FIDO UAF Authenticator Commands v1.0

FIDO Alliance Proposed Standard 08 December 2014

This version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-authnr-cmds-v1.0-ps-20141208.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-authnr-cmds-v1.0-rd-20141008.pdf
Editors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
John Kemp, FIDO Alliance
Contributors:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Roni Sasson, Discretix
Brad Hill, PayPal, Inc.

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

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.

This document defines normative aspects of UAF Authenticators and offers security and implementation guidelines for authenticator implementors.

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].

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.

This document specifies low-level functionality which UAF Authenticators should implement in order to support the UAF protocol. It has the following goals:

- Define normative aspects of UAF Authenticator implementations
- Define a set of commands implementing UAF functionality that may be implemented by different types of authenticators
- Define `UAFV1TLV` assertion scheme-specific structures which will be parsed by a FIDO Server

> NOTE
>
> The UAF Protocol supports various assertion schemes. Commands and structures defined in this document assume that an authenticator supports the `UAFV1TLV` assertion scheme. Authenticators implementing a different assertion scheme do not have to follow requirements specified in this document.

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:

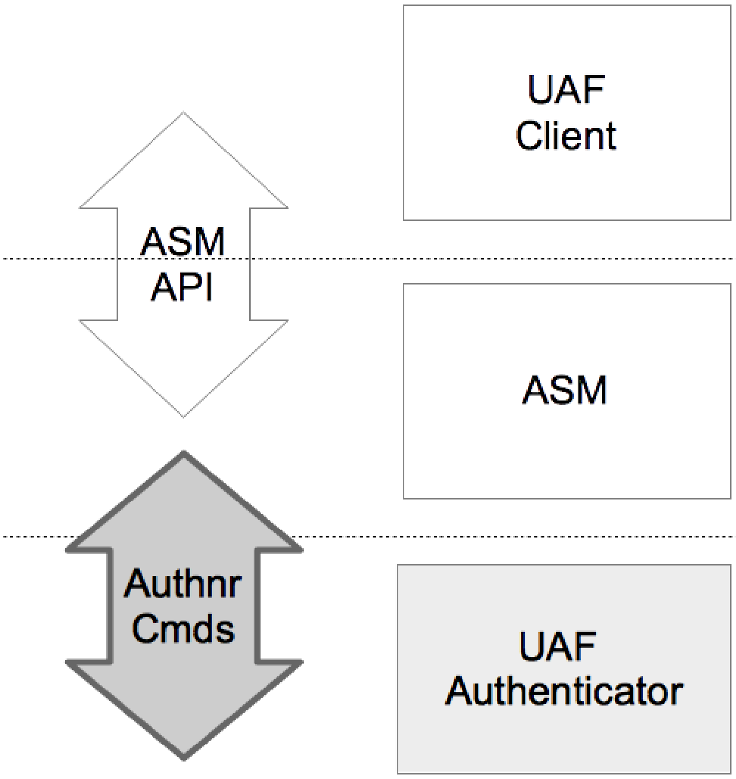

Fig. 1 UAF Authenticator Commands

3. Additional Notations

This section is normative.

Unless otherwise specified all data described in this document must be encoded in **little-endian** format.

All TLV structures can be parsed using a "recursive-descent" parsing approach, and tag order is not significant. In some cases multiple occurrences of a single tag may be allowed within a structure, in which case all values must be preserved.

All fields in TLV structures are *mandatory*, unless explicitly mentioned as otherwise.

4. UAF Authenticator

This section is non-normative.

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 with the verification mechanism built into the authenticator. The verification technology can vary, from biometric verification to simply verifying physical presence, or no user verification at all (the so-called *Silent Authenticator*).
2. [Mandatory] Performing the cryptographic operations defined in [UAFProtocol]
3. [Mandatory] Creating data structures that can be parsed by FIDO Server.
4. [Mandatory] Attesting itself to the FIDO Server if there is a built-in support for attestation
5. [Optional] Displaying the transaction content to the user using the transaction confirmation display

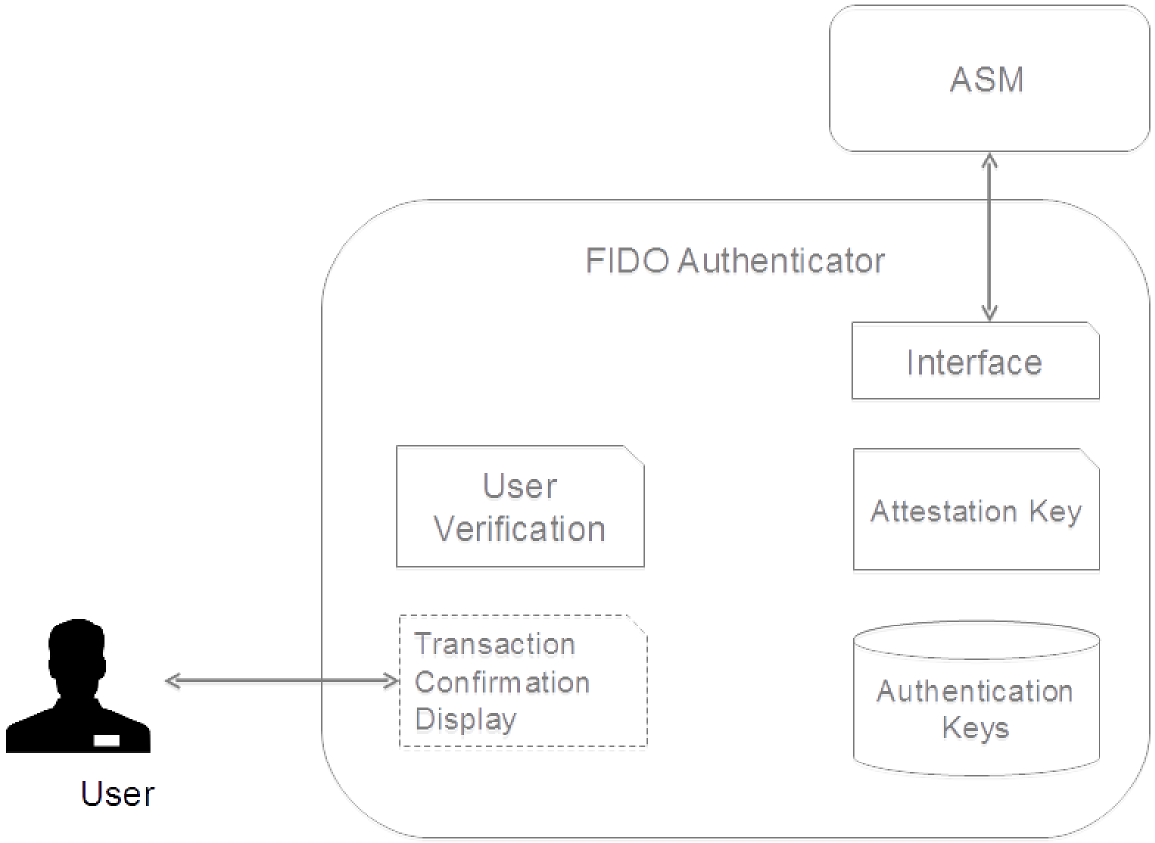

Fig. 2 FIDO Authenticator Logical Sub-Components

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

4.1 Types of Authenticators

There are four types of authenticators defined in this document. These definitions are not normative (unless otherwise stated) and are provided merely for simplifying some of the descriptions.

NOTE

The following is the rationale for considering only these 4 types of authenticators:

- Bound authenticators are typically embedded into a user's computing device and thus can utilize the host's storage for their needs. It makes more sense from an economic perspective to utilize the host's storage rather than have embedded storage. Trusted Execution Environments (TEE), Secure Elements and Trusted Platform Modules (TPM) are typically designed in this manner.
- First-factor roaming authenticators must have an internal storage for key handles.
- Second-factor roaming authenticators can store their key handles on an associated server, in order to avoid the need for internal storage.
- Defining such constraints makes the specification simpler and clearer for defining the mainstream use-cases.

Vendors, however, are not limited to these constraints. For example a bound authenticator which has internal storage for storing key handles is possible. Vendors are free to design and implement such authenticators as long as their design follows the normative requirements described in this document.

- **First-factor Bound Authenticator**
  - 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.
  - There is a logical binding between this authenticator and the device it is attached to (the binding is expressed through a concept called KeyHandleAccessToken). This authenticator cannot be bound with more than one device.
  - These authenticators do not store key handles in their own internal storage. They always return the key handle to the ASM and the latter stores it in its local database.
  - Authenticators of this type may also work as a second factor.
  - Examples
    - A fingerprint sensor built into a laptop, phone or tablet
    - Embedded secure element in a mobile device
    - Voice verification built into a device

- **Second-factor (2ndF) Bound Authenticator**
  - This type of authenticator is similar to first-factor bound authenticators, except that it can operate only as the second-factor in a multi-factor authentication
  - 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

- **First Factor (1stF) Roaming Authenticator**
  - These authenticators are not bound to any device. User can use them with any number of devices.
  - 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 key handles in their own internal secure storage and not expose 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 for a different device on the user's behalf

- **Second-factor Roaming Authenticator**
  - These authenticators are not bound to any device. A user may use them with any number of devices.
  - These authenticators may have an internal matcher. The matcher is able to verify an already enrolled user. If there is more than one user enrolled then the matcher can also identify a particular specific user.
  - It is assumed that these authenticators do not store key handles in their own internal storage. Instead they push key handles to the FIDO Server and receive them back during the authentication operation.
  - 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 these types of authenticators.

NORMATIVE

In some deployments, the combination of ASM and a bound authenticator can act as a roaming authenticator (for example when an ASM with an embedded authenticator on a mobile device acts as a roaming authenticator for another device). When this happens such an authenticator must follow the requirements applying to bound authenticators within the boundary of the system the authenticator is bound to, and follow the requirements that apply to roaming authenticators in any other system it connects to externally.

NOTE

As stated above, the bound authenticator does not store key handles and roaming authenticators to store them. In the example above the ASM would store the key handles of the bound authenticator and hence meets this assumptions.

5. Tags

This section is normative.

In this document UAF Authenticators use "Tag-Length-Value" (TLV) format to communicate with the outside world. All requests and response data must be encoded as TLVs.

Commands and existing predefined TLV tags can be extended by appending other TLV tags (custom or predefined).

Refer to [UAFRegistry] for information about predefined TLV tags.

TLV formatted data has the following simple structure:

2 bytes	2 bytes	Length bytes
Tag	Length in bytes	Data

All lengths are in bytes. e.g. a UINT32[4] will have length 16.

Although 2 bytes are allotted for the tag, only the first 14 bits (values up to 0x3FFF) should be used to accommodate the limitations of some hardware platforms.

Arrays are implicit. The description of some structures indicates where multiple values are permitted, and in these cases, if same tag appears more than once, all values are signifanct and should be treated as an array.

For convenience in decoding TLV-formatted messages, all composite tags - those with values that must be parsed by recursive descent - have the 13th bit (0x1000) set.

A tag that has the 14th bit (0x2000) set indicates that it is critical and a receiver must abort processing the entire message if it cannot process that tag.

Since UAF Authenticators may have extremely constrained processing environments, an ASM must follow a normative ordering of structures when sending commands.

It is assumed that ASM and Server have sufficient resources to handle parsing tags in any order so structures send from authenticator may use tags in any order.

5.1 Command Tags

Name	Value	Description
TAG_UAFV1_GETINFO_CMD	0x3401	Tag for GetInfo command.
TAG_UAFV1_GETINFO_CMD_RESPONSE	0x3601	Tag for GetInfo command response.
TAG_UAFV1_REGISTER_CMD	0x3402	Tag for Register command.
TAG_UAFV1_REGISTER_CMD_RESPONSE	0x3602	Tag for Register command response.
TAG_UAFV1_SIGN_CMD	0x3403	Tag for Sign command.
TAG_UAFV1_SIGN_CMD_RESPONSE	0x3603	Tag for Sign command response.
TAG_UAFV1_DEREGISTER_CMD	0x3404	Tag for Deregister command.
TAG_UAFV1_DEREGISTER_CMD_RESPONSE	0x3604	Tag for Deregister command response.
TAG_UAFV1_OPEN_SETTINGS_CMD	0x3406	Tag for OpenSettings command.
TAG_UAFV1_OPEN_SETTINGS_CMD_RESPONSE	0x3606	Tag for OpenSettings command response.

Table 4.1.1: UAF Authenticator Command TLV tags (0x3400 - 0x34FF, 0x3600-0x36FF)

5.2 Tags used only in Authenticator Commands

Name	Value	Description
TAG_KEYHANDLE	0x2801	Represents key handle. Refer to [FIDOGlossary] for more information about key handle.
TAG_USERNAME_AND_KEYHANDLE	0x3802	Represents an associated Username and key handle. This is a composite tag that contains a TAG_USERNAME and TAG_KEYHANDLE that identify a registration valid oin the authenticator. Refer to [FIDOGlossary] for more information about username.
TAG_USERVERIFY_TOKEN	0x2803	Represents a User Verification Token. Refer to [FIDOGlossary] for more information about user verification tokens.

Name	Value	Description
TAG_APPID	0x2804	A full AppID as a UINT8[] encoding of a UTF-8 string. Refer to [FIDOGlossary] for more information about AppID.
TAG_KEYHANDLE_ACCESS_TOKEN	0x2805	Represents a key handle Access Token.
TAG_USERNAME	0x2806	A Username as a UINT8[] encoding of a UTF-8 string.
TAG_ATTESTATION_TYPE	0x2807	Represents an Attestation Type.
TAG_STATUS_CODE	0x2808	Represents a Status Code.
TAG_AUTHENTICATOR_METADATA	0x2809	Represents a more detailed set of authenticator information.
TAG_ASSERTION_SCHEME	0x280A	A UINT8[] containing the UTF8-encoded Assertion Scheme as defined in [UAFRegistry]. ("UAFV1TLV")
TAG_TC_DISPLAY_PNG_CHARACTERISTICS	0x280B	If an authenticator contains a PNG-capable transaction confirmation display that is not implemented by a higher-level layer, this tag is describing this display. See [UAFAuthnrMetadata] for additional information on the format of this field.
TAG_TC_DISPLAY_CONTENT_TYPE	0x280C	A UINT8[] containing the UTF-8-encoded transaction display content type as defined in [UAFAuthnrMetadata]. ("image/png")
TAG_AUTHENTICATOR_INDEX	0x280D	Authenticator Index
TAG_API_VERSION	0x280E	API Version
TAG_AUTHENTICATOR_ASSERTION	0x280F	The content of this TLV tag is an assertion generated by the authenticator. Since authenticators may generate assertions in different formats - the content format may vary from authenticator to authenticator.
TAG_TRANSACTION_CONTENT	0x2810	Represents transaction content sent to the authenticator.
TAG_AUTHENTICATOR_INFO	0x3811	Includes detailed information about authenticator's capabilities.
TAG_SUPPORTED_EXTENSION_ID	0x2812	Represents extension ID supported by authenticator.

Table 4.2.1: Non-Command Tags (0x2800 - 0x28FF, 0x3800 - 0x38FF)

5.3 Tags used in UAF Protocol

Name	Value	Description
TAG_UAFV1_REG_ASSERTION	0x3E01	Authenticator response to Register command.
TAG_UAFV1_AUTH_ASSERTION	0x3E02	Authenticator response to Sign command.
TAG_UAFV1_KRD	0x3E03	Key Registration Data
TAG_UAFV1_SIGNED_DATA	0x3E04	Data signed by authenticator with the UAuth.priv key
TAG_ATTESTATION_CERT	0x2E05	Each entry contains a single X.509 DER-encoded [ITU-X690-2008] certificate. Multiple occurrences are allowed and form the attestation certificate chain. Multiple occurrences must be ordered. The attestation certificate itself must occur first. Each subsequent occurrence (if exists) must be the issuing certificate of the previous occurrence.
TAG_SIGNATURE	0x2E06	A cryptographic signature
TAG_ATTESTATION_BASIC_FULL	0x3E07	Full Basic Attestation as defined in [UAFProtocol]
TAG_ATTESTATION_BASIC_SURROGATE	0x3E08	Surrogate Basic Attestation as defined in [UAFProtocol]
TAG_KEYID	0x2E09	Represents a KeyID.
TAG_FINAL_CHALLENGE	0x2E0A	Represents a Final Challenge. Refer to [UAFProtocol] for more information about the Final Challenge.

Name	Value	Description
TAG_AAID	0x2E0B	Represents an authenticator Attestation ID. Refer to [UAFProtocol] for more information about the AAID.
TAG_PUB_KEY	0x2E0C	Represents a Public Key.
TAG_COUNTERS	0x2E0D	Represents a use counters for the authenticator.
TAG_ASSERTION_INFO	0x2E0E	Represents assertion information necessary for message processing.
TAG_AUTHENTICATOR_NONCE	0x2E0F	Represents a nonce value generated by the authenticator.
TAG_TRANSACTION_CONTENT_HASH	0x2E10	Represents a hash of transaction content.
TAG_EXTENSION	0x3E11, 0x3E12	This is a composite tag indicating that the content is an extension. If the tag is 0x3E11 - it's a critical extension and if the recipient does not understand the contents of this tag, it must abort processing of the entire message. This tag has two embedded tags - TAG_EXTENSION_ID and TAG_EXTENSION_DATA. For more information about UAF extensions refer to [UAFProtocol] NOTE This tag can be appended to any command and response. Using tag 0x3E11 (as opposed to tag 0x3E12) has the same meaning as the flag `fail_if_unknown` in [UAFProtocol].
TAG_EXTENSION_ID	0x2E13	Represents extension ID. Content of this tag is a UINT8[] encoding of a UTF-8 string.
TAG_EXTENSION_DATA	0x2E14	Represents extension data. Content of this tag is a UINT8[] byte array.

Table 4.3.1: Tags used in the UAF Protocol (0x2E00 - 0x2EFF, 0x3E00 - 0x3EFF). Normatively defined in [UAFRegistry]

5.4 Status Codes

Name	Value	Description
UAF_CMD_STATUS_OK	0x00	Success
UAF_CMD_STATUS_ERR_UNKNOWN	0x01	An unknown error
UAF_CMD_STATUS_ACCESS_DENIED	0x02	Access to this operation is denied
UAF_CMD_STATUS_USER_NOT_ENROLLED	0x03	User is not enrolled with the authenticator
UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT	0x04	Transaction content cannot be rendered
UAF_CMD_STATUS_USER_CANCELLED	0x05	User has cancelled the operation
UAF_CMD_STATUS_CMD_NOT_SUPPORTED	0x06	Command not supported
UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED	0x07	Required attestation not supported

Table 4.4.1: UAF Authenticator Status Codes (0x00 - 0xFF)

6. Structures

This section is normative.

6.1 RawKeyHandle

RawKeyHandle is a structure generated and parsed by the authenticator. Authenticators may define RawKeyHandle in different ways and the internal structure is relevant only to the specific authenticator implementation.

RawKeyHandle for a typical **first-factor bound authenticator** has the following structure.

Depends on hashing algorithm (e.g. 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

First Factor authenticators must store Username inside RawKeyHandle 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 cryptographically wrapped before leaving the authenticator boundary since it contains the user authentication private key (UAuth.priv).

6.1.1 Structures to be parsed by FIDO Server

The structures defined in this section are created by UAF Authenticators and parsed by FIDO Servers.

Authenticators must generate these structures if they implement "UAFV1TLV" assertion scheme.

> NOTE
>
> "UAFV1TLV" assertion scheme assumes that the authenticator has exclusive control over all data included inside TAG_UAFV1_KRD and TAG_UAFV1_SIGNED_DATA.

The nesting structure must be preserved, but the order of tags within a composite tag is not normative. FIDO Servers must be prepared to handle tags appearing in any order.

6.1.1.1 TAG_UAFV1_REG_ASSERTION

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.

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 inside TAG_UAFV1_KRD.

If the authenticator wants to send additional data to FIDO Server without signing it - this data must be included as an additional tag inside TAG_UAFV1_REG_ASSERTION and not inside TAG_UAFV1_KRD.

Currently this document only specifies TAG_ATTESTATION_BASIC_FULL and TAG_ATTESTATION_BASIC_SURROGATE. In case if the authenticator is required to perform "Some_Other_Attestation" on TAG_UAFV1_KRD - it must use the TLV tag and content defined for "Some_Other_Attestation" (defined in [UAFRegistry]).

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_REG_ASSERTION
1.1	UINT16 Length	Length of the structure
1.2	UINT16 Tag	TAG_UAFV1_KRD
1.2.1	UINT16 Length	Length of the structure
1.2.2	UINT16 Tag	TAG_AAID
1.2.2.1	UINT16 Length	Length of AAID
1.2.2.2	UINT8[] AAID	Authenticator Attestation ID
1.2.3	UINT16 Tag	TAG_ASSERTION_INFO
1.2.3.1	UINT16 Length	Length of Assertion Information
1.2.3.2	UINT16 AuthenticatorVersion	Vendor assigned authenticator version
1.2.3.3	UINT8 AuthenticationMode	For Registration this must be 0x01 indicating that the user has explicitly verified the action.
1.2.3.4	UINT16 SignatureAlgAndEncoding	Signature Algorithm and Encoding of the attestation signature. Refer to [UAFRegistry] for information on supported algorithms and their values.
1.2.3.5	UINT16 PublicKeyAlgAndEncoding	Public Key algorithm and encoding of the newly generated `UAuth.pub` key. Refer to [UAFRegistry] for information on supported algorithms and their values.

TLV Structure		Description
1.2.4	UINT16 Tag	TAG_FINAL_CHALLENGE
1.2.4.1	UINT16 Length	Final Challenge length
1.2.4.2	UINT8[] FinalChallenge	(binary value of) Final Challenge provided in the Command
1.2.5	UINT16 Tag	TAG_KEYID
1.2.5.1	UINT16 Length	Length of KeyID
1.2.5.2	UINT8[] KeyID	(binary value of) KeyID generated by Authenticator
1.2.6	UINT16 Tag	TAG_COUNTERS
1.2.6.1	UINT16 Length	Length of Counters
1.2.6.2	UINT32 SignCounter	Signature Counter. Indicates how many times this authenticator has performed signatures in the past.
1.2.6.3	UINT32 RegCounter	Registration Counter. Indicates how many times this authenticator has performed registrations in the past.
1.2.7	UINT16 Tag	TAG_PUB_KEY
1.2.7.1	UINT16 Length	Length of UAuth.pub
1.2.7.2	UINT8[] PublicKey	User authentication public key (UAuth.pub) newly generated by authenticator
1.3 **(choice 1)**	UINT16 Tag	TAG_ATTESTATION_BASIC_FULL
1.3.1	UINT16 Length	Length of structure
1.3.2	UINT16 Tag	TAG_SIGNATURE
1.3.2.1	UINT16 Length	Length of signature
1.3.2.2	UINT8[] Signature	Signature calculated with Basic Attestation Private Key over TAG_UAFV1_KRD content. The entire TAG_UAFV1_KRD content, including the tag and it's length field, must be included during signature computation.
1.3.3	UINT16 Tag	TAG_ATTESTATION_CERT (multiple occurrences possible) Multiple occurrences must be ordered. The attestation certificate must occur first. Each subsequent occurrence (if exists) must be the issuing certificate of the previous occurrence. The last occurence must be chained to one of the certificates included in field `attestationRootCertificate` in the related Metadata Statement [UAFAuthnrMetadata].
1.3.3.1	UINT16 Length	Length of Attestation Cert
1.3.3.2	UINT8[] Certificate	Single X.509 DER-encoded [ITU-X690-2008] Attestation Certificate or a single certificate from the attestation certificate chain (see description above).
1.3 **(choice 2)**	UINT16 Tag	TAG_ATTESTATION_BASIC_SURROGATE
1.3.1	UINT16 Length	Length of structure
1.3.2	UINT16 Tag	TAG_SIGNATURE
1.3.2.1	UINT16 Length	Length of signature
1.3.2.2	UINT8[] Signature	Signature calculated with newly generated UAuth.priv key over TAG_UAFV1_KRD content. The entire TAG_UAFV1_KRD content, including the tag and it's length field, must be included during signature computation.

6.1.1.2 TAG_UAFV1_AUTH_ASSERTION

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_SIGNED_DATA tag.

If the authenticator wants to append custom data to TAG_UAFV1_SIGNED_DATA structure (and thus sign with Attestation Key) - this data must be included as an additional tag inside TAG_UAFV1_SIGNED_DATA.

If the authenticator wants to send additional data to FIDO Server without signing it - this data must be included as an additional tag inside TAG_UAFV1_AUTH_ASSERTION and not inside TAG_UAFV1_SIGNED_DATA.

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_AUTH_ASSERTION
1.1	UINT16 Length	Length of the structure.
1.2	UINT16 Tag	TAG_UAFV1_SIGNED_DATA
1.2.1	UINT16 Length	Length of the structure.
1.2.2	UINT16 Tag	TAG_AAID
1.2.2.1	UINT16 Length	Length of AAID
1.2.2.2	UINT8[] AAID	Authenticator Attestation ID
1.2.3	UINT16 Tag	TAG_ASSERTION_INFO
1.2.3.1	UINT16 Length	Length of Assertion Information
1.2.3.2	UINT16 AuthenticatorVersion	Vendor assigned authenticator version.
1.2.3.3	UINT8 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 the display and user confirmed it by explicitly verifying with authenticator
1.2.3.4	UINT16 SignatureAlgAndEncoding	Signature algorithm and encoding format. Refer to [UAFRegistry] for information on supported algorithms and their values.
1.2.4	UINT16 Tag	TAG_AUTHENTICATOR_NONCE
1.2.4.1	UINT16 Length	Length of authenticator Nonce -must be at least 8 bytes
1.2.4.2	UINT8[] AuthnrNonce	(binary value of) A nonce randomly generated by Authenticator
1.2.5	UINT16 Tag	TAG_FINAL_CHALLENGE
1.2.5.1	UINT16 Length	Length of Final Challenge
1.2.5.2	UINT8[] FinalChallenge	(binary value of) Final Challenge provided in the Command
1.2.6	UINT16 Tag	TAG_TRANSACTION_CONTENT_HASH
1.2.6.1	UINT16 Length	Length of Transaction Content Hash. This length is 0 if AuthenticationMode == 0x01, i.e. authentication, not transaction confirmation.
1.2.6.2	UINT8[] TCHash	(binary value of) Transaction Content Hash
1.2.7	UINT16 Tag	TAG_KEYID
1.2.7.1	UINT16 Length	Length of KeyID
1.2.7.2	UINT8[] KeyID	(binary value of) KeyID
1.2.8	UINT16 Tag	TAG_COUNTERS
1.2.8.1	UINT16 Length	Length of Counters
1.2.8.2	UINT32 SignCounter	Signature Counter. Indicates how many times this authenticator has performed signatures in the past.
1.3	UINT16 Tag	TAG_SIGNATURE

TLV Structure		Description
1.3.1	UINT16 Length	Length of Signature
1.3.2	UINT8[] Signature	Signature calculated using UAuth.priv over TAG_UAFV1_SIGNED_DATA structure. The entire TAG_UAFV1_SIGNED_DATA content, including the tag and it's length field, must be included during signature computation.

6.1.2 UserVerificationToken

This specification doesn't specify how exactly user verification must be performed inside the authenticator. Verification is considered to be an authenticator, and vendor, specific operation.

This document provides an example on how the "vendor_specific_UserVerify" command (a command which verifies the user using Authenticator's built-in technology) could be securely bound to UAF Register and Sign commands. This binding is done through a concept called `UserVerificationToken`. Such a binding allows decoupling "vendor_specific_UserVerify" and "UAF Register/Sign" commands from each other.

Here is how it is defined:

- The ASM invokes the "vendor_specific_UserVerify" command. The authenticator verifies the user and returns a `UserVerificationToken` back.
- The ASM invokes UAF.Register/Sign command and passes `UserVerificationToken` to it. The authenticator verifies the validity of `UserVerificationToken` and performs the FIDO operation if it is valid.

The concept of UserVerificationToken is non-normative. An authenticator might decide to implement this binding in a very different way. For example an authenticator vendor may decide to append a UAF Register request directly to their "vendor_specific_UserVerify" command and process both as a single command.

If `UserVerificationToken` binding is implemented, it should either meet one of the following criteria or implement a mechanism providing similar, or better security:

- `UserVerificationToken` must allow performing only a single UAF Register or UAF Sign operation.
- `UserVerificationToken` must be time bound, and allow performing multiple UAF operations within the specified time.

6.2 Commands

This section is non-normative.

> NORMATIVE
>
> UAF Authenticators which are designed to be interoperable with ASMs from different vendors must implement the command interface defined in this section. Examples of such authenticators:
>
> - Bound Authenticators in which the core authenticator functionality is developed by one vendor, and the ASM is developed by another vendor
> - Roaming Authenticators

> NORMATIVE
>
> UAF Authenticators which are tightly integrated with a custom ASM (typically bound authenticators) may implement a different command interface.

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 TLV tag and figure out which command is being issued.

6.2.1 GetInfo Command

6.2.1.1 Command Description

This command returns information about internal authenticators. It may return 0 or more authenticators. Each authenticator has an assigned `authenticatorIndex` which is used in other commands as an authenticator reference.

6.2.1.2 Command Structure

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

6.2.1.3 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_GETINFO_CMD_RESPONSE
1.1	UINT16 Length	Response length
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 Value	Status Code returned by Authenticator
1.3	UINT16 Tag	TAG_API_VERSION
1.3.1	UINT16 Length	Length of API Version (must be 0x0001)
1.3.2	UINT8 Version	Authenticator API Version (must be 0x01). This version indicates the types of commands, and formatting associated with them, that are supported by the authenticator.
1.4	UINT16 Tag	TAG_AUTHENTICATOR_INFO (multiple occurrences possible)
1.4.1	UINT16 Length	Length of Authenticator Info
1.4.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.4.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.4.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.4.3	UINT16 Tag	TAG_AAID
1.4.3.1	UINT16 Length	Length of AAID
1.4.3.2	UINT8[] AAID	Vendor assigned AAID
1.4.4	UINT16 Tag	TAG_AUTHENTICATOR_METADATA
1.4.4.1	UINT16 Length	Length of Authenticator Metadata
1.4.4.2	UINT16 AuthenticatorType	Indicates whether the authenticator is bound or roaming, and whether it is first-, or second-factor only. The ASM must use this information to understand how to work with the authenticator. Predefined values: • 0x0001 - Indicates second-factor authenticator (first-factor when the flag is not set) • 0x0002 - Indicates roaming authenticator (bound authenticator when the flag is not set) • 0x0004 - Key handles will be stored inside authenticator and won't be returned to ASM • 0x0008 - Authenticator has a built-in UI for enrollment and verification. ASM should not show its custom UI • 0x0010 - Authenticator has a built-in UI for settings, and supports OpenSettings command. • 0x0020 - Authenticator expects TAG_APPID to be passed as an argument to commands where it is defined as an optional argument • 0x0040 - At least one user is enrolled in the authenticator. Authenticators which don't support the concept of user enrollment (e.g. USER_VERIFY_NONE, USER_VERIFY_PRESENCE) must always have this bit set.
1.4.4.3	UINT8 MaxKeyHandles	Indicates maximum number of key handles this authenticator can receive and process in a single command. This information will be used by the ASM when invoking SIGN command with multiple key handles.
1.4.4.4	UINT32 UserVerification	User Verification method (as defined in [UAFRegistry])
1.4.4.5	UINT16 KeyProtection	Key Protection type (as defined in [UAFRegistry]).
1.4.4.6	UINT16 MatcherProtection	Matcher Protection type (as defined in [UAFRegistry]).

TLV Structure		Description
1.4.4.7	UINT16 TransactionConfirmationDisplay	Transaction Confirmation type (as defined in [UAFRegistry]). NOTE If Authenticator doesn't support Transaction Confirmation - this value must be set to 0.
1.4.4.8	UINT16 AuthenticationAlg	Authentication Algorithm (as defined in [UAFRegistry]).
1.4.5	UINT16 Tag	TAG_TC_DISPLAY_CONTENT_TYPE (optional)
1.4.5.1	UINT16 Length	Length of content type.
1.4.5.2	UINT8[] ContentType	Transaction Confirmation Display Content Type. See [UAFAuthnrMetadata] for additional information on the format of this field.
1.4.6	UINT16 Tag	TAG_TC_DISPLAY_PNG_CHARACTERISTICS (optional,multiple occurrences permitted)
1.4.6.1	UINT16 Length	Length of display characteristics information.
1.4.6.2	UINT32 Width	See [UAFAuthnrMetadata] for additional information.
1.4.6.3	UINT32 Height	See [UAFAuthnrMetadata] for additional information.
1.4.6.4	UINT8 BitDepth	See [UAFAuthnrMetadata] for additional information.
1.4.6.5	UINT8 ColorType	See [UAFAuthnrMetadata] for additional information.
1.4.6.6	UINT8 Compression	See [UAFAuthnrMetadata] for additional information.
1.4.6.7	UINT8 Filter	See [UAFAuthnrMetadata] for additional information.
1.4.6.8	UINT8 Interlace	See [UAFAuthnrMetadata] for additional information.
1.4.6.9	UINT8[] PLTE	See [UAFAuthnrMetadata] for additional information.
1.4.7	UINT16 Tag	TAG_ASSERTION_SCHEME
1.4.7.1	UINT16 Length	Length of Assertion Scheme
1.4.7.2	UINT8[] AssertionScheme	Assertion Scheme (as defined in [UAFRegistry])
1.4.8	UINT16 Tag	TAG_ATTESTATION_TYPE (multiple occurrences possible)
1.4.8.1	UINT16 Length	Length of AttestationType
1.4.8.2	UINT16 AttestationType	Attestation Type (as defined in [UAFRegistry])
1.4.9	UINT16 Tag	TAG_SUPPORTED_EXTENSION_ID (optional, multiple occurrences possible)
1.4.9.1	UINT16 Length	Length of SupportedExtensionID
1.4.9.2	UINT8[] SupportedExtensionID	SupportedExtensionID as a UINT8[] encoding of a UTF-8 string

6.2.1.4 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ERR_UNKNOWN`

6.2.2 Register Command

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

6.2.2.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_REGISTER_CMD
1.1	UINT16 Length	Command Length
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)

TLV Structure		Description
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.3	UINT16 Tag	TAG_APPID (optional)
1.3.1	UINT16 Length	Length of AppID
1.3.2	UINT8[] AppID	AppID (max 512 bytes)
1.4	UINT16 Tag	TAG_FINAL_CHALLENGE
1.4.1	UINT16 Length	Final Challenge Length
1.4.2	UINT8[] FinalChallenge	Final Challenge provided by ASM (max 32 bytes)
1.5	UINT16 Tag	TAG_USERNAME
1.5.1	UINT16 Length	Length of Username
1.5.2	UINT8[] Username	Username provided by ASM (max 128 bytes)
1.6	UINT16 Tag	TAG_ATTESTATION_TYPE
1.6.1	UINT16 Length	Length of AttestationType
1.6.2	UINT16 AttestationType	Attestation Type to be used
1.7	UINT16 Tag	TAG_KEYHANDLE_ACCESS_TOKEN
1.7.1	UINT16 Length	Length of KHAccessToken
1.7.2	UINT8[] KHAccessToken	KHAccessToken provided by ASM (max 32 bytes)
1.8	UINT16 Tag	TAG_USERVERIFY_TOKEN (optional)
1.8.1	UINT16 Length	Length of VerificationToken□
1.8.2	UINT8[] VerificationToken□	User verification token□

6.2.2.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_REGISTER_CMD_RESPONSE
1.1	UINT16 Length	Command Length
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 Value	Status code returned by Authenticator
1.3	UINT16 Tag	TAG_AUTHENTICATOR_ASSERTION
1.3.1	UINT16 Length	Length of Assertion
1.3.2	UINT8[] Assertion	Registration Assertion (see section TAG_UAFV1_REG_ASSERTION).
1.4	UINT16 Tag	TAG_KEYHANDLE (optional)
1.4.1	UINT16 Length	Length of key handle
1.4.2	UINT8[] Value	(binary value of) key handle

6.2.2.3 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ACCESS_DENIED`
- `UAF_CMD_STATUS_USER_CANCELLED`
- `UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED`
- `UAF_CMD_STATUS_ERR_UNKNOWN`

6.2.2.4 Command Description

The authenticator must perform the following steps (see below table for command structure):

1. If this authenticator has a transaction confirmation display and is able to display AppID, then make sure `Command.TAG_APPID` is provided, and show its content on the display when verifying the user. Update `Command.KHAccessToken` with `TAG_APPID`:
    - Update Command.KHAccessToken by mixing it with Command.TAG_APPID. An example of such mixing function is a cryptographic hash function.

      > NOTE
      >
      > This method allows us to avoid storing the AppID separately in the RawKeyHandle.

        - For example: Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_APPID)
2. If the user is already enrolled with this 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.
    1. If verification fails - return `UAF_CMD_STATUS_ACCESS_DENIED`
3. If the user is not enrolled with the authenticator then take the user through the enrollment process.
    1. If enrollment fails - return `UAF_CMD_STATUS_ACCESS_DENIED`
    2. If user explicitly cancels the operation - return `UAF_CMD_STATUS_USER_CANCELLED`
4. Make sure that Command.TAG_ATTESTATION_TYPE is supported. If not - return `UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED`
5. Generate a new key pair (UAuth.pub/UAuth.priv)
6. Create a RawKeyHandle
    1. Add UAuth.priv to RawKeyHandle
    2. Add Command.KHAccessToken to RawKeyHandle
    3. If a first-factor authenticator, then add Command.Username to RawKeyHandle
7. Wrap RawKeyHandle with Wrap.sym key
8. Create TAG_UAFV1_KRD structure
    1. If this is a second-factor roaming authenticator - place key handle inside TAG_KEYID. Otherwise generate a random KeyID and place it inside TAG_KEYID.
    2. Copy all the mandatory fields (see section TAG_UAFV1_REG_ASSERTION)
9. Perform attestation on TAG_UAFV1_KRD based on provided Command.AttestationType.
10. Create TAG_AUTHENTICATOR_ASSERTION
    1. Create TAG_UAFV1_REG_ASSERTION
        1. Copy all the mandatory fields (see section TAG_UAFV1_REG_ASSERTION)
        2. If this is a first-factor roaming authenticator - add KeyID and key handle into internal storage
        3. If this is a bound authenticator - return key handle inside TAG_KEYHANDLE
    2. Put the entire TLV structure for TAG_UAFV1_REG_ASSERTION as the value of TAG_AUTHENTICATOR_ASSERTION
11. Return TAG_UAFV1_REGISTER_CMD_RESPONSE
    1. `UAF_CMD_STATUS_OK` as a status
    2. Add TAG_AUTHENTICATOR_ASSERTION
    3. Add TAG_KEY_HANDLE if the key handle must be stored outside the Authenticator

> NORMATIVE
>
> The authenticator must not process a `Register` command without verifying the user (or enrolling the user, if this is the first time the user has used the authenticator).
>
> The authenticator must generate a unique UAuth key pair each time the Register command is called.
>
> The authenticator should either store key handle in its internal secure storage or cryptographically wrap it and export it to the ASM.
>
> For silent authenticators, the key handle must never be stored on a FIDO Server, otherwise this would enable tracking of users without providing the ability for users to clear key handles from the local device.
>
> If KeyID is not the key handle itself (e.g. such as in case of a second-factor roaming authenticator) - it must be a unique and unguessable byte array with a maximum length of 32 bytes. It must be unique within the scope of the AAID.
>
> > NOTE
> >
> > If the KeyID is generated randomly (instead of, for example, being derived from a key handle) - it should be stored inside RawKeyHandle so that it can be accessed by the authenticator while processing the Sign command.
>
> If the authenticator doesn't support `SignCounter` or `RegCounter` it must set these to 0 in TAG_UAFV1_KRD. The `RegCounter` must be set to 0 when a factory reset for the authenticator is performed. The `SignCounter` must be set to 0 when a factory reset for the authenticator is performed.

6.2.3 Sign Command

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

6.2.3.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_SIGN_CMD
1.1	UINT16 Length	Length of Command
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.3	UINT16 Tag	TAG_APPID (optional)
1.3.1	UINT16 Length	Length of AppID
1.3.2	UINT8[] AppID	AppID (max 512 bytes)
1.4	UINT16 Tag	TAG_FINAL_CHALLENGE
1.4.1	UINT16 Length	Length of Final Challenge
1.4.2	UINT8[] FinalChallenge	(binary value of) Final Challenge provided by ASM (max 32 bytes)
1.5	UINT16 Tag	TAG_TRANSACTION_CONTENT (optional)
1.5.1	UINT16 Length	Length of Transaction Content
1.5.2	UINT8[] TransactionContent	(binary value of) Transaction Content provided by ASM
1.6	UINT16 Tag	TAG_KEYHANDLE_ACCESS_TOKEN
1.6.1	UINT16 Length	Length of KHAccessToken
1.6.2	UINT8[] KHAccessToken	(binary value of) KHAccessToken provided by ASM (max 32 bytes)
1.7	UINT16 Tag	TAG_USERVERIFY_TOKEN (optional)
1.7.1	UINT16 Length	Length of the User Verification Token□
1.7.2	UINT8[] VerificationToken□	User Verification Token□
1.8	UINT16 Tag	TAG_KEYHANDLE (optional, multiple occurrences permitted)
1.8.1	UINT16 Length	Length of KeyHandle
1.8.2	UINT8[] KeyHandle	(binary value of) key handle

6.2.3.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_SIGN_CMD_RESPONSE
1.1	UINT16 Length	Entire Length of Command Response
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 Value	Status code returned by authenticator
1.3 **(choice 1)**	UINT16 Tag	TAG_USERNAME_AND_KEYHANDLE (optional, multiple occurances) This TLV tag contains multiple (>=1) {Username, Keyhandle} entries. If this tag is present, TAG_AUTHENTICATOR_ASSERTION must not be present
1.3.1	UINT16 Length	Length of the structure
1.3.2	UINT16 Tag	TAG_USERNAME
1.3.2.1	UINT16 Length	Length of Username

TLV Structure		Description
1.3.2.2	UINT8[] Username	Username
1.3.3	UINT16 Tag	TAG_KEYHANDLE
1.3.3.1	UINT16 Length	Length of `KeyHandle`
1.3.3.2	UINT8[] KeyHandle	(binary value of) key handle
1.3 **(choice 2)**	UINT16 Tag	TAG_AUTHENTICATOR_ASSERTION (optional) If this tag is present, TAG_USERNAME_AND_KEYHANDLE must not be present
1.3.1	UINT16 Length	Assertion Length
1.3.2	UINT8[] Assertion	Authentication assertion generated by the authenticator (see section TAG_UAFV1_AUTH_ASSERTION).

6.2.3.3 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ACCESS_DENIED`
- `UAF_CMD_STATUS_USER_NOT_ENROLLED`
- `UAF_CMD_STATUS_USER_CANCELLED`
- `UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT`
- `UAF_CMD_STATUS_ERR_UNKNOWN`

6.2.3.4 Command Description

> NOTE
>
> First-factor authenticators should implement this command in two stages.
>
> 1. The first stage will be executed only if the authenticator finds out that there are multiple key handles after filtering with the KHAccessToken. In this stage, the authenticator must return a list of usernames along with corresponding key handles
> 2. In the second stage, after the user selects a username, this command will be called with a single key handle and will return a UAF assertion based on this key handle
>
> If a second-factor authenticator is presented with more than one valid key handles, it must exercise only the first one and ignore the rest.
>
> The command is implemented in two stages to ensure that only one assertion can be generated for each command invocation.

Authenticators must take the following steps:

1. If this authenticator has a transaction confirmation display, and is able to display the AppID - make sure Command.TAG_APPID is provided, and show it on the display when verifying the user.
    - Update Command.KHAccessToken by mixing it with Command.TAG_APPID. An example of such a mixing function is a cryptographic hash function.
        - Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_APPID)
2. If the user is already enrolled with the authenticator (such as biometric enrollment, PIN setup, etc.) then verify the user. If the verification has already been done in one of the previous commands, make sure that `Command.TAG_USERVERIFY_TOKEN` is a valid token.
    1. If verification fails - return `UAF_CMD_STATUS_ACCESS_DENIED`
    2. If the user explicitly cancels the operation - return `UAF_CMD_STATUS_USER_CANCELLED`
3. If the user is not enrolled then return `UAF_CMD_STATUS_USER_NOT_ENROLLED`
4. Unwrap all provided key handles from Command.TAG_KEYHANDLE values using Wrap.sym
    1. If a first-factor roaming authenticator:
        - If Command.TAG_KEYHANDLE are provided, then the items in this list are KeyIDs. Use these KeyIDs to locate key handles stored in internal storage
        - If no Command.TAG_KEYHANDLE are provided - unwrap all key handles stored in internal storage
5. Filter RawKeyHandles with Command.KHAccessToken (RawKeyHandle.KHAccessToken == Command.KHAccessToken)
6. If the number of remaining RawKeyHandles is 0, then fail with `UAF_CMD_STATUS_ACCESS_DENIED`
7. If number of remaining RawKeyHandles is > 1

    1. If this is a second-factor authenticator, then choose the first RawKeyHandle only and jump to step #8.
    2. Copy {Command.KeyHandle, RawKeyHandle.username} for all remaining RawKeyHandles into TAG_USERNAME_AND_KEYHANDLE tag.
        - If this is a first-factor roaming authenticator, then the returned TAG_USERNAME_AND_KEYHANDLEs must be ordered by the key handle registration date (the latest-registered key handle must come the latest).
    3. Copy TAG_USERNAME_AND_KEYHANDLE into TAG_UAFV1_SIGN_CMD_RESPONSE and return

8. If number of remaining RawKeyHandles is 1
    1. Create TAG_UAFV1_SIGNED_DATA and set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x01
    2. If `TransactionContent` is not empty
        - If this is a silent authenticator, then return `UAF_CMD_STATUS_ACCESS_DENIED`
        - If the authenticator doesn't support transaction confirmation (it has set `transactionConfirmationDisplay` to 0 in the response to a `GetInfo` Command), then return `UAF_CMD_STATUS_ACCESS_DENIED`
        - If the authenticator has a built-in transaction confirmation display, then show `Command.TransactionContent` and `Command.TAG_APPID` (optional) on display and wait for the user to confirm it:
            - Return `UAF_CMD_STATUS_USER_CANCELLED` if user cancels the transaction
            - Return `UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT` if provided transaction content cannot be rendered
        - Compute hash of TransactionContent
            - TAG_UAFV1_SIGNED_DATA.TAG_TRANSACTION_CONTENT_HASH = hash(Command.TransactionContent)
            - Set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x02
    3. Create TAG_UAFV1_AUTH_ASSERTION
        - Fill in the rest of TAG_UAFV1_SIGNED_DATA fields
            - Increment SignCounter and put into TAG_UAFV1_SIGNED_DATA
            - Copy all the mandatory fields (see section TAG_UAFV1_AUTH_ASSERTION)
            - If TAG_UAFV1_SIGNED_DATA.AuthenticationMode == 0x01 - set TAG_UAFV1_SIGNED_DATA.TAG_TRANSACTION_CONTENT_HASH.Length to 0
        - Sign TAG_UAFV1_SIGNED_DATA with UAuth.priv
    4. Put the entire TLV structure for TAG_UAFV1_AUTH_ASSERTION as the value of TAG_AUTHENTICATOR_ASSERTION
    5. Copy TAG_AUTHENTICATOR_ASSERTION into TAG_UAFV1_SIGN_CMD_RESPONSE and return

NORMATIVE

Authenticator must not process Sign command without verifying the user first.

Authenticator must not reveal Username without verifying the user first.

bound authenticators must not process Sign command without validating KHAccessToken first.

UAuth.priv keys must never leave Authenticator's security boundary in plaintext form. UAuth.priv protection boundary is specified in `Metadata.keyProtection` field in Metadata [UAFAuthnrMetadata]).

If Authenticator's Metadata indicates that it does support Transaction Confirmation Display - it must display provided transaction content in this display and include the hash of content inside TAG_UAFV1_SIGNED_DATA structure.

Silent Authenticators must not operate in first-factor mode in order to follow the assumptions made in [FIDOSecRef].

If Authenticator doesn't support `SignCounter`, then it must set it to 0 in TAG_UAFV1_SIGNED_DATA. The `SignCounter` must be set to 0 when a factory reset for the Authenticator is performed, in order to follow the assumptions made in [FIDOSecRef].

Some Authenticators might support Transaction Confirmation display functionality not inside the Authenticator but within the boundaries of ASM. Typically these are software based Transaction Confirmation displays. When processing the Sign command with a given transaction such Authenticators should assume that they do have a builtin Transaction Confirmation display and should include the hash of transaction content in the final assertion without displaying anything to the user. Also, such Authenticator's Metadata file must clearly indicate the type of Transaction Confirmation display. Typically the flag of Transaction Confirmation display will be TRANSACTION_CONFIRMATION_DISPLAY_ANY or TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE. See [UAFRegistry] for flags describing Transaction Confirmation Display type.

6.2.4 Deregister Command

This command deletes a registered UAF credential from Authenticator.

6.2.4.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_DEREGISTER_CMD
1.1	UINT16 Length	Entire Command Length

TLV Structure		Description
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.3	UINT16 Tag	TAG_APPID (optional)
1.3.1	UINT16 Length	Length of AppID
1.3.2	UINT8[] AppID	AppID (max 512 bytes)
1.4	UINT16 Tag	TAG_KEYID
1.4.1	UINT16 Length	Length of KeyID
1.4.2	UINT8[] KeyID	(binary value of) KeyID provided by ASM
1.5	UINT16 Tag	TAG_KEYHANDLE_ACCESS_TOKEN
1.5.1	UINT16 Length	Length of KeyHandle Access Token
1.5.2	UINT8[] KHAccessToken	(binary value of) KeyHandle Access Token provided by ASM (max 32 bytes)

6.2.4.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_DEREGISTER_CMD_RESPONSE
1.1	UINT16 Length	Entire Length of Command Response
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 StatusCode	StatusCode returned by Authenticator

6.2.4.3 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ACCESS_DENIED`
- `UAF_CMD_STATUS_CMD_NOT_SUPPORTED`
- `UAF_CMD_STATUS_ERR_UNKNOWN`

6.2.4.4 Command Description

Authenticator must take the following steps:

1. If this authenticator has a Transaction Confirmation display and is able to display AppID, then make sure Command.TAG_APPID is provided
    - Update Command.KHAccessToken by mixing it with Command.TAG_APPID. An example of such mixing function is a cryptographic hash function.
        - Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_APPID)
2. If this Authenticator doesn't store key handles internally, then return `UAF_CMD_STATUS_CMD_NOT_SUPPORTED`
3. Find KeyHandle that matches Command.KeyID
4. Unwrap found key handles using Wrap.sym
5. Make sure that RawKeyHandle.KHAccessToken == Command.KHAccessToken
    - If not, then return `UAF_CMD_STATUS_ACCESS_DENIED`
6. Delete this KeyHandle from internal storage
7. Return `UAF_CMD_STATUS_OK`

> NORMATIVE
>
> bound authenticators must not process Deregister command without validating KHAccessToken first.
>
> Deregister command should not explicitly reveal whether the provided keyID was registered or not.

6.2.5 OpenSettings Command

This command instructs the Authenticator to open its built-in settings UI (e.g. change PIN, enroll new fingerprint, etc).

Authenticator must return `UAF_CMD_STATUS_CMD_NOT_SUPPORTED` if it doesn't support such functionality.

6.2.5.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_OPEN_SETTINGS_CMD
1.1	UINT16 Length	Entire Command Length
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index

6.2.5.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_OPEN_SETTINGS_CMD_RESPONSE
1.1	UINT16 Length	Entire Length of Command Response
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 StatusCode	StatusCode returned by Authenticator

6.2.5.3 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_CMD_NOT_SUPPORTED`
- `UAF_CMD_STATUS_ERR_UNKNOWN`

7. KeyIDs and key handles

This section is non-normative.

There are 4 types of Authenticators defined in this document and due to their specifics they behave differently while processing commands. One of the main differences between them is how they store and process key handles. This section tries to clarify it by describing the behavior of every type of Authenticator during the processing of relevant command.

7.1 first-factor Bound Authenticator

Register Command	Authenticator doesn't store key handles. Instead KeyHandle is always returned to ASM and stored in ASM database. KeyID is a randomly generated 32 bytes number (or simply the hash of KeyHandle).
Sign Command	When there is no user session (no cookies, a clear machine) Server doesn't provide any KeyID (since it doesn't know which KeyIDs to provide). In this scenario ASM selects all key handles and passes to Authenticator. During step-up authentication (when there is a user session) Server provides relevant KeyIDs. ASM selects key handles that correspond to provided KeyIDs and pass to Authenticator.
Deregister Command	Since Authenticator doesn't store key handles, then there is nothing to delete inside Authenticator. ASM finds the KeyHandle corresponding to provided KeyID and deletes it.

7.2 2ndF Bound Authenticator

Register Command	Authenticator doesn't store key handles. Instead KeyHandle is always returned to ASM and stored in ASM database. KeyID is a randomly generated 32 bytes number (or simply the hash of KeyHandle).

Sign Command	This Authenticator cannot operate without Server providing KeyIDs. Thus it can't be used when there is no user session (no cookies, a clear machine). During step-up authentication (when there is a user session) Server provides relevant KeyIDs. ASM selects key handles that correspond to provided KeyIDs and pass to Authenticator.
Deregister Command	Since Authenticator doesn't store key handles, then there is nothing to delete inside it. ASM finds the KeyHandle corresponding to provided KeyID and deletes it.

7.3 first-factor Roaming Authenticator

Register Command	Authenticator stores key handles inside its internal storage. KeyHandle is never returned back to ASM. KeyID is a randomly generated 32 bytes number (or simply the hash of KeyHandle)
Sign Command	When there is no user session (no cookies, a clear machine) Server doesn't provide any KeyID (since it doesn't know which KeyIDs to provide). In this scenario Authenticator uses all key handles that correspond to the provided AppID. During step-up authentication (when there is a user session) Server provides relevant KeyIDs. Authenticator selects key handles that correspond to provided KeyIDs and uses them.
Deregister Command	Authenticator finds the right KeyHandle and deletes it from its storage.

7.4 2ndF Roaming Authenticator

Register Command	Neither Authenticator nor ASM store key handles. Instead KeyHandle is sent to the Server (in place of KeyID) and stored in User's record. From Server's perspective it's a KeyID. In fact KeyID is the KeyHandle.
Sign Command	This Authenticator cannot operate without Server providing KeyIDs. Thus it can't be used when there is no user session (no cookies, a clear machine). During step-up authentication Server provides KeyIDs which are in fact key handles. Authenticator finds the right KeyHandle and uses it.
Deregister Command	Since Authenticator and ASM don't store key handles, then there is nothing to delete on client side.

8. Access Control for Commands

This section is normative.

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

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

All UAF Authenticators must satisfy the access control requirements defined below.

Authenticator vendors may offer additional security mechanisms.

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

Command	First-factor Bound Authenticator	2ndF Bound Authenticator	First-factor Roaming Authenticator	2ndF Roaming Authenticator
GetInfo	NoAuth	NoAuth	NoAuth	NoAuth
OpenSettings	NoAuth	NoAuth	NoAuth	NoAuth

Command	First-factor Bound Authenticator	2ndF Bound Authenticator	First-factor Roaming Authenticator	2ndF Roaming Authenticator
Register	UserVerify	UserVerify	UserVerify	UserVerify
Sign	UserVerify KHAccessToken KeyHandleList	UserVerify KHAccessToken KeyHandleList	UserVerify KHAccessToken	UserVerify KHAccessToken KeyHandleList
Deregister	KHAccessToken KeyID	KHAccessToken KeyID	KHAccessToken KeyID	KHAccessToken KeyID

Table 1: Access Control for Commands

9. Relationship to other standards

This section is non-normative.

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.

9.1 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.

9.2 Secure Elements

In order to support UAF inside Secure Element (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.

9.3 TPM

TPMs typically have a built-in attestation capability however the 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 FIDO 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.

9.4 Unreliable Transports

The command structures described in this document assume a reliable transport and provide no support at the application-layer to detect or correct for issues such as unreliable ordering, duplication, dropping or modification of messages. If the transport layer(s) between the ASM and Authenticator are not reliable, the non-normative private contract between the ASM and Authenticator may need to provide a means to detect and correct such errors.

A. Security Guidelines

This section is non-normative.

Category	Guidelines
AppIDs and KeyIDs	Registered AppIDs and KeyIDs must not be returned by an authenticator in plaintext, without first performing user verification. If an attacker gets physical access to a roaming authenticator, then it should not be easy to read out AppIDs and KeyIDs.

Category	Guidelines
Attestation Private Key	Authenticators must protect the attestation private key as a very sensitive asset. The overall security of the authenticator depends on the protection level of this key. It is highly recommended to store and operate this key inside a tamper-resistant hardware module, e.g. [SecureElement]. It is assumed by registration assertion schemes, that the authenticator has exclusive control over the data being signed with the attestation key. FIDO Authenticators must ensure that the attestation private key: 1. is only used to attest authentication keys generated and protected by the authenticator, using the FIDO-defined data structures, KeyRegistrationData. 2. is never accessible outside the security boundary of the authenticator. Attestation must be implemented in a way such that two different relying parties cannot link registrations, authentications or other transactions (see [UAFProtocol]).
Certifications	Vendors should strive to pass common security standard certifications with authenticators, such as [FIPS140-2], [CommonCriteria] and similar. Passing such certifications will positively impact the UAF implementation of the authenticator.
Cryptographic (Crypto) Kernel	The crypto kernel is a module of the authenticator implementing cryptographic functions (key generation, signing, wrapping, etc) necessary for UAF, and having access to UAuth.priv, Attestation Private Key and Wrap.sym. For optimal security, this module should reside within the same security boundary as the UAuth.priv, Att.priv and Wrap.sym keys. If it resides within a different security boundary, then 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 environment [TEE]. 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 a high quality entropy source, for: 1. generating authentication keys 2. generating signatures 3. computing authenticator-generated challenges The authenticator's random number generator (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 strong random numbers, it should fail safely. See the section of this table regarding random numbers
KeyHandle	It is highly recommended to use authenticated encryption while wrapping key handles with Wrap.sym. Algorithms such as AES-GCM and AES-CCM are most suitable for this operation.
Liveness Detection	The user verification method should include liveness detection [NISTCBiometrics], i.e. a technique to ensure that the sample submitted is actually from a (live) user. In the case of PIN-based matching, this could be implemented using [TEESecureDisplay] in order to ensure that malware can't emulate PIN entry.

Category	Guidelines
Matcher	By definition, the matcher component is part of the authenticator. This does not impose any restrictions on the authenticator implementation, but implementers need to make sure that there is a proper security boundary binding the matcher and the other parts of the authenticator together. Tampering with the matcher module may have significant security consequences. It is highly recommended for this module to reside within the integrity boundaries of the authenticator, and be capable of detecting tampering. It is highly recommended to run this module inside a trusted execution environment [TEE] or inside a secure element [SecureElement]. Authenticators which have separated matcher and CryptoKernel modules should implement mechanisms which would allow the CryptoKernel to securely receive assertions from the matcher module indicating the 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 an Authenticator receives an invalid UserVerificationToken it should treat this as an attack, and invalidate the cached UserVerificationToken. A UserVerificationToken should have a lifetime not exceeding 10 seconds. Authenticators must implement anti-hammering protections 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 the biometric data never leaves the security boundaries of authenticators. Matchers must only accept verification reference data enrolled by the user, i.e. they must not include any default PINs or default biometric reference data.
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 a very high probability. 3. They have a structure allowing only very few fields containing uncontrolled values, i.e. values which are neither generated nor verified by the authenticator
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. The (pseudo-)random numbers used by authenticators should successfully pass the randomness test specified in [Coron99] and they should follow the guidelines given in [SP800-90b]. Additionally, authenticators may choose to incorporate entropy provided by the FIDO Server via the `ServerChallenge` sent in requests (see [UAFProtocol]). When mixing multiple entropy sources, a suitable mixing function should be used, such as those described in [RFC4086].
RegCounter	The `RegCounter` provides an anti-fraud signal to the relying parties. Using the `RegCounter`, the relying party can detect authenticators which have been excessively registered. If the `RegCounter` is implemented: ensure that 1. it is increased by any registration operation and 2. it cannot be manipulated/modified otherwise (e.g. via API calls, etc.) A registration counter should be implemented as a global counter, i.e. one covering registrations to all AppIDs. This global counter should be increased by 1 upon any registration operation. Note: The RegCounter value should *not* be decreased by `Deregistration` operations.

Category	Guidelines
SignCounter	When an attacker is able to extract a Uauth.priv key from a registered authenticator, this key can be used independently from the original authenticator. This is considered cloning of an authenticator. Good protection measures of the Uauth private keys is one method to prevent cloning authenticators. In some situations the protection measures might not be sufficient. If the Authenticator maintains a signature counter `SignCounter`, then the FIDO Server would have an additional method to detect cloned authenticators. If the `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.) Signature counters should be implemented that are dedicated for each private key in order to preserve the user's privacy. A per-key `SignCounter` should be increased by 1, whenever the corresponding UAuth.priv key signs an assertion. A per-key `SignCounter` should be deleted whenever the corresponding UAuth key is deleted. If the authenticator is not able to handle many different signature counters, then a global signature counter covering all private keys should be implemented. A global `SignCounter` should be increased by a random positive integer value whenever any of the UAuth.priv keys is used to sign an assertion.
Transaction Confirmation Display	A transaction confirmation 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 [CLICKJACKING] for some examples of threats and potential counter-measures For more guidelines refer to [TEESecureDisplay].

UAuth.priv	An authenticator must protect all UAuth.priv keys as its **most** sensitive assets. The overall security of the authenticator depends **significantly** on the protection level of these keys.
	It is highly recommended that this key is generated, stored and operated inside a trusted execution environment.
	In situations where physical attacks and side channel attacks are considered within 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 the FIDO Server during registration and authentication operations should be mixed into the entropy pool in order to provide 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.
	    1. authenticating to the application (as identified by the AppID) they have been generated for, or
	    2. confirming transactions to the application (as identified by AppID) they have been generated for, or
	    3. are only being used to create the FIDO defined data structures, i.e. KRD, SignData.

Username	A username must not be returned in plaintext in any condition other than the conditions described for the SIGN command. In all other conditions usernames must be stored within a `KeyHandle`.
Verification Reference Data	The verification reference data, such as fingerprint templates or the reference value of a PIN, are by definition part of the authenticator. This does not impose any particular restrictions on the authenticator implementation, but implementers need to make sure that there is a proper security boundary binding all parts of the authenticator together..

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 the authenticator depends on the protection of this key. Wrap.sym key strength must be equal or higher than the strength of secrets stored in a RawKeyHandle. Refer to [SP800-57] and [SP800-38F] 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 to the caller.

B. Table of Figures

C. References

C.1 Normative references

[Coron99]
J. Coron and D. Naccache *An accurate evaluation of Maurer's universal test*. LNCS 1556, February 1999, URL: http://www.jscoron.fr/publications/universal.pdf
[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-glossary-v1.0-ps-20141208.html
PDF: fido-glossary-v1.0-ps-20141208.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). International Telecommunications Union, November 2008 URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en
[SP800-90b]
Elaine Baker and John Kelsey, *NIST Special Publication 800-90b: Recommendation for the Entropy Sources Used for Random Bit Generation*. National Institute of Standards and Technology, August 2012, URL: http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf
[UAFAuthnrMetadata]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO UAF Authenticator Metadata Statements v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-uaf-authnr-metadata-v1.0-ps-20141208.html
PDF: fido-uaf-authnr-metadata-v1.0-ps-20141208.pdf
[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges, *FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-uaf-protocol-v1.0-ps-20141208.html
PDF: fido-uaf-protocol-v1.0-ps-20141208.pdf
[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-uaf-reg-v1.0-ps-20141208.html
PDF: fido-uaf-reg-v1.0-ps-20141208.pdf

C.2 Informative references

[CLICKJACKING]
D. Lin-Shung Huang, C. Jackson, A. Moshchuk, H. Wang, S. Schlechter *Clickjacking: Attacks and Defenses*. USENIX, July 2012, URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf
[CommonCriteria]
CommonCriteria Publications. CCRA Members, Work in progress, accessed March 2014. URL: http://www.commoncriteriaportal.org/cc/
[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-security-ref-v1.0-ps-20141208.html
PDF: fido-security-ref-v1.0-ps-20141208.pdf
[FIPS140-2]
FIPS PUB 140-2: Security Requirements for Cryptographic Modules. National Institute of Standards and Technology, May 2001, URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
[NSTCBiometrics]
NSTC Subcommittee on Biometrics, *Biometrics Glossary*. National Science and Technology Council. 14 September 2006, URL: http://biometrics.gov/Documents/Glossary.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

[RFC4086]

D. Eastlake 3rd, J. Schiller, S. Crocker *Randomness Requirements for Security (RFC 4086)*, IETF, June 2005, URL: http://www.ietf.org/rfc/rfc4086.txt

[SP800-38F]

M. Dworkin, *NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*. National Institute of Standards and Technology, December 2012, URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf

[SP800-57]

Recommendation for Key Management – Part 1: General (Revision 3) SP800-57. July 2012. U.S. Department of Commerce/National Institute of Standards and Technology. URL: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf

[SecureElement]

GlobalPlatform Card Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp

[TEE]

GlobalPlatform Trusted Execution Environment Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp

[TEESecureDisplay]

GlobalPlatform Trusted User Interface API Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp

[TPM]

TPM Main Specification Trusted Computing Group. Accessed March 2014. URL: http://www.trustedcomputinggroup.org/resources/tpm_main_specification