FIDO UAF Authenticator Commands

FIDO Alliance Proposed Standard 20 October 2020

This version:  

Previous version:  
https://fidoalliance.org/specs/fido-uaf-v1.2-id-20180220/fido-uaf-authnr-cmds-v1.2-id-20180220.html

Editors:  
Dr. Rolf Lindemann, Nok Nok Labs, Inc.  
John Kemp, FIDO Alliance

Contributors:  
Davit Baghdasaryan, Nok Nok Labs, Inc.  
Roni Sasson, Discretix  
Brad Hill, PayPal, Inc.  
Jeff Hodges, PayPal, Inc.  
Ka Yang, Nok Nok Labs, Inc.

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

Copyright © 2013-2020 FIDO Alliance All Rights Reserved.

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://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 Alliance 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
  - 1.1 Conformance
- 2. Overview
- 3. UAF Authenticator
  - 3.1 Types of Authenticators
- 4. Tags
  - 4.1 Command Tags
  - 4.2 Tags used only in Authenticator Commands
  - 4.3 Tags used in UAF Protocol
  - 4.4 Status Codes
- 5. Structures
  - 5.1 RawKeyHandle
  - 5.2 Structures to be parsed by FIDO Server
    - 5.2.1 TAG_UAFV1_REG_ASSERTION
    - 5.2.2 TAG_UAFV1_AUTH_ASSERTION
  - 5.3 UserVerificationToken
- 6. Commands
  - 6.1 GetInfo Command
    - 6.1.1 Command Description
    - 6.1.2 Command Structure
    - 6.1.3 Command Response
    - 6.1.4 Status Codes
  - 6.2 Register Command
    - 6.2.1 Command Structure
    - 6.2.2 Command Response
    - 6.2.3 Status Codes
    - 6.2.4 Command Description
  - 6.3 Sign Command
    - 6.3.1 Command Structure
    - 6.3.2 Command Response
    - 6.3.3 Status Codes
    - 6.3.4 Command Description
  - 6.4 Deregister Command
    - 6.4.1 Command Structure
    - 6.4.2 Command Response
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 [FIDO Glossary].

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

1.1 Conformance

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

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** in this specification are to be interpreted as described in [RFC2119].
2. Overview

This section is non-normative.

This document 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

![Diagram of UAF Authenticator Commands]

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:

3. 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 or the user's presence 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

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 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.
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.

Conforming authenticators MUST implement at least one attestation type defined in [UAFRegistry], as well as one authentication algorithm and one key format listed in [FIDORegistry].

NOTE

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

4. 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:

```
<table>
<thead>
<tr>
<th>2 bytes</th>
<th>2 bytes</th>
<th>Length bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Length in bytes</td>
<td>Data</td>
</tr>
</tbody>
</table>
```

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 significant 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.

### 4.1 Command Tags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG_UAVF1_GETINFO_CMD</td>
<td>0x3401</td>
<td>Tag for GetInfo command.</td>
</tr>
<tr>
<td>TAG_UAVF1_GETINFO_CMD_RESPONSE</td>
<td>0x3601</td>
<td>Tag for GetInfo command response.</td>
</tr>
<tr>
<td>TAG_UAVF1_REGISTER_CMD</td>
<td>0x3402</td>
<td>Tag for Register command.</td>
</tr>
<tr>
<td>TAG_UAVF1_REGISTER_CMD_RESPONSE</td>
<td>0x3602</td>
<td>Tag for Register command response.</td>
</tr>
<tr>
<td>TAG_UAVF1_SIGN_CMD</td>
<td>0x3403</td>
<td>Tag for Sign command.</td>
</tr>
<tr>
<td>TAG_UAVF1_SIGN_CMD_RESPONSE</td>
<td>0x3603</td>
<td>Tag for Sign command response.</td>
</tr>
<tr>
<td>TAG_UAVF1_DEREGISTER_CMD</td>
<td>0x3404</td>
<td>Tag for Deregister command.</td>
</tr>
<tr>
<td>TAG_UAVF1_DEREGISTER_CMD_RESPONSE</td>
<td>0x3604</td>
<td>Tag for Deregister command response.</td>
</tr>
<tr>
<td>TAG_UAVF1_OPEN_SETTINGS_CMD</td>
<td>0x3406</td>
<td>Tag for OpenSettings command.</td>
</tr>
<tr>
<td>TAG_UAVF1_OPEN_SETTINGS_CMD_RESPONSE</td>
<td>0x3606</td>
<td>Tag for OpenSettings command response.</td>
</tr>
</tbody>
</table>

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

### 4.2 Tags used only in Authenticator Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG_KEYHANDLE</td>
<td>0x2801</td>
<td>Represents key handle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to [FIDO Glossary] for more information about key handle.</td>
</tr>
<tr>
<td>TAG_USERNAME_AND_KEYHANDLE</td>
<td>0x3802</td>
<td>Represents an associated Username and key handle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is a composite tag that contains a TAG_USERNAME and TAG_KEYHANDLE that identify a registration valid on the authenticator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to [FIDO Glossary] for more information about username.</td>
</tr>
<tr>
<td>TAG_USERVERIFY_TOKEN</td>
<td>0x2803</td>
<td>Represents a User Verification Token.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to [FIDO Glossary] for more information about user verification tokens.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A full AppID as a UINT8[] encoding of a UTF-8 string.</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>TAG_APPID</td>
<td>0x2804</td>
<td>Refer to [FIDOGlossary] for more information about AppID.</td>
</tr>
<tr>
<td>TAG_KEYHANDLE_ACCESS_TOKEN</td>
<td>0x2805</td>
<td>Represents a key handle Access Token.</td>
</tr>
<tr>
<td>TAG_USERNAME</td>
<td>0x2806</td>
<td>A Username as a UINT8[] encoding of a UTF-8 string.</td>
</tr>
<tr>
<td>TAG_ATTESTATION_TYPE</td>
<td>0x2807</td>
<td>Represents an Attestation Type.</td>
</tr>
<tr>
<td>TAG_STATUS_CODE</td>
<td>0x2808</td>
<td>Represents a Status Code.</td>
</tr>
<tr>
<td>TAG_AUTHENTICATOR_METADATA</td>
<td>0x2809</td>
<td>Represents a more detailed set of authenticator information.</td>
</tr>
<tr>
<td>TAG_ASSERTION_SCHEME</td>
<td>0x280A</td>
<td>A UINT8[] containing the UTF8-encoded Assertion Scheme as defined in [UAFRegistry]. (&quot;UAFV1TLV&quot;)</td>
</tr>
<tr>
<td>TAG_TC_DISPLAY_PNG_CHARACTERISTICS</td>
<td>0x280B</td>
<td>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 [FIDOMetadataStatement] for additional information on the format of this field.</td>
</tr>
<tr>
<td>TAG_TC_DISPLAY_CONTENT_TYPE</td>
<td>0x280C</td>
<td>A UINT8[] containing the UTF-8-encoded transaction display content type as defined in [FIDOMetadataStatement]. (&quot;image/png&quot;)</td>
</tr>
<tr>
<td>TAG_AUTHENTICATOR_INDEX</td>
<td>0x280D</td>
<td>Authenticator Index</td>
</tr>
<tr>
<td>TAG_API_VERSION</td>
<td>0x280E</td>
<td>API Version</td>
</tr>
<tr>
<td>TAG_AUTHENTICATOR_ASSERTION</td>
<td>0x280F</td>
<td>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.</td>
</tr>
<tr>
<td>TAG_TRANSACTION_CONTENT</td>
<td>0x2810</td>
<td>Represents transaction content sent to the authenticator.</td>
</tr>
<tr>
<td>TAG_AUTHENTICATOR_INFO</td>
<td>0x3811</td>
<td>Includes detailed information about authenticator's capabilities.</td>
</tr>
<tr>
<td>TAG_SUPPORTED_EXTENSION_ID</td>
<td>0x2812</td>
<td>Represents extension ID supported by authenticator.</td>
</tr>
<tr>
<td>TAG_TRANSACTIONCONFIRMATION_TOKEN</td>
<td>0x2813</td>
<td>Represents a token for transaction confirmation. It might be returned by the authenticator to the ASM and given back to the authenticator at a later stage. The meaning of it is similar to TAG_USERVERIFY_TOKEN, except that it is used for the user's approval of a displayed transaction text.</td>
</tr>
</tbody>
</table>

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

4.3 Tags used in UAF Protocol

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG_UAFV1_REG_ASSERTION</td>
<td>0x3E01</td>
<td>Authenticator response to Register command.</td>
</tr>
<tr>
<td>TAG_UAFV1_AUTH_ASSERTION</td>
<td>0x3E02</td>
<td>Authenticator response to Sign command.</td>
</tr>
<tr>
<td>TAG_UAFV1_KRD</td>
<td>0x3E03</td>
<td>Key Registration Data</td>
</tr>
<tr>
<td>TAG_UAFV1_SIGNED_DATA</td>
<td>0x3E04</td>
<td>Data signed by authenticator with the UAuth.priv key</td>
</tr>
<tr>
<td>TAG_ATTESTATION_CERT</td>
<td>0x2E05</td>
<td>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 <strong>MUST</strong> occur first. Each subsequent occurrence (if exists) <strong>MUST</strong> be the issuing certificate of the previous occurrence.</td>
</tr>
<tr>
<td>TAG_SIGNATURE</td>
<td>0x2E06</td>
<td>A cryptographic signature</td>
</tr>
<tr>
<td>ATTESTATION_BASIC_FULL</td>
<td>0x3E07</td>
<td>Full Basic Attestation as defined in [UAFProtocol]</td>
</tr>
<tr>
<td>ATTESTATION_BASIC_SURROGATE</td>
<td>0x3E08</td>
<td>Surrogate Basic Attestation as defined in [UAFProtocol]</td>
</tr>
<tr>
<td>ATTESTATION_ECDAA</td>
<td>0x3E09</td>
<td>Elliptic curve based direct anonymous attestation as defined in [UAFProtocol]. In this case the signature in TAG_SIGNATURE is an ECDAA signature as specified in [FIDOEcdaaAlgorithm].</td>
</tr>
<tr>
<td>TAG_KEYID</td>
<td>0x2E09</td>
<td>Represents a KeyID.</td>
</tr>
<tr>
<td>TAG_FINAL_CHALLENGE_HASH</td>
<td>0x2E0A</td>
<td>Represents a Hash of the Final Challenge. Refer to [UAFASM] for more information about the Final Challenge Hash.</td>
</tr>
<tr>
<td>TAG_AAID</td>
<td>0x2E0B</td>
<td>Represents an authenticator Attestation ID. Refer to [UAFProtocol] for more information about the AAID.</td>
</tr>
<tr>
<td>TAG_PUB_KEY</td>
<td>0x2E0C</td>
<td>Represents a Public Key.</td>
</tr>
<tr>
<td>TAG_COUNTERS</td>
<td>0x2E0D</td>
<td>Represents a use counters for the authenticator.</td>
</tr>
<tr>
<td>TAG_ASSERTION_INFO</td>
<td>0x2E0E</td>
<td>Represents assertion information necessary for message processing.</td>
</tr>
<tr>
<td>TAG_AUTHENTICATOR_NONCE</td>
<td>0x2E0F</td>
<td>Represents a nonce value generated by the authenticator. The Authenticator Nonce allows the authenticator to enforce the to-be-signed object being different each time it is generated - even under attack scenarios in which the caller (e.g. ASM) sends similar data. Side channels attacks are more difficult to perform if the data to-be-signed is different each time.</td>
</tr>
<tr>
<td>TAG_TRANSACTION_CONTENT_HASH</td>
<td>0x2E10</td>
<td>Represents a hash of transaction content.</td>
</tr>
<tr>
<td>TAG_EXTENSION</td>
<td>0x3E11, 0x3E12</td>
<td>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 <strong>MUST</strong> 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].</td>
</tr>
</tbody>
</table>

**NOTE**

This tag can be appended to any command and response.
### Table 4.3.1: Tags used in the UAF Protocol (0x2E00 - 0x2EFF, 0x3E00 - 0x3EFF). Normatively defined in [UAFRegistry]

#### 4.4 Status Codes

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAF_CMD_STATUS_OK</td>
<td>0x00</td>
<td>Success.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_ERR_UNKNOWN</td>
<td>0x01</td>
<td>An unknown error.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_ACCESS_DENIED</td>
<td>0x02</td>
<td>Access to this operation is denied.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_USER_NOT_ENROLLED</td>
<td>0x03</td>
<td>User is not enrolled with the authenticator and the authenticator cannot automatically trigger enrollment.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT</td>
<td>0x04</td>
<td>Transaction content cannot be rendered.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_USER_CANCELLED</td>
<td>0x05</td>
<td>User has cancelled the operation. No retry should be performed.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_CMD_NOT_SUPPORTED</td>
<td>0x06</td>
<td>Command not supported.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED</td>
<td>0x07</td>
<td>Required attestation not supported.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_PARAMS_INVALID</td>
<td>0x08</td>
<td>The parameters for the command received by the authenticator are malformed/invalid.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY</td>
<td>0x09</td>
<td>The UAuth key which is relevant for this command disappeared from the authenticator and cannot be restored. On some authenticators this error occurs when the user verification reference data set was modified (e.g. new fingerprint template added).</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_TIMEOUT</td>
<td>0x0a</td>
<td>The operation in the authenticator took longer than expected (due to technical issues) and it was finally aborted.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_USER_NOT_RESPONSIVE</td>
<td>0x0e</td>
<td>The user took too long to follow an instruction, e.g. didn't swipe the finger within the accepted time.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_INSUFFICIENT_RESOURCES</td>
<td>0x0f</td>
<td>Insufficient resources in the authenticator to perform the requested task.</td>
</tr>
</tbody>
</table>
The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that. Typically the user would have to enter an alternative password (formally: undergo some other alternative user verification method) to re-enable the use of the main user verification method.

**NOTE**

Any method the user can use to (re-) enable the main user verification method is considered an alternative user verification method and must be properly declared as such. For example, if the user can enter an alternative password to re-enable the use of fingerprints or to add additional fingers, the authenticator obviously supports fingerprint or password based user verification.

<table>
<thead>
<tr>
<th>UAF_CMD_STATUS_USER_LOCKOUT</th>
<th>0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that. Typically the user would have to enter an alternative password (formally: undergo some other alternative user verification method) to re-enable the use of the main user verification method.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UAF_CMD_STATUS_SYSTEM_INTERRUPTED</th>
<th>0x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system interrupted the operation. Retry might make sense.</td>
<td></td>
</tr>
</tbody>
</table>

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

5. Structures

*This section is normative.*

5.1 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.

<table>
<thead>
<tr>
<th>Depends on hashing algorithm (e.g. 32 bytes)</th>
<th>Depends on key type. (e.g. 32 bytes)</th>
<th>Username Size (1 byte)</th>
<th>Max 128 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>KHAccessToken</td>
<td>UAuth.priv</td>
<td>Size</td>
<td>Username</td>
</tr>
</tbody>
</table>

**Table 5.1: RawKeyHandle Structure**

First Factor authenticators *must* store Usernames in the authenticator and they *must* link the Username to the related key. This *may* be achieved by storing the Username inside the RawKeyHandle. Second Factor authenticators *must not* store the Username.

The ability to support Usernames is a key difference between first-, and second-factor authenticators.

The RawKeyHandle *must* be cryptographically wrapped before leaving the authenticator boundary since it typically contains sensitive information, e.g. the user authentication private key (UAuth.priv).
5.2 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 \texttt{TAG\_UAFV1\_KRD} and \texttt{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.

5.2.1 \texttt{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 \texttt{TAG\_UAFV1\_KRD} tag which among other data contains the newly generated \texttt{UAuth.pub}.

If the authenticator wants to append custom data to \texttt{TAG\_UAFV1\_KRD} structure (and thus sign with Attestation Key) - this data **MUST** be included as \texttt{TAG\_EXTENSION\_DATA} in a \texttt{TAG\_EXTENSION} object inside \texttt{TAG\_UAFV1\_KRD}.

If the authenticator wants to send additional data to FIDO Server without signing it - this data **MUST** be included as \texttt{TAG\_EXTENSION\_DATA} in a \texttt{TAG\_EXTENSION} object inside \texttt{TAG\_UAFV1\_REG\_ASSERTION} and not inside \texttt{TAG\_UAFV1\_KRD}.

Currently this document only specifies \texttt{ATTESTATION\_BASIC\_FULL}, \texttt{ATTESTATION\_BASIC\_SURROGATE} and \texttt{ATTESTATION\_ECDAA}. In case if the authenticator is required to perform "Some\_Other\_Attestation" on \texttt{TAG\_UAFV1\_KRD} - it **MUST** use the TLV tag and content defined for "Some\_Other\_Attestation" (defined in [FIDORegistry]).

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UINT16 Tag \texttt{TAG_UAFV1_REG_ASSERTION}</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length Length of the structure</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag \texttt{TAG_UAFV1_KRD}</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length Length of the structure</td>
</tr>
<tr>
<td>1.2.2</td>
<td>UINT16 Tag \texttt{TAG_AAID}</td>
</tr>
<tr>
<td>1.2.2.1</td>
<td>UINT16 Length Length of AAID</td>
</tr>
<tr>
<td>1.2.2.2</td>
<td>UINT8[] AAID Authenticator Attestation ID</td>
</tr>
<tr>
<td>1.2.3</td>
<td>UINT16 Tag \texttt{TAG_ASSERTION_INFO}</td>
</tr>
<tr>
<td>1.2.3.1</td>
<td>UINT16 Length Length of Assertion Information</td>
</tr>
<tr>
<td>1.2.3.2</td>
<td>UINT16 AuthenticatorVersion Vendor assigned authenticator version</td>
</tr>
<tr>
<td>1.2.3.3</td>
<td>UINT8 AuthenticationMode For Registration this must be 0x01 indicating that the user has explicitly verified the action.</td>
</tr>
<tr>
<td>1.2.3.4</td>
<td>UINT16 Signature Algorithm and Encoding of the attestation signature.</td>
</tr>
<tr>
<td>1.2.3.5</td>
<td>SignatureAlgAndEncoding</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>1.2.4</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.4.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.4.2</td>
<td>UINT8[] FinalChallengeHash</td>
</tr>
<tr>
<td>1.2.5</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.5.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.5.2</td>
<td>UINT8[] KeyID</td>
</tr>
<tr>
<td>1.2.6</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.6.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.6.2</td>
<td>UINT32 SignCounter</td>
</tr>
<tr>
<td>1.2.6.3</td>
<td>UINT32 RegCounter</td>
</tr>
<tr>
<td>1.2.7</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.7.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.7.2</td>
<td>UINT8[] PublicKey</td>
</tr>
<tr>
<td>1.3</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.3.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.3.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.3.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.3.2.2</td>
<td>UINT8[] Signature</td>
</tr>
<tr>
<td>1.3.3</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
previous occurrence. The last occurrence **must** be chained to one of the certificates included in field `attestationRootCertificate` in the related Metadata Statement `[FIDOMetadataStatement]`.

| 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)

| 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. |

1.3 (choice 3)

| 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 | The binary ECDAA signature as specified in [FIDOEcdaaAlgorithm]. |

### 5.2.2 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.

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.2.2</td>
<td>UINT8[] AAID</td>
</tr>
<tr>
<td>1.2.3</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.3.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.3.2</td>
<td>UINT16 AuthenticatorVersion</td>
</tr>
</tbody>
</table>
| 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 [FIDO Registry] 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, and NOT longer than 64 bytes. |
| 1.2.4.2 | UINT8[] AuthnrNonce | (binary value of) A nonce randomly generated by Authenticator |
| 1.2.5 | UINT16 Tag | TAG_FINAL_CHALLENGE_HASH |
| 1.2.5.1 | UINT16 Length | Length of Final Challenge Hash |
| 1.2.5.2 | UINT8[] FinalChallengeHash | (binary value of) Final Challenge Hash 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 |
| 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 its length field, MUST be included during signature computation.

5.3 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. 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.

NOTE

Examples of such different command interface include native key store or key chain APIs. It is important to declare whether the Uauth keys are restricted to sign valid FIDO UAF assertions only. See [FIDOMetadataStatement] entry.
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.

6.1 GetInfo Command

6.1.1 Command Description

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

6.1.2 Command Structure

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UINT16 Tag</td>
<td>TAG_UAFV1_GETINFO_CMD</td>
</tr>
<tr>
<td>1.1 UINT16 Length</td>
<td>Entire Command Length - must be 0 for this command</td>
</tr>
</tbody>
</table>

6.1.3 Command Response

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UINT16 Tag</td>
<td>TAG_UAFV1_GETINFO_CMD_RESPONSE</td>
</tr>
<tr>
<td>1.1 UINT16 Length</td>
<td>Response length</td>
</tr>
<tr>
<td>1.2 UINT16 Tag</td>
<td>TAG_STATUS_CODE</td>
</tr>
<tr>
<td>1.2.1 UINT16 Length</td>
<td>Status Code Length</td>
</tr>
<tr>
<td>1.2.2 UINT16 Value</td>
<td>Status Code returned by Authenticator</td>
</tr>
<tr>
<td>1.3 UINT16 Tag</td>
<td>TAG_API_VERSION</td>
</tr>
<tr>
<td>1.3.1 UINT16 Length</td>
<td>Length of API Version (must be 0x0001)</td>
</tr>
<tr>
<td>1.3.2 UINT8 Version</td>
<td>Authenticator API Version (must be 0x01). This version indicates the types of commands, and formatting associated with them, that are supported by the authenticator.</td>
</tr>
<tr>
<td>1.4 UINT16 Tag</td>
<td>TAG_AUTHENTICATOR_INFO (multiple occurrences possible)</td>
</tr>
<tr>
<td>1.4.1 UINT16 Length</td>
<td>Length of Authenticator Info</td>
</tr>
<tr>
<td>1.4.2 UINT16 Tag</td>
<td>TAG_AUTHENTICATOR_INDEX</td>
</tr>
<tr>
<td>1.4.2.1 UINT16 Length</td>
<td>Length of AuthenticatorIndex (must be 0x0001)</td>
</tr>
<tr>
<td>1.4.2.2 UINT8 AuthenticatorIndex</td>
<td>Authenticator Index</td>
</tr>
<tr>
<td>1.4.3 UINT16 Tag</td>
<td>TAG_AAID</td>
</tr>
<tr>
<td>1.4.3.1 UINT16 Length</td>
<td>Length of AAID</td>
</tr>
<tr>
<td>1.4.3.2 UINT8[] AAID</td>
<td>Vendor assigned AAID</td>
</tr>
<tr>
<td>1.4.4 UINT16 Tag</td>
<td>TAG_AUTHENTICATOR_METADATA</td>
</tr>
<tr>
<td>Field (1.4.4.1)</td>
<td>Data Type</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>Length</td>
<td>UINT16</td>
</tr>
</tbody>
</table>
| AuthenticatorType | UINT16 | 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.  
  - 0x0080 - Authenticator supports user verification tokens (UVTs) as described in this document. See section 5.3 UserVerificationToken.  
  - 0x0100 - Authenticator only accepts TAG_TRANSACTION_TEXT_HASH in Sign command. This flag **MAY ONLY** be set if TransactionConfirmationDisplay is set to 0x0003 (see section 6.3 Sign Command). |
| MaxKeyHandles | UINT8 | 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. |
| UserVerification | UINT32 | User Verification method (as defined in [FIDORegistry]). |
| KeyProtection | UINT16 | Key Protection type (as defined in [FIDORegistry]). |
| MatcherProtection | UINT16 | Matcher Protection type (as defined in [FIDORegistry]). |
| TransactionConfirmationDisplay | UINT16 | Transaction Confirmation type (as defined in [FIDORegistry]).  
  
  **NOTE**  
  If Authenticator doesn't support Transaction Confirmation - this value must be set to 0. |
| AuthenticationAlg | UINT16 | Authentication Algorithm (as defined in [FIDORegistry]). |
| Tag | UINT16 | TAG_TC_DISPLAY_CONTENT_TYPE (optional) |
| Length | UINT16 | Length of content type. |
| ContentType | UINT8[] | Transaction Confirmation Display Content Type. See [FIDOMetadataStatement] 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 [FIDOMetadataStatement] for additional information.

1.4.6.3 UINT32 Height
See [FIDOMetadataStatement] for additional information.

1.4.6.4 UINT8 BitDepth
See [FIDOMetadataStatement] for additional information.

1.4.6.5 UINT8 ColorType
See [FIDOMetadataStatement] for additional information.

1.4.6.6 UINT8 Compression
See [FIDOMetadataStatement] for additional information.

1.4.6.7 UINT8 Filter
See [FIDOMetadataStatement] for additional information.

1.4.6.8 UINT8 Interlace
See [FIDOMetadataStatement] for additional information.

1.4.6.9 UINT8[] PLTE
A PLTE packet descriptor, defined by 3 byte word.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>R</td>
<td>Red channel value</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>G</td>
<td>Green channel value</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>B</td>
<td>Blue channel value</td>
</tr>
</tbody>
</table>

See [FIDOMetadataStatement] 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 values are defined in [UAFRegistry] by the constants with the prefix TAG_ATTESTATION.

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

6.1.4 Status Codes

- UAF_CMD_STATUS_OK
- UAF_CMD_STATUS_ERR_UNKNOWN
- UAF_CMD_STATUS_PARAMS_INVALID

6.2 Register Command

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

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.2</td>
<td>UINT8 AuthenticatorIndex</td>
</tr>
<tr>
<td><strong>1.3</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.4</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.5</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.6</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.7</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.8</strong></td>
<td>UINT16 Tag</td>
</tr>
</tbody>
</table>

- **TAG_UAFV1_REGISTER_CMD**
- **Command Length**
- **TAG_AUTHENTICATOR_INDEX**
- **Length of AuthenticatorIndex (must be 0x0001)**
- **Authenticator Index**
- **TAG_APPID (optional)**
- **Length of AppID**
- **AppID (max 512 bytes)**
- **TAG_FINAL_CHALLENGE_HASH**
- **Final Challenge Hash Length**
- **Final Challenge Hash provided by ASM (max 32 bytes)**
- **TAG_USERNAME**
- **Length of Username**
- **Username provided by ASM (max 128 bytes)**
- **TAG_ATTESTATION_TYPE**
- **Length of AttestationType**
- **Attestation Type to be used**
- **TAG_KEYHANDLE_ACCESS_TOKEN**
- **Length of KHAccessToken**
- **KHAccessToken provided by ASM (max 32 bytes)**
- **TAG_USERVERIFY_TOKEN (optional)**
- **Length of VerificationToken**
- **User verification token**

### 6.2.2 Command Response

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.1</strong></td>
<td>UINT16 Length</td>
</tr>
<tr>
<td><strong>1.2</strong></td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td><strong>1.2.1</strong></td>
<td>UINT16 Length</td>
</tr>
</tbody>
</table>

- **TAG_UAFV1_REGISTER_CMD_RESPONSE**
- **Command Length**
- **TAG_STATUS_CODE**
- **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

### 6.2.3 Status Codes

- UAF_CMD_STATUS_OK
- UAF_CMD_STATUS_ERR_UNKNOWN
- UAF_CMD_STATUS_ACCESS_DENIED
- UAF_CMD_STATUS_USER_NOT_ENROLLED
- UAF_CMD_STATUS_USER_CANCELLED
- UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED
- UAF_CMD_STATUS_PARAMS_INVALID
- UAF_CMD_STATUS_TIMEOUT
- UAF_CMD_STATUS_USER_NOT_RESPONSIVE
- UAF_CMD_STATUS_INSUFFICIENT_RESOURCES
- UAF_CMD_STATUS_USER_LOCKOUT

### 6.2.4 Command Description

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

If the command structure is invalid (e.g. cannot be parsed correctly), return UAF_CMD_STATUS_PARAMS_INVALID.

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. Return UAF_CMD_STATUS_PARAMS_INVALID if Command.TAG_APPID is not provided in such case. 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.
   
   If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return UAF_CMD_STATUS_USER_LOCKOUT.

   1. If the user doesn't respond to the request to get verified - return UAF_CMD_STATUS_USER_NOT_RESPONSIVE
2. If verification fails - return UAF_CMD_STATUS_ACCESS_DENIED
3. If user explicitly cancels the operation - return UAF_CMD_STATUS_USER_CANCELLED

3. If the user is not enrolled with the authenticator then take the user through the enrollment process. If the enrollment process cannot be triggered by the authenticator, return UAF_CMD_STATUS_USER_NOT_ENROLLED.
   1. If the authenticator can trigger enrollment, but the user doesn't respond to the request to enroll - return UAF_CMD_STATUS_USER_NOT_RESPONSIVE
   2. If the authenticator can trigger enrollment, but enrollment fails - return UAF_CMD_STATUS_ACCESS_DENIED
   3. If the authenticator can trigger enrollment, but the user explicitly cancels the enrollment 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) If the process takes longer than accepted - return UAF_CMD_STATUS_TIMEOUT

6. Create a RawKeyHandle, for example as follows
   1. Add UAuth.priv to RawKeyHandle
   2. Add Command.KHAccessToken to RawKeyHandle
   3. If a first-factor authenticator, then add Command.Username to RawKeyHandle

If there are not enough resources in the authenticator to perform this task - return UAF_CMD_STATUS_INSUFFICIENT_RESOURCES.

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 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. Use UAF_CMD_STATUS_OK as status code
    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.
In the case of bound authenticators implementing a different command interface, the ASM could generate a temporary KeyID and provide it as input to the authenticator in a Register command and change it to the final KeyID (e.g. derived from the public key) when the authenticator has completed the Register command execution.

NOTE

If the KeyID is generated randomly (instead of, for example, being derived from a key handle or the public key) - 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.

6.3 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.

6.3.1 Command Structure

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TAM_UAFV1_SIGN_CMD</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Length of Command</td>
</tr>
<tr>
<td>1.2</td>
<td>TAG_AUTHENTICATOR_INDEX</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Length of AuthenticatorIndex (must be 0x0001)</td>
</tr>
<tr>
<td>1.3</td>
<td>Authenticator Index</td>
</tr>
<tr>
<td>1.4</td>
<td>TAM_FINAL_CHALLENGE_HASH</td>
</tr>
<tr>
<td>1.4.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Length of Final Challenge Hash</td>
</tr>
<tr>
<td>1.5</td>
<td>TAM_TRANSACTION_CONTENT</td>
</tr>
<tr>
<td>1.5.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Length of Transaction Content</td>
</tr>
<tr>
<td>1.5</td>
<td>TAM_TRANSACTION_CONTENT_HASH</td>
</tr>
<tr>
<td>1.5.2</td>
<td>(binary value of) Transaction Content provided by the ASM</td>
</tr>
</tbody>
</table>

NOTE

If the KeyID is generated randomly (instead of, for example, being derived from a key handle or the public key) - it should be stored inside RawKeyHandle so that it can be accessed by the authenticator while processing the Sign command.
1.5.1 UINT16 Length  
Length of Transaction Content Hash

1.5.2 UINT8[]  
TransactionContentHash (binary value of) Transaction Content Hash provided by the 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

### 6.3.2 Command Response

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.2</td>
<td>UINT16 Value</td>
</tr>
<tr>
<td>1.3 (choice 1)</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.3.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.3.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.3.2.1</td>
<td>UINT16 Length</td>
</tr>
</tbody>
</table>

This TLV tag can be used to convey multiple (>=1) {Username, Keyhandle} entries. Each occurrence of TAG_USERNAME_AND_KEYHANDLE contains one pair. If this tag is present, TAG_AUTHENTICATOR_ASSERTION must not be present.
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).

6.3.3 Status Codes

- UAF_CMD_STATUS_OK
- UAF_CMD_STATUS_ERR_UNKNOWN
- 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_PARAMS_INVALID
- UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY
- UAF_CMD_STATUS_TIMEOUT
- UAF_CMD_STATUS_USER_NOT_RESPONSIVE
- UAF_CMD_STATUS_USER_LOCKOUT

6.3.4 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:

If the command structure is invalid (e.g. cannot be parsed correctly), return UAF_CMD_STATUS_PARAMS_INVALID.

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. Return UAF_CMD_STATUS_PARAMS_INVALID if Command.TAG_APPID is not provided in such case.
   - 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 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 and the Authenticator implements displaying transaction text before user verification, then show Command.TransactionContent and Command.TAG_APPID (optional) on display and wait for the user to confirm it by passing user verification (see step below):
     - Return UAF_CMD_STATUS_USER_NOT_RESPONSIVE if the user doesn't respond.
     - Return UAF_CMD_STATUS_USER_CANCELLED if the user cancels the transaction.
     - Return UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT if the 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. 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.
   - If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return UAF_CMD_STATUS_USER_LOCKOUT.
     1. If the user doesn't respond to the request to get verified - return UAF_CMD_STATUS_USER_NOT_RESPONSIVE
     2. If verification fails - return UAF_CMD_STATUS_ACCESS_DENIED
     3. If the user explicitly cancels the operation - return UAF_CMD_STATUS_USER_CANCELLED
   - 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 this is 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

   If no RawKeyHandles are found - return UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY.

NOTE

This should not occur as the Uauth key must be protected by the authenticator's user verification method. If the authenticator supports alternative user verification methods (e.g. alternative password and finger print verification and the alternative password must be provided before enrolling a finger and only the finger print is verified as part of the Register or Sign operation, then the authenticator should automatically and implicitly ask the user to enroll the modality required in the operation (instead of just returning an error).
6. Filter RawKeyHandles with Command.KHAccessToken (RawKeyHandle.KHAccessToken == Command.KHAccessToken)

7. If the number of remaining RawKeyHandles is 0, then fail with UAF_CMD_STATUS_ACCESS_DENIED

8. If number of remaining RawKeyHandles is > 1
   1. If this authenticator has a user interface and wants to use it for this purpose: Ask the user which of the usernames he wants to use for this operation. Select the related RawKeyHandle and jump to step #8.
   2. If this is a second-factor authenticator, then choose the first RawKeyHandle only and jump to step #8.
   3. 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).

   **NOTE**
   If two or more key handles with the same username are found, a first-factor roaming authenticator may only keep the one that is registered most recently and delete the rest. This avoids having unusable (old) private key in the authenticator which (surprisingly) might become active after deregistering the newly generated one.

4. Copy TAG_USERNAME_AND_KEYHANDLE into TAG_UAFV1_SIGN_CMD_RESPONSE and return

9. If number of remaining RawKeyHandles is 1
   1. If the Uauth key related to the RawKeyHandle cannot be used or disappeared and cannot be restored - return UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY.
   2. Create TAG_UAFV1_SIGNED_DATA and set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x01
   3. If TransactionContent is not empty
      - If the authenticator has a built-in transaction confirmation display and the authenticator implements displaying transaction text after user verification, then show Command.TransactionContent and Command.TAG_APPID (optional) on display and wait for the user to confirm it:
        - Return UAF_CMD_STATUS_USER_NOT_RESPONSIVE if the user doesn't respond.
        - Return UAF_CMD_STATUS_USER_CANCELLED if the user cancels the transaction.
        - Return UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT if the 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

4. If TransactionContent is not set, but TransactionContentHash is not empty
   - If this is a silent authenticator, then return UAF_CMD_STATUS_ACCESS_DENIED
   - If the conditions for receiving TransactionContentHash are not satisfied (if the authenticator's TransactionConfirmationDisplay is NOT set to 0x0003 in the response to a GetInfo Command), then return UAF_CMD_STATUS_PARAMS_INVALID
     - Perform the following steps
       - TAG_UAFV1_SIGNED_DATA.TAG_TRANSACTION_CONTENT_HASH = Command.TransactionContentHash
       - Set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x02

5. Create TAG_UAFV1_AUTH_ASSERTION
   - Fill in the rest of TAG_UAFV1_SIGNED_DATA fields
   - Perform the following steps
     - 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
If these steps take longer than expected by the authenticator - return UAF_CMD_STATUS_TIMEOUT.

6. Put the entire TLV structure for TAG_UAFV1_AUTH_ASSERTION as the value of TAG_AUTHENTICATOR_ASSERTION

7. 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.

Bound authenticators implementing a different command interface, **MAY** implement a different method for binding keys to a specific AppId, if such method provides at least the same security level (i.e. relying the OS/platform to determine the calling App). See [UAFASM] section "KHAccessToken" for more details.

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 [FIDOMetadataStatement]).

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.

Authenticators supporting Transaction Confirmation Display **SHALL** either display the transaction text before user verification (see step #2) or after it (see step 9.3). Displaying the transaction text before user verification is preferred.

Silent Authenticators **MUST** not operate in first-factor mode in order to follow the assumptions made in [FIDOSecRef]. However, a native App or web page could "cache" the keyHandle or a Cookie and hence would be considered a first-factor that could be combined with a Silent Authenticator (when doing so).

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 built-in 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 [FIDORegistry] for flags describing Transaction Confirmation Display type.

6.4 Deregister Command

This command deletes a registered UAF credential from Authenticator.

6.4.1 Command Structure

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td></td>
<td>TAG_UAFV1_DEREGISTER_CMD</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td></td>
<td>Entire Command Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td></td>
<td>TAG_AUTHENTICATOR_INDEX</td>
</tr>
</tbody>
</table>
1.2.1 UINT16 Length  
Length of Authenticator Index (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)

6.4.2 Command Response

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2.2</td>
<td>UINT16 StatusCode</td>
</tr>
</tbody>
</table>

6.4.3 Status Codes

- UAF_CMD_STATUS_OK
- UAF_CMD_STATUS_ERR_UNKNOWN
- UAF_CMD_STATUS_ACCESS_DENIED
- UAF_CMD_STATUS_CMD_NOT_SUPPORTED
- UAF_CMD_STATUS_PARAMS_INVALID

6.4.4 Command Description

Authenticator must take the following steps:

If the command structure is invalid (e.g. cannot be parsed correctly), return UAF_CMD_STATUS_PARAMS_INVALID.

1. If this authenticator has a Transaction Confirmation display and is able to display AppID, then make sure Command.TAG_APPID is provided. Return UAF_CMD_STATUS_PARAMS_INVALID if Command.TAG_APPID is not provided in such case.
   - 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. If the length of **TAG_KEYID** is zero (i.e., 0000 Hex), then
   - if **TAG_APPID** is provided, then
     1. if RawKeyHandle.KHAccessToken == Command.KHAccessToken, then delete KeyHandle from internal storage, otherwise, note an error occurred
     2. if an error occurred, then return UAF_CMD_STATUS_ACCESS_DENIED
   - if **TAG_APPID** is not provided, then delete all KeyHandles from internal storage where RawKeyHandle.KHAccessToken == Command.KHAccessToken
   - Go to step 5

4. If the length of **TAG_KEYID** is NOT zero, then
   - Find KeyHandle that matches Command.KeyID
   - Ensure that RawKeyHandle.KHAccessToken == Command.KHAccessToken
     - If not, then return UAF_CMD_STATUS_ACCESS_DENIED
   - Delete this KeyHandle from internal storage

5. Return UAF_CMD_STATUS_OK

**NOTE**
The authenticator must unwrap the relevant KeyHandles using Wrap.sym as needed.

**NORMATIVE**
Bound authenticators **MUST NOT** process Deregister command without validating KHAccessToken first.

Bound authenticators implementing a different command interface, **MAY** implement a different method for binding keys to a specific AppID, if such method provides at least the same security level (i.e. relying the OS/platform to determine the calling App). See [UAFASM] section "KHAccessToken" for more details.

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

**NOTE**
This command *never* returns UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY as this could reveal the keyID registration status.

### 6.5 OpenSettings Command

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

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

If the command structure is invalid (e.g. cannot be parsed correctly), the authenticator must return UAF_CMD_STATUS_PARAMS_INVALID.

#### 6.5.1 Command Structure

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TAG_UAFV1_OPEN_SETTINGS_CMD</td>
</tr>
<tr>
<td></td>
<td>Entire Command Length</td>
</tr>
<tr>
<td></td>
<td>TAG_AUTHENTICATOR_INDEX</td>
</tr>
</tbody>
</table>
### 6.5.2 Command Response

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UINT16 Tag</td>
<td>TAG_UAFV1_OPEN_SETTINGS_CMD_RESPONSE</td>
</tr>
<tr>
<td>1.1 UINT16 Length</td>
<td>Entire Length of Command Response</td>
</tr>
<tr>
<td>1.2 UINT16 Tag</td>
<td>TAG_STATUS_CODE</td>
</tr>
<tr>
<td>1.2.1 UINT16 Length</td>
<td>Status Code Length</td>
</tr>
<tr>
<td>1.2.2 UINT16 StatusCode</td>
<td>StatusCode returned by Authenticator</td>
</tr>
</tbody>
</table>

### 6.5.3 Status Codes

- **UAF_CMD_STATUS_OK**
- **UAF_CMD_STATUS_ERR_UNKNOWN**
- **UAF_CMD_STATUS_CMD_NOT_SUPPORTED**
- **UAF_CMD_STATUS_PARAMS_INVALID**

### 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 the KeyHandle or the public key). |
|------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Sign Command     | When there is no user session (no cookies, a clear machine) the Server doesn't provide any KeyID (since it doesn't know which KeyIDs to provide). In this scenario the ASM selects all key handles and passes them 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. ASC finds the KeyHandle corresponding to provided KeyID and deletes it. |

#### 7.2 2ndF Bound Authenticator
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register</strong> Command</td>
<td>Authenticator might not store key handles. Instead the KeyHandle might be returned to the ASM and stored in the ASM database. KeyID is a randomly generated 32 bytes number (or simply the hash of the KeyHandle or the public key).</td>
</tr>
<tr>
<td><strong>Sign</strong> Command</td>
<td>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); unless, for example, the user identifies their account and the server is then able to provide a KeyID. 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.</td>
</tr>
<tr>
<td><strong>Deregister</strong> Command</td>
<td>If the Authenticator doesn't store key handles, then there is nothing to delete inside it. The ASM finds the KeyHandle corresponding to provided KeyID and deletes it.</td>
</tr>
</tbody>
</table>

7.3 first-factor Roaming Authenticator

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register</strong> Command</td>
<td>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)</td>
</tr>
<tr>
<td><strong>Sign</strong> Command</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Deregister</strong> Command</td>
<td>Authenticator finds the right KeyHandle and deletes it from its storage.</td>
</tr>
</tbody>
</table>

7.4 2ndF Roaming Authenticator

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register</strong> Command</td>
<td>Typically neither the Authenticator nor the ASM store key handles. Instead the 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 the KeyID is identical to the KeyHandle.</td>
</tr>
<tr>
<td><strong>Sign</strong> Command</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Deregister</strong> Command</td>
<td>Since Authenticator and ASM don't store key handles, then there is nothing to delete on client side.</td>
</tr>
</tbody>
</table>

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 (or alternative method with similar security level must be used)
- KeyHandleList - must be known to the caller
- KeyID - must be known to the caller

<table>
<thead>
<tr>
<th>Command</th>
<th>First-factor Bound Authenticator</th>
<th>2ndF Bound Authenticator</th>
<th>First-factor Roaming Authenticator</th>
<th>2ndF Roaming Authenticator</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetInfo</td>
<td>NoAuth</td>
<td>NoAuth</td>
<td>NoAuth</td>
<td>NoAuth</td>
</tr>
<tr>
<td>OpenSettings</td>
<td>NoAuth</td>
<td>NoAuth</td>
<td>NoAuth</td>
<td>NoAuth</td>
</tr>
<tr>
<td>Register</td>
<td>UserVerify</td>
<td>UserVerify</td>
<td>UserVerify</td>
<td>UserVerify</td>
</tr>
<tr>
<td>Sign</td>
<td>UserVerify KHAccessToken KeyHandleList</td>
<td>UserVerify KHAccessToken KeyHandleList</td>
<td>UserVerify KHAccessToken</td>
<td>UserVerify KHAccessToken KeyHandleList</td>
</tr>
<tr>
<td>Deregister</td>
<td>KHAccessToken KeyID</td>
<td>KHAccessToken KeyID</td>
<td>KHAccessToken KeyID</td>
<td>KHAccessToken KeyID</td>
</tr>
</tbody>
</table>

Table 1: Access Control for Commands

9. Considerations

This section is non-normative.

9.1 Algorithms and Key Sizes

The proposed algorithms and key sizes are chosen such that compatibility to TPMv2 is possible.

9.2 Indicating the Authenticator Model

Some authenticators (e.g. TPMv2) do not have the ability to include their model identifier (i.e. vendor ID and model name) in attested messages (i.e. the to-be-signed part of the registration assertion). The TPM's endorsement key certificate typically contains that information directly or at least it allows the model to be derived from the endorsement key certificate.

In FIDO, the relying party expects the ability to cryptographically verify the authenticator model (i.e. AAID).

If the authenticator cannot securely include its model (i.e. AAID) in the registration assertion (i.e. in the KRD object), we require the ECDAA-Issuers public key (ipkk) to be dedicated to one single authenticator model (identified by its AAID).

Using this method, the issuer public key is uniquely related to one entry in the Metadata Statement and can be used by the FIDO server to get a cryptographic proof of the Authenticator model.

10. 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.

10.1 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.

10.2 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 with the Applet.

10.3 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 [FIDOMetadataStatement]) 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.

10.4 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

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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppIDs and KeyIDs</td>
<td>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.</td>
</tr>
</tbody>
</table>
| **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]). |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Certifications</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| **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.

It is highly recommended to use authenticated encryption while wrapping key handles with Wrap.sym. |
<table>
<thead>
<tr>
<th><strong>KeyHandle</strong></th>
<th>Algorithms such as AES-GCM and AES-CCM are most suitable for this operation.</th>
</tr>
</thead>
</table>
| **Liveness Detection / Presentation Attack Detection** | The user verification method should include liveness detection [NSTCBiometrics], 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. |
| **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 |
| | 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. |
### Random Numbers

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.

### 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.

---

**NOTE**

There are multiple reasons why the `SignCounter` value could be 0 in a registration response. A `SignCounter` value of 0 in an authentication response indicates that the authenticator doesn't support the `SignCounter` concept.

---

A transaction confirmation display must ensure that the user is presented with the provided transaction
<table>
<thead>
<tr>
<th>Transaction Confirmation Display</th>
<th>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].</th>
</tr>
</thead>
</table>
| **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. |
| 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

Fig. 1 UAF Authenticator Commands
Fig. 2 FIDO Authenticator Logical Sub-Components

C. References

C.1 Normative references

[Coron99]

[FIDOEcdaaAlgorithm]

[FIDOGlossary]

[FIDOMetadataStatement]

[FIDORegistry]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Registry of Predefined Values. Proposed Standard. URL: https://fidoalliance.org/specs/common-scheme/fido-registry-v2.1-ps-20191217.html

[ITU-X690-2008]

[RFC2119]

[SP800-90b]
Meltem Sönmez Turan; Elaine Barker; John Kelsey; Kerry McKay; Mary Baish; Michael Boyle. NIST Special Publication 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation. January 2018. URL: https://csrc.nist.gov/publications/detail/sp/800-90b/final

[UAFProtocol]

[UAFRegistry]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of Predefined Values. Review Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-registry-v2.0-id-20180227.html

C.2 Informative references

[CLICKJACKING]

[CommonCriteria]

[FIDOSecRef]

[FIPS140-2]

[NSTCBiometrics]


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