VleHZVM095MndNQW9HQ0NxR1NNNDlCQU1DTUhzeElEQWVC
Z05WQkFNTQ0KRjFOaGJYQnNaU0JCZEhSbGMzUmhkR2x2YmlCU2IyOTBNUll3RkFZRFZRUUtEQTFH
U1VSUElFRnNiR2xoYm1ObA0KTVJFd0R3WURWUVFMREFoVlFVWWdWRmRITERFU01CQUdBMVVFQnd3
SlVHRnNieUJCYkhSdk1Rc3dDUVlEVlFRSQ0KREFKRFFURUxNQWtHQTFVRUJoTUNWVk13SGhjTk1U
UXdOakU0TVRNek16TXlXaGNOTkRFeE1UQXpNVE16TXpNeQ0KV2pCN01TQXdIZ1lEVlFRRERCZFRZ
VzF3YkdVZ1FYUjBaWE4wWVhScGIyNGdVbTl2ZERFV01CUUdBMVVFQ2d3Tg0KUmtsRVR5QkJiR3hw
WVc1alpURVJNQThHQTFVRUN3d0lWVUZHSUZSWFJ5d3hFakFRQmdOVkJBY01DVkJoYkc4Zw0KUVd4
MGJ6RUxNQWtHQTFVRUNBd0NRMEV4Q3pBSkJnTlZCQVlUQWxWVE1Ga3dFd1lIS29aSXpqMENBUVlJ
S29aSQ0KemowREFRY0RRZ0FFSDhodjJEMEhYYTU5L0JtcFE3UlplaEwvRk1HekZkMVFCZzl2QVVw
TlozYWpudVE5NFBSNw0KYU16SDMzblVTQnI4ZkhZRHJxT0JiNThweEdxSEpSeVgvNk5RTUU0d0hR
WURWUjBPQkJZRUZQb0hBM0NMaHhGYg0KQzBJdDd6RTR3OGhrNUVKL01COEdBMVVkSXdRWU1CYUFG
UG9IQTNDTGh4RmJDMEl0N3pFNHc4aGs1RUovTUF3Rw0KQTFVZEV3UUZNQU1CQWY4d0NnWUlLb1pJ
emowRUF3SURTQUF3UlFJaEFKMDZRU1h0OWloSWJFS1lLSWpzUGtyaQ0KVmRMSWd0ZnNiRFN1N0Vy
SmZ6cjRBaUJxb1lDWmYwK3pJNTVhUWVBSGpJekE5WG02M3JydUF4Qlo5cHM5ejJYTg0KbFE9PSIs
DQogICJEZXNjcmlwdGlvbiI6ICJGSURPIEFsbGlhbmNlIFNhbXBsZSBVQUYgQXV0aGVudGljYXRv
ciIsDQogICJVc2VyVmVyaWZpY2F0aW9uTWV0aG9kcyI6IDIsDQogICJWYWxpZEF0dGFjaG1lbnRU
eXBlcyI6IDEsDQogICJLZXlQcm90ZWN0aW9uIjogNiwNCiAgIk1hdGNoZXJQcm90ZWN0aW9uIjog
MiwNCiAgIlNlY3VyZURpc3BsYXkiOiA0LA0KICAiU2VjdXJlRGlzcGxheUNvbnRlbnRUeXBlcyI6
IFsiaW1hZ2UvcG5nIl0sDQogICJTZWN1cmVEaXNwbGF5UE5HQ2hhcmFjdGVyaXN0aWNzIjogW1sw
LDAsMSw2NCwwLDAsMSwyMjQsMTYsMiwwLDAsMF1dLA0KICAiaXNTZWNvbmRGYWN0b3JPbmx5Ijog
ImZhbHNlIiwNCiAgIkljb24iOiAiZGF0YTppbWFnZS9wbmc7YmFzZTY0LGlWQk9SdzBLR2dvQUFB
QU5TVWhFVWdBQUFFOEFBQUF2Q0FZQUFBQ2l3SmZjQUFBQUFYTlNSMElBcnM0YzZRQUFBQVJuUVUx
QkFBQ3gNCmp3djhZUVVBQUFBSmNFaFpjd0FBRHNNQUFBN0RBY2R2cUdRQUFBYWhTVVJCVkdoRDda
cjVieFJsR0lmmOUt6VEI4QU0vWUVoRTJXN3ANClFaY1dLS0JjbFNwSEFUbEVMMQVJFN2tORUNDTNG
a1dLMENLS1NNRDRklzS0JjZ1ZZDRFdHTkVTZEFZZaWR3Z2dnSkpwUmlNaEZjLzR3eTgNCjg4NHp1OU5k
bG5HVGZaSlAybbjNuTysrODg5MzNmdmVCQngrUHFrRGVkprVFV2QmJtMbXBVRFd2Ql1RJbXBjQ1NadlhM
Q2RYOVIwNVNrMTkNCmJiNWF0ZjU5OWZHKy9sckE1NDFxNDdhUDFMMTFhOVNJeVZZOVdlk4SWk4ZDVr
R1RzaTMwTkZ2N2FpOW43UVpQTXdiZHlmMmVyVTJYTENlVkeTgrWmNhTm1HaWFIOHlYTjlSVWQz
YTE4bkYwZlVsb3ZaaKzBDVHpxXcGQyVmorZU9tMWJFeXk2RHg0aTVwVVFVV3ZlbzUwNnEyMjNjNCmR0
dVdCSXVmZnI2b1dwVjBGUE5MaG93MTc1MU5tMjFMdlBIM3JWdFdxZnNo2NkxmcwW4dFg3RlJsOVlG
U1hzbVNzZWI5Y2VPR2JZazcNCk1OVWNIVGc4WnNpTWU5cmZzVWFhVi9KTVg5c3FFkekRDVU3ZwMGta
SG1UWmc5eERdiTEhjTW5UaGIxNmVKK21WZlFxOHlhVVpRTkc2NGkNCmhaKzAva3E2dU9aRk8wUXRh
dGRXS2ZYblJROTlCajkxUjVPSUZuazU0ak4wbWtVaXFsTzNYRFcrTWwrOThtS0I2dFc3cldwWmNQ
YysNCjB6ZzR0HJZbFVjODZFNmVHRGpTXViVnBjdXNlYXJmZ0lZR1JrNmJjyaFpWci9KY0h6b29M
```

```
VU1SakhFS2trN3phRktZUUEyaEdRVTENCnorODVORldwWERya3ozdngxMEdxeFE2QnplTmJvQms1
bjhrNG5lYlJoK2sxaFdmeFRGMEQxRXlXVXM1bnYrZGdRcUtheHp1Q2RFMGkNCnNIbDAyTlE4YWgw
bVhyMTJMYTNtMGY5d2lrOSt3TE5UTVkvODZNUG84eWkzMU9meG1UNlBXb3FHOStEWnVrWW5hNTZt
U1p0NVdXU3kNCjVxVkExcndVeUpxWEFsbnpraWFpL2dIU0Q3UmtUeWlob2dBQUFBQkpSVTVFcmtK
Z2dnPT0iLA0KICAiQXNzZXJ0aW9uU2NoZW1lIjogIlVBRlYxVEWIiwNCiAgIkF1dGhlbnRpY2F0
aW9uQWxnb3JpdGhtIjogMSwNCiAgIkF0dGVzdGF0aW9uVHlwZXMiOiBbMTYzOTFdLA0KICAiVVBW
IjogW1sxLDBdXQ0KfQ0K
```


EXAMPLE 4: JWT Header

```
{"typ":"JWT",
 "alg":"ES256"
 "x5t#S256":"7231962210d2933ec993a77b4a7203898ab74cdf974ff02d2de3f1ec7cb9de68"}
```

In order to produce the tbsPayload, we first need the base64url-encoded (without padding) JWT Header:

EXAMPLE 5: Encoded JWT Header


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IkVTMjU2IiwKICJ4NXQjUzI1NiI6IjcyMzE5NjIyMTBkMjkz
M2VjOTkzYTc3YjRhNzIwMzg5OGFiNzRjZGY5NzRmZjAyZDJkZTNmMWVjN2NiOWRlNjgifQ
```


then we have to append a period (".") and the base64url encoding of the `EncodedMetadataTOCPayload` (taken from the example in section Metadata TOC Format):

EXAMPLE 6: tbsPayload


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IkVTMjU2IiwKICJ4NXQjUzI1NiI6IjcyMzE5NjIyMTBkMjkz
M2VjOTkzYTc3YjRhNzIwMzg5OGFiNzRjZGY5NzRmZjAyZDJkZTNmMWVjN2NiOWRlNjgifQ.
eyAibm8iOiAxMjM0LCAibmV4dC11cGRhdGUiOiAiMzEtMDMtMjAxNCIsDQogICJlbnRyaWVzIjog
Ww0KICAgeyAiYWFpZCI6ICIxMjM0IzU2NzgiLCANCiAgICAgImhhc2giOiAiOTBkYThkYTZkZTIz
MjQ4YWJiMzRkYTBkNDg2MWY0YjMwYTc5M2UxOThhOGQ1YmFhN2Y5OGYyNjBkYjcxYWNkNCIsIA0K
ICAgICAidXJsIjogImh0dHBzOi8vZmlkb2FsbGlhbmNlLm9yZy9tZXRhZGF0YS8xMjM0JXgyM2Fi
Y2QiLCANCiAgICAgInN0YXR1cyI6ICJmaWRvQ2VydGlmaWVkIg0KICAgICAidGltZU9mTGFzdFN0
YXR1c0NoYW5nZSI6ICIiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAiMjAxNC0wMS0wNCIg
fSwNCiAgIHsgImFhaWQiOiAiOTg3NiM0MzIxIiwgDQogICAgICJoYXNoIjogIjc4NWQxNmRmNjQw
ZmQ3YjUwZWQxNzRjYjU2NDVjYzBmMWU3MmI3ZjE5Y2YyYjk1OTA1MmRkMjBiOTU0MWM2NGQiLA0K
ICAgICAidXJsIjogImh0dHBzOi8vYXV0aG5yLXZlbmRvci1hLmNvbS9tZXRhZGF0YS85ODc2JXgy
MzQzMjEiLA0KICAgICAic3RhdHVzIjogImZpZG9DZXJ0aWZpZWQiDQogICAgICJ0aW1lT2ZMYXN0
U3RhdHVzQ2hhbmdlIjogIjIwMTQtMDItMTkiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAi
MjAxNC0wMS0wNyIgfQ0KICBdDQp9DQo
```


and finally we have to append another period (".") followed by the base64url-encoded signature.

EXAMPLE 7: JWT


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IkVTMjU2IiwKICJ4NXQjUzI1NiI6IjcyMzE5NjIyMTBkMjkz
M2VjOTkzYTc3YjRhNzIwMzg5OGFiNzRjZGY5NzRmZjAyZDJkZTNmMWVjN2NiOWRlNjgifQ.
eyAibm8iOiAxMjM0LCAibmV4dC11cGRhdGUiOiAiMzEtMDMtMjAxNCIsDQogICJlbnRyaWVzIjog
Ww0KICAgeyAiYWFpZCI6ICIxMjM0IzU2NzgiLCANCiAgICAgImhhc2giOiAiOTBkYThkYTZkZTIz
MjQ4YWJiMzRkYTBkNDg2MWY0YjMwYTc5M2UxOThhOGQ1YmFhN2Y5OGYyNjBkYjcxYWNkNCIsIA0K
ICAgICAidXJsIjogImh0dHBzOi8vZmlkb2FsbGlhbmNlLm9yZy9tZXRhZGF0YS8xMjM0JXgyM2Fi
Y2QiLCANCiAgICAgInN0YXR1cyI6ICJmaWRvQ2VydGlmaWVkIg0KICAgICAidGltZU9mTGFzdFN0
YXR1c0NoYW5nZSI6ICIiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAiMjAxNC0wMS0wNCIg
fSwNCiAgIHsgImFhaWQiOiAiOTg3NiM0MzIxIiwgDQogICAgICJoYXNoIjogIjc4NWQxNmRmNjQw
ZmQ3YjUwZWQxNzRjYjU2NDVjYzBmMWU3MmI3ZjE5Y2YyYjk1OTA1MmRkMjBiOTU0MWM2NGQiLA0K
ICAgICAidXJsIjogImh0dHBzOi8vYXV0aG5yLXZlbmRvci1hLmNvbS9tZXRhZGF0YS85ODc2JXgy
MzQzMjEiLA0KICAgICAic3RhdHVzIjogImZpZG9DZXJ0aWZpZWQiDQogICAgICJ0aW1lT2ZMYXN0
U3RhdHVzQ2hhbmdlIjogIjIwMTQtMDItMTkiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAi
MjAxNC0wMS0wNyIgfQ0KICBdDQp9DQo.
AP-qoJ3VPzj7L6lCE1UzHzJYQnszFQ8d2hJz51sPASgyABK5VXOFnAHzBTQRRkgwGqULy6PtTyUV
zKxM0HrvoyZq
```


> **NOTE**
>
> The line breaks are for display purposes only.

The signature in the example above was computed with the following ECDSA key

EXAMPLE 8: ECDSA Key used for signature computation

```
x: d4166ba8843d1731813f46f1af32174b5c2f6013831fb16f12c9c0b18af3a9b4
y: 861bc2f803a2241f4939bd0d8ecd34e468e42f7fdccd424edb1c3ce7c4dd04e
d: 3744c426764f331f153e182d24f133190b6393cea480a8eec1c722fce161fe2d
```

3.1.7 Metadata TOC object processing rules

The FIDO Server must follow these processing rules:

1. The FIDO Server must be able to download the latest metadata TOC object from the well-known URL, when appropriate. The `nextUpdate` field of the [Metadata TOC] specifies a date when the download should occur at latest.
2. If the `x5u` attribute is present in the JWT Header, then:
    1. The FIDO Server must verify that the URL specified by the `x5u` attribute has the same web-origin as the URL used to download the metadata TOC from. The FIDO Server should ignore the file if the web-origin differs (in order to prevent loading objects from arbitrary sites).
    2. The FIDO Server must download the certificate (chain) from the URL specified by the `x5u` attribute [JWS]. The certificate chain must be verified to properly chain to the metadata TOC signing trust anchor according to [RFC5280]. All certificates in the chain must be checked for revocation according to [RFC5280].
    3. The FIDO Server should ignore the file if the chain cannot be verified or if one of the chain certificates is revoked.
3. If the `x5u` attribute is missing, the chain should be retrieved from the `x5c` attribute. If that attribute is missing as well, Metadata TOC signing trust anchor is considered the TOC signing certificate chain.
4. Verify the signature of the Metadata TOC object using the TOC signing certificate chain (as determined by the steps above). The FIDO Server should ignore the file if the signature is invalid. It should also ignore the file if its number (`no`) is less or equal to the number of the last Metadata TOC object cached locally.
5. Write the verified object to a local cache as required.
6. Iterate through the individual entries (of type `MetadataTOCPayloadEntry`). For each entry:
    1. Ignore the entry if the AAID, AAGUID or attestationCertificateKeyIdentifiers is not relevant to the relying party (e.g. not acceptable by any policy)
    2. Download the metadata statement from the URL specified by the field `url`. Some authenticator vendors might require authentication in order to provide access to the data. Conforming FIDO Servers should support the HTTP Basic, and HTTP Digest authentication schemes, as defined in [RFC2617].
    3. Check whether the status report of the authenticator model has changed compared to the cached entry by looking at the fields `timeOfLastStatusChange` and `statusReport`. Update the status of the cached entry. It is up to the relying party to specify behavior for authenticators with status reports that indicate a lack of certification, or known security issues. However, the status `REVOKED` indicates significant security issues related to such authenticators.

        NOTE

        Authenticators with an unacceptable status should be marked accordingly. This information is required for building registration and authentication policies included in the registration request and the authentication request [UAFProtocol].

    4. Compute the hash value of the (base64url encoding without padding of the UTF-8 encoded) metadata statement downloaded from the URL and verify the hash value to the hash specified in the field `hash` of the metadata TOC object. Ignore the downloaded metadata statement if the hash value doesn't match.
    5. Update the cached metadata statement according to the dowloaded one.

4. Considerations

This section is non-normative.

This section describes the key considerations for designing this metadata service.

Need for Authenticator Metadata When defining policies for acceptable authenticators, it is often better to describe the required authenticator characteristics in a generic way than to list individual authenticator AAIDs. The metadata statements provide such information. Authenticator metadata also provides the trust anchor required to verify attestation objects.

The metadata service provides a standardized method to access such metadata statements.

Integrity and Authenticity Metadata statements include information relevant for the security. Some business verticals might even have the need to document authenticator policies and trust anchors used for verifying attestation objects for auditing purposes.

It is important to have a strong method to verify and proof integrity and authenticity and the freshness of metadata statements. We are using a single digital signature to protect the integrity and authenticity of the Metadata TOC object and we protect the integrity and authenticity of the individual metadata statements by including their cryptographic hash values into the Metadata TOC object. This allows for flexible distribution of the metadata statements and the Metadata TOC object using standard content distribution networks.

Organizational Impact Authenticator vendors can delegate the publication of metadata statements to the metadata service in its entirety. Even if authenticator vendors choose to publish metadata statements themselves, the effort is very limited as the metadata statement can be published like a normal document on a website. The FIDO Alliance has control over the FIDO certification process and receives the metadata as part of that process anyway. With this metadata service, the list of known authenticators needs to be updated, signed and published regularly. A single signature needs to be generated in order to protect the integrity and authenticity of the metadata TOC object.

Performance Impact Metadata TOC objects and metadata statements can be cached by the FIDO Server.

The update policy can be specified by the relying party.

The metadata TOC object includes a date for the next scheduled update. As a result there is *no additional impact* to the FIDO Server during FIDO Authentication or FIDO Registration operations.

Updating the Metadata TOC object and metadata statements can be performed asynchronously. This reduces the availability requirements for the metadata service and the load for the FIDO Server.

The metadata TOC object itself is relatively small as it does not contain the individual metadata statements. So downloading the metadata TOC object does not generate excessive data traffic.

Individual metadata statements are expected to change less frequently than the metadata TOC object. Only the modified metadata statements need be downloaded by the FIDO Server.

Non-public Metadata Statements Some authenticator vendors might want to provide access to metadata statements only to their subscribed customers.

They can publish the metadata statements on access protected URLs. The access URL and the cryptographic hash of the metadata statement is included in the metadata TOC object.

High Security Environments Some high security environments might only trust internal policy authorities. FIDO Servers in such environments could be restricted to use metadata TOC objects from a proprietary trusted source only. The metadata service is the baseline for most relying parties.

Extended Authenticator Information Some relying parties might want additional information about authenticators before accepting them. The policy configuration is under control of the relying party, so it is possible to only accept authenticators for which additional data is available and meets the requirements.

A. References

A.1 Normative references

[FIDOMetadataStatement]
B. Hill; D. Baghdasaryan; J. Kemp. *FIDO Metadata Statements v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-metadata-statement-v2.0-ps-20170927.html

[JWS]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Signature (JWS)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7515

[JWT]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Token (JWT)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7519

[RFC4648]
S. Josefsson. *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*. October 2006. URL: http://www.ietf.org/rfc/rfc4648.txt

[RFC5280]
D. Cooper; S. Santesson; S. Farrell; S.Boeyen; R. Housley; W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. URL: http://www.ietf.org/rfc/rfc5280.txt

[WebIDL-ED]
Cameron McCormack. *Web IDL*. 13 November 2014. Editor's Draft. URL: http://heycam.github.io/webidl/'

A.2 Informative references

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAA Algorithm*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html

[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). November 2008. URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC2617]
J. Franks; P. Hallam-Baker; J. Hostetler; S. Lawrence; P. Leach; A. Luotonen; L. Stewart *HTTP Authentication: Basic and Digest Access Authentication*. June 1999. Draft Standard. URL:

https://tools.ietf.org/html/rfc2617
[RFC3986]
T. Berners-Lee; R. Fielding; L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986
[UAFProtocol]
R. Lindemann; D. Baghdasaryan; E. Tiffany; D. Balfanz; B. Hill; J. Hodges. *FIDO UAF Protocol Specification v1.0*. Proposed Standard. URL:https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html
[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

FIDO ECDAA Algorithm

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Jan Camenisch, IBM
Manu Drijvers, IBM
Alec Edgington, Trustonic
Anja Lehmann, IBM
Rainer Urian, Infineon

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The FIDO Basic Attestation scheme uses attestation "group" keys shared across a set of authenticators with identical characteristics in order to preserve privacy by avoiding the introduction of global correlation handles. If such an attestation key is extracted from one single authenticator, it is possible to create a "fake" authenticator using the same key and hence indistinguishable from the original authenticators by the relying party. Removing trust for registering new authenticators with the related key would affect the entire set of authenticators sharing the same "group" key. Depending on the number of authenticators, this risk might be unacceptable high.

This is especially relevant when the attestation key is primarily protected against malware attacks as opposed to targeted physical attacks.

An alternative approach to "group" keys is the use of individual keys combined with a Privacy-CA [TPMv1-2-Part1]. Translated to FIDO, this approach would require one Privacy-CA interaction for each Uauth key. This means relatively high load and high availability requirements for the Privacy-CA. Additionally the Privacy-CA aggregates sensitive information (i.e. knowing the relying parties the user interacts with). This might make the Privacy-CA an interesting attack target.

Another alternative is the Direct Anonymous Attestation [BriCamChe2004-DAA]. Direct Anonymous Attestation is a cryptographic scheme combining privacy with security. It uses the authenticator specific secret once to communicate with a single DAA Issuer and uses the resulting DAA credential in the DAA-Sign protocol with each relying party. The DAA scheme has been adopted by the Trusted Computing Group for TPM v1.2 [TPMv1-2-Part1].

In this document, we specify the use of an improved DAA scheme based on elliptic curves and bilinear pairings largely compatible with [CheLi2013-ECDAA] called ECDAA. This scheme provides significantly improved performance compared with the original DAA and basic building blocks for its implementation

are part of the TPMv2 specification [TPMv2-Part1].

Our improvements over [CheLi2013-ECDAA] mainly consist of security fixes (see [ANZ-2013] and [XYZF-2014]) when splitting the sign operation into two parts.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the [FIDO Alliance specifications index](https://www.fidoalliance.org/specifications/) at https://www.fidoalliance.org/specifications/.

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

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

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as`code`.

String literals are enclosed in “”, e.g. “ED256”.

In formulas we use “|” to denote byte wise concatenation operations.

$X = P^x$ denotes scalar multiplication (with scalar x) of a (elliptic) curve point P.

RAND(x) denotes generation of a random number between 0 and x-1.

RAND(G) denotes generation of a random number belonging to Group G.

Specific terminology used in this document is defined in [FIDOGlossary].

The type `BigNumber` denotes an arbitrary length integer value.

The type `ECPoint` denotes an elliptic curve point with its affine coordinates x and y.

The type `ECPoint2` denotes a point on the sextic twist of a BN elliptic curve over$F(q^2)$. The ECPoint2 has two affine coordinates each having two components of type BigNumber

1.1 Conformance

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

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

FIDO uses the concept of attestation to provide a cryptographic proof of the **authenticator** [FIDOGlossary] model to the relying party. When the authenticator is registered to the relying party (RP), it generates a new authentication key pair and includes the public key in the attestation message (also known as key registration data object, **KRD**). When using the ECDAA algorithm, the KRD object is signed using 3.5 ECDAA-Sign.

For privacy reasons, the authentication key pair is dedicated to one RP (to an application identifier **AppID** [FIDOGlossary] to be more specific). Consequently the attestation method needs to provide the same level of unlinkability. This is the reason why the FIDO ECDAA Algorithm doesn't use a basename (bsn) often found in other direct anonymous attestation algorithms, e.g. [BriCamChe2004-DAA] or [BFGSW-2011].

The authenticator encapsulates all user verification operations and cryptographic functions. An authenticator specific module (**ASM**) [FIDOGlossary] is used to provide a standardized communication interface for authenticators. The authenticator might be implemented in separate hardware or trusted execution environments. The ASM is assumed to run in the normal operating system (e.g. Android, Windows, ...).

2.1 Scope

This document describes the FIDO ECDAA attestation algorithm in detail.

2.2 Architecture Overview

ECDAA attestation defines global system parameters and issuer specific parameters. Both parameter sets need to be installed on the host, in the authenticator and in the FIDO Server. The ECDAA method consists of two steps:

- ECDAA-Join to be performed *before* the first FIDO Registration
  - n = GetNonceFromECDAAIssuer()
  - (Q, c1, s1) = EcdaaJoin1(X, Y, n)
  - (A, B, C, D, s2, c2) = EcdaaIssuerJoin(Q, c1, s1)
  - EcdaaJoin2(A, B, C, D, c2, s2) // store cre=(A, B, C, D)
- and the pair of ECDAA-Sign performed by the authenticator and ECDAA-Verify performed by the FIDO Server as part of the FIDO Registration.
  - Client: Attestation = (signature, KRD) = EcdaaSign(AppID)
  - Server: success=EcdaaVerify(signature, KRD, AppID)

The technical implementation details of the ECDAA-Join step are out-of-scope for FIDO. In this document we normatively specify the general algorithm to the extent required for interoperability and we outline examples of some possible implementations for this step.

The ECDAA-Sign and ECDAA-Verify steps and the encoding of the related ECDAA Signature are normatively specified in this document. The generation and encoding of the KRD object is defined in other FIDO specifications.

The algorithm and terminology are inspired by [BFGSW-2011]. The algorithm was modified in order to fix security weaknesses (e.g. as mentioned by [ANZ-2013] and [XYZF-2014]). Our algorithm proposes an improved task split for the sign operation while still being compatible to TPMv2 (without fixing the TPMv2 weaknesses in such case).

3. FIDO ECDAA Attestation

This section is normative.

3.1 Object Encodings

We need to convert `BigNumber` and `ECPoint` objects to byte strings using the following encoding functions:

3.1.1 Encoding `BigNumber` values as byte strings (BigNumberToB)

We use the I2OSP algorithm as defined in [RFC3447] for converting big numbers to byte arrays. The bytes from the big endian encoded (non-negative) number `n` will be copied right-aligned into the buffer area `b`. The unused bytes will be set to 0. Negative values will not occur due to the construction of the algorithms.

EXAMPLE 1: Converting BigNumber n to byte string b

```
b0 b1 b2 b3 b4 b5 b6 b7
 0  0 n0 n1 n2 n3 n4 n5
```

The algorithm implemented in Java looks like this:

EXAMPLE 2: Algorithm for converting BigNumber to byte strings

```
ByteArray BigNumberToB(
      BigNumber inVal, // IN: number to convert
      int size         // IN: size of the output.
     )
{
  ByteArray buffer = new ByteArray(size);
  int oversize = size - inVal.length;
  if (oversize < 0)
    return null;
  for (int i=overize; i > 0; i--)
     buffer[i] = 0;
  ByteCopy( inVal.bytes, &buffer[oversize], inVal.length);
 return buffer;
}
```

3.1.2 Encoding `ECPoint` values as byte strings (ECPointToB)

We use the ANSI X9.62 Point-to-Octet-String [ECDSA-ANSI] conversion using the expanded format, i.e. the format where the compression byte (i.e. 0x04 for expanded) is followed by the encoding of the affine x coordinate, followed by the encoding of the affine y coordinate.

EXAMPLE 3: Converting ECPoint P to byte string

```
(x, y) = ECPointGetAffineCoordinates(P)
len = G1.byteLength
byte string = 0x04 | BigIntegerToB(x,len) | BigIntegerToB(y,len)
```

3.1.3 Encoding `ECPoint2` values as byte strings (ECPoint2ToB)

The type `ECPoint2` denotes a point on the sextic twist of a BN elliptic curve over $F(q^2)$, see section 4.1 Supported Curves for ECDAA. Each `ECPoint2` is represented by a pair `(a, b)` of elements of $F(q)$.

The group zero element is always encoded (using the encoding rules as described below) as a an element having all components set to zero (i.e. cx.a=0, cx.b=0, cy.a=0, cy.b=0).

We always assume normalized (non-zero) ECPoint2 values (i.e. cz = 1) before encoding them. Non-zero values are encoded using the expanded format (i.e. 0x04 for expanded) followed by the cx followed by the cy value. This leads to the concatenation of 0x04 followed by the first element (`cx.a`) and second element (`cx.b`) of the pair of cx followed by the first element (`cy.a`) and second element (`cy.b`) of the pair of cy. All individual numbers are padded to the same length (i.e. the maximum byte length of all relevant 4 numbers).

EXAMPLE 4: Converting ECPoint2 P2 to byte string

```
(cx, cy) = ECPointGetAffineCoordinates(P2)
len = G2.byteLength
byte string = 0x04 | BigIntegerToB(cx.a,len) | BigIntegerToB(cx.b,len)
              | BigIntegerToB(cy.a,len) | BigIntegerToB(cy.b,len)
```

3.2 Global ECDAA System Parameters

1. Groups G_1, G_2 and G_T, of sufficiently large prime order p
2. Two generators P_1 and P_2, such that $G_1 = \langle P_1 \rangle$ and $G_2 = \langle P_2 \rangle$
3. A bilinear pairing $e : G_1 \times G_2 \rightarrow G_T$. We propose the use of "ate" pairing (see [BarNae-2006]). For example source code on this topic, see BNPairings.
4. Hash function H with $H : \{0,1\}^* \rightarrow Z_p$.
5. (G_1, P_1, p, H) are installed in all authenticators implementing FIDO ECDAA attestation.

Definition of G_1, G_2, G_T, Pairings and hash function H

See section 4.1 Supported Curves for ECDAA.

3.3 Issuer Specific ECDAA Parameters

Issuer Parameters parI

1. Randomly generated issuer private key $isk = (x, y)$ with $[x, y = RAND(p)]$.
2. ECDAA-Issuer public key (X, Y), with $X = P_2^x$ and $Y = P_2^y$.
3. A proof that the issuer key was correctly computed
    1. BigInteger $r_x = RAND(p)$
    2. BigInteger $r_y = RAND(p)$

3. ECPoint2 $U_x = P_2^{r_x}$
4. ECPoint2 $U_y = P_2^{r_y}$
5. BigInteger $c = H(U_x|U_y|P_2|X|Y)$
6. BigInteger $s_x = r_x + c \cdot x \pmod p$
7. BigInteger $s_y = r_y + c \cdot y \pmod p$

4. $ipk = X, Y, c, s_x, s_y$

Whenever a party uses ipk for the first time, it must first verify that it was correctly generated:

$$H(P_2^{s_x} \cdot X^{-c}|P_2^{s_y} \cdot Y^{-c}|P_2|X|Y) \stackrel{?}{=} c$$

NOTE

$$P_2^{s_x} \cdot X^{-c} = P_2^{r_x+cx} \cdot P_2^{-cx} = P_2^{r_x} = U_x$$

$$P_2^{s_y} \cdot Y^{-c} = P_2^{r_y+cy} \cdot P_2^{-cy} = P_2^{r_y} = U_y$$

The ECDAA-Issuer public key ipk must be dedicated to a single authenticator model.

We use the element c of ipk as an identifier for the ECDAA-Issuer public key (called **ECDAA-Issuer public key identifier**).

3.4 ECDAA-Join

NOTE

One ECDAA-Join operation is required once in the lifetime of an authenticator prior to the first registration of a credential.

In order to use ECDAA, the authenticator must first receive ECDAA credentials from an ECDAA-Issuer. This is done by the ECDAA-Join operation. This operation needs to be performed a single time (before the first credential registration can take place). After the ECDAA-Join, the authenticator will use the ECDAA-Sign operation as part of each FIDO Registration. The ECDAA-Issuer is not involved in this step. ECDAA plays no role in FIDO Authentication / Transaction Confirmation operations.

In order to use ECDAA, (at least) one ECDAA-Issuer is needed. The approach specified in this document easily scales to multiple ECDAA-Issuers, e.g. one per authenticator vendor. FIDO lets the authenticator vendor choose any ECDAA-Issuer (similar to his current freedom for selecting any PKI infrastructure/service provider to issuing attestation certificates required for FIDO Basic Attestation).

- All ECDAA-Join operations (of the related authenticators) are performed with one of the ECDAA-Issuer entities.
- Each ECDAA-Issuer has a set of public parameters, i.e. ECDAA public key material. The related Attestation Trust Anchor is contained in the metadata of each authenticator model identified by its AAGUID.

There are two different implementation options relevant for the authenticator vendors (the authenticator vendor can freely choose them):

1. In-Factory ECDAA-Join
2. Remote ECDAA-Join and

In the first case, physical proximity is used to locally establish the trust between the ECDAA-Issuer and the authenticator (e.g. using a key provisioning station in a production line). There is no requirement for the ECDAA-Issuer to operate an online web service.

In the second case, some credential is required to remotely establish the trust between the ECDAA-Issuer and the authenticator. As this operation is performed once and only with a single ECDAA-Issuer, privacy is preserved and an authenticator specific credential can and should be used.

Not all ECDAA authenticators might be able to add theirauthenticator model IDs (e.g. AAGUID) to the registration assertion (e.g. TPMs). In all cases, the ECDAA-Issuer will be able to derive the exact the authenticator model from either the credential or the physically proximiate authenticator. So the ECDAA-Issuer root key must be dedicated to a single authenticator model.

3.4.1 ECDAA-Join Algorithm

This section is normative.

> NOTE
>
> If this join is not in-factory, the value Q must be authenticated by theauthenticator. Upon receiving this value, the issuer must verify that this authenticator did not join before.

1. The authenticator asks the issuer for a nonce.
2. The issuer chooses a nonce BigInteger $n = RAND(p)$ and sends n via the ASM to the authenticator.
3. The authenticator chooses and stores the ECDAA private key BigInteger $sk = RAND(p)$
4. The authenticator computes its ECDAA public key ECPoint $Q = P_1^{sk}$
5. The authenticator proves knowledge of sk as follows
    1. BigInteger $r_1 = RAND(p)$
    2. ECPoint $U_1 = P_1^{r_1}$
    3. BigInteger $c_1 = H(U_1|P_1|Q|n)$
    4. BigInteger $s_1 = r_1 + c_1 \cdot sk$
6. The authenticator sends Q, c_1, s_1 via the ASM to the issuer
7. The issuer verifies that the authenticator is "authentic" and that Q was indeed generated by the authenticator. In the case of an in-factory Join, this might be trivial; in the case of a remote Join this typically requires the use of other cryptographic methods. Since ECDAA-Join is a one-time operation, unlinkability is not a concern for that.
8. The issuer verifies that $Q \in G_1$ and verifies $H(P_1^{s_1} \cdot Q^{-c_1}|P_1|Q|n) \stackrel{?}{=} c_1$ (check proof-of-possession of private key).

    > NOTE
    >
    > $P_1^{s_1} \cdot Q^{-c_1} = P_1^{r_1+c_1 sk} \cdot Q^{-c_1} = P_1^{r_1+c_1 sk} \cdot P_1^{-c_1 sk} = P_1^{r_1} = U_1$

9. The issuer creates credential (A, B, C, D) as follows
    1. BigInteger $l_J = RAND(p)$
    2. ECPoint $A = P_1^{l_J}$
    3. ECPoint $B = A^y$
    4. ECPoint $C = A^x \cdot Q^{xyl_J}$
    5. ECPoint $D = Q^{l_J y}$
10. The issuer proves that it computed this credential correctly:

1. BigInteger $r2 = RAND(p)$
2. ECPoint $U2 = P1^{r2}$
3. ECPoint $V2 = Q^{r2}$
4. BigInteger $c2 = H(U2|V2|P1|B|Q|D)$
5. BigInteger $s2 = r2 + c2 \cdot lJ \cdot y$

11. The issuer sends $A, B, C, D, c2, s2$ to the authenticator.
12. The authenticator checks that $A, B, C, D \in G1$ and $A \neq 1_{G1}$
13. The authenticator checks $H(P1^{s2} \cdot B^{-c2}|Q^{s2} \cdot D^{-c2}|P1|B|Q|D) \stackrel{?}{=} c2$

> NOTE
>
> $P1^{s2} \cdot B^{-c2} = P1^{r2} \cdot P1^{c2 \cdot lJ \cdot y} \cdot B^{-c2} = U2 \cdot B^{c2} \cdot B^{-c2} = U2$
>
> $Q^{s2} \cdot D^{-c2} = Q^{r2} \cdot Q^{c2 \cdot lJ \cdot y} \cdot D^{-c2} = V2 \cdot D^{c2} \cdot D^{-c2} = V2$

14. The authenticator checks $e(A, Y) \stackrel{?}{=} e(B, P2)$

> NOTE
>
> $e(A, Y) = e(P1^{lJ}, P2^{y}); e(B, P2) = e(A^{y}, P2) = e(P1^{ylJ}, P2)$

15. and the authenticator checks $e(C, P2) \stackrel{?}{=} e(A \cdot D, X)$

> NOTE
>
> $e(C, P2) = e(A^{x} \cdot Q^{xylJ}, P2); e(A \cdot D, X) = e(A \cdot Q^{ylJ}, P2^{x})$

16. The authenticator stores credential A, B, C, D

3.4.2 ECDAA-Join Split between Authenticator and ASM

This section is non-normative.

> NOTE
>
> If this join is not in-factory, the value Q must be authenticated by theauthenticator. Upon receiving this value, the issuer must verify that this authenticator did not join before.

1. The ASM asks the issuer for a nonce.
2. The issuer chooses a nonce BigInteger $n = RAND(p)$ and sends n to the ASM.
3. The ASM forwards n to the authenticator
4. The authenticator chooses and stores the private key BigInteger $sk = RAND(p)$
5. The authenticator computes its ECDAA public key ECPoint $Q = P1^{sk}$

6. The authenticator proves knowledge of sk as follows
    1. BigInteger $r1 = RAND(p)$
    2. ECPoint $U1 = P1^{r1}$
    3. BigInteger $c1 = H(U1|P1|Q|n)$
    4. BigInteger $s1 = r1 + c1 \cdot sk$
7. The authenticator sends $Q, c1, s1$ to the ASM, who forwards it to the issuer.
8. The issuer verifies that the authenticator is "authentic" and that Q was indeed generated by the authenticator. In the case of an in-factory Join, this might be trivial; in the case of a remote Join this typically requires the use of other cryptographic methods. Since ECDAA-Join is a one-time operation, unlinkability is not a concern for that.
9. The issuer verifies that $Q \in G1$ and verifies $H(P1^{s1} \cdot Q^{-c1}|P1|Q|n) \stackrel{?}{=} c1$.
10. The issuer creates credential (A, B, C, D) as follows
    1. BigInteger $lJ = RAND(p)$
    2. ECPoint $A = P1^{lJ}$
    3. ECPoint $B = A^{y}$
    4. ECPoint $C = A^{x} \cdot Q^{xylJ}$
    5. ECPoint $D = Q^{lJy}$
11. The issuer proves that it computed this credential correctly:
    1. BigInteger $r2 = RAND(p)$
    2. ECPoint $U2 = P1^{r2}$
    3. ECPoint $V2 = Q^{r2}$
    4. BigInteger $c2 = H(U2|V2|P1|B|Q|D)$
    5. BigInteger $s2 = r2 + c2 \cdot lJ \cdot y$
12. The issuer sends $A, B, C, D, c2, s2$ to the ASM. The issuer authenticates $B, D, c2, s2$ such that the authenticator can verify they were created by the issuer.
13. The ASM checks that $A, B, C, D \in G1$ and $A \neq 1_{G1}$
14. The ASM checks $H(P1^{s2} \cdot B^{-c2}|Q^{s2} \cdot D^{-c2}|P1|B|Q|D) \stackrel{?}{=} c2$
15. The ASM checks $e(A, Y) \stackrel{?}{=} e(B, P2)$
16. and the ASM checks that $e(C, P2) \stackrel{?}{=} e(A \cdot D, X)$
17. The ASM stores A, B, C, D and sends $B, D, c2, s2$ to the authenticator
18. The authenticator checks $B, D \in G1$ and $B \neq 1_{G1}$, and verifies that $B, D, c2, s2$ were sent by the issuer.
19. The authenticator checks $H(P1^{s2} \cdot B^{-c2}|Q^{s2} \cdot D^{-c2}|P1|B|Q|D) \stackrel{?}{=} c2$
20. The authenticator stores B, D and ignores further join requests.

> NOTE
>
> These values belong to the ECDAA secret key sk. They should persist even in the case of a factory reset.

3.4.3 ECDAA-Join Split between TPM and ASM

This section is non-normative.

NOTE

The Endorsement key credential (EK-C) and TPM2_ActivateCredentials are used for supporting the remote Join.

This description is based on the principles described in [TPMv2-Part1] section 24 and [Arthur-Challener-2015], page 109 ("Activating a Credential").

1. The ASM asks the ECDAA Issuer for a nonce.
2. The ECDAA Issue chooses a nonce BigInteger $n = RAND(p)$ and sends n to the ASM.
3. The ASM
    1. instructs the TPM to create a restricted key by calling TPM2_Create, giving the public key template `TPMT_PUBLIC` [TPMv2-Part2] (including the public key Q in field `unique`) to the ASM.
    2. retrieves TPM Endorsement Key Certificate (EK-C) from the TPM
    3. calls TPM2_Commit(keyhandle, P1, s2, y2) where keyhandle is the handle of the restricted key generated before (see above), P1 is set to P_1, and s2 and y2 are left empty. This call returns K, L, E, and ctr; where K and L will be empty.
    4. computes BigInteger $c_1 = H(E|P_1|Q|n)$
    5. call TPM2_Sign(c_1, ctr), returning s_1.
    6. sends EK-C, `TPMT_PUBLIC` (including Q in field `unique`), c_1, s_1 to the ECDAA Issuer.
4. The ECDAA Issuer
    1. verifies EK-C and its certificate chain. As a result the ECDAA Issuer knows the TPM model related to EK-C.
    2. verifies that this EK-C was not used in a (successful) Join before
    3. Verifies that the `objectAttributes` in `TPMT_PUBLIC` [TPMv2-Part2] matches the following flags: `fixedTPM` = 1; `fixedParent` = 1; `sensitiveDataOrigin` = 1; `encryptedDuplication` = 0; `restricted` = 1; `decrypt` = 0; `sign` = 1.
    4. examines the public key Q, i.e. it verifies that $Q \in G_1$
    5. checks $H(P_1^{s_1} \cdot Q^{-c_1}|P_1|Q|n) \stackrel{?}{=} c_1$
    6. generates the ECDAA credential (A, B, C, D) as follows
        1. BigInteger $l_J = RAND(p)$
        2. ECPoint $A = P_1^{l_J}$
        3. ECPoint $B = A^y$
        4. ECPoint $C = A^x \cdot Q^{xyl_J}$
        5. ECPoint $D = Q^{l_J y}$
    7. proves that it computed this credential correctly:
        1. BigInteger $r_2 = RAND(p)$
        2. ECPoint $U_2 = P_1^{r_2}$
        3. ECPoint $V_2 = Q^{r_2}$
        4. BigInteger $c_2 = H(U_2|V_2|P_1|B|Q|D)$

5. BigInteger $s2 = r2 + c2 \cdot l^J \cdot y$

8. generates a *secret* (derived from a *seed*) and wraps the credential A, B, C, D using that *secret*.
9. encrypts the *seed* using the public key included in EK-C.
10. uses *seed* and *name* in KDFa (see [TPMv2-Part2] section 24.4) to derive HMAC and *symmetric encryption key*. Wrap the *secret* in *symmetric encryption key* and protect it with the *HMAC key*.

> NOTE
>
> The parameter *name* in KDFa is derived from `TPMT_PUBLIC`, see [TPMv2-Part1], section 16.

11. sends the credential proof $c2, s2$ and the wrapped object including the credential from previous step to the ASM.

5. The ASM instructs the TPM (by calling TPM2_ActivateCredential) to
   1. decrypt the *seed* using the TPM Endorsement key
   2. compute the *name* (for the ECDAA attestation key)
   3. use the *seed* in KDFa (with *name*) to derive the *HMAC key* and the *symmetric encryption key*.
   4. use the *symmetric encryption key* to unwrap the *secret*.
6. The ASM
   1. unwraps the credential A, B, C, D using the *secret* received from the TPM.
   2. checks that $A, B, C, D \in G1$ and $A \neq 1_{G1}$
   3. checks $H(P1^{s2} \cdot B^{-c2} | Q^{s2} \cdot D^{-c2} | P1 | B | Q | D) \stackrel{?}{=} c2$
   4. checks $e(A, Y) \stackrel{?}{=} e(B, P2)$ and $e(C, P2) \stackrel{?}{=} e(A \cdot D, X)$
   5. stores A, B, C, D

3.5 ECDAA-Sign

> NOTE
>
> One ECDAA-Sign operation is required for the client-side environment whenever a new credential is being registered at a relying party.

3.5.1 ECDAA-Sign Algorithm

This section is normative.

(signature, KRD) = EcdaaSign(String AppID)

Parameters

- p: System parameter prime order of group G1 (global constant)
- AppID: FIDO AppID (i.e. https-URL of TrustedFacets object)

Algorithm outline

1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here
2. BigNumber $l = RAND(p)$
3. ECPoint $R = A^l$;
4. ECPoint $S = B^l$;

5. ECPoint $T = C^l$;
6. ECPoint $W = D^l$;
7. BigInteger $r = RAND(p)$
8. ECPoint $U = S^r$
9. BigInteger $c = H(U|S|W|AppID|H(KRD))$
10. BigInteger $s = r + c \cdot sk \pmod p$
11. signature = (c, s, R, S, T, W)
12. return (signature, KRD)

3.5.2 ECDAA-Sign Split between Authenticator and ASM

This section is non-normative.

> NOTE
>
> This split requires both the authenticator and ASM to be honest to achieve anonymity. Only the authenticator must be trusted for unforgeability. The communication between ASM and authenticator must be secure.

Algorithm outline

1. The ASM randomizes the credential
   1. BigNumber $l = RAND(p)$
   2. ECPoint $R = A^l$;
   3. ECPoint $S = B^l$;
   4. ECPoint $T = C^l$;
   5. ECPoint $W = D^l$;
2. The ASM sends $l, AppID$ to the authenticator
3. The authenticator performs the following tasks
   1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here
   2. ECPoint $S' = B^l$
   3. ECPoint $W' = D^l$
   4. BigInteger $r = RAND(p)$
   5. ECPoint $U = S^r$
   6. BigInteger $c = H(U|S'|W'|AppID|H(KRD))$
   7. BigInteger $s = r + c \cdot sk \pmod p$
   8. Send c, s, KRD to the ASM
4. The ASM sets signature = (c, s, R, S, T, W) and outputs (signature,KRD)

3.5.3 ECDAA-Sign Split between TPM and ASM

This section is non-normative.

> NOTE
>
> This algorithm is for the special case of a TPMv2 as authenticator. This case requires both the TPM

and ASM to be honest for anonymity and unforgeability (see [XYZF-2014]).

Algorithm outline

1. The ASM randomizes the credential
   1. BigNumber $l = RAND(p)$
   2. ECPoint $R = A^l$;
   3. ECPoint $S = B^l$;
   4. ECPoint $T = C^l$;
   5. ECPoint $W = D^l$;
2. The ASM calls TPM2_Commit() with $P1$ set to S and $s2, y2$ empty buffers. The ASM receives the result values $K, L, E = S^r$ and ctr. K and L are empty since $s2, y2$ are empty buffers.
3. The ASM calls TPM2_Create to generate the new authentication key pair.
4. The ASM calls TPM2_Certify() on the newly created key with ctr from the TPM2_Commit and $E, S, W, AppID$ as qualifying data ($E = S^r$ is returned by step 2). The ASM receives signature c, s and attestation block KRD (i.e. TPMS_ATTEST structure in this case).
5. The ASM sets signature = (c, s, R, S, T, W) and outputs (signature,KRD)

3.6 ECDAA-Verify Operation

This section is normative.

NOTE

One ECDAA-Verify operation is required for the FIDO Server as part of each FIDO Registration.

boolean EcdaaVerify(signature, AppID, KRD, ModelName)

Parameters

- p: System parameter prime order of group G_1 (global constant)
- P_2: System parameter generator of group G_2 (global constant)
- signature: (c, s, R, S, T, W)
- AppID: FIDO AppID
- KRD: Attestation Data object as defined in other specifications.
- ModelName: the claimed FIDO authenticator model (i.e. either AAID or AAGUID)

Algorithm outline

1. Based on the claimed ModelName, look up X, Y from trusted source
2. Check that $R, S, T, W \in G_1$, $R \neq 1_{G_1}$, and $S \neq 1_{G_1}$.
3. $H(S^s \cdot W^{-c} | S | W | AppID | H(KRD)) \stackrel{?}{=} c$; fail if not equal

   NOTE

   $B = A^y = P_1^{ly}$

   $D = Q^{lJy} = P_1^{sklJy} = B^{sk}$

$S = B^l$ and $W = D^l$

$U = S^r$

$S^s \cdot W^{-c} = S^{r+csk} \cdot W^{-c} = U \cdot S^{csk} \cdot W^{-c}$
$= U \cdot B^{lcsk} \cdot D^{-lc} = U \cdot B^{lcsk} \cdot B^{-lcsk} = U$

4. $e(R, Y) \stackrel{?}{=} e(S, P_2)$; fail if not equal

NOTE

$e(R, Y) = e(A^l, P_2^y); e(S, P_2) = e(B^l, P_2) = e(A^{ly}, P_2)$

5. $e(T, P_2) \stackrel{?}{=} e(R \cdot W, X)$; fail if not equal

NOTE

$e(T, P_2) = e(C^l, P_2) = e(A^{xl} \cdot Q^{xlylJ}, P_2); e(A^l \cdot D^l, X) = e(A^l \cdot Q^{lylJ}, P_2^x)$

6. for (all sk' on RogueList) do if $W \stackrel{?}{=} S^{sk'}$ fail;
7. // perform all other processing steps for new credential registration

NOTE
In the case of a TPMv2, i.e. KRD is a `TPMS_ATTEST` object. In this case the verifier must check whether the `TPMS_ATTEST` object starts with `TPM_GENERATED` magic number and whether its field `objectAttributes` contains the flag `fixedTPM`=1 (indicating that the key was generated by the TPM).

8. return true;

4. FIDO ECDAA Object Formats and Algorithm Details

This section is normative.

4.1 Supported Curves for ECDAA

Definition of G1

G1 is an elliptic curve group E : $y^2 = x^3 + ax + b$ over $F(q)$ with $a = 0$.

Definition of G2

G2 is the p-torsion subgroup of $E'(F_{q^2})$ where E' is a sextic twist of E. With E' : $y'^2 = x'^3 + b'$.

An element of $F(q^2)$ is represented by a pair (a,b) where a + bX is an element of $F(q)[X]/ < X^2 + 1 >$. We use angle brackets $< Y >$ to signify the ideal generated by the enclosed value.

NOTE

In the literature the pair (a,b) is sometimes also written as a complex number $a + b * i$.

Definition of GT

GT is an order-p subgroup of Fq^{12}.

Pairings

We propose the use of Ate pairings as they are efficient (more efficient than Tate pairings) on Barreto-Naehrig curves [DevScoDah2007].

Supported BN curves

We use pairing-friendly Barreto-Naehrig [BarNae-2006] [ISO15946-5] elliptic curves. The curves `TPM_ECC_BN_P256` and `TPM_ECC_BN_P638` curves are defined in [TPMv2-Part4].

BN curves have a Modulus $q = 36 \cdot u^4 + 36 \cdot u^3 + 24 \cdot u^2 + 6 \cdot u + 1$ [ISO15946-5] and a related order of the group $p = 36 \cdot u^4 + 36 \cdot u^3 + 18 \cdot u^2 + 6 \cdot u + 1$ [ISO15946-5].

- `TPM_ECC_BN_P256` is a curve of form E(F(q)), where q is the field modulus [TPMv2-Part4] [BarNae-2006]. This curve is identical to the P256 curve defined in [ISO15946-5] section C.3.5.
  - The values have been generated using u=-7 530 851 732 716 300 289.
  - Modulus q = 115 792 089 237 314 936 872 688 561 244 471 742 058 375 878 355 761 205 198 700 409 522 629 664 518 163
  - Group order p = 115 792 089 237 314 936 872 688 561 244 471 742 058 035 595 988 840 268 584 488 757 999 429 535 617 037
  - p and q have length of 256 bit each.
  - b = 3
  - $P1$_256 = (x=1, y=2)
  - b' = (a=3, b=3)
  - $P2$_256 = (x,y), with
    - $P2$_256.x = (a=114 909 019 869 825 495 805 094 438 766 505 779 201 460 871 441 403 689 227 802 685 522 624 680 861 435, b=35 574 363 727 580 634 541 930 638 464 681 913 209 705 880 605 623 913 174 726 536 241 706 071 648 811)
    - $P2$_256.y = (a=65 076 021 719 150 302 283 757 931 701 622 350 436 355 986 716 727 896 397 520 706 509 932 529 649 684, b=113 380 538 053 789 372 416 298 017 450 764 517 685 681 349 483 061 506 360 354 665 554 452 649 749 368)
- `TPM_ECC_BN_P638` [TPMv2-Part4] uses
  - The values have been generated using u=365 375 408 992 443 362 629 982 744 420 548 242 302 862 098 433
  - Modulus q = 641 593 209 463 000 238 284 923 228 689 168 801 117 629 789 043 238 356 871 360 716 989 515 584 497 239 494 051 781 991 794 253 619 096 481 315 470 262 367 432 019 698 642 631 650 152 075 067 922 231 951 354 925 301 839 708 740 457 083 469 793 717 125 223
  - The related order of the group is p = 641 593 209 463 000 238 284 923 228 689 168 801 117 629 789 043 238 356 871 360 716 989 515 584 497 239 494 051 781 991 794 252 818 101 344 337 098 690 003 906 272 221 387 599 391 201 666 378 807 960 583 525 233 832 645 565 592 955 122 034 352 630 792 289
  - p and q have length of 638 bit each.
  - b = 257
  - $P1$_638 = (x=641 593 209 463 000 238 284 923 228 689 168 801 117 629 789 043 238 356 871 360 716 989 515 584 497 239 494 051 781 991 794 253 619 096 481 315 470 262 367 432 019 698 642 631 650 152 075 067 922 231 951 354 925 301 839 708 740 457 083 469 793 717 125 222, y=16)
  - b' = (a=771, b=1542)

- $P2$_638 = (x, y), with
  - $P2$_638.x = (a=192 492 098 325 059 629 927 844 609 092 536 807 849 769 208 589 403 233 289 748 474 758 010 838 876 457 636 072 173 883 771 602 089 605 233 264 992 910 618 494 201 909 695 576 234 119 413 319 303 931 909 848 663 554 062 144 113 485 982 076 866 968 711 247, b=166 614 418 891 499 184 781 285 132 766 747 495 170 152 701 259 472 324 679 873 541 478 330 301 406 623 174 002 502 345 930 325 474 988 134 317 071 869 554 535 111 092 924 719 466 650 228 182 095 841 246 668 361 451 788 368 418 036 777 197 454 618 413 255)
  - $P2$_638.y = (a=622 964 952 935 200 827 531 506 751 874 167 806 262 407 152 244 280 323 674 626 687 789 202 660 794 092 633 841 098 984 322 671 973 226 667 873 503 889 270 602 870 064 426 165 592 237 410 681 318 519 893 784 898 821 343 051 339 820 566 224 981 344 169 470, b=514 285 963 827 225 043 076 463 721 426 569 583 576 029 220 880 138 564 906 219 230 942 887 639 456 599 654 554 743 732 087 558 187 149 207 036 952 474 092 411 405 629 612 957 921 369 286 372 038 525 830 610 755 207 588 843 864 366 759 521 090 861 911 494)

- `ECC_BN_DSD_P256` [DevScoDah2007] section 3 uses
  - The values have been generated using u=6 917 529 027 641 089 837
  - Modulus q = 82434016654300679721217353503190038836571781811386228921167322412819029493183
  - The related order of the group is p = 82434016654300679721217353503190038836284668564296686430114510052556401373769
  - p and q have length of 256 bit each.
  - b = 3
  - $P1$_DSD_P256 = (1, 2)
  - b' = (a=3, b=6)
  - $P2$_DSD_P256 = (x, y), with
    - $P2$_DSD_P256.x = (a=73 481 346 555 305 118 071 940 904 527 347 990 526 214 212 698 180 576 973 201 374 397 013 567 073 039, b=28 955 468 426 222 256 383 171 634 927 293 329 392 145 263 879 318 611 908 127 165 887 947 997 417 463)
    - $P2$_DSD_P256.y = (a=3 632 491 054 685 712 358 616 318 558 909 408 435 559 591 759 282 597 787 781 393 534 962 445 630 353, b=60 960 585 579 560 783 681 258 978 162 498 088 639 544 584 959 644 221 094 447 372 720 880 177 666 763)

- `ECC_BN_ISOP512` [ISO15946-5] section C.3.7 uses
  - The values have been generated using u=138 919 694 570 470 098 040 331 481 282 401 523 727
  - Modulus q = 13 407 807 929 942 597 099 574 024 998 205 830 437 246 153 344 875 111 580 494 527 427 714 590 099 881 795 845 981 157 516 604 994 291 639 750 834 285 779 043 186 149 750 164 319 950 153 126 044 364 566 323
  - The related order of the group is p = 13 407 807 929 942 597 099 574 024 998 205 830 437 246 153 344 875 111 580 494 527 427 714 590 099 881 680 053 891 920 200 409 570 720 654 742 146 445 677 939 306 408 461 754 626 647 833 262 056 300 743 149
  - p and q have length of 512 bit each.
  - b = 3
  - $P1$_ISO_P512 = (x=1,y=2)
  - b' = (a=3, b=3)
  - $P2$_ISO_P512 = (x, y), with
    - $P2$_ISO_P512.x = (a=3 094 648 157 539 090 131 026 477 120 117 259 896 222 920 557 994 037 039 545 437 079 729 804 516 315 481 514 566 156 984 245 473 190 248 967 907 724 153 072 490 467 902 779 495 072 074 156 718 085 785 269, b=3 776 690 234 788 102 103 015 760 376 468 067 863 580 475 949 014 286 077 855 600 384 033 870 546 339 773 119 295 555 161 718 985 244 561 452 474 412 673 836 012 873 126 926 524 076 966 265 127 900 471 529)
    - $P2$_ISO_P512.y = (a=7 593 872 605 334 070 150 001 723 245 210 278 735 800 573

263 881 411 015 285 406 372 548 542 328 752 430 917 597 485 450 360 707 892 769 159 214 115 916 255 816 324 924 295 339 525 686 777 569 132 644 242, b=9 131 995 053 349 122 285 871 305 684 665 648 028 094 505 015 281 268 488 257 987 110 193 875 868 585 868 792 041 571 666 587 093 146 239 570 057 934 816 183 220 992 460 187 617 700 670 514 736 173 834 408)

> NOTE
>
> Spaces are used inside numbers to improve readability.

Hash Algorithms

Depending on the curve, we use `H(x) = SHA256(x) mod p` or `H(x) = SHA512(x) mod p` as hash algorithm H:$\{0,1\}^* \rightarrow Zp$.

The argument of the hash function must always be converted to a byte string using the appropriate encoding function specific in section 3.1 Object Encodings, e.g. according to section 3.1.3 Encoding ECPoint2 values as byte strings (ECPoint2ToB) in the case of `ECPoint2` points.

> NOTE
>
> We don't use IEEE P1363.3 section 6.1.1 IHF1-SHA with security parameter t (e.g. t=128 or 256) as it is more complex and not supported by TPMv2.

4.2 ECDAA Algorithm Names

We define the following JWS-style algorithm names (see [RFC7515]):

ED256
: `TPM_ECC_BN_P256` curve, using SHA256 as hash algorithm H.

ED256-2
: `ECC_BN_DSD_P256` curve, using SHA256 as hash algorithm H.

ED512
: `ECC_BN_ISOP512` curve, using SHA512 as hash algorithm H.

ED638
: `TPM_ECC_BN_P638` curve, using SHA512 as hash algorithm H.

4.3 ecdaaSignature object

The fields c and s both have length N. The fields R, S, T, W have equal length (2*N+1 each).

In the case of BN_P256 curve (with key length N=32 bytes), the fields R, S, T, W have length 2*32+1=65 bytes. The fields c and s have length N=32 each.

The ecdaaSignature object is a binary object generated as the concatenation of the binary fields in the order described below (total length of 324 bytes for 256bit curves):

Value	Length (in Bytes)	Description
UINT8[] ECDAA_Signature_c	N	The c value, c=H(U I S I W IKRD I AppID) as returned by AuthnrEcdaaSign encoded as byte string according to BigNumberToB. Where • $U = S^r$, with $r = RAND(p)$ computed by the signer. • KRD is the the entire to-be-signed object (e.g. TAG_UAFV1_KRD in the case of FIDO UAF). • $S = B^l$, with $l = RAND(p)$ computed by the signer and $B = A^y$ computed in the ECDAA-Join

Value	Length (in Bytes)	Description
UINT8[] ECDAA_Signature_s	N	The s value, s=r + c * sk (mo[illegible]rned by AuthnrEcdaaSign encoded as byte string according to BigNumberToB. Where • r = RAND(p), computed by the signer at FIDO registration (see 3.5.2 ECDAA-Sign Split between Authenticator and ASM) • p is the group order of G1 • sk: is the authenticator's attestation secret key, see above
UINT8[] ECDAA_Signature_R	2*N+1	$R = A^l$; computed by the ASM or the authenticator at FIDO registration; encoded as byte string according to ECPointToB. Where • l = RAND(p), i.e. random number 0≤l≤p. Computed by the ASM or the authenticator at FIDO registration. • And where $R = A^l$ denotes the scalar multiplication (of scalar l) of a curve point A. • Where A has been provided by the ECDAA-Issuer as part of ECDAA-Join: $A = P_1^{lJ}$, see 3.4.1 ECDAA-Join Algorithm. • Where P_1 and p are system values, injected into the authenticator and lJ is a random number computed by the ECDAA-Issuer on Join.
UINT8[] ECDAA_Signature_S	2*N+1	$S = B^l$; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where B has been provided by the ECDAA-Issuer on Join: $B = A^y$, see 3.4.1 ECDAA-Join Algorithm.
UINT8[] ECDAA_Signature_T	2*N+1	$T = C^l$; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where • $C = A^x \cdot Q^{xylJ}$, provided by the ECDAA-Issuer on Join • $lJ = RAND(p)$ computed by the ECDAA-Issuer at Join (see 3.4.1 ECDAA-Join Algorithm) • x and y are components of the ECDAA-Issuer private key, iskk=(x,y). • Q is the authenticator public key
UINT8[] ECDAA_Signature_W	2*N+1	$W = D^l$; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where $D = Q^{lJy}$ is computed by the ECDAA-Issuer at Join (see 3.4.1 ECDAA-Join Algorithm).

5. Considerations

This section is non-normative.

A detailed security analysis of this algorithm can be found in [FIDO-DAA-Security-Proof].

5.1 Algorithms and Key Sizes

The proposed algorithms and key sizes are chosen such that compatibility to TPMv2 is possible.

5.2 Indicating the Authenticator Model

Some authenticators (e.g. TPMv2) do not have the ability to include their model (i.e. vendor ID and model name) in attested messages (i.e. the to-be-signed part of the registration assertion). The TPM's endorsement key certificate typically contains that information directly or at least it allows the model to be derived from the endorsement key certificate.

In FIDO, the relying party expects the ability to cryptographically verify the authenticator model.

We require the ECDAA-Issuers public key (ipk=(X,Y,c,sx,sy)) to be dedicated to one single authenticator model (e.g. as identified by AAID or AAGUID).

5.3 Revocation

If the private ECDAA attestation key sk of an authenticator has been leaked, it can be revoked by adding its value to a RogueList.

The ECDAA-Verifier (i.e. FIDO Server) check for such revocations. See section 3.6 ECDAA-Verify Operation.

The ECDAA-Issuer is expected to check revocation by other means:

1. if ECDAA-Join is done in-factory, it is assumed that produced devices are known to be uncompomised (at time of production).
2. if a remote ECDAA-Join is performed, the (remote) ECDAA-Issuer already must use a different method to remotely authenticate the authenticator (e.g. using some endorsement key). We expect the ECDAA-Issuer to perform a revocation check based on that information. This is even more flexible as it does not require access to the authenticator ECDAA private key sk.

5.4 Pairing Algorithm

The pairing algorithm e needs to be used by the ASM as part of the Join process and by the verifier (i.e. FIDO relying party) as part of the verification (i.e. FIDO registration) process.

The result of such a pairing operation is only compared to the result of another pairing operation computed by the same entity. As a consequence, it doesn't matter whether the ASM and the verifier use the exact same pairings or not (as long as they both use valid pairings).

5.5 Performance

For performance reasons the calculation of Sig2=(R, S, T, W) may be performed by the ASM running on the FIDO user device (as opposed to inside the authenticator). See section 3.5.2 ECDAA-Sign Split between Authenticator and ASM.

The cryptographic computations to be performed inside the authenticator are limited to G1. The ECDAA-Issuer has to perform two G2 point multiplications for computing the public key. The Verifier (i.e. FIDO relying party) has to perform G1 operations and two pairing operations.

5.6 Binary Concatentation

We use a simple byte-wise concatenation function for the different parameters, i.e. H(a,b) = H(a | b).

This approach is as secure as the underlying hash algorithm since the authenticator controls the length of the (fixed-length) values (e.g. U, S, W). The AppID is provided externally and has unverified structure and length. However, it is only followed by a fixed length entry - the (system defined) hash of KRD. As a consequence, no parts of the AppID would ever be confused with the fixed length value.

5.7 IANA Considerations

This specification registers the algorithm names "ED256", "ED512", and "ED638" defined in section 4. FIDO ECDAA Object Formats and Algorithm Details with the IANA JSON Web Algorithms registry as defined in section "Cryptographic Algorithms for Digital Signatures and MACs" in [RFC7518].

Algorithm Name	"ED256"
Algorithm Description	FIDO ECDAA algorithm based on TPM_ECC_BN_P256 [TPMv2-Part4] curve using SHA256 hash algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

Algorithm Name	"ED512"
Algorithm Description	ECDAA algorithm based on ECC_BN_ISOP512 [ISO15946-5] curve using SHA512 algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

Algorithm Name	"ED638"
Algorithm Description	ECDAA algorithm based on TPM_ECC_BN_P638 [TPMv2-Part4] curve using SHA512 algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

A. References

A.1 Normative references

[ECDSA-ANSI]
Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62-2005. November 2005. URL: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005

[RFC2119]

S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3447]
J. Jonsson; B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. February 2003. Informational. URL: https://tools.ietf.org/html/rfc3447

[TPMv2-Part4]
Trusted Platform Module Library, Part 4: Supporting Routines. URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C6CABBC-1A4B-B294-D0DA8CE1B452CAB4/TPM%20Rev%202.0%20Part%204%20-%20Supporting%20Routines%2001.16-code.pdf

A.2 Informative references

[ANZ-2013]
Tolga Acar; Lan Nguyen; Greg Zaverucha. *A TPM Diffie-Hellman Oracle*. October 18, 2013. URL: http://eprint.iacr.org/2013/667.pdf

[Arthur-Challener-2015]
Will Arthur; David Challener; Kenneth Goldman. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. 2014. URL: http://www.apress.com/9781430265832

[BFGSW-2011]
D. Bernhard; G. Fuchsbauer; E. Ghadafi; N. P. Smart; B. Warinschi. *Anonymous Attestation with User-controlled Linkability*. 2011. URL: http://eprint.iacr.org/2011/658.pdf

[BarNae-2006]
Paulo S. L. M. Barreto; Michael Naehrig. *Pairing-Friendly Elliptic Curves of Prime Order*. 2006. URL: http://research.microsoft.com/pubs/118425/pfcpo.pdf

[BriCamChe2004-DAA]
Ernie Brickell; Jan Camenisch; Liqun Chen. *Direct Anonymous Attestation*. 2004. URL: http://eprint.iacr.org/2004/205.pdf

[CheLi2013-ECDAA]
Liqun Chen; Jiangtao Li. *Flexible and Scalable Digital Signatures in TPM 2.0*. 2013. URL: http://dx.doi.org/10.1145/2508859.2516729

[DevScoDah2007]
Augusto Jun Devegili; Michael Scott; Ricardo Dahab. *Implementing Cryptographic Pairings over Barreto-Naehrig Curves*. 2007. URL: https://eprint.iacr.org/2007/390.pdf

[FIDO-DAA-Security-Proof]
Jan Camenisch; Manu Drijvers; Anja Lehmann. *Universally Composable Direct Anonymous Attestation*. 2015. URL: https://eprint.iacr.org/2015/1246

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAA Algorithm*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html

[ISO15946-5]
ISO/IEC 15946-5 Information Technology - Security Techniques - Cryptographic techniques based on elliptic curves - Part 5: Elliptic curve generation. URL: https://webstore.iec.ch/publication/10468

[RFC7515]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Signature (JWS) (RFC7515)*. May 2015. URL: http://www.ietf.org/rfc/rfc7515.txt

[RFC7518]
M. Jones. *JSON Web Algorithms (JWA)*. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7518

[TPMv1-2-Part1]
TPM 1.2 Part 1: Design Principles. URL: http://www.trustedcomputinggroup.org/files/static_page_files/72C26AB5-1A4B-B294-D002BC0B8C062FF6/TPM%20Main-Part%201%20Design%20Principles_v1.2_rev116_01032011.pdf

[TPMv2-Part1]
Trusted Platform Module Library, Part 1: Architecture. URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C56AE3E-1A4B-B294-D0F43097156A55D8/TPM%20Rev%202.0%20Part%201%20-%20Architecture%2001.16.pdf

[TPMv2-Part2]
Trusted Platform Module Library, Part 2: Structures. URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C583202-1A4B-B294-D0469592DB10A6CD/TPM%20Rev%202.0%20Part%202%20-%20Structures%2001.16.pdf

[XYZF-2014]
Li Xi; Kang Yang; Zhenfeng Zhang; Dengguo Feng. *DAA-Related APIs in TPM 2.0 Revisited, in T. Holz and S. Ioannidis (Eds.)*. 2014. URL:

FIDO Security Reference

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-security-ref-v2.0-ps-20170927.html
Previous version:
https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-security-ref-v2.0-rd-20161004.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal, Inc.
Dr. Joshua E. Hill, InfoGard Laboratories
Douglas Biggs, InfoGard Laboratories

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This document analyzes the security properties of the FIDO UAF and U2F families of protocols.

Status of This Document

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

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

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

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as`code`.

String literals are enclosed in “”, e.g. “UAF-TLV”.

In formulas we use “|” to denote byte wise concatenation operations.

UAF specific terminology used in this document is defined in [FIDOGlossary].

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Introduction

This document analyzes the security properties of the FIDO UAF and U2F families of protocols. Although a brief architectural summary is provided below, readers should familiarize themselves with the the FIDO Glossary of Terms [FIDOGlossary] for definitions of terms used throughout. For technical details of various aspects of the architecture, readers should refer to the FIDO Alliance specifications in the Bibliography.

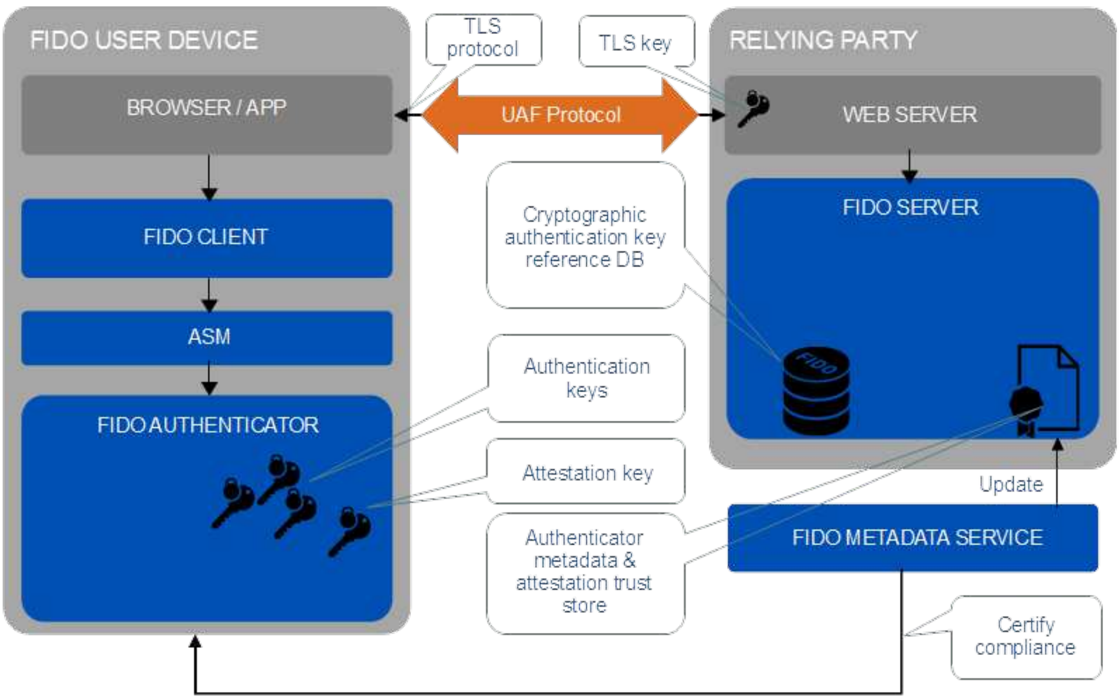

Fig. 1 FIDO Reference Architecture

Conceptually, FIDO involves a conversation between a computing environment controlled by a Relying Party and one controlled by the user to be authenticated. The Relying Party's environment consists conceptually of at least a web server and the server-side portions of a web application, plus a FIDO Server. The FIDO Server has a trust store, containing the (public) trust anchors for the attestation of FIDO Authenticators. The users' environment, referred to as the FIDO user device, consists of one or more FIDO Authenticators, a piece of software called the FIDO Client that is the endpoint for UAF and U2F conversations, and User Agent software. The User Agent software may be a browser hosting a web application delivered by the Relying Party, or it may be a standalone application delivered by the Relying Party. In either case, the FIDO Client, while a conceptually distinct entity, may actually be implemented in whole or part within the boundaries of the User Agent.

2.1 Intended Audience

This document assumes a technical audience that is proficient with security analysis of computing systems and network protocols as well as the specifics of the FIDO architecture and protocol families. It discusses the security goals, security measures, security assumptions and a series of threats to FIDO systems, including the users' computing environment, the Relying Party's computing environment, and the supply chain, including the vendors of FIDO components.

3. Attack Classification

The following attacks all result in user impersonation if successful. However, they have distinguishing characteristics which we use as the basis for attack classification:

1. Automated attacks not focused on the users systems, which affect the user.
2. Automated attacks which are focused on the users' device and which are performed once and lead to the ability to impersonate the user on an on-going basis without involving him or his device directly.
3. Automated attacks which involve the user or his device for each successful impersonation.
4. Automated attacks to sessions authenticated by the user.
5. Not automatable attacks to the user or his device which are performed once and lead to the ability to impersonate the user on an on-going basis without involving him or his device directly.
6. Not automatable attacks to the user or his device which involve the user or his device for each successful impersonation.

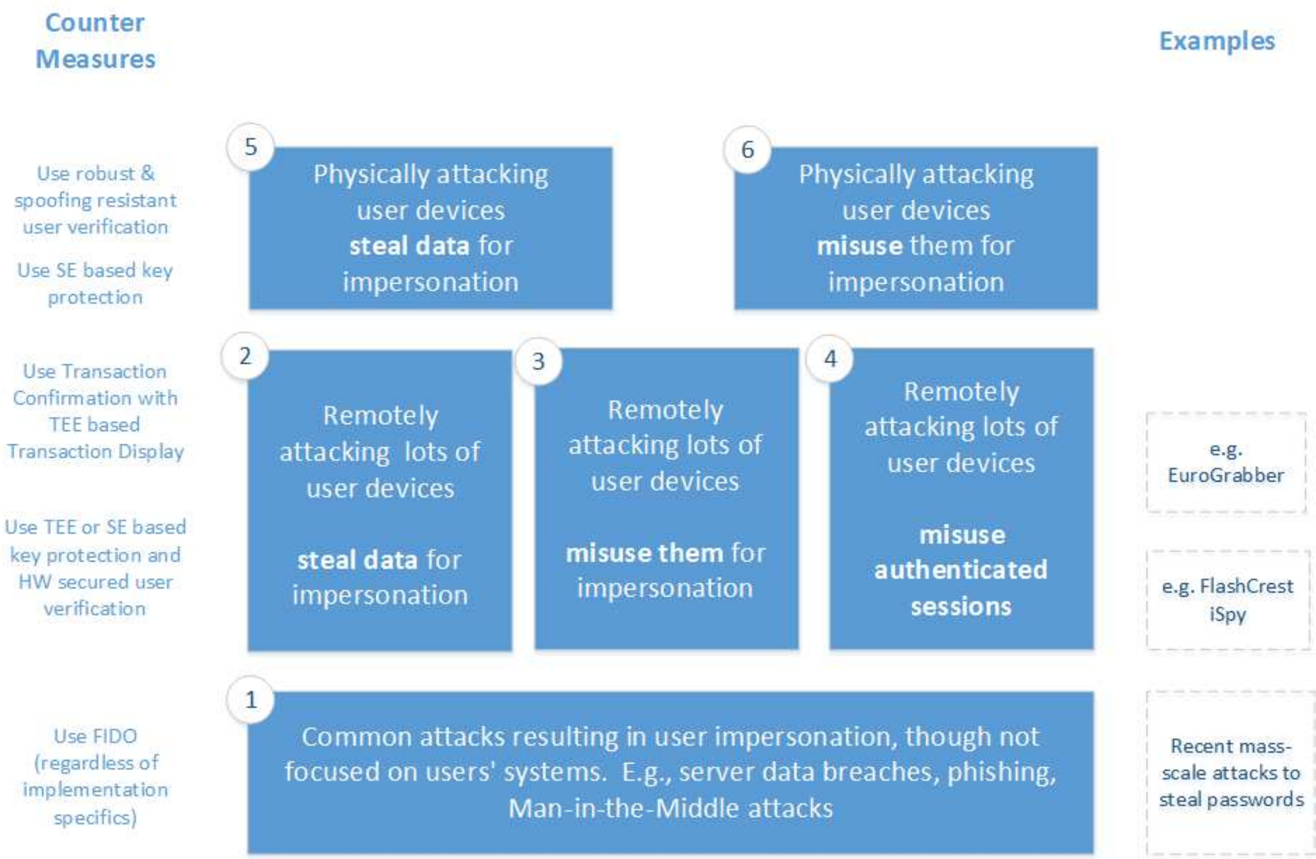

Fig. 2 Attack Classes

The first four attack classes are considered scalable as they are nominally automatable. The attack classes 5 and 6 are not automatable; they involve some kind of manual/physical interaction of the attacker with the user or his device. We will attribute the threats analyzed in this document with the related attack class (AC1 – AC6).

NOTE

1. FIDO uses asymmetric cryptography to protect against AC1. This gives control back to the user, i.e. when using good random numbers, the user's authenticator can make breaking the key as hard as the underlying factoring (in the case of RSA) or discrete logarithm (in the case of DSA or ECDSA) problem.
2. Once counter-measures for this kind of attack are commonly in place, attackers will likely focus on another attack class.
3. The numbers at the attack classes do not imply a feasibility ranking of the related attacks, e.g. it is not necessarily more difficult to perform (AC4) than it is to perform (AC3).
4. The user has almost no influence on the feasibility of attack class (AC1). This makes this attack class really bad.
5. The concept of physical security (i.e. "protect your Authenticator from being stolen"), related to attack classes (AC5) and (AC6) is much better internalized by users than the concept of logical security, related to attack classes (AC2), (AC3) and (AC4).
6. In order to protect against misuse of authenticated sessions (e.g. MITB attacks), the FIDO Authenticator must support the concept of transaction confirmation and the relying party must use it.
7. For an attacker to succeed in impersonating the user, any attack class is sufficient.

Attack Classes

We define the term scalable attack as any attack where the marginal cost of adding an additional target is near zero and which leads to violations of the FIDO security goals.

NOTE

The first four attack classes (AC1, AC2, AC3, and AC4) are considered scalable. The last two attack classes (AC5 and AC6) are not scalable and are performed as one-off user/Relying Party style compromises. We will attribute the threats analyzed in this document with the related attack class (AC1 – AC6).

AC1
: Attacks not focused on the users' devices and which lead to violations of FIDO security goals. (e.g., compromise of a Relying Party FIDO database and successful decryption of wrapped keys within the database, phishing, MITM attacks, etc.).

AC2
: Scalable attacks involving the Authenticator which, once performed, lead to the ability to violate FIDO security goals on an ongoing basis without later involving the users or their devices directly (e.g., a scalable attack on FIDO Authenticators that recovers the user private keys, allowing the attacker to impersonate the users on an ongoing basis).

AC3
: Scalable attacks which involve the user or his device for each instance where the FIDO security goals are violated (e.g., a scalable attack that requires the Authenticator for each successful impersonation).

AC4
: Scalable attacks on sessions authenticated by the user which violate FIDO security goals.

AC5
: Non-scalable attacks involving the Authenticator which, once performed, lead to the ability to violate FIDO security goals on an ongoing basis without later involving the users or their devices directly (e.g., a non-scalable attack on FIDO Authenticators that recovers the user private keys, allowing the attacker to impersonate the users on an ongoing basis).

AC6
: Non-scalable attacks which involve the user or his device for each instance where the FIDO security goals are violated (e.g., a non-scalable attack that requires the Authenticator for each successful impersonation).

NOTE

At this time we are not explicitly addressing classes of physical attacks on the authenticator that may lead to reduced security if the legitimate user uses the authenticator *after* the attacker having physical access to it.

4. FIDO Security Goals

In this section the specific security goals of FIDO are described. The FIDO UAF protocol [UAFProtocol] and U2F protocol [U2FOverview] support a variety of different FIDO Authenticators. Even though the security of those authenticators varies, the UAF protocol and the FIDO Server should provide a very high level of security - at least on a conceptual level. In reality it might require a FIDO Authenticator with a high security level in order to fully leverage the FIDO security strength.

NOTE

In certain environments the overall security of the explicit authentication (as provided by FIDO) is less important, as it might be supplemented with a high degree of implicit authentication or the application doesn't even require a high level of authentication strength.

The FIDO U2F protocol [U2FOverview] supports a more constrained set of Authenticator capabilities. It shares the same security goals as UAF, with the exception of [SG-14] Transaction Non- Repudiation.

The FIDO protocols have the following security goals:

[SG-1]
: Strong User Authentication: Authenticate (i.e. recognize) a user and/or a device to a relying party with high (cryptographic) strength.

[SG-2]
: Credential Guessing Resilience: Provide robust protection against eavesdroppers, e.g. *be resilient to physical observation, resilient to targeted impersonation, resilient to throttled and unthrottled guessing*.

[SG-3]
: Credential Disclosure Resilience: Be *resilient to phishing attacks* and real-time phishing attack, including resilience to online attacks by adversaries able to actively manipulate network traffic.

[SG-4]
: Unlinkablity: Protect the protocol conversation such that any two relying parties cannot link the conversation to one user (i.e. be *unlinkable*).

[SG-5]
: Verifier Leak Resilience: Be *resilient to leaks from other relying parties* I.e., nothing that a verifier could possibly leak can help an attacker impersonate the user to another relying party.

[SG-6]
: Authenticator Leak Resilience: Be resilient to leaks from other FIDO Authenticators. I.e., nothing that a particular FIDO Authenticator could possibly leak can help an attacker to impersonate any other user to any relying party.

[SG-7]
: User Consent: Notify the user before a relationship to a new relying party is being established (*requiring explicit consent*).

[SG-8]
: Limited PII: Limit the amount of personal identifiable information (PII) exposed to the relying party to the absolute minimum.

[SG-9]
: Attestable Properties: Relying Party must be able to verify FIDO Authenticator model/type (in order to calculate the associated risk).

[SG-10]
: DoS Resistance: Be resilient to *Denial of Service Attacks*. I.e. prevent attackers from inserting invalid registration information for a legitimate user for the next login phase. Afterward, the legitimate user will not be able to login successfully anymore.

[SG-11]
: Forgery Resistance: Be resilient to *Forgery Attacks (Impersonation Attacks)*. I.e. prevent attackers from attempting to modify intercepted communications in order to masquerade as the legitimate user and login to the system.

[SG-12]
: Parallel Session Resistance: Be resilient to *Parallel Session Attacks*. Without knowing a user's authentication credential, an attacker can masquerade as the legitimate user by creating a valid authentication message out of some eavesdropped communication between the user and the server.

[SG-13]
: Forwarding Resistance: Be resilient to *Forwarding and Replay Attacks*. Having intercepted previous communications, an attacker can impersonate the legal user to authenticate to the system. The attacker can replay or forward the intercepted messages.

[SG-14] (not covered by U2F)
: Transaction Non-Repudiation: Provide strong cryptographic non-repudiation for secure transactions.

[SG-15]
: Respect for Operating Environment Security Boundaries: Ensure that registrations and private key material as a shared system resource is appropriately protected according to the operating environment privilege boundaries in place on the FIDO user device.

[SG-16]
: Assessable Level of Security: Ensure that the design and implementation of the Authenticator allows for the testing laboratory / FIDO Alliance to assess the level of security provided by the Authenticator.

> NOTE
>
> For a definition of the phrases printed in *italics*, refer to [QuestToReplacePasswords] and to [PasswordAuthSchemesKeyIssues]

4.1 Assets to be Protected

Independent of any particular implementation, the FIDO protocols assume some assets to be present and to be protected.

1. Cryptographic Authentication Private Key. Typically, private keys in FIDO are unique for each tuple of (relying party, user account, authenticator).
2. Cryptographic Authentication Key Reference. This is the cryptographic material stored at the relying party and used to uniquely verify the Cryptographic Authentication Key, typically the public key corresponding to the authentication private key.
3. Authenticator Attestation Key (as stored in each authenticator). This should only be usable to attest a Cryptographic Authentication Key and the type/model and manufacturing batch of an Authenticator. Attestation keys are either ECDAA keys [FIDOEcdaaAlgorithm] or the attestation keys and certificates are shared by a large number of authenticators in a device class from a given vendor in order to prevent their becoming a linkable identifier across relying parties. Authenticator attestation certificates may be self-signed, or signed by an authority key controlled by the vendor.
4. Authenticator Attestation Authority Key. An authenticator vendor may elect to sign authenticator attestation certificates with a per-vendor certificate authority key.
5. Authenticator Attestation Authority Certificate. Contained in the initial/default trust store as part of the FIDO Server and contained in the active trust store maintained by each relying party.
6. Active Trust Store. Contains all trusted attestation root certificates for a given FIDO server.
7. All data items suitable for uniquely identifying the authenticator across relying parties. An attack on those would break the non-linkability security goal.
8. Private key of Relying Party TLS server certificate.
9. TLS root certificate trust store for the users' browser/app.

5. FIDO Security Measures

> NOTE
>
> Particular implementations of FIDO Clients, Authenticators, Servers and participating applications may not implement all of these security measures (e.g. Secure Display, [SM-10] Transaction Confirmation) and they also might (and should) implement add itional security measures.

> NOTE
>
> The U2F protocol lacks support for [SM-5] Secure Display, [SM-10] Transaction Confirmation, has only server-supplied [SM-8] Protocol Nonces, and [SM-3] Authenticator Class Attestation is implicit as there is only a single class of device.

[SM-1] (U2F + UAF)
: Key Protection: Authentication key is protected against misuse. Misuse means any use violating the FIDO specification or the details given in the Metadata Statement. Before a key can be used, it requires the User to unlock it using the user verification method specified in the Authenticator Metadata Statement (Silent Authenticators do not require any user verification method).

[SM-2] (U2F + UAF)
: Unique Authentication Keys: Cryptographic authentication key is specific and unique to the tuple of (FIDO Authenticator, User, Relying Party).

[SM-3] (U2F + UAF)
: Authenticator Class Attestation: Hardware-based FIDO Authenticators support authenticator attestation using an attestation key using one of the FIDO specified attestation types and algorithms. Each relying party receives regular updates of the trust store (through the FIDO Metadata service).

[SM-4] (UAF)
: Authenticator Status Checking: Relying Parties can download latest known status of authenticators included in the FIDO Metadata Service. The FIDO Server should take this information into account. Authenticator manufacturers should notify the FIDO Alliance about compromised authenticators. In the case of FIDO certified authenticators, such notification might even be mandatory.

[SM-5] (UAF)
: User Consent: FIDO Client implements a user interface for getting user's consent on any actions (except authentication with silent authenticator) and displaying RP name (derived from server URL).

[SM-6] (U2F + UAF)
: Cryptographically Secure Verifier Database: The relying party stores only the public portion of an asymmetric key pair, or an encrypted key handle, as a cryptographic authentication key reference.

[SM-7] (U2F + UAF)
: Secure Channel with Server Authentication: The TLS protocol with server authentication or a transport with equivalent properties is used as transport protocol for UAF. The use of https is enforced by a browser or Relying Party application.

[SM-8] (UAF)
: Protocol Nonces: Both server and client supplied nonces are used for UAF registration and authentication. U2F requires server supplied nonces.

[SM-9] (U2F + UAF)
: Authenticator Certification: The FIDO Metadata Service includes the Authenticator certification status.

[SM-10] (UAF)
: Transaction Confirmation (WYSIWYS): Secure Display (WYSIWYS) (optionally) implemented by the FIDO Authenticators is used by FIDO Client for displaying relying party name and transaction data to be confirmed by the user.

[SM-11] (U2F + UAF)
Round Trip Integrity: FIDO server verifies that the transaction data related to the server challenge received in the UAF message from the FIDO client is identical to the transaction data and server challenge delivered as part of the UAF request message.

[SM-12] (U2F + UAF)
Channel Binding: Relying Party servers may verify the continuity of a secure channel with a client application.

[SM-13] (UAF)
Key Handle Access Token: Authenticators not intended to roam between untrusted systems are able to constrain the use of registration keys within the privilege boundaries defined by the operating environment of the user device (per-user, or per application, or per-user + per-application as appropriate).

[SM-14] (U2F + UAF)
AppID Separation: A Relying Party can declare the application identities allowed to access its registered keys, for operating environments on user devices that support this concept.

[SM-15] (U2F + UAF)
Signature Counter: Authenticators send a monotonically increasing signature counter that a Relying Party can check to possibly detect cloned authenticators.

[SM-16] (U2F + UAF)
Use of strong, modern Cryptographic Primitives: The FIDO specifications stipulate the use of strong, modern cryptographic primitives helping to ensure the overall security of conformant FIDO implementations. The FIDO Authenticator certification program defines the "Allowed Cryptography List" for allowed cryptographic primitives to be used in FIDO certified authenticators.

[SM-17] (U2F + UAF)
Resistance to Side Channel Attacks.

[SM-18] (U2F + UAF)
Resistance to Injected Faults in Cryptographic Functions. This security measure purely deals with the cryptographic functions, as compared to the much more general [SM-28].

[SM-19] (UAF)
Bounded Probability of a Birthday Collision. For randomly generated nonces, the total number of nonces that can be generated is limited to bound the probability of a birthday collision of generated values.

[SM-20] (U2F + UAF)
Individual authenticators are indistinguishable provided authenticators sharing attestation keys are manufactured in sufficiently large (e.g. > 100000) per-model batches.

[SM-21] (U2F + UAF)
Authentication and replay-resistance (freshness assurance) of externally-stored protected information.

[SM-22] (U2F + UAF)
Certified FIDO Authenticators fully described by the vendor, and tested to verify that it functions as specified.

[SM-23] (U2F + UAF)
Key Handles containing a key are cryptographically linked with the Authenticator that produced the Key Handle and with the Relying Party associated with the Key Handle.

[SM-24] (U2F + UAF)
Design, implementation and manufacture of certified FIDO Authenticators supports Authenticator security.

[SM-25] (U2F + UAF)
Depending on the certification level, certified authenticators are required to implement a Trusted Path for all user / Authenticator direct interactions.

[SM-26] (U2F + UAF)
Input Data Validation: Malformed or maliciously crafted input data does not result in unexpected Authenticator behavior.

[SM-27] (U2F + UAF)
Protection of user verification reference data and biometric data.

[SM-28] (U2F + UAF)
Resistance to Fault Injection Attacks.

[SM-29] (U2F + UAF)
Resistance to Remote Timing Attacks: No leakage of secret information to remote entities via variation of operation execution time.

5.1 Relation between Measures and Goals

Security Goal	Supporting Security Measures
[SG-1] Strong User Authentication	[SM-1] Key Protection [SM-12] Channel Binding [SM-14] AppID Separation [SM-15] Signature Counter [SM-16] Allowed Crypto Primitives [SM-17] Resistance to Side Channel Attacks [SM-21] Authentication and replay-resistance [SM-23] Key Handles cryptographically linked with the Authenticator [SM-25] Trusted path for all user interactions [SM-29] Resistance to Remote Timing Attacks
[SG-2] Credential Guessing Resilience	[SM-1] Key Protection [SM-6] Cryptographically Secure Verifier Database [SM-16] Allowed Crypto Primitives
[SG-3] Credential Disclosure Resilience	[SM-1] Key Protection [SM-9] Authenticator Certification [SM-15] Signature Counter

Security Goal	Supporting Security Measures
	[SM-17] Resistance to Side Channel Attacks [SM-29] Resistance to Remote Timing Attacks
[SG-4] Unlinkability	[SM-2] Unique Authentication Keys [SM-3] Authenticator Class Attestation [SM-20] No Identifying Information
[SG-5] Verifier Leak Resilience	[SM-2] Unique Authentication Keys [SM-6] Cryptographically Secure Verifier Database [SM-16] Allowed Crypto Primitives
[SG-6] Authenticator Leak Resilience	[SM-9] Authenticator Certification [SM-15] Signature Counter [SM-16] Allowed Crypto Primitives
[SG-7] User Consent	[SM-1] Key Protection [SM-5] User Consent [SM-7] Secure Channel with Server Authentication [SM-10] Transaction Confirmation (WYSIWYS) [SM-25] Trusted path for all user interactions
[SG-8] Limited PII	[SM-2] Unique Authentication Keys [SM-20] No Identifying Information
[SG-9] Attestable Properties	[SM-3] Authenticator Class Attestation [SM-4] Authenticator Status Checking [SM-9] Authenticator Certification
[SG-10] DoS Resistance	[SM-8] Protocol Nonces
[SG-11] Forgery Resistance	[SM-7] Secure Channel with Server Authentication [SM-8] Protocol Nonces [SM-11] Round Trip Integrity [SM-12] Channel Binding [SM-17] Resistance to Side Channel Attacks [SM-23] Key Handles cryptographically linked with the Authenticator [SM-29] Resistance to Remote Timing Attacks
[SG-12] Parallel Session Resistance	[SM-7] Secure Channel with Server Authentication [SM-8] Protocol Nonces [SM-11] Round Trip Integrity [SM-12] Channel Binding
[SG-13] Forwarding Resistance	[SM-7] Secure Channel with Server Authentication [SM-8] Protocol Nonces [SM-11] Round Trip Integrity

Security Goal	[SM-12] Channel Binding Supporting Security Measures
[SG-14] Transaction Non-Repudiation	[SM-1] Key Protection [SM-2] Unique Authentication Keys [SM-8] Protocol Nonces [SM-9] Authenticator Certification [SM-10] Transaction Confirmation (WYSIWYS) [SM-11] Round Trip Integrity [SM-12] Channel Binding [SM-25] Trusted path for all user interactions
[SG-15] Respect for Operating Environment Security Boundaries	[SM-13] Key Handle Access Token [SM-14] AppID Separation

6. FIDO Security Assumptions

In this section, we enumerate the assumptions we are making regarding the security characteristics of the operating environment components on which a FIDO implementation depends.

[SA-1]
: The Authenticator and its cryptographic algorithms and parameters (key size, mode, output length, etc.) in use are not subject to unknown weaknesses that make them unfit for their purpose in encrypting, digitally signing, and authenticating messages.

[SA-2]
: Operating system privilege separation mechanisms relied up on by the software modules involved in a FIDO operation on the user device perform as advertised. E.g. boundaries between user and kernel mode, between user accounts, and between applications (where applicable) are securely enforced and security principals can be mutually, securely identifiable.

[SA-3]
: Applications on the user device are able to establish secure channels that provide trustworthy server authentication, and confidentiality and integrity for messages (e.g., through TLS).

[SA-4]
: The computing environment on the FIDO user device and the and applications involved in a FIDO operation act as trustworthy agents of the user.

[SA-5]
: The inherent value of a cryptographic key resides in the confidence it imparts, and this commodity decays with the passage of time, irrespective of any compromise event. As a result the effective assurance level of authenticators will be reduced over time.

[SA-6]
: The computing resources at the Relying Party involved in processing a FIDO operation act as trustworthy agents of the Relying Party.

6.1 Discussion

With regard to [SA-4] and malicious computation on the FIDO user device, only very limited guarantees can be made within the scope of these assumptions. Malicious code privileged at the level of the trusted computing base can always violate [SA-2] and [SA- 3]. Malicious code privileged at the level of the users' account in traditional multi-user environments will also likely be able to violate [SA-3].

FIDO can also provide only limited protections when a user chooses to deliberately violate [SA-4], e.g. by roaming a USB authenticator to an untrusted system like a kiosk, or by granting permissions to access all authentication keys to a malicious app in a mobile environment. Transaction Confirmation can be used as a method to protect against compromised FIDO user devices.

In to components such as the FIDO Client, Server, Authenticators and the mix of software and hardware modules they are comprised of, the end-to-end security goals also depend on correct implementation and adherence to FIDO security guidance by other participating components, including web browsers and relying party applications. Some configurations and uses may not be able to meet all security goals. For example, authenticators may lack a secure display, they may be composed only of unattestable software components, they may be deliberately designed to roam between untrusted operating environments, and some operating environments may not provide all necessary security primitives (e.g., secure IPC, application isolation, modern TLS implementations, etc.)

7. Threat Analysis

In the following tables describing threats, we mention the relevant attack class(es) in the left column if the threat might lead to user impersonation.

7.1 Threats to Client Side

7.1.1 Exploiting User’s pattern matching weaknesses

T-1.1.1	Homograph Mis-Registration	Violates
AC3	The user registers a FIDO authentication key with a fraudulent web site instead of the genuine Relying Party. **Consequences:** The fraudulent site may convince the user to disclose a set of non-FIDO credentials sufficient to allow the attacker to register a FIDO Authenticator under its own control, at the genuine Relying Party, on the users' behalf, violating [SG-1] Strong User Authentication. **Mitigations:** Disclosure of non-FIDO credentials is outside of the scope of the FIDO security measures, but Relying Parties should be aware that the initial strength of an authentication key is no better than the identity-proofing applied as part of the registration process.	SG-1

T-1.1.1	Homograph Mis-Registration	Violates

T-1.1.2	Homograph Mis-Authentication	Violates
AC3	The user accidentally browses to a fraudulent web site. The attacker tries to act as man-in-the-middle (MITM) and requests the user to authenticate. In the case of username/password based authentication this is a typical phishing attack. **Consequences:** The FIDO subsystem will determine that either (a) no FIDO authenticator has been registered with the fraudulent site or (b) it will use the FIDO Uauth key registered to the fraudulent site - which is different from the Uauth key for the relying party's site. **Mitigations:** FIDO inherently ties keys to the relying party (formally identified by the AppID, and authenticated by TLS and the CA infrastructure).	SG-1, SG-4

7.1.2 Threats to the User Device, FIDO Client and Relying Party Client Applications

T-1.2.1	FIDO Client Corrpution	Violates
AC3	Attacker gains ability to execute code in the security context of the FIDO Client. **Consequences:** Violation of [SA-4]. **Mitigations:** When the operating environment on the FIDO user device allows, the FIDO Client should operate in a privileged and isolated context under [SA-2] to protect itself from malicious modification by anything outside of the Trusted Computing Base.	SA-4

T-1.2.2	Logical/Physical User Device Attack	Violates
AC3 / AC5	Attacker gains physical access to the FIDO user device but not the FIDO Authenticator. **Consequences:** Possible violation of [SA-4] by installing malicious software or otherwise tampering with the FIDO user device. **Mitigations:** [SM-1] Key Protection prevents the disclosure of authentication keys or other assets during a transient compromise of the FIDO user device. A persistent compromise of the FIDO user device can lead to a violation of [SA-4] unless additional protection measures outside the scope of FIDO are applied to the FIDO user device. (e.g. whole disk encryption and boot-chain integrity).	SA-4

T-1.2.3	User Device Account Access	Violates
AC3 / AC4	Attacker gains access to a user's login credentials on the FIDO user device. **Consequences:** Authenticators might be remotely abused, or weakly-verifying authenticators might be locally abused, violating [SG-1] Strong User Authentication and [SG-13] Transaction Non-Repudiation. Possible violation of [SA-4] by the installation of malicious software. **Mitigations:** Relying Parties can use [SM-9] Authenticator Certification and [SM-3] Authenticator Class Attestation to determine the nature of authenticators and not rely on weak, or weakly-verifying authenticators for high value operations.	SG-1, SG-13; SA-4

T-1.2.4	App Server Verification Error	Violates
AC3	A client application fails to properly validate the remote sever identity, accepts forged or stolen credentials for a remote server, or allows weak or missing cryptographic protections for the secure channel. **Consequences:** An active network adversary can modify the Relying Party's authenticator policy and downgrade the client's choice of authenticator to make it easier to attack. An active network adversary can intercept or view FIDO messages intended for the Relying Party. It may be able to use this ability to violate [SG-12] Parallel Session Resistance, [SG-11] Forgery Resistance or [SG-13] Forwarding Resistance. **Mitigations:** The server can verify [SM-8] Protocol Nonces to detect replayed messages and protect from an adversary that can read but not modify traffic in a secure channel. The server can mandate a channel with strong cryptographic protections to prevent message forgery and can verify a [SM-12] Channel Binding to detect forwarded messages.	SG-11, SG-12, SG-13

T-1.2.5	RP App Corruption	Violates
	An attacker is able to obtain malicious execution in the security context of the Relying Party client application (e.g. via Cross-Site Scripting (XSS)) or abuse the secure channel or session identifier after the user has successfully authenticated. This is a client side attack. **Consequences:** The attacker is able to control the users' session, violating [SG-14] Transaction Non-Repudiation.	SG-14

Mitigations: The server can employ [SM-10] Transaction Confirmation to gain additional assurance for high value operations.

T-1.2.5	RP App Corruption	Violates

T-1.2.6	Fingerprinting Authenticators	Violates
	A remote adversary is able to uniquely identify a FIDO user device using the fingerprint of discoverable configuration of its FIDO Authenticators. **Consequences:** The exposed information violates [SG-8] Limited PII, allowing an adversary to violate [SG-7] User Consent by strongly identfying the user without their knowledge and [SG-4] Unlinkablity by sharing that fingerprint. **Mitigations:** [SM-3] Authenticator Class Attestation ensures that the fingerprint of an Authenticator will not be unique. For web browsing situations where this threat is most prominent, user agents may provide additional user controls around the discoverability of FIDO Authenticators.	SG-4, SG7, SG-8

T-1.2.7	App to FIDO Client full MITM attack	Violates
AC3	Malicious software on the FIDO user device is able to read, tamper with, or spoof the endpoint of inter-process communication channels between the FIDO Client and browser or Relying Party application. **Consequences:** Adversary is able to subvert [SA-2]. **Mitigations:** On platforms where [SA-2] is not strong the security of the system may depend on preventing malicious applications from being loaded onto the FIDO user device. Such protections, e.g. app store policing, are outside the scope of FIDO. When using [SM-10] Transaction Confirmation, the user will be presented with the relevant AppID and transaction text and will be able to evaluate whether or not to consent to the transaction.	SA-2

T-1.2.8	Authenticator to App Read-Only MITM attack	Violates
AC3	An adversary is able to obtain an authenticator's signed protocol response message. **Consequences:** The attacker attempts to replay the message to authenticate as the user, violating [SG-1] Strong User Authentication, [SG-13] Forwarding Resistance and [SG-12] Parallel Session Resistance. **Mitigations:** The server can use [SM-8] Protocol Nonces to detect replay of messages and verify [SM-11] Round Trip Integrity to detect modified messages.	SG-1, SG-12, SG-13

T-1.2.9	Malicious App	Violates
AC3	A user installs an application that represents itself as being associated with to one Relying Party application but actually initiates a protocol conversation with a different Relying Party and attempts to abuse previously registered authentication keys at that Relying Party. **Consequences:** Adversary is able to violate [SG-7] User Consent by misrepresenting the target of authentication. Other consequences equivalent to [T-1.2.5] **Mitigations:** If a [SM-5] Transaction Confirmation Display is present, the user may be able to verify the true target of an operation. If the malicious application attempts to communicate directly with an Authenticator that uses [SM-13] KeyHandleAccessToken, it should not be able to access keys registered by other FIDO Clients. If the operating environment on the FIDO user device supports it, the FIDO client may be able to determine the application's identity and verify if it is authorized to target that Relying Party using a [SM-14] AppID Separation.	SG-7

T-1.2.10	Phishing Attack	Violates
AC2	A Phisher convinces the user to enter his PIN used for user verification into an application / web site disclosing the PIN to the Phisher. In the traditional username/password world this enables the attacker to successfully impersonate the user (to the relying party). **Consequences:** None as the phisher additionally would need access to the Authenticator in order to pass user verification [SM-1]. In FIDO, the user verification PIN (if user verification is done via PIN) is not known to the relying party and hence isn't sufficient for user impersonation. If user verification is done using an alternative user verification method, this applies accordingly. **Mitigations:** In FIDO, the Uauth.priv key is used to sign a relying party supplied challenge. without (use) access to that key, no impersonation is possible.	SG-1

7.1.3 Creating a Fake Client

T-1.3.1	Malicious FIDO Client	Violates

T-1.3.1	Malicious FIDO Client	Violates
AC3	Attacker convinces users to install and use a malicious FIDO Client. **Consequences:** Violation of [SA-4] **Mitigations:** Mitigating malicious software installation is outside the scope of FIDO. If an authenticator implements [SM-1] Key Protection, the user may be able to recover full control of their registered authentication keys by removing the malicious software from their user device. When using [SM-10] Transaction Confirmation, the user sees the real AppIDs and transaction text and can decide to accept or reject the action.	SA-4

7.1.4 Threats to FIDO Authenticator

T-1.4.1	Malicious Authenticator	Violates
AC2, AC3	Attacker convinces users to use a maliciously implemented authenticator. **Consequences:** The fake authenticator does not implement any appropriate security measures and is able to violate all security goals of FIDO. **Mitigations:** A user may be unable to distinguish a malicious authenticator, but a Relying Party can use [SM-3] Authenticator Class Attestation to identify and only allow registration of reliable authenticators that have passed [SM-9] Authenticator Certification. A Relying Party can additionally rely on [SM-4] Authenticator Status Checking to check if an attestation presented by a malicious authenticator has been marked as compromised.	SG-1

T-1.4.2	Uauth.priv Key Compromise	Violates
AC2	Attacker succeeds in extracting a user's cryptographic authentication private key for use in a different context. **Consequences:** The attacker could impersonate the user with a cloned authenticator that does not do trustworthy user verification, violating [SG-1]. **Mitigations:** [SM-1] Key Protection measures are intended to prevent this. Each authentication private key is only used for one relying party. Relying Parties can check [SM-9] Authenticator Certification attributes to determine the type of key protection in use by a given authenticator class. Relying Parties can additionally verify the [SM-15] Signature Counter and detect that an authenticator has been cloned if it ever fails to advance relative to the prior operation.	SG-1

T-1.4.3	User Verification By-Pass	Violates
AC3, AC5	Attacker could use the cryptographic authentication key (inside the authenticator) either with or without being noticed by the legitimate user. **Consequences:** Attacker could impersonate user, violating [SG-1]. **Mitigations:** A user can only register and a Relying Party only allow authenticators that perform [SM-1] Key Protection with an appropriately secure user verification process. Does not apply to Silent Authenticators (see [FIDOGlossary]).	SG-1

T-1.4.4	Physical Authenticator Attack	Violates
AC2, AC5, AC6	Attacker could get physical access to FIDO Authenticator (e.g. by stealing it). **Consequences:** Attacker could bring the authenticator in a lab in order to use the authentication key (e.g. by-passing user verification and knowing the RP related to this key). If this physical attack succeeds, the attacker could successfully impersonate the user, violating [SG-1] Strong User Authentication. Attacker can introduce a low entropy situation to recover an ECDSA signature key (or optherwise extract the Uauth.priv key), violating [SG-9] Attestable Properties if the attestation key is targeted or [SG-1] Strong User Authentication if a user key is targeted. **Mitigations:** [SM-1] Key Protection includes requirements to implement strong protections for key material, including resiliance to offline attacks and low entropy situations. Relying Parties should use [SM-3] Authenticator Class Attestation to only accept Authenticators implementing a sufficiently strong user verification method.	SG-1

T-1.4.6	Fake Authenticator	Violates
	Attacker is able to extract the authenticator attestation key from an authenticator, e.g. by neutralizing physical countermeasures in a laboratory setting. **Consequences:** Attacker can violate [SG-9] Attestable Properties by creating a malicious hardware or software	

T-1.4.6	Fake Authenticator	Violates
AC2	device that represents itself as a legitimate one. **Mitigations:** Relying Parties can use [SM-4] Authenticator Status Checking to identify known-compromised keys. Identification of such compromise is outside the strict scope of the FIDO protocols.	SG-9

T-1.4.7	Transaction Confirmation Display Overlay Attack	Violates
AC6	Attacker is able to subvert [SM-5] Secure Display functionality (WYSIWYS), perhaps by overlaying the display with false information. **Consequences:** Violation of [SG-14] Transaction Non-Repudiation. **Mitigations:** Authenticator implementations must take care to protect in their implementation of a secure display, e.g. by implementing a distinct hardware display or employing appropriate privileges in the operating environment of the user device to protect against spoofing and tampering. [SM-9] Authenticator Certification will provide Relying Parties with metadata about the nature of a transaction confirmation display information that can be used to assess whether it matches the assurance level and risk tolerance of the Relying Party for that particular transaction.	SG-14

T-1.4.8	Signature Algorithm Attack	Violates
AC1, AC2, AC3, AC5	A cryptographic attack is discovered against the public key cryptography system used to sign data by the FIDO authenticator. See also T-1.4.10. **Consequences:** Attacker is able to use messages generated by the client to violate [SG-2] Credential Guessing Resistance. **Mitigations:** [SM-8] Protocol Nonces, including client-generated entropy, limit the amount of control any adversary has over the internal structure of an authenticator. [SM-1] Key Protection for non-silent authenticators requires user interaction to authorize any operation performed with the authentication key, severely limiting the rate at which an adversary can perform adaptive cryptographic attacks.	SG-2

T-1.4.9	Abuse Functionality	Violates
AC2, AC3, AC5, AC6	It might be possible for an attacker to abuse the Authenticator functionality by sending commands with invalid parameters or invalid commands to the Authenticator. **Consequences:** This might lead to e.g., user verification by-pass or potential key extraction. **Mitigations:** Proper robustness (e.g. due to testing) of the Authenticator firmware.	SG-1

T-1.4.10	Random Number prediction	Violates
AC2, AC3, AC5, AC6	It might be possible for an attacker to get access to information allowing the prediction of RNG data. **Consequences:** This might lead to key compromise situation [T-1.4.2] when using ECDSA (if the k value is used multiple times or if it is predictable). **Mitigations:** Proper robustness of the Authenticator's RNG and verification of the relevant operating environment parameters (e.g. temperature, ...).	SG-1

T-1.4.11	Firmware Rollback	Violates
	Attacker might be able to install a previous and potentially buggy version of the firmware. **Consequences:** This might lead to successful attacks, e.g. T-1.4.9. **Mitigations:** Proper robustness firmware update and verification method.	SG-1

T-1.4.12	User Verification Data Injection	Violates
AC3, AC6	Attacker might be able to inject pre-captured user verification data into the Authenticator. For example, if a password is used as user verification method, the attacker could capture the password entered by the user and then send the correct password to the Authenticator (by-passing the expected keyboard/PIN pad). Passwords could be captured ahead of the attack e.g. by convincing the user to enter the password into a malicious app (“phishing”) or by spying directly or indirectly the password data. In another example, some malware could play an audio stream which would be recorded by the microphone and used by a Speaker-Recognition based Authenticator. **Consequences:** This might lead to successful user impersonation (if the attacker has access to valid user verification data). **Mitigations:** Use a physically secured user verification input method, e.g. Fingerprint Sensor or Trusted-User-Interface for PIN entry which cannot be by-passed by malware.	SG-1

T-1.4.12	User Verification Data Injection	Violates

T-1.4.13	Verification Reference Data Modification	Violates
AC3, AC6	An attacker gains logical or physical access to the Authenticator and modifies Verification Reference Data (e.g. hashed PIN value, fingerprint templates) stored in the Authenticator and adds reference data known to or reproducible by the attacker. **Consequences:** The attacker would be recognized as the legitimate User and could impersonate the user. **Mitigations:** [SM-27] Proper protection of the the verification reference data and biometric data in the Authenticator.	SG-1

T-1.4.14	Read access to captured user verification data	Violates
AC3, AC6	The Attacker gained read access to the captured user verification data (e.g. PIN, fingerprint image, ...). **Consequences:** The attacker gets access to PII and could disclose it violating [SG-8]. **Mitigations:** Limiting access to the user verification data to the Authenticator exclusively.	SG-8

T-1.4.15	Compromised the internal PRNG state and the entropy source	Violates
AC1, AC2, AC5	In this threat, an attacker compromises the entropy source prior to the Authenticator initially seeding the PRNG during initialization or otherwise compromises the internal PRNG state, and the attacker is able to know or specify all future entropy inputs to the PRNG. No PRNG is able to recover to a secure status under this threat, but it serves as a useful point for comparison. **Consequences:** May undermine [SG-1], [SG-2], [SG-3], [SG-4], [SG-11], [SG-14]. **Mitigations:** This constitutes a complete compromise of the RNG, with no ability to recover, so mitigation for this threat involves reducing the impact of a compromised RNG. This is partially mitigated by using an allowed random number generator that allows secure integration of additional input [SM-16] and introduction of data derived from the RP challenge additional input to the PRNG, which can help so long as the attacker has not additionally compromised the TLS session or the ASM / Authenticator link. Using the deterministic signature generation methods (e.g., RFC 6979) can reduce the risk of compromise of existing keys during the signature process, as can using the private key and hash of the signed message as additional input to the PRNG during signature generation. Prevention of non-scalable versions of this style of attack is at least partially addressed by [SM-17] and [SM-18].	SG-1, SG-2, SG-3, SG-4, SG-11, SG-14

T-1.4.16	Compromised entropy source after successful seeding during initialization	Violates
AC1, AC2, AC5	In this threat, an attacker gains the ability to influence the Authenticator's entropy source, but only after the initial seeding has been conducted (e.g., if initial seeding occurred prior to the attack and / or as per-Authenticator factory injection of entropy). **Consequences:** May undermine [SG-1], [SG-2], [SG-3], [SG-4], [SG-11], [SG-14]. **Mitigations:** This is mitigated by using an allowed PRNG which retains PRNG state between power cycles; i.e., which conserves PRNG state even when being reseeded [SM-16]. Prevention of non-scalable versions of this style of attack is at least partially addressed by [SM-17] and [SM-18].	SG-1, SG-2, SG-3, SG-4, SG-11, SG-14

T-1.4.17	Compromised the internal PRNG state, but not the entropy source	Violates
AC1, AC2, AC5	In this threat, an attacker compromises the entropy source prior to seeding the PRNG or otherwise compromises the internal PRNG state, but then at some point, the attacker no longer can access / control the entropy source. **Consequences:** May undermine [SG-1], [SG-2], [SG-3], [SG-4], [SG-11], [SG-14] **Mitigations:** This can be mitigated by Authenticators reseeding periodically from an internal entropy source [SM-16]. As a note, this imposes a total number of random number generator requests prior to a required reseed event; in the event that the Authenticator does not have an entropy source internally, this may act as a hard limit on the number of registrations / authentications that such an Authenticator can perform. Prevention of non-scalable versions of this style of attack is at least partially addressed by [SM-17] and [SM-18].	SG-1, SG-2, SG-3, SG-4, SG-11, SG-14

T-1.4.18	Bad Key Generation	Violates
AC1, AC2, AC5	In this threat, random chance or active attack causes the key generated to be cryptographically flawed; e.g., an RSA key that can be factored using the Pollard p-1 algorithm more quickly than with the General Number Field Sieve. See also T-1.4.21. **Consequences:** May undermine [SG-1], [SG-2], [SG-4], [SG-11], [SG-14] **Mitigations:** This is mitigated by requiring use of an allowed random number generator (in the case of certified authenticators), requiring that keys be generated in the way required in the relevant standard specified in the Allowed Cryptography List [SM-16], and making the key generation process resistant to tampering by the attacker [SM-18].	SG-1, SG-2, SG-4, SG-11, SG-14

T-1.4.19	Local external side channel attacks	Violates
	In this threat, an attacker with possession of the Authenticator may be able to extract keys using timing,	

T-1.4.19	Local external side channel attacks	Violates
AC2 (associated with shared keys), AC5	power, RF, or near-field analysis. The impact depends on the key or secret recovered. **Consequences:** May undermine [SG-1], [SG-2], [SG-4], [SG-11], [SG-14]. **Mitigations:** This is mitigated by the side channel resistance security measure [SM-17].	SG-1, SG-2, SG-4, SG-11, SG-14

T-1.4.20	Internal side channel attacks	Violates
AC2 (associated with shared keys), AC5	In this threat, an attacker controlling a process running on the same hardware environment as the Authenticator may be able to recover keys by using information leaked by hardware or operating system characteristics (e.g., how often the attacker's process is scheduled, the state of the L1, L2 caches, etc.). **Consequences:** May undermine [SG-1], [SG-4], [SG-11], [SG-14]. **Mitigations:** This is mitigated by the side channel resistance security measure [SM-17].	SG-1, SG-4, SG-11, SG-14

T-1.4.21	Error injection during key or signature generation	Violates
AC2 (associated with shared keys), AC5	In this threat, an attacker is able to inject an error in the key or signature generation process that leaks part or all of the private key. **Consequences:** May undermine [SG-1], [SG-4], [SG-11], [SG-14]. **Mitigations:** This is mitigated by [SM-18] and [SM-28].	SG-1, SG-4, SG-11, SG-14

T-1.4.22	Birthday Paradox Collision	Violates
AC3, AC6	In this threat, a set of randomly generated parameters collide. The probability of this occurrence can be bounded using analysis similar to that associated with the classical Birthday Paradox. **Consequences:** May undermine [SG-1], [SG-11], [SG-14]. **Mitigations:** Establishing a bounded number of allowable outputs based on the size of the randomly generated value [SM-19].	SG-1, SG-11, SG-14

T-1.4.23	Privacy Reduction	Violates
AC1	In this threat, a small number of Authenticators share an attestation key which leaks information about the user across Relying Parties. **Consequences:** May undermine [SG-4]. **Mitigations:** This is mitigated by [SM-20].	SG-4

T-1.4.24	Covert Channel	Violates
AC1	In this threat, an Authenticator is malicious (either by design, or after having been independently compromised) and it is configured to leak secret or identifying data within apparently normal exchanges, or to other processes on the same hardware platform as the Authenticator. **Consequences:** May undermine [SG-1], [SG-4], [SG-5], [SG-6], [SG-8], [SG-11], [SG-14]. **Mitigations:** Note: This is an interesting thought experiment; use of random nonces and other non-deterministic elements make protection against this threat problematic.	SG-1, SG-4, SG-5, SG-6, SG-8, SG-11, SG-14

T-1.4.25	Subsitution of Protected Information	Violates
AC1, AC3, AC5, AC6	In this threat, an attacker substitutes protected information, either by modifying it piecemeal, or by completely substituting it with another value. (Some encryption modes allow an attacker to target bit-level changes to the plaintext. Authenticated data may also have been replaced with data that had previously been authenticated in the same way.) **Consequences:** May undermine [SG-1], [SG-4], [SG-11], [SG-14]. **Mitigations:** This threat is mitigated by [SM-1], [SM-16], [SM-21].	SG-1, SG-4, SG-11, SG-14

T-1.4.26	Compromise of Protected Information	Violates
AC1, AC2, AC5, AC6	In this threat, an attacker recovers data that should be protected by the Authenticator. **Consequences:** May undermine [SG-1], [SG-2], [SG-4], [SG-5], [SG-7], [SG-8], [SG-11], [SG-14]. **Mitigations:** This threat is mitigated by using allowed cryptographic primitives [SM-1], [SM-16].	SG-1, SG-2, SG-4, SG-5, SG-7, SG-8, SG-11, SG-14

T-1.4.27	Signature or registration counter non-monotonicity	Violates
AC1	In this threat, an attacker may be able to cause these counters to be reset, to roll over, or otherwise to decrease in value. **Consequences:** May undermine [SG-1], [SG-12], [SG-14]. **Mitigations:** This threat is mitigated by [SM-15].	SG-1, SG-12, SG-14

T-1.4.28	Hostile ASM / Client	Violates
AC3, AC5, AC6	In this threat, the Authenticator support infrastructure is hostile, and can feed arbitrary data to the Authenticator. **Consequences:** May undermine [SG-4], [SG-5], [SG-7], [SG-8]. **Mitigations:** This threat is mitigated by [SM-10], [SM-13].	SG-4, SG-5, SG-7, SG-8

T-1.4.29	Debug Interface	Violates
AC2 (associated with shared keys), AC3 (associated with shared keys), AC5, AC6	In this threat, the Authenticator has a hardware or software debugging interface that is not completely disabled prior to distribution of the Authenticator (e.g., pads for a JTAG port). **Consequences:** May undermine [SG-1], [SG-4], [SG-5], [SG-6], [SG-8], [SG-11], [SG-14]. **Mitigations:** This threat is mitigated by [SM-18], [SM-22], and [SM-28].	SG-1, SG-4, SG-5, SG-6, SG-8, SG-11, SG-14

T-1.4.30	Fault induced by malformed input	Violates
AC2, AC3, AC5, AC6	In this threat, the Authenticator behaves in an unexpected fashion due to an error in processing malformed input. The result of this style of attack is poorly controllable, absent strong internal segmentation of the Authenticator. **Consequences:** May undermine [SG-1], [SG-2], [SG-3], [SG-4], [SG-6], [SG-7], [SG-8], [SG-11], [SG-14], [SG-16]. **Mitigations:** This threat is mitigated by [SM-1], [SM-2], [SM-4], [SM-5], [SM-10], [SM-5], [SM-23], [SM-13], [SM-26].	SG-1, SG-2, SG-3, SG-4, SG-6, SG-7, SG-8, SG-11, SG-14, SG-16

T-1.4.31	Fault Injection Attack	Violates
AC2 (associated with shared keys), AC5, AC6	In this threat, an attacker subjects the Authenticator to conditions that induce hardware faults (e.g., exposure to photons or charged particles, inducing variations in supply voltage or external clock, altering the temperature, etc.) in an attempt to subvert some logical or physical protection. The result of this style of attack is poorly controllable, absent active detection and response functionality within the Authenticator. This is related to T-1.4.21, but applies more broadly. **Consequences:** May undermine [SG-1], [SG-2], [SG-3], [SG-4], [SG-6], [SG-7], [SG-8], [SG-11], [SG-14], [SG-16]. **Mitigations:** Mitigated by [SM-1], [SM-2], [SM-4], [SM-5], [SM-10], [SM-5], [SM-18], [SM-23], [SM-13], [SM-26], [SM-28].	SG-1, SG-2, SG-3, SG-4, SG-6, SG-7, SG-8, SG-11, SG-14, SG-16

T-1.4.32	Remote Timing Attacks	Violates
AC2, AC5	In this threat, an attacker may be able to extract keys using a timing attack from a remote location. The impact depends on the key or secret recovered. **Consequences:** May undermine [SG-1], [SG-2], [SG-4], [SG-11], [SG-14]. **Mitigations:** This threat is mitigated by the remote timing attack resistance security measure [SM-29].	SG-1, SG-2, SG-4, SG-11, SG-14

7.1.5 Threats to Relying Party

7.1.5.1 Threats to FIDO Server Data

T-2.1.1	FIDO Server DB Read Attack	Violates
	Attacker could obtains read-access to FIDO Server registration database. **Consequences:**Attacker can access all cryptographic key handles and authenticator characteristics associated with a username. If an authenticator or combination of authenticators is unique, they might use this to try to violate [SG-2] Unlinkability. Attacker attempts to perform factorization of public keys by virtue of having access to a large corpus of data, violating [SG-5] Verifier Leak Resiliance and [SG-2] Credential Guessing Resilience. **Mitigations:** [SM-2] Unique Authentication Keys help prevent disclosed key material from being useful against any	SG-2,

T-2.1.1	FIDO Server DB Read Attack	Violates
	other Relying Party, even if successfully attacked. The use of an [SM-6] Cryptographically Secure Verifier Database helps assure that it is infeasible to attack any leaked verifier keys. [SM-9] Authenticator Certification along with [SM-16] should help prevent authenticators with poor entropy from entering the market, reducing the likelihood that even a large corpus of key material will be useful in mounting attacks.	SG-5

T-2.1.2	FIDO Server DB Modification Attack	Violates
AC1	Attacker gains write-access to the FIDO Server registration database. **Consequences:** Violation of [SA-6] The attacker may inject a key registration under its control, violating [SG-1] Strong User Authentication. **Mitigations:** Mitigating such attacks is outside the scope of the FIDO specifications. The Relying Party must maintain the integrity of any information it relies up on to identify a user as part of [SA-6].	SA-6

T-2.2.1	Web App Malware	Violates
	Attacker gains ability to execute code in the security context of the Relying Party web application or FIDO Server. **Consequences:** Attacker is able to violate [SG-1], [SG-10], [SG-9] and any other Relying Party controls. **Mitigations:** The consequences of such an incident are limited to the relationship between the user and that particular Relying Party by [SM-1], [SM-2], and [SM-5]. Even within the Relying Party to user relationship, a user can be protected by [SM-10] Transaction Confirmation if the compromise does not include the users' computing environment.	SG-1, SG-9, SG-10

T-2.2.2	Linking through compromised Relying Party database	Violates
AC1	In this threat, a Relying Party is able to access another Relying Party's database (either because the Relying Parties are collaborating or because of the compromise of another Relying Party's database). The malicious party then sends Key Handles (which may contain a wrapped private key) from the other Relying Party's database in an attempt to link the two separate accounts to the same Authenticator (thus user). **Consequences:** May undermine [SG-1], [SG-4]. **Mitigations:** This threat is mitigated by [SM-1], [SM-2], [SM-5], [SM-23].	SG-1, SG-4

7.1.6 Threats to the Secure Channel between Client and Relying Party

7.1.6.1 Exploiting Weaknesses in the Secure Transport of FIDO Messages

FIDO takes as a base assumption that [SA-3] applications on the user device are able to establish secure channels that provide trustworthy server authentication, and confidentiality and integrity for messages. e.g. through TLS. [T-1.2.4] Discusses some consequences of violations of this assumption due to implementation errors in a browser or client application, but other threats exist in different layers.

T-3.1.1	TLS Proxy	Violates
AC3	The FIDO user device is administratively configured to connect through a proxy that terminates TLS connections. The client trusts this device, but the connection between the user and FIDO server is no longer end-to-end secure. **Consequences:** Any such proxies introduce a new party into the protocol. If this party is untrustworthy, consequences may be as for [T-1.2.4]. **Mitigations:** Mitigations for [T-1.2.4] apply, except that the proxy is considered trusted by the client, so certain methods of [SM-12] Channel Binding may indicate a compromised channel even in the absence of an attack. Servers should use multiple methods and adjust their risk scoring appropriately. A trustworthy client that reports a server certificate that is unknown to the server and does not chain to a public root may indicate a client behind such a proxy. A client reporting a server certificate that is unknown to the server but validates for the server's identity according to commonly used public trust roots is more likely to indicate [T-3.1.2].	SG-11, SG-12, SG-13

T-3.1.2	Fraudulent TLS Server Certificate	Violates
AC3	An attacker is able to obtain control of a certificate credential for a Relying Party, perhaps from a compromised Certification Authority or poor protection practices by the Relying Party. **Consequences:**As for [T-1.2.4]. **Mitigations:**As for [T-1.2.4].	SG-11, SG-12, SG-13

T-3.1.3	Protocol level real-time MITM attack	Violates

T-3.1.3	Protocol level real-time MITM attack	Violates
AC3	An adversary can intercept and manipulate network packets sent from the relying party to the client. The adversary uses this capability to (a) terminate the underlying TLS session from the client at the adversary and to (b) simultaneously use another TLS session from the adversary to the relying party. In the traditional username/password world, this allows the adversary to intercept the username and the password and then successfully impersonate the user at the relying party. **Consequences:** None if FIDO channelBinding [SM-12] or transaction confirmation [SM-10] are used. **Mitigations:** In the case of channelBinding [SM-12], the FIDO server will detect the MITM in the TLS channel by comparing the channel binding information provided by the client and the channel binding information retrieved locally by the server. In the case of transaction confirmation [SM-10], the user verifies and approves a particular transaction. The adversary could modify the transaction before approval. This would lead to rejection by the user. Alternatively, the adversary could modify the transaction after approval. This will break the signature in the transaction confirmation response. The FIDO Server will not accept it as a consequence. HTTP Public Key Pinning (RFC7469) can also be used to mitigate this attack (outside the FIDO stack).	SG-11, SG-12, SG-13

7.1.7 Threats to the Infrastructure

7.1.7.1 Threats to FIDO Authenticator Manufacturers

T-4.1.1	Manufacturer Level Attestation Key Compromise	Violates
AC2	Attacker obtains control of an attestation key or attestation key issuing key. **Consequences:** Same as [T-1.4.6]: Attacker can violate [SG-9] Attestable Properties by creating a malicious hardware or software device that represents itself as a legitimate one. **Mitigations:** Same as [T-1.4.6]: Relying Parties can use [SM-4] Authenticator Status Checking to identify known-compromised keys. Identification of such compromise is outside the strict scope of the FIDO protocols.	SG-9

T-4.1.2	Malicious Authenticator HW	Violates
AC1, AC2, AC3, AC5, AC6	FIDO Authenticator manufacturer relies on hardware or software components that generate weak cryptographic authentication key material or contain backdoors. **Consequences:** Effective violation of [SA-1] in the context of such an Authenticator. **Mitigations:** The process of [SM-9] Authenticator Certification may reveal a subset of such threats, but it is not possible that all such can be revealed with black box testing and white box examination may be is economically infeasible. Users and Relying Parties with special concerns about this class of threat must exercise their own necessary caution about the trustworthiness and verifiability of their vendors and supply chain. [SM-24] builds confidence that an Authenticator is not malicious or poorly implemented.	SA-1

7.1.7.2 Threats to FIDO Server Vendors

T-4.2.1	Vendor Level Trust Anchor Injection Attack	Violates
	Attacker adds malicious trust anchors to the trust list shipped by a FIDO Server vendor. **Consequences:** Attacker can deploy fake Authenticators which Relying Parties cannot detect as such, which do not implement any appropriate security measures, and is able to violate all security goals of FIDO. **Mitigations:** This type of supply chain threat is outside the strict scope of the FIDO protocols and violates [SA-6]. Relying Parties can verify their trust list against the data published by the FIDO Alliance Metadata Service [FIDOMetadataService] (see `https://fidoalliance.org/mds`).	SA-6

7.1.7.3 Threats to FIDO Metadata Service Operators

T-4.3.1	Metadata Service Signing Key Compromise	Violates
	The attacker gets access to the private Metadata TOC signing key. **Consequences:** The attacker could sign invalid Metadata. The attacker could • make trustworthy authenticators look less trustworthy (e.g. by increasing FAR). • make weak authenticators look strong (e.g. by changing the key protection method to a more secure one) • inject malicious attestation trust anchors, e.g. root certificates which cross-signed the original attestation trust anchor and the cross-signed original attestation root certificate. This malicious trust anchors could be used to sign attestation certificates for fraudulent authenticators, e.g. authenticators using the AAID of trustworthy authenticators but not protecting their keys as stated in the metadata. **Mitigations:** The Metadata Service operator should protect the Metadata signing key appropriately, e.g. using a hardware protected key storage. Relying parties could use out-of-band methods to cross-check Metadata Statements with the respective vendors and	SG-9

T-4.3.1	cross-check the revocation state of the Metadata signing key with the provider of the Metadata Service. **Metadata Service Signing Key Compromise**	**Violates**

T-4.3.2	**Metadata Statement Data Injection**	**Violates**
	An attacker injects malicious Authenticator data into the Metadata Statement. **Consequences:** The attacker could make the Metadata Service operator sign invalid Metadata Statements. The attacker could • make trustworthy authenticators look less trustworthy (e.g. by increasing FAR). • make weak authenticators look strong (e.g. by changing the key protection method to a more secure one) • inject malicious attestation trust anchors, e.g. root certificates which cross-signed the original attestation trust anchor and the cross-signed original attestation root certificate. This malicious trust anchors could be used to sign attestation certificates for fraudulent authenticators, e.g. authenticators using the AAID of trustworthy authenticators but not protecting their keys as stated in the metadata. **Mitigations:** The Metadata Service operator could carefully review the delta between the old and the new Metadata Statements. Authenticator vendors could verify the published Metadata Statements related to their Authenticators.	SG-9

7.1.8 Threats Specific to Second Factor Authenicators (UAF / U2F)

T-5.1.1	**Error Status Side Channel**	**Violates**
	Relying parties issues an authentication challenge to an authenticator and can infer from error status if it is already registered. **Consequences:** UAF Silent authenticators / U2F authenticators not requiring user interaction for generating a signed response may be used to track users without their consent by issuing a pre-authentication challenge to them, revealing the identity of an otherwise anonymous user. Users would be identifiable by relying parties without their knowledge, violating [SG-7]. **Mitigations:** The U2F specification recommends that browsers prompt users whether to allow this operation using mechanisms similar to those defined for other privacy sensitive operations like Geolocation.	SG-7

T-5.1.2	**Malicious RP**	**Violates**
AC1	Malicious relying party mounts a cryptographic attack on a key handle it is storing. **Consequences:** If the Relying Party is able to recover the contents of the key handle, it might forge logs of protocol exchanges to associate the user with actions he or she did not perform. If the Relying Party is able to recover the key used to wrap a key handle, that key is likely used for all key handles, and hence might be used to decrypt key handles stored with other Relying Parties and violate [SG-1] Strong User Authentication. **Mitigations:** None. U2F depends on [SA-1] to hold for key wrapping operations.	SG-1

T-5.1.3	**Physical Attack on a User Presence Authenticator**	**Violates**
AC5	Attacker gains physical access to U2F authenticator or a UAF authenticator with only user presence check (e.g., by stealing it). **Consequences:** Same as for [T-1.4.4]. Such authenticators have weak local user verification. If the attacker can guess the username and password/PIN, they can impersonate the user, violating [SG-1] Strong User Authentication. **Mitigations:** Relying Parties can use strong additional factors. Relying Parties should provide users a means to revoke keys associated with a lost device.	SG-1

T-5.1.4	**Physical Attack**	**Violates**
AC2 (associated with shared keys), AC5	In this threat, keys or other sensitive information is read out by directly accessing it from the authenticator that the attacker has physically compromised. **Consequences:** May undermine [SG-1], [SG-4], [SG-11], [SG-14]. Authenticator with user presence check have weak local user verification. If the attacker can guess the username and password/PIN, they can impersonate the user, violating [SG-1] Strong User Authentication. **Mitigations:** Mitigated by resistance to injected faults [SM-18] and [SM-28].	SG-1, SG-4, SG-11, SG-14

7.2 Acknowledgements

We thank iSECpartners for their review of, and contributions to, this document.

A. References

A.1 Informative references

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAA Algorithm*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html

[FIDOMetadataService]
R. Lindemann; B. Hill; D. Baghdasaryan. *FIDO Metadata Service v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-metadata-service-v2.0-ps-20170927.html

[PasswordAuthSchemesKeyIssues]
Chwei-Shyong Tsai; Cheng-Chi Lee; Min-Shiang Hwang. *Password Authentication Schemes: Current Status and Key Issues*. September 2006. URL: http://ijns.femto.com.tw/contents/ijns-v3-n2/ijns-2006-v3-n2-p101-115.pdf

[QuestToReplacePasswords]
Joseph Bonneau; Cormac Herley; Paul C. van Oorschot; Frank Stajano. *The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes*. March 2012. URL: http://research.microsoft.com/pubs/161585/QuestToReplacePasswords.pdf

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[U2FOverview]
S. Srinivas; D. Balfanz; E. Tiffany. *FIDO U2F Overview v1.0*. Draft. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html

[UAFProtocol]
R. Lindemann; D. Baghdasaryan; E. Tiffany; D. Balfanz; B. Hill; J. Hodges. *FIDO UAF Protocol Specification v1.0*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html

FIDO Registry of Predefined Values

FIDO Alliance Proposed Standard 27 September 2017

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-registry-v2.0-ps-20170927.html
Previous version:
https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-registry-v2.0-rd-20161004.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This document defines all the strings and constants reserved by FIDO protocols. The values defined in this document are referenced by various FIDO specifications.

Status of This Document

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

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

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as `code`.

String literals are enclosed in "", e.g. "UAF-TLV".

In formulas we use "|" to denote byte wise concatenation operations.

FIDO specific terminology used in this document is defined in [FIDOGlossary].

Some entries are marked as **"(optional)"** in this spec. The meaning of this is defined in other FIDO specifications referring to this document.

1.1 Conformance

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

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

This document defines the registry of FIDO-specific constants common to multiple FIDO protocol families. It is expected that, over time, new constants will be added to this registry. For example new authentication algorithms and new types of authenticator characteristics will require new constants to be defined for use within the specifications.

3. Authenticator Characteristics

This section is normative.

3.1 User Verification Methods

The `USER_VERIFY` constants are flags in a bitfield represented as a 32 bit long integer. They describe the methods and capabilities of an UAF authenticator for *locally* verifying a user. The operational details of these methods are opaque to the server. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form authenticator policies in UAF protocol messages.

All user verification methods must be performed locally by the authenticator in order to meet FIDO privacy principles.

USER_VERIFY_PRESENCE 0x00000001
: This flag must be set if the authenticator is able to confirm user presence in any fashion. If this flag and no other is set for user verification, the guarantee is only that the authenticator cannot be operated without some human intervention, not necessarily that the sensing of "presence" provides any level of user verification (e.g. a device that requires a button press to activate).

USER_VERIFY_FINGERPRINT 0x00000002
: This flag must be set if the authenticator uses any type of measurement of a fingerprint for user verification.

USER_VERIFY_PASSCODE 0x00000004
: This flag must be set if the authenticator uses a local-only passcode (i.e. a passcode not known by the server) for user verification.

USER_VERIFY_VOICEPRINT 0x00000008
: This flag must be set if the authenticator uses a voiceprint (also known as speaker recognition) for user verification.

USER_VERIFY_FACEPRINT 0x00000010
: This flag must be set if the authenticator uses any manner of face recognition to verify the user.

USER_VERIFY_LOCATION 0x00000020
: This flag must be set if the authenticator uses any form of location sensor or measurement for user verification.

USER_VERIFY_EYEPRINT 0x00000040
: This flag must be set if the authenticator uses any form of eye biometrics for user verification.

USER_VERIFY_PATTERN 0x00000080
: This flag must be set if the authenticator uses a drawn pattern for user verification.

USER_VERIFY_HANDPRINT 0x00000100
: This flag must be set if the authenticator uses any measurement of a full hand (including palm-print, hand geometry or vein geometry) for user verification.

USER_VERIFY_NONE 0x00000200
: This flag must be set if the authenticator will respond without any user interaction (e.g. Silent Authenticator).

USER_VERIFY_ALL 0x00000400
: If an authenticator sets multiple flags for user verification types, itmay also set this flag to indicate that all verification methods will be enforced (e.g. faceprint AND voiceprint). If flags for multiple user verification methods are set and this flag is not set, verification with only one is necessary (e.g. fingerprint OR passcode).

3.2 Key Protection Types

The `KEY_PROTECTION` constants are flags in a bit field represented as a 16 bit long integer. They describe the method an authenticator uses to protect the private key material for FIDO registrations. Refer to [UAFAuthnrCommands] for more details on the relevance of keys and key protection. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form authenticator policies in UAF protocol messages.

When used in metadata describing an authenticator, several of these flags are *exclusive* of others (i.e. can not be combined) - the certified metadata may have at most one of the mutually exclusive bits set to 1. When used in authenticator policy, any bit may be set to 1, e.g. to indicate that a server is willing to accept authenticators using either `KEY_PROTECTION_SOFTWARE` or `KEY_PROTECTION_HARDWARE`.

> NOTE
>
> These flags must be set according to the *effective* security of the keys, in order to follow the assumptions made in [FIDOSecRef]. For example, if a key is stored in a secure element *but* software running on the FIDO User Device could call a function in the secure element to export the key either in the clear or using an arbitrary wrapping key, then the effective security is `KEY_PROTECTION_SOFTWARE` and not `KEY_PROTECTION_SECURE_ELEMENT`.

`KEY_PROTECTION_SOFTWARE 0x0001`
: This flag must be set if the authenticator uses software-based key management. Exclusive in authenticator metadata with `KEY_PROTECTION_HARDWARE`, `KEY_PROTECTION_TEE`, `KEY_PROTECTION_SECURE_ELEMENT`

`KEY_PROTECTION_HARDWARE 0x0002`
: This flag should be set if the authenticator uses hardware-based key management. Exclusive in authenticator metadata with `KEY_PROTECTION_SOFTWARE`

`KEY_PROTECTION_TEE 0x0004`
: This flag should be set if the authenticator uses the Trusted Execution Environment [TEE] for key management. In authenticator metadata, this flag should be set in conjunction with `KEY_PROTECTION_HARDWARE`. Mutually exclusive in authenticator metadata with `KEY_PROTECTION_SOFTWARE`, `KEY_PROTECTION_SECURE_ELEMENT`

`KEY_PROTECTION_SECURE_ELEMENT 0x0008`
: This flag should be set if the authenticator uses a Secure Element [SecureElement] for key management. In authenticator metadata, this flag should be set in conjunction with `KEY_PROTECTION_HARDWARE`. Mutually exclusive in authenticator metadata with `KEY_PROTECTION_TEE`, `KEY_PROTECTION_SOFTWARE`

`KEY_PROTECTION_REMOTE_HANDLE 0x0010`
: This flag must be set if the authenticator does not store (wrapped) UAuth keys at the client, but relies on a server-provided key handle. This flag must be set in conjunction with one of the other `KEY_PROTECTION` flags to indicate how the local key handle wrapping key and operations are protected. Servers may unset this flag in authenticator policy if they are not prepared to store and return key handles, for example, if they have a requirement to respond indistinguishably to authentication attempts against userIDs that do and do not exist. Refer to [UAFProtocol] for more details.

3.3 Matcher Protection Types

The `MATCHER_PROTECTION` constants are flags in a bit field represented as a 16 bit long integer. They describe the method an authenticator uses to protect the matcher that performs user verification. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form authenticator policies in UAF protocol messages. Refer to [UAFAuthnrCommands] for more details on the matcher component.

> NOTE

> These flags must be set according to the *effective* security of the matcher, in order to follow the assumptions made in [FIDOSecRef]. For example, if a passcode based matcher is implemented in a secure element, but the passcode is expected to be provided as unauthenticated parameter, then the effective security is `MATCHER_PROTECTION_SOFTWARE` and not `MATCHER_PROTECTION_ON_CHIP`.

`MATCHER_PROTECTION_SOFTWARE` 0x0001
: This flag must be set if the authenticator's matcher is running in software. Exclusive in authenticator metadata with `MATCHER_PROTECTION_TEE`, `MATCHER_PROTECTION_ON_CHIP`

`MATCHER_PROTECTION_TEE` 0x0002
: This flag should be set if the authenticator's matcher is running inside the Trusted Execution Environment [TEE]. Mutually exclusive in authenticator metadata with `MATCHER_PROTECTION_SOFTWARE`, `MATCHER_PROTECTION_ON_CHIP`

`MATCHER_PROTECTION_ON_CHIP` 0x0004
: This flag should be set if the authenticator's matcher is running on the chip. Mutually exclusive in authenticator metadata with `MATCHER_PROTECTION_TEE`, `MATCHER_PROTECTION_SOFTWARE`

3.4 Authenticator Attachment Hints

The `ATTACHMENT_HINT` constants are flags in a bit field represented as a 32 bit long. They describe the method an authenticator uses to communicate with the FIDO User Device. These constants are reported and queried through the UAF Discovery APIs [UAFAppAPIAndTransport], and used to form Authenticator policies in UAF protocol messages. Because the connection state and topology of an authenticator may be transient, these values are only hints that can be used by server-supplied policy to guide the user experience, e.g. to prefer a device that is connected and ready for authenticating or confirming a low-value transaction, rather than one that is more secure but requires more user effort.

> NOTE
>
> These flags are not a mandatory part of authenticator metadata and, when present, only indicate possible states that may be reported during authenticator discovery.

`ATTACHMENT_HINT_INTERNAL` 0x0001
: This flag may be set to indicate that the authenticator is permanently attached to the FIDO User Device.

  A device such as a smartphone may have authenticator functionality that is able to be used both locally and remotely. In such a case, the FIDO client must filter and exclusively report only the relevant bit during Discovery and when performing policy matching.

  This flag cannot be combined with any other `ATTACHMENT_HINT` flags.

`ATTACHMENT_HINT_EXTERNAL` 0x0002
: This flag may be set to indicate, for a hardware-based authenticator, that it is removable or remote from the FIDO User Device.

  A device such as a smartphone may have authenticator functionality that is able to be used both locally and remotely. In such a case, the FIDO UAF Client must filter and exclusively report only the relevant bit during discovery and when performing policy matching.
  This flag must be combined with one or more other ATTACHMENT_HINT flag(s).

`ATTACHMENT_HINT_WIRED` 0x0004
: This flag may be set to indicate that an external authenticator currently has an exclusive wired connection, e.g. through USB, Firewire or similar, to the FIDO User

Device.

ATTACHMENT_HINT_WIRELESS 0x0008
: This flag may be set to indicate that an external authenticator communicates with the FIDO User Device through a personal area or otherwise non-routed wireless protocol, such as Bluetooth or NFC.

ATTACHMENT_HINT_NFC 0x0010
: This flag may be set to indicate that an external authenticator is able to communicate by NFC to the FIDO User Device. As part of authenticator metadata, or when reporting characteristics through discovery, if this flag is set, the `ATTACHMENT_HINT_WIRELESS` flag should also be set as well.

ATTACHMENT_HINT_BLUETOOTH 0x0020
: This flag may be set to indicate that an external authenticator is able to communicate using Bluetooth with the FIDO User Device. As part of authenticator metadata, or when reporting characteristics through discovery, if this flag is set, the `ATTACHMENT_HINT_WIRELESS` flag should also be set.

ATTACHMENT_HINT_NETWORK 0x0040
: This flag may be set to indicate that the authenticator is connected to the FIDO User Device over a non-exclusive network (e.g. over a TCP/IP LAN or WAN, as opposed to a PAN or point-to-point connection).

ATTACHMENT_HINT_READY 0x0080
: This flag may be set to indicate that an external authenticator is in a "ready" state. This flag is set by the ASM at its discretion.

> NOTE
>
> Generally this should indicate that the device is immediately available to perform user verification without additional actions such as connecting the device or creating a new biometric profile enrollment, but the exact meaning may vary for different types of devices. For example, a USB authenticator may only report itself as ready when it is plugged in, or a Bluetooth authenticator when it is paired and connected, but an NFC-based authenticator may always report itself as ready.

ATTACHMENT_HINT_WIFI_DIRECT 0x0100
: This flag may be set to indicate that an external authenticator is able to communicate using WiFi Direct with the FIDO User Device. As part of authenticator metadata and when reporting characteristics through discovery, if this flag is set, the `ATTACHMENT_HINT_WIRELESS` flag should also be set.

3.5 Transaction Confirmation Display Types

The `TRANSACTION_CONFIRMATION_DISPLAY` constants are flags in a bit field represented as a 16 bit long integer. They describe the availability and implementation of a transaction confirmation display capability required for the transaction confirmation operation. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form authenticator policies in UAF protocol messages. Refer to [UAFAuthnrCommands] for more details on the security aspects of TransactionConfirmation Display.

TRANSACTION_CONFIRMATION_DISPLAY_ANY 0x0001
: This flag must be set to indicate that a transaction confirmation display, of any type, is available on this authenticator. Other `TRANSACTION_CONFIRMATION_DISPLAY` flags may also be set if this flag is set. If the authenticator does not support a transaction confirmation display, then the value of `TRANSACTION_CONFIRMATION_DISPLAY` must be set to 0.

TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE 0x0002
: This flag must be set to indicate, that a software-based transaction confirmation display operating in a privileged context is available on this authenticator.

A FIDO client that is capable of providing this capability may set this bit (in conjunction with TRANSACTION_CONFIRMATION_DISPLAY_ANY) for all authenticators of type ATTACHMENT_HINT_INTERNAL, even if the authoritative metadata for the authenticator does not indicate this capability.

> NOTE
>
> Software based transaction confirmation displays might be implemented within the boundaries of the ASM rather than by the authenticator itself [UAFASM].

This flag is mutually exclusive with TRANSACTION_CONFIRMATION_DISPLAY_TEE and TRANSACTION_CONFIRMATION_DISPLAY_HARDWARE.

TRANSACTION_CONFIRMATION_DISPLAY_TEE 0x0004

This flag should be set to indicate that the authenticator implements a transaction confirmation display in a Trusted Execution Environment ([TEE], [TEESecureDisplay]). This flag is mutually exclusive with TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE and TRANSACTION_CONFIRMATION_DISPLAY_HARDWARE.

TRANSACTION_CONFIRMATION_DISPLAY_HARDWARE 0x0008

This flag should be set to indicate that a transaction confirmation display based on hardware assisted capabilities is available on this authenticator. This flag is mutually exclusive with TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE and TRANSACTION_CONFIRMATION_DISPLAY_TEE.

TRANSACTION_CONFIRMATION_DISPLAY_REMOTE 0x0010

This flag should be set to indicate that the transaction confirmation display is provided on a distinct device from the FIDO User Device. This flag can be combined with any other flag.

3.6 Tags used for crypto algorithms and types

These tags indicate the specific authentication algorithms, public key formats and other crypto relevant data.

3.6.1 Authentication Algorithms

The ALG_SIGN constants are 16 bit long integers indicating the specific signature algorithm and encoding.

> NOTE
>
> FIDO UAF supports RAW and DER signature encodings in order to allow small footprint authenticator implementations.

ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW 0x0001

An ECDSA signature on the NIST secp256r1 curve which must have raw R and S buffers, encoded in big-endian order. This is the signature encoding as specified in [ECDSA-ANSI].

I.e. `[R (32 bytes), S (32 bytes)]`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW
- ALG_KEY_ECC_X962_DER

ALG_SIGN_SECP256R1_ECDSA_SHA256_DER 0x0002

DER [ITU-X690-2008] encoded ECDSA signature [RFC5480] on the NIST secp256r1 curve.

I.e. a DER encoded `SEQUENCE { r INTEGER, s INTEGER }`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW
- ALG_KEY_ECC_X962_DER

`ALG_SIGN_RSASSA_PSS_SHA256_RAW 0x0003`

RSASSA-PSS [RFC3447] signature must have raw S buffers, encoded in big-endian order [RFC4055] [RFC4056]. The default parameters as specified in [RFC4055] must be assumed, i.e.

- Mask Generation Algorithm MGF1 with SHA256
- Salt Length of 32 bytes, i.e. the length of a SHA256 hash value.
- Trailer Field value of 1, which represents the trailer field with hexadecimal value `0xBC`.

I.e. `[ S (256 bytes) ]`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_RSA_2048_RAW
- ALG_KEY_RSA_2048_DER

`ALG_SIGN_RSASSA_PSS_SHA256_DER 0x0004`

DER [ITU-X690-2008] encoded OCTET STRING (not BIT STRING!) containing the RSASSA-PSS [RFC3447] signature [RFC4055] [RFC4056]. The default parameters as specified in [RFC4055] must be assumed, i.e.

- Mask Generation Algorithm MGF1 with SHA256
- Salt Length of 32 bytes, i.e. the length of a SHA256 hash value.
- Trailer Field value of 1, which represents the trailer field with hexadecimal value `0xBC`.

I.e. a DER encoded `OCTET STRING` (including its tag and length bytes).

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_RSA_2048_RAW
- ALG_KEY_RSA_2048_DER

`ALG_SIGN_SECP256K1_ECDSA_SHA256_RAW 0x0005`

An ECDSA signature on the secp256k1 curve which must have raw R and S buffers, encoded in big-endian order.

I.e.`[R (32 bytes), S (32 bytes)]`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW
- ALG_KEY_ECC_X962_DER

`ALG_SIGN_SECP256K1_ECDSA_SHA256_DER 0x0006`
DER [ITU-X690-2008] encoded ECDSA signature [RFC5480] on the secp256k1 curve.

I.e. a DER encoded `SEQUENCE { r INTEGER, s INTEGER }`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW
- ALG_KEY_ECC_X962_DER

`ALG_SIGN_SM2_SM3_RAW 0x0007` **(optional)**
Chinese SM2 elliptic curve based signature algorithm combined with SM3 hash algorithm [OSCCA-SM2][OSCCA-SM3]. We use the 256bit curve [OSCCA-SM2-curve-param].

This algorithm is suitable for authenticators using the following key representation format: ALG_KEY_ECC_X962_RAW.

`ALG_SIGN_RSA_EMSA_PKCS1_SHA256_RAW 0x0008`
This is the EMSA-PKCS1-v1_5 signature as defined in [RFC3447]. This means that the encoded message EM will be the input to the cryptographic signing algorithm RSASP1 as defined in [RFC3447]. The result s of RSASP1 is then encoded using function I2OSP to produce the raw signature octets.

- `EM = 0x00 | 0x01 | PS | 0x00 | T`
- with the padding string PS with length=emLen - tLen - 3 octets having the value 0xff for each octet, e.g. `(0x) ff ff ff ff ff ff ff ff`
- with the DER [ITU-X690-2008] encoded DigestInfo value T: `(0x)30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20 | H`, where H denotes the bytes of the SHA256 hash value.

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_RSA_2048_RAW
- ALG_KEY_RSA_2048_DER

> NOTE
>
> Implementers should verify that their implementation of the PKCS#1 V1.5 signature follows the recommendations in [RFC3218] to protect against adaptive chosen-ciphertext attacks such as Bleichenbacher.

`ALG_SIGN_RSA_EMSA_PKCS1_SHA256_DER 0x0009`
DER [ITU-X690-2008] encoded OCTET STRING (not BIT STRING!) containing the EMSA-PKCS1-v1_5 signature as defined in [RFC3447]. This means that the encoded message EM will be the input to the cryptographic signing algorithm RSASP1 as defined in [RFC3447]. The result s of RSASP1 is then encoded using function I2OSP to produce the raw signature. The raw signature is DER [ITU-X690-2008] encoded as an OCTET STRING to produce the final signature octets.

- `EM = 0x00 | 0x01 | PS | 0x00 | T`
- with the padding string PS with length=emLen - tLen - 3 octets having the value 0xff for each octet, e.g. `(0x) ff ff ff ff ff ff ff ff`
- with the DER encoded DigestInfo value T: `(0x)30 31 30 0d 06 09 60 86 48 01`

`65 03 04 02 01 05 00 04 20 | H`, where H denotes the bytes of the SHA256 hash value.

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_RSA_2048_RAW
- ALG_KEY_RSA_2048_DER

> NOTE
>
> Implementers should verify that their implementation of the PKCS#1 V1.5 signature follows the recommendations in [RFC3218] to protect against adaptive chosen-ciphertext attacks such as Bleichenbacher.

3.6.2 Public Key Representation Formats

The `ALG_KEY` constants are 16 bit long integers indicating the specific Public Key algorithm and encoding.

> NOTE
>
> FIDO UAF supports RAW and DER encodings in order to allow small footprint authenticator implementations. By definition, the authenticator must encode the public key as part of the registration assertion.

ALG_KEY_ECC_X962_RAW 0x0100

Raw ANSI X9.62 formatted Elliptic Curve public key [SEC1].

I.e. `[0x04, X (32 bytes), Y (32 bytes)]`. Where the byte `0x04` denotes the uncompressed point compression method.

ALG_KEY_ECC_X962_DER 0x0101

DER [ITU-X690-2008] encoded ANSI X.9.62 formatted `SubjectPublicKeyInfo` [RFC5480] specifying an elliptic curve public key.

I.e. a DER encoded `SubjectPublicKeyInfo` as defined in [RFC5480].

Authenticator implementations must generate `namedCurve` in the `ECParameters` object which is included in the `AlgorithmIdentifier`. A FIDO UAF Server must accept `namedCurve` in the `ECParameters` object which is included in the `AlgorithmIdentifier`.

ALG_KEY_RSA_2048_RAW 0x0102

Raw encoded 2048-bit RSA public key [RFC3447].

That is, `[n (256 bytes), e (N-256 bytes)]`. Where `N` is the total length of the field.

This total length should be taken from the object containing this key, e.g. the TLV encoded field.

ALG_KEY_RSA_2048_DER 0x0103

ASN.1 DER [ITU-X690-2008] encoded 2048-bit RSA [RFC3447] public key [RFC4055].

That is a DER encoded `SEQUENCE { n INTEGER, e INTEGER }`.

ALG_KEY_COSE 0x0104

COSE_Key format, as defined in Section 7 of [RFC8152]. This encoding includes its own field for indicating the public key algorithm.

A. References

A.1 Normative references

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html

[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). November 2008. URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en

[OSCCA-SM2]
SM2: Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves: Part 1: General. December 2010. URL: http://www.oscca.gov.cn/UpFile/2010122214822692.pdf

[OSCCA-SM2-curve-param]
SM2: Elliptic Curve Public-Key Cryptography Algorithm: Recommended Curve Parameters. December 2010. URL: http://www.oscca.gov.cn/UpFile/2010122214836668.pdf

[OSCCA-SM3]
SM3 Cryptographic Hash Algorithm. December 2010. URL: http://www.oscca.gov.cn/UpFile/20101222141857786.pdf

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3447]
J. Jonsson; B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. February 2003. Informational. URL: https://tools.ietf.org/html/rfc3447

[RFC4055]
J. Schaad; B. Kaliski; R. Housley. *Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. June 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc4055

[RFC4056]
J. Schaad. *Use of the RSASSA-PSS Signature Algorithm in Cryptographic Message Syntax (CMS)*. June 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc4056

[RFC5480]
S. Turner; D. Brown; K. Yiu; R. Housley; T. Polk. *Elliptic Curve Cryptography Subject Public Key Information*. March 2009. Proposed Standard. URL: https://tools.ietf.org/html/rfc5480

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

[SEC1]
SEC1: Elliptic Curve Cryptography, Version 2.0. September 2000. URL: http://secg.org/download/aid-780/sec1-v2.pdf

A.2 Informative references

[ECDSA-ANSI]
Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62-2005. November 2005. URL: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005

[FIDOSecRef]
R. Lindemann; D. Baghdasaryan; B. Hill. *FIDO Security Reference*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-security-ref-v2.0-

ps-20170927.html
[RFC3218]
E. Rescorla. *Preventing the Million Message Attack on Cryptographic Message Syntax*. January 2002. Informational. URL:https://tools.ietf.org/html/rfc3218
[SecureElement]
GlobalPlatform Card Specifications. URL: https://www.globalplatform.org/specifications.asp
[TEE]
GlobalPlatform Trusted Execution Environment Specifications. URL: https://www.globalplatform.org/specifications.asp
[TEESecureDisplay]
GlobalPlatform Trusted User Interface API Specifications. URL: https://www.globalplatform.org/specifications.asp
[UAFASM]
D. Baghdasaryan; J. Kemp; R. Lindemann; B. Hill; R. Sasson. *FIDO UAF Authenticator-Specific Module API*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-asm-api-v1.2-rd-20171128.html
[UAFAppAPIAndTransport]
B. Hill; D. Baghdasaryan; B. Blanke. *FIDO UAF Application API and Transport Binding Specification*. Implementation Draft. URL:https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-client-api-transport-v1.2-rd-20171128.html
[UAFAuthnrCommands]
D. Baghdasaryan; J. Kemp; R. Lindemann; R. Sasson; B. Hill. *FIDO UAF Authenticator Commands v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-authnr-cmds-v1.2-rd-20171128.html
[UAFProtocol]
R. Lindemann; D. Baghdasaryan; E. Tiffany; D. Balfanz; B. Hill; J. Hodges. *FIDO UAF Protocol Specification v1.0*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html

FIDO Technical Glossary

FIDO Alliance Proposed Standard 27 September 2017

This version:
: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html

Previous version:
: https://fidoalliance.org/specs/fido-v2.0-rd-20161004/fido-glossary-v2.0-rd-20161004.html

Editor:
: Rolf Lindemann, Nok Nok Labs, Inc.

Contributors:
: Davit Baghdasaryan, Nok Nok Labs, Inc.
: Brad Hill, PayPal
: Jeff Hodges, PayPal

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This document defines all the strings and constants reserved by UAF protocols. The values defined in this document are referenced by various UAF specifications.

Status of This Document

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

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

Implementation of certain elements of this Specification may require licenses under third

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

Type names, attribute names and element names are written as `code`.

String literals are enclosed in "", e.g. "UAF-TLV".

In formulas we use "|" to denote byte wise concatenation operations.

1.1 Key Words

The key words "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this document are to be interpreted as described in [RFC2119].

2. Introduction

This document is the FIDO Alliance glossary of normative technical terms.

This document is not an exhaustive compendium of all FIDO technical terminology because the FIDO terminology is built upon existing terminology. Thus many terms that are commonly used within this context are not listed. They may be found in the glossaries/documents/specifications referenced in the bibliography. Terms defined here that are not attributed to other glossaries/documents/specifications are being defined here.

This glossary is expected to evolve along with theFIDO Alliance specifications and documents.

3. Definitions

AAID

Authenticator Attestation ID. See Attestation ID.

Application

A set of functionality provided by a common entity (the application owner, aka the Relying Party), and perceived by the user as belonging together.

Application Facet

An (application) facet is how an application is implemented on various platforms. For example, the application MyBank may have an Android app, an iOS app, and a Web app. These are all facets of the MyBank application.

Application Facet ID

A platform-specific identifier (URI) for an application facet.

- For Web applications, the facet id is the RFC6454 origin [RFC6454].
- For Android applications, the facet id is the URI android:apk-key-hash:*<hash-of-apk-signing-cert>*
- For iOS, the facet id is the URI ios:bundle-id:*<ios-bundle-id-of-app>*

AppID

The AppID is an identifier for a set of different Facets of a relying party's application. The AppID is a URL pointing to the TrustedFacets, i.e. list of FacetIDs related to this AppID.

Attestation

In the FIDO context, attestation is how Authenticators make claims to a Relying Party that the keys they generate, and/or certain measurements they report, originate from genuine devices with certified characteristics.

Attestation Certificate

A public key certificate related to an Attestation Key.

Authenticator Attestation ID / AAID

A unique identifier assigned to a model, class or batch of FIDO Authenticators that all share the same characteristics, and which a Relying Party can use to look up an Attestation Public Key and Authenticator Metadata for the device.

Attestation [Public / Private] Key

A key used for FIDO Authenticator attestation.

Attestation Root Certificate

A root certificate explicitly trusted by the FIDO Alliance, to which Attestation Certificates chain to.

Authentication

Authentication is the process in which user employs their FIDO Authenticator to prove possession of a registered key to a relying party.

Authentication Algorithm

The combination of signature and hash algorithms used for authenticator-to-relying party authentication.

Authentication Scheme

The combination of an Authentication Algorithm with a message syntax or framing that is used by an Authenticator when constructing a response.

Authenticator, Authnr

See FIDO Authenticator.

Authenticator, 1stF / First Factor

A FIDO Authenticator that transactionally provides a username and at least two authentication factors: cryptographic key material (something you have) plus user verification (something you know / something you are) and so can be used by itself to complete an authentication.

It is assumed that these authenticators have an internal matcher. The matcher is able to verify an already enrolled user. If there is more than one user enrolled – the matcher is also able to identify the right user.

Examples of such authenticator is a biometric sensor or a PIN based verification. Authenticators which only verify presence, such as a physical button, or perform no verification at all, cannot act as a first-factor authenticator.

Authenticator, 2ndF / Second Factor

A FIDO Authenticator which acts only as a second factor. Second-factor authenticators always require a single key handle to be provided before responding to a `Sign` command. They might or might not have a user verification method. It is assumed that these authenticators may or may not have an internal matcher.

Authenticator Attestation

The process of communicating a cryptographic assertion to a relying party that a key presented during authenticator registration was created and protected by a genuine authenticator with verified characteristics.

Authenticator Metadata

Verified information about the characteristics of a certified authenticator, associated with an AAID and available from the FIDO Alliance. FIDO Servers are expected to have access to up-to-date metadata to be able to interact with a given authenticator.

Authenticator Policy

A JSON data structure that allows a relying party to communicate to a FIDO Client the capabilities or specific authenticators that are allowed or disallowed for use in a given operation.

ASM / Authenticator Specific Module

Software associated with a FIDO Authenticator that provides a uniform interface between the hardware and FIDO Client software.

AV

ASM Version

Bound Authenticator

A FIDO Authenticator or combination of authenticator and ASM, which uses an access control mechanism to restrict the use of registered keys to trusted FIDO Clients and/or trusted FIDO User Devices. Compare to a *Roaming Authenticator*.

Certificate

An X.509v3 certificate defined by the profile specified in [RFC5280] and its successors.

Channel Binding

See: [RFC5056], [RFC5929] and [ChannelID]. A channel binding allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication to the higher layer to the channel at the lower layer.

Client

This term is used “in context”, and may refer to a FIDO UAF Client or some other type of client, e.g. a TLS client. See FIDO Client.

Confused Deputy Problem

A confused deputy is a computer program that is innocently fooled by some other party into misusing its authority. It is a specific type of privilege escalation.

Correlation Handle

Any piece of information that may allow, in the context of FIDO protocols, implicit or explicit association and or attribution of multiple actions, believed by the user to be distinct and unrelated, back to a single unique entity. An example of a correlation handle outside of the FIDO context is a client certificate used in traditional TLS mutual authentication: because it sends the same data to multiple Relying Parties, they can therefore collude to uniquely identify and track the user across unrelated activities. [AnonTerminology]

Deregistration

A phase of a FIDO protocol in which a Relying Party tells a FIDO Authenticator to forget a specified piece of (or all) locally managed key material associated with a specific Relying Party account, in case such keys are no longer considered valid by the Relying Party.

Discovery

A phase of a FIDO protocol in which a Relying Party is able to determine the availability of FIDO capabilities at the client’s device, including metadata about the available authenticators.

E(K,D)

Denotes the Encryption of data D with key K

ECDAA

Elliptic Curve based Direct Anonymous Attestation. ECDAA is an attestation scheme alternative to FIDO Basic Attestation. It is an improved Direct Anonymous Attestation scheme based on elliptic curves and bilinear pairings. Direct Anonymous Attestation schemes use individual private keys in the Authenticator while avoiding global correlation handles. ECDAA provides significantly improved performance compared with the original DAA scheme. FIDO ECDAA [FIDOEcdaaAlgorithm] defines object encodings, pairing friendly curves etc. in order to lead to interoperable ECDAA implementations across different FIDO Servers and FIDO Authenticators.

ECDSA

Elliptic Curve Digital Signature Algorithm, as defined by ANSI X9.62 [ECDSA-ANSI].

Enrollment

The process of making a user known to an authenticator. This might be a biometric enrollment as defined in [NSTCBiometrics] or involve processes such as taking ownership of, and setting a PIN or password for, a non-biometric cryptographic storage device. Enrollment may happen as part of a FIDO protocol ceremony, or it may happen outside of the FIDO context for multi-purpose authenticators.

Facet

See Application Facet

Facet ID

See Application Facet ID

FIDO Authenticator

An authentication entity that meets the FIDO Alliance's requirements and which has related metadata.

A FIDO Authenticator is responsible for user verification, and maintaining the cryptographic material required for the relying party authentication.

It is important to note that a FIDO Authenticator is only considered such for, and in relation to, its participation in FIDO Alliance protocols. Because the FIDO Alliance aims to utilize a diversity of existing and future hardware, many devices used for FIDO may have other primary or secondary uses. To the extent that a device is used for non-FIDO purposes such as local operating system login or network login with non-FIDO protocols, it is not considered a FIDO Authenticator and its operation in such modes is *not* subject to FIDO Alliance guidelines or restrictions, including those related to security and privacy.

A FIDO Authenticator may be referred to as simply an authenticator or abbreviated as "authnr". Important distinctions in an authenticator's capabilities and user experience may be experienced depending on whether it is a roaming or bound authenticator, and whether it is a first-factor, or second-factor authenticator.

It is assumed by registration assertion schemes that the authenticator has exclusive control over the data being signed by the attestation key.

Authenticators specify in the Metadata Statement whether they have exclusive control over the data being signed by the `Uauth key`.

FIDO Client

This is the software entity processing the UAF or U2F protocol messages on the FIDO User Device. FIDO Clients may take one of two forms:

- A software component implemented in a user agent (either web browser or native application).
- A standalone piece of software shared by several user agents. (web browsers or native applications).

FIDO Data / FIDO Information

Any information gathered or created as part of completing a FIDO transaction. This includes but is not limited to, biometric measurements of or reference data for the user and FIDO transaction history.

FIDO Server

Server software typically deployed in the relying party's infrastructure that meets UAF protocol server requirements.

FIDO UAF Client

See FIDO Client

See FIDO Client.

FIDO User Device

The computing device where the FIDO Client operates, and from which the user initiates an action that utilizes FIDO.

Key Identifier (KeyID)

The KeyID is an opaque identifier for a key registered by an authenticator with a FIDO Server, for first-factor authenticators. It is used in concert with an AAID to identify a particular authenticator that holds the necessary key. Thus key identifiers must be unique within the scope of an AAID.

One possible implementation is that the KeyID is the SHA256 hash of the `KeyHandle` managed by the ASM.

KeyHandle

A key container created by a FIDO Authenticator, containing a private key and (optionally) other data (such as Username). A key handle may be wrapped (encrypted with a key known only to the authenticator) or unwrapped. In the unwrapped form it is referred to as a *raw key handle*. Second-factor authenticators must retrieve their key handles from the relying party to function. First-factor authenticators manage the storage of their own key handles, either internally (for roaming authenticators) or via the associated ASM (for bound authenticators).

Key Registration

The process of securely establishing a key between FIDO Server and FIDO Authenticator.

KeyRegistrationData (KRD)

A `KeyRegistrationData` object is created and returned by an authenticator as the result of the authenticator's `Register` command. The KRD object contains items such as the authenticator's AAID, the newly generated UAuth.pub key, as well as other authenticator-specific information such as algorithms used by the authenticator for performing cryptographic operations, and counter values. The KRD object is signed using the authenticator's attestation private key.

KHAccessToken

A secret value that acts as a guard for authenticator commands. KHAccessTokens are generated and provided by an ASM.

Matcher

A component of a FIDO Authenticator which is able to perform (local) user verification, e.g. biometric comparison [ISOBiometrics], PIN verification, etc.

Matcher Protections

The security mechanisms that an authenticator may use to protect the matcher component.

Persona

All relevant data stored in an authenticator (e.g. cryptographic keys) are related to a single "persona" (e.g. “business” or “personal” persona). Some administrative interface (not standardized by FIDO) provided by the authenticator may allow maintenance and switching of personas.

The user can switch to the “Personal” Persona and register new accounts. After switching back to the “Business” Persona, these accounts will not be recognized by the authenticator (until the User switches back to “Personal” Persona again).

This mechanism may be used to provide an additional measure of privacy to the user, where the user wishes to use the same authenticator in multiple contexts, without allowing correlation via the authenticator across those contexts.

PersonaID

An identifier provided by an ASM, PersonaID is used to associate different registrations. It can be used to create virtual identities on a single authenticator, for example to differentiate “personal” and “business” accounts. PersonaIDs can be used to manage privacy settings on the authenticator.

Reference Data

A (biometric) reference data (also called template) is a digital reference of distinct characteristics that have been extracted from a biometric sample. Biometric reference data is used during the biometric user verification process [ISOBiometrics]. Non-biometric reference data is used in conjunction with PIN-based user verification.

Registration

A FIDO protocol operation in which a user generates and associates new key material with an account at the Relying Party, subject to policy set by the server, and acceptable attestation that the authenticator and registration matches that policy.

Registration Scheme

The registration scheme defines how the authentication key is being exchanged between the FIDO Server and the FIDO Authenticator.

Relying Party

A web site or other entity that uses a FIDO protocol to directly authenticate users (i.e., performs peer-entity authentication). Note that if FIDO is composed with federated identity management protocols (e.g., SAML, OpenID Connect, etc.), the identity provider will also be playing the role of a FIDO Relying Party.

Roaming Authenticator

A FIDO Authenticator configured to move between different FIDO Clients and FIDO User Devices lacking an established trust relationship by:

1. Using only its own internal storage for registrations
2. Allowing registered keys to be employed without access control mechanisms at the API layer. (Roaming authenticators still may perform user verification.)

Compare to Bound Authenticator.

S(K, D)

Signing of data D with key K

Server Challenge

A random value provided by the FIDO Server in the UAF protocol requests.

Sign Counter

A monotonically increasing counter maintained by the Authenticator. It is increased on every use of the UAuth.priv key. This value can be used by the FIDO Server to detect cloned authenticators.

SignedData

A `SignedData` object is created and returned by an authenticator as the result of the authenticator's `Sign` command. The to-be-signed data input to the signature operation is represented in the returned SignedData object as intact values or as hashed values. The SignedData object also contains general information about the authenticator and its mode, a nonce, information about authenticator-specific cryptographic algorithms, and a use counter. The `SignedData` object is signed using a relying party-specific UAuth.priv key.

Silent Authenticator

FIDO Authenticator that does not prompt the user or perform any user verification.

Step-up Authentication

An authentication which is performed on top of an already authenticated session.

Example: The user authenticates the session initially using a username and password, and the web site later requests a FIDO authentication on top of this authenticated session.

One reason for requesting step-up authenication could be a request for a high value resource.

FIDO U2F is always used as a step-up authentication. FIDO UAF could be used as step-up authentication, but it could also be used as an initial authentication mechanism.

Note: In general, there is no implication that the step-up authentication method itself is "stronger" than the initial authentication. Since the step-up authentication is performed on top of an existing authentication, the resulting combined authentication strength will increase most likely, but it will never decrease.

Template

See reference data.

Test of User Presence

See User Presence Check

TLS

Transport Layer Security

Token

In FIDO U2F, the term Token is often used to mean what is called an authenticator in UAF. Also, note that other uses of “token”, e.g. KHAccessToken, User Verification Token, etc., are separately distinct. If they are not explicitly defined, their meaning needs to be determined from context.

Transaction Confirmation

An operation in the FIDO protocol that allows a relying party to request that a FIDO Client, and authenticator with the appropriate capabilities, display some information to the user, request that the user authenticate locally to their FIDO Authenticator to confirm the information, and provide proof-of-possession of previously registered key material and an attestation of the confirmation back to the relying party.

Transaction Confirmation Display

This is a feature of FIDO Authenticators able to show content of a message to a user, and protect the integrity of this message. It could be implemented using the GlobalPlatform specified TrustedUI [TEESecureDisplay].

TrustedFacets

The data structure holding a list of trusted FacetIDs. The AppID is used to retrieve this data structure.

TTEXT

Transaction Text, i.e. text to be confirmed in the case of transaction confirmation.

Type-length-value/tag-length-value (TLV)

A mechanism for encoding data such that the type, length and value of the data are given. Typically, the type and length data fields are of a fixed size. This format offers some advantages over other data encoding mechanisms, that make it suitable for some of the FIDO UAF protocols.

Universal Second Factor (U2F)

The FIDO protocol and family of authenticators which enable a cloud service to offer its users the options of using an easy–to–use, strongly–secure open standards–based second-factor device for authentication. The protocol relies on the server to know the (expected) user before triggering the authentication.

Universal Authentication Framework (UAF)

. The FIDO Protocol and family of authenticators which enable a service to offer its users flexible and interoperable authentication. This protocol allows triggering the authentication before the server knows the user.

UAF Client

See FIDO Client.

UAuth.pub / UAuth.priv / UAuth.key

User authentication keys generated by FIDO Authenticator. UAuth.pub is the public part of key pair. UAuth.priv is the private part of the key. UAuth.key is the more generic notation to refer to UAuth.priv.

UINT8

An 8 bit (1 byte) unsigned integer.

UINT16

A 16 bit (2 bytes) unsigned integer.

UINT32

A 32 bit (4 bytes) unsigned integer.

UPV

UAF Protocol Version

User

Relying party's user, and owner of the FIDO Authenticator.

User Agent

The user agent is a client application that is acting on behalf of a user in a client-server system. Examples of user agents include web browsers and mobile apps.

User Presence Check

The User Presence check in the authenticator verifies that some user is present at the authenticator and agrees with a generic authentication operation.

User Verification

The process by which a FIDO Authenticator locally authorizes use of key material, for example through a touch, pin code, fingerprint match or other biometric.

User Verification Token

The user verification token is generated by Authenticator and handed to the ASM after successful user verification. Without having this token, the ASM cannot invoke special commands such as `Register` or `Sign`.

The lifecycle of the user verification token is managed by the authenticator. The concrete techniques for generating such a token and managing its lifecycle are vendor-specific and non-normative.

Username

A human-readable string identifying a user's account at a relying party.

Verification Factor

The specific means by which local user verification is accomplished. e.g. fingerprint, voiceprint, or PIN.

This is also known as modality.

Web Application, Client-Side

The portion of a relying party application built on the "Open Web Platform" which executes in the context of the user agent. When the term "Web Application" appears unqualified or without specific context in FIDO documents, it generally refers to either the client-side portion or the combination of both client-side and server-side pieces of such an application.

Web Application, Server-Side

The portion of a relying party application that executes on the web server, and responds to HTTP requests. When the term "Web Application" appears unqualified or without specific context in FIDO documents, it generally refers to either the client-side portion or the combination of both client-side and server-side pieces of such an application.

A. References

A.1 Normative references

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAA Algorithm*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

A.2 Informative references

[AnonTerminology]

A. Pfitzmann; M. Hansen. *Anonymity, Unlinkability, Unobservability, Pseudonymity, and Identity Management - A Consolidated Proposal for Terminology, Version 0.34*. August 2010. URL: http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf

[ChannelID]
D. Balfanz. *Transport Layer Security (TLS) Channel IDs*. Work In Progress. URL: http://tools.ietf.org/html/draft-balfanz-tls-channelid

[ECDSA-ANSI]
Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62-2005. November 2005. URL: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005

[ISOBiometrics]
ISO/IEC 2382-37 Harmonized Biometric Vocabulary. 15 December 2012. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194_ISOIEC_2382-37_2012.zip

[NSTCBiometrics]
Biometrics Glossary. 14 September 2006. URL: http://biometrics.gov/Documents/Glossary.pdf

[RFC5056]
N. Williams. *On the Use of Channel Bindings to Secure Channels (RFC 5056)* November 2007. URL: http://www.ietf.org/rfc/rfc5056.txt

[RFC5280]
D. Cooper; S. Santesson; S. Farrell; S.Boeyen; R. Housley; W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. URL: http://www.ietf.org/rfc/rfc5280.txt

[RFC5929]
J. Altman; N. Williams; L. Zhu. *Channel Bindings for TLS (RFC 5929)*. July 2010. URL: http://www.ietf.org/rfc/rfc5929.txt

[RFC6454]
A. Barth. *The Web Origin Concept (RFC 6454)*. June 2011. URL: http://www.ietf.org/rfc/rfc6454.txt

[TEESecureDisplay]
GlobalPlatform Trusted User Interface API Specifications. URL: https://www.globalplatform.org/specifications.asp