UAF Authenticator Commands

Specification Set: fido-uaf-v1.0-rd-20140209 REVIEW DRAFT

Editors:

Davit Baghdasaryan, NokNok Labs
John Kemp, FIDO Alliance

Contributors:

Rolf Lindemann, NokNok Labs
Roni Sasson, Discretix

Abstract:

UAF Authenticators may take different forms. Implementations may range from a secure application running inside tamper-resistant hardware to software-only solutions on consumer devices. Independent of its form, the implementation must follow all the normative notes stated in this document.

This document has two goals:

1) Define the normative aspects of Authenticator implementations

2) Propose a common, non-normative set of commands implementing UAF functionality

Status:

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

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

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

Table of Contents

1 Terminology

Type names, attribute names and element names are written in *italics*.

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].

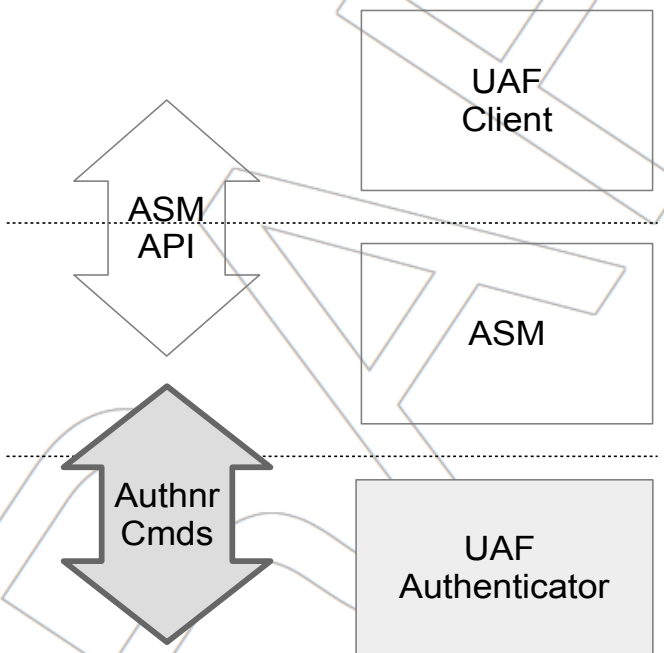
1.1 Key Words

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 Overview

This document specifies low-level functionality which UAF Authenticators should implement in order to support the UAF protocol. While implementing the exact set of commands specified in this document is not mandatory, implementors must follow all the normative notes that apply regardless of the exact commands in use.

The overall architecture of the UAF protocol and its various operations is described in [UAFProtocol]. The following simplified architecture diagram illustrates the interactions and actors this document is concerned with:



2.1 Additonal Notations

- All data described in this document MUST be encoded in **little-endian** format.

- The following is an example of a buffer presented in the form of a table. It reads as follows:
  - A variable length buffer which has a 2-byte Tag in the beginning with a value of TAG_KHANDLE_LIST.
  - The tag is followed by 2 bytes of overall Length
  - What follows is a list of KeyHandles. The sum of all KeyHandle sizes should be equal to the value of Length.
    - Length = (N * sizeof(UINT16)) + (N * KeyHandleSize) where N is the number of KeyHandles

1.	UINT16	Tag	TAG_KHANDLE_LIST
1.1	UINT16	Length	Entire Length of list of KeyHandles
1.2	UINT16	Size	Each entry is a KeyHandle with length and content. This is the length part.
1.3	BYTE	KeyHandle[Size]	Each entry is a KeyHandle with length and content. This is the content part.

- Details of commands are described using a pseudo-code based on Javascript syntax.

3 UAF Authenticator

The UAF Authenticator is an authentication component that meets the UAF protocol requirements as described in [UAFProtocol]. The main functions to be provided by UAF Authenticators are:

1. [Mandatory] Verifying the User using the verification mechanism built into the authenticator. Note that the verification technology can vary from biometric verification to simply verifying physical presence, or no user verification at all.
2. [Mandatory] Performing the cryptographic operations defined in [UAFProtocol]
3. [Mandatory] Attesting itself to the FIDO Server if there is a built-in support for attestation
4. [Optional] Securely displaying the transaction content to the User

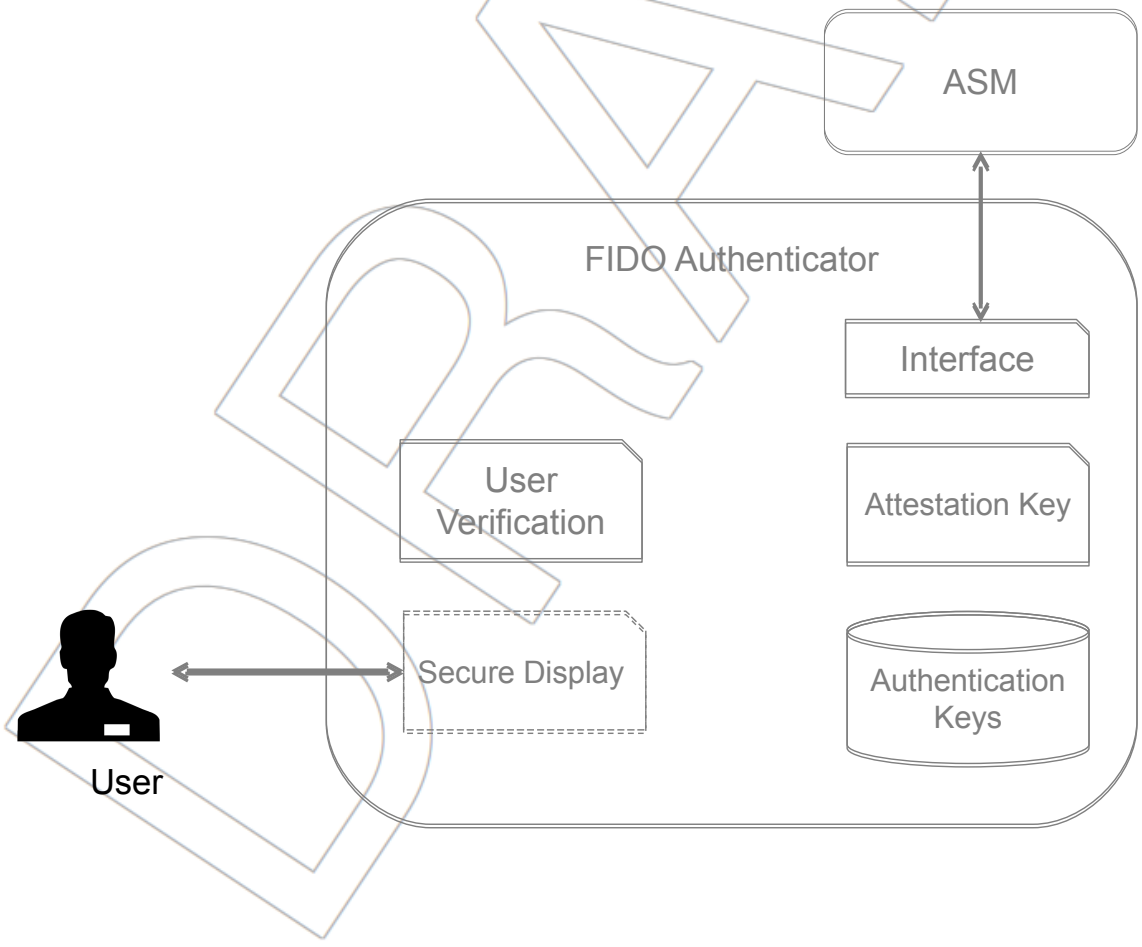


Figure 1: FIDO Authenticator Logical Sub-Components

Some examples of UAF Authenticators:

- A fingerprint sensor built into a mobile device
- PIN authenticator implemented inside a secure element
- A mobile phone acting as an authenticator to a different device
- A USB token with built-in user presence verification
- A voice or face verification technology built into a device

3.1 Types of Authenticators

There are three types of authenticators defined in this document.

- First Factor (1stF) Bound Authenticator - Bound Authenticator with first-factor capability
  - It is assumed that these authenticators MAY or MAY NOT store RawKeyHandles in their own internal storage. If they don't - they return KeyHandle to ASM [UAFASM] and the latter needs to store it in its local database.
  - These authenticators MAY also work as a second factor..
  - Examples
    - A fingerprint sensor built into a laptop, phone, tablet, etc
    - Embedded Secure Element in a mobile device
    - Voice verification built into a device

- First Factor Roaming Authenticator - Roaming Authenticator with first-factor capability
  - 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 can also identify a user.
  - It is assumed that these authenticators are designed to store RawKeyHandles in their own internal secure storage and don't provide these to the ASM.
  - Note that in some deployments, Bound Authenticators can act as Roaming Authenticators. When this happens such an Authenticator MUST follow the requirements applying to Bound Authenticators within the bound-

ary of the system the authenticator is bound to, and follow the requirements that apply to Roaming Authenticators in any system it connects to externally.
    - These authenticators MAY also work as a second factor.
    - Examples
        - A Bluetooth LE based hardware token with built-in fingerprint sensor
        - PIN protected USB hardware token
        - A first-factor Bound Authenticator acting as a roaming authenticator to a different device on user's behalf

- Second Factor (2ndF) Authnenticator - Internal or Roaming Authenticator with second-factor capability
    - It is assumed that these authenticators MAY or MAY NOT store RawKeyHandles in their own internal storage. If they don't - they return KeyHandle to ASM and the latter needs to store it in its local database.
    - These authenticators can only work as second factors.
    - Examples
        - USB dongle with a built-in capacitive touch device for verifying user presence
        - A "Trustlet" application running on the trusted execution environment of a mobile phone, and leveraging a secure keyboard to verify user presence

Throughout the document there will be special conditions applying to some of these types of authenticators.

4 Tags

Informative Notes:

In this document UAF Authenticators use TLV (Type Length Value) format to communicate with outside world. All requests and response data are encoded via TLV.

Commands and existing predefined TAGs can be extended with appending other TAGs (custom or predefined).

Refer to [FIDORegistry] for information about predefined TAGs.

TAG values defined in this section are custom and non-normative.

TLV formatted data has the following simple structure:

2 bytes	2 bytes	“Length” bytes
Tag	Length	Data

Name	Value	Description
TAG_KEYHANDLE	0x1001	Represents KeyHandle. Refer to [FIDOGlossary] for more information about KeyHandle.
TAG_KHANDLE_LIST	0x1002	Represents a list of KeyHandles.
TAG_UNAME_LIST	0x1003	Represents a list of Usernames. Refer to [FIDOGlossary] for more information about username.
TAG_USERVERIFY_TOKEN	0x1004	Represents a User Verification Token. Refer to [FIDOGlossary] for more information about user verification tokens.
TAG_FULL_APPID	0x1005	Represents a full AppID. Refer to [FIDOGlossary] for more information about AppID.
TAG_KEYID	0x1006	Represents a generated KeyID.

Table 4.1: Non-normative TAGs used in this document only (0x1000 – 0x10FF)

Name	Value	Description
TAG_UAFV1_GETINFO_CMD	0x2001	Tag for GetInfo command.
TAG_UAFV1_GETINFO_CMD_RESP	0x2101	Tag for GetInfo command response.
TAG_UAFV1_REG_CMD	0x2002	Tag for Register command.
TAG_UAFV1_REG_CMD_RESP	0x2102	Tag for Register command response.
TAG_UAFV1_SIGN_CMD	0x2003	Tag for Sign command.
TAG_UAFV1_SIGN_CMD_RESP	0x2103	Tag for Sign command response.
TAG_UAFV1_DEREG_CMD	0x2004	Tag for Deregister command.
TAG_UAFV1_DEREG_CMD_RESP	0x2104	Tag for Deregister command response.

Table 4.2: UAF Authenticator Command TAGs (0x2000 - 0x21FF)

Name	Value	Description
UAF_STATUS_OK	0x0000	Success
UAF_STATUS_ERR_UNKNOWN	0x0001	A generic error
UAF_STATUS_ERR_ACCESS_DENIED	0x0002	Access to command is denied
UAF_STATUS_USER_NOT_ENROLLED	0x0003	User is not enrolled

Table 4.3: UAF Authenticator Status Codes

135 5 Structures

136 5.1 RawKeyHandle

137 RawKeyHandle is a structure generated and parsed by the Authenticator. Authentica-
138 tors may define RawKeyHandle in different ways and its internal structure is relevant
139 only to the specific Authenticator implementation.

140 This section provides a sample reference definition.

32 bytes	Depends on key type. (e.g. 32 bytes)	Username Size (1 byte)	Max 128 bytes
KHAccessToken	Uauth.priv	Size	Username

Table 5.1: RawKeyHandle Structure

> ***Normative Note:***
> *First Factor Authenticators MUST store Username and Second Factor Authenticators MUST NOT store it. The ability to support Username is a key difference between first-, and second-factor Authenticators.*
>
> *RawKeyHandle MUST be wrapped before leaving the Authenticator boundary since it contains the user authentication private key (Uauth.priv).*

141 5.2 Structures to be parsed by FIDO Server

> ***Normative Note:***
>
> *This section is normative.*
>
> *The structures defined in this section are parsed by FIDO Server and Authenticators MUST therefore follow exactly the same structure as defined here.*

5.2.1 TAG_UAFV1_REG_RESPONSE

The following TLV structure is generated by the Authenticator during processing of a Register command. It is then delivered to FIDO Server intact, and parsed by the server. The structure embeds a TAG_UAFV1_KRD tag which among other data contains the newly generated Uauth.pub. The entire TAG_UAFV1_KRD tag, including the tag value and length (1.2-1.2.14) MUST be signed with the Attestation Private Key.

Note that if the Authenticator wants to append custom data to TAG_UAFV1_KRD structure (and thus sign with Attestation Key) – this data MUST be included as an additional tag to TAG_UAFV1_KRD, following the PublicKey (1.2.14).

If the Authenticator wants to send additional data to FIDO Server without signing it - this data MUST be included as an additional tag to TAG_UAFV1_REG_RESPONSE, following TAG_ATTESTATION_CERT (1.4.2).

	TLV Structure		Description
1	UINT16	Tag	TAG_UAFV1_REG_RESPONSE
1.1	UINT16	Length	Entire Length of the structure
1.2	UINT16	Tag	TAG_UAFV1_KRD
1.2.1	UINT16	Length	Entire Length of the structure
1.2.2	UINT16	Size	AAID size
1.2.3	BYTE	AAID[Size]	Authenticator AAID
1.2.4	BYTE	AuthenticatorVersion	Vendor assigned authenticator version
1.2.5	UINT16	PublicKeyAlgAndEncoding	Public Key algorithm and encoding. Refer to [FIDORegistry] for information on supported algorithms and their values.
1.2.6	UINT16	SignatureAlgAndEncoding	Signature Algorithm and Encoding. Refer to [FIDORegistry] for information on supported algorithms and their values.
1.2.7	UINT16	Size	Final Challenge size
1.2.8	BYTE	FinalChallenge[Size]	Final Challenge provided in the Command

	TLV Structure		Description
1.2.9	UINT16	Size	KeyID size
1.2.10	BYTE	KeyID[Size]	KeyID generated by Authenticator
1.2.11	UINT32	RegCounter	Registration Counter. Indicates how many times this Authenticator has performed registrations in the past.
1.2.12	UINT32	SignCounter	Signature Counter. Indicates how many times this Authenticator has performed signatures with any Uauth.priv in the past.
1.2.13	UINT16	Size	Size of Uauth.pub
1.2.14	BYTE	PublicKey[Size]	User authentication public key (Uauth.pub) newly generated by Authenticator
1.3	UINT16	Tag	TAG_SIGNATURE
1.3.1	UINT16	Size	Signature size
1.3.2	BYTE	Signature[Size]	Signature calculated with Attestation Private Key over TAG_UAFV1_KRD content. Note that entire TAG_UAFV1_KRD content, including the tag and it's length field, MUST be included during signature computation.
1.4	UINT16	Tag	TAG_ATTESTATION_CERT
1.4.1	UINT16	Length	Entire Length of Attestation Cert
1.4.2	BYTE	Certificate[Length]	Attestation Certificate byte array

5.2.2 TAG_UAFV1_SIGN_RESPONSE

The following TLV structure is generated by an Authenticator during processing of a Sign command. It is then delivered to FIDO Server intact and parsed by the server. The structure embeds a TAG_UAFV1_SIGNEDDATA tag. The entire TAG_UAFV1_SIGNEDDATA tag, including the tag value and length (1.2-1.2.11) MUST be signed with the appropriate Uauth.priv key.

Note that if the Authenticator wants to append custom data to TAG_UAFV1_SIGNED-DATA structure (and thus sign with Uauth.priv) – this data MUST be included as an additional tag to TAG_UAFV1_SIGNEDDATA, following the SignCounter (1.2.11).

If the Authenticator wants to send additional data to FIDO Server without signing it - this data MUST be included as an additional tag to TAG_UAFV1_SIGN_RESPONSE, following TAG_SIGNATURE (1.3.2).

	TLV Structure		Description
1	UINT16	Tag	TAG_UAFV1_SIGN_RESPONSE
1.1	UINT16	Length	Entire Length of the structure.
1.2	UINT16	Tag	TAG_UAFV1_SIGNEDDATA
1.2.1	UINT16	Length	Entire Length of the structure.
1.2.2	BYTE	AuthenticatorVersion	Vendor assigned authenticator version.
1.2.3	BYTE	AuthenticationMode	Authentication Mode indicating whether user explicitly verified or not and indicating if there is a transaction content or not. • 0x01 means that user has been explicitly verified • 0x02 means that transaction content has been shown on secure display and user confirmed it by explicitly verifying with authenticator
1.2.4	UINT16	SignatureAlgAndEncoding	Signature algorithm and encoding scheme. Refer to [FIDORegistry] for infor-

	TLV Structure		Description
			mation on supported algorithms and their values.
1.2.5	UINT16	Size	Authenticator Nonce size - MUST be at least 8 bytes
1.2.6	BYTE	AuthnrNonce[Size]	A nonce randomly generated by Authenticator
1.2.7	UINT16	Size	Final Challenge size
1.2.8	BYTE	FinalChallenge[Size]	Final Challenge provided in the Command
1.2.9	UINT16	Size	Transaction Content Hash size
1.2.10	BYTE	TCHash[Size]	Transaction Content Hash
1.2.11	UINT32	SignCounter	Signature Counter. Indicates how many times this Authenticator has performed signatures with any user authentication keys (Uauth.priv) in the past.
1.3	UINT16	Tag	TAG_SIGNATURE
1.3.1	UINT16	Size	Signature size
1.3.2	BYTE	Signature[Size]	Signature calculated using Uauth.priv over TAG_UAFV1_SIGNEDDATA structure. Note that entire TAG_UAFV1_SIGNEDDATA content, including the tag and it's length field, MUST be included during signature computation.

6 Commands

All UAF Authenticator commands and responses are semantically similar - they are all represented as TLV-encoded blobs. The first 2 bytes of each command is the command code. After receiving a command, the Authenticator MUST parse the first TAG and figure out which command is being issued.

> ***Informative Note:***
> *Supporting exactly the same semantics of commands is not a requirement but is recommended. This applies to all commands described in this section.*

6.1 GetInfo Command

6.1.1 General Description

This command returns a subset of the Authenticator's UAF metadata.

6.1.2 Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_GETINFO_CMD
1.1	UINT16	Length	Entire Command Length - must be 0 for this command

Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_GETINFO_CMD_RESP
1.1	UINT16	Length	Command response length
1.2	BYTE	AIVersion	Authenticator Interface Version. MUST be the number 1. This version indicates the types of commands and formatting associated with them supported by Authenticator.
1.3	BYTE	AAID[9]	Vendor assigned AAID
1.4	UINT32	AuthFactor	Authentication Factor (as defined in [FIDORegistry])
1.5	UINT32	KeyProtection	Key Protection type (as defined in [FIDORegistry])
1.6	UINT32	SecureDisplay	Secure Display type (as defined in [FIDORegistry])
1.7	UINT32	AuthenticationAlg	Authentication Algorithm (as defined in [FIDORegistry])
1.8	BYTE	Scheme[8]	Authentication Scheme (as defined in [FIDORegistry])
1.9	BYTE	IsSecondFactor	Indicates if the Authenticator is designed to function only as second factor

Informative Note:
Refer to [FIDORegistry] for bitflag definitions of AuthFactor, KeyProtection, SecureDisplay, AuthenticationAlg and Scheme.

6.2 Register Command

6.2.1 General Description

This command generates a UAF registration assertion. This assertion can be used to register the Authenticator with a FIDO Server.

Authenticator MUST perform the following steps:

1. If this Authenticator has a Secure Display – make sure Command.TAG_FULL_APPID is provided, and display its content on the Secure Display when verifying the user. Update Command.KHAccessToken with TAG_FULL_APPID:
    - Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_FULL_APPID)
2. If User is already enrolled with Authenticator (via biometric enrollment, PIN setup or similar mechanism) - verify the user. If the verification has been already done in a previous command – make sure that Command.TAG_USERVERIFY_TOKEN is a valid token.
    a. If verification fails - return UAF_STATUS_ACCESS_DENIED
3. If User is not enrolled with Authenticator – take the User through enrollment process.
    a. If enrollment fails - return UAF_STATUS_ACCESS_DENIED
4. Generate a new key pair (Uauth.pub/Uauth.priv)
5. Create a RawKeyHandle
    a. Add Uauth.priv to RawKeyHandle
    b. If it's a Bound Authenticator - add Command.KHAccessToken to RawKeyHandle
    c. If it's a 1stF Authenticator - add Command.Username to RawKeyHandle
6. Wrap RawKeyHandle with Wrap.sym key
7. Create TAG_UAFV1_KRD structure
    a. Copy all the mandatory fields (see section 5.2.1)

200 b. If it's a 2ndF Roaming Authenticator with no internal storage for KeyHandles –
201 copy KeyHandle into TAG_UAFV1_KRD.KeyID. Otherwise generate a random
202 KeyID (note that hash of KeyHandle can be safely used as a KeyID).
203 8. Sign TAG_UAFV1_KRD with Attestation Private Key
204 9. Create TAG_UAFV1_REG_RESPONSE
205 a. Copy all the mandatory fields (see section 5.2.1)
206 b. If this is a Roaming Authenticator with internal storage for its KeyHandles
207 ■ Add KeyID and KeyHandle into internal storage
208 ■ Return KeyID in TAG_KEYID tag
209 c. If this a Bound Authenticator
210 ■ Return KeyHandle in TAG_KEYHANDLE and KeyID in TAG_KEYID
211 10. Return TAG_UAFV1_REG_CMD_RESP

212 Error Codes:

213 • UAF_STATUS_ERR_ACCESS_DENIED

6.2.2 Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_REG_CMD
1.1	UINT16	Length	Command Length
1.2	UINT16	Size	Final Challenge size
1.3	BYTE	FinalChallenge[Size]	Final Challenge provided by ASM (max 32 bytes)
1.4	UINT16	Size	KHAccessToken size
1.5	BYTE	KHAccessToken[Size]	KHAccessToken provided by ASM (max 32 bytes)
1.6	UINT16	Size	Username size
1.7	BYTE	Username[Size]	Username provided by ASM (max 128 bytes)
1.8	UINT16	Length	Entire Length of User Verification Token
1.9	BYTE	VerificationToken[Length]	User Verification Token. If Authenticator doesn't have a matcher – the length must be 0.
1.10	UINT16	Tag	TAG_FULL_APPID (optional) This tag must be present only in case if Authenticator has a Secure Display and is able to show AppID to user.
1.10.1	UINT16	Length	Entire Length of AppID
1.10.2	BYTE	AppID[Length]	Full AppID (max 256 bytes)

6.2.3 Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_REG_CMD_RESP

	TLV Structure		Description
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	StatusCode	Error Code returned by Authenticator
1.3	UINT16	Tag	TAG_UAFV1_REG_RESPONSE This is a placeholder for TAG_UAFV1_REG_RESPONSE. Refer to section 5.2.1 for details on how this tag is defined and what fields it contains.
...	...		...
1.4	UINT16	Tag	TAG_KEYHANDLE (optional)
1.4.1	UINT16	Length	Entire Length of KeyHandle
1.4.2	BYTE	KeyHandle[Length]	KeyHandle
1.5	UINT16	Tag	TAG_KEYID (optional)
1.5.1	UINT16	Length	Entire Length of KeyID
1.5.2	BYTE	KeyID[Length]	KeyID

Normative Note:

For Silent Authenticators, KeyHandle MUST never be stored on a FIDO Server, otherwise this would enable tracking of users without providing the ability for users to clear KeyHandles from local device.

KeyID MUST be a unique and unguessable 32 bytes byte array. The uniqueness MUST be within the scope of AAID.

If an Authenticator is not able to protect an attestation private key - it's RECOMMENDED to not support attestation at all. If an Authenticator doesn't support attestation – the final TAG_UAFV1_KRD object MUST be signed with newly generated Uauth.priv key. In addition the content of TAG_ATTESTATION_CERT MUST have 0 length.

If Authenticator doesn't support SignCounter or Reg Counter it MUST set these to 0 in TAG_UAFV1_KRD.

6.3 Sign Command

6.3.1 General Description

This command generates a UAF assertion. This assertion can be further verified by a FIDO Server which has a prior registration with this Authenticator.

Informative Note:

1stF Authenticators MUST implement this command in two stages.

1. *The first stage will be executed only if Authenticator finds out that there are multiple keyHandles after filtering with KHAccessToken. In this stage Authenticator must return a list of usernames along with corresponding keyHandles*
2. *In the second stage, after user selects a username, this command will be called with a single keyHandle and will return a UAF assertion based on this keyHandle*

2ndF Authenticators SHOULD NOT support the first stage.

Authenticator MUST take the following steps:

1. If this authenticator has a Secure Display – make sure Command.TAG_FULL_APPID is provided and display it on Secure Display when verifying the user
    a. Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_FULL_APPID)

2. If User is already enrolled with authenticator (such as biometric enrollment, PIN setup, etc.) - verify the user. If the verification has been already done in one of previous commands – make sure that Command.TAG_USERVERIFY_TOKEN is a correct token.
    a. If verification fails - return UAF_STATUS_ACCESS_DENIED
3. If User is not enrolled – return UAF_STATUS_USER_NOT_ENROLLED
4. Unwrap all provided KeyHandles from Command.TAG_KEYHANDLE_LIST using Wrap.sym
    a. If it's a 1stF Roaming Authenticator
        - If Command.TAG_KEYHANDLE_LIST is empty – unwrap KeyHandles stored in its internal storage
        - If Command.TAG_KEYHANDLE_LIST is not empty – the items in this list are KeyIDs. Use these KeyIDs to locate KeyHandles stored in internal storage
5. Filter RawKeyHandles with Command.KHAccessToken
    a. RawKeyHandle.KHAccessToken == Command.KHAccessToken
6. If number of remaining RawKeyHandles is 0 – fail with UAF_STATUS_ACCESS_DENIED
7. If number of remained RawKeyHandles is > 1
    a. If it’s not a 1stF Authenticator – return UAF_STATUS_ACCESS_DENIED
    b. Create TAG_UNAME_LIST and copy {Command.KeyHandle, RawKeyHandle.username} pairs for all remaining RawKeyHandles into this tag
    c. Copy TAG_UNAME_LIST into TAG_UAFV1_SIGN_CMD_RESP and return
8. If number of remaining RawKeyHandles is 1
    a. Set Response.AuthenticationMode to 0x01
    b. If TransactionContent is not empty
        - If this is a Silent Authenticator – return UAF_STATUS_ACCESS_DENIED
        - If Authenticator doesn’t have a built-in Secure Display – return UAF_STATUS_ACCESS_DENIED

- Show Command.TransactionContent and Command.TAG_FULL_APPID on Secure Display and wait for the user to confirm it
  - Return UAF_STATUS_ACCESS_DENIED if user cancels the transaction
- Compute hash of TransactionContent
  - TAG_UAFV1_SIGNEDDATA.TCHash = hash(Command.TransactionContent)
  - Set TAG_UAFV1_SIGNEDDATA.AuthenticationMode to 0x02

c. Create TAG_UAFV1_SIGN_RESPONSE

- Create TAG_UAFV1_SIGNEDDATA
  - Increment SignCounter and put into TAG_UAFV1_SIGNEDDATA
  - Copy all the mandatory fields (see section 5.2.2)
  - If Command.AuthenticationMode == 0x01 - set TAG_UAFV1_SIGNEDDATA.TCHash size to 0
- Sign TAG_UAFV1_SIGNEDDATA with Uauth.priv

d. Copy TAG_UAFV1_SIGN_RESPONSE into TAG_UAFV1_SIGN_CMD_RESP and return it

Error Codes:

- UAF_STATUS_ERR_ACCESS_DENIED
- UAF_STATUS_USER_NOT_ENROLLED

6.3.2 Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_SIGN_CMD
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	Size	Final Challenge size

	TLV Structure		Description
1.3	BYTE	FinalChallenge[Size]	Final Challenge provided by ASM (max 32 bytes)
1.4	UINT16	Size	KHAccessToken size
1.5	BYTE	KHAccessToken[Size]	KHAccessToken provided by ASM (max 32 bytes)
1.6	UINT16	Size	Transaction Content size
1.7	BYTE	TransactionContent[Size]	Transaction Content provided by ASM (max 1024 bytes)
1.8	UINT16	Length	Entire Length of this tag
1.9	BYTE	VerificationToken[Length]	User Verification Token. If Authenticator doesn't have a Matcher – length must be 0.
1.8	UINT16	Tag	TAG_KHANDLE_LIST (optional)
1.8.1	UINT16	Length	Entire Length of list of KeyHandles in bytes. This TAG contains one or more (>=1) KeyHandle entries. Each entry has a size and content. Length = Sum(KeyHandleSizes) + NumberOfKeyHandles * sizeof(UINT16)
1.8.2	UINT16	Size	Size of KeyHandle
1.8.3	BYTE	KeyHandle[Size]	KeyHandle
1.9	UINT16	Tag	TAG_FULL_APPID (optional) This tag must be present only in case if Authenticator has a Secure Display and is able to show AppID to user.
1.10.1	UINT16	Length	Entire Length of AppID
1.10.2	BYTE	AppID[Length]	Full AppID (max 256 bytes)

6.3.3 Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_SIGN_CMD_RESP
1.1	UINT16	Length	Entire Length of Command Response
1.2	UINT16	StatusCode	StatusCode returned by Authenticator
1.3	UINT16	Tag	TAG_UNAME_LIST (optional) This TAG contains multiple (>=1) {Username, Keyhandle} entries.
1.3.1	UINT16	Length	Entire Length of list of {Username, KeyHandles} pairs. Length = Sum(UsernameSizes) + Sum(KeyHandleSizes) + (NumberOfUsernames * sizeof(UINT16) * 2)
1.3.2	UINT16	Size	Size of Username
1.3.3	BYTE	Username[Size]	Username
1.3.4	UINT16	Size	Size of KeyHandle
1.3.5	BYTE	KeyHandle[Size]	KeyHandle
1.4	UINT16	Tag	TAG_UAFV1_SIGN_RESPONSE (optional). This is a placeholder for TAG_UAFV1_SIGN_RESPONSE. Refer to section 5.2.2 for details on how this tag is defined and what fields it contains.
...	...		...

Normative Note:

Silent Authenticators MUST always behave as 2ndF Authnrs.

If Authenticator doesn't support SignCounter - it MUST set it to 0 in TAG_UAFV1_SIGNEDDATA.

Roaming Authenticators MUST send KeyID as a TAG in the response of SIGN command.

Some Authenticators might support Secure Display functionality not inside the Authenticator but within the boundaries of ASM. Typically these are software based Secure Displays. When processing the Sign command with a given transaction such Authenticators SHOULD assume that they do have a builtin Secure Display and SHOULD include the hash of transaction content in the final assertion without displaying anyting to the user. Also, such Authenticator's Metadata file MUST clearly indicate the type of Secure Display. Typically the flag of Secure Display will be SECURE_DISPLAY_ANY or SECURE_DISPLAY_PRIVILEGED_SOFTWARE.
See [FIDORegistry] for flags describing Secure Display type.

6.4 Deregister Command

6.4.1 General Description

This command deletes a registered UAF credential from Authenticator. Only Authenticators which store KeyHandle in internal storage must support this command.

Authenticator MUST take the following steps:

1. If it's an Authenticator which stores KeyHandles inside its internal storage
    - Find KeyHandle that matches Command.KeyID
    - Unwrap found KeyHandles using Wrap.sym
    - Make sure that RawKeyHandle.KHAccessToken == Command.KHAccessToken
        - If not - return UAF_STATUS_ACCESS_DENIED
    - Delete these KeyHandles from internal storage

287 2. Return TAG_UAFV1_DEREG_CMD_RESP

288 Error Codes:

289 • UAF_STATUS_ERR_ACCESS_DENIED

290 ### 6.4.2 Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_DEREG_CMD
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	Size	KHAccessToken size
1.3	BYTE	KHAccessToken[Size]	KHAccessToken provided by ASM (max 32 bytes)
1.4	UINT16	Size	KeyID size
1.5	BYTE	KeyID[Size]	KeyID provided by ASM (max 32 bytes)

291 ### 6.4.3 Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_SIGN_CMD_RESP
1.1	UINT16	Length	Entire Length of Command Response
1.2	UINT16	StatusCode	StatusCode returned by Authenticator

Informative Note:

Deregister command SHOULD not explicitly reveal whether provided keyID was registered or not.

7 Access Control for Commands

FIDO Authenticators may implement various mechanisms to guard access to privileged commands.

The following table summarizes the access control requirements for each command.

Terms used in the table:

- NoAuth – no access control
- UserVerify – explicit user verification
- KHAccessToken – must be known to the caller
- KeyHandleList – must be known to the caller
- KeyID - must be known to the caller

Com-mands	1stF Bound Authenticator	2ndF Bound Authenticator	1stF Roaming Authenticator	2ndF Roaming Authenticator
GetInfo	NoAuth	NoAuth	NoAuth	NoAuth
Register	UserVerify	UserVerify	UserVerify	UserVerify
Sign	UserVerify KHAccessToken KeyHandleList (provided by ASM)	UserVerify KHAccessToken KeyHandleList (provided by ASM)	UserVerify KHAccessToken	UserVerify KHAccessToken KeyHandleList (provided by ASM)
Deregister	KHAccessToken KeyID	KHAccessToken KeyID	KHAccessToken KeyID	KHAccessToken KeyID

Table 1: Access Control for Commands

Normative Note:
All UAF Authenticators MUST satisfy the access control requirements defined above.

Authenticator vendors MAY offer additional security mechanisms.

8 Relationship to other standards

The existing standard specifications most relevant to UAF authenticator are [TPM], [TEE] and [SecureElement].

Hardware modules implementing these standards may be extended to incorporate UAF functionality through their extensibility mechanisms such as by loading secure applications (trustlets, applets, etc) into them. Modules which do not support such extensibility mechanisms cannot be fully leveraged within UAF framework.

8.1 TEE

In order to support UAF inside TEE a special Trustlet (trusted application running inside TEE) may be designed which implements UAF Authenticator functionality specified in this document and also implements some kind of user verification technology (biometric verification, PIN or anything else).

An additional ASM must be created which knows how to work with the Trustlet.

8.2 Secure Elements

In order to support UAF inside SE a special Applet (trusted application running inside SE) may be designed which implements UAF Authenticator functionality specified in this document and also implements some kind of user verification technology (biometric verification, PIN or similar mechanisms).

An additional ASM must be created which knows how to work the Applet.

8.3 TPM

TPMs typically have a built-in attestation capability however this attestation model supported in TPMs is currently incompatible with UAF's basic attestation model. The future enhancements of UAF may include compatible attestation schemes.

Typically TPMs also have a built-in PIN verification functionality which may be leveraged for UAF.

In order to support UAF with an existing TPM module, the vendor should write an ASM which:

- Translates UAF data to TPM data by calling TPM APIs

- Creates assertions using TPMs API
- Reports itself as a valid UAF authenticator to UAF Client

A special AssertionScheme, designed for TPMs, must be also created (see [UAFAuthnrMetadata]) and published by FIDO Alliance. When FIDO Server receives an assertion with this AssertionScheme it will treat the received data as TPM-generated data and will parse/validate it accordingly.

Bibliography

FIDO Alliance Documents:

[FIDOGlossary] Rolf Lindemann, Davit Baghdasaryan, Brad Hill, John Kemp. FIDO Technical Glossary. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-glossary-v1.0-rd-20140209.pdf

[UAFProtocol] Rolf Lindemann, Davit Baghdasaryan, Eric Tiffany. FIDO Universal Authentication Framework Protocol. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-uaf-protocol-v1.0-rd-20140209.pdf

[UAFASM] Davit Baghdasaryan, John Kemp. FIDO Universal Authentication Framework Authenticator-specific Modules. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-uaf-asm-api-v1.0-rd-20140209.pdf

[UAFAuthnrMetadata] Davit Baghdasaryan, Brad Hill. FIDO Universal Authentication Framework Authenticator Metadata. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-uaf-authnr-metadata-v1.0-rd-20140209.pdf

[FIDORegistry] Rolf Lindemann, Davit Baghdasaryan, Brad Hill. FIDO Universal Authentication Framework Registry of Predefined Values. Version v1.0-rd-20140209, FIDO Alliance. February 2014. See http://fidoalliance.org/specs/fido-uaf-reg-v1.0-rd-20140209.pdf

Other References:

[**BioVocab**] Harmonized Biometric Vocabulary. Text of Standing Document 2 (SD 2) Version 8, WD 2.8, work-in-progress, ISO/IEC JTC 1/SC 37: Biometrics, 2007-08-22. Download: http://isotc.iso.org/livelink/livelink/fetch/2000/2122/327993/327973/654118/6687752/N_3004_JTC_1_SC_37_-_Harmonized_Biometric_Vocabulary_-_for_information.pdf?nodeid=6719683&vernum=0

[**CLICKJACKING**] Clickjacking: Attacks and Defenses, Lin-Shung Huang and Collin Jackson Carnegie Mellon University; Alex Moshchuk, Helen J. Wang, and Stuart Schlechter Microsoft Research. July 2012. Download https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf

[CommonCriteria] The Common Criteria (Specifications)

[FIPS-140-2] Security Requirements for Cryptographic Modules (Specification)

[FIPS 180-4] Secure Hash Standard (SHS) (**FIPS 180-4)**

[FIPS 186-4] NIST DIGITAL SIGNATURE STANDARD (DSS) (FIPS 186-4), National Institute of Standards and Technology, July 2013

[**PKCS#**1] Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 (RFC 3447), J. Jonsson et al, February 2003

[RFC2119] Key words for use in RFCs to Indicate Requirement Levels (RFC2119), S. Bradner, March 1997

[RFC4086] Randomness Requirements for Security (RFC 4086), D. Eastlake 3rd et al, June 2005

[RFC5639] Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation (RFC 5639), M Lochter et al, March 2010

[SecureElement] Global Platform Card Specifications (https://www.globalplatform.org/specifications.asp)

[SP800-131A] NIST Special Publication 800-131A, Transitions: Recommendations for Transitioning the Use of Cryptographic Algorithms and Key Lengths, E. Barker et al, National Institute of Standards and Technology, January 2011

[SP800-63-1] NIST Electronic Authentication Guideline SP 800-63-1 (NIST SP 800-63-1). W. Burr et al, National Institute of Standards and Technology, December 2011

[SP-800-57] Recommendation for Key Management – Part 1: General (Revision 3) (NIST SP 800-57)

[SP-800-38F] Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping (NIST SP 800-38F)

[SP-800-38D] Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC (NIST SP 800-38D)

[SP-800-38C] Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality (NIST SP 800-38C)

[SP-800-63-1] Electronic Authentication Guideline (NIST SP 800-63-1)

[TEE] Global Platform Trusted Execution Environment Specifications (https://www.globalplatform.org/specifications.asp)

[TEESecureDisplay] Trusted User Interface API Specficiation (https://www.globalplatform.org/specifications.asp)

[TPM] TPM Main Specification (http://www.trustedcomputinggroup.org/resources/tpm_main_specification)

Appendix A: Security Guidelines

This section is informative only.

Category	Guidelines
Wrap.sym	If the Authenticator has a wrapping key (Wrap.sym), then the Authenticator must protect this key as its most sensitive asset. The overall security of Authenticator highly depends on the protection of this key. Wrap.sym strength MUST be equal or higher than the strength of secrets stored in RawKeyHandle. Refer to [SP-800-57] and [SP-800-38F] publications for more information about choosing the right wrapping algorithm and implementing it correctly. It is highly recommended to generate, store and operate this key inside a trusted execution environment. In situations where physical attacks and side channel attacks are considered in the threat model it is highly recommended to use a tamper-resistant hardware module. If the Authenticator uses Wrap.sym, it must ensure that unwrapping corrupted KeyHandle and unwrapping data which has invalid contents (e.g. KeyHandle from invalid origin) are indistinguishable for the caller.
Private Keys (Uauth.priv and Attestation Private Key)	This document requires (a) the attestation key to be used for attestation purposes only and (b) the authentication keys to be used for FIDO authentication purposes only. The related to-be-signed objects (i.e. Key Registration Data and SignData) are designed to reduce the likelihood of such attacks: 1. They start with a tag marking them as specific FIDO objects 2. They include an Authenticator generated random value. As a consequence all to-be-signed objects are unique with very high probability. 3. They have a structure allowing only very few fields containing uncontrolled values, i.e. value which are neither generated nor verified by the Authenticator
Attestation Private Key	Authenticator must protect Attestation Private Key as a very sensitive asset. The overall security of Authenticator depends on the protection

Category	Guidelines
	level of this key. It is highly recommended to store and operate this key inside a tamper-resistant hardware module. Authenticators must ensure that the Attestation Private Key 1. Is *only* used to attest Authentication Keys generated and protected by the FIDO Authenticator using the FIDO defined data structures, KeyRegistrationData. 2. Never is accessible outside the FIDO Authenticator boundary. Attestation must be implemented in a way that two different relying parties cannot link registrations, authentications or other transactions.
Uauth.priv	Authenticator must protect all Uauth.priv keys as its most sensitive assets. The overall security of Authenticator highly depends on the protection level of these keys. It is highly recommended to generate, store and operate this key inside a trusted execution environment. In situations where physical attacks and side channel attacks are considered in the threat model it is highly recommended to use a tamper-resistant hardware module. FIDO Authenticators must ensure that Uauth.priv keys 1. are specific to the particular account at one relying party (relying party is identified by an AppID) 2. are generated based on good random numbers with sufficient entropy. The challenge provided by FIDO Server SHOULD be mixed into the entropy pool in order to add additional entropy. 3. are never directly revealed, i.e. always remain in exclusive control of the FIDO Authenticator 4. are only being used for the defined Authentication Modes, i.e. a. authenticating to the AppID they have been generated for, or b. confirming transaction to the AppID they have been generated for, or 5. are only being used to create the FIDO defined data structures, i.e. KRD, SignData.

Category	Guidelines
Username	Username MUST NOT be returned in plaintext in any condition other than the conditions described for SIGN command. In all other conditions Usernames MUST be stored inside KeyHandle.
AppIDs and KeyIDs	Registered AppIDs and KeyIDs MUST NOT be returned by authenticator in plaintext without user verification first. If attacker gets physical access to a roaming authenticator - it should not be easy to read out AppIDs and KeyIDs.
Crypto Kernel	Crypto Kernel is a module of the Authenticator implementing crypto functions (key generation, signing, wrapping, etc) necessary for UAF and having access to Uauth.priv, Attestation Private Key and Wrap.sym. This module must reside within the same security boundaries as Uauth.priv, Att.priv and Wrap.sym keys are residing. If it resides in a different module than the implementation must guarantee the same level of security as if they would reside within the same module. It is highly recommended to generate, store and operate this key inside a trusted execution environments. In situations where physical attacks and side channel attacks are considered in the threat model it is highly recommended to use a tamper-resistant hardware module. “Software” based Authenticators must make sure to use state of the art code protection and obfuscation techniques to protect this module and whitebox encryption techniques to protect the associated keys. Authenticators need good Random Number Generators using good entropy source enough for 1. generating authentication keys 2. generating signatures 3. computing Authenticator generated challenges Authenticator’s RNG should be such that it cannot be disabled or controlled in a way that may cause it to generate predictable outputs. If the Authenticator doesn’t have sufficient entropy for generating

Category	Guidelines
	strong random numbers, it should fail safe.
KeyHandle	It is highly recommended to use authenticated encryption while wrapping KeyHandles with Wrap.sym. Algorithms such as AES-GCM and AES-CCM are most suitable for this operation.
Matcher	Tampering with the Matcher module may have significant security consequences. It is highly recommended for this module to reside within integrity boundaries of Authenticator and detect tampering of itself. It is highly recommended to run this module inside a trusted execution environment (TEE). Authenticators which have separated Matcher and CryptoKernel modules should implement mechanisms which would allow CryptoKernel to securely receive assertions from Matcher module indicating user's local verification status. Software based Authenticators (if not in trusted execution environment) must make sure to use state of the art code protection and obfuscation techniques to protect this module. When Authenticator receives a wrong UserVerificationToken it should treat it as an attack and invalidate the cached UserVerificationToken. UserVerificationToken should have a lifetime not longer than 10 seconds. Authenticators must implement anti-hammering for their matchers. Biometrics based authenticators must protect the captured biometrics data (such as fingerprints) as well as the reference data (templates) and make sure that they never leave the security boundaries of authenticators.
Random Numbers	The FIDO Authenticator uses its random number generator to generate authentication key pairs, client side challenges and potentially for creating ECDSA signatures. Weak random numbers will make FIDO vulnerable to certain attacks. It is important for the FIDO Authenticator to work with good random numbers only.

Category	Guidelines
Secure Display	Secure Display MUST ensure that the user is presented with the provided Transaction Content, e.g. not overlaid by other display elements and clearly recognizable. See [Clickjack] for some examples of threats and potential counter-measures For more guidelines refer to [TEESecureDisplay].
Certifications	Vendors must strive passing security standard certifications with Authenticators, such as [FIPS-140-2], [CommonCriteria] and similar. Passing such certifications will positively impact the UAF implementation inside Authenticator.
SignCounter	Good protection measures of the Attestation Private Key is one method to prevent cloning authenticators. In some situations the protection measures might not be sufficient. If the Authenticator maintains a SignCounter, then the FIDO Server would have an additional method to detect cloned authenticators. If SignCounter is implemented: ensure that 1. It is increased by any authentication / transaction confirmation operation and 2. it cannot be manipulated/modified otherwise (e.g. API calls, etc.)