FIDO UAF Android Protected Confirmation Assertion Format

FIDO Alliance Proposed Standard 20 October 2020

This version:  
https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-apccbor-v1.2-ps-20201020.html

Editor:  
Dr. Rolf Lindemann, 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

This document defines the assertion format "APCV1CBOR" in order to use Android Protected Confirmation for FIDO UAF Transaction Confirmation.

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 NONINFRINGEMENT, 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 Key Words
- 2. Overview
- 3. Data Structures for APCV1CBOR
  - 3.1 Registration Assertion
  - 3.2 Authentication Assertion
- 4. Processing Rules
  - 4.1 Registration Response Processing Rules for ASM
  - 4.2 Registration Response Processing Rules for FIDO Server
  - 4.3 Authentication Response Generation Rules for ASM
  - 4.4 Authentication Response Processing Rules for FIDO Server
- 5. Example for FIDO Metadata Statement
- A. References
  - A.1 Normative references
  - A.2 Informative references

1. Notation

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

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

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

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

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

1.1 Key Words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

2. Overview
This document defines the assertion format "APCV1CBOR" in order to use Android Protected Confirmation for FIDO Transaction Confirmation.

3. Data Structures for APCV1CBOR

3.1 Registration Assertion

The registration assertion for the assertion format "APCV1CBOR" contains an object as specified in section 5.2.1 in [UAFAuthnrCommands], with the following specifics:

1. Only Surrogate Basic Attestation is supported. The extension "fido.uaf.android.key_attestation" [UAFRegistry] MUST be present.
2. The signature field (TAG_SIGNATURE) SHALL have zero bytes length, since the key cannot be used to create a self-signature.

3.2 Authentication Assertion

The authentication assertion is a TLV structure containing a CBOR encoded to-be-signed object:

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TAG_APCV1CBOR_AUTH_ASSERTION</td>
</tr>
<tr>
<td>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td></td>
<td>Length of the structure.</td>
</tr>
<tr>
<td>1.2</td>
<td>TAG_APCV1CBOR_SIGNED_DATA</td>
</tr>
<tr>
<td>1.2.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td></td>
<td>Length of the structure.</td>
</tr>
<tr>
<td>1.2.2</td>
<td>UINT8 tbsData</td>
</tr>
<tr>
<td></td>
<td>The serialized Android Protected Confirmation CBOR object.</td>
</tr>
<tr>
<td>1.3</td>
<td>TAG_AAID</td>
</tr>
<tr>
<td>1.3.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td></td>
<td>Length of AAID</td>
</tr>
<tr>
<td>1.3.2</td>
<td>UINT8[] AAID</td>
</tr>
<tr>
<td></td>
<td>Authenticator Attestation ID</td>
</tr>
<tr>
<td>1.4</td>
<td>TAG_KEYID</td>
</tr>
<tr>
<td>1.4.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td></td>
<td>Length of KeyID</td>
</tr>
<tr>
<td>1.4.2</td>
<td>UINT8[]</td>
</tr>
<tr>
<td></td>
<td>(binary value of) KeyID</td>
</tr>
</tbody>
</table>
### KeyID

<table>
<thead>
<tr>
<th>KeyID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.5.1</td>
<td>UINT16</td>
</tr>
<tr>
<td>1.5.2</td>
<td>UINT8[]</td>
</tr>
</tbody>
</table>

**NOTE**

Only the data in tbsData is included in the signature computation. All other fields are essentially unauthenticated and are treated as 'hints' only.

### 4. Processing Rules

*This section is normative.*

#### 4.1 Registration Response Processing Rules for ASM

Refer to [UAFAuthnrCommands] document for more information about the TAGs and structure mentioned in this paragraph.

1. Locate authenticator using authenticatorIndex. If the authenticator cannot be located, then fail with UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED.

2. If a user is already enrolled with this authenticator (such as biometric enrollment, PIN setup, etc. for example) then the ASM **MUST** request that the authenticator verifies the user.

   **NOTE**

   If the authenticator supports UserVerificationToken (see [UAFAuthnrCommands]), then the ASM must obtain this token in order to later include it with the Register command.

3. If the user is not enrolled with the authenticator then take the user through the enrollment process.
   - If neither the ASM nor the Authenticator can trigger the enrollment process, return UAF_ASM_STATUS_USER_NOT_ENROLLED.
   - If enrollment fails, return UAF_ASM_STATUS_ACCESS_DENIED.

4. Hash the provided RegisterIn.finalChallenge using the authenticator-specific hash function (FinalChallengeHash)

   An authenticator's preferred hash function information **MUST** meet the algorithm defined in the
5. Generate a key pair with appropriate protection settings and mark it for use with Android Protected Confirmation, see https://developer.android.com/training/articles/security-android-protected-confirmation.

6. Create a TAG_AUTHENTICATOR_ASSERTION structure containing a TAG_UAFV1_REG_ASSERTION object with the following specifics:
   1. set signature of Surrogate Basic Attestation to 0 bytes length
   2. add the Android Hardware Key Attestation extension

7. If the authenticator is a bound authenticator
   1. Store CallerID (see [UAFASM]), AppID, TAG_KEYHANDLE, TAG_KEYID and CurrentTimestamp in the ASM's database.

```
NOTE

What data an ASM will store at this stage depends on underlying authenticator's architecture. For example some authenticators might store AppID, KeyHandle, KeyID inside their own secure storage. In this case ASM doesn't have to store these data in its database.
```

8. Create a RegisterOut object
   1. Set RegisterOut.assertionScheme according to "APCV1CBOR"
   2. Encode the content of TAG_AUTHENTICATOR_ASSERTION (i.e. TAG_UAFV1_REG_ASSERTION) in base64url format and set as RegisterOut.assertion as described in section "Data Structures for APCV1CBOR".
   3. Return RegisterOut object

4.2 Registration Response Processing Rules for FIDO Server

Instead of skipping the assertion as described in step 6.9, follow these rules:

1. if a.assertionScheme == "APCV1CBOR" AND a.assertion.TAG_UAFV1_REG_ASSERTION contains TAG_UAFV1_KRD as first element:
   1. Obtain Metadata(AAID).AttestationType for the AAID and make sure that a.assertion.TAG_UAFV1_REG_ASSERTION contains the most preferred attestation tag specified in field MatchCriteria.attestationTypes in RegistrationRequest.policy (if this field is present).
      - If a.assertion.TAG_UAFV1_REG_ASSERTION doesn't contain the preferred attestation - it is RECOMMENDED to skip this assertion and continue with next one
   2. Make sure that a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.FinalChallengeHash == FCHash
      - If comparison fails - continue with next assertion
   3. Obtain Metadata(AAID).AuthenticatorVersion for the AAID and make sure that it is lower or equal to a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.AuthenticatorVersion.
      - If Metadata(AAID).AuthenticatorVersion is higher (i.e. the authenticator firmware is outdated), it is RECOMMENDED to assume increased risk. See sections "StatusReport
4. Check whether `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.RegCounter` is 0 since it is not supported in this assertion scheme.
   - If `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.RegCounter` is non-zero, this assertion might be skipped and processing will continue with next one

5. Make sure `a.assertion.TAG_UAFV1_REG_ASSERTION` contains an object of type `ATTESTATION_BASIC_SURROGATE`
   1. There is no real attestation for the AAID, so we just assume the AAID is the real one.
   2. If entry `AttestationRootCertificates` for the AAID in the metadata is not empty - continue with next assertion (as the AAID obviously is expecting a different attestation method).
   3. Verify that extension "fido.uaf.android.key_attestation" is present and check whether it is positively verified according to its server processing rules as specified [UAFRegistry].
      - If verification fails – continue with next assertion
   4. Verify that the attestation statement included in that extension includes the flag `TRUSTED_CONFIRMATION_REQUIRED` indicating that the key will be restricted to sign valid transaction confirmation assertions (see `https://developer.android.com/training/articles/security-key-attestation` and `https://developer.android.com/training/articles/security-android-protected-confirmation`).
      - If verification fails – continue with next assertion
   5. Mark assertion as positively verified

6. Extract `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.PublicKey` into `PublicKey`,
   `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.KeyID` into `KeyID`,
   `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.SignCounter` into `SignCounter`,
   `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_ASSERTION_INFO.authenticatorVersion` into `AuthenticatorVersion`,
   `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_AAID` into `AAID`.

4.3 Authentication Response Generation Rules for ASM

See [UAFASM] for details of the ASM API.

1. if this is a bound authenticator, verify `callerid` against the one stored at registration time and return `UAF_ASM_STATUS_ACCESS_DENIED` if it doesn't match.

2. The ASM **MUST** request the authenticator to verify the user.

3. Hash the provided `AuthenticateIn.finalChallenge` using the preferred authenticator-specific hash function (`FinalChallengeHash`).
   The authenticator's preferred hash function information **MUST** meet the algorithm defined in the `AuthenticatorInfo.authenticationAlgorithm` field.

4. If `AuthenticateIn.keyIDs` is not empty,
   1. If this is a bound authenticator, then look up ASM's database with `AuthenticateIn.appID` and `AuthenticateIn.keyIDs` and obtain the KeyHandles associated with it.
      - Return `UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY` if the related key disappeared permanently from the authenticator.

   **UAF_ASM_STATUS_ACCESS_DENIED**
2. If this is a roaming authenticator, then treat `AuthenticateIn.keyIDs` as KeyHandles.

5. If `AuthenticateIn.keyIDs` is empty, lookup all KeyHandles matching this request.

6. If multiple KeyHandles exist that match this request, show the related distinct usernames and ask the user to choose a single username. Remember the KeyHandle related to this key.

7. Call `ConfirmationPrompt.Builder` and pass the `transactionText` as parameter to method `setPromptText` see also https://developer.android.com/training/articles/security-android-protected-confirmation.


9. Call `build` method of the `ConfirmationPrompt` and then call method `presentPrompt` providing an appropriate callback that will sign the `dataThatWasConfirmed` with the key identified by the KeyHandle remembered earlier.

10. Create `TAG_APCV1CBOR_AUTH_ASSERTION` structure.
   1. Copy the serialized `dataThatWasConfirmed` CBOR object into field `tbsData`.
   2. Copy `AAID` and `KeyID` into the respective TLV fields.
   3. Copy `signature` into the `TAG_SIGNATURE` field.

11. Create the `AuthenticateOut` object
    1. Set `AuthenticateOut.assertionScheme` to "APCV1CBOR"
    2. Encode the content of `TAG_APCV1CBOR_AUTH_ASSERTION` in base64url format and set as `AuthenticateOut.assertion`
    3. Return the `AuthenticateOut` object

The authenticator metadata statement **MUST** truly indicate the type of transaction confirmation display implementation. Typically the "Transaction Confirmation Display" flag will be set to `TRANSACTION_CONFIRMATION_DISPLAY_ANY` (bitwise) or `TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE`.

### 4.4 Authentication Response Processing Rules for FIDO Server

Instead of skipping the assertion according to step 6.6. in section 3.5.7.5 [UAFProtocol], follow these rules:

**NOTE**

The extraData in `tbsData.dataThatWasConfirmed` is the finalChallengeHash as computed by the ASM. The promptText in `tbsData.dataThatWasConfirmed` is the `AuthenticateIn.Transaction.content` value. `AuthenticateIn.Transaction.contentType` is "text/plain".

1. if `a.assertionScheme == "APCV1CBOR"` AND `a.assertion` starts with a valid CBOR structure as defined in section 3.2 Authentication Assertion, then
   1. set `tbsData` to the CBOR object contained in `a.assertion.tbsData`.
   2. Verify the `AAID` against the `AAID` stored in the user's record at time of Registration.
      - If comparison fails – continue with next assertion
   3. Locate `UAuth.pub` associated with `(a.assertion.AAID, a.assertion.KeyID)` in the user's record.
If such record doesn't exist - continue with next assertion

4. Locate authenticator specific authentication algorithms from authenticator metadata (field AuthenticationAlgs).

5. If fcp is of type FinalChallengeParams, then hash AuthenticationResponse.FinalChallengeParams using the hashing algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator Metadata, field AuthenticationAlgs. It is the hash algorithm associated with the first entry related to a constant with prefix ALG_SIGN.
   - FCHash = hash(AuthenticationResponse.FinalChallengeParams)

6. If fcp is of type ClientData, then hash AuthenticationResponse.fcParams using hashing algorithm specified in fcp.hashAlg.
   - FCHash = hash(AuthenticationResponse.fcParams)

7. Make sure that tbsData.dataThatWasConfirmed.extraData == FCHash
   - If comparison fails – continue with next assertion

8. Make sure there is a transaction cached on Relying Party side in the list cachedTransactions.
   - If not – continue with next assertion

9. Make sure that tbsData.dataThatWasConfirmed.promptText is included in the list cachedTransactions
   - If it’s not in the list – continue with next assertion

10. Use the UAuth.pub key found in step 1.2 and the appropriate authentication algorithm to verify the signature a.assertion.Signature of the to-be-signed object tbsData.
    1. If signature verification fails – continue with next assertion

5. Example for FIDO Metadata Statement

This section is non-normative.

This example Authenticator has the following characteristics:

- Authenticator implementing transaction confirmation display using TrustedUI (i.e. in TEE)
- Leveraging TEE backed key store and user verification
- Only fingerprint based user verification is implemented - no alternative password

EXAMPLE 1: MetadataStatement for UAF Authenticator

```json
{
   "description": "FIDO Alliance Sample UAF Authenticator supporting Android Protected Confirmation",
```
A. References

A.1 Normative references

[FIDO Glossary]

[RFC2119]

[UAFASM]

[UAFAuthnrCommands]

[UAFProtocol]

[UAFRegistry]

A.2 Informative references

[FIDO Metadata Service]
FIDO UAF APDU

FIDO Alliance Proposed Standard 20 October 2020

This version:
https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-apdu-v1.2-ps-20201020.html

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

Editor:
Naama Bak, Morpho

Contributors:
Virginie Galindo, Gemalto
Rolf Lindemann, Nok Nok Labs, Inc.
Ullrich Martini, Giesecke & Devrient
Chris Edwards, Intercede
Jeff Hodges, Paypal

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

This specification defines a mapping of FIDO UAF Authenticator commands to Application Protocol Data Units (APDUs) thus facilitating UAF authenticators based on Secure Elements.

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

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

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

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

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

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

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

All TLV structures defined in this document MUST be encoded in little-endian format.

All APDU defined in this document MUST be encoded as defined in [ISOIEC-7816-4-2013].

1.1 Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Introduction

This section is non-normative.

This specification defines the interface between the FIDO UAF Authenticator Specific Module (ASM) [UAFASM] and authenticators based upon "Secure Element" technology. The applicable secure element form factors are UICC (SIM card), embedded Secure Element (eSE), µSD, NFC card, and USB token. Their common characteristic is they communicate using Application Programming Data Units (APDU) in compliance with [ISOIEC-7816-4-2013].
Implementation of this specification is optional in the UAF framework, however, products claiming to implement the transport of UAF messages over APDUs should implement it.

This specification first describes the various fashions in which Secure Elements can be incorporated into UAF authenticator implementations — known as SE-based authenticators or just SE authenticators — and which components are responsible for handling user verification as well as cryptographic operations. The specification then describes the overall architecture of an SE-based authenticator stack from the ASM down to the secure element, the role of the "UAF Applet" running in the secure element, and outlines the nominal communication flow between the ASM and the SE. It then defines the mapping of UAF Authenticator commands to APDUs, as well as the FIDO-specific variants of the VERIFY APDU command.

NOTE
This specification does not define how an SE-based authenticator stack may be implemented, e.g., its integration with TEE or biometric sensors. However, SE-based authenticator vendors should reflect such implementation characteristics in the authenticator metadata such that FIDO Relying Parties wishing to be informed of said characteristics may have access to it.

3. SE-based Authenticator Implementation Use Cases

This section is non-normative.

Secure elements can be leveraged in different scenarios in the UAF technology. It can support user gestures (used to unlock access to FIDO credentials) or it can be involved in the actual cryptographic operations related to FIDO authentication. In this specification, we will be considering the following SE-based authenticator implementation use cases:

1. The Secure Element (SE) is the (silent) Authenticator.
2. The SE is part of the Authenticator which is composed of a Trusted Application (TEE) based User Verification component, potentially a TEE based transaction confirmation display and the crypto kernel inside the SE (Hybrid SE Authenticator).
3. The authenticator (Hybrid SE Authenticator) consists of
   - the SE implementing the matcher and the crypto kernel
   - and a specific software module (e.g. running on the FIDO User Device) to capture the user verification data (e.g. PIN, Face, Fingerprint).

3.1 Hybrid SE Authenticator

In FIDO UAF, the access to credentials for performing the actual authentication can be protected by a user verification step. This user verification step can be based on a PIN, a biometric or other methods. The authenticator functionality might be implemented in different components, including combinations such as TEE and SE, or fingerprint sensor and SE. In that case the SE implements only a part of the authenticator functionality.

NOTE
The reason for using such hybrid configuration is that Secure Elements do not have any user interface and hence cannot directly distinguish physical user interaction from programmatic communication (e.g. by malware). The ability to require a physical user interaction that cannot be emulated by malware is essential for protecting against scalable attacks (see [FIDOSecRef]). On the other hand, TEEs (or biometric sensors implemented in separate hardware) which can provide a trusted user interface typically do not offer the same level of key protection as Secure Elements.

Strictly spoken, a Hybrid SE Authenticator (voluntarily) uses the Authenticator Command interface [UAFAuthnrCommands] inside the authenticator, e.g. between the crypto kernel and the user verification component.

Examples of Hybrid SE authenticators are:

1. User PIN code capture and verification are implemented entirely in a TEE relying on Trusted User Interface and secure storage capabilities of the TEE and, once the PIN code is verified, the FIDO UAF crypto operations are performed in the SE.
2. User fingerprint is captured via a fingerprint sensor, the fingerprint match is performed in the TEE, relying on matching algorithms. Once the fingerprint has been positively checked, the cryptographic operations are executed in the Secure Element.
3. The user verification is implemented as match-on-chip in separate hardware and FIDO UAF cryptographic operations are implemented in the SE.

In all those cases, the hybrid nature of the authenticator will be managed by the software-based host, regardless of its nature (TEE, SW, Biometric sensor...). There are a number of possible interactions between the ASM and the SE actually implementing the verification and the cryptographic operations to consider within those use cases.
1. PIN user verification where the user interaction for the PIN entry is performed externally to the SE. The PIN may then be passed within a VERIFY command to the SE, followed by the actual cryptographic operations (such as the Register and Sign UAF authenticator commands).

2. Biometric user verification where the sample capture and matching is performed externally to the SE (e.g. in TEE or in a match-on-chip FP sensor). This would then only need to send to the SE the actual cryptographic operation needed in this session (such as the Register and Sign UAF authenticator commands).

3. User verification sample (Faceprint, Fingerprint..) capture is performed externally to the SE. The sample is then sent to a match-on-card applet in the SE that behaves as a global PIN to enable access to the cryptographic operation required within this session.

### 3.1.1 Architecture of the Hybrid SE Authenticator

In order to support an Hybrid SE Authenticator, a dedicated software-based host **MUST** be created which knows how the SE applet works. The communication between the SE applet and the host is defined based on [ISOIEC-7816-4-2013]. Whether a PC or mobile device the architecture is still the same, as defined below:

- **Application Layer**: This component is responsible for acquiring the user verification sample and mapping UAF commands to APDU commands.
- **Communication layer**: This is the [ISOIEC-7816-4-2013] APDUs interface, which provides methods to list and select readers, connect to a Secure Element and interact with it.
- **SE Access OS APIs**: OMA, PC/SC, NFC API, CCID...
- **Secure Element**: UICC, micro SD, eSE, Dual Interface card...

![Fig. 1 Architecture of Hybrid SE Authenticator](image)

APDU command-response pair are handled as indicated in [ISOIEC-7816-4-2013].

### 3.1.2 Communication flow between the ASM and the Hybrid SE Authenticator

The host is the entity communicating with the SE and which knows how the SE and the applet running in the SE can be accessed. The host could be a Trusted Application (TA) which runs inside a TEE or simply an application which runs in the normal world.

The following diagram illustrates how the Host of the Hybrid SE Authenticator **MAY** map the UAF commands to APDU commands. In this diagram, the User Verification Module is considered inside the SE applet.
4. FIDO UAF Applet and APDU commands

This section is normative.

4.1 UAF Applet in the Authenticator

4.1.1 Application Identifier

The FIDO UAF AID is defined in [UAFRegistry].

4.1.2 User Verification

The User verification is based on the submission of a PIN/password (i.e., knowledge based) or a biometric template (i.e., biometric based). In this document, the envisaged user verification methods are PIN and biometric based.

NOTE
If the User Verification Module is inside the Host, for example in the context of the TEE, the UserVerificationToken shall be generated in the Host and not in the SE. As a result step 6 (Figure 2) should be executed in the Host instead of the SE.
4.1.3 Cryptographic operations

The SE applet must be able to perform a set of cryptographic operations, such as key generation and signature computation. The cryptographic operations are defined in [UAFAuthnrCommands]. The SE applet must be able also to create data structures that can be parsed by FIDO Server. The SE applet SHALL use the cryptographic algorithms indicated in [UAFRegistry].

4.2 APDU Commands for FIDO UAF

4.2.1 Class byte coding

CLA indicates the class of the command.

<table>
<thead>
<tr>
<th>Commands</th>
<th>CLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT, VERIFY (ISO Version), GET RESPONSE (ISO Version)</td>
<td>0x00</td>
</tr>
<tr>
<td>VERIFY, UAF, GET RESPONSE</td>
<td>0x80</td>
</tr>
</tbody>
</table>

Table 1: Class byte coding

NOTE

If the payload of an APDU command is longer than 255 bytes, command chaining as described in [ISOIEC-7816-4-2013] should be used, even though CLA is proprietary.

4.2.2 APDU command "UAF"

4.2.2.1 Mapping between FIDO UAF authenticator commands and APDU commands

This section describes the mapping between FIDO UAF authenticator commands and APDU commands.

The mapping consists of encapsulating the entire UAF Authenticator Command in the payload of the APDU command, and the UAF Authenticator Command response in the payload of the APDU Response.

The host SHALL set the INS byte to "0x36" for all UAF commands The SE SHALL read the UAF command number and data from the payload in the data part of the command.

The payload of the APDU command is encoded according to [UAFAuthnrCommands], the first 2 bytes of each command are the UAF command number. Upon command reception, the SE applet MUST parse the first TLV tag (2 bytes) and figure out which UAF command is being issued. The SE applet SHALL parse the rest of the FIDO Authenticator Command payload according to [UAFAuthnrCommands].

The mapping of UAF Authenticator Commands to APDU commands is defined in the following table:

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data In</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proprietary(See Table 1)</td>
<td>0x36</td>
<td>0x00</td>
<td>0x00</td>
<td>Variable</td>
<td>UAF Authenticator Command structure</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 2: UAF APDU command

The UAF Authenticator Command structures are defined in part 6.2 of [UAFAuthnrCommands].

NOTE

If the UserVerificationToken is supported, The ASM must set the TAG_USERVERIFY_TOKEN flag in the value of the UserVerificationToken, received previously contained in either a Register or Sign command. Please refer to the FIG 1 in Use-Case section.

4.2.2.2 Response message and status conditions of an "UAF" APDU command

The status word of an "UAF" APDU response is handled at the Host level; the host must interpret and map the status word based on the table below.

If the status word is equals to "9000", the host shall return back to the ASM the entire data field of the APDU response. If the status word is "61xx", the host shall issue GET RESPONSE (see below) until no more data is available, concatenate these response parts and then return the entire response. Otherwise, the host has to build an UAF TLV response with the mapped status codes TAG_STATUS_CODE, using the following table.
For example, if the status word returned by the Applet is “6A88”, the host shall put \texttt{UAF\_CMD\_STATUS\_USER\_NOT\_ENROLLED} in the status codes of the UAF TLV response.

<table>
<thead>
<tr>
<th>APDU STATUS CODE</th>
<th>FIDO UAF STATUS CODE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000</td>
<td>0x00</td>
<td>UAF_CMD_STATUS_OK</td>
<td>Success.</td>
</tr>
<tr>
<td>61xx</td>
<td>0x00</td>
<td>UAF_CMD_STATUS_OK</td>
<td>Success, xx bytes available for GET RESPONSE.</td>
</tr>
<tr>
<td>6982</td>
<td>0x02</td>
<td>UAF_CMD_STATUS_ACCESS_DENIED</td>
<td>Access to this operation is denied.</td>
</tr>
<tr>
<td>6A88</td>
<td>0x03</td>
<td>UAF_CMD_STATUS_USER_NOT_ENROLLED</td>
<td>User is not enrolled with the authenticator.</td>
</tr>
<tr>
<td>N/A</td>
<td>0x04</td>
<td>UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT</td>
<td>Transaction content cannot be rendered.</td>
</tr>
<tr>
<td>N/A</td>
<td>0x05</td>
<td>UAF_CMD_STATUS_USER_CANCELLED</td>
<td>User has cancelled the operation.</td>
</tr>
<tr>
<td>6400</td>
<td>0x06</td>
<td>UAF_CMD_STATUS_CMD_NOT_SUPPORTED</td>
<td>Command not supported.</td>
</tr>
<tr>
<td>6A81</td>
<td>0x07</td>
<td>UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED</td>
<td>Required attestation not supported.</td>
</tr>
<tr>
<td>6A80</td>
<td>0x08</td>
<td>UAF_CMD_STATUS_PARAMS_INVALID</td>
<td>The request was rejected due to an incorrect data field.</td>
</tr>
<tr>
<td>6983</td>
<td>0x09</td>
<td>UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY</td>
<td>The UAuth key which is relevant for this command disappeared from the authenticator and cannot be restored.</td>
</tr>
<tr>
<td>N/A</td>
<td>0x0a</td>
<td>UAF_CMD_STATUS_TIMEOUT</td>
<td>The operation in the authenticator took longer than expected.</td>
</tr>
<tr>
<td>N/A</td>
<td>0x0e</td>
<td>UAF_CMD_STATUS_USER_NOT_RESPONSIVE</td>
<td>The user took too long to follow an instruction.</td>
</tr>
<tr>
<td>6A84</td>
<td>0x0f</td>
<td>UAF_CMD_STATUS_INSUFFICIENT_RESOURCES</td>
<td>Insufficient resources in the authenticator to perform the requested task.</td>
</tr>
<tr>
<td>63C0</td>
<td>0x10</td>
<td>UAF_CMD_STATUS_USER_LOCKOUT</td>
<td>The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that.</td>
</tr>
<tr>
<td>All other codes</td>
<td>0x01</td>
<td>UAF_CMD_STATUS_ERR_UNKNOWN</td>
<td>An unknown error</td>
</tr>
</tbody>
</table>

Table 3: Mapping between APDU Status Codes and FIDO Status Codes [UAFAuthnrCommands]

The response message of an UAF APDU command is defined in the following table:

<table>
<thead>
<tr>
<th>Data field</th>
<th>SW1 - SW2</th>
</tr>
</thead>
<tbody>
<tr>
<td>not present</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“6982” – The request was rejected due to user verification being required.</td>
</tr>
<tr>
<td></td>
<td>“6A80” – The request was rejected due to an incorrect data field.</td>
</tr>
<tr>
<td></td>
<td>“6A81” – Required attestation not supported</td>
</tr>
<tr>
<td></td>
<td>“6A88” – The user is not enrolled with the SE</td>
</tr>
<tr>
<td></td>
<td>“6400” – Execution error, undefined UAF command</td>
</tr>
<tr>
<td></td>
<td>“6983” – Authentication data not usable, Auth key disappeared</td>
</tr>
</tbody>
</table>
4.2.3 APDU Command "SELECT"

A successful SELECT AID allows the host to know that the applet is active in the SE, and to open a logical channel with this end.

In Android smartphones apps are not allowed to use the basic channel to the SIM because this channel is reserved for the baseband processor and the GSM/UMTS/LTE activities. In this case the app must select the applet in a logical channel.

The host must send a SELECT APDU command to the SE applet before any others commands.

As a result, the command for selecting the applet using the FIDO UAF AID is:

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data In</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0xA4</td>
<td>0x04</td>
<td>0x0C</td>
<td>0x08</td>
<td>0xA000000647AF0001</td>
<td>No response data is requested if the SELECT command's &quot;Le&quot; field is absent. Otherwise, if the &quot;Le&quot; field is present, vendor-proprietary data is being requested.</td>
</tr>
</tbody>
</table>

4.2.4 APDU Command "VERIFY"

This command is used to request access rights using a PIN or Biometric sample. The SE applet shall verify the sample data given by the Host against the reference PIN or Biometric held in the SE.

Please refer to [ISOIEC-7816-4-2013] and [ISOIEC-19794] for Personal verification through biometric methods.

If the verification is successful and UserVerificationToken is supported by the SE applet, a token SHALL be generated and sent to the Host. Without having this token, the Host cannot invoke special UAF commands such as Register or Sign.

The support of UserVerificationToken can be checked by examining the contents of the GetInfo response in the AuthenticatorType TAG or the response of SELECT APDU command [UAFAuthnrCommands].

Refer to [FIDOGlossary] for more information about UserVerificationToken.

4.2.4.1 Command structure

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Lc</th>
<th>Data In</th>
<th>Le</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO or Proprietary: see [ISOIEC-7816-4-2013]</td>
<td>0x20 (for PIN) or 0x21 (for biometry)</td>
<td>0x00</td>
<td>0x00</td>
<td>Variable Verification data</td>
<td>None or expected Le for UserVerificationToken</td>
<td></td>
</tr>
</tbody>
</table>

4.2.4.2 Response message and status conditions

<table>
<thead>
<tr>
<th>Data Out</th>
<th>SW1 - SW2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absent (ISO-Variant) or UserVerificationToken (proprietary)</td>
<td>See [ISOIEC-7816-4-2013]</td>
</tr>
</tbody>
</table>

Table 7: Response message and status conditions

NOTE

An SE applet that does not support UserVerificationToken, may use the [ISOIEC-7816-4-2013] VERIFY command. In this case, the VERIFY command must be securely bound to Register and Sign commands, so a secure bound method shall be implemented in the SE applet, such as Secure Messaging.

4.3 Managing Long APDU Commands and Responses

If a Secure Element is able to send a complete response (e.g. extended length APDU, block chaining), GET RESPONSE APDU command SHALL be
used, as defined in ISO Variant section. Otherwise, the proprietary solution shall be used, as defined in section Proprietary Variant.

4.3.1 ISO Variant

The [ISOIEC-7816-4-2013] GET RESPONSE command is used in order to retrieve big data returned by APDU command "UAF".

4.3.2 Proprietary Variant

In order to avoid using Get Response APDU command which is not supported by all devices and terminals, a proprietary method is defined for managing the long data answers at application level.

When using the proprietary variant, the response to the UAF APDU command shall include the Tag "0x2813", that specifies the length of the response.

Response Data Out description

Tag
0x2813

Length
variable (2 bytes)

Value
Expected data length (2 bytes)

In the case where the data does not fit into a single Data Out message, the host shall repeat the "UAF" command with P2 = 1 value mentioning this is a repetition of the incoming APDU to get all the data. This process shall be repeated until the entire data are collected by the host.

Here is an example of an APDU Response which contains more than 255 bytes in the payload.
5. Security considerations

This section is non-normative.

Guaranteeing trust and security in a fragmented architecture such as the one levering on SE is a challenge that the Host has to address regardless of its nature (TEE or Software based), which results in different challenges from a security and architecture perspective. One could list the following ones:

- use of a trusted user interface to enter a PIN on the device,
- secure transmission of PIN or fingerprint minutiae,
• minutiae extraction format,
• integrity of data transmitted between a Host and a SE.

Hence, we will only consider here, security challenges affecting the interface between the Host and the SE.

A possible way to maintain the integrity and confidentiality when APDUs commands are exchanged is to enable a secure channel between the Host and the SE. While this is left to implementation, there are several technologies allowing to build a secure channel between a SE and a device, that may be implemented.

• Secure channel between a trusted application in a TEE and an applet in a SE [GlobalPlatform-TEE-SE].
• Secure channel between a device and an applet in a secure element [GlobalPlatform-Card].
• Secure channel between a device and a SE [ETSI-Secure-Channel].

A. References

A.1 Normative references

[RFC4648]

A.2 Informative references

[ETSI-Secure-Channel]
• ETSI TS 102 484 Smart Cards: Secure channel between a UICC and an end-point terminal. URL:

[FIDO glossary]

[FIDO SecRef]

[GlobalPlatform-Card]
• Secure Channel Protocol 03 – GlobalPlatform Card Specification v.2.2 – Amendment D. URL:

[GlobalPlatform-TEE-SE]
• TEE Secure Element API Specification v1.0 | GPD_SPE_024. URL:

[ISOIEC-19794]
• ISO 19794: Information technology - Biometric data interchange formats. URL:

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

[RFC2119]

[UAFASM]
• FIDO UAF Authenticator-Specific Module API. Review Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-asm-api-v1.2-ps-20201020.html

[UAF Authnr Cmds]

[UAF Registry]
FIDO UAF Authenticator-Specific Module API

FIDO Alliance Proposed Standard 20 October 2020

This version:  

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

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

Contributors:  
Davit Baghdasaryan, Nok Nok Labs, Inc.  
Brad Hill, PayPal, Inc.  
Roni Sasson, Discretix, 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 be connected to a user device via various physical interfaces (SPI, USB, Bluetooth, etc). The UAF Authenticator-Specific Module (ASM) is a software interface on top of UAF authenticators which gives a standardized way for FIDO UAF Clients to detect and access the functionality of UAF authenticators and hides internal communication complexity from FIDO UAF Client.

This document describes the internal functionality of ASMs, defines the UAF ASM API and explains how FIDO UAF Clients should use the API.

This document's intended audience is FIDO authenticator and FIDO FIDO UAF Client vendors.

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 Key Words
- 2. Overview
  - 2.1 Code Example format
- 3. ASM Requests and Responses
  - 3.1 Request enum
  - 3.2 StatusCode Interface
    - 3.2.1 Constants
    - 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes
  - 3.3 ASMRequest Dictionary
    - 3.3.1 Dictionary ASMRequest Members
  - 3.4 ASMResponse Dictionary
    - 3.4.1 Dictionary ASMResponse Members
  - 3.5 GetInfo Request
    - 3.5.1 GetInfoOut Dictionary
      - 3.5.1.1 Dictionary GetInfoOut Members
    - 3.5.2 AuthenticatorInfo Dictionary
      - 3.5.2.1 Dictionary AuthenticatorInfo Members
  - 3.6 Register Request
    - 3.6.1 RegisterIn Object
      - 3.6.1.1 Dictionary RegisterIn Members
    - 3.6.2 RegisterOut Object
      - 3.6.2.1 Dictionary RegisterOut Members
    - 3.6.3 Detailed Description for Processing the Register Request
  - 3.7 Authenticate Request
    - 3.7.1 Authenticateln Object
3.7.1.1 Dictionary AuthenticateIn Members

- 3.7.2 Transaction Object
  - 3.7.2.1 Dictionary Transaction Members

- 3.7.3 AuthenticateOut Object
  - 3.7.3.1 Dictionary AuthenticateOut Members

- 3.7.4 Detailed Description for Processing the Authenticate Request

- 3.8 Deregister Request
  - 3.8.1 DeregisterIn Object
    - 3.8.1.1 Dictionary DeregisterIn Members
  - 3.8.2 Detailed Description for Processing the Deregister Request

- 3.9 GetRegistrations Request
  - 3.9.1 GetRegistrationsOut Object
    - 3.9.1.1 Dictionary GetRegistrationsOut Members
  - 3.9.2 AppRegistration Object
    - 3.9.2.1 Dictionary AppRegistration Members
  - 3.9.3 Detailed Description for Processing the GetRegistrations Request

- 3.10 OpenSettings Request

- 4. Using ASM API
- 5. ASM APIs for various platforms
  - 5.1 Android ASM Intent API
    - 5.1.1 Discovering ASMs
    - 5.1.2 Alternate Android AIDL Service ASM Implementation
  - 5.2 Java ASM API for Android
  - 5.3 C++ ASM API for iOS
  - 5.4 Windows ASM API
- 6. CTAP2 Interface
  - 6.1 authenticatorMakeCredential
    - 6.1.1 Processing rules for authenticatorMakeCredential
  - 6.2 authenticatorGetAssertion
    - 6.2.1 Processing rules for authenticatorGetAssertion
  - 6.3 authenticatorGetNextAssertion
  - 6.4 authenticatorCancel
  - 6.5 authenticatorReset
  - 6.6 authenticatorGetInfo
    - 6.6.1 Processing rules for authenticatorGetInfo
- 7. Security and Privacy Guidelines
  - 7.1 KHAccessToken
  - 7.2 Access Control for ASM APIs

- A. References
  - A.1 Normative references
  - A.2 Informative references

1. Notation
Type names, attribute names and element names are written as code.

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

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

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

The notation base64url refers to "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] without padding.

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

WebIDL dictionary members MUST NOT have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it MUST NOT be empty.

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

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

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

1.1 Key Words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

UAF authenticators may be connected to a user device via various physical interfaces (SPI, USB, Bluetooth, etc.). The UAF Authenticator-Specific module (ASM) is a software interface on top of UAF authenticators which gives a standardized way for FIDO UAF Clients to detect and access the functionality of UAF authenticators, and hides internal communication complexity from clients.

The ASM is a platform-specific software component offering an API to FIDO UAF Clients, enabling them to discover and communicate with one or more available authenticators.

A single ASM may report on behalf of multiple authenticators.

The intended audience for this document is FIDO UAF authenticator and FIDO UAF Client vendors.

NOTE

Platform vendors might choose to not expose the ASM API defined in this document to applications. They
The FIDO UAF protocol and its various operations is described in the FIDO UAF Protocol Specification [UAFProtocol]. The following simplified architecture diagram illustrates the interactions and actors this document is concerned with:

```
Fig. 1 UAF ASM API Architecture
```

2.1 Code Example format

ASM requests and responses are presented in WebIDL format.

3. ASM Requests and Responses

*This section is normative.*

The ASM API is defined in terms of JSON-formatted [ECMA-404] request and reply messages. In order to send a request to an ASM, a FIDO UAF Client creates an appropriate object (e.g., in ECMAscript), "stringifies" it (also known as serialization) into a JSON-formatted string, and sends it to the ASM. The ASM de-serializes the JSON-formatted string, processes the request, constructs a response, stringifies it, returning it as a JSON-formatted string.
Authenticator implementers MAY create custom authenticator command interfaces other than the one defined in [UAFAuthnrCommands]. Such implementations are not required to implement the exact message-specific processing steps described in this section. However,

1. the command interfaces MUST present the ASM with external behavior equivalent to that described below in order for the ASM to properly respond to the client request messages (e.g. returning appropriate UAF status codes for specific conditions).
2. all authenticator implementations MUST support an assertion scheme as defined [UAFRegistry] and MUST return the related objects, i.e. TAG_UAFV1_REG_ASSERTION and TAG_UAFV1_AUTH_ASSERTION as defined in [UAFAuthnrCommands].

3.1 Request enum

```
enum Request {
    "GetInfo",
    "Register",
    "Authenticate",
    "Deregister",
    "GetRegistrations",
    "OpenSettings"
};
```

<table>
<thead>
<tr>
<th>Enumeration description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetInfo</td>
</tr>
<tr>
<td>Register</td>
</tr>
<tr>
<td>Authenticate</td>
</tr>
<tr>
<td>Deregister</td>
</tr>
<tr>
<td>GetRegistrations</td>
</tr>
<tr>
<td>OpenSettings</td>
</tr>
</tbody>
</table>

3.2 StatusCode Interface

If the ASM needs to return an error received from the authenticator, it SHALL map the status code received from the authenticator to the appropriate ASM status code as specified here.

If the ASM doesn't understand the authenticator's status code, it SHALL treat it as UAF_CMD_STATUS_ERR_UNKNOWN and map it to UAF_ASM_STATUS_ERROR if it cannot be handled otherwise.

If the caller of the ASM interface (i.e. the FIDO Client) doesn't understand a status code returned by the ASM, it SHALL treat it as UAF_ASM_STATUS_ERROR. This might occur when new error codes are introduced.
3.2.1 Constants

**UAF_ASM_STATUS_OK** of type **short**
No error condition encountered.

**UAF_ASM_STATUS_ERROR** of type **short**
An unknown error has been encountered during the processing.

**UAF_ASM_STATUS_ACCESS_DENIED** of type **short**
Access to this request is denied.

**UAF_ASM_STATUS_USER_CANCELLED** of type **short**
Indicates that user explicitly canceled the request.

**UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT** of type **short**
Transaction content cannot be rendered, e.g. format doesn't fit authenticator's need.

**UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY** of type **short**
Indicates that the UAuth key disappeared from the authenticator and cannot be restored.

**UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED** of type **short**
Indicates that the authenticator is no longer connected to the ASM.

**UAF_ASM_STATUS_USER_NOT_RESPONSIVE** of type **short**
The user took too long to follow an instruction, e.g. didn't swipe the finger within the accepted time.

**UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES** of type **short**
Insufficient resources in the authenticator to perform the requested task.

**UAF_ASM_STATUS_USER_LOCKOUT** of type **short**
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.

**UAF_ASM_STATUS_USER_NOT_ENROLLED** of type **short**
The operation failed because the user is not enrolled to the authenticator and the authenticator cannot automatically trigger user enrollment.

**UAF_ASM_STATUS_SYSTEM_INTERRUPTED** of type **short**
Indicates that the system interrupted the operation. Retry might make sense.
### 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes

Authenticators are returning a status code in their responses to the ASM. The ASM needs to act on those responses and also map the status code returned by the authenticator to an ASM status code.

The mapping of authenticator status codes to ASM status codes is specified here:

<table>
<thead>
<tr>
<th>Authenticator Status Code</th>
<th>ASM Status Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAF_CMD_STATUS_OK</td>
<td>UAF_ASM_STATUS_OK</td>
<td>Pass-through success status.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_ERR_UNKNOWN</td>
<td>UAF_ASM_STATUS_ERROR</td>
<td>Pass-through unspecific error status.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_ACCESS_DENIED</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
<td>Pass-through status code.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_USER_NOT_ENROLLED</td>
<td>UAF_ASM_STATUS_USER_NOT_ENROLLED (or UAF_ASM_STATUS_ACCESS_DENIED in some situations)</td>
<td>According to [UAFAuthnrCommands], this might occur at the Sign command or at the Register command if the authenticator cannot automatically trigger user enrollment. The mapping depends on the command as follows. In the case of &quot;Register&quot; command, the error is mapped to UAF_ASM_STATUS_USER_NOT_ENROLLED in order to tell the calling FIDO Client there is an authenticator present but the user enrollment needs to be triggered outside the authenticator. In the case of the &quot;Sign&quot; command, the Uauth key needs to be protected by one of the authenticator's user verification methods at all times. So if this error occurs it is considered an internal error and hence mapped to UAF_ASM_STATUS_ACCESS_DENIED.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT</td>
<td>UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT</td>
<td>Pass-through status code as it indicates a problem to be resolved by the entity providing the transaction text.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_USER_CANCELLED</td>
<td>UAF_ASM_STATUS_USER_CANCELLED</td>
<td>Map to UAF_ASM_STATUS_USER_CANCELLED.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_CMD_NOT_SUPPORTED</td>
<td>UAF_ASM_STATUS_OK or UAF_ASM_STATUS_ERROR</td>
<td>If the ASM is able to handle that command on behalf of the authenticator (e.g. removing the key handle in the case of Dereg command for a bound authenticator), the UAF_ASM_STATUS_OK must be returned. Map the status code to UAF_ASM_STATUS_ERROR otherwise.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED</td>
<td>UAF_ASM_STATUS_ERROR</td>
<td>Indicates an ASM issue as the ASM has obviously not requested one of the supported attestation types indicated in the authenticator's response to the GetInfo command.</td>
</tr>
<tr>
<td>UAF_CMD_STATUS_PARAMS_INVALID</td>
<td>UAF_ASM_STATUS_ERROR</td>
<td>Indicates an ASM issue as the ASM has obviously not provided the correct parameters to the authenticator when sending the</td>
</tr>
</tbody>
</table>
Pass-through status code. It indicates that the Uauth key disappeared permanently and the RP App might want to trigger re-registration of the authenticator.

Retry operation and map to UAF_ASM_STATUS_ERROR if the problem persists.

Pass-through status code. The RP App might want to retry the operation once the user pays attention to the application again.

Pass-through status code.

Pass-through status code.

Map any unknown error code to UAF_ASM_STATUS_ERROR. This might happen when an ASM communicates with an authenticator implementing a newer UAF specification than the ASM.

### 3.3 ASMRequest Dictionary

All ASM requests are represented as `ASMRequest` objects.

```webidl
dictionary ASMRequest {
    required Request requestType;
    Version asmVersion;
    unsigned short authenticatorIndex;
    object args;
    Extension[] exts;
};
```

### 3.3.1 Dictionary `ASMRequest` Members

**requestType** of type required `Request`

- Request type

**asmVersion** of type `Version`

- ASM message version to be used with this request. For the definition of the `Version` dictionary see [UAFProtocol]. The `asmVersion` **MUST** be 1.2 (i.e. major version is 1 and minor version is 2) for this version of the specification.

**authenticatorIndex** of type `unsigned short`

- Refer to the `GetInfo` request for more details. Field `authenticatorIndex` **MUST NOT** be set for `GetInfo` request.

**args** of type `object`

- Request-specific arguments. If set, this attribute **MAY** take one of the following types:

  - `RegisterIn`
- AuthenticateIn
- DeregisterIn

```js
exts of type array of Extension
List of UAF extensions. For the definition of the Extension dictionary see [UAFProtocol].
```

### 3.4 ASMResponse Dictionary

All ASM responses are represented as `ASMResponse` objects.

```js
WebIDL
dictionary ASMResponse {
  required short statusCode;
  object responseData;
  Extension[] exts;
};
```

#### 3.4.1 Dictionary `ASMResponse` Members

- **statusCode** of type `required short`  
  `MUST` contain one of the values defined in the `StatusCode` interface

- **responseData** of type `object`  
  Request-specific response data. This attribute `MUST` have one of the following types:

  - GetInfoOut
  - RegisterOut
  - AuthenticateOut
  - GetRegistrationOut

- **exts** of type array of `Extension`  
  List of UAF extensions. For the definition of the `Extension` dictionary see [UAFProtocol].

### 3.5 GetInfo Request

Return information about available authenticators.

1. Enumerate all of the authenticators this ASM supports
2. Collect information about all of them
3. Assign indices to them (`authenticatorIndex`)
4. Return the information to the caller

**NOTE**

Where possible, an `authenticatorIndex` should be a persistent identifier that uniquely identifies an authenticator over time, even if it is repeatedly disconnected and reconnected. This avoids possible confusion if the set of available authenticators changes between a `GetInfo` request and subsequent ASM requests, and allows a FIDO client to perform caching of information about removable authenticators for a better user experience.

**NOTE**
For a GetInfo request, the following `ASMRequest` member(s) **MUST** have the following value(s). The remaining `ASMRequest` members **SHOULD** be omitted:

- `ASMRequest.requestType` **MUST** be set to `GetInfo`.

For a GetInfo response, the following `ASMResponse` member(s) **MUST** have the following value(s). The remaining `ASMResponse` members **SHOULD** be omitted:

- `ASMResponse.statusCode` **MUST** have one of the following values
  - `UAF_ASM_STATUS_OK`
  - `UAF_ASM_STATUS_ERROR`
- `ASMResponse.responseData` **MUST** be an object of type `GetInfoOut`. In the case of an error the values of the fields might be empty (e.g. array with no members).

See section 3.2.2 *Mapping Authenticator Status Codes to ASM Status Codes* for details on the mapping of authenticator status codes to ASM status codes.

### 3.5.1 GetInfoOut Dictionary

```
WebIDL

dictionary GetInfoOut {
    required AuthenticatorInfo[] Authenticators;
};
```

### 3.5.1.1 Dictionary GetInfoOut Members

- **Authenticators** of type array of optional `AuthenticatorInfo` List of authenticators reported by the current ASM. **MAY** be empty an empty list.

### 3.5.2 AuthenticatorInfo Dictionary

```
WebIDL

dictionary AuthenticatorInfo {
    required unsigned short authenticatorIndex;
    required Version[] asmVersions;
    required boolean isUserEnrolled;
    required boolean hasSettings;
    required AAID aaid;
    required DOMString assertionScheme;
    required DOMString authenticationAlgorithm;
    required authenticationType[] attestationTypes;
    required unsigned long userVerification;
    required unsigned short keyProtection;
    required unsigned short matcherProtection;
    required unsigned long attachmentHint;
    required boolean isSecondFactorOnly;
    required boolean isRoamingAuthenticator;
    required DOMString[] supportedExtensionIDs;
    required DOMString tcDisplay;
    required DOMString tcDisplayContentType;
};
```
3.5.2.1 Dictionary AuthenticatorInfo Members

**authenticatorIndex** of type required unsigned short
Authenticator index. Unique, within the scope of all authenticators reported by the ASM, index referring to an authenticator. This index is used by the UAF Client to refer to the appropriate authenticator in further requests.

**asmVersions** of type array of required Version
A list of ASM Versions that this authenticator can be used with. For the definition of the Version dictionary see [UAFProtocol].

**isUserEnrolled** of type required boolean
Indicates whether a user is enrolled with this authenticator. Authenticators which don't have user verification technology MUST always return true. Bound authenticators which support different profiles per operating system (OS) user MUST report enrollment status for the current OS user.

**hasSettings** of type required boolean
A boolean value indicating whether the authenticator has its own settings. If so, then a FIDO UAF Client can launch these settings by sending a OpenSettings request.

**aaid** of type required AAID
The "Authenticator Attestation ID" (AAID), which identifies the type and batch of the authenticator. See [UAFProtocol] for the definition of the AAID structure.

**assertionScheme** of type required DOMString
The assertion scheme the authenticator uses for attested data and signatures. AssertionScheme identifiers are defined in the UAF Protocol specification [UAFProtocol].

**authenticationAlgorithm** of type required unsigned short
Indicates the authentication algorithm that the authenticator uses. Authentication algorithm identifiers are defined in are defined in [FIDORegistry] with ALG_ prefix.

**attestationTypes** of type array of required unsigned short
Indicates attestation types supported by the authenticator. Attestation type TAGs are defined in [UAFRegistry] with TAG_ATTESTATION prefix.

**userVerification** of type required unsigned long
A set of bit flags indicating the user verification method(s) supported by the authenticator. The algorithm for combining the flags is defined in [UAFProtocol], section 3.1.12.1. The values are defined by the USER_VERIFY constants in [FIDORegistry].

**keyProtection** of type required unsigned short
A set of bit flags indicating the key protections used by the authenticator. The values are defined by the KEY_PROTECTION constants in [FIDORegistry].

**matcherProtection** of type required unsigned short
A set of bit flags indicating the matcher protections used by the authenticator. The values are defined by the MATCHER_PROTECTION constants in [FIDORegistry].

**attachmentHint** of type required unsigned long
A set of bit flags indicating how the authenticator is currently connected to the system hosting the FIDO
isSecondFactorOnly of type required boolean
Indicates whether the authenticator can be used only as a second factor.

isRoamingAuthenticator of type required boolean
Indicates whether this is a roaming authenticator or not.

supportedExtensionIDs of type array of required DOMString
List of supported UAF extension IDs. MAY be an empty list.

tcDisplay of type required unsigned short
A set of bit flags indicating the availability and type of the authenticator's transaction confirmation display. The values are defined by the TRANSACTION_CONFIRMATIONDISPLAY constants in [FIDORegistry].

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

tcDisplayContentType of type DOMString
Supported transaction content type [FIDOMetadataStatement].

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

tcDisplayPNGCharacteristics of type array of DisplayPNGCharacteristicsDescriptor
Supported transaction Portable Network Graphic (PNG) type [FIDOMetadataStatement]. For the definition of the DisplayPNGCharacteristicsDescriptor structure see [FIDOMetadataStatement].

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

title of type DOMString
A human-readable short title for the authenticator. It should be localized for the current locale.

NOTE
If the ASM doesn't return a title, the FIDO UAF Client must provide a title to the calling App. See section "Authenticator interface" in [UAFAppAPIAndTransport].

description of type DOMString
Human-readable longer description of what the authenticator represents.

NOTE
This text should be localized for current locale.
3.6 Register Request

Verify the user and return an authenticator-generated UAF registration assertion.

For a Register request, the following `ASMRequest` member(s) **MUST** have the following value(s). The remaining `ASMRequest` members **SHOULD** be omitted:

- `ASMRequest.requestType` **MUST** be set to `Register`
- `ASMRequest.asmVersion` **MUST** be set to the desired version
- `ASMRequest.authenticatorIndex` **MUST** be set to the target authenticator index
- `ASMRequest.args` **MUST** be set to an object of type `RegisterIn`
- `ASMRequest.exts` **MAY** include some extensions to be processed by the ASM or the by Authenticator.

For a Register response, the following `ASMResponse` member(s) **MUST** have the following value(s). The remaining `ASMResponse` members **SHOULD** be omitted:

- `ASMResponse.statusCode` **MUST** have one of the following values:
  - `UAF_ASM_STATUS_OK`
  - `UAF_ASM_STATUS_ERROR`
  - `UAF_ASM_STATUS_ACCESS_DENIED`
  - `UAF_ASM_STATUS_USER_CANCELLED`
  - `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`
  - `UAF_ASM_STATUS_USER_NOT_RESPONSIVE`
  - `UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES`
  - `UAF_ASM_STATUS_USER_LOCKOUT`
  - `UAF_ASM_STATUS_USER_NOT_ENROLLED`
- `ASMResponse.responseData` **MUST** be an object of type `RegisterOut`. In the case of an error the values of the fields might be empty (e.g. empty strings).

### 3.6.1 RegisterIn Object

```webidl
dictionary RegisterIn {
```
3.6.1.1 Dictionary RegisterIn Members

- **appID** of type required DOMString
  The FIDO server Application Identity.

- **username** of type required DOMString
  Human-readable user account name

- **finalChallenge** of type required DOMString
  base64url-encoded challenge data [RFC4648]

- **attestationType** of type required unsigned short
  Single requested attestation type

3.6.2 RegisterOut Object

```webidl

dictionary RegisterOut {
    required DOMString assertion;
    required DOMString assertionScheme;
}
```

3.6.2.1 Dictionary RegisterOut Members

- **assertion** of type required DOMString
  FIDO UAF authenticator registration assertion, base64url-encoded

- **assertionScheme** of type required DOMString
  Assertion scheme.

  AssertionScheme identifiers are defined in the UAF Protocol specification [UAFProtocol].

3.6.3 Detailed Description for Processing the Register Request

Refer to [UAFAuthnrCommands] document for more information about the TAGs and structure mentioned in this paragraph.

1. Locate authenticator using `authenticatorIndex`. If the authenticator cannot be located, then fail with `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`.

2. If a user is already enrolled with this authenticator (such as biometric enrollment, PIN setup, etc. for example) then the ASM MUST request that the authenticator verifies the user.

**NOTE**

If the authenticator supports `UserVerificationToken` (see [UAFAuthnrCommands]), then the ASM must obtain this token in order to later include it with the `Register` command.
If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return `UAF_ASM_STATUS_USER_LOCKOUT`.

- If verification fails, return `UAF_ASM_STATUS_ACCESS_DENIED`

3. If the user is not enrolled with the authenticator then take the user through the enrollment process.
   - If neither the ASM nor the Authenticator can trigger the enrollment process, return `UAF_ASM_STATUS_USER_NOT_ENROLLED`.
   - If enrollment fails, return `UAF_ASM_STATUS_ACCESS_DENIED`

4. Verify whether registerIn.appID and the appID included in the finalChallenge parameter are identical. The registerIn.finalChallenge value needs to be (1) base64url decoded and (2) parsed into a JSON object first.
   - If verification fails, return `UAF_ASM_STATUS_ACCESS_DENIED`

5. Construct `KHAccessToken` (see section `KHAccessToken` for more details)

6. Hash the provided `RegisterIn.finalChallenge` using the authenticator-specific hash function (`FinalChallengeHash`)

   An authenticator’s preferred hash function information **MUST** meet the algorithm defined in the `AuthenticatorInfo.authenticationAlgorithm` field.

7. Create a `TAG_UAFV1_REGISTER_CMD` structure and pass it to the authenticator
   1. Copy `FinalChallengeHash`, `KHAccessToken`, `RegisterIn.Username`, `UserVerificationToken`, `RegisterIn.AppID`, `RegisterIn.AttestationType`
      1. Depending on `AuthenticatorType` some arguments may be optional. Refer to `[UAFAuthnrCommands]` for more information on authenticator types and their required arguments.
      2. Add the extensions from the `ASMRequest.exts` dictionary appropriately to the `TAG_UAFV1_REGISTER_CMD` as `TAG_EXTENSION` object.

8. Invoke the command and receive the response. If the authenticator returns an error, handle that error appropriately. If the connection to the authenticator gets lost and cannot be restored, return `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`. If the operation finally fails, map the authenticator error code to the the appropriate ASM error code (see section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes for details).

9. Parse `TAG_UAFV1_REGISTER_CMD_RESP`
   1. Parse the content of `TAG_AUTHENTICATOR_ASSERTION` (e.g. `TAG_UAFV1_REG_ASSERTION`) and extract `TAG_KEYID`

10. If the authenticator is a bound authenticator

[NOTE]

What data an ASM will store at this stage depends on underlying authenticator’s architecture. For example some authenticators might store AppID, KeyHandle, KeyID inside their own secure storage. In this case ASM doesn’t have to store these data in its database.

11. Create a `RegisterOut` object
    1. Set `RegisterOut.assertionScheme` according to `AuthenticatorInfo.assertionScheme`
    2. Encode the content of `TAG_AUTHENTICATOR_ASSERTION` (e.g. `TAG_UAFV1_REG_ASSERTION`) in base64url format and set as `RegisterOut.assertion`.
    3. Return `RegisterOut` object

3.7 Authenticate Request
Verify the user and return authenticator-generated UAF authentication assertion.

For an Authenticate request, the following **ASMRequest** member(s) **MUST** have the following value(s). The remaining **ASMRequest** members **SHOULD** be omitted:

- **ASMRequest.requestType** **MUST** be set to **Authenticate**.
- **ASMRequest.asmVersion** **MUST** be set to the desired version.
- **ASMRequest.authenticatorIndex** **MUST** be set to the target authenticator index.
- **ASMRequest.args** **MUST** be set to an object of type **AuthenticateIn**
- **ASMRequest.exts** **MAY** include some extensions to be processed by the ASM or the by Authenticator.

For an Authenticate response, the following **ASMResponse** member(s) **MUST** have the following value(s). The remaining **ASMResponse** members **SHOULD** be omitted:

- **ASMResponse.statusCode** **MUST** have one of the following values:
  - UAF_ASM_STATUS_OK
  - UAF_ASM_STATUS_ERROR
  - UAF_ASM_STATUS_ACCESS_DENIED
  - UAF_ASM_STATUS_USER_CANCELLED
  - UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT
  - UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY
  - UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED
  - UAF_ASM_STATUS_USER_NOT_RESPONSIVE
  - UAF_ASM_STATUS_USER_LOCKOUT
  - UAF_ASM_STATUS_USER_NOT_ENROLLED
- **ASMResponse.responseData** **MUST** be an object of type **AuthenticateOut**. In the case of an error the values of the fields might be empty (e.g. empty strings).

### 3.7.1 AuthenticateIn Object

**WebIDL**

```idldoc
dictionary AuthenticateIn {
  required DOMString appID;
  DOMString[] keyIDs;
  required DOMString finalChallenge;
  Transaction[] transaction;
}
```

### 3.7.1.1 Dictionary **AuthenticateIn** Members

- **appID** of type **required DOMString**
  appID string

- **keyIDs** of type array of **DOMString**
  base64url [RFC4648] encoded keyIDs

- **finalChallenge** of type **required DOMString**
  base64url [RFC4648] encoded final challenge

- **transaction** of type array of **Transaction**
  An array of transaction data to be confirmed by user. If multiple transactions are provided, then the ASM
MUST select the one that best matches the current display characteristics.

NOTE
This may, for example, depend on whether user's device is positioned horizontally or vertically at the moment of transaction.

3.7.2 Transaction Object

```webidl
dictionary Transaction {
    required DOMString contentType;
    required DOMString content;
    DisplayPNGCharacteristicsDescriptor tcDisplayPNGCharacteristics;
}
```

3.7.2.1 Dictionary Transaction Members

- **contentType** of type required DOMString
  Contains the MIME Content-Type supported by the authenticator according to its metadata statement (see [FIDOMetadataStatement])

- **content** of type required DOMString
  Contains the base64url-encoded [RFC4648] transaction content according to the `contentType` to be shown to the user.

- **tcDisplayPNGCharacteristics** of type DisplayPNGCharacteristicsDescriptor
  Transaction content PNG characteristics. For the definition of the `DisplayPNGCharacteristicsDescriptor` structure See [FIDOMetadataStatement].

3.7.3 AuthenticateOut Object

```webidl
dictionary AuthenticateOut {
    required DOMString assertion;
    required DOMString assertionScheme;
}
```

3.7.3.1 Dictionary AuthenticateOut Members

- **assertion** of type required DOMString
  Authenticator UAF authentication assertion.

- **assertionScheme** of type required DOMString
  Assertion scheme

3.7.4 Detailed Description for Processing the Authenticate Request

Refer to the [UAFAuthnrCommands] document for more information about the TAGs and structure mentioned in this paragraph.

1. Locate the authenticator using `authenticatorIndex`. If the authenticator cannot be located, then fail with
2. If no user is enrolled with this authenticator (such as biometric enrollment, PIN setup, etc.), return UAF_ASM_STATUS_ACCESS_DENIED

3. The ASM must request the authenticator to verify the user.
   - If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return UAF_ASM_STATUS_USER_LOCKOUT.
   - If verification fails, return UAF_ASM_STATUS_ACCESS_DENIED

NOTE

If the authenticator supports UserVerificationToken (see [UAFAuthnrCommands]), the ASM must obtain this token in order to later pass to Sign command.

4. Construct KHAccessToken (see section KHAccessToken for more details)

5. Hash the provided AuthenticateIn.finalChallenge using an authenticator-specific hash function (FinalChallengeHash).

   The authenticator's preferred hash function information must meet the algorithm defined in the AuthenticatorInfo.authenticationAlgorithm field.

6. If this is a Second Factor authenticator and AuthenticateIn.keyIDs is empty, then return UAF_ASM_STATUS_ACCESS_DENIED

7. If AuthenticateIn.keyIDs is not empty,
   1. If this is a bound authenticator, then look up ASM's database with AuthenticateIn.appId and AuthenticateIn.keyIDs and obtain the KeyHandles associated with it.
      - Return UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY if the related key disappeared permanently from the authenticator.
      - Return UAF_ASM_STATUS_ACCESS_DENIED if no entry has been found.
   2. If this is a roaming authenticator, then treat AuthenticateIn.keyIDs as KeyHandles

8. Create TAG_UAFV1_SIGN_CMD structure and pass it to the authenticator.
   1. Copy AuthenticateIn.AppID, AuthenticateIn.Transaction.content (if not empty), FinalChallengeHash, KHAccessToken, UserVerificationToken, KeyHandles
      - Depending on AuthenticatorType some arguments may be optional. Refer to [UAFAuthnrCommands] for more information on authenticator types and their required arguments.
      - If multiple transactions are provided, select the one that best matches the current display characteristics.

NOTE

This may, for example, depend on whether user's device is positioned horizontally or vertically at the moment of transaction.

   - Decode the base64url encoded AuthenticateIn.Transaction.content before passing it to the authenticator
   2. Add the extensions from the ASMRequest.exts dictionary appropriately to the TAG_UAFV1_REGISTER_CMD as TAG_EXTENSION object.

9. Invoke the command and receive the response. If the authenticator returns an error, handle that error appropriately. If the connection to the authenticator gets lost and cannot be restored, return
If the operation finally fails, map the authenticator error code to the appropriate ASM error code (see section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes for details).

10. Parse TAG_UAFV1_SIGN_CMD_RESP
   - If it's a first-factor authenticator and the response includes TAG_USERNAME_AND_KEYHANDLE, then
     1. Extract usernames from TAG_USERNAME_AND_KEYHANDLE fields
     2. If two or more equal usernames are found, then choose the one which has registered most recently

   **NOTE**

   After this step, a first-factor bound authenticator which stores KeyHandles inside the ASM's database may delete the redundant KeyHandles from the ASM's database. This avoids having unusable (old) private key in the authenticator which (surprisingly) might become active after deregistering the newly generated one.

   3. Show remaining distinct usernames and ask the user to choose a single username
   4. Set TAG_UAFV1_SIGN_CMD.KeyHandles to the single KeyHandle associated with the selected username.
   5. Go to step #8 and send a new TAG_UAFV1_SIGN_CMD command

11. Create the AuthenticateOut object
    1. Set AuthenticateOut.assertionScheme as AuthenticatorInfo.assertionScheme
    2. Encode the content of TAG_AUTHENTICATOR_ASSERTION (e.g. TAG_UAFV1_AUTH_ASSERTION) in base64url format and set as AuthenticateOut.assertion
    3. Return the AuthenticateOut object

   **NOTE**

   Some authenticators might support "Transaction Confirmation Display" functionality not inside the authenticator but within the boundaries of the ASM. Typically these are software based Transaction Confirmation Displays. When processing the Sign command with a given transaction such ASM should show transaction content in its own UI and after user confirms it -- pass the content to authenticator so that the authenticator includes it in the final assertion.

   See [FIDORegistry] for flags describing Transaction Confirmation Display type.

The authenticator metadata statement **MUST** truly indicate the type of transaction confirmation display implementation. Typically the "Transaction Confirmation Display" flag will be set to TRANSACTION_CONFIRMATION_DISPLAY_ANY (bitwise) or TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGEDSOFTWARE.

### 3.8 Deregister Request

Delete registered UAF record from the authenticator.

For a Deregister request, the following ASMRequest member(s) **MUST** have the following value(s). The remaining ASMRequest members **SHOULD** be omitted:

- **ASMRequest.requestType** **MUST** be set to Deregister
- **ASMRequest.asmVersion** **MUST** be set to the desired version
- **ASMRequest.authenticatorIndex** **MUST** be set to the target authenticator index
ASMRequest.args MUST be set to an object of type DeregisterIn

For a Deregister response, the following ASMResponse member(s) MUST have the following value(s). The remaining ASMResponse members SHOULD be omitted:

- ASMResponse.statusCode MUST have one of the following values:
  - UAF_ASM_STATUS_OK
  - UAF_ASM_STATUS_ERROR
  - UAF_ASM_STATUS_ACCESS_DENIED
  - UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED

### 3.8.1 DeregisterIn Object

```
WebIDL
dictionary DeregisterIn {
  required DOMString appID;
  required DOMString keyID;
};
```

#### 3.8.1.1 Dictionary DeregisterIn Members

- **appID** of type required DOMString
  
  FIDO Server Application Identity

- **keyID** of type required DOMString
  
  Base64url-encoded [RFC4648] key identifier of the authenticator to be de-registered. The keyID can be an empty string. In this case all keyIDs related to this appID MUST be deregistered.

### 3.8.2 Detailed Description for Processing the Deregister Request

Refer to [UAFAuthnrCommands] for more information about the TAGs and structures mentioned in this paragraph.

1. Locate the authenticator using authenticatorIndex
2. Construct KHAccessToken (see section KHAccessToken for more details).
3. If this is a bound authenticator, then
   - If the value of DeregisterIn.keyID is an empty string, then lookup all pairs of this appID and any keyID mapped to this authenticatorIndex and delete them. Go to step 4.
   - Otherwise, lookup the authenticator related data in the ASM database and delete the record associated with DeregisterIn.appID and DeregisterIn.keyID. Go to step 4.
4. Create the TAG_UAFV1_DEREGISTER_CMD structure, copy KHAccessToken and DeregisterIn.keyID and pass it to the authenticator.

**NOTE**

In the case of roaming authenticators, the keyID passed to the authenticator might be an empty string. The authenticator is supposed to deregister all keys related to this appID in this case.

5. Invoke the command and receive the response. If the authenticator returns an error, handle that error appropriately. If the connection to the authenticator gets lost and cannot be restored, return UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED. If the operation finally fails, map the authenticator error code to
the appropriate ASM error code (see section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes for details). Return proper ASMResponse.

3.9 GetRegistrations Request

Return all registrations made for the calling FIDO UAF Client.

For a GetRegistrations request, the following ASMRequest member(s) MUST have the following value(s). The remaining ASMRequest members SHOULD be omitted:

- ASMRequest.requestType MUST be set to GetRegistrations
- ASMRequest.asmVersion MUST be set to the desired version
- ASMRequest.authenticatorIndex MUST be set to corresponding ID

For a GetRegistrations response, the following ASMResponse member(s) MUST have the following value(s). The remaining ASMResponse members SHOULD be omitted:

- ASMResponse.statusCode MUST have one of the following values:
  - UAF_ASM_STATUS_OK
  - UAF_ASM_STATUS_ERROR
  - UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED
- The ASMResponse.responseData MUST be an object of type GetRegistrationsOut. In the case of an error the values of the fields might be empty (e.g. empty strings).

3.9.1 GetRegistrationsOut Object

```webidl
dictionary GetRegistrationsOut {
  required AppRegistration[] appRegs;
};
```

3.9.1.1 Dictionary GetRegistrationsOut Members

- `appRegs` of type array of required AppRegistration
  - List of registrations associated with an `appID` (see AppRegistration below). MAY be an empty list.

3.9.2 AppRegistration Object

```webidl
dictionary AppRegistration {
  required DOMString appID;
  required DOMString[] keyIDs;
};
```

3.9.2.1 Dictionary AppRegistration Members

- `appID` of type required DOMString
  - FIDO Server Application Identity.
- `keyIDs` of type array of required DOMString
  - List of key identifiers associated with the `appID`
3.9.3 Detailed Description for Processing the GetRegistrations Request

1. Locate the authenticator using authenticatorIndex
2. If this is bound authenticator, then
   - Lookup the registrations associated with CallerID and AppID in the ASM database and construct a list of AppRegistration objects
   
   **NOTE**
   Some ASMs might not store this information inside their own database. Instead it might have been stored inside the authenticator's secure storage area. In this case the ASM must send a proprietary command to obtain the necessary data.

3. If this is not a bound authenticator, then set the list to empty.
4. Create GetRegistrationsOut object and return

3.10 OpenSettings Request

Display the authenticator-specific settings interface. If the authenticator has its own built-in user interface, then the ASM **MUST** invoke TAG_UAFV1_OPEN_SETTINGS_CMD to display it.

For an OpenSettings request, the following ASMRequest member(s) **MUST** have the following value(s). The remaining ASMRequest members **SHOULD** be omitted:

- ASMRequest.requestType **MUST** be set to OpenSettings
- ASMRequest.asmVersion **MUST** be set to the desired version
- ASMRequest.authenticatorIndex **MUST** be set to the target authenticator index

For an OpenSettings response, the following ASMResponse member(s) **MUST** have the following value(s). The remaining ASMResponse members **SHOULD** be omitted:

- ASMResponse.statusCode **MUST** have one of the following values:
  - UAF_ASM_STATUS_OK

4. Using ASM API

*This section is non-normative.*

In a typical implementation, the FIDO UAF Client will call GetInfo during initialization and obtain information about the authenticators. Once the information is obtained it will typically be used during FIDO UAF message processing to find a match for given FIDO UAF policy. Once a match is found the FIDO UAF Client will send the appropriate request (Register/Authenticate/Deregister...) to this ASM.

The FIDO UAF Client may use the information obtained from a GetInfo response to display relevant information about an authenticator to the user.

5. ASM APIs for various platforms

*This section is normative.*
5.1 Android ASM Intent API

On Android systems FIDO UAF ASMs **may** be implemented as a separate APK-packaged application.

The FIDO UAF Client invokes ASM operations via Android Intents. All interactions between the FIDO UAF Client and an ASM on Android takes place through the following intent identifier:

```
org.fidoalliance.intent.FIDO_OPERATION
```

To carry messages described in this document, an intent **must** also have its `type` attribute set to `application/fido.uaf_asm+json`.

ASMs **must** register that intent in their manifest file and implement a handler for it.

FIDO UAF Clients **must** append an extra, `message`, containing a `String` representation of a `ASMRequest`, before invoking the intent.

FIDO UAF Clients **must** invoke ASMs by calling `startActivityForResult()`.

FIDO UAF Clients **should** assume that ASMs will display an interface to the user in order to handle this intent, e.g. prompting the user to complete the verification ceremony. However, the ASM **should not** display any user interface when processing a `GetInfo` request.

After processing is complete the ASM will return the response intent as an argument to `onActivityResult()`. The response intent will have an extra, `message`, containing a `String` representation of a `ASMResponse`.

5.1.1 Discovering ASMs

FIDO UAF Clients can discover the ASMs available on the system by using `PackageManager.queryIntentActivities(Intent intent, int flags)` with the FIDO Intent described above to see if any activities are available.

A typical FIDO UAF Client will enumerate all ASM applications using this function and will invoke the `GetInfo` operation for each one discovered.

5.1.2 Alternate Android AIDL Service ASM Implementation

The Android Intent API can also be implemented using Android AIDL services as an alternative transport mechanism to Android Intents. Please see Android Intent API section [UAFAppAPIAndTransport] for differences between the Android AIDL service and Android Intent implementation.

This API should be used if the ASM itself doesn't implement any user interface.

**NOTE**

The advantage of this AIDL Server based API is that it doesn't cause a focus lose on the caller App.

5.2 Java ASM API for Android

**NOTE**

The Java ASM API is useful for ASMs for KeyStore based authenticators. In this case the platform limits key use-access to the application generating the key. The ASM runs in the process scope of the RP App.
public interface IASM {
    enum Event {
        PLUGGED,  /** Indicates that the authenticator was Plugged to system */
        UNPLUGGED  /** Indicates that the authenticator was Unplugged from system */
    }
}

public interface IEnumeratorListener {
    /**
     * This function is called when an authenticator is plugged or
     * unplugged.
     * @param eventType event type (plugged/unplugged)
     * @param serialized AuthenticatorInfo JSON based GetInfoResponse object
     */
    void onNotify(Event eventType, String authenticatorInfo);
}

public interface IResponseReceiver {
    /**
     * This function is called when ASM's response is ready.
     * @param response serialized response JSON based event data
     * @param exchangeData for ASM if it needs some
     * data back right after calling the callback function.
     * onResponse will set the exchangeData to the data to
     * be returned to the ASM.
     */
    void onResponse(String response, StringBuilder exchangeData);
}

/**
 * Initializes the ASM. This is the first function to
 * be called.
 * @param ctx the Android Application context of the calling application (or null)
 * @param enumeratorListener caller provided Enumerator
 * @return ASM StatusCode value
 */
short init(Context ctx, IEnumeratorListener enumeratorListener);

/**
 * Process given JSON request and returns JSON response.
 * If the caller wants to execute a function defined in ASM JSON
 * schema then this is the function that must be called.
 * @param act the calling Android Activity or null
 * @param inData input JSON data
 * @param ProcessListener event listener for receiving events from ASM
 * @return ASM StatusCode value
 */
short process(Activity act, String inData, IResponseReceiver responseReceiver);

/**
 * Uninitializes (shut's down) the ASM.
 * @return ASM StatusCode value
 */
short uninit();

5.3 C++ ASM API for iOS

NOTE

The C++ ASM API is useful for ASMs for KeyChain based authenticators. In this case the platform limits key
use-access to the application generating the key. The ASM runs in the process scope of the RP App.

#include
namespace FIDO_UAF {

class IASM {
public:

typedef enum {
    PLUGGED,  /** Indicates that the authenticator was Plugged to system */
    UNPLUGGED  /** Indicates that the authenticator was Unplugged from system */
} Event;

class IEnumeratorListener {
virtual ~IEnumeratorListener() {}}

/**
 * This function is called when an authenticator is plugged or
 * unplugged.
 * @param eventType event type (plugged/unplugged)
 * @param serialized AuthenticatorInfo JSON based GetInfoResponse object
 */
virtual void onNotify(Event eventType, const std::string& authenticatorInfo) {};

class IResponseReceiver {
  virtual ~IResponseReceiver() {} /* This function is called when ASM’s response is ready. */
  virtual void onResponse(const std::string& response, std::string &exchangeData) {};

  /**
   * Initializes the ASM. This is the first function to be called.
   * @param unc the platform UINavigationController or one of the derived classes (e.g. UINavigationController) in order to allow smooth UI integration of the ASM.
   * @param EnumerationListener caller provided Enumerator
   * @return ASM StatusCode value
   */
  virtual short int init(UINavigationController unc, IEnumerator EnumerationListener)=0;

  /**
   * Process given JSON request and returns JSON response.
   * If the caller wants to execute a function defined in ASM JSON schema then this is the function that must be called.
   * @param unc the platform UINavigationController or one of the derived classes (e.g. UINavigationController) in order to allow smooth UI integration of the ASM
   * @param InData input JSON data
   * @param ProcessListener event listener for receiving events from ASM
   * @return ASM StatusCode value
   */
  virtual short int process(UINavigationController unc, const std::string& InData, ICallback ProcessListener)=0;

  /**
   * Uninitializes (shut’s down) the ASM.
   * @return ASM StatusCode value
   */
  virtual short int uninit()=0;
};

5.4 Windows ASM API

On Windows, an ASM is implemented in the form of a Dynamic Link Library (DLL). The following is an example header file defining a Windows ASM API:

```cpp
/*! @file asm.h */
#ifndef __ASMH
#define __ASMH
#ifdef _WIN32
#define ASM_API __declspec(dllexport)
#endif
#ifdef _WIN32
#pragma warning ( disable : 4251 )
#endif
#define ASM_FUNC extern "C" ASM_API
#define ASM_NULL 0

/*! rief Error codes returned by ASM Plugin API. 
* Authenticator specific error codes are returned in JSON form. 
* See JSON schemas for more details. */
enum asmResult_t {
  Success = 0, /**< Success */
  Failure /**< Generic failure */
};

/*! rief Generic structure containing JSON string in UTF-8 format. 
* This structure is used throughout functions to pass and receives
```
* JSON data.
*/

struct asmJSONData_t
{
    int length; /**< JSON data length */
    char *pData; /**< JSON data */
};

/*! \brief Enumeration event types for authenticators. 
These events will be fired when an authenticator becomes available (plugged) or unavailable (unplugged). */

enum asmEnumerationType_t
{
    Plugged = 0, /**< Indicates that authenticator Plugged to system */
    Unplugged /**< Indicates that authenticator Unplugged from system */
};

namespace ASM
{
/*! \brief Callback listener. 
FIDO UAF Client must pass an object implementing this interface to Authenticator::Process function. This interface is used to provide ASM JSON based response data. */

class ICallback
{ public
    virtual ~ICallback() {}
    /**
     * This function is called when ASM's response is ready.
     * @param response JSON based event data
     * @param exchangeData must be provided by ASM if it needs some data back right after calling the callback function.
     * The lifecycle of this parameter must be managed by ASM. ASM must allocate enough memory for getting the data back.
     */
    virtual void Callback(const asmJSONData_t &response, asmJSONData_t &exchangeData) = 0;
};

/*! \brief Authenticator Enumerator. 
FIDO UAF Client must provide an object implementing this interface. It will be invoked when a new authenticator is plugged or when an authenticator has been unplugged. */

class IEnumerator
{ public
    virtual ~IEnumerator() {}
    /**
     * This function is called when an authenticator is plugged or unplugged.
     * @param eventType event type (plugged/unplugged)
     * @param AuthenticatorInfo JSON based GetInfoResponse object
     */
    virtual void Notify(const asmEnumerationType_t eventType, const asmJSONData_t &AuthenticatorInfo) = 0;
};
}

/**
Initializes ASM plugin. This is the first function to be called.
* @param pEnumerationListener caller provided Enumerator
*/
ASM_FUNC asmResult_t asmInit(ASM::IEnumerator *pEnumerationListener);

/**
Process given JSON request and returns JSON response.
* If the caller wants to execute a function defined in ASM JSON schema then this is the function that must be called.
* @param pInData input JSON data
* @param pListener event listener for receiving events from ASM
*/
ASM_FUNC asmResult_t asmProcess(const asmJSONData_t *pInData, ASM::ICallback *pListener);

/**
Uninitializes ASM plugin.
*/
ASM_FUNC asmResult_t asmUninit();
A Windows-based FIDO UAF Client **must** look for ASM DLLs in the following registry paths:

- HKCU\Software\FIDO\UAF\ASM
- HKLM\Software\FIDO\UAF\ASM

The FIDO UAF Client iterates over all keys under this path and looks for "path" field:

```
[HKCU\Software\FIDO\UAF\ASM\exampleASMName]
"path"="<ABSOLUTE_PATH_TO_ASM>.dll"
```

**path** **must** point to the absolute location of the ASM DLL.

### 6. CTAP2 Interface

*This section is normative.*

ASMs can (optionally) provide a FIDO CTAP 2 interface in order to allow the authenticator being used as external authenticator from a FIDO2 or Web Authentication enabled platform supporting the CTAP 2 protocol [FIDOCTAP].

In this case the CTAP2 enabled ASM provides the CTAP2 interface upstream through one or more of the transport protocols defined in [FIDOCTAP] (e.g. USB, NFC, BLE). Note that the CTAP2 interface is the connection to the FIDO Client / FIDO enabled platform.

In the following section we specify how the ASM needs to map the parameters received via the FIDO CTAP2 interface to FIDO UAF Authenticator Commands [UAFAuthnrCommands].

#### 6.1 authenticatorMakeCredential

*This section is normative.*

**NOTE**

This interface has the following input parameters (see [FIDOCTAP]):

1. clientDataHash (required, byte array).
2. rp (required, PublicKeyCredentialEntity). Identity of the relying party.
3. user (required, PublicKeyCredentialUserEntity).
4. pubKeyCredParams (required, CBOR array).
5. excludeList (optional, sequence of PublicKeyCredentialDescriptors).
6. extensions (optional, CBOR map). Parameters to influence authenticator operation.
7. options (optional, sequence of authenticator options, i.e. "rk" and "uv"). Parameters to influence authenticator operation.
8. pinAuth (optional, byte array).
9. pinProtocol (optional, unsigned integer).

The output parameters are (see [FIDOCTAP]):

1. authData (required, sequence of bytes). The authenticator data object.
2. fmt (required, String). The attestation statement format identifier.
6.1.1 Processing rules for authenticatorMakeCredential

This section is normative.

1. invoke Register command for UAF authenticator as described in [UAFAuthnrCommands] section 6.2.4 using
   the following field mapping instructions:
   - authenticatorIndex set appropriately, e.g. 1.
     - If webauthn_appid is present, then
       1. Verify that the effective domain of AppID is identical to the effective domain of rp.id.
       2. Set AppID to the value of extension webauthn_appid (see [WebAuth]).
     - If webauthn_appid is not present, then set AppId to rp.id (see [WebAuth]).
   - FinalChallengeHash set to clientDataHash.
   - Username set to user.displayName (see [WebAuth]). This string will be displayed to the user in order to
     select a specific account if the user has multiple accounts at that relying party.
   - attestationType set to the attestation supported by that authenticator, e.g. ATTESTATION_BASIC_FULL or
     ATTESTATION_ECDAA.
   - KHAccessToken set to some persistent identifier used for this authenticator. If the authenticator is bound to
     the platform this ASM is running on, it needs to be a secret identifier only known to this ASM instance. If
     the authenticator is a "roaming authenticator", i.e. external to the platform this ASM is running on, the
     identifier can have value 0.
   - Add the fido.uaf.userid extension with value user.id to the Register command.
   - Use the pinAuth and pinProtocol parameters appropriately when communicating with the authenticator (if
     supported).

2. If this is a bound authenticator and the Authenticator doesn't support the fido.uaf.userid, let the ASM
   remember the user.id value related to the generated UAuth key pair.

3. If the command was successful, create the result object as follows
   - set authData to a freshly generated authenticator data object, containing the corresponding values taken
     from the assertion generated by the authenticator. That means:
     - set authData.rpID to the SHA256 hash of AppID.
     - initialize authData with 0 and then set set flag authData.AT to 1 and set authData.UP to 1 if the
       authenticator is not a silent authenticator. Set flag authData.uv to 1 if the authenticator is not a silent
       authenticator. The flags authData.UP and authData.UV need to be 0 if it is a silent authenticator. Set
       authData.ED to 1 if the authenticator added extensions to the assertion. In this case add the
       individual extensions to the CBOR map appropriately.
     - set authData.signCount to the uafAssertion.signCounter.
     - set authData.attestationData.AAGUID to the AAID of this authenticator. Setting the remaining bytes
       to 0.
     - set authData.attestationData.CredentialID to uafAssertion.keyHandle and set the length L of the
       Credential ID to the size of the keyHandle.
     - set authData.attestationData.pubKey to uafAssertion.publicKey with appropriate encoding
       conversion
     - set fmt to the "fido-uaf".
     - set attStmt to the AUTHENTICATOR_ASSERTION element of the TAG_UAFV1_REGISTER_CMD_RESPONSE returned by
       the authenticator.

4. Return authData, fmt and attStmt.
6.2 authenticatorGetAssertion

This section is normative.

NOTE

This interface has the following input parameters (see [FIDOCTAP]):

1. rpId (required, String). Identity of the relying party.
2. clientDataHash (required, byte array).
3. allowList (optional, sequence of PublicKeyCredentialDescriptors).
4. extensions (optional, CBOR map).
5. options (optional, sequence of authenticator options, i.e. "up" and "uv").

The output parameters are (see [FIDOCTAP]):

1. credential (optional, PublicKeyCredentialDescriptor).
2. authData (required, byte array).
3. signature (required, byte array).
4. user (required, PublicKeyCredentialUserEntity).
5. numberOfCredentials (optional, integer).

6.2.1 Processing rules for authenticatorGetAssertion

This section is normative.

1. invoke Sign command for UAF authenticator as described in [UAFAuthnrCommands] section 6.3.4 using the following field mapping instructions
   - authenticatorIndex set appropriately, e.g. 1.
   - If webauthn_appid is present, then
     1. Verify that the effective domain of AppID is identical to the effective domain of rpId.
     2. Set AppID to the value of extension webauthn_appid (see [WebAuthn]).
   - If webauthn_appid is not present, then set AppID to rpId (see [WebAuthn]).
   - FinalChallengeHash set to clientDataHash.
   - TransactionContent set to value of extension webauthn_txAuthGeneric or webauthn_txAuthsimple (see [WebAuthn]) depending on which extension is present and supported by this authenticator. If the authenticator doesn't natively support transactionConfirmation, the hash of the value included in either of the webauthn_tx* extensions can be computed by the ASM and passed to the authenticator in TransactionContentHash. See [UAFAuthnrCommands] section 6.3.1 for details.
   - KHAccessToken set to the persistent identifier used for this authenticator (at authenticatorMakeCredential).
   - If allowList is present then add the .id field of each element as KeyHandle element to the command.
   - Use the pinAuth and pinProtocol parameters appropriately when communicating with the authenticator (if supported).

2. If the command was successful (with potential ambiguities of RawKeyHandles resolved), create the result object as follows
   - set credential.id to the keyHandle returned by the authenticator command. Set credential.type to "public-key-uaf" and set credential.transports to the transport currently being used by this authenticator (e.g. "usb").
- set authData to the UAFV1_SIGNED_DATA element included in the AUTHENTICATOR_ASSERTION element.
- set signature to the SIGNATURE element included in the AUTHENTICATOR_ASSERTION element.
- If the authenticator returned the fido.uaf.userid extension, then set user.id to the value of the fido.uaf.userid extension as returned by the authenticator.
- If the authenticator did not return the fido.uaf.userid extension but the ASM remembered the user ID, then set user.id to the value of the user ID remembered by the ASM.

3. Return credential, authData, signature, user.

6.3 authenticatorGetNextAssertion

*This section is normative.*

Not supported. This interface will always return a single assertion.

6.4 authenticatorCancel

*This section is normative.*

Cancel the existing authenticator command if it is still pending.

6.5 authenticatorReset

*This section is normative.*

Reset the authenticator back to factory default state. In order to prevent accidental trigger of this mechanism, some form of user approval MAY be performed by the authenticator itself.

6.6 authenticatorGetInfo

*This section is normative.*

This interface has no input parameters.

**NOTE**

Output parameters are (see [FIDOCTAP]):

1. versions (required, sequence of strings). List of FIDO protocol versions supported by the authenticator.
2. extensions (optional, sequence of strings). List of extensions supported by the authenticator.
3. aaguid (optional, string). The AAGUID claimed by the authenticator.
4. options (optional, map). Map of "plat", "rk", "clientPin", "up", "uv"
5. maxMsgSize (optional, unsigned integer). The maximum message size accepted by the authenticator.
6. pinProtocols (optional, array of unsigned integers).

6.6.1 Processing rules for authenticatorGetInfo

*This section is normative.*

This interface is expected to report a single authenticator only.

1. Invoke the GetInfo command [UAFAuthnrCommands] for the connected authenticator.
2. If the command was successful, create the result object as follows
   - set `versions` to "FIDO_2_0" as this is the only version supported by CTAP2 at this time.
   - set `extensions` to the list of extensions returned by the authenticator (one entry per field `SupportedExtensionID`).
   - set `aaguid` to the AAID returned by the authenticator, setting all remaining bytes to 0.
   - set `options` appropriately.
   - set `maxMsgSize` to the maximum message size supported by the authenticator - if known
   - set `pinProtocols` to the list of supported pin protocols (if any).

3. Return `versions`, `extensions`, `aaguid`, `options`, `maxMsgSize` (if known) and `pinProtocols` (if any).

7. Security and Privacy Guidelines

   This section is normative.

   ASM developers must carefully protect the FIDO UAF data they are working with. ASMs must follow these security guidelines:

   - ASMs **MUST** implement a mechanism for isolating UAF credentials registered by two different FIDO UAF Clients from one another. One FIDO UAF Client **MUST NOT** have access to FIDO UAF credentials that have been registered via a different FIDO UAF Client. This prevents malware from exercising credentials associated with a legitimate FIDO Client.

   **NOTE**

   ASMs must properly protect their sensitive data against malware using platform-provided isolation capabilities in order to follow the assumptions made in [FIDOSecRef]. Malware with root access to the system or direct physical attack on the device are out of scope for this requirement.

   **NOTE**

   The following are examples for achieving this:
   - If an ASM is bundled with a FIDO UAF Client, this isolation mechanism is already built-in.
   - If the ASM and FIDO UAF Client are implemented by the same vendor, the vendor may implement proprietary mechanisms to bind its ASM exclusively to its own FIDO UAF Client.
   - On some platforms ASMs and the FIDO UAF Clients may be assigned with a special privilege or permissions which regular applications don't have. ASMs built for such platforms may avoid supporting isolation of UAF credentials per FIDO UAF Clients since all FIDO UAF Clients will be considered equally trusted.

   - An ASM designed specifically for bound authenticators **MUST** ensure that FIDO UAF credentials registered with one ASM cannot be accessed by another ASM. This is to prevent an application pretending to be an ASM from exercising legitimate UAF credentials.
     - Using a [KHAccessToken] offers such a mechanism.

   - An ASM **MUST** implement platform-provided security best practices for protecting UAF-related stored data.
ASMs **MUST NOT** store any sensitive FIDO UAF data in its local storage, except the following:

- **CallerID**, **ASMToken**, **PersonaID**, **KeyID**, **KeyHandle**, **AppID**

**NOTE**

An ASM, for example, must never store a username provided by a FIDO Server in its local storage in a form other than being decryptable exclusively by the authenticator.

ASMs **SHOULD** ensure that applications cannot use silent authenticators for tracking purposes. ASMs implementing support for a silent authenticator **MUST** show, during every registration, a user interface which explains what a silent authenticator is, asking for the users consent for the registration. Also, it is **RECOMMENDED** that ASMs designed to support roaming silent authenticators either

- Run with a special permission/privilege on the system, or
- Have a built-in binding with the authenticator which ensures that other applications cannot directly communicate with the authenticator by bypassing this ASM.

### 7.1 KHAccessToken

**KHAccessToken** is an access control mechanism for protecting an authenticator's FIDO UAF credentials from unauthorized use. It is created by the ASM by mixing various sources of information together. Typically, a **KHAccessToken** contains the following four data items in it: **AppID**, **PersonaID**, **ASMToken** and **CallerID**.

**AppID** is provided by the FIDO Server and is contained in every FIDO UAF message.

**PersonaID** is obtained by the ASM from the operational environment. Typically a different **PersonaID** is assigned to every operating system user account.

**ASMToken** is a randomly generated secret which is maintained and protected by the ASM.

**NOTE**

In a typical implementation an ASM will randomly generate an ASMToken when it is launched the first time and will maintain this secret until the ASM is uninstalled.

**CallerID** is the ID the platform has assigned to the calling FIDO UAF Client (e.g. "bundle ID" for iOS). On different platforms the **CallerID** can be obtained differently.

**NOTE**

For example on Android platform ASM can use the hash of the caller's apk-signing-cert.

The ASM uses the **KHAccessToken** to establish a link between the ASM and the key handle that is created by authenticator on behalf of this ASM.

The ASM provides the **KHAccessToken** to the authenticator with every command which works with key handles.
Bound authenticators **MUST** support a mechanism for binding generated key handles to ASMs. The binding mechanism **MUST** have at least the same security characteristics as mechanism for protecting KHAccessToken described above. As a consequence it is **RECOMMENDED** to securely derive KHAccessToken from AppID, ASMToken, PersonaID and the CallerID.

Alternative methods for binding key handles to ASMs can be used if their security level is equal or better.

From a security perspective, the KHAccessToken method relies on the OS/platform to:

1. allow the ASM keeping the ASMToken secret
2. and let the ASM determine the CalledID correctly
3. and let the FIDO Client verify the AppID/FacetID correctly

**NOTE**

It is recommended for roaming authenticators that the KHAccessToken contains only the AppID since otherwise users won’t be able to use them on different machines (PersonaID, ASMToken and CallerID are platform specific). If the authenticator vendor decides to do that in order to address a specific use case, however, it is allowed.

Including PersonaID in the KHAccessToken is optional for all types of authenticators. However an authenticator designed for multi-user systems will likely have to support it.

If an ASM for roaming authenticators doesn't use a KHAccessToken which is different for each AppID, the ASM **MUST**

---

**NOTE**

The following example describes how the ASM constructs and uses KHAccessToken.

- During a Register request
  - Set KHAccessToken to a secret value only known to the ASM. This value will always be the same for this ASM.
  - Append AppID
    - KHAccessToken = AppID
  - If a bound authenticator, append ASMToken, PersonaID and CallerID
    - KHAccessToken |= ASMToken | PersonaID | CallerID
  - Hash KHAccessToken
    - Hash KHAccessToken using the authenticator’s hashing algorithm. The reason of using authenticator specific hash function is to make sure of interoperability between ASMs. If interoperability is not required, an ASM can use any other secure hash function it wants.
    - KHAccessToken=hash(KHAccessToken)
  - Provide KHAccessToken to the authenticator
    - The authenticator puts the KHAccessToken into RawKeyHandle (see [UFAuthnrCommands] for more details)
- During other commands which require KHAccessToken as input argument
  - The ASM computes KHAccessToken the same way as during the Register request and provides it to the authenticator along with other arguments.
  - The authenticator unwraps the provided key handle(s) and proceeds with the command only if RawKeyHandle.KHAccessToken is equal to the provided KHAccessToken.
include the AppId in the command for a deregister request containing an empty KeyID.

7.2 Access Control for ASM APIs

The following table summarizes the access control requirements for each API call.

ASM vendors **must** implement the access control requirements defined below. ASM vendors **may** implement additional security mechanisms.

Terms used in the table:

- **NoAuth** -- no access control
- **CallerID** -- FIDO UAF Client's platform-assigned ID is verified
- **UserVerify** -- user must be explicitly verified
- **KeyIDList** -- must be known to the caller

<table>
<thead>
<tr>
<th>Commands</th>
<th>First-factor bound authenticator</th>
<th>Second-factor bound authenticator</th>
<th>First-factor roaming authenticator</th>
<th>Second-factor 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>Authenticate</td>
<td>UserVerify, AppId, CallerID, PersonalID</td>
<td>UserVerify, AppId, KeyIDList, CallerID, PersonalID</td>
<td>UserVerify, AppId</td>
<td>UserVerify, AppId, KeyIDList</td>
</tr>
<tr>
<td>GetRegistrations*</td>
<td>CallerID, PersonalID</td>
<td>CallerID, PersonalID</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Deregister</td>
<td>AppId, KeyID, CallerID, PersonalID</td>
<td>AppId, KeyID, CallerID, PersonalID</td>
<td>AppId, KeyID</td>
<td>AppId, KeyID</td>
</tr>
</tbody>
</table>

A. References

A.1 Normative references

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

[FIDOCTAP]

[FIDOglossary]

[FIDOMetadataStatement]
B. Hill; D. Baghdasaryan; J. Kemp. FIDO Metadata Statements. Review Draft. URL:
A.2 Informative references

[ECMA-404]

[FIDOSecRef]

[RFC2397]

[UAFAppAPIAndTransport]

[WebAuthn]
Dirk Balfanz; Alexei Czeskis; Jeff Hodges; J.C. Jones; Michael B. Jones; Akshay Kumar; Angelo Liao; Rolf Lindemann; Emil Lundberg. *Web Authentication: An API for accessing Public Key Credentials Level 1*. March 2019. TR. URL: https://www.w3.org/TR/webauthn/

[WebIDL]
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
6.4.3 Status Codes

6.4.4 Command Description

6.5 OpenSettings Command

6.5.1 Command Structure

6.5.2 Command Response

6.5.3 Status Codes

7. KeyIDs and key handles

7.1 first-factor Bound Authenticator

7.2 2ndF Bound Authenticator

7.3 first-factor Roaming Authenticator

7.4 2ndF Roaming Authenticator

8. Access Control for Commands

9. Considerations

9.1 Algorithms and Key Sizes

9.2 Indicating the Authenticator Model

10. Relationship to other standards

10.1 TEE

10.2 Secure Elements

10.3 TPM

10.4 Unreliable Transports

A. Security Guidelines

B. Table of Figures

C. References

C.1 Normative references

C.2 Informative references

1. Notation

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

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

In formulas we use "\|" to denote byte wise concatenation operations.

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

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

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

![Fig. 1 UAF Authenticator Commands](image)

### 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

**Fig. 2 FIDO Authenticator Logical Sub-Components**

Some examples of UAF Authenticators:

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

### 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:

- **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_UAFV1_GETINFO_CMD</td>
<td>0x3401</td>
<td>Tag for GetInfo command.</td>
</tr>
<tr>
<td>TAG_UAFV1_GETINFO_CMD_RESPONSE</td>
<td>0x3601</td>
<td>Tag for GetInfo command response.</td>
</tr>
<tr>
<td>TAG_UAFV1_REGISTER_CMD</td>
<td>0x3402</td>
<td>Tag for Register command.</td>
</tr>
<tr>
<td>TAG_UAFV1_REGISTER_CMD_RESPONSE</td>
<td>0x3602</td>
<td>Tag for Register command response.</td>
</tr>
<tr>
<td>TAG_UAFV1_SIGN_CMD</td>
<td>0x3403</td>
<td>Tag for Sign command.</td>
</tr>
<tr>
<td>TAG_UAFV1_SIGN_CMD_RESPONSE</td>
<td>0x3603</td>
<td>Tag for Sign command response.</td>
</tr>
<tr>
<td>TAG_UAFV1_DEREGISTER_CMD</td>
<td>0x3404</td>
<td>Tag for Deregister command.</td>
</tr>
<tr>
<td>TAG_UAFV1_DEREGISTER_CMD_RESPONSE</td>
<td>0x3604</td>
<td>Tag for Deregister command response.</td>
</tr>
<tr>
<td>TAG_UAFV1_OPEN_SETTINGS_CMD</td>
<td>0x3406</td>
<td>Tag for OpenSettings command.</td>
</tr>
<tr>
<td>TAG_UAFV1_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 [FIDO Glossary] 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 [UAF Registry]. (&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_TRANSACTION_CONFIRMATION_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 Name</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</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.
TAG_EXTENSION_ID 0x2E13 Represents extension ID. Content of this tag is a UINT8[] encoding of a UTF-8 string.

TAG_EXTENSION_DATA 0x2E14 Represents extension data. Content of this tag is a UINT8[] byte array.

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

<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>

Using tag 0x3E11 (as opposed to tag 0x3E12) has the same meaning as the flag fail_if_unknown in [UAFProtocol].
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.

<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 TAG_UAFV1_KRD and TAG_UAFV1_SIGNED_DATA.

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

### 5.2.1 TAG_UAFV1_REG_ASSERTION

The following TLV structure is generated by the authenticator during processing of a Register command. It is then delivered to FIDO Server intact, and parsed by the server. The structure embeds a TAG_UAFV1_KRD tag which among other data contains the newly generated UAuth.pub.

If the authenticator wants to append custom data to TAG_UAFV1_KRD structure (and thus sign with Attestation Key) - this data **must** be included as TAG_EXTENSION_DATA in a TAG_EXTENSION object inside TAG_UAFV1_KRD.

If the authenticator wants to send additional data to FIDO Server without signing it - this data **must** be included as TAG_EXTENSION_DATA in a TAG_EXTENSION object inside TAG_UAFV1_REG_ASSERTION and not inside TAG_UAFV1_KRD.

Currently this document only specifies ATTESTATION_BASIC_FULL, ATTESTATION_BASIC_SURROGATE and ATTESTATION_ECDAA. In case if the authenticator is required to perform "Some_Other_Attestation" on TAG_UAFV1_KRD - it **must** use the TLV tag and content defined for "Some_Other_Attestation" (defined in [FIDORegistry]).

<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>
<tr>
<td>1.2.3.3</td>
<td>UINT8 AuthenticationMode</td>
</tr>
<tr>
<td>1.2.3.4</td>
<td>UINT16</td>
</tr>
<tr>
<td>1.2.3.5</td>
<td>SignatureAlgAndEncoding</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1.2.3.5.1</td>
<td>UINT16 PublicKeyAlgAndEncoding</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 (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>
<tr>
<td>1.3.2.2</td>
<td>UINT8[] Signature</td>
</tr>
<tr>
<td>1.3.3</td>
<td>UINT16 Tag</td>
</tr>
</tbody>
</table>
| 1.3.3.1 | UINT16 Length | Multiple occurrences must be ordered. The attestation certificate **MUST** occur first. Each subsequent occurrence (if exists) **MUST** be the issuing certificate of the
previous occurrence. The last occurrence **must** be chained to one of the certificates included in field `attestationRootCertificate` in the related Metadata Statement `[FIDOMetadataStatement]`.

<table>
<thead>
<tr>
<th>1.3.3.1</th>
<th>UINT16 Length</th>
<th>Length of Attestation Cert</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.3.2</td>
<td>UINT8[] Certificate</td>
<td>Single X.509 DER-encoded [ITU-X690-2008] Attestation Certificate or a single certificate from the attestation certificate chain (see description above).</td>
</tr>
<tr>
<td>1.3 (choice 2)</td>
<td>UINT16 Tag</td>
<td>ATTESTATION_BASIC_SURROGATE</td>
</tr>
<tr>
<td>1.3.1</td>
<td>UINT16 Length</td>
<td>Length of structure</td>
</tr>
<tr>
<td>1.3.2</td>
<td>UINT16 Tag</td>
<td>TAG_SIGNATURE</td>
</tr>
<tr>
<td>1.3.2.1</td>
<td>UINT16 Length</td>
<td>Length of signature</td>
</tr>
<tr>
<td>1.3.2.2</td>
<td>UINT8[] Signature</td>
<td>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, <strong>must</strong> be included during signature computation.</td>
</tr>
<tr>
<td>1.3 (choice 3)</td>
<td>UINT16 Tag</td>
<td>ATTESTATION_ECDAA</td>
</tr>
<tr>
<td>1.3.1</td>
<td>UINT16 Length</td>
<td>Length of structure</td>
</tr>
<tr>
<td>1.3.2</td>
<td>UINT16 Tag</td>
<td>TAG_SIGNATURE</td>
</tr>
<tr>
<td>1.3.2.1</td>
<td>UINT16 Length</td>
<td>Length of signature</td>
</tr>
<tr>
<td>1.3.2.2</td>
<td>UINT8[] Signature</td>
<td>The binary ECDAA signature as specified in [FIDOEcdaaAlgorithm].</td>
</tr>
</tbody>
</table>

### 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>
</tbody>
</table>
### 1.2.2 UINT16 Length
- Length of AAID

### 1.2.2.2 UINT8[] AAID
- Authenticator Attestation ID

### 1.2.3 UINT16 Tag
- TAG_ASSERTION_INFO

### 1.2.3.1 UINT16 Length
- Length of Assertion Information

### 1.2.3.2 UINT16 AuthenticatorVersion
- Vendor assigned authenticator version.

#### 1.2.3.3 UINT8 AuthenticationMode
- Authentication Mode indicating whether user explicitly verified or not and indicating if there is a transaction content or not.
  - 0x01 means that user has been explicitly verified
  - 0x02 means that transaction content has been shown on the display and user confirmed it by explicitly verifying with authenticator

### 1.2.3.4 UINT16 SignatureAlgAndEncoding
- Signature algorithm and encoding format.
  - Refer to [FIDORegistry] 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>1.4.4.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>1.4.4.2</td>
<td>UINT16 AuthenticatorType</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.4.3</td>
<td>UINT8 MaxKeyHandles</td>
</tr>
<tr>
<td>1.4.4.4</td>
<td>UINT32 UserVerification</td>
</tr>
<tr>
<td>1.4.4.5</td>
<td>UINT16 KeyProtection</td>
</tr>
<tr>
<td>1.4.4.6</td>
<td>UINT16 MatcherProtection</td>
</tr>
<tr>
<td>1.4.4.7</td>
<td>UINT16 TransactionConfirmationDisplay</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.4.8</td>
<td>UINT16 AuthenticationAlg</td>
</tr>
<tr>
<td>1.4.5</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.4.5.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.4.5.2</td>
<td>UINT8[] ContentType</td>
</tr>
</tbody>
</table>
1.4.6 UINT16 Tag

TAG_TC_DISPLAY_PNG_CHARACTERISTICS (optional, multiple occurrences permitted)

<table>
<thead>
<tr>
<th>1.4.6.1</th>
<th>UINT16 Length</th>
<th>Length of display characteristics information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.6.2</td>
<td>UINT32 Width</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
<tr>
<td>1.4.6.3</td>
<td>UINT32 Height</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
<tr>
<td>1.4.6.4</td>
<td>UINT8 BitDepth</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
<tr>
<td>1.4.6.5</td>
<td>UINT8 ColorType</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
<tr>
<td>1.4.6.6</td>
<td>UINT8 Compression</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
<tr>
<td>1.4.6.7</td>
<td>UINT8 Filter</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
<tr>
<td>1.4.6.8</td>
<td>UINT8 Interlace</td>
<td>See [FIDOMetadataStatement] for additional information.</td>
</tr>
</tbody>
</table>

| 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

<table>
<thead>
<tr>
<th>1.4.7.1</th>
<th>UINT16 Length</th>
<th>Length of Assertion Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.7.2</td>
<td>UINT8[] AssertionScheme</td>
<td>Assertion Scheme (as defined in [UAFRegistry])</td>
</tr>
</tbody>
</table>

1.4.8 UINT16 Tag

TAG_ATTESTATION_TYPE (multiple occurrences possible)

<table>
<thead>
<tr>
<th>1.4.8.1</th>
<th>UINT16 Length</th>
<th>Length of AttestationType</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.8.2</td>
<td>UINT16 AttestationType</td>
<td>Attestation Type values are defined in [UAFRegistry] by the constants with the prefix TAG_ATTESTATION.</td>
</tr>
</tbody>
</table>

1.4.9 UINT16 Tag

TAG_SUPPORTED_EXTENSION_ID (optional, multiple occurrences possible)

<table>
<thead>
<tr>
<th>1.4.9.1</th>
<th>UINT16 Length</th>
<th>Length of SupportedExtensionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.9.2</td>
<td>UINT8[] SupportedExtensionID</td>
<td>SupportedExtensionID as a UINT8[] encoding of a UTF-8 string</td>
</tr>
</tbody>
</table>

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>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>UINT8[] AuthenticatorIndex</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>UINT8[] AppID</td>
</tr>
<tr>
<td>1.4</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.4.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.4.2</td>
<td>UINT8[] FinalChallengeHash</td>
</tr>
<tr>
<td>1.5</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.5.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.5.2</td>
<td>UINT8[] Username</td>
</tr>
<tr>
<td>1.6</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.6.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.6.2</td>
<td>UINT16 AttestationType</td>
</tr>
<tr>
<td>1.7</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.7.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.7.2</td>
<td>UINT8[] KHAccessToken</td>
</tr>
<tr>
<td>1.8</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.8.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.8.2</td>
<td>UINT8[] VerificationToken</td>
</tr>
</tbody>
</table>

### 6.2.2 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>
</tbody>
</table>
## 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):

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

2. 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)**

3. 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**.
1. If verification fails - return UAF_CMD_STATUS_ACCESS_DENIED
2. 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 \texttt{SignCounter} or \texttt{RegCounter} it must set these to 0 in TAG\textunderscore UAFV1\textunderscore KRD. The \texttt{RegCounter} must be set to 0 when a factory reset for the authenticator is performed. The \texttt{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>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>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>UINT8[] AppID</td>
</tr>
<tr>
<td>1.4</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.4.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.4.2</td>
<td>UINT8[] FinalChallengeHash</td>
</tr>
<tr>
<td>1.5</td>
<td>UINT16 Tag</td>
</tr>
<tr>
<td>1.5.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.5.2</td>
<td>UINT8[] TransactionContent</td>
</tr>
<tr>
<td>1.5</td>
<td>UINT16 Tag</td>
</tr>
</tbody>
</table>
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 UINT16 Tag</td>
<td>TAG_UAFV1_SIGN_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 Value</td>
<td>Status code returned by authenticator</td>
</tr>
<tr>
<td>1.3 (choice 1) UINT16 Tag</td>
<td>TAG_USERNAME_AND_KEYHANDLE (optional, multiple occurrences)</td>
</tr>
<tr>
<td>1.3.1 UINT16 Length</td>
<td>Length of the structure</td>
</tr>
<tr>
<td>1.3.2 UINT16 Tag</td>
<td>TAG_USERNAME</td>
</tr>
<tr>
<td>1.3.2.1 UINT16 Length</td>
<td>Length of Username</td>
</tr>
</tbody>
</table>
### 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
   - 4. If the user is not enrolled then return UAF_CMD_STATUS_USER_NOT_ENROLLED

**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).

5. 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.
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).

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)

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.
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>1.1</td>
<td>UINT16 Length</td>
</tr>
<tr>
<td>1.2</td>
<td>UINT16 Tag</td>
</tr>
</tbody>
</table>
### 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
     1. 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>
<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.

ASM 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>Register</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>Sign</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>Deregister</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>Register</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>Sign</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>Deregister</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>Register</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>Sign</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>Deregister</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.
| **Attestation Private Key** | Authenticators must protect the attestation private key as a very sensitive asset. The overall security of the authenticator depends on the protection level of this key.  
It is highly recommended to store and operate this key inside a tamper-resistant hardware module, e.g. [SecureElement].  
It is assumed by registration assertion schemes, that the authenticator has exclusive control over the data being signed with the attestation key.  
FIDO Authenticators must ensure that the attestation private key:  
1. is only used to attest authentication keys generated and protected by the authenticator, using the FIDO-defined data structures, KeyRegistrationData.  
2. is never accessible outside the security boundary of the authenticator.  
Attestation must be implemented in a way such that two different relying parties cannot link registrations, authentications or other transactions (see [UAFProtocol]). |
| **Certifications** | Vendors should strive to pass common security standard certifications with authenticators, such as [FIPS140-2], [CommonCriteria] and similar. Passing such certifications will positively impact the UAF implementation of the authenticator. |
| **Cryptographic (Crypto) Kernel** | The crypto kernel is a module of the authenticator implementing cryptographic functions (key generation, signing, wrapping, etc) necessary for UAF, and having access to UAuth.priv, Attestation Private Key and Wrap.sym.  
For optimal security, this module should reside within the same security boundary as the UAuth.priv, Attest.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. |
<p>| | It is highly recommended to use authenticated encryption while wrapping key handles with Wrap.sym. |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KeyHandle</strong></td>
<td>Algorithms such as AES-GCM and AES-CCM are most suitable for this operation.</td>
</tr>
<tr>
<td><strong>Liveness Detection / Presentation Attack Detection</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Matcher</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Private Keys (UAuth.priv and Attestation Private Key)</strong></td>
<td>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</td>
</tr>
<tr>
<td><strong>Software Based Authenticators</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
### 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
| Transaction Confirmation Display | 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]. |
|----------------------------------|------------------------------------------------------------------------------------------------|
| 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


C.2 Informative references


[NSTCBiometrics]
[RFC4086]

[SP800-38F]

[SP800-57]

[SecureElement]
GlobalPlatform Card Specifications. URL: https://www.globalplatform.org/specifications.asp

[TEE]
GlobalPlatform Trusted Execution Environment Specifications. URL: https://www.globalplatform.org/specifications.asp

[TEESecureDisplay]
GlobalPlatform Trusted User Interface API Specifications. URL: https://www.globalplatform.org/specifications.asp

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

[UAFASM]
D. Baghdasaryan; J. Kemp; R. Lindemann; B. Hill; R. Sasson. FIDO UAF Authenticator-Specific Module API. Review Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-asm-api-v1.2-ps-20201020.html
Abstract

Describes APIs and an interoperability profile for client applications to utilize FIDO UAF. This includes methods of communicating with a FIDO UAF Client for both Web platform and Android applications, transport requirements, and an HTTPS interoperability profile for sending FIDO UAF messages to a compatible server.

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.
6.4 Security Considerations for Android Implementations

7. iOS Custom URL API
   7.1 iOS-specific Definitions
      ■ 7.1.1 X-Callback-URL Transport
      ■ 7.1.2 Secret Key Generation
      ■ 7.1.3 Origin
      ■ 7.1.4 channelBindings
      ■ 7.1.5 UAFxType
   7.2 JSON Values
      ■ 7.2.1 DISCOVER
      ■ 7.2.2 DISCOVER_RESULT
      ■ 7.2.3 CHECK_POLICY
      ■ 7.2.4 CHECK_POLICY_RESULT
      ■ 7.2.5 UAF_OPERATION
      ■ 7.2.6 UAF_OPERATION_RESULT
      ■ 7.2.7 UAF_OPERATION_COMPLETION_STATUS
   7.3 Implementation Guidelines for iOS Implementations
   7.4 Security Considerations for iOS Implementations

8. Transport Binding Profile
   8.1 Transport Security Requirements
   8.2 TLS Security Requirements
   8.3 HTTPS Transport Interoperability Profile
      ■ 8.3.1 Obtaining a UAF Request message
      ■ 8.3.2 Operation enum
      ■ 8.3.3 GetUAFRequest dictionary
         ■ 8.3.3.1 Dictionary GetUAFRequest Members
      ■ 8.3.4 ReturnUAFRequest dictionary
         ■ 8.3.4.1 Dictionary ReturnUAFRequest Members
      ■ 8.3.5 SendUAFResponse dictionary
         ■ 8.3.5.1 Dictionary SendUAFResponse Members
      ■ 8.3.6 Delivering a UAF Response
      ■ 8.3.7 ServerResponse Interface
         ■ 8.3.7.1 Attributes
      ■ 8.3.8 Token interface
         ■ 8.3.8.1 Attributes
      ■ 8.3.9 TokenType enum
      ■ 8.3.10 Security Considerations

A. References
   ■ A.1 Normative references
   ■ A.2 Informative references

1. Notation

Type names, attribute names and element names are written as `code`. String literals are enclosed in "" , e.g. “UAF-TLV”.

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

The notation base64url refers to "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] without padding.

DOM APIs are described using the ECMAScript [ECMA-262] bindings for WebIDL [WebIDL-ED]. Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as required.

WebIDL dictionary members MUST NOT have a value of null.
Unless otherwise specified, if a WebIDL dictionary member is DOMString, it **MUST NOT** be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it **MUST NOT** be an empty list.

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

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

### 1.1 Key Words

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

### 2. Overview

This section is non-normative.

The FIDO UAF technology replaces traditional username and password-based authentication solutions for online services, with a stronger and simpler alternative. The core UAF protocol consists of four conceptual conversations between a FIDO UAF Client and FIDO Server: Registration, Authentication, Transaction Confirmation, and Deregistration. As specified in the core protocol, these messages do not have a defined network transport, or describe how application software that a user interfaces with can use UAF. This document describes the API surface that a client application can use to communicate with FIDO UAF Client software, and transport patterns and security requirements for delivering UAF Protocol messages to a remote server.

The reader should also be familiar with the FIDO Glossary of Terms [FIDOGlossary] and the UAF Protocol specification [UAFProtocol].

### 2.1 Audience

This document is of interest to client-side application authors that wish to utilize FIDO UAF, as well as implementers of web browsers, browser plugins and FIDO clients, in that it describes the API surface they need to expose to application authors.

### 2.2 Scope

This document describes:

- The local ECMAScript [ECMA-262] API exposed by a FIDO UAF-enabled web browser to client-side web applications.
- The mechanisms and APIs for Android [ANDROID] applications to discover and utilize a shared FIDO UAF Client service.
- The general security requirements for applications initiating and transporting UAF protocol exchanges.
- An interoperability profile for transporting FIDO UAF messages over HTTPS [RFC2818].

The following are out of scope for this document:

- The format and details of the underlying UAF Protocol messages
- APIs for, and any details of interactions between FIDO Server software and the server-side application stack.

### 2.3 Architecture

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

![UAF Application API Architecture and Transport Layers](image)

This document describes the shaded components in Fig 1.

### 2.3.1 Protocol Conversation

The core UAF protocol consists of five conceptual phases:

- **Discovery** allows the relying party server to determine the availability of FIDO capabilities at the client, including metadata about the available authenticators.
- **Registration** allows the client to generate and associate new key material with an account at the relying party server, subject to policy set by the server and acceptable attestation that the authenticator and registration matches that policy.
- **Authentication** allows a user to provide an account identifier, proof-of-possession of previously registered key material associated with that identifier, and potentially other attested data, to the relying party server.
- **Transaction Confirmation** allows a server to request that a FIDO client and authenticator with the appropriate capabilities display some information to the user, request that the user authenticate locally to their FIDO authenticator to confirm it, and provide proof-of-possession of previously registered key material and an attestation of the confirmation back to the relying party server.
- **Deregistration** allows a relying party server to tell an authenticator to forget selected locally managed key material associated with that relying party in case such keys are no longer considered valid by the relying party.

Discovery does not involve a protocol exchange with the FIDO Server. However, the information available through the discovery APIs might be communicated back to the server in an application-specific manner, such as by obtaining a UAF protocol request message containing an authenticator policy tailored to the specific capabilities of the FIDO user device.

Although the UAF protocol abstractly defines the FIDO server as the initiator of requests, UAF client applications working as described in this document will always transport UAF protocol messages over a client-initiated request/response protocol such as HTTP.

The protocol flow from the point of view of the relying party client application for registration, authentication, and transaction confirmation is as follows:

1. The client application either explicitly contacts the server to obtain a UAF Protocol Request Message, or this message is delivered along with other client application content.
2. The client application invokes the appropriate API to pass the UAF protocol request message asynchronously to the FIDO UAF Client, and receives a set of callbacks.
3. The FIDO UAF Client performs any necessary interactions with the user and authenticator(s) to complete the request and uses a callback to either notify the client application of an error, or to return a UAF response message.
4. The client application delivers the UAF response message to the server over a transport protocol such as HTTP.
5. The server optionally returns an indication of the results of the operation and additional data such as authorization tokens or a redirect.
6. The client application optionally uses the appropriate API to inform the FIDO UAF Client of the results of the operation. This allows the FIDO UAF Client to perform "housekeeping" tasks for a better user experience, e.g. by not attempting to use again later a key that the server refused to register.
7. The client application optionally processes additional data returned to it in an application-specific manner, e.g. processing new authorization tokens, redirecting the user to a new resource or interpreting an error code to determine if and how it should retry a failed operation.

Deregister does not involve a UAF protocol round-trip. If the relying party server instructs the client application to perform a deregistration, the client application simply delivers the UAF protocol Request message to the FIDO UAF Client using the appropriate API. The FIDO UAF Client does not return the results of a deregister operation to the relying party client application or FIDO Server.

UAF protocol Messages are JSON [ECMA-404] structures, but client applications are discouraged from modifying them. These messages may contain embedded cryptographic integrity protections and any modifications might invalidate the messages from the point of view of the FIDO UAF Client or Server.

### 3. Common Definitions

*This section is normative.*

These elements are shared by several APIs and layers.

#### 3.1 UAF Status Codes

This table lists UAF protocol status codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>OK. Operation completed</td>
</tr>
<tr>
<td>1202</td>
<td>Accepted. Message accepted, but not completed at this time. The RP may need time to process the attestation, run risk scoring, etc. The server <strong>SHOULD NOT</strong> send an authenticationToken with a 1202 response</td>
</tr>
<tr>
<td>1400</td>
<td>Bad Request. The server did not understand the message</td>
</tr>
</tbody>
</table>

**NOTE**

These codes indicate the result of the UAF operation at the FIDO Server. They do not represent the HTTP [RFC7230] layer or other transport layers. These codes are intended for consumption by both the client-side web app and FIDO UAF Client to inform application-specific error reporting, retry and housekeeping behavior.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1401</td>
<td>Unauthorized. The userid must be authenticated to perform this operation, or this KeyID is not associated with this UserID.</td>
</tr>
<tr>
<td>1403</td>
<td>Forbidden. The userid is not allowed to perform this operation. Client <em>SHOULD NOT</em> retry</td>
</tr>
<tr>
<td>1404</td>
<td>Not Found.</td>
</tr>
<tr>
<td>1408</td>
<td>Request Timeout.</td>
</tr>
<tr>
<td>1480</td>
<td>Unknown AAID. The server was unable to locate authoritative metadata for the AAID.</td>
</tr>
<tr>
<td>1481</td>
<td>Unknown KeyID. The server was unable to locate a registration for the given UserID and KeyID combination.</td>
</tr>
<tr>
<td></td>
<td>This error indicates that there is an invalid registration on the user's device. It is recommended that FIDO UAF Client deletes the key from local device when this error is received.</td>
</tr>
<tr>
<td>1490</td>
<td>Channel Binding Refused. The server refused to service the request due to a missing or mismatched channel binding(s).</td>
</tr>
<tr>
<td>1491</td>
<td>Request Invalid. The server refused to service the request because the request message nonce was unknown, expired or the server has previously serviced a message with the same nonce and user ID.</td>
</tr>
<tr>
<td>1492</td>
<td>Unacceptable Authenticator. The authenticator is not acceptable according to the server's policy, for example because the capability registry used by the server reported different capabilities than client-side discovery.</td>
</tr>
<tr>
<td>1493</td>
<td>Revoked Authenticator. The authenticator is considered revoked by the server.</td>
</tr>
<tr>
<td>1494</td>
<td>Unacceptable Key. The key used is unacceptable. Perhaps it is on a list of known weak keys or uses insecure parameter choices.</td>
</tr>
<tr>
<td>1495</td>
<td>Unacceptable Algorithm. The server believes the authenticator to be capable of using a stronger mutually-agreeable algorithm than was presented in the request.</td>
</tr>
<tr>
<td>1496</td>
<td>Unacceptable Attestation. The attestation(s) provided were not accepted by the server.</td>
</tr>
<tr>
<td>1497</td>
<td>Unacceptable Client Capabilities. The server was unable or unwilling to use required capabilities provided supplementally to the authenticator by the client software.</td>
</tr>
<tr>
<td>1498</td>
<td>Unacceptable Content. There was a problem with the contents of the message and the server was unwilling or unable to process it.</td>
</tr>
<tr>
<td>1500</td>
<td>Internal Server Error</td>
</tr>
</tbody>
</table>

### 4. Shared Definitions

*This section is normative.*

**NOTE**

This section defines a number of JSON structures, specified with WebIDL [WebIDL-ED]. These structures are shared among APIs for multiple target platforms.

#### 4.1 UAFMessage Dictionary

The UAFMessage dictionary is a wrapper object that contains the raw UAF protocol Message and additional JSON data that may be used to carry application-specific data for use by either the client application or FIDO UAF Client.

**WebIDL**

```webidl
dictionary UAFMessage {
  required DOMString uafProtocolMessage;
  Object additionalData;
};
```

#### 4.1.1 Dictionary UAFMessage Members

- **uafProtocolMessage** of type **required DOMString**
  This key contains the UAF protocol Message that will be processed by the FIDO UAF Client or Server. Modification by the client application may invalidate the message. A client application _MAY_ examine the contents of a message, for example, to determine if a message is still fresh. Details of the structure of the message can be found in the UAF protocol Specification [UAFProtocol].

- **additionalData** of type **Object**
  This key allows the FIDO Server or client application to attach additional data for use by the FIDO UAF Client as a JSON object, or the FIDO UAF Client or client application to attach additional data for use by the client application.
4.2 Version interface

Describes a version of the UAF protocol or FIDO UAF Client for compatibility checking.

```webidl
interface Version {
    readonly attribute unsigned short major;
    readonly attribute unsigned short minor;
};
```

4.2.1 Attributes

- `major` of type `unsigned short`, readonly
  Major version number.

- `minor` of type `unsigned short`, readonly
  Minor version number.

4.3 Authenticator interface

Used by several phases of UAF, the `Authenticator` interface exposes a subset of both verified metadata [FIDOMetadataStatement] and transient information about the state of an available authenticator.

```webidl
interface Authenticator {
    readonly attribute DOMString title;
    readonly attribute AAID aaid;
    readonly attribute DOMString description;
    readonly attribute Version[] supportedUAFVersions;
    readonly attribute unsigned short authenticationAlgorithm;
    readonly attribute attestationTypes;
    readonly attribute unsigned short userVerification;
    readonly attribute unsigned short attestationTypes;
    readonly attribute unsigned short keyProtection;
    readonly attribute unsigned short matcherProtection;
    readonly attribute unsigned long attachmentHint;
    readonly attribute boolean isSecondFactorOnly;
    readonly attribute DOMString tcDisplay;
    readonly attribute DOMString tcDisplayContentType;
    readonly attribute DisplayPNGCharacteristicsDescriptor[] tcDisplayPNGCharacteristics;
    readonly attribute DOMString icon;
    readonly attribute DOMString[] supportedExtensionIDs;
};
```

4.3.1 Attributes

- `title` of type `DOMString`, readonly
  A short, user-friendly name for the authenticator.

**NOTE**

This text must be localized for current locale.

If the ASM doesn’t return a title in the `AuthenticatorInfo` object [UAFASM], the FIDO UAF Client must generate a title based on the other fields in `AuthenticatorInfo`, because `title` must not be empty (see section 1. Notation).

- `aaid` of type `AAID`, readonly
  The `Authenticator Attestation ID`, which identifies the type and batch of the authenticator. See [UAFProtocol] for the definition of the AAID structure.

- `description` of type `DOMString`, readonly
  A user-friendly description string for the authenticator.

**NOTE**

This text must be localized for current locale.

It is intended to be displayed to the user. It might deviate from the description specified in the authenticator’s metadata.
If the ASM doesn't return a description in the AuthenticatorInfo object [UAFASM], the FIDO UAF Client must generate a meaningful description to the calling App based on the other fields in AuthenticatorInfo, because description must not be empty (see section 1. Notation).

supportedUAFVersions of type array of Version, readonly
Indicates the UAF protocol Versions supported by the authenticator.

assertionScheme of type DOMString, readonly
The assertion scheme the authenticator uses for attested data and signatures.

Assertion scheme identifiers are defined in the UAF Registry of Predefined Values. [UAFRegistry]

authenticationAlgorithm of type unsigned short, readonly
Supported Authentication Algorithm. The value MUST be related to constants with prefix ALG_SIGN.

attestationTypes of type array of unsigned short, readonly
A list of supported attestation types. The values are defined in [UAFRegistry] by the constants with the prefix TAG_ATTESTATION.

userVerification of type unsigned long, readonly
A set of bit flags indicating the user verification methods supported by the authenticator. The algorithm for combining the flags is defined in [UAFProtocol], section 3.1.12.1. The values are defined by the constants with the prefix USER_VERIFY.

keyProtection of type unsigned short, readonly
A set of bit flags indicating the key protection used by the authenticator. The values are defined by the constants with the prefix KEY_PROTECTION.

matcherProtection of type unsigned short, readonly
A set of bit flags indicating the matcher protection used by the authenticator. The values are defined by the constants with the prefix MATCHER_PROTECTION.

attachmentHint of type unsigned long, readonly
A set of bit flags indicating how the authenticator is currently connected to the FIDO User Device. The values are defined by the constants with the prefix ATTACHMENT_HINT.

NOTE
Because the connection state and topology of an authenticator may be transient, these values are only hints that can be used in applying server-supplied policy to guide the user experience. This can be used to, for example, prefer a device that is connected and ready for authenticating or confirming a low-value transaction, rather than one that is more secure but requires more user effort.

These values are not reflected in authenticator metadata and cannot be relied upon by the relying party, although some models of authenticator may provide attested measurements with similar semantics as part of UAF protocol messages.

isSecondFactorOnly of type boolean, readonly
Indicates whether the authenticator can only be used as a second-factor.

tcDisplay of type unsigned short, readonly
A set of bit flags indicating the availability and type of transaction confirmation display. The values are defined by the constants with the prefix TRANSACTION_CONFIRMATION_DISPLAY.

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

tcDisplayContentType of type DOMString, readonly
The MIME content-type [RFC2045] supported by the transaction confirmation display, such as text/plain or image/png.

This value MUST be non-empty if transaction confirmation is supported (tcDisplay is non-zero).

tcDisplayPNGCharacteristics of type array of DisplayPNGCharacteristicsDescriptor, readonly
The set of PNG characteristics currently supported by the transaction confirmation display (if any).

NOTE
See [FIDOMetadataStatement] for additional information on the format of this field and the definition of the DisplayPNGCharacteristicsDescriptor structure.
This list **MUST** be non-empty if PNG-image based transaction confirmation is supported, i.e. \( \text{tcDisplay} \) is non-zero and \( \text{tcDisplayContentType} \) is image/png.

**icon** of type \( \text{DOMString} \), readonly
A PNG [PNG] icon for the authenticator, encoded as a [data: url [RFC2397]].

NOTE
If the ASM doesn't return an icon in the AuthenticatorInfo object [UAFASM], the FIDO UAF Client must set a default icon, because **icon** must not be empty (see section 1. Notation).

**supportedExtensionIDs** of type array of \( \text{DOMString} \), readonly
A list of supported UAF protocol extension identifiers. These **MAY** be vendor-specific.

### 4.3.2 Authenticator Interface Constants

A number of constants are defined for use with the bit flag fields \( \text{userVerification} \), \( \text{keyProtection} \), \( \text{attachmentHint} \), and \( \text{tcDisplay} \). To avoid duplication and inconsistencies, these are defined in the FIDO Registry of Predefined Values [FIDORegistry].

### 4.4 DiscoveryData dictionary

```
WebIDL
dictionary DiscoveryData {
  required Version[] supportedUAFVersions;
  required DOMString clientVendor;
  required Version clientVersion;
  required Authenticator[] availableAuthenticators;
};
```

**NOTE**
If the ASM doesn't return an icon in the AuthenticatorInfo object [UAFASM], the FIDO UAF Client must set a default icon, because **icon** must not be empty (see section 1. Notation).

### 4.4.1 Dictionary DiscoveryData Members

**supportedUAFVersions** of type array of required `Version`
A list of the FIDO UAF protocol versions supported by the client, most-preferred first.

**clientVendor** of type required `DOMString`
The vendor of the FIDO UAF Client.

**clientVersion** of type required `Version`
The version of the FIDO UAF Client. This is a vendor-specific version for the client software, not a UAF version.

**availableAuthenticators** of type array of required `Authenticator`
An array containing Authenticator dictionaries describing the available UAF authenticators. The order is not significant. The list **MAY** be empty.

### 4.5 ErrorCode interface

```
WebIDL
interface ErrorCode {
  const short NO_ERROR = 0x0;
  const short WAIT_USER_ACTION = 0x01;
  const short INSECURE_TRANSPORT = 0x02;
  const short USER_CANCELLED = 0x03;
  const short UNSUPPORTED_VERSION = 0x04;
  const short NO_SUITABLE_AUTHENTICATOR = 0x05;
  const short PROTOCOL_ERROR = 0x06;
  const short UNTRUSTED_FACET_ID = 0x07;
  const short KEY_DISAPPEARED_PERMANENTLY = 0x09;
  const short AUTHENTICATOR_ACCESS_DENIED = 0x0e;
  const short INVALID_TRANSACTION_CONTENT = 0x0d;
  const short USER_NOT_ENROLLED = 0x0f;
  const short SYSTEM_INTERRUPTED = 0x00;
  const short UNSUPPORTED_AUTHENTICATOR_RESOURCES = 0x0f;
  const short USER_LOCKOUT = 0x10;
  const short USER_NOT_RESPONSIVE = 0x11;
  const short INSUFFICIENT_AUTHENTICATOR_RESOURCES = 0x12;
  const short INSTRUMENTED = 0xFF;
};
```
4.5.1 Constants

NO_ERROR of type short
   The operation completed with no error condition encountered. Upon receipt of this code, an application should no longer expect an associated UAFResponseCallback to fire.

WAIT_USER_ACTION of type short
   Waiting on user action to proceed. For example, selecting an authenticator in the FIDO client user interface, performing user verification, or completing an enrollment step with an authenticator.

INSECURE_TRANSPORT of type short
   window.location.protocol is not "https" or the DOM contains insecure mixed content.

USER_CANCELLED of type short
   The user declined any necessary part of the interaction to complete the registration.

UNSUPPORTED_VERSION of type short
   The UAFMessage does not specify a protocol version supported by this FIDO UAF Client.

NO_SUITABLE_AUTHENTICATOR of type short
   No authenticator matching the authenticator policy specified in the UAFMessage is available to service the request, or the user declined to consent to the use of a suitable authenticator.

PROTOCOL_ERROR of type short
   A violation of the UAF protocol occurred. The interaction may have timed out; the origin associated with the message may not match the origin of the calling DOM context, or the protocol message may be malformed or tampered with.

UNTRUSTED_FACET_ID of type short
   The client declined to process the operation because the caller's calculated facet identifier was not found in the trusted list for the application identifier specified in the request message.

KEY_DISAPPEARED_PERMANENTLY of type short
   The UAuth key disappeared from the authenticator and cannot be restored.

NOTE
   The RP App might want to re-register the authenticator in this case.

AUTHENTICATOR_ACCESS_DENIED of type short
   The authenticator denied access to the resulting request.

INVALID_TRANSACTION_CONTENT of type short
   Transaction content cannot be rendered, e.g. format doesn't fit authenticator's need.

NOTE
   The transaction content format requirements are specified in the authenticator's metadata statement.

USER_NOT_RESPONSIVE of type short
   The user took too long to follow an instruction, e.g. didn't swipe the finger within the accepted time.

INSUFFICIENT_AUTHENTICATOR_RESOURCES of type short
   Insufficient resources in the authenticator to perform the requested task.

USER_LOCKOUT of type short
   The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that. For example, an authenticator could allow the user to enter an alternative password to re-enable the use of fingerprints after too many failed finger verification attempts. This error will be reported if such method either doesn't exist or the ASM / authenticator cannot automatically trigger it.

USER_NOT_ENROLLED of type short
   The operation failed because the user is not enrolled to the authenticator and the authenticator cannot automatically trigger user enrollment.

SYSTEM_INTERRUPTED of type short
   The system interrupted the operation. Retry might make sense.

UNKNOWN of type short
5. DOM API

This section is normative.

This section describes the API details exposed by a web browser or browser plugin to a client-side web application executing in a Document [DOM] context.

5.1 Feature Detection

FIDO's UAF DOM APIs are rooted in a new fido object, a property of window.navigator code; the existence and properties of which MAY be used for feature detection.

Example 1

```html
<script>
if(!window.navigator.fido.uaf) { var useUAF = true; }
</script>
```

5.2 uaf Interface

The window.navigator.fido.uaf interface is the primary means of interacting with the FIDO UAF Client. All operations are asynchronous.

```
interface uaf {
    void discover (DiscoveryCallback completionCallback, ErrorCallback errorCallback);
    void checkPolicy (UAFMessage message, ErrorCallback cb);
    void processUAFOperation (UAFMessage message, UAFResponseCallback completionCallback, ErrorCallback errorCallback);
    void notifyUAFResult (int responseCode, UAFMessage uafResponse);
}
```

5.2.1 Methods

**discover**

Discover if the user's client software and devices support UAF and if authenticator capabilities are available that it may be willing to accept for authentication.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Nullable</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>completionCallback</td>
<td>DiscoveryCallback</td>
<td></td>
<td></td>
<td>The callback that receives DiscoveryData from the FIDO UAF Client.</td>
</tr>
<tr>
<td>errorCallback</td>
<td>ErrorCallback</td>
<td></td>
<td></td>
<td>A callback function to receive error and progress events.</td>
</tr>
</tbody>
</table>

*Return type:* `void`

**checkPolicy**

Ask the browser or browser plugin if it would be able to process the supplied request message without prompting the user.

Unlike other operations using an ErrorCallback, this operation MUST always trigger the callback and return NO_ERROR if it believes that the message can be processed and a suitable authenticator matching the embedded policy is available, or the appropriate ErrorCode value otherwise.

**NOTE**

Because this call should not prompt the user, it should not incur a potentially disrupting context-switch even if the FIDO UAF Client is implemented out-of-process.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Nullable</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>UAFMessage</td>
<td></td>
<td></td>
<td>A UAFMessage containing the policy and operation to be tested.</td>
</tr>
<tr>
<td>cb</td>
<td>ErrorCallback</td>
<td></td>
<td></td>
<td>The callback function which receives the status of the operation.</td>
</tr>
</tbody>
</table>

*Return type:* `void`
processUAFOperation

Invokes the FIDO UAF Client, transferring control to prompt the user as necessary to complete the operation, and returns to the callback a message in one of the supported protocol versions indicated by the UAFMessage.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Nullable</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>UAFMessage</td>
<td></td>
<td></td>
<td>The UAFMessage to be used by the FIDO client software.</td>
</tr>
<tr>
<td>completionCallback</td>
<td>UAFResponseCallback</td>
<td></td>
<td></td>
<td>The callback that receives the client response UAFMessage from the FIDO UAF Client, to be delivered to the relying party server.</td>
</tr>
<tr>
<td>errorCallback</td>
<td>ErrorCallback</td>
<td></td>
<td></td>
<td>A callback function to receive error and progress events from the FIDO UAF Client.</td>
</tr>
</tbody>
</table>

Return type: void

notifyUAFResult

Used to indicate the status code resulting from a FIDO UAF message delivered to the remote server. Applications MUST make this call when they receive a UAF status code from a server. This allows the FIDO UAF Client to perform housekeeping for a better user experience, for example not attempting to use keys that a server refused to register.

NOTE

If, and how, a status code is delivered by the server, is application and transport specific. A non-normative example can be found below in the HTTPS Transport Interoperability Profile.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Nullable</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>responseCode</td>
<td>int</td>
<td></td>
<td></td>
<td>The uafResult field of a ServerResponse.</td>
</tr>
<tr>
<td>uafResponse</td>
<td>UAFMessage</td>
<td></td>
<td></td>
<td>The UAFMessage to which this responseCode applies.</td>
</tr>
</tbody>
</table>

Return type: void

5.3 UAFResponseCallback

A UAFResponseCallback is used upon successful completion of an asynchronous operation by the FIDO UAF Client to return the protocol response message to the client application for transport to the server.

NOTE

This callback is also called in the case of deregistration completion, even though the response object is empty then.

WebIDL

```
callback UAFResponseCallback = void (UAFMessage uafResponse);
```

5.3.1 Callback UAFResponseCallback Parameters

uafResponse of type UAFMessage

The message and any additional data representing the FIDO UAF Client's response to the server's request message.

5.4 DiscoveryCallback

A DiscoveryCallback is used upon successful completion of an asynchronous discover operation by the FIDO UAF Client to return the DiscoveryData to the client application.

WebIDL

```
callback DiscoveryCallback = void (DiscoveryData data);
```

5.4.1 Callback DiscoveryCallback Parameters

data of type DiscoveryData

Describes the current state of FIDO UAF client software and authenticators available to the application.

5.5 ErrorCallback
An ErrorCallback is used to return progress and error codes from asynchronous operations performed by the FIDO UAF Client.

```webidl
callback ErrorCallback = void (ErrorCode code);
```

### 5.5.1 Callback ErrorCallback Parameters

**code** of type **ErrorCode**

A value from the **ErrorCode** interface indicating the result of the operation.

For certain operations, an ErrorCallback may be called multiple times, for example with the **WAIT_USER_ACTION** code.

### 5.6 Privacy Considerations for the DOM API

This section is non-normative.

Differences in the FIDO capabilities on a user device may (among many other characteristics) allow a server to "fingerprint" a remote client and attempt to persistently identify it, even in the absence of any explicit session state maintenance mechanism. Although it may contribute some amount of signal to servers attempting to fingerprint clients, the attributes exposed by the Discovery API are designed to have a large anonymity set size and should present little or no qualitatively new privacy risk. Nonetheless, an unusual configuration of FIDO Authenticators may be sufficient to uniquely identify a user.

It is recommended that user agents expose the Discovery API to all applications without requiring explicit user consent by default, but user agents or FIDO Client implementers should provide users with the means to opt-out of discovery if they wish to do so for privacy reasons.

### 5.7 Security Considerations for the DOM API

This section is non-normative.

#### 5.7.1 Insecure Mixed Content

When FIDO UAF APIs are called and operations are performed in a **Document** context in a web user agent, such a context **MUST NOT** contain insecure mixed content. The exact definition insecure mixed content is specific to each user agent, but generally includes any script, plugins and other "active" content, forming part of or with access to the DOM, that was not itself loaded over HTTPS.

The UAF APIs must immediately trigger the **ErrorCallback** with the **INSECURE_TRANSPORT** code and cease any further processing if any APIs defined in this document are invoked by a Document context that was not loaded over a secure transport and/or which contains insecure mixed content.

#### 5.7.2 The Same Origin Policy, HTTP Redirects and Cross-Origin Content

When retrieving or transporting UAF protocol messages over HTTP, it is important to maintain consistency among the web origin of the document context and the origin embedded in the UAF protocol message. Mismatches may cause the protocol to fail or enable attacks against the protocol. Therefore:

FIDO UAF messages should not be transported using methods that opt-out of the Same Origin Policy [SOP], for example, using `<script src="url">` to non-same-origin URLs or by setting the `Access-Control-Allow-Origin` header at the server.

When transporting FIDO UAF messages using XMLHttpRequest [XHR] the client should not follow redirects that are to URLs with a different origin than the requesting document.

FIDO UAF messages should not be exposed in HTTP responses where the entire response body parses as valid ECMAScript. Resources exposed in this manner may be subject to unauthorized interactions by hostile applications hosted at untrusted origins through cross-origin embedding using `<script src="url">`.

Web applications should not share FIDO UAF messages across origins through channels such as `postMessage()` [webmessaging].

### 5.8 Implementation Notes for Browser/Plugin Authors

This section is non-normative.

Web applications utilizing UAF depend on services from the web browser as a trusted platform. The APIs for web applications do not provide a means to assert an origin as an application identity for the purposes of FIDO operations as this will be provided to the FIDO UAF Client by the browser based on its privileged understanding of the actual origin context.

The browser must enforce that the web origin communicated to the FIDO UAF Client as the application identity is accurate.

The browser must also enforce that resource instances containing insecure mixed-content cannot utilize the UAF DOM APIs.
6. Android Intent API

This section is normative.

This section describes how an Android [ANDROID] client application can locate and communicate with a conforming FIDO Client installation operating on the host device.

NOTE

As with web applications, a variety of integration patterns are possible on the Android platform. The API described here allows an app to communicate with a shared FIDO UAF Client on the user device in a loosely-coupled fashion using Android Intents.

6.1 Android-specific Definitions

6.1.1 org.fidoalliance.uaf.permissions.FIDO_CLIENT

FIDO UAF Clients running on Android versions prior to Android 5 MUST declare the org.fidoalliance.uaf.permissions.FIDO_CLIENT permission and they also MUST declare the related "uses-permission". See the below example of this permission expressed in an Android app manifest file <permission/> and <uses-permission/> element [AndroidAppManifest].

FIDO UAF Clients running on Android version 5 or later MUST NOT declare this permission and they also MUST NOT declare the related "uses-permission".

EXAMPLE 2

<permission
  android:name="org.fidoalliance.uaf.permissions.FIDO_CLIENT"
  android:label="Act as a FIDO Client."
  android:description="This application acts as a FIDO Client. It may access authentication devices available on the system, create and delete FIDO registrations on behalf of other applications."
  android:protectionLevel="dangerous"
/>
<uses-permission android:name="org.fidoalliance.uaf.permissions.FIDO_CLIENT"/>

NOTE

- Since FIDO Clients perform security relevant tasks (e.g. verifying the AppID/FacetID relation and asking for user consent), users should carefully select the FIDO Clients they use. Requiring apps acting as FIDO Clients to declare and use this permission allows them to be identified as such to users.
- There are not any FIDO Client resources needing "protection" based upon the FIDO_CLIENT permission. The reason for having FIDO Client declare the FIDO_CLIENT permission is solely that users should be able to carefully decide which FIDO Clients to install.
- Android version 5 changed the way it handles the case where multiple apps declare the same permission [Android5Changes]; it blocks the installation of all subsequent apps declaring that permission.
- The best way to flag the fact that an app may act as a FIDO Client needs to be determined for Android version 5.

6.1.2 org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER

Android applications requesting services from the FIDO UAF Client can do so under their own identity, or they can act as the user’s agent by explicitly declaring an RFC6454 [RFC6454] serialization of the remote server's origin when invoking the FIDO UAF Client.

An application that is operating on behalf of a single entity MUST NOT set an explicit origin. Omitting an explicit origin will cause the FIDO UAF Client to determine the caller’s identity as android:apk-key-hash:hash-of-public-key. The FIDO UAF Client will then compare this with the list of authorized application facets for the target AppID and proceed if it is listed as trusted.

NOTE

See the UAF Protocol Specification [UAFProtocol] for more information on application and facet identifiers.

If the application is explicitly intended to operate as the user's agent in the context of an arbitrary number of remote applications (as when implementing a full web browser) it may set its origin to the RFC6454 [RFC6454] Unicode serialization of the remote application's Origin. The application MUST satisfy the necessary conditions described in Transport Security Requirements for authenticating the remote server before
setting the origin.

Use of the origin parameter requires the application to declare the `org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER` permission, and the FIDO UAF Client must verify that the calling application has this permission before processing the operation.

**EXAMPLE 3**

```xml
<permission
    android:name="org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER"
    android:label="Act as a browser for FIDO registrations."
    android:description="This application may act as a web browser, creating new and accessing existing FIDO registrations for any domain."
    android:protectionLevel="dangerous"
/>
```

### 6.1.3 channelBindings

*This section is non-normative.*

In the DOM API, the browser or browser plugin is responsible for supplying any available channel binding information to the FIDO Client, but an Android application, as the direct owner of the transport channel, must provide this information itself.

The `channelBindings` data structure is:

```java
Map<String,String>
```

with the keys as defined for the `ChannelBinding` structure in the UAF Protocol Specification. [UAFProtocol]

The use of channel bindings for TLS helps assure the server that the channel over which UAF protocol messages are transported is the same channel the legitimate client is using and that messages have not been forwarded through a malicious party.

UAF defines support for the `tls-unique` and `tls-server-end-point` bindings from [RFC5929], as well as server certificate and ChannelID [ChannelID] bindings. The client should supply all channel binding information available to it.

Missing or invalid channel binding information may cause a relying party server to reject a transaction.

### 6.1.4 UAFIntentType enumeration

This enumeration describes the type of operation for the intent implementing the Android API.

**NOTE**

UAF uses only a single intent to simplify behavior in the situation even where multiple FIDO clients may be installed. In such a case, the user will be prompted which of the installed FIDO UAF clients should be used to handle an implicit intent.

If the user selected to make different FIDO UAF Clients the default for different intents representing different phases, it could produce inconsistent results or fail to function at all.

If the application workflow requires multiple calls to the client (and it usually does) the application should read the `componentName` from the intent extras it receives from `startActivityForResult()` and pass it to `setComponent()` for subsequent intents to be sure they are explicitly resolved to the same FIDO UAF Client.

**WebIDL**

```idl
enum UAFIntentType {
    "DISCOVER",
    "DISCOVER_RESULT",
    "CHECK_POLICY",
    "CHECK_POLICY_RESULT",
    "UAF_OPERATION",
    "UAF_OPERATION_RESULT",
    "UAF_OPERATION_COMPLETION_STATUS"
};
```

**Enumeration description**

<table>
<thead>
<tr>
<th>UAFIntentType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISCOVER</td>
<td>Discovery</td>
</tr>
<tr>
<td>DISCOVER_RESULT</td>
<td>Discovery results</td>
</tr>
<tr>
<td>CHECK_POLICY</td>
<td>Perform a no-op check if a message could be processed.</td>
</tr>
<tr>
<td>CHECK_POLICY_RESULT</td>
<td>Check Policy results.</td>
</tr>
<tr>
<td>UAF_OPERATION</td>
<td>Process a Registration, Authentication, Transaction Confirmation or Deregistration message.</td>
</tr>
</tbody>
</table>
6.2 org.fidoalliance.intent.FIDO_OPERATION Intent

All interactions between a FIDO UAF Client and an application on Android take place via a single Android intent:

org.fidoalliance.intent.FIDO_OPERATION

The specifics of the operation are carried by the MIME media type and various extra data included with the intent.

The operations described in this document are of MIME media type application/fido.uaf_client+json and this MUST be set as the type attribute of any intent.

The following table shows what intent extras are expected, depending on the value of the UAFIntentType extra:

<table>
<thead>
<tr>
<th>UAFIntentType value</th>
<th>discoveryData</th>
<th>componentName</th>
<th>errorCode</th>
<th>message</th>
<th>origin</th>
<th>channelBindings</th>
<th>responseCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DISCOVER&quot;</td>
<td>OPTIONAL</td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;DISCOVER_RESULT&quot;</td>
<td></td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;CHECK_POLICY&quot;</td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;CHECK_POLICY_RESULT&quot;</td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;UAF_OPERATION&quot;</td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;UAF_OPERATION_RESULT&quot;</td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;UAF_OPERATION_COMPLETION_STATUS&quot;</td>
<td>REQUIRED</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REQUIRED</td>
</tr>
</tbody>
</table>

6.2.1 UAFIntentType.DISCOVER

This Android intent invokes the FIDO UAF Client to discover the available authenticators and capabilities. The FIDO UAF Client generally will not show a UI associated with the handling of this intent, but immediately return the JSON structure. The calling application cannot depend on this however, as the FIDO UAF Client MAY show a UI for privacy purposes, allowing the user to choose whether and which authenticators to disclose to the calling application.

This intent MUST be invoked with startActivityForResult().
6.2.2 UAFIntentType.DISCOVER_RESULT

An intent with this type is returned by the FIDO UAF Client as an argument to onActivityResult() in response to receiving an intent of type DISCOVER.

If the resultCode passed to onActivityResult() is RESULT_OK, and the intent extra errorCode is NO_ERROR, this intent has an extra, discoveryData, containing a String representation of a DiscoveryData JSON dictionary with the available authenticators and capabilities.

6.2.3 UAFIntentType.CHECK_POLICY

This intent invokes the FIDO UAF Client to discover if it would be able to process the supplied message without prompting the user. The action handling this intent SHOULD NOT show a UI to the user.

This intent requires the following extras:

- message, containing a String representation of a UAFMessage representing the request message to test.
- origin, an OPTIONAL extra that allows a caller with the org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER permission to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

This intent MUST be invoked with startActivityForResult().

6.2.4 UAFIntentType.CHECK_POLICY_RESULT

This Android intent is returned by the FIDO UAF Client as an argument to onActivityResult() in response to receiving a CHECK_POLICY intent.

In addition to the resultCode passed to onActivityResult(), this intent has an extra, errorCode, containing an ErrorCode value indicating the specific error condition or NO_ERROR if the FIDO UAF Client could process the message.

6.2.5 UAFIntentType.UAF_OPERATION

This Android intent invokes the FIDO UAF Client to process the supplied request message and return a response message ready for delivery to the FIDO UAF Server.

The sender SHOULD assume that the FIDO UAF Client will display a user interface allowing the user to handle this intent, for example, prompting the user to complete their verification ceremony.

This intent requires the following extras:

- message, containing a String representation of a UAFMessage representing the request message to process.
- channelBindings, containing a String representation of a JSON dictionary as defined by the ChannelBinding structure in the FIDO UAF Protocol Specification [UAFProtocol].
- origin, an OPTIONAL parameter that allows a caller with the org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER permission to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

This intent MUST be invoked with startActivityForResult().

6.2.6 UAFIntentType.UAF_OPERATION_RESULT

This intent is returned by the FIDO UAF Client as an argument to onActivityResult(), in response to receiving a UAF_OPERATION intent.

If the resultCode passed to onActivityResult() is RESULT_CANCELLED, this intent will have an extra, errorCode, parameter, containing an ErrorCode value indicating the specific error condition.

If the resultCode passed to onActivityResult() is RESULT_OK, and the errorCode is NO_ERROR, this intent has a message, containing a String representation of a UAFMessage, being the UAF protocol response message to be delivered to the FIDO Server.

6.2.7 UAFIntentType.UAF_OPERATION_COMPLETION_STATUS

This intent MUST be delivered to the FIDO UAF Client to indicate the processing status of a FIDO UAF message delivered to the remote server. This is especially important as a new registration may be considered by the client to be in a pending state until it is communicated that the server accepted it.

6.3 Alternate Android AIDL Service UAF Client Implementation

The Android Intent API can also be implemented using Android AIDL services as an alternative transport mechanism to Android Intents. While Android Intents work at the UI layer, Android AIDL services are performed at a lower level. This can ease integration with relying party apps, since UAF requests can be fulfilled without interfering with existing relying party app UI and application lifecycle behavior.
The UAF Android AIDL service needs to be defined in the UAF client manifest. This is done using the `<service>` tag for an Android AIDL service instead of the `<activity>` tag in Android Intents. Just as with Android intents, the manifest definition for the AIDL service uses an intent filter (note `org.fidoalliance.aidl.FIDO_OPERATION` versus `org.fidoalliance.intent.FIDO_OPERATION`) to identify itself as a FIDO UAF client to the relying party app:

```xml
<service android:name="foo">
    <intent-filter>
        <action android:name="org.fidoalliance.aidl.FIDO_OPERATION"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="application/fido.uaf_client+json"/>
    </intent-filter>
</service>
```

Once the relying party app chooses a UAF client from the list discovered by `PackageManager.queryIntentServices()`, the relying party app and the FIDO UAF client share the following AIDL interface to service UAF requests:

```java
package org.fidoalliance.aidl

oneway interface IUAFOperation
{
    void process(in Intent uafRequest, in IUAFResponseListener uafResponseListener);
}
```

For consistency, the Intents for the Android AIDL service are the same as defined in the Android Intent specification in the UAF standard. In `process()`, the `uafRequest` parameter is the Intent that would be passed to `startActivityForResult()`. The `uafResponseListener` parameter is a listener interface that receives the result. The following AIDL defines this interface:

```java
package org.fidoalliance.aidl

interface IUAFResponseListener
{
    void onResult(in Intent uafResponse);
}
```

In the listener, the `uafResponse` parameter is the Intent that would be passed to `onActivityResult`.

### 6.4 Security Considerations for Android Implementations

*This section is non-normative.*

Android applications may choose to implement the user-interactive portion of FIDO in at least two ways:

- by authoring an Android Activity using Android-native user interface components, or
- with an HTML-based experience by loading an Android WebView and injecting the UAF DOM APIs with `addJavaScriptInterface()`.

An application that chooses to inject the UAF interface into a WebView *MUST* follow all appropriate security considerations that apply to usage of the DOM APIs, and those that apply to user agent implementers.

In particular, the content of a WebView into which an API will be injected *MUST* be loaded only from trusted local content or over a secure channel as specified in [Transport Security Requirements](#) and must not contain insecure mixed-content.

Applications *SHOULD NOT* declare the `ACT AS WEB BROWSER` permission unless they need to act as the user's agent for an un-predetermined number of third party applications. Where an Android application has an explicit relationship with a relying party application(s), the preferred method of access control is for those applications to list the Android application's identity as a trusted facet. See the UAF Protocol Specification [UAFProtocol] for more information on application and facet identifiers.

To protect against a malicious application registering itself as a FIDO UAF Client, relying party applications can obtain the identity of the responding application, and utilize it in risk management decisions around the authentication or transaction events.

For example, a relying party might maintain a list of application identities known to belong to malware and refuse to accept operations completed with such clients, or a list of application identities of known-good clients that receive preferred risk-scoring.

**NOTE**

Android AIDL services use `Binder.getCallingUid()` instead of `Activity.getCallingActivity()` with Android Intents to identify the caller and obtain FacetID information.
**NOTE**

Relying party applications **SHOULD** implement the check on Android prior to 5 by using the package manager to verify that the FIDO Client indeed declared the `org.fidoalliance.uaf.permissions.FIDO_CLIENT` permission (see example below). Relying party applications **SHOULD NOT** use a "uses-permission" for `FIDO_CLIENT`.

**EXAMPLE 7**

```java
boolean checkFIDOClientPermission(String packageName) throws NameNotFoundException {
    for (String requestedPermission : getUserPermissions(packageName)) {
        if (requestedPermission.matches("org.fidoalliance.uaf.permissions.FIDO_CLIENT")) {
            return true;
        }
    }
    return false;
}
```

Relying party applications which use the AIDL service implementation of the UAF Client Intent API **MUST** use an explicit intent to bind to the AIDL service. Failing to do so may result in binding to an unexpected and possibly malicious service, because intent filter resolution depends on application installation order and intent filter priority. Android 5.0 and later will throw a `SecurityException` if an implicit intent is used, but earlier versions do not enforce this behavior.

7. iOS Custom URL API

*This section is normative.*

This section describes how an iOS relying party application can locate and communicate with a conforming FIDO UAF Client installed on the host device.

**NOTE**

Because of sandboxing and no true multitasking support, the iOS operating system offers very limited ways to do interprocess communication (IPC).

Any IPC solution for a FIDO UAF Client must be able to:

1. Identify the calling app in order to provide FacetID approval.
2. Allow transition to another app without user intervention

Currently the only IPC method on iOS that satisfies both of these requirements is custom URL handlers.

Custom URL handlers use the iOS operating system to handle URL requests from the sender, launch the receiving app, and then pass the request to the receiving app for processing. By enabling custom URL handlers for two different applications, it is possible to achieve bidirectional IPC between them--one custom URL handler to send data from app A to app B and another custom URL handler to send data from app B to app A.

Because iOS has no true multitasking, there must be an app transition to process each request and response. Too many app transitions can negatively affect the user experience, so relying party applications must carefully choose when it is necessary to query the FIDO UAF Client.

7.1 iOS-specific Definitions

7.1.1 X-Callback-URL Transport

When the relying party application communicates with the FIDO UAF Client, it sends a URL with the standard `x-callback-url` format (see `x-callback-url.com`):

**EXAMPLE 8**

```
FidoUAFClient1://x-callback-url/[UAFxRequestType]?x-success=[RelyingPartyURL]://x-callback-url/
[UAFxResponseType]&key=[SecretKey]&state=[STATE]
```
FidoUAFClient1 is the iOS custom URL scheme used by FIDO UAF Clients. As specified in the x-callback-url standard, version information for the transport layer is encoded in the URL scheme itself (in this case, FidoUAFClient1). This is so other applications can check for support for the 1.0 version by using the canOpenURL call.

[UAFxRequestType] is the type that should be used for request operations, which are described later in this document.

[RelyingPartyURL] is the URL that the relying party app has registered in order to receive the response. According to the x-callback-url standard, this is defined using the x-success parameter.

[UAFxResponseType] is the type that should be used for response operations, which are described later in this document.

[SecretKey] is a base64url-encoded, without padding, random key generated for each request by the calling application.

The response from the FIDO UAF Client will be encrypted with this key in order to prevent rogue applications from obtaining information by spoofing the return URL.

[STATE] is data that can be used to match the request with the response.

Finally [Base64URLEncodedJSON] contains the message to be sent to the FIDO UAF Client.

Items are stored in JSON format and then base64url-encoded without padding.

For FIDO UAF Clients, the custom URL scheme handler entrypoint is the openURL() function:

Objective-C

```objective-c
(BOOL)application:(UIApplication *)application openURL:(NSURL *)url sourceApplication:(NSString *)sourceApplication annotation:(id)annotation
```

SWIFT

```swift
func application(_ application: UIApplication, open url: URL, sourceApplication: String?, annotation: Any) -> Bool { ...
}
```

Here, the URL above is received via the url parameter. For security considerations, the sourceApplication parameter contains the iOS bundle ID of the relying party application. This bundle ID MUST be used to verify the application FacetID.

Conversely, when the FIDO UAF Client responds to the request, it sends the following URL back in standard x-callback-url format:

```swift
[RelyingPartyURL]:/x-callback-url/
[UAFxResponseType]? state=[STATE]&
json=[Base64URLEncodedJWE]
```

The parameters in the response are similar to those of the request, except that the [Base64URLEncodedEncryptedJSON] parameter is encrypted with the public key before being base64url-encoded without padding. [STATE] is the same STATE as was sent in the request—it is echoed back to the sender to verify the matched response.

In the relying party application's openURL() handler, the url parameter will be the URL listed above and the sourceApplication parameter will be the iOS bundle ID for the FIDO client application.

### 7.1.2 Secret Key Generation

A new secret encryption key MUST be generated by the calling application every time it sends a request to FIDO UAF Client. The FIDO UAF Client MUST then use this key to encrypt the response message before responding to the caller.

**JSON Web Encryption (JWE)**, JSON Serialization (JWE Section 7.2) format MUST be used to represent the encrypted response message.

The encryption algorithm is that specified in "A128CBC-HS256" where the JWE "Key Management Mode" employed is "Direct Encryption" and the JWE "Content Encryption Key (CEK)" is the secret key generated by the calling application and passed to the FIDO UAF Client in the key parameter of the request.
7.1.3 Origin

iOS applications requesting services from the FIDO Client can do so under their own identity, or they can act as the user's agent by explicitly declaring an RFC6454 [RFC6454] serialization of the remote server's origin when invoking the FIDO UAF Client.

An application that is operating on behalf of a single entity must not set an explicit origin. Omitting an explicit origin will cause the FIDO UAF Client to determine the caller's identity as "ios:bundle-id". The FIDO UAF Client will then compare this with the list of authorized application facets for the target AppID and proceed if it is listed as trusted.

See the UAF Protocol Specification [UAFProtocol] for more information on application and facet identifiers.

If the application is explicitly intended to operate as the user's agent in the context of an arbitrary number of remote applications (as when implementing a full web browser) it may set origin to the RFC6454 [RFC6454] Unicode serialization of the remote application's Origin. The application must satisfy the necessary conditions described in Transport Security Requirements for authenticating the remote server before setting origin.

7.1.4 channelBindings

This section is non-normative.

In the DOM API, the browser or browser plugin is responsible for supplying any available channel binding information to the FIDO Client, but an iOS application, as the direct owner of the transport channel, must provide this information itself.

The channelBindings data structure is `Map<String,String>` with the keys as defined for the `ChannelBinding` structure in the FIDO UAF Protocol Specification. [UAFProtocol]

The use of channel bindings for TLS helps assure the server that the channel over which UAF protocol messages are transported is the same channel the legitimate client is using and that messages have not been forwarded through a malicious party. UAF defines support for the `tls-unique` and `tls-server-end-point` bindings from [RFC5929], as well as server certificate and `ChannelID` bindings. The client should supply all channel binding information available to it.

Missing or invalid channel binding information may cause a relying party server to reject a transaction.

7.1.5 UAFxType

This value describes the type of operation for the `x-callback-url` operations implementing the iOS API.

```webidl
enum UAFxType {
    "DISCOVER",
    "DISCOVER_RESULT",
    "CHECK_POLICY",
    "CHECK_POLICY_RESULT",
    "UAF_OPERATION",
    "UAF_OPERATION_RESULT",
    "UAF_OPERATION_COMPLETION_STATUS"
};
```

### Enumeration description

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISCOVER</td>
<td>Discovery</td>
</tr>
<tr>
<td>DISCOVER_RESULT</td>
<td>Discovery results</td>
</tr>
<tr>
<td>CHECK_POLICY</td>
<td>Perform a no-op check if a message could be processed.</td>
</tr>
<tr>
<td>CHECK_POLICY_RESULT</td>
<td>Check Policy results.</td>
</tr>
<tr>
<td>UAF_OPERATION</td>
<td>The UAF message operation type (for example Registration).</td>
</tr>
<tr>
<td>UAF_OPERATION_RESULT</td>
<td>UAF Operation results.</td>
</tr>
<tr>
<td>UAF_OPERATION_COMPLETION_STATUS</td>
<td>Inform the FIDO UAF Client of the completion status of a UAF operation (such as Registration).</td>
</tr>
</tbody>
</table>

7.2 JSON Values

The specifics of the UAFxType operation are carried by various JSON values encoded in the `json x-callback-url` parameter.
The following table shows what JSON values are expected, depending on the value of the `UAFxType` x-callback-url operation:

<table>
<thead>
<tr>
<th>UAFxType operation</th>
<th>discoveryData</th>
<th>errorCode</th>
<th>message</th>
<th>origin</th>
<th>channelBindings</th>
<th>responseCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DISCOVER&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;DISCOVER_RESULT&quot;</td>
<td>OPTIONAL</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;CHECK_POLICY&quot;</td>
<td>REQUIRED</td>
<td>OPTIONAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;CHECK_POLICY_RESULT&quot;</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;UAF_OPERATION&quot;</td>
<td>REQUIRED</td>
<td>OPTIONAL</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;UAF_OPERATION_RESULT&quot;</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;UAF_OPERATION_COMPLETION_STATUS*&quot;</td>
<td>REQUIRED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REQUIRED</td>
</tr>
</tbody>
</table>

7.2.1 DISCOVER

This operation invokes the FIDO UAF Client to discover the available authenticators and capabilities. The FIDO UAF Client generally will not show a user interface associated with the handling of this operation, but will simply return the resulting JSON structure.

The calling application cannot depend on this however, as the client MAY show a user interface for privacy purposes, allowing the user to choose whether and which authenticators to disclose to the calling application.

NOTE

iOS custom URL scheme handlers always require an application switch for every request and response, even if no user interface is displayed.

7.2.2 DISCOVER_RESULT

An operation with this type is returned by the FIDO UAF Client in response to receiving an x-callback-url operation of type `DISCOVER`.

If x-callback-url `errorCode` JSON value is `NO_ERROR`, this x-callback-url operation has a JSON value, `discoveryData`, containing a `String` representation of a `DiscoveryData` JSON dictionary listing the available authenticators and their capabilities.

7.2.3 CHECK_POLICY

This operation invokes the FIDO UAF Client to discover if the client would be able to process the supplied message, without prompting the user.

The related Action handling this operation SHOULD NOT show an interface to the user.

NOTE

iOS custom URL scheme handlers always require an application switch for every request and response, even if no UI is displayed.

This x-callback-url operation requires the following JSON values:

- message, containing a `String` representation of a `UAFMessage` representing the request message to test.
- origin, an `OPTIONAL` JSON value that allows a caller to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's
own identity.

7.2.4 CHECK_POLICY_RESULT

This operation is returned by the FIDO UAF Client in response to receiving a `CHECK_POLICY` x-callback-url operation. The x-callback-url JSON value `errorCode` containing an `ErrorCode` value indicating the specific error condition or `NO_ERROR` if the FIDO Client could process the message.

7.2.5 UAF_OPERATION

This operation invokes the FIDO UAF Client to process the supplied request message and return a result message ready for delivery to the FIDO UAF Server. The sender SHOULD assume that the FIDO UAF Client will display a UI to the user to handle this x-callback-url operation, e.g. prompting the user to complete their verification ceremony.

This x-callback-url operation requires the following JSON values:

- `message`, containing a `String` representation of a `UAMessage` representing the request message to process.
- `channelBindings`, containing a `String` representation of a JSON dictionary as defined by the `ChannelBinding` structure in the UAF Protocol Specification [UAFProtocol].
- `origin`, an OPTIONAL JSON value that allows a caller to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

7.2.6 UAF_OPERATION_RESULT

This x-callback-url operation is returned by the FIDO UAF Client in response to receiving a `UAF_OPERATION` x-callback-url operation. The x-callback-url JSON value `errorCode` containing an `ErrorCode` value indicating the specific error condition.

If x-callback-url JSON value `errorCode` is `NO_ERROR`, this x-callback-url operation has a JSON value, `message`, containing a `String` representation of a `UAMessage`, being the UAF protocol response message to be delivered to the FIDO Server.

7.2.7 UAF_OPERATION_COMPLETION_STATUS

This x-callback-url operation MUST be delivered to the FIDO UAF Client to indicate the completion status of a FIDO UAF message delivered to the remote server. This is especially important as, e.g. a new registration may be considered in a pending status until it is known the server accepted it.

7.3 Implementation Guidelines for iOS Implementations

Each iOS Custom URL based request results in a human-noticeable context switch between the App and FIDO UAF Client and vice versa. This will be most noticeable when invoking DISCOVER and CHECK_POLICY requests since typically these requests will be invoked automatically, without user's involvement. Such a context switch impacts the User Experience and therefore it's RECOMMENDED to avoid making these two requests and integrate FIDO without using them.

7.4 Security Considerations for iOS Implementations

This section is non-normative.

A security concern with custom URLs under iOS is that any app can register any custom URL. If multiple applications register the same custom URL, the behavior for handling the URL call in iOS is undefined.

On the FIDO UAF Client side, this issue with custom URL scheme handlers is solved by using the `sourceApplication` parameter which provides the bundle ID of the URL originator. This is effective as long as the device has not been jailbroken and as long as Apple has done due diligence vetting submissions to the app store for malware with faked bundle IDs. The `sourceApplication` parameter can be matched with the FacetID list to ensure that the calling app is approved to use the credentials for the relying party.

On the relying party app side, encryption is used to prevent a rogue app from spoofing the relying party app's response URL. The relying party app generates a random encryption key on every request and sends it to the FIDO client. The FIDO client then encrypts the response to this key. In this manner, only the relying party app can decrypt the response. Even in the event that malware is able to spoof the relying party app's URL and intercept the response, it would not be able to decode it.

To protect against potentially malicious applications registering themselves to handle the FIDO UAF Client custom URL scheme, relying party Applications can obtain the bundle-id of the responding app and utilize it in risk management decisions around the authentication or transaction events. For example, a relying party might maintain a list of bundle-ids known to belong to malware and refuse to accept operations completed with such clients, or a list of bundle-ids of known-good clients that receive preferred risk-scoring.
8. Transport Binding Profile

This section is normative.

This section describes general normative security requirements for how a client application transports FIDO UAF protocol messages, gives specific requirements for Transport Layer Security (TLS), and describes an interoperability profile for using HTTP over TLS [RFC2818] with the FIDO UAF protocol.

8.1 Transport Security Requirements

This section is non-normative.

The UAF protocol contains no inherent means of identifying a relying party server, or for end-to-end protection of UAF protocol messages. To perform a secure UAF protocol exchange, the following abstract requirements apply:

1. The client application must securely authenticate the server endpoint as authorized, from that client’s viewpoint, to represent the Web origin [RFC6454] (scheme:host:port tuple) reported to the FIDO UAF Client by the client application. Most typically this will be done by using TLS and verifying the server’s certificate is valid, asserts the correct DNS name, and chains up to a root trusted by the client platform. Clients MAY also utilize other means to authenticate a server, such as via a pre-provisioned certificate or key that is distributed with an application, or alternative network authentication protocols such as Kerberos [RFC4120].

2. The transport mechanism for UAF protocol messages must provide confidentiality for the message, to prevent disclosure of their contents to unauthorized third parties. These protections should be cryptographically bound to proof of the server's identity as described above.

3. The transport mechanism for UAF protocol messages must protect the integrity of the message from tampering by unauthorized third parties. These protections should be cryptographically bound to proof of the server's identity in as described above.

8.2 TLS Security Requirements

This section is non-normative.

If using HTTP over TLS ([RFC2246] [RFC4346], [RFC5246] or [TLS13draft02]) to transport an UAF protocol exchange, the following specific requirements apply:

1. If there are any TLS errors, whether "warning" or "fatal" or any other error level with the TLS connection, the HTTP client must terminate the connection without prompting the user. For example, this includes any errors found in certificate validity checking that HTTP clients employ, such as via TLS server identity checking [RFC6125], Certificate Revocation Lists (CRLs) [RFC5280], or via the Online Certificate Status Protocol (OCSP) [RFC2560].

2. Whenever comparisons are made between the presented TLS server identity (as presented during the TLS handshake, typically within the server certificate) and the intended source TLS server identity (e.g., as entered by a user, or embedded in a link), [RFC6125] server identity checking must be employed. The client must terminate the connection without prompting the user upon any error condition.

3. The TLS server certificate must either be provisioned explicitly out-of-band (e.g. packaged with an app as a "pinned certificate") or be trusted by chaining to a root included in the certificate store of the operating system or a major browser by virtue of being currently in compliance with their root store program requirements. The client must terminate the connection without user recourse if there are any error conditions when building the chain of trust.

4. The "anon" and "null" crypto suites are not allowed and insecure cryptographic algorithms in TLS (e.g. MD4, RC4, SHA1) should be avoided (see NIST SP800-131A [SP800-131A]).

5. The client and server should use the latest practicable TLS version.

6. The client should supply, and the server should verify whatever practicable channel binding information is available, including a ChannelID [ChannelID] public key, the tls-unique and tls-server-endpoint bindings [RFC5929], and TLS server certificate binding [UAFProtocol]. This information provides protection against certain classes of network attackers and the forwarding of protocol messages, and a server may reject a message that lacks or has channel binding data that does not verify correctly.

8.3 HTTPS Transport Interoperability Profile

This section is normative.

Conforming applications MAY support this profile.

Complex and highly-optimized applications utilizing UAF will often transport UAF protocol messages in-line with other application protocol messages. The profile defined here for transporting UAF protocol messages over HTTPS is intended to:

- Provide an interoperability profile to enable easier composition of client-side application libraries and server-side implementations for FIDO UAF-enabled products from different vendors.
- Provide detailed illustration of specific necessary security properties for the transport layer and HTTP interfaces, especially as they may interact with a browser-hosted application.
- This profile is also utilized in the examples that constitute the appendices of this document. This profile is OPTIONAL to implement. RFC 2119
key words are used in this section to indicate necessary security and other properties for implementations that intend to use this profile to interoperate [RFC2119].

NOTE
Certain FIDO UAF operations, in particular, transaction confirmation, will always require an application-specific implementation. This interoperability profile only provides a skeleton framework suitable for replacing username/password authentication.

8.3.1 Obtaining a UAF Request message

A UAF-enabled web application might typically deliver request messages as part of a response body containing other application content, e.g. in a script block as such:

EXAMPLE 13

```xml
...<script type="application/json">
  "initialRequest": { // initial request message here
    "lifetimeMillis": 60000; // hint: this initial request is valid for 60 seconds
  }
</script>...
```

However, request messages have a limited lifetime, and an installed application cannot be delivered with a request, so client applications generally need the ability to retrieve a fresh request.

When sending a request message over HTTPS with XMLHttpRequest [XHR] or another HTTP API:

1. The URI of the server endpoint, and how it is communicated to the client, is application-specific.
2. The client MUST set the HTTP method to POST. [RFC7231]
3. The client SHOULD set the HTTP "Content-Type" header to "application/fido+uaf; charset=utf-8". [RFC7231]
4. The client SHOULD include "application/fido+uaf" as a media type in the HTTP "Accept" header [RFC7231]. Conforming servers MUST accept "application/fido+uaf" as media type.
5. The client MAY need to supply additional headers, such as a HTTP Cookie [RFC6265], to demonstrate, in an application-specific manner, their authorization to perform a request.
6. The entire POST body MUST consist entirely of a JSON [ECMA-404] structure described by the `GetUAFRequest` dictionary.
7. The server's response SHOULD set the HTTP "Content-Type" to "application/fido+uaf; charset=utf-8".
8. The client SHOULD decode the response byte string as UTF-8 with error handling. [HTML5]
9. The decoded body of the response MUST consist entirely of a JSON structure described by the `ReturnUAFRequest` interface.

8.3.2 Operation enum

Describes the operation type of a FIDO UAF message or request for a message.

```webidl
enum Operation {
  "Reg",
  "Auth",
  "Dereg"
};
```

**Enumeration description**

<table>
<thead>
<tr>
<th>Reg</th>
<th>Registration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auth</td>
<td>Authentication or Transaction Confirmation</td>
</tr>
<tr>
<td>Dereg</td>
<td>Deregistration</td>
</tr>
</tbody>
</table>

8.3.3 GetUAFRequest dictionary

```webidl```
dictionary GetUAFRequest {
  Operation op;  
  DOMString previousRequest;  
  DOMString context;  
};

8.3.3.1 Dictionary GetUAFRequest Members

`op` of type `Operation`
The type of the UAF request message desired. Allowable string values are defined by the `Operation` enum. This field is `OPTIONAL` but `MUST` be set if the operation is not known to the server through other context, e.g. an operation-specific URL endpoint.

`previousRequest` of type `DOMString`
If the application is requesting a new UAF request message because a previous one has expired, this `OPTIONAL` key can include the previous one to assist the server in locating any state that should be re-associated with a new request message, should one be issued.

`context` of type `DOMString`
Any additional contextual information that may be useful or necessary for the server to generate the correct request message. This key is `OPTIONAL` and the format and nature of this data is application-specific.

8.3.4 ReturnUAFRequest dictionary

```WebIDL
dictionary ReturnUAFRequest {
  required unsigned long statusCode;  
  DOMString uafRequest;  
  Operation op;  
  long lifetimeMillis;  
};
```

8.3.4.1 Dictionary ReturnUAFRequest Members

`statusCode` of type `required unsigned long`
The UAF Status Code for the operation (see section 3.1 UAF Status Codes).

`uafRequest` of type `DOMString`
The new UAF Request Message, `OPTIONAL`, if the server decided to issue one.

`op` of type `Operation`
An `OPTIONAL` hint to the client of the operation type of the message, useful if the server might return a different type than was requested. For example, a server might return a deregister message if an authentication request referred to a key it no longer considers valid. Allowable string values are defined by the `Operation` enum.

`lifetimeMillis` of type `long`
If the server returned a `uafRequest`, this is an `OPTIONAL` hint informing the client application of the lifetime of the message in milliseconds.

8.3.5 SendUAFResponse dictionary

```WebIDL
dictionary SendUAFResponse {
  required DOMString uafResponse;  
  DOMString context;  
};
```

8.3.5.1 Dictionary SendUAFResponse Members

`uafResponse` of type `required DOMString`
The UAF Response Message. It `MUST` be set to `UAFMessage.uafProtocolMessage` returned by FIDO UAF Client.

`context` of type `DOMString`
Any additional contextual information that `MAY` be useful or necessary for the server to process the response message. This key is `OPTIONAL` and the format and nature of this data is application-specific.

8.3.6 Delivering a UAF Response
Although it is not the only pattern possible, an asynchronous HTTP request is a useful way of delivering a UAF Response to the remote server for either web applications or standalone applications.

When delivering a response message over HTTPS with XMLHttpRequest [XHR] or another API:

1. The URI of the server endpoint and how it is communicated to the client is application-specific.
2. The client MUST set the HTTP method to POST. [RFC7231]
3. The client MUST set the HTTP "Content-Type" header to "application/fido+uaf; charset=utf-8". [RFC7231]
4. The client SHOULD include "application/fido+uaf" as a media type in the HTTP "Accept" header. [RFC7231]
5. The client MAY need to supply additional headers, such as a HTTP Cookie [RFC6265], to demonstrate, in an application-specific manner, their authorization to perform an operation.
6. The entire POST body MUST consist entirely of a JSON [ECMA-404] structure described by the SendUAFResponse.
7. The server's response SHOULD set the "Content-Type" to "application/fido+uaf; charset=utf-8" and the body of the response MUST consist entirely of a JSON structure described by the ServerResponse interface.

### 8.3.7 ServerResponse Interface

The ServerResponse interface represents the completion status and additional application-specific additional data that results from successful processing of a Register, Authenticate, or Transaction Confirmation operation. This message is not formally part of the UAF protocol, but the statusCode should be posted to the FIDO UAF Client, for housekeeping, using the notifyUAFResult() operation.

```webidl
interface ServerResponse {
  [Optional]
  readonly attribute int statusCode;
  [Optional]
  readonly attribute DOMString description;
  [Optional]
  readonly attribute Token[] additionalTokens;
  [Optional]
  readonly attribute DOMString location;
  [Optional]
  readonly attribute DOMString postData;
  [Optional]
  readonly attribute DOMString newUAFRequest;
};
```

#### 8.3.7.1 Attributes

- **statusCode** of type int, readonly
  The FIDO UAF response status code. Note that this status code describes the result of processing the tunneled UAF operation, not the status code for the outer HTTP transport.

- **description** of type DOMString, readonly
  A detailed message describing the status code or providing additional information to the user.

- **additionalTokens** of type array of Token, readonly
  This key contains new authentication or authorization token(s) for the client that are not natively handled by the HTTP transport. Tokens SHOULD be processed prior to processing of `location`.

- **location** of type DOMString, readonly
  If present, indicates to the client web application that it should navigate the Document context to the URI contained on this field after processing any tokens.

- **postData** of type DOMString, readonly
  If present in combination with `location`, indicates that the client should POST the contents to the specified location after processing any tokens.

- **newUAFRequest** of type DOMString, readonly
  The server may use this to return a new UAF protocol message. This might be used to supply a fresh request to retry an operation in response to a transient failure, to request additional confirmation for a transaction, or to send a deregistration message in response to a permanent failure.

### 8.3.8 Token interface

**NOTE**

The UAF Server is not responsible for creating additional tokens returned as part of a UAF response. Such tokens exist to provide a
means for the relying party application to update the authentication/authorization state of the client in response to a successful UAF operation. For example, these fields could be used to allow UAF to serve as the initial authentication leg of a federation protocol, but the scope and details of any such federation are outside of the scope of UAF.

```webidl
interface Token {
    readonly attribute TokenType type;
    readonly attribute DOMString value;
};
```

8.3.8.1 Attributes

- **type** of type `TokenType`, readonly
  The type of the additional authentication / authorization token.

- **value** of type `DOMString`, readonly
  The string value of the additional authentication / authorization token.

8.3.9 TokenType enum

```webidl
enum TokenType {
    "HTTP_COOKIE",
    "OAUTH",
    "OAUTH2",
    "SAML1_1",
    "SAML2",
    "JWT",
    "OPENID_CONNECT"
};
```

<table>
<thead>
<tr>
<th>Enumeration description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HTTP_COOKIE</strong></td>
</tr>
<tr>
<td>If the user agent is a standard web browser or other HTTP native client with a cookie store, this TokenType SHOULD NOT be used. Cookies should be set directly with the Set-Cookie HTTP header for processing by the user agent. For non-HTTP or non-browser contexts this indicates a token intended to be set as an HTTP cookie. [RFC6265] For example, a native VPN client that authenticates with UAF might use this TokenType to automatically add a cookie to the browser cookie jar.</td>
</tr>
<tr>
<td><strong>OAUTH</strong></td>
</tr>
<tr>
<td>Indicates that the token is of type OAUTH. [RFC5849].</td>
</tr>
<tr>
<td><strong>OAUTH2</strong></td>
</tr>
<tr>
<td>Indicates that the token is of type OAUTH2. [RFC6749].</td>
</tr>
<tr>
<td><strong>SAML1_1</strong></td>
</tr>
<tr>
<td>Indicates that the token is of type SAML 1.1. [SAML11].</td>
</tr>
<tr>
<td><strong>SAML2</strong></td>
</tr>
<tr>
<td>Indicates that the token is of type SAML 2.0. [SAML2-CORE]</td>
</tr>
<tr>
<td><strong>JWT</strong></td>
</tr>
<tr>
<td>Indicates that the token is of type JSON Web Token (JWT). [JWT]</td>
</tr>
<tr>
<td><strong>OPENID_CONNECT</strong></td>
</tr>
<tr>
<td>Indicates that the token is an OpenID Connect &quot;id_token&quot;. [OpenIDConnect]</td>
</tr>
</tbody>
</table>

8.3.10 Security Considerations

*This section is non-normative.*

It is important that the client set, and the server require, the method be POST and the “Content-Type” HTTP header be the correct values. Because the response body is valid ECMAScript, to protect against unauthorized cross-origin access, a server must not respond to the type of request that can be generated by a script tag, e.g. `<script src="https://example.com/fido/uaf/getRequest">`. The request a user agent generates with this kind of embedding cannot set custom headers.

Likewise, by requiring a custom “Content-Type” header, cross-origin requests cannot be made with an XMLHttpRequest [XHR] without triggering a CORS preflight access check. [CORS]

As FIDO UAF messages are only valid when used same-origin, servers should not supply an “Access-Control-Allow-Origin” [CORS] header with responses that would allow them to be read by non-same-origin content.

To protect from some classes of cross-origin, browser-based, distributed denial-of-service attacks, request endpoints should ignore, without performing additional processing, all requests with an “Access-Control-Request-Method” [CORS] HTTP header or an incorrect “Content-Type” HTTP header.

If a server chooses to respond to requests made with the GET method and without the custom “Content-Type” header, it should apply a prefix string such as "while(1);" or "&amp;&amp;BEGIN_UAF_RESPONSE&amp;&amp;" to the body of all replies and so prevent their being read through cross-origin `<script>`
tag embedding. Legitimate same-origin callers will need to (and alone be able to) strip this prefix string before parsing the JSON content.

A. References

A.1 Normative references

[AndroidAppManifest]

[ChannelID]

[DOM]
Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMA-262]
The ECMA Script Language Specification. URL: https://tc39.es/ecma262/

[ECMA-404]

[FIDO Glossary]

[FIDO Metadata Statement]

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

[HTML5]
I. Hickson; R. Berjon; S. Faulkner; T. Leithead; E. D. Navara; E. O'Connor; S. Pfeiffer. HTML5: A vocabulary and associated APIs for HTML and XHTML. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/

[JWT]

[OpenID Connect]
OpenID Connect. Work in Progress. URL: http://openid.net/connect/

[PNG]

[RFC 2119]

[RFC 2397]

[RFC 2818]

[RFC 4648]

[SAML 1.1]
A.2 Informative references

[ANDROID]
The Android™ Operating System. URL: http://developer.android.com/

[Android5Changes]
Android 5.0 Behavior Changes. Work in progress. URL: http://developer.android.com/about/versions/android-5.0-changes.html

[CORS]

[RFC2045]

[RFC2246]

[RFC2560]

[RFC4120]

[RFC4346]

[RFC5246]

[RFC5280]

[SOP]

[SP800-131A]

[TLS13draft02]

[UAFASM]
D. Baghdasaryan; J. Kemp; R. Lindemann; B. Hill; R. Sasson. FIDO UAF Authenticator-Specific Module API. Review Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-asm-api-v1.2-ps-20201020.html

[XHRL]

[XHR]
Anne van Kesteren. XMLHttpRequest Standard. Living Standard. URL: https://xhr.spec.whatwg.org/

[webmessaging]
Ian Hickson. HTML5 Web Messaging. 19 May 2015. W3C Recommendation. URL: https://www.w3.org/TR/webmessaging/
Abstract

The FIDO UAF strong authentication framework enables online services and websites, whether on the open Internet or within enterprises, to transparently leverage native security features of end-user computing devices for strong user authentication and to reduce the problems associated with creating and remembering many online credentials. The FIDO UAF Reference Architecture describes the components, protocols, and interfaces that make up the FIDO UAF strong authentication ecosystem.

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. Introduction
1. Introduction

This section is non-normative.

This document describes the FIDO Universal Authentication Framework (UAF) Reference Architecture. The target audience for this document is decision makers and technical architects who need a high-level understanding of the FIDO UAF strong authentication solution and its relationship to other relevant industry standards.

The FIDO UAF specifications are as follows:

- FIDO UAF Protocol
- FIDO UAF Application API and Transport Binding
- FIDO UAF Authenticator Commands
- FIDO UAF Authenticator-Specific Module API
- FIDO UAF Registry of Predefined Values
- FIDO UAF APDU

The following additional FIDO documents provide important information relevant to the UAF specifications:

- FIDO AppID and Facets Specification
- FIDO Metadata Statements
- FIDO Metadata Service
- FIDO Registry of Predefined Values
- FIDO ECDAA Algorithm
- FIDO Security Reference
- FIDO Glossary

These documents may all be found on the FIDO Alliance website at [http://fidoalliance.org/specifications/download/](http://fidoalliance.org/specifications/download/)

1.1 Background

This section is non-normative.

The FIDO Alliance mission is to change the nature of online strong authentication by:

- Developing technical specifications defining open, scalable, interoperable mechanisms that supplant reliance on passwords to securely

...
authenticate users of online services.

- Operating industry programs to help ensure successful worldwide adoption of the specifications.
- Submitting mature technical specifications to recognized standards development organization(s) for formal standardization.

The core ideas driving the FIDO Alliance’s efforts are 1) ease of use, 2) privacy and security, and 3) standardization. The primary objective is to enable online services and websites, whether on the open Internet or within enterprises, to leverage native security features of end-user computing devices for strong user authentication and to reduce the problems associated with creating and remembering many online credentials.

There are two key protocols included in the FIDO architecture that cater to two basic options for user experience when dealing with Internet services. The two protocols share many of underpinnings but are tuned to the specific intended use cases.

**Universal Authentication Framework (UAF) Protocol**

The UAF protocol allows online services to offer password-less and multi-factor security. The user registers their device to the online service by selecting a local authentication mechanism such as swiping a finger, looking at the camera, speaking into the mic, entering a PIN, etc. The UAF protocol allows the service to select which mechanisms are presented to the user.

Once registered, the user simply repeats the local authentication action whenever they need to authenticate to the service. The user no longer needs to enter their password when authenticating from that device. UAF also allows experiences that combine multiple authentication mechanisms such as fingerprint + PIN.

This document that you are reading describes the UAF reference architecture.

**Universal 2nd Factor (U2F) Protocol**

The U2F protocol allows online services to augment the security of their existing password infrastructure by adding a strong second factor to user login. The user logs in with a username and password as before. The service can also prompt the user to present a second factor device at any time it chooses. The strong second factor allows the service to simplify its passwords (e.g. 4-digit PIN) without compromising security.

During registration and authentication, the user presents the second factor by simply pressing a button on a USB device or tapping over NFC. The user can use their FIDO U2F device across all online services that support the protocol leveraging built-in support in web browsers.

Please refer to the FIDO website for an overview and documentation set focused on the U2F protocol.

1.2 FIDO UAF Documentation

*This section is non-normative.*

To understand the FIDO UAF protocol, it is recommended that new audiences start by reading this architecture overview document and become familiar with the technical terminology used in the specifications (the glossary). Then they should proceed to the individual UAF documents in the recommended order listed below.

- **FIDO UAF Overview**: This document. Provides an introduction to the FIDO UAF architecture, protocols, and specifications.
- **FIDO Technical Glossary**: Defines the technical terms and phrases used in FIDO Alliance specifications and documents.
- **Universal Authentication Framework (UAF)**
  - **UAF Protocol Specification**: Message formats and processing rules for all UAF protocol messages.
  - **UAF Application API and Transport Binding Specification**: APIs and interoperability profile for client applications to utilize FIDO UAF.
  - **UAF Authenticator Commands**: Low-level functionality that UAF Authenticators should implement to support the UAF protocol.
  - **UAF Authenticator-specific Module API**: Authenticator-specific Module API provided by an ASM to the FIDO client.
  - **UAF Registry of Predefined Values**: defines all the strings and constants reserved by UAF protocols.
  - **UAF APDU**: defines a mapping of FIDO UAF Authenticator commands to Application Protocol Data Units (APDUs).
- **FIDO AppID and Facet Specification**: Scope of user credentials and how a trusted computing base which supports application isolation may make access control decisions about which keys can be used by which applications and web origins.
- **FIDO Metadata Statements**: Information describing form factors, characteristics, and capabilities of FIDO Authenticators used to inform interactions with and make policy decisions about the authenticators.
- **FIDO Metadata Service**: Baseline method for relying parties to access the latest Metadata statements.
- **FIDO ECDAA Algorithm**: Defines the direct anonymous attestation algorithm for FIDO Authenticators.
- **FIDO Registry of Predefined Values**: defines all the strings and constants reserved by FIDO protocols with relevance to multiple FIDO protocol families.
- **FIDO Security Reference**: Provides an analysis of FIDO security based on detailed analysis of security threats pertinent to the FIDO protocols based on its goals, assumptions, and inherent security measures.

The remainder of this Overview section of the reference architecture document introduces the key drivers, goals, and principles which inform the design of FIDO UAF.
Following the Overview, this document describes:

- A high-level look at the components, protocols, and APIs defined by the architecture
- The main FIDO UAF use cases and the protocol message flows required to implement them.
- The relationship of the FIDO protocols to other relevant industry standards.

1.3 FIDO UAF Goals

This section is non-normative.

In order to address today's strong authentication issues and develop a smoothly-functioning low-friction ecosystem, a comprehensive, open, multi-vendor solution architecture is needed that encompasses:

- User devices, whether personally acquired, enterprise-issued, or enterprise BYOD, and the device's potential operating environment, e.g. home, office, in the field, etc.
- Authenticators
- Relying party applications and their deployment environments
- Meeting the needs of both end users and Relying Parties
- Strong focus on both browser- and native-app-based end-user experience

This solution architecture must feature:

- FIDO UAF Authenticator discovery, attestation, and provisioning
- Cross-platform strong authentication protocols leveraging FIDO UAF Authenticators
- A uniform cross-platform authenticator API
- Simple mechanisms for Relying Party integration

The FIDO Alliance envisions an open, multi-vendor, cross-platform reference architecture with these goals:

- **Support strong, multi-factor authentication**: Protect Relying Parties against unauthorized access by supporting end user authentication using two or more strong authentication factors (“something you know”, “something you have”, “something you are”).
- **Build on, but not require, existing device capabilities**: Facilitate user authentication using built-in platform authenticators or capabilities (fingerprint sensors, cameras, microphones, embedded TPM hardware), but do not preclude the use of discrete additional authenticators.
- **Enable selection of the authentication mechanism**: Facilitate Relying Party and user choice amongst supported authentication mechanisms in order to mitigate risks for their particular use cases.
- **Simplify integration of new authentication capabilities**: Enable organizations to expand their use of strong authentication to address new use cases, leverage new device's capabilities, and address new risks with a single authentication approach.
- **Incorporate extensibility for future refinements and innovations**: Design extensible protocols and APIs in order to support the future emergence of additional types of authenticators, authentication methods, and authentication protocols, while maintaining reasonable backwards compatibility.
- **Leverage existing open standards where possible, openly innovate and extend where not**: An open, standardized, royalty-free specification suite will enable the establishment of a virtuous-circle ecosystem, and decrease the risk, complexity, and costs associated with deploying strong authentication. Existing gaps -- notably uniform authenticator provisioning and attestation, a uniform cross-platform authenticator API, as well as a flexible strong authentication challenge-response protocol leveraging the user's authenticators will be addressed.
- **Complement existing single sign-on, federation initiatives**: While industry initiatives (such as OpenID, OAuth, SAML, and others) have created mechanisms to reduce the reliance on passwords through single sign-on or federation technologies, they do not directly address the need for an initial strong authentication interaction between end users and Relying Parties.
- **Preserve the privacy of the end user**: Provide the user control over the sharing of device capability information with Relying Parties, and mitigate the potential for collusion amongst Relying Parties.
- **Unify end-User Experience**: Create easy, fun, and unified end-user experiences across all platforms and across similar Authenticators.

2. FIDO UAF High-Level Architecture

This section is non-normative.

The FIDO UAF Architecture is designed to meet the FIDO goals and yield the desired ecosystem benefits. It accomplishes this by filling in the status-quo's gaps using standardized protocols and APIs.

The following diagram summarizes the reference architecture and how its components relate to typical user devices and Relying Parties.

The FIDO-specific components of the reference architecture are described below.
2.1 FIDO UAF Client

A FIDO UAF Client implements the client side of the FIDO UAF protocols, and is responsible for:

- Interacting with specific FIDO UAF Authenticators using the FIDO UAF Authenticator Abstraction layer via the FIDO UAF Authenticator API.
- Interacting with a user agent on the device (e.g. a mobile app, browser) using user agent-specific interfaces to communicate with the FIDO UAF Server. For example, a FIDO-specific browser plugin would use existing browser plugin interfaces or a mobile app may use a FIDO-specific SDK. The user agent is then responsible for communicating FIDO UAF messages to a FIDO UAF Server at a Relying Party.

The FIDO UAF architecture ensures that FIDO client software can be implemented across a range of system types, operating systems, and Web browsers. While FIDO client software is typically platform-specific, the interactions between the components should ensure a consistent user experience from platform to platform.

2.2 FIDO UAF Server

A FIDO UAF server implements the server side of the FIDO UAF protocols and is responsible for:

- Interacting with the Relying Party web server to communicate FIDO UAF protocol messages to a FIDO UAF Client via a device user agent.
- Validating FIDO UAF authenticator attestations against the configured authenticator metadata to ensure only trusted authenticators are registered for use.
- Managing the association of registered FIDO UAF Authenticators to user accounts at the Relying Party.
- Evaluating user authentication and transaction confirmation responses to determine their validity.

The FIDO UAF server is conceived as being deployable as an on-premise server by Relying Parties or as being outsourced to a FIDO-enabled third-party service provider.

2.3 FIDO UAF Protocols

The FIDO UAF protocols carry FIDO UAF messages between user devices and Relying Parties. There are protocol messages addressing:

- Authenticator Registration: The FIDO UAF registration protocol enables Relying Parties to:
  - Discover the FIDO UAF Authenticators available on a user's system or device. Discovery will convey FIDO UAF Authenticator attributes to the Relying Party thus enabling policy decisions and enforcement to take place.
  - Verify attestation assertions made by the FIDO UAF Authenticators to ensure the authenticator is authentic and trusted. Verification occurs using the attestation public key certificates distributed via authenticator metadata.
  - Register the authenticator and associate it with the user's account at the Relying Party. Once an authenticator attestation has been validated, the Relying Party can provide a unique secure identifier that is specific to the Relying Party and the FIDO UAF Authenticator. This identifier can be used in future interactions between the pair {RP, Authenticator} and is not known to any other devices.

- User Authentication: Authentication is typically based on cryptographic challenge-response authentication protocols and will facilitate user choice regarding which FIDO UAF Authenticators are employed in an authentication event.

- Secure Transaction Confirmation: If the user authenticator includes the capability to do so, a Relying Party can present the user with a
secure message for confirmation. The message content is determined by the Relying Party and could be used in a variety of contexts such as confirming a financial transaction, a user agreement, or releasing patient records.

- Authenticator Deregistration: Deregistration is typically required when the user account is removed at the Relying Party. The Relying Party can trigger the deregistration by requesting the Authenticator to delete the associated UAF credential with the user account.

2.4 FIDO UAF Authenticator Abstraction Layer

The FIDO UAF Authenticator Abstraction Layer provides a uniform API to FIDO Clients enabling the use of authenticator-based cryptographic services for FIDO-supported operations. It provides a uniform lower-layer "authenticator plugin" API facilitating the deployment of multi-vendor FIDO UAF Authenticators and their requisite drivers.

2.5 FIDO UAF Authenticator

A FIDO UAF Authenticator is a secure entity, connected to or housed within FIDO user devices, that can create key material associated to a Relying Party. The key can then be used to participate in FIDO UAF strong authentication protocols. For example, the FIDO UAF Authenticator can provide a response to a cryptographic challenge using the key material thus authenticating itself to the Relying Party.

In order to meet the goal of simplifying integration of trusted authentication capabilities, a FIDO UAF Authenticator will be able to attest to its particular type (e.g., biometric) and capabilities (e.g., supported crypto algorithms), as well as to its provenance. This provides a Relying Party with a high degree of confidence that the user being authenticated is indeed the user that originally registered with the site.

2.6 FIDO UAF Authenticator Metadata Validation

In the FIDO UAF context, attestation is how Authenticators make claims to a Relying Party during registration that the keys they generate, and/or certain measurements they report, originate from genuine devices with certified characteristics. An attestation signature, carried in a FIDO UAF registration protocol message is validated by the FIDO UAF Server. FIDO UAF Authenticators are created with attestation private keys used to create the signatures and the FIDO UAF Server validates the signature using that authenticator's attestation public key certificate located in the authenticator metadata. The metadata holding attestation certificates is shared with FIDO UAF Servers out of band.

3. FIDO UAF Usage Scenarios and Protocol Message Flows

This section is non-normative.

The FIDO UAF ecosystem supports the use cases briefly described in this section.

3.1 FIDO UAF Authenticator Acquisition and User Enrollment

It is expected that users will acquire FIDO UAF Authenticators in various ways: they purchase a new system that comes with embedded FIDO UAF Authenticator capability; they purchase a device with an embedded FIDO UAF Authenticator, or they are given a FIDO Authenticator by their employer or some other institution such as their bank.

After receiving a FIDO UAF Authenticator, the user must go through an authenticator-specific enrollment process, which is outside the scope of the FIDO UAF protocols. For example, in the case of a fingerprint sensing authenticator, the user must register their fingerprint(s) with the authenticator. Once enrollment is complete, the FIDO UAF Authenticator is ready for registration with FIDO UAF enabled online services and websites.

3.2 Authenticator Registration

Given the FIDO UAF architecture, a Relying Party is able to transparently detect when a user begins interacting with them while possessing an initialized FIDO UAF Authenticator. In this initial introduction phase, the website will prompt the user regarding any detected FIDO UAF Authenticator(s), giving the user options regarding registering it with the website or not.
3.3 Authentication

Following registration, the FIDO UAF Authenticator will be subsequently employed whenever the user authenticates with the website (and the authenticator is present). The website can implement various fallback strategies for those occasions when the FIDO Authenticator is not present. These might range from allowing conventional login with diminished privileges to disallowing login.
This overall scenario will vary slightly depending upon the type of FIDO UAF Authenticator being employed. Some authenticators may sample biometric data such as a face image, fingerprint, or voice print. Others will require a PIN or local authenticator-specific passphrase entry. Still others may simply be a hardware bearer authenticator. Note that it is permissible for a FIDO Client to interact with external services as part of the authentication of the user to the authenticator as long as the FIDO Privacy Principles are adhered to.

3.4 Step-up Authentication

Step-up authentication is an embellishment to the basic website login use case. Often, online services and websites allow unauthenticated, and/or only nominally authenticated use -- for informational browsing, for example. However, once users request more valuable interactions, such as entering a members-only area, the website may request further higher-assurance authentication. This could proceed in several steps if the user then wishes to purchase something, with higher-assurance steps with increasing transaction value.

FIDO UAF will smoothly facilitate this interaction style since the website will be able to discover which FIDO UAF Authenticators are available on FIDO-wielding users' systems, and select incorporation of the appropriate one(s) in any particular authentication interaction. Thus online services and websites will be able to dynamically tailor initial, as well as step-up authentication interactions according to what the user is able to wield and the needed inputs to website's risk analysis engine given the interaction the user has requested.

3.5 Transaction Confirmation

There are various innovative use cases possible given FIDO UAF-enabled Relying Parties with end-users wielding FIDO UAF Authenticators. Website login and step-up authentication are relatively simple examples. A somewhat more advanced use case is secure transaction processing.
Imagine a situation in which a Relying Party wants the end-user to confirm a transaction (e.g. financial operation, privileged operation, etc) so that any tampering of a transaction message during its route to the end device display and back can be detected. FIDO architecture has a concept of "secure transaction" which provides this capability. Basically if a FIDO UAF Authenticator has a transaction confirmation display capability, FIDO UAF architecture makes sure that the system supports What You See is What You Sign mode (WYSIWYS). A number of different use cases can derive from this capability -- mainly related to authorization of transactions (send money, perform a context specific privileged action, confirmation of email/address, etc).

3.6 Authenticator Deregistration

There are some situations where a Relying Party may need to remove the UAF credentials associated with a specific user account in FIDO Authenticator. For example, the user’s account is cancelled or deleted, the user’s FIDO Authenticator is lost or stolen, etc. In these situations, the RP may request the FIDO Authenticator to delete authentication keys that are bound to user account.
3.7 Adoption of New Types of FIDO UAF Authenticators

Authenticators will evolve and new types are expected to appear in the future. Their adoption on the part of both users and Relying Parties is facilitated by the FIDO architecture. In order to support a new FIDO UAF Authenticator type, Relying Parties need only to add a new entry to their configuration describing the new authenticator, along with its FIDO Attestation Certificate. Afterwards, end users will be able to use the new FIDO UAF Authenticator type with those Relying Parties.

4. Privacy Considerations

This section is non-normative.

User privacy is fundamental to FIDO and is supported in UAF by design. Some of the key privacy-aware design elements are summarized here:

- A UAF device does not have a global identifier visible across relying parties and does not have a global identifier within a particular relying party. If for example, a person loses their UAF device, someone finding it cannot “point it at a relying party” and discover if the original user had any accounts with that relying party. Similarly, if two users share a UAF device and each has registered their account with the same relying party with this device, the relying party will not be able to discern that the two accounts share a device, based on the UAF protocol alone.

- The UAF protocol generates unique asymmetric cryptographic key pairs on a per-device, per-user account, and per-relying party basis. Cryptographic keys used with different relying parties will not allow any one party to link all the actions to the same user, hence the unlinkability property of UAF.

- The UAF protocol operations require minimal personal data collection: at most they incorporate a user’s relying party username. This personal data is only used for FIDO purposes, for example to perform user registration, user verification, or authorization. This personal data does not leave the user’s computing environment and is only persisted locally when necessary.

- In UAF, user verification is performed locally. The UAF protocol does not convey biometric data to relying parties, nor does it require the storage of such data at relying parties.

- Users explicitly approve the use of a UAF device with a specific relying party. Unique cryptographic keys are generated and bound to a relying party during registration only after the user’s consent.

- UAF authenticators can only be identified by their attestation certificates on a production batch-level or on manufacturer- and device model-level. They cannot be identified individually. The UAF specifications require implementers to ship UAF authenticators with the same attestation certificate and private key in batches of 100,000 or more in order to provide unlinkability.

5. Relationship to Other Technologies

This section is non-normative.

OpenID, SAML, and OAuth

FIDO protocols (both UAF and U2F) complement Federated Identity Management (FIM) frameworks, such as OpenID and SAML, as well as web
authorization protocols, such as OAuth. FIM Relying Parties can leverage an initial authentication event at an identity provider (IdP). However, OpenID and SAML do not define specific mechanisms for direct user authentication at the IdP.

When an IdP is integrated with a FIDO-enabled authentication service, it can subsequently leverage the attributes of the strong authentication with its Relying Parties. The following diagram illustrates this relationship. FIDO-based authentication (1) would logically occur first, and the FIM protocols would then leverage that authentication event into single sign-on events between the identity provider and its federated Relying Parties (2).2

![Fig. 6 FIDO UAF & Federated Identity Frameworks](image)

6. OATH, TCG, PKCS#11, and ISO 24727

These are either initiatives (OATH, Trusted Computing Group (TCG)), or industry standards (PKCS#11, ISO 24727). They all share an underlying focus on hardware authenticators.

PKCS#11 and ISO 24727 define smart-card-based authenticator abstractions.

TCG produces specifications for the Trusted Platform Module, as well as networked trusted computing.

OATH, the "Initiative for Open AuTHentication", focuses on defining symmetric key provisioning protocols and authentication algorithms for hardware One-Time Password (OTP) authenticators.

The FIDO framework shares several core notions with the foregoing efforts, such as an authentication abstraction interface, authenticator attestation, key provisioning, and authentication algorithms. FIDO's work will leverage and extend some of these specifications.

Specifically, FIDO will complement them by addressing:

- Authenticator discovery
- User experience
- Harmonization of various authenticator types, such as biometric, OTP, simple presence, smart card, TPM, etc.
7. Table of Figures

Fig. 1 FIDO UAF High-Level Architecture
Fig. 2 Registration Message Flow
Fig. 3 Authentication Message Flow
Fig. 4 Confirmation Message Flow
Fig. 5 Deregistration Message Flow
Fig. 6 FIDO UAF & Federated Identity Frameworks

1. Also known as: Authentication Tokens, Security Tokens, etc.

2. FIM protocols typically convey IdP <-> RP interactions through the browser via HTTP redirects and POSTs.
Abstract

The goal of the Universal Authentication Framework is to provide a unified and extensible authentication mechanism that supplants passwords while avoiding the shortcomings of current alternative authentication approaches.

This approach is designed to allow the relying party to choose the best available authentication mechanism for a particular end user or interaction, while preserving the option to leverage emerging device security capabilities in the future without requiring additional integration effort.

This document describes the FIDO architecture in detail, it defines the flow and content of all UAF protocol messages and presents the rationale behind the design choices.

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 Key Words

- 2. Overview
  - 2.1 Scope
  - 2.2 Architecture
  - 2.3 Protocol Conversation
    - 2.3.1 Registration
    - 2.3.2 Authentication
    - 2.3.3 Transaction Confirmation
    - 2.3.4 Deregistration
  - 2.4 Relationship to Other Specifications

- 3. Protocol Details
  - 3.1 Shared Structures and Types
    - 3.1.1 Version Interface
      - 3.1.1.1 Attributes
    - 3.1.2 Operation enumeration
    - 3.1.3 OperationHeader dictionary
      - 3.1.3.1 Dictionary OperationHeader Members
    - 3.1.4 Authenticator Attestation ID (AAID) typedef
    - 3.1.5 KeyID typedef
    - 3.1.6 ServerChallenge typedef
    - 3.1.7 FinalChallengeParams dictionary
      - 3.1.7.1 Dictionary FinalChallengeParams Members
    - 3.1.8 CollectedClientData dictionary
    - 3.1.9 TLS ChannelBinding dictionary
      - 3.1.9.1 Dictionary ChannelBinding Members
    - 3.1.10 JwkKey dictionary
      - 3.1.10.1 Dictionary JwkKey Members
    - 3.1.11 Extension dictionary
      - 3.1.11.1 Dictionary Extension Members
    - 3.1.12 MatchCriteria dictionary
      - 3.1.12.1 Dictionary MatchCriteria Members
    - 3.1.13 Policy dictionary
      - 3.1.13.1 Dictionary Policy Members
  - 3.2 Processing Rules for the Server Policy
    - 3.2.1 Examples
  - 3.3 Version Negotiation
  - 3.4 Registration Operation
    - 3.4.1 Registration Request Message
    - 3.4.2 RegistrationRequest dictionary
      - 3.4.2.1 Dictionary RegistrationRequest Members
    - 3.4.3 AuthenticatorRegistrationAssertion dictionary
      - 3.4.3.1 Dictionary AuthenticatorRegistrationAssertion Members
    - 3.4.4 Registration Response Message
    - 3.4.5 RegistrationResponse dictionary
3.4.5.1 Dictionary RegistrationResponse Members

3.4.6 Registration Processing Rules
- 3.4.6.1 Registration Request Generation Rules for FIDO Server
- 3.4.6.2 Registration Request Processing Rules for FIDO UAF Clients
  - 3.4.6.2.1 Mapping ASM Status Codes to ErrorCode
- 3.4.6.3 Registration Request Processing Rules for FIDO Authenticator
- 3.4.6.4 Registration Response Generation Rules for FIDO UAF Client
- 3.4.6.5 Registration Response Processing Rules for FIDO Server

3.5 Authentication Operation
- 3.5.1 Transaction dictionary
  - 3.5.1.1 Dictionary Transaction Members
- 3.5.2 Authentication Request Message
- 3.5.3 AuthenticationRequest dictionary
  - 3.5.3.1 Dictionary AuthenticationRequest Members
- 3.5.4 AuthenticatorSignAssertion dictionary
  - 3.5.4.1 Dictionary AuthenticatorSignAssertion Members
- 3.5.5 AuthenticationResponse dictionary
  - 3.5.5.1 Dictionary AuthenticationResponse Members
- 3.5.6 Authentication Response Message
- 3.5.7 Authentication Processing Rules
  - 3.5.7.1 Authentication Request Generation Rules for FIDO Server
  - 3.5.7.2 Authentication Request Processing Rules for FIDO UAF Client
  - 3.5.7.3 Authentication Request Processing Rules for FIDO Authenticator
  - 3.5.7.4 Authentication Response Generation Rules for FIDO UAF Client
  - 3.5.7.5 Authentication Response Processing Rules for FIDO Server

3.6 Deregistration Operation
- 3.6.1 Deregistration Request Message
- 3.6.2 DeregisterAuthenticator dictionary
  - 3.6.2.1 Dictionary DeregisterAuthenticator Members
- 3.6.3 DeregistrationRequest dictionary
  - 3.6.3.1 Dictionary DeregistrationRequest Members
- 3.6.4 Deregistration Processing Rules
  - 3.6.4.1 Deregistration Request Generation Rules for FIDO Server
  - 3.6.4.2 Deregistration Request Processing Rules for FIDO UAF Client
  - 3.6.4.3 Deregistration Request Processing Rules for FIDO Authenticator

4. Considerations
- 4.1 Protocol Core Design Considerations
  - 4.1.1 Authenticator Metadata
  - 4.1.2 Authenticator Attestation
    - 4.1.2.1 Basic Attestation
      - 4.1.2.1.1 Full Basic Attestation
      - 4.1.2.1.2 Surrogate Basic Attestation
    - 4.1.2.2 Direct Anonymous Attestation (ECDAA)
  - 4.1.3 Error Handling
  - 4.1.4 Assertion Schemes
  - 4.1.5 Username in Authenticator
  - 4.1.6 Silent Authenticators
  - 4.1.7 TLS Protected Communication
- 4.2 Implementation Considerations
  - 4.2.1 Server Challenge and Random Numbers
  - 4.2.2 Revealing KeyIDs
- 4.3 Security Considerations
1. Notation

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

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

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

The notation base64url refers to "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] without padding.

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

WebIDL dictionary members must not have a value of null — i.e., there are no declarations of nullable dictionary members in this specification.

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

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

UAF specific terminology used in this document is defined in [FIDO-Glossary].

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

---

NOTE

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

1.1 Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Overview
The goal of this Universal Authentication Framework is to provide a unified and extensible authentication mechanism that supplants passwords while avoiding the shortcomings of current alternative authentication approaches.

The design goal of the protocol is to enable Relying Parties to leverage the diverse and heterogeneous set of security capabilities available on end users' devices via a single, unified protocol.

This approach is designed to allow the FIDO Relying Parties to choose the best available authentication mechanism for a particular end user or interaction, while preserving the option for a relying party to leverage emerging device security capabilities in the future, without requiring additional integration effort.

2.1 Scope

This document describes FIDO architecture in detail and defines the UAF protocol as a network protocol. It defines the flow and content of all UAF messages and presents the rationale behind the design choices.

Particular application-level bindings are outside the scope of this document. This document is not intended to answer questions such as:

- What does an HTTP binding look like for UAF?
- How can a web application communicate to FIDO UAF Client?
- How can FIDO UAF Client communicate to FIDO enabled Authenticators?

The answers to these questions can be found in other UAF specifications, e.g. [UAFAppAPIAndTransport] [UAFASM] [UAFAuthnrCommands].

2.2 Architecture

The following diagram depicts the entities involved in UAF protocol.

![The UAF Architecture](image)

Of these entities, only these three directly create and/or process UAF protocol messages:

- FIDO Server, running on the relying party's infrastructure
- FIDO UAF Client, part of the user agent and running on the FIDO user device
- FIDO Authenticator, integrated into the FIDO user device

It is assumed in this document that a FIDO Server has access to the UAF Authenticator Metadata [FIDOMetadataStatement] describing all the authenticators it will interact with.

2.3 Protocol Conversation

The core UAF protocol consists of four conceptual conversations between a FIDO UAF Client and FIDO Server.
- **Registration**: UAF allows the relying party to register a FIDO Authenticator with the user's account at the relying party. The relying party can specify a policy for supporting various FIDO Authenticator types. A FIDO UAF Client will only register existing authenticators in accordance with that policy.

- **Authentication**: UAF allows the relying party to prompt the end user to authenticate using a previously registered FIDO Authenticator. This authentication can be invoked any time, at the relying party's discretion.

- **Transaction Confirmation**: In addition to providing a general authentication prompt, UAF offers support for prompting the user to confirm a specific transaction. This prompt includes the ability to communicate additional information to the client for display to the end user, using the client's transaction confirmation display. The goal of this additional authentication operation is to enable relying parties to ensure that the user is confirming a specified set of the transaction details (instead of authenticating a session to the user agent).

- **Deregistration**: The relying party can trigger the deletion of the account-related authentication key material.

Although this document defines the FIDO Server as the initiator of requests, in a real world deployment the first UAF operation will always follow a user agent's (e.g. HTTP) request to a relying party.

The following sections give a brief overview of the protocol conversation for individual operations. More detailed descriptions can be found in the sections Registration Operation, Authentication Operation, and Deregistration Operation.

### 2.3.1 Registration

The following diagram shows the message flows for registration.

![UAF Registration Message Flow](image)

**NOTE**

The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow the FIDO UAF Client to do some "housekeeping" tasks.

### 2.3.2 Authentication

The following diagram depicts the message flows for the authentication operation.
2.3.3 Transaction Confirmation

The following figure depicts the transaction confirmation message flow.

NOTE
The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow FIDO UAF Client to do some "housekeeping" tasks.
2.3.4 Deregistration

The following diagram depicts the deregistration message flow.

![Deregistration Message Flow Diagram]

The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow the FIDO UAF Client to do some “housekeeping” tasks.

2.4 Relationship to Other Specifications

The following data elements might be referenced by other specifications and hence should not be changed in their fundamental data type or high-level semantics without liaising with the other specifications:

1. `aaid`, data type byte string and identifying the authenticator model, i.e. identical values mean that they refer to the same authenticator model and different values mean they refer to different authenticator models.
2. `AppID`, data type string representing the Application Identifier, i.e. identical values mean that they refer to the same relying party.
3. `keyID`, data type byte string identifying a specific credential, i.e. identical values mean that they refer to the same credential and different values mean they refer to different credentials.

Some of the data elements might have an internal structure that might change. Other specifications shall not rely on such internal structure.

3. Protocol Details

This section is normative.

This section provides a detailed description of operations supported by the UAF Protocol.

Support of all protocol elements is mandatory for conforming software, unless stated otherwise.

All string literals in this specification are constructed from Unicode codepoints within the set \( \text{U+0000..U+007F} \).

Unless otherwise specified, protocol messages are transferred with a UTF-8 content encoding.

All data used in this protocol must be exchanged using a secure transport protocol (such as TLS/HTTPS) established between the FIDO UAF Client and the relying party in order to follow the assumptions made in [FIDOSecRef]; details are specified in section 4.1.7 TLS Protected Communication.
The notation `base64url(byte[8..64])` reads as 8-64 bytes of data encoded in base64url, "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] without padding.

The notation `string[5]` reads as five unicode characters, represented as a UTF-8 [RFC3629] encoded string of the type indicated in the declaration, typically a WebIDL [WebIDL-ED] DOMString.

As the UTF-8 representation has variable length, the maximum byte length of `string[5]` is `string[4*5]`.

All strings are case-sensitive unless stated otherwise.

This document uses WebIDL [WebIDL-ED] to define UAF protocol messages.

Implementations must serialize the UAF protocol messages for transmission using UTF-8 encoded JSON [RFC4627].

3.1 Shared Structures and Types

This section defines types and structures shared by various operations.

3.1.1 Version Interface

Represents a generic version with major and minor fields.

```webidl
interface Version {
    readonly attribute unsigned short major;
    readonly attribute unsigned short minor;
}
```

3.1.1.1 Attributes

- `major` of type `unsigned short`, readonly
  - Major version.

- `minor` of type `unsigned short`, readonly
  - Minor version.

3.1.2 Operation enumeration

Describes the operation type of a UAF message or request for a message.

```webidl
enum Operation {
    "Reg",
    "Auth",
    "Dereg"
};
```

<table>
<thead>
<tr>
<th>Enumeration description</th>
<th>Reg</th>
<th>Auth</th>
<th>Dereg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registration</td>
<td></td>
<td>Authentication</td>
<td>Dereg</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Or Transaction</td>
<td></td>
</tr>
</tbody>
</table>

3.1.3 OperationHeader dictionary

Represents a UAF message Request and Response header

```webidl
dictionary OperationHeader {
    required Version appy;
    required Operation reg;
    DOMString appID;
    DOMString serverData;
    Extension[];
};
```
3.1.3 Dictionary `OperationHeader` Members

**upv** of type `required Version`

UAF protocol version (`upv`). To conform with this version of the UAF spec set, the `major` value **MUST** be 1 and the `minor` value **MUST** be 2.

**op** of type `required Operation`

Name of FIDO operation (`op`) this message relates to.

NOTE

"Auth" is used for both authentication and transaction confirmation.

**appID** of type `DOMString`

`string[0..512]`.

The application identifier that the relying party would like to assert.

There are three ways to set the `appID [FIDOAppIDAndFacets]`:

1. If the element is missing or empty in the request, the FIDO UAF Client **MUST** set it to the `FacetID` of the caller.
2. If the `appID` present in the message is identical to the `FacetID` of the caller, the FIDO UAF Client **MUST** accept it.
3. If it is an URI with HTTPS protocol scheme, the FIDO UAF Client **MUST** use it to load the list of trusted facet identifiers from the specified URI. The FIDO UAF Client **MUST** only accept the request, if the facet identifier of the caller matches one of the trusted facet identifiers in the list returned from dereferencing this URI.

NOTE

The new key pair that the authenticator generates will be associated with this application identifier.

*Security Relevance:* The application identifier is used by the FIDO UAF Client to verify the eligibility of an application to trigger the use of a specific `UAuth.Key`. See [FIDOAppIDAndFacets]

**serverData** of type `DOMString`

`string[1..1536]`.

A session identifier created by the relying party.

NOTE

The relying party can opaquely store things like expiration times for the registration session, protocol version used and other useful information in `serverData`. This data is opaque to FIDO UAF Clients. FIDO Servers may reject a response that is lacking this data or is containing unauthorized modifications to it.

Servers that depend on the integrity of `serverData` should apply appropriate security measures, as described in Registration Request Generation Rules for FIDO Server and section ServerData and KeyHandle.

**exts** of type `array of Extension`

List of UAF Message Extensions.

3.1.4 Authenticator Attestation ID (AAID) typedef

```webidl
typedef DOMString AAID;
```

`string[9]`

Each authenticator **MUST** have an AAID to identify UAF enabled authenticator models globally. The AAID **MUST** uniquely identify a specific authenticator model within the range of all UAF-enabled authenticator models made by all authenticator vendors, where authenticators of a specific model must share identical security characteristics within the model (see Security Considerations).

The AAID is a string with format "V#M", where
"#" is a separator

"V" indicates the authenticator Vendor Code. This code consists of 4 hexadecimal digits.

"M" indicates the authenticator Model Code. This code consists of 4 hexadecimal digits.

The Augmented BNF [ABNF] for the AAID is:

```
AAID = 4(HEXDIG) "#" 4(HEXDIG)
```

**NOTE**

HEXDIG is case insensitive, i.e. "03EF" and "03ef" are identical.

The FIDO Alliance is responsible for assigning authenticator vendor Codes.

Authenticator vendors are responsible for assigning authenticator model codes to their authenticators. Authenticator vendors **MUST** assign unique AAIDs to authenticators with different security characteristics.

AAIDs are unique and each of them must relate to a distinct authentication metadata file ([FIDOMetadataStatement])

**NOTE**

Adding new firmware/software features, or changing the underlying hardware protection mechanisms will typically change the security characteristics of an authenticator and hence would require a new AAID to be used. Refer to ([FIDOMetadataStatement]) for more details.

### 3.1.5 KeyID typedef

```
WebIDL
typedef DOMString KeyID;
```

base64url(byte[32...2048])

KeyID is a unique identifier (within the scope of an AAID) used to refer to a specific UAAuth.Key. It is generated by the authenticator or ASM and registered with a FIDO Server.

The (AAID, KeyID) tuple **MUST** uniquely identify an authenticator's registration for a relying party. Whenever a FIDO Server wants to provide specific information to a particular authenticator it **MUST** use the (AAID, KeyID) tuple.

KeyID **MUST** be base64url encoded within the UAF message (see above).

During step-up authentication and deregistration operations, the FIDO Server **SHOULD** provide the KeyID back to the authenticator for the latter to locate the appropriate user authentication key, and perform the necessary operation with it.

Roaming authenticators which don't have internal storage for, and cannot rely on any ASM to store, generated key handles **SHOULD** provide the key handle as part of the [AAAuthnrCommands] and get the key handle back from the FIDO Server during the step-up authentication (in the [MatchCriteria] dictionary which is part of the policy) or deregistration operations (see [AAAuthnrCommands] for more details).

**NOTE**

The exact structure and content of a KeyID is specific to the authenticator / ASM implementation.

### 3.1.6 ServerChallenge typedef

```
WebIDL
typedef DOMString ServerChallenge;
```

base64url(byte[8...64])

ServerChallenge is a server-provided random challenge. **Security Relevance:** The challenge is used by the FIDO Server to verify whether an
incoming response is new, or has already been processed. See section Replay Attack Protection for more details.

The ServerChallenge SHOULD be mixed into the entropy pool of the authenticator. Security Relevance: The FIDO Server SHOULD provide a challenge containing strong cryptographic randomness whenever possible. See section Server Challenge and Random Numbers.

NOTE
The minimum challenge length of 8 bytes follows the requirement in [SP800-63] and is equivalent to the 20 decimal digits as required in [RFC6287].

NOTE
The maximum length has been defined such that SHA-512 output can be used without truncation.

NOTE
The mixing of multiple sources of randomness is recommended to improve the quality of the random numbers generated by the authenticator, as described in [RFC4086].

3.1.7 FinalChallengeParams dictionary

```webidl
dictionary FinalChallengeParams {
    required DOMString appID;
    required ServerChallenge challenge;
    required DOMString facetID;
    required ChannelBinding channelBinding;
};
```

3.1.7.1 Dictionary FinalChallengeParams Members

- **appID** of type required DOMString
  
  string[1..512]

  The value MUST be taken from the appID field of the OperationHeader.

- **challenge** of type required ServerChallenge
  
  The value MUST be taken from the challenge field of the request (e.g. RegistrationRequest.challenge, AuthenticationRequest.challenge).

- **facetID** of type required DOMString
  
  string[1..512]

  The value is determined by the FIDO UAF Client and it depends on the calling application. See [FIDOAppIDAndFacets] for more details. Security Relevance: The facetID is determined by the FIDO UAF Client and verified against the list of trusted facets retrieved by dereferencing the appID of the calling application.

- **channelBinding** of type required ChannelBinding
  
  Contains the TLS information to be sent by the FIDO Client to the FIDO Server, binding the TLS channel to the FIDO operation.

3.1.8 CollectedClientData dictionary

CollectedClientData is an alternative to the FinalChallengeParams structure. It is used by platforms supporting CTAP2 and Web Authentication. The exact definition of CollectedClientData can be found in [WebAuthn].

```webidl
dictionary CollectedClientData {
    required DOMString challenge;
};
```
3.1.9 TLS ChannelBinding dictionary

ChannelBinding contains channel binding information [RFC5056].

NOTE

Security Relevance: The channel binding may be verified by the FIDO Server in order to detect and prevent MITM attacks.

At this time, the following channel binding methods are supported:

- TokenBinding ID (tokenBinding [RFC8471])
- TLS ChannelID (cid_pubkey) [ChannelID]
- serverEndPoint [RFC5929]
- tlsServerCertificate
- tlsUnique [RFC5929]

Further requirements:

1. If data related to any of the channel binding methods, described here, is available to the FIDO UAF Client (i.e. included in this dictionary), it MUST be used according to the relevant specification.
2. All channel binding methods described here MUST be supported by the FIDO Server. The FIDO Server MAY reject operations if the channel binding cannot be verified successfully.
3.1.9.1 Dictionary ChannelBinding Members

**serverEndPoint** of type DOMString

The field `serverEndPoint` **MUST** be set to the base64url-encoded hash of the TLS server certificate if this is available. The hash function **MUST** be selected as follows:

1. if the certificate's `signatureAlgorithm` uses a single hash function and that hash function is either MD5 [RFC1321] or SHA-1 [RFC5829], then use SHA-256 [FIPS180-4];
2. if the certificate's `signatureAlgorithm` uses a single hash function and that hash function is neither MD5 nor SHA-1, then use the hash function associated with the certificate's `signatureAlgorithm`;
3. if the certificate's `signatureAlgorithm` uses no hash functions, or uses multiple hash functions, then this channel binding type's channel bindings are undefined at this time (updates to this channel binding type may occur to address this issue if it ever arises)

This field **MUST** be absent if the TLS server certificate is not available to the processing entity (e.g., the FIDO UAF Client) or the hash function cannot be determined as described.

**tlsServerCertificate** of type DOMString

This field **MUST** be absent if the TLS server certificate is not available to the FIDO UAF Client.

This field **MUST** be set to the base64url-encoded, DER-encoded TLS server certificate, if this data is available to the FIDO UAF Client.

**tlsUnique** of type DOMString

**MUST** be set to the base64url-encoded TLS channel Finished structure. It **MUST**, however, be absent, if this data is not available to the FIDO UAF Client [RFC5929].

The use of the tlsUnique is deprecated as the security of the tls-unqiue channel binding type [RFC5929] is broken, see [TLSAUTH].

**cid_pubkey** of type DOMString

**MUST** be absent if the client TLS stack doesn't provide TLS ChannelID [ChannelID] information to the processing entity (e.g., the web browser or client application).

**MUST** be set to "unused" if TLS ChannelID information is supported by the client-side TLS stack but has not been signaled by the TLS (web) server.

Otherwise, it **MUST** be set to the base64url-encoded serialized [RFC4627] `JwkKey` structure using UTF-8 encoding.

**tokenBinding** of type DOMString

**MUST** be absent if the client TLS stack doesn't provide Token Binding ID [RFC8471] information to the processing entity (e.g., the web browser or client application).

**MUST** be set to "unused" if Token Binding ID information is supported by the client-side TLS stack but has not been signaled by the TLS (web) server.

Otherwise, it **MUST** be set to the base64url-encoded serialized [RFC8471] TokenBindingID structure using UTF-8 encoding.

---

NOTE

- If channel binding data or Token Binding ID is accessible to the web browser or client application, it must be relayed to the FIDO UAF Client in order to follow the assumptions made in [FIDOSecRef].
- If channel binding data or Token Binding ID is accessible to the web server, it must be relayed to the FIDO Server in order to follow the assumptions made in [FIDOSecRef]. The FIDO Server relies on the web server to provide accurate channel binding information.
3.1.10 **JwkKey** dictionary

**JwkKey** is a dictionary representing a JSON Web Key encoding of an elliptic curve public key [JWK].

This public key is the ChannelID public key minted by the client TLS stack for the particular relying party. [ChannelID] stipulates using only a particular elliptic curve, and the particular coordinate type.

```webidl
dictionary JwkKey {
  required DOMString kty = "EC";
  required DOMString crv = "P-256";
  required DOMString x;
  required DOMString y;
}
```

**3.1.10.1 Dictionary JwkKey Members**

- **kty** of type required DOMString, defaulting to "EC"
  Denotes the key type used for Channel ID. At this time only elliptic curve is supported by [ChannelID], so it **must** be set to "EC" [JWA].

- **crv** of type required DOMString, defaulting to "P-256"
  Denotes the elliptic curve on which this public key is defined. At this time only the NIST curve secp256r1 is supported by [ChannelID], so the **crv** parameter **must** be set to "P-256".

- **x** of type required DOMString
  Contains the base64url-encoding of the x coordinate of the public key (big-endian, 32-byte value).

- **y** of type required DOMString
  Contains the base64url-encoding of the y coordinate of the public key (big-endian, 32-byte value).

3.1.11 **Extension** dictionary

FIDO extensions can appear in several places, including the UAF protocol messages, authenticator commands, or in the assertion signed by the authenticator.

Each extension has an identifier, and the namespace for extension identifiers is FIDO UAF global (i.e. doesn't depend on the message where the extension is present).

Extensions can be defined in a way such that a processing entity which doesn't understand the meaning of a specific extension **must** abort processing, or they can be specified in a way that unknown extension can (safely) be ignored.

Extension processing rules are defined in each section where extensions are allowed.

Generic extensions used in various operations.

```webidl
dictionary Extension {
  required DOMString id;
  required DOMString data;
  required boolean fail_if_unknown;
}
```

**3.1.11.1 Dictionary Extension Members**

- **id** of type required DOMString
  String[1..32].

  Identifies the extension.

- **data** of type required DOMString
  Contains arbitrary data with a semantics agreed between server and client. Binary data is base64url-encoded.

  This field **may** be empty.

- **fail_if_unknown** of type required boolean
  Indicates whether unknown extensions must be ignored (**false**) or must lead to an error (**true**).
A value of `false` indicates that unknown extensions **MUST** be ignored.

A value of `true` indicates that unknown extensions **MUST** result in an error.

**NOTE**

The FIDO UAF Client might (a) process an extension or (b) pass the extension through to the ASM. Unknown extensions must be passed through.

The ASM might (a) process an extension or (b) pass the extension through to the FIDO authenticator. Unknown extensions must be passed through.

The FIDO authenticator must handle the extension or ignore it (only if it doesn't know how to handle it and `fail_if_unknown` is not set). If the FIDO authenticator doesn't understand the meaning of the extension and `fail_if_unknown` is set, it must generate an error (see definition of `fail_if_unknown` above).

When passing through an extension to the next entity, the `fail_if_unknown` flag must be preserved (see [UAFASM] [UAFAuthnrCommands]).

FIDO protocol messages are not signed. If the security depends on an extension being known or processed, then such extension should be accompanied by a related (and signed) extension in the authenticator assertion (e.g. `TAG_UAFV1_REG_ASSERTION`, `TAG_UAFV1_AUTH_ASSERTION`). If the security has been increased (e.g. the FIDO authenticator according to the description in the metadata statement accepts multiple fingers but in this specific case indicates that the finger used at registration was also used for authentication) there is no need to mark the extension as `fail_if_unknown` (i.e. tag `0x3E12` should be used [UAFAuthnrCommands]). If the security has been degraded (e.g. the FIDO authenticator according to the description in the metadata statement accepts only the finger used at registration for authentication but in this specific case indicates that a different finger was used for authentication) the extension must be marked as `fail_if_unknown` (i.e. tag `0x3E11` must be used [UAFAuthnrCommands]).

### 3.1.12 MatchCriteria dictionary

Represents the matching criteria to be used in the server policy.

The `MatchCriteria` object is considered to match an authenticator, if *all* fields in the object are considered to match (as indicated in the particular fields).

```webidl
dictionary MatchCriteria {
  AAID[] aaid;
  DOMString[] vendorID;
  KeyID[] keyIDs;
  unsigned long userVerification;
  unsigned short keyProtection;
  unsigned short matcherProtection;
  unsigned long attachmentHint;
  unsigned short tcDisplay;
  unsigned short[] authenticationAlgorithms;
  DOMString[] assertionSchemes;
  unsigned short[] attestationTypes;
  unsigned short authenticatorVersion;
  Extension[] exts;
};
```

### 3.1.12.1 Dictionary `MatchCriteria` Members

- **aaid** of type array of `AAID`
  List of AAIDs, causing matching to be restricted to certain AAIDs.

  The field `m.aaid` **MAY** be combined with (one or more of) `m.keyIDs`, `m.attachmentHint`, `m.authenticatorVersion`, and `m.exts`, but `m.aaid` **MUST NOT** be combined with any other match criteria field.

  If `m.aaid` is not provided - both `m.authenticationAlgorithms` and `m.assertionSchemes` **MUST** be provided.

  The match succeeds if at least one AAID entry in this array matches `AuthenticatorInfo.aaid` [UAFASM].

**NOTE**

This field corresponds to `MetadataStatement.aaid` [FIDOMetadataStatement].
vendorID of type array of DOMString
The vendorID causing matching to be restricted to authenticator models of the given vendor. The first 4 characters of the AAID are the vendorID (see AAID).

The match succeeds if at least one entry in this array matches the first 4 characters of the AuthenticatorInfo.aaid [UAFASM].

NOTE
This field corresponds to the first 4 characters of MetadataStatement.aaid [FIDOMetadataStatement].

keyIDs of type array of KeyID
A list of authenticator KeyIDs causing matching to be restricted to a given set of KeyID instances. (see TAG_KEYID in [UAFRegistry]).
This match succeeds if at least one entry in this array matches.

NOTE
This field corresponds to AppRegistration.keyIDs [UAFASM].

userVerification of type unsigned long
A set of 32 bit flags which may be set if matching should be restricted by the user verification method (see [FIDORegistry]).

NOTE
The match with AuthenticatorInfo.userVerification ([UAFASM]) succeeds, if the following condition holds (written in Java):

```java
if {
    // They are equal
    (AuthenticatorInfo.userVerification == MatchCriteria.userVerification) ||
    // USER_VERIFY_ALL is not set in both of them and they have at least one common bit set
    {
        ((AuthenticatorInfo.userVerification & USER_VERIFY_ALL) == 0) &&
        ((MatchCriteria.userVerification & USER_VERIFY_ALL) == 0) &&
        (AuthenticatorInfo.userVerification & MatchCriteria.userVerification) != 0
    }
}
```

NOTE
This field value can be derived from MetadataStatement.userVerificationDetails as follows (in order to write matchCriteria that apply to the respective authenticator model):

For each entry in MetadataStatement.userVerificationDetails combine all sub-entries MetadataStatement.userVerificationDetails[i][0].userVerification to MetadataStatement.userVerificationDetails[i][N-1].userVerification into a single value using a bitwise OR operation.

The combined bitflags will either all be interpreted as alternatives or as "and" combinations (depending on the flag USER_VERIFY_ALL). For example, an authenticator that allows Passcode OR (both, Voice AND Face), will either look like:

1. Passcode OR Voice OR Face, or it will look like
2. Passcode AND Voice AND Face.

The algorithm above will encode it as alternative (1) if the USER_VERIFY_ALL flag is not set. It will encode it as alternative (2) if the USER_VERIFY_ALL flag is set.

keyProtection of type unsigned short
A set of 16 bit flags which may be set if matching should be restricted by the key protections used (see [FIDORegistry]).

This match succeeds, if at least one of the bit flags matches the value of AuthenticatorInfo.keyProtection [UAFASM].

NOTE
This field corresponds to MetadataStatement.keyProtection [FIDOMetadataStatement].
matcherProtection of type unsigned short
A set of 16 bit flags which may be set if matching should be restricted by the matcher protection (see [FIDORegistry]).

The match succeeds if at least one of the bit flags matches the value of AuthenticatorInfo.matcherProtection [UAFASM].

NOTE
This field corresponds to the MetadataStatement.matcherProtection metadata statement. See [FIDOMetadataStatement].

attachmentHint of type unsigned long
A set of 32 bit flags which may be set if matching should be restricted by the authenticator attachment mechanism (see [FIDORegistry]).

This field is considered to match, if at least one of the bit flags matches the value of AuthenticatorInfo.attachmentHint [UAFASM].

NOTE
This field corresponds to the MetadataStatement.attachmentHint metadata statement.

tcDisplay of type unsigned short
A set of 16 bit flags which may be set if matching should be restricted by the transaction confirmation display availability and type. (see [FIDORegistry]).

This match succeeds if at least one of the bit flags matches the value of AuthenticatorInfo.tcDisplay [UAFASM].

NOTE
This field corresponds to the MetadataStatement.tcDisplay metadata statement. See [FIDOMetadataStatement].

authenticationAlgorithms of type array of unsigned short
An array containing values of supported authentication algorithm TAG values (see [FIDORegistry], prefix ALG_SIGN) if matching should be restricted by the supported authentication algorithms. This field MUST be present, if field aaid is missing.

This match succeeds if at least one entry in this array matches the AuthenticatorInfo.authenticationAlgorithm [UAFASM].

NOTE
This field corresponds to the MetadataStatement.authenticationAlgorithm metadata statement. See [FIDOMetadataStatement].

assertionSchemes of type array of DOMString
A list of supported assertion schemes if matching should be restricted by the supported schemes. This field MUST be present, if field aaid is missing.

See section UAF Supported Assertion Schemes for details.

This match succeeds if at least one entry in this array matches AuthenticatorInfo.assertionScheme [UAFASM].

NOTE
This field corresponds to the MetadataStatement.assertionScheme metadata statement. See [FIDOMetadataStatement].

attestationTypes of type array of unsigned short
An array containing the preferred attestation TAG values (see [UAFRegistry], prefix TAG_ATTESTATION). The order of items MUST be preserved. The most-preferred attestation type comes first.

This match succeeds if at least one entry in this array matches one entry in AuthenticatorInfo.attestationTypes [UAFASM].

NOTE
This field corresponds to the MetadataStatement.attestationTypes metadata statement. See [FIDOMetadataStatement].
**authenticatorVersion** of type **unsigned short**
Contains an authenticator version number, if matching should be restricted by the authenticator version in use.

This match succeeds if the value is **lower or equal** to the field `authenticatorVersion` included in `TAG_UAFV1_REG_ASSERTION` or `TAG_UAFV1_AUTH_ASSERTION` or a corresponding value in the case of a different assertion scheme.

---

**NOTE**
Since the semantic of the `authenticatorVersion` depends on the AAID, the field `authenticatorVersion` should always be combined with a single `aaid` in `MatchCriteria`.

This field corresponds to the `MetadataStatement.authenticatorVersion` metadata statement. See [FIDOMetadataStatement].

The use of `authenticatorVersion` in the policy is deprecated since there is no standardized way for the FIDO Client to learn the `authenticatorVersion`. The `authenticatorVersion` is included in the authentication assertion and hence can still be evaluated in the FIDO Server.

**exts** of type array of **Extension**
Extensions for matching policy.

### 3.1.13 Policy dictionary

Contains a specification of accepted authenticators and a specification of disallowed authenticators.

#### WebIDL

```webidl
dictionary Policy {
    required MatchCriteria[][] accepted;
    MatchCriteria[] disallowed;
};;
```

#### 3.1.13.1 Dictionary Policy Members

**accepted** of type array of array of **required** MatchCriteria

This field is a two-dimensional array describing the required authenticator characteristics for the server to accept either a FIDO registration, or authentication operation for a particular purpose.

This two-dimensional array can be seen as a list of sets. List elements (i.e. the sets) are alternatives (OR condition).

All elements within a set **MUST** be combined:

The first array index indicates OR conditions (i.e. the list). Any set of authenticator(s) satisfying these `MatchCriteria` in the first index is acceptable to the server for this operation.

Sub-arrays of `MatchCriteria` in the second index (i.e. the set) indicate that multiple authenticators (i.e. each set element) **MUST** be registered or authenticated to be accepted by the server.

The `MatchCriteria` array represents ordered preferences by the server. Servers **MUST** put their preferred authenticators first, and FIDO UAF Clients **SHOULD** respect those preferences, either by presenting authenticator options to the user in the same order, or by offering to perform the operation using only the highest-preference authenticator(s).

---

**NOTE**
This list **MUST NOT** be empty. If the FIDO Server accepts any authenticator, it can follow the example below.

**EXAMPLE 1:** Example for an ‘any’ policy

```json
{
    "accepted": [
        [ { "userVerification": 1023 } ]
    ]
}
```

---

**NOTE**
3.2 Processing Rules for the Server Policy

*This section is normative.*

The FIDO UAF Client must follow the following rules while parsing server policy:

1. During registration:
   1. **Policy.accepted** is a list of combinations. Each combination indicates a list of criteria for authenticators that the server wants the user to register.
   2. Follow the priority of items in **Policy.accepted**[]. The lists are ordered with highest priority first.
   3. Choose the combination whose criteria best match the features of the currently available authenticators
   4. Collect information about available authenticators
   5. Ignore authenticators which match the **Policy.disallowed** criteria
   6. Match collected information with the matching criteria imposed in the policy (see **MatchCriteria dictionary** for more details on matching)
   7. Guide the user to register the authenticators specified in the chosen combination

2. During authentication and transaction confirmation:

   **NOTE**

   **Policy.accepted** is a list of combinations. Each combination indicates a set of criteria which is enough to completely authenticate the current pending operation

   1. Follow the priority of items in **Policy.accepted**[]. The lists are ordered with highest priority first.
   2. Choose the combination whose criteria best match the features of the currently available authenticators
   3. Collect information about available authenticators
   4. Ignore authenticators which meet the **Policy.disallowed** criteria
   5. Match collected information with the matching criteria described in the policy
   6. Guide the user to authenticate with the authenticators specified in the chosen combination
   7. A pending operation will be approved by the server only after all criteria of a single combination are entirely met

3.2.1 Examples

*This section is non-normative.*

**EXAMPLE 2:** Policy matching either a FPS-, or Face Recognition-based Authenticator

```json
{  "accepted": [    {      "userVerification": 2,      "authenticationAlgorithms": [1, 2, 5, 6],      "assertionSchemes": ["UAFV1TLV"]    },    {      "userVerification": 16,      "authenticationAlgorithms": [1, 2, 5, 6],      "assertionSchemes": ["UAFV1TLV"]    }  ]}
```

**EXAMPLE 3:** Policy matching authenticators implementing FPS and Face Recognition as alternative combination of user verification methods.

```json
{  "accepted": [    {      "userVerification": 18,      "authenticationAlgorithms": [1, 2, 5, 6],      "assertionSchemes": ["UAFV1TLV"]    }  ]}
```

Combining these two bit-flags and the flag USER_VERIFY_ALL (USER_VERIFY_ALL = 1024) into a single userVerification value would match authenticators implementing FPS and Face Recognition as a mandatory combination of user verification methods.
The next example requires two authenticators to be used:

**EXAMPLE 4: Policy matching authenticators implementing FPS and Face Recognition as mandatory combination of user verification methods.**

```
{ "accepted": [ [{ "userVerification": 1042, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"] } ] }
```

Other criteria can be specified in addition to the `userVerification`:

**EXAMPLE 5: Policy matching the combination of a FPS based and a Face Recognition based authenticator**

```
{ "accepted": [ [ { "userVerification": 2, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"] }, { "userVerification": 16, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"] } ] ]
```

The policy for accepting authenticators of vendor with ID 1234 only is as follows:

**EXAMPLE 6: Policy requiring the combination of a bound FPS based and a bound Face Recognition based authenticator**

```
{ "accepted": [ [ { "userVerification": 2, "attachmentHint": 1, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"] }, { "userVerification": 16, "attachmentHint": 1, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"] } ] ]
```

**EXAMPLE 7: Policy accepting all authenticators from vendor with ID 1234**

```
{ "accepted": [ [{ "vendorID": "1234", "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"] } ] }
```

### 3.3 Version Negotiation

The UAF protocol includes multiple versioned constructs: UAF protocol version, the version of Key Registration Data and Signed Data objects (identified by their respective tags, see [UAFRegistry]), and the ASM version, see [UAFASM].

**NOTE**

The Key Registration Data and Signed Data objects have to be parsed and verified by the FIDO Server. This verification is only possible if the FIDO Server understands their encoding and the content. Each UAF protocol version supports a set of Key Registration Data and SignedData object versions (called Assertion Schemes). Similarly each of the ASM versions supports a set Assertion Scheme versions.

As a consequence the FIDO UAF Client MUST select the authenticators which will generate the appropriately versioned constructs.

For version negotiation the FIDO UAF Client MUST perform the following steps:

1. Create a set (`FC_Version_Set`) of version pairs, ASM version (`asmVersion`) and UAF Protocol version (`upv`) and add all pairs supported by the FIDO UAF Client into `FC_Version_Set`
   - e.g. `[(upv1, asmVersion1), (upv2, asmVersion1), ...]`

**NOTE**

The ASM versions are retrieved from the `AuthenticatorInfo.asmVersion` field. The UAF protocol version is derived from the related `AuthenticatorInfo.assertionScheme` field.
2. Intersect `FC_Version_Set` with the set of `upv` included in UAF Message (i.e. keep only those pairs where the `upv` value is also contained in the UAF Message).

3. Select authenticators which are allowed by the UAF Message Policy. For each authenticator:
   - Construct a set (`Authnr_Version_Set`) of version pairs including authenticator supported `asmVersion` and the compatible `upv(s)`.
     - **e.g.** `{{upv1, asmVersion1}, {upv2, asmVersion1}, ...}`
   - Intersect `Authnr_Version_Set` with `FC_Version_Set` and select highest version pair from it.
     - Take the pair where the `upv` is highest. In all these pairs leave only the one with highest `asmVersion`.
   - Use the remaining version pair with this authenticator

**NOTE**

Each version consists of major and minor fields. In order to compare two versions - compare the Major fields and if they are equal compare the Minor fields.

Each UAF message contains a version field `upv`. UAF Protocol version negotiation is always between FIDO UAF Client and FIDO Server.

A possible implementation optimization is to have the RP web application itself preemptively convey to the FIDO Server the UAF protocol version(s) (UPV) supported by the FIDO Client. This allows the FIDO Server to craft its UAF messages using the UAF version most preferred by both the FIDO client and server.

3.4 Registration Operation

**NOTE**

The Registration operation allows the FIDO Server and the FIDO Authenticator to agree on an authentication key.
The steps 11a and 11b and 12 to 13 are not always necessary as the related data could be cached.

The following diagram depicts the cryptographic data flow for the registration sequence.
3.4.1 Registration Request Message

UAF Registration request message is represented as an array of dictionaries. The array **MUST** contain exactly one dictionary. The request is defined as **RegistrationRequest** dictionary.

**EXAMPLE 8: UAF Registration Request**

```json
[
  {
    "header": {
      "upv": {
        "major": 1,
        "minor": 2
      },
      "op": "Reg",
      "appID": "https://uaf.example.com/facets.json",
      "serverData": "ZQ_fRGDH2ar_LvrTM8JnQcl-wfnaOutiyCmpBgmMcuE"
    },
    "challenge": "Yb39SdUhU2B0089pS5L7VBW8afdlp1nvR4BlAna5vk4",
    "username": "alice@website.org",
    "policy": {
      "accepted": [
        {
          "aaid": "FFFF#FC03"
        },
        {
          "userVerification": 512,
          "keyProtection": 1,
          "tcDisplay": 1,
          "authenticationAlgorithms": [1],
          "assertionSchemes": ["UAFV1TLV"]
        },
        {
          "userVerification": 4,
          "keyProtection": 1,
          "tcDisplay": 1,
          "authenticationAlgorithms": [1],
          "assertionSchemes": ["UAFV1TLV"]
        }
      ]
    }
  }
]```
3.4.2 RegistrationRequest dictionary

RegistrationRequest contains a single, versioned, registration request.

```webidl
dictionary RegistrationRequest {
  required OperationHeader header;
  required ServerChallenge challenge;
  required DOMString username;
  required Policy policy;
};
```

3.4.2.1 Dictionary RegistrationRequest Members

- **header** of type **required OperationHeader**
  - Operation header. Header.op MUST be "Reg"

- **challenge** of type **required ServerChallenge**
  - Server-provided challenge value

- **username** of type **required DOMString**
  - string[1..128]

  A human-readable user name intended to allow the user to distinguish and select from among different accounts at the same relying party.
3.4.3 AuthenticatorRegistrationAssertion dictionary

Contains the authenticator’s response to a RegistrationRequest message:

```
WebIDL
dictionary AuthenticatorRegistrationAssertion {
  required DOMString assertionScheme;
  required DOMString assertion;
  DisplayPNGCharacteristicsDescriptor[] tcDisplayPNGCharacteristics;
  Extension[] exts;
};
```

3.4.3.1 Dictionary AuthenticatorRegistrationAssertion Members

- **assertionScheme** of type required DOMString
  The name of the Assertion Scheme used to encode the assertion. See UAF Supported Assertion Schemes for details.

- **assertion** of type required DOMString
  base64url(byte[1..4096])
  Contains the TAG_UAFV1_REG_ASSERTION object containing the assertion scheme specific KeyRegistrationData (KRD) object which in turn contains the newly generated UAuth.pub and is signed by the Attestation Private Key.
  This assertion MUST be generated by the authenticator and it MUST be used only in this Registration operation. The format of this assertion can vary from one assertion scheme to another (e.g. for "UAFV1TLV" assertion scheme it MUST be TAG_UAFV1_KRD).

- **tcDisplayPNGCharacteristics** of type array of DisplayPNGCharacteristicsDescriptor
  Supported transaction PNG type [FIDOMetadataStatement]. For the definition of the DisplayPNGCharacteristicsDescriptor structure See [FIDOMetadataStatement].

- **exts** of type array of Extension
  Contains Extensions prepared by the authenticator

3.4.4 Registration Response Message

A UAF Registration response message is represented as an array of dictionaries. Each dictionary contains a registration response for a specific protocol version. The array MUST NOT contain two dictionaries of the same protocol version. The response is defined as RegistrationResponse dictionary.

```
EXAMPLE 9: Registration Response

[{
  "header": {
    "upv": {
      "major": 1,
      "minor": 2
    },
    "op": "Reg",
    "appID": "https://uaf.example.com/facets.json",
    "serverData": "ZQ_fRGDH2ar_LvrTM8JnQcl-wfnaOutiyCmpBgmMcuE"
  },
  "fcParams": "eyJmYWNldElEIjoiaHR0cHM6Ly91YWYuZ2VuZXJhbXBlb24zNzg2MDI3MDAyMSwvY3VubG9hZ3JvdXJyZWN0b3JrcnkuZXMvdjIuYXVzdF9fXVApJSIsInN0YW5uZWxCaW5kaW5nIjp7fX0",
  "assertions": [{
    "assertionScheme": "UAFV1TLV",
    "assertion": "AT73AgM-sQALLgkARkZGRiNGQzAzDi4HAAEAAQIAAAEKLiAAbkZZjz4ysihP9vVgevgoHSEV2JfK
    TxFPSFzNaAioqZLlALaMmnnfJyYi00cXGLVvZEdQhj1z7qogsmi1zCtezEcvlONLnpaaIAAEAAAKLxAR1YnN1A1p
    cRFr8o.9q6qS5yW7oaMmkgcB8yifF1aCX9hj7Y1YR080Gop0uV6rWzHBmyUQ9gF8P1.1Q9IaM0Shp4cB150A8EAI
    C577z2CmWhC11IuYwUSeFjz3yy16KfPp rgWbK5050w1gbT62m3Y7654pveOxAcATFb9n6Y/LcncK-ykOVX8KfLh
    ABWlJ770C3aZpKAwhB18gB1bDKQ2agphk7qOVhQrj9we6rQwCQYDQQG6w304ljEjWCEGAL1UEwawKlE7P2U25mb3jW
    Nj1FLF13c9Q9gV9bibiMNPJAUbNgVAeM0W5URE8gqWxDeWVYu2UXjDA1bgVBb7M0G1CnRpmljY1R9P94V2Y2u12u2y
    bHcm9icAfExeN2zAgyhJXkXN5m7M1uJ2Yw7yjAyjHgkWmmf7jAmjHxZ5H57JAHbASn2EHC1qgB3625H57QQ25D8P0
    00033373zVR9Q6hYQ9QdC beckBu9cR2c8Mo8B0MdDA1UECwqrK81y8PByBbSgYPM5j77F5MC5GAIaUcwwqQ25YyDiG1maWnhv01
    vbi
    BX3j3A9n5iEidyh3W4MDxwEvHIKo21ij7C9A9YK6i51izX0AQaDgloEllj8B2e8341f7A171L1Jt40-12wq
    1fDw_syyiyT6mL9GC0OVXE-L6701ID8K924f1BPR-o-qMksNswyAYVcrTBH0wAwEB aZLbk9YlWQR8EBAMCBAwCo
    YK2Zf1j5J9Ew1Z5AAaw0PqP0y10z8U8F6c4OXeCv5Xb1e1omexgF-1v9K0gP59cC7qF6PCAr3h3-Eycz_sFiP9K
    2k02h0Mn9cn-bBrYQ"
  }]
}
```
### 3.4.5 RegistrationResponse dictionary

Contains all fields related to the registration response.

```webIDL
dictionary RegistrationResponse {
    required OperationHeader header;
    required DOMString fcParams;
    required AuthenticatorRegistrationAssertion[] assertions;
}
```

#### 3.4.5.1 Dictionary RegistrationResponse Members

- **header** of type required OperationHeader
  
  `Header.op` **MUST** be "Reg".

- **fcParams** of type required DOMString
  
  The base64url-encoded serialized [RFC4627] FinalChallengeParams using UTF8 encoding (see FinalChallengeParams dictionary) or alternatively it contains the serialized CollectedClientData object. In both cases, all parameters required for the server to verify the Final Challenge are included.

- **assertions** of type array of required AuthenticatorRegistrationAssertion
  
  Response data for each Authenticator being registered.

### 3.4.6 Registration Processing Rules

#### 3.4.6.1 Registration Request Generation Rules for FIDO Server

The policy contains a two-dimensional array of allowed MatchCriteria (see Policy). This array can be considered a list (first dimension) of sets (second dimension) of authenticators (identified by MatchCriteria). All authenticators in a specific set **MUST** be registered simultaneously in order to match the policy. But any of those sets in the list are valid, as the list elements are alternatives.

The FIDO Server **MUST** follow the following steps:

1. Construct appropriate authentication policy `p`
   1. for each set of alternative authenticators do
      1. Create an array of MatchCriteria objects, containing the set of authenticators to be registered simultaneously that need to be identified by separate MatchCriteria objects `m`.
         1. For each collection of authenticators `a` to be registered simultaneously that can be identified by the same rule, create a MatchCriteria object `m`, where
            - `m.aaid` **MAY** be combined with (one or more of) `m.keyIDs`, `m.attachmentHint`, `m.authenticatorVersion`, and `m.exts`, but `m.aaid` **MUST NOT** be combined with any other match criteria field.
            - If `m.aaid` is not provided - both `m.authenticationAlgorithms` and `m.assertionSchemes` **MUST** be provided
         2. Add `m` to `v`, e.g. `v[j+1]=m`.
      2. Add `v` to `p.allowed`, e.g. `p.allowed[i+1]=v`
   2. Create MatchCriteria objects `m[]` for all disallowed Authenticators.
      1. For each already registered AAID for the current user
         1. Create a MatchCriteria object `m` and add AAID and corresponding KeyIDs to `m.aaid` and `m.KeyIDs`.
            
            The FIDO Server **MUST** include already registered AAIDs and KeyIDs into field `p.disallowed` to hint that the client should not register these again.
         2. Create a MatchCriteria object `m` and add the AAIDs of all disallowed Authenticators to `m.aaid`.
            
            The status (as provided in the metadata TOC (Table-of-Contents file) [FIDOMetadataService]) of some authenticators might
be unacceptable. Such authenticators **SHOULD** be included in \( p \text{.disallowed} \).

3. If needed - create MatchCriteria \( m \) for other disallowed criteria (e.g. unsupported authenticationAlgs)
4. Add all \( m \) to \( p \text{.disallowed} \).

2. Create a RegistrationRequest object \( r \) with appropriate \( r \text{.header} \) for each supported version, and
1. FIDO Servers **SHOULD NOT** assume any implicit integrity protection of \( r \text{.header.serverData} \).

FIDO Servers that depend on the integrity of \( r \text{.header.serverData} \) **SHOULD** apply and verify a cryptographically secure Message Authentication Code (MAC) to serverData and they **SHOULD** also cryptographically bind serverData to the related message, e.g. by re-including \( r \text{.challenge} \), see also section **ServerData and KeyHandle**.

2. Generate a random challenge and assign it to \( r \text{.challenge} \)
3. Assign the username of the user to be registered to \( r \text{.username} \)
4. Assign \( p \) to \( r \text{.policy} \).
5. Append \( r \) to the array \( o \) of message with various versions (RegistrationRequest)
3. Send \( o \) to the FIDO UAF Client

### 3.4.6.2 Registration Request Processing Rules for FIDO UAF Clients

The FIDO UAF Client **MUST** perform the following steps:

1. Choose the message \( m \) with upv set to the appropriate version number.
2. Parse the message \( m \)
3. If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
4. Filter the available authenticators with the given policy and present the filtered authenticators to User. Make sure to not include already registered authenticators for this user specified in \( \text{ReqRequest.policy.disallowed}[] \text{.keyIDs} \)
5. Obtain FacetID of the requesting Application. If the AppID is missing or empty, set the AppID to the FacetID.

   Verify that the FacetID is authorized for the AppID according to the algorithms in [FIDOAppIDAndFacets].

   - If the FacetID of the requesting Application is not authorized, reject the operation
6. Obtain TLS data if it is available
7. Create a FinalChallengeParams structure \( fcp \) and set \( fcp \text{.appID} \), \( fcp \text{.challenge} \), \( fcp \text{.facetID} \), and \( fcp \text{.channelBinding} \) appropriately.
   Serialize [RFC4627] \( fcp \) using UTF8 encoding and base64url encode it.

   - FinalChallenge = base64url(serialize(utf8encode(fcp)))
8. For each authenticator that matches UAF protocol version (see section **Version Negotiation**) and user agrees to register:
   1. Add AppID, Username, FinalChallenge, AttestationType and all other required fields to the ASMRequest [UAFASM].

      The FIDO UAF Client **MUST** follow the server policy and find the single preferred attestation type. A single attestation type **MUST** be provided to the ASM.

   2. Send the ASMRequest to the ASM. If the ASM returns an error, handle that error appropriately. The status code returned by the ASM [UAFASM] must be mapped to a status code defined in [UAFAppAPIAndTransport] as specified in section **3.4.6.2.1 Mapping ASM Status Codes to ErrorCode**.

### 3.4.6.2.1 Mapping ASM Status Codes to ErrorCode

ASMs are returning a status code in their responses to the FIDO Client. The FIDO Client needs to act on those responses and also map the status code returned the ASM [UAFASM] to an ErrorCode specified in [UAFAppAPIAndTransport].

The mapping of ASM status codes to ErrorCode is specified here:
<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAF_ASM_STATUS_OK</td>
<td>Pass-through success status.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_ERROR</td>
<td>Map to UNKNOWN.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
<td>Map to AUTHENTICATOR_ACCESS_DENIED.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_USER_CANCELLED</td>
<td>Pass-through status code.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT</td>
<td>Map to INVALID_TRANSACTION_CONTENT. This code indicates a problem to be</td>
</tr>
<tr>
<td></td>
<td>resolved by the entity providing the transaction text.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY</td>
<td>Pass-through status code. It indicates that the Uauth key disappeared</td>
</tr>
<tr>
<td></td>
<td>permanently and the RP App might want to trigger re-registration of the</td>
</tr>
<tr>
<td></td>
<td>authenticator.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED</td>
<td>Retry operation with other suitable authenticators and map to</td>
</tr>
<tr>
<td></td>
<td>NO_SUITABLE_AUTHENTICATOR or WAIT_USER_ACTION if the problem persists.</td>
</tr>
<tr>
<td></td>
<td>Return WAIT_USER_ACTION if being called while retrying.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_USER_NOT_RESPONSIVE</td>
<td>Pass-through status code. The RP App might want to retry the operation</td>
</tr>
<tr>
<td></td>
<td>once the user pays attention to the application again.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES</td>
<td>The FIDO Client SHALL try other authenticators matching the policy. If</td>
</tr>
<tr>
<td></td>
<td>none exist, pass-through status code.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_USER_LOCKOUT</td>
<td>Pass-through status code.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_USER_NOT_ENROLLED</td>
<td>Pass-through status code.</td>
</tr>
<tr>
<td>UAF_ASM_STATUS_SYSTEM_INTERRUPTED</td>
<td>Pass-through status code.</td>
</tr>
<tr>
<td>Any other status code</td>
<td>Map any unknown error code to UNKNOWN. This might happen when a FIDO Client</td>
</tr>
<tr>
<td></td>
<td>communicates with an ASM implementing a newer UAF specification than the</td>
</tr>
<tr>
<td></td>
<td>FIDO Client.</td>
</tr>
</tbody>
</table>

3.4.6.3 Registration Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section "Register Command".

3.4.6.4 Registration Response Generation Rules for FIDO UAF Client

The FIDO UAF Client must follow the steps:

1. Create a RegistrationResponse message
2. Copy RegistrationRequest.header into RegistrationResponse.header

**NOTE**

When the appID provided in the request was empty, the FIDO Client must set the appID in this header to the facetID (see [FIDOAppIDAndFacets]).

The header might include extensions. Extension specific rules might affect the copy process. Those rules are defined in the related section in [UAFRegistry].
3. Set `RegistrationResponse.fcParams` to `FinalChallenge` (base64url encoded serialized and utf8 encoded `FinalChallengeParams`)
4. Append the response from each Authenticator into `RegistrationResponse.assertions`
5. Send `RegistrationResponse` message to FIDO Server

### 3.4.6.5 Registration Response Processing Rules for FIDO Server

**NOTE**

The following processing rules assume that Authenticator supports "UAFV1TLV" assertion scheme. Currently "UAFV1TLV" is the only defined and supported assertion scheme. When a new assertion scheme is added to UAF protocol - this section will be extended with corresponding processing rules.

The FIDO Server **MUST** follow the steps:

1. Parse the message
   1. If protocol version (`RegistrationResponse.header.upv`) is not supported – reject the operation
   2. If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
2. Verify that `RegistrationResponse.header.serverData`, if used, passes any implementation-specific checks against its validity. See also section ServerData and KeyHandle.
3. base64url decode `RegistrationResponse.fcParams` and convert it into an object (`fcp`)
4. If this `fcp` object is a `FinalChallengeParams` object, then verify each field in `fcp` and make sure it is valid:
   1. Make sure `fcp.appID` corresponds to the one stored by the FIDO Server
      **NOTE**
      When the `appID` provided in the request was empty, the FIDO Client must set the `appID` to the facetID (see [FIDOAppIDAndFacets]). In this case, the Uauth key cannot be used by other application facets.
   2. Make sure `fcp.facetID` is in the list of trusted FacetIDs [FIDOAppIDAndFacets]
   3. Make sure `fcp.channelBinding` is as expected (see section ChannelBinding dictionary)
      **NOTE**
      There might be legitimate situations in which some methods of channel binding fail (see section 4.3.4 TLS Binding).
   4. Make sure `fcp.challenge` has really been generated by the FIDO Server for this operation and it is not expired
   5. Reject the response if any of these checks fails
5. If this `fcp` object is a `CollectedClientData` object, then verify each field in `fcp` and make sure it is valid:
   1. Make sure `fcp.origin` is considered a legitimate origin for this registration request.
   2. Make sure `fcp.tokenBinding` is as expected (see field `cid_pubkey` in section ChannelBinding dictionary)
      **NOTE**
      There might be legitimate situations in which some methods of channel binding fail (see section 4.3.4 TLS Binding).
   3. Make sure `fcp.challenge` has really been generated by the FIDO Server for this operation and it is not expired
   4. Reject the response if any of these checks fails
6. For each assertion `a` in `RegistrationResponse.assertions`
   1. Parse data from `a` assuming it is encoded according to the suspected assertion scheme `a.assertionScheme` and make sure it contains all mandatory fields (indicated in Authenticator Metadata) it is supposed to have, verify that the assertion has a valid syntax, and verify that the assertion doesn't include unknown fields (identified by TAGs or IDs) that belong to extensions marked as "fail-if-unknown" set to true [FIDOMetadataStatement].
      - If it doesn't - continue with next assertion
   2. if `a.assertion` contains an object of type `TAG_UAFV1_REG_ASSERTION`, then
1. Retrieve the AAID from the assertion.

**NOTE**
The AAID in `TAG_UAF_V1_KRD` is contained in `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_AAID`.

2. Verify that `a.assertionScheme` matches `Metadata(AAID).assertionScheme`
   - If it doesn't match - continue with next assertion

3. Verify that the AAID indeed matches the policy specified in the registration request.

**NOTE**
Depending on the policy (e.g. in the case of AND combinations), it might be required to evaluate other assertions included in this `RegistrationResponse` in order to determine whether this AAID matches the policy.

- If it doesn't match the policy - continue with next assertion

4. Locate authenticator-specific authentication algorithms from the authenticator metadata [FIDOMetadataStatement] using the AAID.

5. If `fcp` is of type `FinalChallengeParams`, then hash `RegistrationResponse.fcParams` using hashing algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator metadata, field `AuthenticationAlgs`. It is the hash algorithm associated with the first entry related to a constant with prefix `ALG_SIGN`.
   - `FCHash = hash(RegistrationResponse.fcParams)`

6. If `fcp` is of type `CollectedClientData`, then hash `RegistrationResponse.fcParams` using hashing algorithm specified in `fcp.hashAlg`.
   - `FCHash = hash(RegistrationResponse.fcParams)`

7. If `a.assertion.TAG_UAFV1_REG_ASSERTION` contains `TAG_UAFV1_KRD` as first element:
   1. Obtain `Metadata(AAID).AttestationType` for the AAID and make sure that `a.assertion.TAG_UAFV1_REG_ASSERTION` contains the most preferred attestation tag specified in field `MatchCriteria.attestationTypes` in `RegistrationRequest.policy` (if this field is present).
      - If `a.assertion.TAG_UAFV1_REG_ASSERTION` doesn't contain the preferred attestation - it is RECOMMENDED to skip this assertion and continue with next one
   2. Make sure that `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.FinalChallengeHash == FCHash`
      - If comparison fails - continue with next assertion
   3. Obtain `Metadata(AAID).AuthenticatorVersion` for the AAID and make sure that it is lower or equal to `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.AuthenticatorVersion`.
      - If `Metadata(AAID).AuthenticatorVersion` is higher (i.e. the authenticator firmware is outdated), it is RECOMMENDED to assume increased risk. See sections "StatusReport dictionary" and "Metadata TOC object Processing Rules" in [FIDOMetadataService] for more details on this.
   4. Check whether `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.RegCounter` is acceptable, i.e. it is either not supported (value is 0 or the field isKeyRestricted is set to 'false' in the related Metadata Statement) or it is not exceedingly high
      - If `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.RegCounter` is exceedingly high, this assertion might be skipped and processing will continue with next one

5. If `a.assertion.TAG_UAFV1_REG_ASSERTION` contains `ATTESTATION_BASIC_FULL` tag
   1. If entry `AttestationRootCertificates` for the AAID in the metadata [FIDOMetadataStatement] contains at least one element:
      1. Obtain contents of all `TAG_ATTESTATION_CERT` tags from `a.assertion.TAG_UAFV1_REG_ASSERTION.ATTESTATION_BASIC_FULL` object. The occurrences are ordered (see [UAFAuthnrCommands]) and represent the attestation certificate followed by the related certificate chain.
      2. Obtain all entries of `AttestationRootCertificates` for the AAID in authenticator Metadata, field `AttestationRootCertificates`.
      3. Verify the attestation certificate and the entire certificate chain up to the Attestation Root Certificate using Certificate Path Validation as specified in [RFC5280]
         - If verification fails – continue with next assertion
   4. Verify `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.ATTESTATION_BASIC_FULL.Signature` using the attestation certificate (obtained before).
      - If verification fails – continue with next assertion
2. If Metadata(AAID).AttestationRootCertificates for this AAID is empty - continue with next assertion
3. Mark assertion as positively verified

6. If a.assertion.TAG_UAFV1_REG_ASSERTION contains an object of type ATTESTATION_BASIC_SURROGATE
   1. There is no real attestation for the AAID, so we just assume the AAID is the real one.
   2. If entry AttestationRootCertificates for the AAID in the metadata is empty
      ▪ Verify a.assertion.TAG_UAFV1_REG_ASSERTION.ATTESTATION_BASIC_SURROGATE.Signature using
        a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_PUB_KEY
      ▪ If verification fails – continue with next assertion
   3. If entry AttestationRootCertificates for the AAID in the metadata is not empty - continue with next assertion (as the
      AAID obviously is expecting a different attestation method).
   4. Mark assertion as positively verified

7. If a.assertion.TAG_UAFV1_REG_ASSERTION contains an object of type ATTESTATION_ECDAA
   1. If entry ecdaaTrustAnchors for the AAID in the metadata contains at least one element:
      ▪ For each of the ecdaaTrustAnchors entries, perform the ECDAA Verify operation as specified in
        [FIDOEcdaaAlgorithm].
      ▪ If verification fails – continue with next ecdaaTrustAnchors entry
   2. If no ECDAA Verify operation succeeded – continue with next assertion
   3. If Metadata(AAID).ecdaaTrustAnchors for this AAID is empty - continue with next assertion
   4. Mark assertion as positively verified and the authenticator indeed is of model as indicated by the AAID.

8. If a.assertion.TAG_UAFV1_REG_ASSERTION contains another TAG_ATTESTATION tag - verify the attestation by following
   appropriate processing rules applicable to that attestation. Currently this document defines the processing rules for
   Basic Attestation and direct anonymous attestation (ECDAA).

8. if a.assertion.TAG_UAFV1_REG_ASSERTION contains a different object than TAG_UAFV1_KRD as first element, then follow the rules
   specific to that object.

9. Extract a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.PublicKey into PublicKey,
   a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.KeyID into KeyID,
   a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.SignCounter into SignCounter,
   a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_ASSERTION_INFO.authenticatorVersion into AuthenticatorVersion,
   a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_AAID into AAID.

3. if a.assertion doesn't contain an object of type TAG_UAFV1_REG_ASSERTION, then then follow the respective processing rules of that
   assertion format if supported - otherwise skip this assertion.

7. For each positively verified assertion a
   • Store PublicKey, KeyID, SignCounter, AuthenticatorVersion, AAID and a.tcDisplayPNGCharacteristics into a record associated
     with the user's identity. If an entry with the same pair of AAID and KeyID already exists then fail (should never occur).

3.5 Authentication Operation

NOTE
The steps 7a and 7a and 8 to 9 are not always necessary as the related data could be cached.

The TransactionText (TranTxt) is only required in the case of Transaction Confirmation (see section 3.5.1 Transaction dictionary), it is absent in the case of a pure Authenticate operation.

During this operation, the FIDO Server asks the FIDO UAF Client to authenticate user with server-specified authenticators, and return an authentication response.

In order for this operation to succeed, the authenticator and the relying party must have a previously shared registration.
3.5.1 Transaction dictionary

Contains the Transaction Content provided by the FIDO Server:

```webidl
dictionary Transaction {
  required DOMString contentType;
  required DOMString content;
  DisplayPNGCharacteristicsDescriptor tcDisplayPNGCharacteristics;
};
```

3.5.1.1 Dictionary Transaction Members

`contentType` of type `required DOMString` 
Contains the MIME Content-Type supported by the authenticator according to its metadata statement (see [FIDOMetadataStatement]).

**NOTE**

For best interoperability, at least the values `text/plain` and/or `image/png` should be supported.

`content` of type `required DOMString`

base64url(byte[1...])
Contains the base64url encoded transaction content according to the contentType to be shown to the user.

If contentType is "text/plain" then the content MUST be the base64url encoding of the UTF8 [RFC3629] encoded text with a maximum length of 200 characters. The Authenticator SHALL display the default character if it doesn’t know how to display the intended one.

If contentType is "image/png" or any other type, then it must be base64url encoded (i.e. the base64url encoded PNG [PNG] image in the case of "image/png").

tcDisplayPNGCharacteristics of type DisplayPNGCharacteristicsDescriptor
Transaction content PNG characteristics. For the definition of the DisplayPNGCharacteristicsDescriptor structure See [FIDOMetadataStatement]. This field MUST be present if the contentType is "image/png".

3.5.2 Authentication Request Message

UAF Authentication request message is represented as an array of dictionaries. The array MUST contain exactly one dictionary. The request is defined as AuthenticationRequest dictionary.

EXAMPLE 10: UAF Authentication Request

```json
{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Auth",
        "appID": "https://uaf.example.com/facets.json",
        "serverData": "mZ0YSKHLXHd_STbbDIN2aRv03Pa6srrNMPYp2q0z3-V"
    },
    "challenge": "4D8eUxdS9Q_Rbk7Gf0SooK7Xr902LU-g150stOpK0go",
    "policy": {
        "accepted": [
            {
                "aaid": ["FFFF#FC01"]
            },
            {
                "userVerification": 512,
                "keyProtection": 1,
                "tcDisplay": 1,
                "authenticationAlgorithms": [1],
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 4,
                "keyProtection": 1,
                "tcDisplay": 1,
                "authenticationAlgorithms": [1],
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 4,
                "keyProtection": 1,
                "tcDisplay": 1,
                "authenticationAlgorithms": [1],
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 4,
                "keyProtection": 1,
                "tcDisplay": 1,
                "authenticationAlgorithms": [2]
            },
            {
                "userVerification": 2,
                "keyProtection": 4,
                "tcDisplay": 1,
                "authenticationAlgorithms": [2]
            },
            {
                "userVerification": 4,
                "keyProtection": 2,
                "tcDisplay": 1,
                "authenticationAlgorithms": [1, 3]
            },
            {
                "userVerification": 2,
                "keyProtection": 2,
                "authenticationAlgorithms": [2]
            },
            {
                "userVerification": 32,
                "keyProtection": 2,
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 2,
                "authenticationAlgorithms": [1, 3],
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 2,
                "authenticationAlgorithms": [1, 3],
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 4,
                "keyProtection": 1,
                "authenticationAlgorithms": [1, 3],
                "assertionSchemes": ["UAFV1TLV"]
            }
        ]
    }
}
```
3.5.3 AuthenticationRequest dictionary

Contains the UAF Authentication Request Message:

```webidl
dictionary AuthenticationRequest {
    required OperationHeader header;
    required ServerChallenge challenge;
    Transaction[] transaction;
    required Policy policy;
};
```
3.5.3.1 Dictionary AuthenticationRequest Members

- **header** of type required OperationHeader
  
  `Header.op` **MUST** be "Auth"

- **challenge** of type required ServerChallenge
  
  Server-provided challenge value

- **transaction** of type array of Transaction
  
  Transaction data to be explicitly confirmed by the user.

  The list contains the same transaction content in various content types and various image sizes. Refer to [FIDOMetadataStatement] for more information about Transaction Confirmation Display characteristics.

- **policy** of type required Policy
  
  Server-provided policy defining what types of authenticators are acceptable for this authentication operation.

3.5.4 AuthenticatorSignAssertion dictionary

Represents a response generated by a specific Authenticator:

```
WebIDL
dictionary AuthenticatorSignAssertion {
  required DOMString assertionScheme;
  required DOMString assertion;
  Extension[] exts;
};
```

3.5.4.1 Dictionary AuthenticatorSignAssertion Members

- **assertionScheme** of type required DOMString
  
  The name of the Assertion Scheme used to encode `assertion`. See UAF Supported Assertion Schemes for details.

  **NOTE**
  
  This assertionScheme is not part of a signed object and hence considered the suspected assertionScheme.

- **assertion** of type required DOMString
  
  `base64url(byte[1..4096])` Contains the assertion containing a signature generated by UAuth.priv, i.e. `TAG_UAFV1_AUTH_ASSERTION`.

- **exts** of type array of Extension
  
  Any extensions prepared by the Authenticator

3.5.5 AuthenticationResponse dictionary

Represents the response to a challenge, including the set of signed assertions from registered authenticators.

```
WebIDL
dictionary AuthenticationResponse {
  required OperationHeader header;
  required DOMString fcParams;
  required AuthenticatorSignAssertion[] assertions;
};
```

3.5.5.1 Dictionary AuthenticationResponse Members

- **header** of type required OperationHeader
  
  `Header.op` **MUST** be "Auth"

- **fcParams** of type required DOMString
  
  The field fcParams is the base64url-encoded serialized [RFC4627] FinalChallengeParams in UTF8 encoding (see FinalChallengeParams dictionary) or alternatively it contains the serialized CollectedClientData object. In both cases, all parameters required for the server to verify the Final Challenge are included.
assertions of type array of required AuthenticatorSignAssertion
The list of authenticator responses related to this operation.

3.5.6 Authentication Response Message

UAF Authentication response message is represented as an array of dictionaries. The array must contain exactly one dictionary. The response is defined as AuthenticationResponse dictionary.

EXAMPLE 12: UAF Authentication Response
```json
[
  {
    "header": {
      "upv": {
        "major": 1,
        "minor": 2
      },
      "op": "Auth",
      "appId": "https://uaf.example.com/facets.json",
      "serverData": "mzUSKHLDsd_StbdDIN3aW6x3yNMPyPyp2Qz3-Y"
    },
    "fcParams": "eyJmYWNndE1EIjoiaHR0cMN6Ly91YVYuZXhhbXBza2551b20iLChJci6JZCJjZCAjBADggIAYIgAAQIADy4IAB4gsCir67EvCi4gAMYR1ZGyPlNpY1om3JvY2ZQQRSSlLTHqfZ2zC4AAAanU1AiaAued-MDjNrrZecYHF14R1GAIrPuqiCyrHyhNQWuI8QfUBAABAAA
Bi5GADBEAIdd4-pzmEWykCWeGdtBQLIXt5wL3e7jE1C1YrQ_tQ1gjwONMIQXOGHd2M6e12boGfw_Jfow_vu5z
p-VxAlFo"
  }
]
```

EXAMPLE 13: UAF Authentication Response for text/plain Transaction
```json
[
  {
    "header": {
      "upv": {
        "major": 1,
        "minor": 2
      },
      "op": "Auth",
      "appId": "https://uaf.example.com/facets.json",
      "serverData": "mzUSKHLDsd_StbdDIN3aW6x3yNMPyPyp2Qz3-Y"
    },
    "fcParams": "eyJmYWNndE1EIjoiaHR0cMN6Ly91YVYuZXhhbXBza2551b20iLChJci6JZCJjZCAjBADggIAYIgAAQIADy4IAB4gsCir67EvCi4gAMYR1ZGyPlNpY1om3JvY2ZQQRSSlLTHqfZ2zC4AAAanU1AiaAued-MDjNrrZecYHF14R1GAIrPuqiCyrHyhNQWuI8QfUBAABAAA
Bi5GADBEAIdd4-pzmEWykCWeGdtBQLIXt5wL3e7jE1C1YrQ_tQ1gjwONMIQXOGHd2M6e12boGfw_Jfow_vu5z
p-VxAlFo"
  }
]
```

NOTE
Line breaks in fcParams have been inserted for improving readability.

3.5.7 Authentication Processing Rules

3.5.7.1 Authentication Request Generation Rules for FIDO Server

The policy contains a 2-dimensional array of allowed MatchCriteria (see Policy). This array can be considered a list (first dimension) of sets (second dimension) of authenticators (identified by MatchCriteria). All authenticators in a specific set must be used for authentication simultaneously in order to match the policy. But any of those sets in the list are valid, i.e. the list elements are alternatives.

The FIDO Server must follow the steps:

1. Construct appropriate authentication policy
   1. for each set of alternative authenticators do
      1. Create an 1-dimensional array of MatchCriteria objects containing the set of authenticators to be used for authentication simultaneously that need to be identified by separate MatchCriteria objects.
      1. For each collection of authenticators to be used for authentication simultaneously that can be identified by the same rule, create a MatchCriteria object, where
m.aaid MAY be combined with (one or more of) m.keyIDs, m.attachmentHint, m.authenticatorVersion, and m.exts, but m.aaid MUST NOT be combined with any other match criteria field.

- If m.aaid is not provided - both m.authenticationAlgorithms and m.assertionSchemes MUST be provided
- In case of step-up authentication (i.e. in the case where it is expected the user is already known due to a previous authentication step) every item in Policy.accepted MUST include the AAID and KeyID of the authenticator registered for this account in order to avoid ambiguities when having multiple accounts at this relying party.

2. Add m to v, e.g. v[j+1]=m.
   2. Add v to p.allowed, e.g. p.allowed[i+1]=v
2. Create MatchCriteria objects m[] for all disallowed authenticators.
   1. Create a MatchCriteria object m and add AAIDs of all disallowed authenticators to m.aaid.
      The status (as provided in the metadata TOC [FIDOMetadataService]) of some authenticators might be unacceptable. Such authenticators SHOULD be included in p.disallowed.
   2. If needed - create MatchCriteria m for other disallowed criteria (e.g. unsupported authenticationAlgs)
   3. Add all m to p.disallowed.
2. Create an AuthenticationRequest object r with appropriate r.header for the supported version, and
   1. FIDO Servers SHOULD NOT assume any implicit integrity protection of r.header.serverData. FIDO Servers that depend on the integrity of r.header.serverData SHOULD apply and verify a cryptographically secure Message Authentication Code (MAC) to serverData and they SHOULD also cryptographically bind serverData to the related message, e.g. by re-including r.challenge, see also section ServerData and KeyHandle.

   NOTE
   All other FIDO components (except the FIDO server) will treat r.header.serverData as an opaque value. As a consequence the FIDO server can implement any suitable cryptographic protection method.

2. Generate a random challenge and assign it to r.challenge
3. If this is a transaction confirmation operation - look up TransactionConfirmationDisplayContentTypes/TransactionConfirmationDisplayPNGCharacteristics from authenticator metadata of every participating AAID, generate a list of corresponding transaction content and insert the list into r.transaction.
   - If the authenticator reported (a dynamic) AuthenticatorRegistrationAssertion.tcDisplayPNGCharacteristics during Registration - it MUST be preferred over the (static) value specified in the authenticator Metadata.
4. Set r.policy to our new policy object p created above, e.g. r.policy = p.
5. Add the authentication request message the array
3. Send the array of authentication request messages to the FIDO UAF Client

3.5.7.2 Authentication Request Processing Rules for FIDO UAF Client

The FIDO UAF Client MUST follow the steps:

1. Choose the message m with upv set to the appropriate version number.
2. Parse the message m
   - If a mandatory field in the UAF message is not present or a field doesn't correspond to its type and value then reject the operation
3. Obtain FacetID of the requesting Application. If the AppID is missing or empty, set the AppID to the FacetID.
   Verify that the FacetID is authorized for the AppID according to the algorithms in [FIDOAppIDAndFacets].
   - If the FacetID of the requesting Application is not authorized, reject the operation
4. Filter available authenticators with the given policy and present the filtered list to User.
5. Let the user select the preferred Authenticator.
6. Obtain TLS data if its available
   - FinalChallenge = base64url(serialize(utf8encode(fcp)))
8. For each authenticator that supports an Authenticator Interface Version AIV compatible with message version AuthenticationRequest.header.upv (see Version Negotiation) and user agrees to authenticate with:
1. Add **AppID**, **FinalChallenge**, **Transactions** (if present), and all other fields to the ASMRequest.
2. Send the ASMRequest to the ASM. If the ASM returns an error, handle that error appropriately. The status code returned by the ASM [UAFASM] must be mapped to a status code defined in [UAFAppAPIAndTransport] as specified in section 3.4.6.2.1 Mapping ASM Status Codes to ErrorCode.

### 3.5.7.3 Authentication Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section "Sign Command".

### 3.5.7.4 Authentication Response Generation Rules for FIDO UAF Client

The FIDO UAF Client **MUST** follow the steps:

1. Create an AuthenticationResponse message
2. Copy `AuthenticationRequest.header` into `AuthenticationResponse.header`

**NOTE**

When the **appID** provided in the request was empty, the FIDO Client must set the **appID** in this header to the facetID (see [FIDOAppIDAndFacets]).

The header might include extensions. Extension specific rules might affect the copy process. Those rules are defined in the related section in [UAFRegistry].

3. Fill out `AuthenticationResponse.FinalChallengeParams` with appropriate fields and then stringify it
4. Append the response from each authenticator into `AuthenticationResponse.assertions`
5. Send AuthenticationResponse message to the FIDO Server

### 3.5.7.5 Authentication Response Processing Rules for FIDO Server

**NOTE**

The following processing rules assume that authenticator supports "UAFV1TLV" assertion scheme. Currently "UAFV1TLV" is the only defined and supported assertion scheme. When a new assertion scheme is added to UAF protocol - this section will be extended with corresponding processing rules.

The FIDO Server **MUST** follow the steps:

1. Parse the message
   1. If protocol version (`AuthenticationResponse.header.upv`) is not supported – reject the operation
   2. If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
2. Verify that `AuthenticationResponse.header.serverData`, if used, passes any implementation-specific checks against its validity. See also section ServerData and KeyHandle.
3. base64url decode `AuthenticationResponse.fcParams` and convert into an object (`fcp`)
4. If this `fcp` object is a `FinalChallengeParams` object, then verify each field in `fcp` and make sure it's valid:
   1. Make sure `fcp.appID` corresponds to the one stored by the FIDO Server

**NOTE**

When the **appID** provided in the request was empty, the FIDO Client must set the **appID** to the facetID (see [FIDOAppIDAndFacets]). In this case, the Uauth key cannot be used by other application facets.

2. Make sure `fcp.facetID` is in the list of trusted FacetIDs [FIDOAppIDAndFacets]
3. Make sure `ChannelBinding` is as expected (see section ChannelBinding dictionary)
4. Make sure \texttt{fcp.challenge} has really been generated by the FIDO Server for this operation and it is not expired.

5. Reject the response if any of the above checks fails.

5. If this \texttt{fcp} object is a \texttt{CollectedClientData} object, then verify each field in \texttt{fcp} and make sure it's valid:
   1. Make sure \texttt{fcp.origin} is considered a legitimate origin for this registration request.
   2. Make sure \texttt{fcp.tokenBinding} is as expected (see field \texttt{cid_pubkey} in section \texttt{ChannelBinding} dictionary).

### NOTE
There might be legitimate situations in which some methods of channel binding fail (see section 4.3.4 TLS Binding).

3. Make sure \texttt{fcp.challenge} has really been generated by the FIDO Server for this operation and it is not expired.

4. Reject the response if any of the above checks fails.

6. For each assertion \texttt{a} in \texttt{AuthenticationResponse.assertions}:
   1. Parse data from \texttt{a.assertion} assuming it is encoded according to the suspected assertion scheme \texttt{a.assertionScheme} and make sure it contains all mandatory fields (indicated in authenticator Metadata) it is supposed to have, verify that the assertion has a valid syntax, and verify that the assertion doesn't include unknown fields (identified by TAGs or IDs) that belong to extensions marked as "fail-if-unknown" set to true [\texttt{FIDOMetadataStatement}].
     - If it doesn't - continue with next assertion.
   2. if \texttt{a.assertion} contains an object of type \texttt{TAG_UAFV1_AUTH_ASSERTION}, then:
     1. if \texttt{a.assertion.TAG_UAFV1_AUTH_ASSERTION} contains \texttt{TAG_UAFV1_SIGNED_DATA} as first element:
        1. Retrieve the AAID from the assertion.
   3. Make sure that the AAID indeed matches the policy of the Authentication Request.
     - If it doesn't meet the policy – continue with next assertion.
   4. Obtain \texttt{Metadata(AAID).AuthenticatorVersion} for this AAID and make sure that it is lower or equal to \texttt{a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_ASSERTION_INFO.AuthenticatorVersion}.
     - If \texttt{Metadata(AAID).AuthenticatorVersion} is higher (i.e. the authenticator firmware is outdated), it is RECOMMENDED to assume increased authentication risk. See "StatusReport dictionary" and "Metadata TOC object Processing Rules" in [\texttt{FIDOMetadataService}] for more details on this.
   5. Retrieve \texttt{a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_AAID} as KeyID.
   6. Locate \texttt{UAuth.pub} public key associated with (AAID, KeyID) in the user's record.
     - If such record doesn't exist - continue with next assertion.
   7. Verify the AAID against the AAID stored in the user's record at time of Registration.
     - If comparison fails – continue with next assertion.
   8. Locate authenticator specific authentication algorithms from authenticator metadata (field \texttt{AuthenticationAlgs})
   9. Check the Signature Counter \texttt{a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.SignCounter} and make sure it is either not supported by the authenticator (i.e. the value provided and the value stored in the user's record are both 0 or the value isKeyRestricted is set to 'false' in the related Metadata Statement) or it has been incremented (compared to the value stored in the user's record).
     - If it is greater than 0, but didn't increment - continue with next assertion (as this is a cloned authenticator or a cloned authenticator has been used previously).
   10. If \texttt{fcp} is of type \texttt{FinalChallengeParams}, then hash \texttt{AuthenticationResponse.FinalChallengeParams} using the hashing algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator Metadata, field \texttt{AuthenticationAlgs}. It is the hash algorithm associated with the first entry related to a constant with prefix ALG_SIGN.

\[ FCHash = \text{hash(AuthenticationResponse.FinalChallengeParams)} \]
11. If `is` of type `CollectedClientData`, then hash `fcp.hashAlg`
    - `FHash = hash(AuthenticationResponse.fcParams)`
12. Make sure that `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_FINAL_CHALLENGE_HASH == FHash`
    - If comparison fails – continue with next assertion
13. If `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_ASSERTION_INFO.authenticationMode == 2`

**NOTE**

The transaction hash included in this `AuthenticationResponse` must match the transaction content specified in the related `AuthenticationRequest`. As FIDO doesn’t mandate any specific FIDO Server API, the transaction content could be cached by any relying party software component, e.g. the FIDO Server or the relying party Web Application.

1. Make sure there is a transaction cached on Relying Party side.
   - If not – continue with next assertion
2. Go over all cached forms of the transaction content (potentially multiple cached PNGs for the same transaction) and calculate their hashes using hashing algorithm suitable for this authenticator (same hash algorithm as used for `FinalChallenge`).
   - For each `cachedTransaction` add `hash(cachedTransaction)` into `cachedTransactionHashList`
3. Make sure that `a.TransactionHash` is in `cachedTransactionHashList`
   - If it’s not in the list – continue with next assertion
14. Use `UAuth.pub` key and appropriate authentication algorithm to verify
   - `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_SIGNATURE`
     1. If signature verification fails – continue with next assertion
     2. Update `SignCounter` in user’s record with
        - `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.SignCounter`
   2. if `a.assertion.TAG_UAFV1_AUTH_ASSERTION` contains a different object than `TAG_UAFV1_SIGNED_DATA` as first element, then follow the rules specific to that object.
3. if `a.assertion` doesn’t contain an object of type `TAG_UAFV1_AUTH_ASSERTION`, then follow the respective processing rules of that assertion format if supported - otherwise skip this assertion.
4. Treat this assertion `a` as positively verified.


### 3.6 Deregistration Operation

This operation allows FIDO Server to ask the FIDO Authenticator to delete keys related to the particular relying party.

The FIDO Server **MAY** explicitly enumerate the keys to be deleted, or the FIDO server **MAY** signal deregistration of all keys on all authenticators managed by the FIDO UAF Client and relating to a given appID.

**NOTE**

There are various deregistration use cases that both FIDO Server and FIDO Client implementations should allow for. Two in particular are:

1. FIDO Servers should trigger this operation in the event a user removes their account at the relying party.
2. FIDO Clients should ensure that relying party application facets -- e.g., mobile apps, web pages -- have means to initiate a deregistration operation without having necessarily received a UAF protocol message with an `op` value of “Dereg”. This allows the relying party app facet to remove a user's keys from authenticators during events such as relying party app removal or installation.

### 3.6.1 Deregistration Request Message

The FIDO UAF Deregistration request message is represented as an array of dictionaries. The array **MUST** contain exactly one dictionary. The request is defined as `DeregistrationRequest` dictionary.

**EXAMPLE 14: UAF Deregistration Request**

```json
[
  {
    "header": {
```
The example above contains a deregistration request. This request will deregister the key with the specified keyID registered for the authenticator with aaid "FFFF#FC03" for the given appID.

NOTE
There is no deregistration response object.

3.6.2 DeregisterAuthenticator dictionary

```webidl
dictionary DeregisterAuthenticator {
  required AAID aaid;
  required KeyID keyID;
};
```

3.6.2.1 Dictionary DeregisterAuthenticator Members

- **aaid** of type **required AAID**
  AAID of the authenticator housing the UAuth.priv key to deregister, or an empty string if all keys related to the specified appID are to be de-registered.

- **keyID** of type **required KeyID**
  The unique KeyID related to UAuth.priv. KeyID is assumed to be unique within the scope of an AAID only. If aaid is not an empty string, then:
  1. keyID MAY contain a value of type KeyID, or,
  2. keyID MAY be an empty string.

(1) signals deletion of a particular UAuth.priv key mapped to the (AAID, KeyID) tuple.

(2) signals deletion of all KeyIDs associated with the specified aaid.

If aaid is an empty string, then keyID MUST also be an empty string. This signals deregistration of all keys on all authenticators that are mapped to the specified appID.

3.6.3 DeregistrationRequest dictionary

```webidl
dictionary DeregistrationRequest {
  required OperationHeader header;
  required DeregisterAuthenticator[] authenticators;
};
```

3.6.3.1 Dictionary DeregistrationRequest Members

- **header** of type **required OperationHeader**
  Header.op MUST be "Dereg".

- **authenticators** of type array of **required DeregisterAuthenticator**
  List of authenticators to be deregistered.

3.6.4 Deregistration Processing Rules
3.6.4.1 Deregistration Request Generation Rules for FIDO Server

The FIDO Server MUST follow the steps:

1. Create a DeregistrationRequest message \( m \) with \( m.header.upv \) set to the appropriate version number.
2. If the FIDO Server intends to deregister all keys on all authenticators managed by the FIDO UAF Client for this appID, then:
   1. create one and only one DeregisterAuthenticator object \( o \)
   2. Set \( o.aaid \) and \( o.keyID \) to be empty string values
   3. Append \( o \) to \( m.authenticators \), and go to step 5
3. If the FIDO Server intends to deregister all keys on all authenticators with a given AAID managed by the FIDO UAF Client for this appID, then:
   1. create one and only one DeregisterAuthenticator object \( o \)
   2. Set \( o.aaid \) to the intended AAID and set \( o.keyID \) to be an empty string.
   3. Append \( o \) to \( m.authenticators \), and go to step 5
4. Otherwise, if the FIDO Server intends to deregister specific \((\text{AAID}, \text{KeyID})\) tuples, then for each tuple to be deregistered:
   1. create a DeregisterAuthenticator object \( o \)
   2. Set \( o.aaid \) and \( o.keyID \) appropriately
   3. Append \( o \) to \( m.authenticators \)
5. delete related entry (or entries) in FIDO Server's account database
6. Send message to FIDO UAF Client

3.6.4.2 Deregistration Request Processing Rules for FIDO UAF Client

The FIDO UAF Client MUST follow the steps:

1. Choose the message \( m \) with \( upv \) set to the appropriate version number.
2. Parse the message
   - If a mandatory field in DeregistrationRequest message is not present or a field doesn't correspond to its type and value – reject the operation
   - Empty string values for \( o.aaid \) and \( o.keyID \) MUST occur in the first and only DeregisterAuthenticator object \( o \), otherwise reject the operation
3. Obtain FacetID of the requesting Application. If the AppID is missing or empty, set the AppID to the FacetID.
   - Verify that the FacetID is authorized for the AppID according to the algorithms in [FIDOAppIDAndFacets].
   - If the FacetID of the requesting Application is not authorized, reject the operation
4. If the set of authenticators compatible with the message version DeregistrationRequest.header.upv and having an AAID matching one of the provided AAIDs (an AAID of an authenticator matches if it is either (a) equal to one of the AAIDs in the DeregistrationRequest or if (b) the AAID in the DeregistrationRequest is an empty string) is empty, then return NO_SUITABLE_AUTHENTICATOR.
5. For each authenticator compatible with the message version DeregistrationRequest.header.upv and having an AAID matching one of the provided AAIDs (an AAID of an authenticator matches if it is either (a) equal to one of the AAIDs in the DeregistrationRequest or if (b) the AAID in the DeregistrationRequest is an empty string):
   1. Create appropriate ASMRequest for Deregister function and send it to the ASM. If the ASM returns an error, handle that error appropriately. The status code returned by the ASM [UAFASM] must be mapped to a status code defined in [UAFAppAPIAndTransport] as specified in section 3.6.2.1 Mapping ASM Status Codes to ErrorCode.

3.6.4.3 Deregistration Request Processing Rules for FIDO Authenticator

See [UAFASM] section "Deregister request".

4. Considerations

This section is non-normative.

4.1 Protocol Core Design Considerations

This section describes the important design elements used in the protocol.
4.1.1 Authenticator Metadata

It is assumed that FIDO Server has access to a list of all supported authenticators and their corresponding Metadata. Authenticator metadata [FIDOMetadataStatement] contains information such as:

- Supported Registration and Authentication Schemes
- Authentication Factor, Installation type, supported content-types and other supplementary information, etc.

In order to make a decision about which authenticators are appropriate for a specific transaction, FIDO Server looks up the list of authenticator metadata by AAID and retrieves the required information from it.

NORMATIVE

Each entry in the authenticator metadata repository MUST be identified with a unique authenticator Attestation ID (AAID).

4.1.2 Authenticator Attestation

Authenticator Attestation is the process of validating authenticator model identity during registration. It allows Relying Parties to cryptographically verify that the authenticator reported by FIDO UAF Client is really what it claims to be.

Using authenticator Attestation, a relying party "example-rp.com" will be able to verify that the authenticator model of the "example-Authenticator", reported with AAID "1234#5678", is not malware running on the FIDO User Device but is really a authenticator of model "1234#5678".

NORMATIVE

FIDO Authenticators SHOULD support "Basic Attestation" or "ECDAA" described below. New Attestation mechanisms may be added to the protocol over time.

NORMATIVE

FIDO Authenticators not providing sufficient protection for Attestation keys (non-attested authenticators) MUST use the UAuth.priv key in order to formally generate the same KeyRegistrationData object as attested authenticators. This behavior MUST be properly declared in the Authenticator Metadata.

4.1.2.1 Basic Attestation

NORMATIVE

There are two different flavors of Basic Attestation:

**Full Basic Attestation**

Based on an attestation private key shared among a class of authenticators (e.g. same model).

**Surrogate Basic Attestation**

Just syntactically a Basic Attestation. The attestation object self-signed, i.e. it is signed using the UAuth.priv key, i.e. the key corresponding to the UAuth.pub key included in the attestation object. As a consequence it does not provide a cryptographic proof of the security characteristics. But it is the best thing we can do if the authenticator is not able to have an attestation private key.

4.1.2.1.1 Full Basic Attestation

NOTE

FIDO Servers must have access to a trust anchor for verifying attestation public keys (i.e. Attestation Certificate trust store) in order to follow the assumptions made in [FIDOSecRef]. Authenticators must provide its attestation signature during the registration process for the same reason. The attestation trust anchor is shared with FIDO Servers out of band (as part of the Metadata). This sharing process should be done according to [FIDOMetadataService].

NOTE
In this Full Basic Attestation model, a large number of authenticators must share the same Attestation certificate and Attestation Private Key in order to provide non-linkability (see Protocol Core Design Considerations). Authenticators can only be identified on a production batch level or an AAID level by their Attestation Certificate, and not individually. A large number of authenticators sharing the same Attestation Certificate provides better privacy, but also makes the related private key a more attractive attack target.

**NOTE**

When using Full Basic Attestation: A given set of authenticators sharing the same manufacturer and essential characteristics must not be issued a new Attestation Key before at least 100,000 devices are issued the previous shared key.

![Attestation Certificate Chain](image)

4.1.2.1.2 **Surrogate Basic Attestation**

**NORMATIVE**

In this attestation method, the UAuth.priv key **MUST** be used to sign the Registration Data object. This behavior **MUST** be properly declared in the Authenticator Metadata.

**NOTE**

FIDO Authenticators not providing sufficient protection for Attestation keys (non-attested authenticators) must use this attestation method.

4.1.2.2 **Direct Anonymous Attestation (ECDAA)**

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

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

Another alternative is the Direct Anonymous Attestation [BriCamChe2004-DAA]. Direct Anonymous Attestation is a cryptographic scheme combining privacy with security. It uses the Authenticator specific secret once to communicate with a single DAA Issuer (either at manufacturing time or after being sold before first use) and uses the resulting DAA credential in the DAA-Sign protocol with each relying party. The (original) DAA scheme has been adopted by the Trusted Computing Group for TPM v1.2 [TPMv1-2-Part1].

ECDAA (see [FIDOEcdaaAlgorithm] for details) is an improved DAA scheme based on elliptic curves and bilinear pairings [CheLi2013-ECDAA]. This scheme provides significantly improved performance compared with the original DAA and it is part of the TPMv2 specification [TPMv2-Part1].

4.1.3 Error Handling

NOTE

FIDO Servers must inform the calling Relying Party Web Application Server (see FIDO Interoperability Overview) about any error conditions encountered when generating or processing UAF messages through their proprietary API.

NORMATIVE

FIDO Authenticators MUST inform the FIDO UAF Client (see FIDO Interoperability Overview) about any error conditions encountered when processing commands through the Authenticator Specific Module (ASM). See [UAFASM] and [UAFAuthnrCommands] for details.

4.1.4 Assertion Schemes

UAF Protocol is designed to be compatible with a variety of existing authenticators (TPMs, Fingerprint Sensors, Secure Elements, etc.) and also future authenticators designed for FIDO. Therefore extensibility is a core capability designed into the protocol.

It is considered that there are two particular aspects that need careful extensibility. These are:

- Cryptographic key provisioning (KeyRegistrationData)
- Cryptographic authentication and signature (SignedData)

The combination of KeyRegistrationData and SignedData schemes is called an Assertion Scheme.

The UAF protocol allows plugging in new Assertion Schemes. See also UAF Supported Assertion Schemes.

The Registration Assertion defines how and in which format a cryptographic key is exchanged between the authenticator and the FIDO Server.

The Authentication Assertion defines how and in which format the authenticator generates a cryptographic signature.

The generally-supported Assertion Schemes are defined in [UAFRegistry].

4.1.5 Username in Authenticator

FIDO UAF supports authenticators acting as first authentication factor (i.e. replacing username and password). As part of the FIDO UAF Registration, the Uauth key is registered (linked) to the related user account at the RP. The authenticator stores the username (allowing the user to select a specific account at the RP in the case he has multiple ones). See [UAFAuthnrCommands], section "Sign Command" for details.

4.1.6 Silent Authenticators

FIDO UAF supports authenticators not requiring any types of user verification or user presence check. Such authenticators are called Silent Authenticators.

In order to meet user's expectations, such Silent Authenticators need specific properties:

- It must be possible for a user to effectively remove a Uauth key maintained by a Silent Authenticator (in order to avoid being tracked) at
the user's discretion (see [UAFAuthnrCommands]). This is not compatible with stateless implementations storing the Uauth private key wrapped inside a KeyHandle on the FIDO Server.

- TransactionConfirmation is not supported (as it would require user input which is not intended), see [UAFAuthnrCommands].
- They might not operate in first factor mode (see [UAFAuthnrCommands]) as this might violate the privacy principles.

The MetadataStatement has to truthfully reflect the Silent Authenticator, i.e. field userVerification needs to be set to USER_VERIFY_NONE.

4.1.7 TLS Protected Communication

**NOTE**

In order to protect the data communication between FIDO UAF Client and FIDO Server a protected TLS channel must be used by FIDO UAF Client (or User Agent) and the Relying Party for all protocol elements.

1. The server endpoint of the TLS connection must be at the Relying Party
2. The client endpoint of the TLS connection must be either the FIDO UAF Client or the User Agent / App
3. TLS Client and Server should use TLS v1.2 or newer and should only use TLS v1.1 if TLS v1.2 or higher are not available. The "anon" and "null" TLS crypto suites are not allowed and must be rejected; insecure crypto-algorithms in TLS (e.g. MD5, RC4, SHA1) should be avoided [SP800-131A] [RFC7525].
4. TLS Extended Master Secret Extension [RFC7627] and TLS Renegotiation Indication Extension [RFC5746] should be used to protect against MITM attacks.
5. The use of the tls-unique method is deprecated as its security is broken, see [TLSAUTH].

We recommend, that the

1. TLS Client verifies and validates the server certificate chain according to [RFC5280], section 6 "Certificate Path Validation". The certificate revocation status should be checked (e.g. using OCSP [RFC2560] or CRL based validation [RFC5280]) and the TLS server identity should be checked as well [RFC6125].
2. TLS Client's trusted certificate root store is properly maintained and at least requires the CAs included in the root store to annually pass Web Trust or ETSI (ETSI TS 101 456, or ETSI TS 102 042) audits for SSL CAs.

See [TR-03116-4] and [SHEFFER-TLS] for more recommendations on how to use TLS.

4.2 Implementation Considerations

4.2.1 Server Challenge and Random Numbers

**NOTE**

A ServerChallenge needs appropriate random sources in order to be effective (see [RFC4086] for more details). The (pseudo-)random numbers used for generating the Server Challenge should successfully pass the randomness test specified in [Coron99] and they should follow the guideline given in [SP800-90b].

4.2.2 Revealing KeyIDs

FIDO UAF uses key identifiers (KeyIDs) to identify Uauth keys registered by an authenticator to a relying party. By design (see [UAFAuthnrCommands], section 6.2.4), KeyIDs do not reveal any secret information. However, if an attacker could provide a username to a relying party and the relying party server would reveal the related KeyID if an account for that username exists or give an error otherwise, the attacker would implicitly learn whether the user has an account at that relying party.

As a consequence, relying parties should reveal a KeyID only after performing some basic authentication steps, e.g. verifying the existence of a Cookie, authentication using FIDO Silent Authenticator, etc.).

4.3 Security Considerations

There is no "one size fits all" authentication method. The FIDO goal is to decouple the user verification method from the authentication protocol and the authentication server, and to support a broad range of user verification methods and a broad range of assurance levels. FIDO authenticators should be able to leverage capabilities of existing computing hardware, e.g. mobile devices or smart cards.

The overall assurance level of electronic user authentications highly depends (a) on the security and integrity of the user's equipment involved and (b) on the authentication method being used to authenticate the user.
When using FIDO, users should have the freedom to use any available equipment and a variety of authentication methods. The relying party needs reliable information about the security relevant parts of the equipment and the authentication method itself in order to determine whether the overall risk of an electronic authentication is acceptable in a particular business context. The FIDO Metadata Service [FIDOMetadataService] is intended to provide such information.

It is important for the UAF protocol to provide this kind of reliable information about the security relevant parts of the equipment and the authentication method itself to the FIDO server.

The overall security is determined by the weakest link. In order to support scalable security in FIDO, the underlying UAF protocol needs to provide a very high conceptual security level, so that the protocol isn’t the weakest link.

**Relying Parties define Acceptable Assurance Levels.** The FIDO Alliance envisions a broad range of FIDO UAF Clients, FIDO Authenticators and FIDO Servers to be offered by various vendors. Relying parties should be able to select a FIDO Server providing the appropriate level of security. They should also be in a position to accept FIDO Authenticators meeting the security needs of the given business context, to compensate assurance level deficits by adding appropriate implicit authentication measures, and to reject authenticators not meeting their requirements. FIDO does not mandate a very high assurance level for FIDO Authenticators, instead it provides the basis for authenticator and user verification method competition.

**Authentication vs. Transaction Confirmation.** Existing Cloud services are typically based on authentication. The user launches an application (i.e. User Agent) assumed to be trusted and authenticates to the Cloud service in order to establish an authenticated communication channel between the application and the Cloud service. After this authentication, the application can perform any actions to the Cloud service using the authenticated channel. The service provider will attribute all those actions to the user. Essentially the user authenticates all actions performed by the application in advance until the service connection or authentication times out. This is a very convenient way as the user doesn’t get distracted by manual actions required for the authentication. It is suitable for actions with low risk consequences.

However, in some situations it is important for the relying party to know that a user really has seen and accepted a particular content before he authenticates it. This method is typically being used when non-repudiation is required. The resulting requirement for this scenario is called What You See Is What You Sign (WYSIWYS).

UAF supports both methods; they are called "Authentication" and "Transaction Confirmation". The technical difference is, that with Authentication the user confirms a random challenge, where in the case of Transaction Confirmation the user also confirms a human readable content, i.e. the contract. From a security point, in the case of authentication the application needs to be trusted as it performs any action once the authenticated communication channel has been established. In the case of Transaction Confirmation only the transaction confirmation display component implementing WYSIWYS needs to be trusted, not the entire application.

**Distinct Attestable Security Components.** For the relying party in order to determine the risk associated with an authentication, it is important to know details about some components of the user’s environment. Web Browsers typically send a "User Agent" string to the web server. Unfortunately any application could send any string as "User Agent" to the relying party. So this method doesn’t provide strong security. FIDO UAF is based on a concept of cryptographic attestation. With this concept, the component to be attested owns a cryptographic secret and authenticates its identity with this cryptographic secret. In FIDO UAF the cryptographic secret is called "Authenticator Attestation Key". The relying party gets access to reference data required for verifying the attestation.

In order to enable the relying party to appropriately determine the risk associated with an authentication, all components performing significant security functions need to be attestable.

In FIDO UAF significant security functions are implemented in the "FIDO Authenticators". Security functions are:

1. Protecting the attestation key.
2. Generating and protecting the Authentication key(s), typically one per relying party and user account on relying party.
3. Verifying the user.
4. Providing the WYSIWYS capability ("Transaction Confirmation Display" component).

Some FIDO Authenticators might implement these functions in software running on the FIDO User Device, others might implement these functions in "hardware", i.e. software running on a hardware segregated from the FIDO User Device. Some FIDO Authenticators might even be formally evaluated and accredited to some national or international scheme. Each FIDO Authenticator model has an attestation ID (AAID), uniquely identifying the related security characteristics. Relying parties get access to these security properties of the FIDO Authenticators and the reference data required for verifying the attestation.

**Resilience to leaks from other verifiers.** One of the important issues with existing authentication solutions is a weak server side implementation, affecting the security of authentication of typical users to other relying parties. It is the goal of the FIDO UAF protocol to decouple the security of different relying parties.

**Decoupling User Verification Method from Authentication Protocol.** In order to decouple the user verification method from the authentication protocol, FIDO UAF is based on an extensible set of cryptographic authentication algorithms. The cryptographic secret will be unlocked after user verification by the Authenticator. This secret is then used for the authenticator-to-relying party authentication. The set of cryptographic algorithms is chosen according to the capabilities of existing cryptographic hardware and computing devices. It can be extended in order to support new cryptographic hardware.
Privacy Protection. Different regions in the world have different privacy regulations. The FIDO UAF protocol should be acceptable in all regions and hence must support the highest level of data protection. As a consequence, FIDO UAF doesn't require transmission of biometric data to the relying party nor does it require the storage of biometric reference data [ISOBiometrics] at the relying party. Additionally, cryptographic secrets used for different relying parties shall not allow the parties to link actions to the same user entity. UAF supports this concept, known as non-linkability. Consequently, the UAF protocol doesn't require a trusted third party to be involved in every transaction.

Relying parties can interactively discover the AAIDs of all enabled FIDO Authenticators on the FIDO User Device using the Discovery interface [UAFAppAPIAndTransport]. The combination of AAIDs adds to the entropy provided by the client to relying parties. Based on such information, relying parties can fingerprint clients on the internet (see Browser Uniqueness at eff.org and https://wiki.mozilla.org/Fingerprinting). In order to minimize the entropy added by FIDO, the user can enable/disable individual authenticators – even when they are embedded in the device (see [UAFAppAPIAndTransport], section "privacy considerations").

4.3.1 FIDO Authenticator Security

See [UAFAuthnrCommands].

4.3.2 Cryptographic Algorithms

In order to keep key sizes small and to make private key operations fast enough for small devices, it is suggested that implementers prefer ECDSA [ECDSA-ANSI] in combination with SHA-256 / SHA-512 hash algorithms. However, the RSA algorithm is also supported. See [FIDORegistry] "Authentication Algorithms" and "Public Key Representation Formats" for a list of generally supported cryptographic algorithms.

One characteristic of ECDSA is that it needs to produce, for each signature generation, a fresh random value. For effective security, this value must be chosen randomly and uniformly from a set of modular integers, using a cryptographically secure process. Even slight biases in that process may be turned into attacks on the signature schemes.

NOTE

If such random values cannot be provided under all possible environmental conditions, then a deterministic version of ECDSA should be used (see [RFC6979]).

4.3.3 FIDO Client Trust Model

The FIDO environment on a FIDO User Device comprises 4 entities:

- User Agents (a native app or a browser)
- FIDO UAF Clients (a shared service potentially used by multiple User Agents)
- Authenticator Specific Modules (ASMs)
- Authenticators
The security and privacy principles that underpin mobile operating systems require certain behaviours from apps. FIDO must uphold those principles wherever possible. This means that each of these components has to enforce specific trust relationships with the others to avoid the risk of rogue components subverting the integrity of the solution.

One specific requirement on handsets is that apps originating from different vendors must not be allowed directly to view or edit each other’s data (e.g. FIDO UAF credentials).

Given that FIDO UAF Clients are intended to provide a shared service, the principle of siloed app data has been applied to the FIDO UAF Client, rather than individual apps. This means that if two or more FIDO UAF Clients are present on a device, then each FIDO UAF Client is unable to access authentication keys created by another FIDO UAF Client. A given FIDO UAF Client may however provide services to multiple User Agents, so that the same authentication key can authenticate to different facets of the same Relying Party, even if one facet is a 3rd party browser.

This exclusive access restriction is enforced through the KHAccessToken. When a FIDO UAF Client communicates with an ASM, the ASM reads the identity of the FIDO UAF Client caller1 and includes that Client ID in the KHAccessToken that it sends to the authenticator. Subsequent calls to the authenticator must include the same Client ID in the KHAccessToken. Each authentication key is also bound to the ASM that created it, by means of an ASMToken (a random unique ID for the ASM) that is also included in the KHAccessToken.

Finally, the User Agents that a FIDO UAF Client will recognise are determined by the Relying Party itself. The FIDO UAF Client requests a list of Trusted Apps from the RP as part of the Registration and Authentication protocols. This prevents User Agents that have not been explicitly authorized by the Relying Party from using the FIDO credentials.

In this manner, in a compliant FIDO installation, UAF credentials can only be accessed via apps that the relying party explicitly trusts and through the same client and ASM that performed the original registration.

It should be noted that the specification allows for FIDO UAF Clients to be built directly into User Agents. However, such implementations will restrict the ability to support multiple facets for relying party applications unless they also expose the UAF Client API for other User Agents to consume.

4.3.3.1 Isolation using KHAccessToken
Authenticators might be implemented in dedicated hardware and hence might not be able to verify the calling software entity (i.e. the ASM).

The KHAccessToken allows restricting access to the keys generated by the FIDO Authenticator to the intended ASM. It is based on a Trust On First Use (TOFU) concept.

FIDO Authenticators are capable of binding UAuth.Key with a key provided by the caller (i.e. the ASM). This key is called KHAccessToken. This technique allows making sure that registered keys are only accessible by the caller that originally registered them. A malicious App on a mobile platform won't be able to access keys by bypassing the related ASM (assuming that this ASM originally registered these keys).

The KHAccessToken is typically specific to the AppID, PersonalID, ASMToken and the CallerID. See [UAFASM] for more details.

4.3.4 TLS Binding

Various channel binding methods have been proposed (e.g. [RFC5929] and [ChannelID]).

UAF relies on TLS server authentication for binding authentication keys to AppIDs. There are threats:

1. Attackers might fraudulently get a TLS server certificate for the same AppID as the relying party and they might be able to manipulate the DNS system.
2. Attackers might be able to steal the relying party's TLS server private key and certificate and they might be able to manipulate the DNS system.

And there are functionality requirements:

1. UAF transactions might span across multiple TLS sessions. As a consequence, "tls-unique" defined in [RFC5929] might be difficult to implement.
2. Data centers might use SSL concentrators.
3. Data centers might implement load-balancing for TLS endpoints using different TLS certificates. As a consequence, "tls-server-end-point" defined in [RFC5929], i.e. the hash of the TLS server certificate might be inappropriate.
4. Unfortunately, hashing of the TLS server certificate (as in "tls-server-end-point") also limits the usefulness of the channel binding in a particular, but quite common circumstance. If the client is operated behind a trusted (to that client) proxy that acts as a TLS man-in-the-middle, your client will see a different certificate than the one the server is using. This is actually quite common on corporate or military networks with a high security posture that want to inspect all incoming and outgoing traffic. If the FIDO Server just gets a hash value, there's no way to distinguish this from an attack. If sending the entire certificate is acceptable from a performance perspective, the server can examine it and determine if it is a certificate for a valid name from a non-standard issuer (likely administratively trusted) or a certificate for a different name (which almost certainly indicates a forwarding attack).

See ChannelBinding dictionary for more details.

4.3.5 Session Management

FIDO does not define any specific session management methods. However, several FIDO functions rely on a robust session management being implemented by the relying party's web application:

**FIDO Registration**
A web application might trigger FIDO Registration after authenticating an existing user via legacy credentials. So the session is used to maintain the authentication state until the FIDO Registration is completed.

**FIDO Authentication**
After success FIDO Authentication, the session is used to maintain the authentication state during the operations performed by the user agent or mobile app.

Best practices should be followed to implement robust session management (e.g. [OWASP2013]).

4.3.6 Personas

FIDO supports unlinkability [AnonTerminology] of accounts at different relying parties by using relying party specific keys.

Sometimes users have multiple accounts at a particular relying party and even want to maintain unlinkability between these accounts.

Today, this is difficult and requires certain measures to be strictly applied.

NOTE
On some platforms, the ASM additionally might need special permissions in order to communicate with the FIDO Authenticator. Some platforms do not provide means to reliably enforce access control among applications.
FIDO does not want to add more complexity to maintaining unlinkability between accounts at a relying party.

In the case of roaming authenticators, it is recommended to use different authenticators for the various personas (e.g. "business", "personal"). This is possible as roaming authenticators typically are small and not excessively expensive.

In the case of bound authenticators, this is different. FIDO recommends the "Persona" concept for this situation.

All relevant data in an authenticator are related to one Persona (e.g. "business" or "personal"). Some administrative interface (not standardized by FIDO) of the authenticator may allow maintaining and switching Personas.

**NORMATIVE**

The authenticator **MUST** only "know" / "recognize" data (e.g. authentication keys, usernames, KeyIDs, …) related to the Persona being active at that time.

With this concept, the User can switch to the "Personal" Persona and register new accounts. After switching back to "Business" Persona, these accounts will not be recognized by the authenticator (until the User switches back to "Personal" Persona again).

In order to support the persona feature, the FIDO Authenticator-specific Module API [UAFASM] supports the use of a 'PersonaID' to identify the persona in use by the authenticator. How Personas are managed or communicated with the user is out of scope for FIDO.

4.3.7 ServerData and KeyHandle

Data contained in the field serverData (see Operation Header dictionary) of UAF requests is sent to the FIDO UAF Client and will be echoed back to the FIDO Server as part of the related UAF response message.

**NOTE**

The FIDO Server should not assume any kind of implicit integrity protection of such data nor any implicit session binding. The FIDO Server must explicitly bind the serverData to an active session.

**NOTE**

In some situations, it is desirable to protect sensitive data such that it can be stored in arbitrary places (e.g. in serverData or in the KeyHandle). In such situations, the confidentiality and integrity of such sensitive data must be protected. This can be achieved by using a suitable encryption algorithm, e.g. AES with a suitable cipher mode, e.g. CBC or CTR [CTRMode]. This cipher mode needs to be used correctly. For CBC, for example, a fresh random IV for each encryption is required. The data might have to be padded first in order to obtain an integral number of blocks in length. The integrity protection can be achieved by using a MAC or a digital signature on the ciphertext, using a different key than for the encryption, e.g. using HMAC [FIPS198-1]. Alternatively, an authenticated encryption scheme such as AES-GCM [SP800-38D] or AES-CCM [SP800-38C] could be used. Such a scheme provides both integrity and confidentiality in a single algorithm and using a single key.

**NOTE**

When protecting serverData, the MAC or digital signature computation should include some data that binds the data to its associated message, for example by re-including the challenge value in the authenticated serverData.

4.3.8 Authenticator Information retrieved through UAF Application API vs. Metadata

Several authenticator properties (e.g. UserVerificationMethods, KeyProtection, TransactionConfirmationDisplay, ...) are available in the metadata [FIDOMetadataStatement] and through the FIDO UAF Application API. The properties included in the metadata are authoritative and are provided by a trusted source. When in doubt, decisions should be based on the properties retrieved from the Metadata as opposed to the data retrieved through the FIDO UAF Application API.

However, the properties retrieved through the FIDO UAF Application API provide a good "hint" what to expect from the Authenticator. Such "hints" are well suited to drive and optimize the user experience.

4.3.9 Policy Verification

FIDO UAF Response messages do not include all parameters received in the related FIDO UAF request message into the to-be-signed object. As a consequence, any MITM could modify such entries.

FIDO Server will detect such changes if the modified value is unacceptable.
For example, a MITM could replace a generic policy by a policy specifying only the weakest possible FIDO Authenticator. Such a change will be detected by FIDO Server if the weakest possible FIDO Authenticator does not match the initial policy (see Registration Response Processing Rules and Authentication Response Processing Rules).

### 4.3.10 Replay Attack Protection

The FIDO UAF protocol specifies two different methods for replay-attack protection:

1. Secure transport protocol (TLS)
2. Server Challenge.

The TLS protocol by itself protects against replay-attacks when implemented correctly [TLS].

Additionally, each protocol message contains some random bytes in the `ServerChallenge` field. The FIDO server should only accept incoming FIDO UAF messages which contain a valid `ServerChallenge` value. This is done by verifying that the `ServerChallenge` value, sent by the client, was previously generated by the FIDO server. See FinalChallengeParams.

It should also be noted that under some (albeit unlikely) circumstances, random numbers generated by the FIDO server may not be unique, and in such cases, the same `ServerChallenge` may be presented more than once, making a replay attack harder to detect.

### 4.3.11 Protection against Cloned Authenticators

FIDO UAF relies on the UAuth.Key to be protected and managed by an authenticator with the security characteristics specified for the model (identified by the AAID). The security is better when only a single authenticator with that specific UAuth.Key instance exists. Consequently FIDO UAF specifies some protection measures against cloning of authenticators.

Firstly, if the UAuth private keys are protected by appropriate measures then cloning should be hard as such keys cannot be extracted easily.

Secondly, UAF specifies a Signature Counter (see Authentication Response Processing Rules and [UAFAuthnrCommands]). This counter is increased by every signature operation. If a cloned authenticator is used, then the subsequent use of the original authenticator would include a signature counter lower to or equal to the previous (malicious) operation. Such an incident can be detected by the FIDO Server.

### 4.3.12 Anti-Fraud Signals

There is the potential that some attacker misuses a FIDO Authenticator for committing fraud, more specifically they would:

1. Register the authenticator to some relying party for one account
2. Commit fraud
3. Deregister the Authenticator
4. Register the authenticator to some relying party for another account
5. Commit fraud
6. Deregister the Authenticator
7. and so on...

**NOTE**

Authenticators might support a Registration Counter (`RegCounter`). The `RegCounter` will be incremented on each registration and hence might become exceedingly high in such fraud scenarios. See [UAFAuthnrCommands] for more details.

### 4.4 Interoperability Considerations

FIDO supports Web Applications, Mobile Applications and Native PC Applications. Such applications are referred to as FIDO enabled applications.
Web applications typically consist of the web application server and the related Web App. The Web App code (e.g. HTML and JavaScript) is rendered and executed on the client side by the User Agent. The Web App code talks to the User Agent via a set of JavaScript APIs, e.g. HTML DOM. The FIDO DOM API is defined in [UAFAppAPIAndTransport]. The protocol between the Web App and the Relying Party Web Application Server is typically proprietary.

Mobile Apps play the role of the User Agent and the Web App (Client). The protocol between the Mobile App and the Relying Party Web Application Server is typically proprietary.

Native PC Applications play the role of the User Agent, the Web App (Client). Those applications are typically expected to be independent from any particular Relying Party Web Application Server.

It is recommended for FIDO enabled applications to use the FIDO messages according to the format specified in this document.

It is recommended for FIDO enabled application to use the UAF HTTP Binding defined in [UAFAppAPIAndTransport].

NOTE

The KeyRegistrationData and SignedData objects [UAFAuthnrCommands] are generated and signed by the FIDO Authenticators and have to be verified by the FIDO Server. Verification will fail if the values are modified during transport.

The ASM API [UAFASM] specifies the standardized API to access authenticator Specific Modules (ASMs) on Desktop PCs and Mobile Devices.

The document [UAFAuthnrCommands] does not specify a particular protocol or API. Instead it lists the minimum data set and a specific message format which needs to be transferred to and from the FIDO Authenticator.

5. UAF Supported Assertion Schemes

This section is normative.

5.1 Assertion Scheme "UAFV1TLV"

This scheme is mandatory to implement for FIDO Servers. This scheme is mandatory to implement for FIDO Authenticators.

This Assertion Scheme allows the authenticator and the FIDO Server to exchange an asymmetric authentication key generated by the Authenticator.

This assertion scheme is using Tag Length Value (TLV) compact encoding to encode registration and authentication assertions generated by
authenticators. This is the default assertion scheme for UAF protocol.

TAGs and Algorithms are defined in [UAFRegistry].

The authenticator **must** use a dedicated key pair (UAuth.pub/UAuth.priv) suitable for the authentication algorithm specified in the metadata statement [FIDOMetadataStatement] for each relying party. This key pair **should** be generated as part of the registration operation.

Conforming FIDO Servers **must** implement all authentication algorithms and key formats listed in document [FIDORegistry] unless they are explicitly marked as optional in [FIDORegistry].

Conforming FIDO Servers **must** implement all attestation types (TAG_ATTESTATION_*) listed in document [UAFRegistry] unless they are explicitly marked as optional in [UAFRegistry].

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].

### 5.1.1 KeyRegistrationData

See [UAFAuthnrCommands], section "TAG_UAFV1_KRD".

### 5.1.2 SignedData

See [UAFAuthnrCommands], section "TAG_UAFV1_SIGNED_DATA".

### 6. Definitions

See [FIDOGlossary].

### 7. Table of Figures

- Fig. 1 The UAF Architecture
- Fig. 2 UAF Registration Message Flow
- Fig. 3 Authentication Message Flow
- Fig. 4 Transaction Confirmation Message Flow
- Fig. 5 Deregistration Message Flow
- Fig. 6 UAF Registration Sequence Diagram
- Fig. 7 UAF Registration Cryptographic Data Flow
- Fig. 8 UAF Authentication Sequence Diagram
- Fig. 9 UAF Authentication Cryptographic Data Flow
- Fig. 10 Attestation Certificate Chain
- Fig. 11 UAF Client Trust Model
- Fig. 12 FIDO Interoperability Overview

### A. References

#### A.1 Normative references

[ABNF]  

[ChannelID]  

[Coron99]  

[FIDOAppIDAndFacets]  

[FIDOEcdaaAlgorithm]  

[FIDOGlossary]  
A.2 Informative references

[AnonTerminology]

[BriCamChe2004-DAA]

[CTRMode]

[CheLi2013-ECDAA]
Liqun Chen; Jiangtao Li. Flexible and Scalable Digital Signatures in TPM 2.0. 2013. URL: http://dx.doi.org/10.1145/2508859.2516729

[CTRMode]

[CryptoBiometrics]

[OWASP2013]

[RFC2560]

[RFC5746]

[RFC6125]

[RFC6287]

[RFC6454]

[RFC7525]

[RFC7627]

[SHEFFER-TLS]

[SP800-131A]

[SP800-38C]

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

[SP800-63]

[TLS]
FIDO UAF Registry of Predefined Values

FIDO Alliance Proposed Standard 20 October 2020

This version: https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-reg-v1.2-ps-20201020.html
Previous version: https://fidoalliance.org/specs/fido-uaf-v1.2-id-20180220/fido-uaf-reg-v1.2-id-20180220.html

Editor: Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Contributors: Davit Baghdasaryan, Nok Nok Labs, Inc., Brad Hill, PayPal

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

Copyright © 2013-2020 FIDO Alliance All Rights Reserved.

Abstract

This document defines all the strings and constants reserved by UAF protocols. The values defined in this document are referenced by various UAF specifications.

Status of This Document

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

1.1 Key Words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

This document defines the registry of UAF-specific constants that are used and referenced in various UAF specifications. It is expected that, over time, new constants will be added to this registry. For example new authentication algorithms and new types of authenticator characteristics will require new constants to be defined for use within the specifications.

FIDO-specific constants that are common to multiple protocol families are defined in [FIDO Registry].

3. Authenticator Characteristics
This section is normative.

3.1 Assertion Schemes

Names of assertion schemes are strings with a length of 8 characters.

UAF TLV based assertion scheme “UAFV1TLV”
This assertion scheme allows the authenticator and the FIDO Server to exchange an asymmetric authentication key generated by the authenticator. The authenticator must generate a key pair (UAuth.pub/UAuth.priv) to be used with algorithm suites listed in [FIDORegistry] section "Authentication Algorithms" (with prefix ALG_). This assertion scheme is using a compact Tag Length Value (TLV) encoding for the KRD and SignData messages generated by the authenticators. This is the default assertion scheme for the UAF protocol.

4. Predefined Tags

This section is normative.

The internal structure of UAF authenticator commands is a “Tag-Length-Value” (TLV) sequence. The tag is a 2-byte unique unsigned value describing the type of field the data represents, the length is a 2-byte unsigned value indicating the size of the value in bytes, and the value is the variable-sized series of bytes which contain data for this item in the sequence.

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.

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.

A tag that has the 13th bit (0x1000) set indicates a composite tag that can be parsed by recursive descent.

4.1 Tags used in the protocol

The following tags have been allocated for data types in UAF protocol messages:

TAG_UAFV1_REG_ASSERTION 0x3E01
The content of this tag is the authenticator response to a Register command.

TAG_UAFV1_AUTH_ASSERTION 0x3E02
The content of this tag is the authenticator response to a Sign command.

TAG_UAFV1_KRD 0x3E03
Indicates Key Registration Data.

TAG_UAFV1_SIGNED_DATA 0x3E04
Indicates data signed by the authenticator using UAuth.priv key.

TAG_APCVCBOR_AUTH_ASSERTION 0x3E05
The content of this tag is the authenticator response to a Sign command.

TAG_APCVCBOR_SIGNED_DATA 0x3E06
Indicates Android Protected Confirmation data signed by the authenticator using UAuth.priv key.

TAG_ATTESTATION_CERT 0x2E05
Indicates DER encoded attestation certificate.

TAG_SIGNATURE 0x2E06
Indicates a cryptographic signature.

TAG_KEYID 0x2E09
Represents a generated KeyID.

TAG_FINAL_CHALLENGE_HASH 0x2E0A
Represents a generated final challenge hash as defined in [UAFProtocol].

TAG_AAAD 0x2E0B
Represents an Authenticator Attestation ID as defined in [UAFProtocol].

TAG_PUB_KEY 0x2E0C
Represents a generated public key.

TAG_COUNTERS 0x2E0D
Represents the use counters for an authenticator.

TAG_ASSERTION_INFO 0x2E0E
Represents authenticator information necessary for message processing.

TAG_AUTHENTICATOR_NONCE 0x2E0F
Represents a nonce value generated by the authenticator.

TAG_TRANSACTION_CONTENT_HASH 0x2E10
Represents a hash of the transaction content sent to the authenticator.

**TAG_EXTENSION 0x3E11, 0x3E12**
This is a composite tag indicating that the content is an extension.

**TAG_EXTENSION_ID 0x2E13**
Represents extension ID. Content of this tag is a UINT8[] encoding of a UTF-8 string.

**TAG_EXTENSION_DATA 0x2E14**
Represents extension data. Content of this tag is a UINT8[] byte array.

**TAG_RAW_USER_VERIFICATION_INDEX 0x0103**
This is the raw UVI as it might be used internally by authenticators. This TAG SHALL NOT appear in assertions leaving the authenticator boundary as it could be used as a global correlation handle.

**TAG_USER_VERIFICATION_INDEX 0x0104**
The user verification index (UVI) is a value uniquely identifying a user verification data record.

Each UVI value MUST be specific to the related key (in order to provide unlinkability). It also must contain sufficient entropy that makes guessing impractical. UVI values MUST NOT be reused by the Authenticator (for other biometric data or users).

The UVI data can be used by FIDO Servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".

As an example, the UVI could be computed as SHA256(KeyID | SHA256(rawUVI)), where the rawUVI reflects (a) the biometric reference data, (b) the related OS level user ID and (c) an identifier which changes whenever a factory reset is performed for the device, e.g. rawUVI = biometricReferenceData | OSLevelUserID | FactoryResetCounter.

FIDO Servers supporting UVI extensions MUST support a length of up to 32 bytes for the UVI value.

Example of the TLV encoded UVI extension (contained in an assertion, i.e. TAG_UAFV1_REG_ASSERTION or TAG_UAFV1_AUTH_ASSERTION)

```
04 01 -- TAG_USER_VERIFICATION_INDEX (0x0104)
20 -- length of UVI
00 43 B8 E3 BE 27 95 8C -- the UVI value itself
28 D5 74 BF 46 8A 85 CF
46 9A 14 F0 E5 16 69 31
DA 4B CF FF C1 BB 11 32
82
...
```

**TAG_RAW_USER_VERIFICATION_STATE 0x0105**
This is the raw UVS as it might be used internally by authenticators. This TAG SHALL NOT appear in assertions leaving the authenticator boundary as it could be used as a global correlation handle.

**TAG_USER_VERIFICATION_STATE 0x0106**
The user verification state (UVS) is a value uniquely identifying the set of active user verification data records.

Each UVS value MUST be specific to the related key (in order to provide unlinkability). It also must contain sufficient entropy that makes guessing impractical. UVS values MUST NOT be reused by the Authenticator (for other biometric data sets or users).

The UVS data can be used by FIDO Servers to understand whether an authentication was authorized by one of the biometric data records already known at the initial key generation.

As an example, the UVS could be computed as SHA256(KeyID | SHA256(rawUVS)), where the rawUVS reflects (a) the biometric reference data sets, (b) the related OS level user ID and (c) an identifier which changes whenever a factory reset is performed for the device, e.g. rawUVS = biometricReferenceDataSet | OSLevelUserID | FactoryResetCounter.

FIDO Servers supporting UVS extensions MUST support a length of up to 32 bytes for the UVS value.

Example of the TLV encoded UVS extension (contained in an assertion, i.e. TAG_UAFV1_REG_ASSERTION or TAG_UAFV1_AUTH_ASSERTION)

```
06 01 -- TAG_USER_VERIFICATION_STATE (0x0106)
20 -- length of UVS
00 18 C3 47 81 73 2B 65 -- the UVS value itself
83 E7 43 31 46 8A 85 CF
93 6C 36 F0 AF 16 69 14
DA 4B 1D 43 FE C7 43 24
45
...
```

**TAG_USER_VERIFICATION_CACHING 0x0108**
This extension allows an app to specify such user verification caching time, i.e. the time for which the user verification status can be "cached" by the authenticator.
The value of this extension is defined as follows:

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UINT16 Tag</td>
<td>TAG_USER_VERIFICATION_CACHING</td>
</tr>
<tr>
<td>1.1 UINT16</td>
<td>Length of UVC structure in bytes</td>
</tr>
<tr>
<td>1.2 UINT16</td>
<td>maxUVC in seconds</td>
</tr>
<tr>
<td>1.3 UINT8</td>
<td>(optional) verifyIfExceeded. If 0(:false): return error if maxUVC exceeded. If non-zero (:true): verify user if maxUVC exceeded.</td>
</tr>
</tbody>
</table>

Example of the TLV encoded UVC extension (contained in an authentication request)

```
... 08 01         -- TAG_USER_VERIFICATION_CACHING (0x0108)
 05            -- length of UVC
 2c 01 00 00   -- the UVC value itself: maxUVC = 0x012c (300 secs),
 01            -- followed by verifyIfExceeded = 1 (true)
...
```

TAG_RESIDENT_KEY 0x0109
Is the key resident in the authenticator. The value is a boolean. See section Require Resident Key Extension for details.

TAG_RESERVED_5 0x0201
Reserved for future use. Name of the tag will change, value is fixed.

5. Predefined Extensions

This section is normative.

5.1 User Verification Method Extension

This extension can be added

- by FIDO Servers to the UAF Request object (request extension) in the OperationHeader in order to ask the authenticator for using a specific user verification method and confirm that in the related response extension.
- by FIDO Clients to the ASM Request object (request extension) in order to ask the authenticator for using a specific user verification method and confirm that in the related response extension.
- by ASMs to the authenticator command (request extension) in order to ask the authenticator for using a specific user verification method and confirm that in the related response extension.
- by Authenticators to the assertion generated in response to a request in order to confirm a specific user verification method that was used for the action.

Extension identifier
fido.uaf.uvm

When present in a request (request extension)
Same as described in Authenticator argument.

FIDO Client processing
The client SHOULD pass the (request) extension through to the Authenticator.

Authenticate argument
The payload of this extension is an array of:

```
UINT32 userVerificationMethod
```

The array can have multiple entries. Each entry SHALL have a single bit flag set. In this case the authenticator SHALL verify the user using all (multiple) methods as indicated.
The semantics of the fields are as follows:

**userVerificationMethod**
The authentication method used by the authenticator to verify the user. Available values are defined in [FIDORegistry], "User Verification Methods" section.

**Authenticator processing**
The authenticator supporting this extension

1. **SHOULD** limit the user verification methods selectable by the user to the user verification method(s) specified in the request extension.
2. **SHALL** truthfully report the selected user verification method(s) back in the related response extension added to the assertion.

**Authenticator data**
The payload of this extension is an array of the following structure:

```
UINT32 userVerificationMethod
UINT16 keyProtection
UINT16 matcherProtection
```

The array can have multiple entries describing all user verification methods used.

The semantics of the fields are as follows:

**userVerificationMethod**
The authentication method used by the authenticator to verify the user. Available values are defined in [FIDORegistry], "User Verification Methods" section.

**keyProtection**
The method used by the authenticator to protect the FIDO registration private key material. Available values are defined in [FIDORegistry], "Key Protection Types" section. This value has no meaning in the request extension.

**matcherProtection**
The method used by the authenticator to protect the matcher that performs user verification. Available values are defined in [FIDORegistry], "Matcher Protection Types" section.

**Server processing**
If the FIDO Server requested the UVM extension,

1. **SHOULD** verify that a proper response is provided (if client side support can be assumed), and
2. **SHOULD** verify that the UVM response extension specifies one or more acceptable user verification method(s).

### 5.2 User ID Extension

This extension can be added

- by FIDO Servers to the UAF Request object (request extension) in the OperationHeader.
- by FIDO Clients to the ASM Request object (request extension).
- by ASMs to the `TAG_UAFV1REGISTER_CMD` object using `TAG_EXTENSION` (request extension).
- by Authenticators to the registration or authentication assertion using `TAG_EXTENSION` (response extension).

The main purpose of this extension is to allow relying parties finding the related user record by an existing index (i.e. the user ID). This user ID is not intended to be displayed.

Authenticators **SHOULD** truthfully indicate support for this extension in their Metadata Statement.

**Extension identifier**
`fido.uaf.userid`

**Extension fail-if-unknown flag**
`false`, i.e. this (request and response) extension can safely be ignored by all entities.

**Extension data value**
Content of this tag is the UINT8[] encoding of the user ID as UTF-8 string.

### 5.3 Android SafetyNet Extension
This extension can be added

- by FIDO Servers to the UAF Request object (request extension) in the OperationHeader in order to trigger generation of the related response extension.
- by FIDO Clients to the ASM Request object (request extension) in order to trigger generation of the related response extension.
- by the ASM to the respective exts array in the ASMResponse object (response extension).
- by the FIDO Client to the respective exts array in either the OperationHeader, or the AuthenticatorRegistrationAssertion, or the AuthenticatorSignAssertion of the UAF Response object (response extension).

Extension identifier
fido.uaf.safetynet

Extension fail-if-unknown flag
false, i.e. this (request and response) extension can safely be ignored by all entities.

Extension data value

When present in a request (request extension)
empty string, i.e. the FIDO Server might add this extension to the UAF Request with an empty data value in order to trigger the generation of this extension for the UAF Response.

**EXAMPLE 1: SafetyNet Request Extension**
```json
"exts": [{"id": "fido.uaf.safetynet", "data": ",", "fail_if_unknown": false}]
```

When present in a response (response extension)

- If the request extension was successfully processed, the data value is set to the JSON Web Signature attestation response as returned by the call to com.google.android.gms.safetynet.SafetyNetApi.AttestationResponse.
- If the FIDO Client or the ASM support this extension, but the underlying Android platform does not support it (e.g. Google Play Services is not installed), the data value is set to the string "p" (i.e. platform issue).

**EXAMPLE 2: SafetyNet Response Extension - not supported by platform**
```json
"exts": [{"id": "fido.uaf.safetynet", "data": "p", "fail_if_unknown": false}]
```

- If the FIDO Client or the ASM support this extension and the underlying Android platform supports it, but the functionality is temporarily unavailable (e.g. Google servers are unreachable), the data value is set to the string "a" (i.e. availability issue).

**EXAMPLE 3: SafetyNet Response Extension - temporarily unavailable**
```json
"exts": [{"id": "fido.uaf.safetynet", "data": "a", "fail_if_unknown": false}]
```

**NOTE**
If neither the FIDO Client nor the ASM support this extension, it won't be present in the response object.

**FIDO Client processing**

FIDO Clients running on Android should support processing of this extension.

If the FIDO Client finds this (request) extension with empty data value in the UAF Request and it supports processing this extension, then the FIDO Client

1. **MUST** call the Android API SafetyNet.SafetyNetApi.attest(mGoogleApiClient, nonce) (see SafetyNet online documentation) and add the response (or an error code as described above) as extension to the response object.
2. **MUST NOT** copy the (request) extension to the ASM Request object (deviating from the general rule in [UAFProtocol], section 3.4.6.2 and 3.5.7.2).

If the FIDO Client does not support this extension it **MUST** copy this extension from the UAF Request to the ASM Request object
If the ASM supports this extension it **must** call the SafetyNet API (see above) and add the response as extension to the ASM Response object. The FIDO Client **must** copy the extension in the ASM Response to the UAF Response object (according to sections 3.4.6.4. and 3.5.7.4 step 4 in [UAFProtocol]).

When calling the Android API, the nonce parameter **must** be set to the serialized JSON object with the following structure:

```json
{
  "hashAlg": "S256", // the hash algorithm
  "fcHash": "..."    // the finalChallengeHash
}
```

Where

- `hashAlg` identifies the hash algorithm according to [FIDOSignatureFormat], section IANA Considerations.
- `fcHash` is the base64url encoded hash value of FinalChallenge (see section 3.6.3 and 3.7.4 in [UAFASM] for details on how to compute `finalChallengeHash`).

We use this method to bind this SafetyNet extension to the respective FIDO UAF message.

Only hash algorithms belonging to the Authentication Algorithms mentioned in [FIDORegistry] **shall** be used (e.g. SHA256 because it belongs to ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW).

**Authenticator argument**

N/A

**Authenticator processing**

N/A. This extension is related to the Android platform in general and not to the authenticator in particular. As a consequence there is no need for an authenticator to receive the (request) extension nor to process it.

**Authenticator data**

N/A

**Server processing**

If the FIDO Server requested the SafetyNet extension,

1. it **should** verify that a proper response is provided (if client side support can be assumed), and
2. it **should** verify the SafetyNet AttestationResponse (see SafetyNet online documentation).

**NOTE**

The package name in AttestationResponse might relate to either the FIDO Client or the ASM.

**NOTE**

The response extension is not part of the signed assertion generated by the authenticator. If an MITM or MITB attacker would remove the response extension, the FIDO server might not be able to distinguish this from the "SafetyNet extension not supported by FIDO Client/ASM" case.

### 5.4 Android Key Attestation

This extension can be added

- by FIDO Servers to the UAF Registration Request object (request extension) in the OperationHeader in order to trigger generation of the related response extension.
- by FIDO Clients to the ASM Registration Request object (request extension) in order to trigger generation of the related response extension.
- by the ASM to the respective `exts` array in the ASMResponse object related to a registration response (response extension).
- by the FIDO Client to the respective `exts` array in either the OperationHeader, or the AuthenticatorRegistrationAssertion of the UAF Registration Response object (response extension).

**Extension identifier**
**Extension fail-if-unknown flag**

- **false**, i.e. this (request and response) extension can safely be ignored by all entities.

**Extension data value**

- When present in a request (request extension), the **empty string**, i.e. the FIDO Server might add this extension to the UAF Request with an empty **data** value in order to trigger the generation of this extension for the UAF Response.

**EXAMPLE 4: Android KeyAttestation Request Extension**

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": ",", "fail_if_unknown": false}]
```

- When present in a response (response extension),
  - If the request extension was successfully processed, the **data** value is set to a JSON array containing the base64 encoded entries of the array returned by the call to the KeyStore API function `getCertificateChain`.

**EXAMPLE 5: Retrieve KeyAttestation and add it as extension**

```
Calendar notBefore = Calendar.getInstance();
notAfter.add(Calendar.YEAR, 10);

KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC, "AndroidKeyStore");
kpGenerator.initialize(new KeyPairGeneratorSpec.Builder(keyUUID, KeyProperties.PURPOSE_SIGN)
  .setDigests(KeyProperties.DIGEST_SHA256)
  .setCertificateSubject(new X500Principal(String.format("CN=%s, OU=%s", keyUUID, aContext.getPackageName())))
  .setCertificateSerialNumber(BigInteger.ONE)
  .setKeyValidityStart(notBefore.getTime())
  .setKeyValidityEnd(notAfter.getTime())
  .setUserAuthenticationRequired(true)
  .setAttestationChallenge(fcHash) -- bind to Final Challenge .build());

kpGenerator.generateKeyPair(); // generate Uauth key pair
```

**EXAMPLE 6: Example of successfull key attestation extension response**

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "["MIIClDCCAjugAwIBAgIBATAKBggqhkjOPQgMBAQR..."
```

**EXAMPLE 6: Example of successfull key attestation extension response**

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": ",", "fail_if_unknown": false}]
```

```
Calendar notBefore = Calendar.getInstance();
notAfter.add(Calendar.YEAR, 10);

KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC, "AndroidKeyStore");
kpGenerator.initialize(new KeyPairGeneratorSpec.Builder(keyUUID, KeyProperties.PURPOSE_SIGN)
  .setDigests(KeyProperties.DIGEST_SHA256)
  .setCertificateSubject(new X500Principal(String.format("CN=%s, OU=%s", keyUUID, aContext.getPackageName())))
  .setCertificateSerialNumber(BigInteger.ONE)
  .setKeyValidityStart(notBefore.getTime())
  .setKeyValidityEnd(notAfter.getTime())
  .setUserAuthenticationRequired(true)
  .setAttestationChallenge(fcHash) -- bind to Final Challenge .build());

kpGenerator.generateKeyPair(); // generate Uauth key pair
```

**EXAMPLE 5: Retrieve KeyAttestation and add it as extension**

```
Calendar notBefore = Calendar.getInstance();
notAfter.add(Calendar.YEAR, 10);

KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC, "AndroidKeyStore");
kpGenerator.initialize(new KeyPairGeneratorSpec.Builder(keyUUID, KeyProperties.PURPOSE_SIGN)
  .setDigests(KeyProperties.DIGEST_SHA256)
  .setCertificateSubject(new X500Principal(String.format("CN=%s, OU=%s", keyUUID, aContext.getPackageName())))
  .setCertificateSerialNumber(BigInteger.ONE)
  .setKeyValidityStart(notBefore.getTime())
  .setKeyValidityEnd(notAfter.getTime())
  .setUserAuthenticationRequired(true)
  .setAttestationChallenge(fcHash) -- bind to Final Challenge .build());

kpGenerator.generateKeyPair(); // generate Uauth key pair
```

**EXAMPLE 6: Example of successfull key attestation extension response**

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "["MIIClDCCAjugAwIBAgIBATAKBggqhkjOPQgMBAQR..."
```
If the FIDO Client or the ASM support this extension, but the underlying Android platform does not support it (e.g. Android version doesn't yet support it), the data value is set to the string "p" (i.e. platform issue).

If the FIDO Client or the ASM support this extension and the underlying Android platform supports it, but the functionality is temporarily unavailable (e.g. Google servers are unreachable), the data value is set to the string "a".

NOTE
Line-breaks been added for readability.

- If the FIDO Client or the ASM support this extension, but the underlying Android platform does not support it (e.g. Android version doesn't yet support it), the data value is set to the string "p" (i.e. platform issue).

EXAMPLE 7: KeyAttestation Response Extension - not supported by platform

```json
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "p", "fail_if_unknown": false}]
```

- If the FIDO Client or the underlying Android platform supports it, but the functionality is temporarily unavailable (e.g. Google servers are unreachable), the data value is set to the string "a".

EXAMPLE 8: KeyAttestation Response Extension - temporarily unavailable

```json
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "a", "fail_if_unknown": false}]
```

NOTE
If neither the FIDO Client nor the ASM support this extension, it won't be present in the response object.

FIDO Client processing

FIDO Clients running on Android **MUST** pass this (request) extension with empty data value to the ASM.

If the ASM supports this extension it **MUST** call the KeyStore API (see above) and add the response as extension to the ASM Response object. The FIDO Client **MUST** copy the extension in the ASM Response to the UAF Response object (according to section 3.4.6.4 step 4 in [UAFProtocol]).

More details on Android key attestation can be found at:

- https://developer.android.com/training/articles/keystore.html
- https://developer.android.com/training/articles/security-key-attestation
- https://source.android.com/security/keystore/
- https://source.android.com/security/keystore/implementer-ref.html

Authenticator argument
N/A

Authenticator processing
The authenticator generates the attestation response. The call keyStore.getCertificateChain is finally processed by the authenticator.

**Authenticator data**

N/A

**Server processing**

If the FIDO Server requested the key attestation extension,

1. it **MUST** follow the registration response processing rules (see FIDO UAF Protocol, section 3.4.6.5) before processing this extension
2. it **MUST** verify the syntax of the key attestation extension and it **MUST** perform RFC5280 compliant chain validation of the entries in the array to one attestationRootCertificate specified in the Metadata Statement - accepting that that the keyCertSign bit in the key usage extension of the certificate issuing the leaf certificate is **NOT set** (which is a deviation from RFC5280).
3. it **MUST** determine the leaf certificate from that chain, and it **MUST** perform the following checks on this leaf certificate
   1. Verify that KeyDescription.attestationChallenge == FCHash (see FIDO UAF Protocol, section 3.4.6.5 Step 6.)
   2. Verify that the public key included in the leaf certificate is identical to the public key included in the FIDO UAF Surrogate attestation block
   3. If the related Metadata Statement claims keyProtection KEY_PROTECTION_TEE, then refer to KeyDescription.teeEnforced using "authzList". If the related Metadata Statement claims keyProtection KEY_PROTECTION_SOFTWARE, then refer to KeyDescription.softwareEnforced using "authzList".
4. Verify that
   1. authzList.origin == KM_TAG_GENERATED
   2. authzList.purpose == KM_PURPOSE_SIGN
   3. authzList.keySize is acceptable, i.e. =2048 (bit) RSA or =256 (bit) ECDSA.
   4. authzList.digest == KM_DIGEST_SHA_2_256.
   5. authzList.userAuthType only contains acceptable user verification methods.
   6. authzList.authTimeout == 0 (or **not** present).
   7. authzList.noAuthRequired is **not** present (unless the Metadata Statement marks this authenticator as silent authenticator, i.e. userVerification set to USER_VERIFY_NONE).
   8. authzList.allApplications is **not** present, since FIDO Uauth keys **MUST** be bound to the generating app (AppID).

**NOTE**

The response extension is not part of the signed assertion generated by the authenticator. If an MITM or MITB attacker would remove the response extension, the FIDO server might not be able to distinguish this from the "KeyAttestation extension not supported by ASM/Authenticator" case.

**ExtensionDescriptor data value (for Metadata Statement)**

In the case of extension id="fido.uaf.android.key_attestation", the data field of the ExtensionDescriptor as included in the Metadata Statement will contain a dictionary containing the following data fields

**DOMString attestationRootCertificates[]**

Each element of this array represents a PKIX [RFC5280] X.509 certificate that is valid for this authenticator model. Multiple certificates might be used for different batches of the same model. The array does not represent a certificate chain, but only the trust anchor of that chain.

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

**NOTE**

A certificate listed here is either a root certificate or an intermediate CA certificate.

**NOTE**

The field data is specified with type DOMString in [FIDOMetadataStatement] and hence will contain the serialized object as described above.
An example for the supportedExtensions field in the Metadata Statement could look as follows (with line breaks to improve readability):

```
"supportedExtensions": [{
  "id": "fido.uaf.android.key_attestation",
  "data": {
    "attestationRootCertificates": [
      "MIICPTCAoGhCcA9AgIBJuowxyzvU0y2W4oACGcGSM49BAMCMxIDIAEaBgNVBAgM
      FinHbX5B5BdNItlBlfSb9vYsYAYFDVQQDEA1GSURFIPbSbGh0bmcNL
      MR5Gw0hXYVQmQDNAhDAyxyQyFEd1LR3KBAGA1UEwRUQzgYeYYSGfSBbHRbWQGwQYVQQI
      DADJTDQELmkGCA1UEBhoMCV5NUMhcNMTQwNjE4MTM0MzMyNcNNDDEMTA2MTMzMzMy
      WlJ7BMSwqIhM1YXVQDB0fTm1wbGQxQXRUZXoYXpb24gdmVudENBMBQGA1UEBgcW
      cRKX00zELmkGCA1UECAwCQ0EeCzA8gNBYcTVFkRwWhKOz1zj0CAOYIKoZI
      zj0DAAQDBCQAIhNhzIDNHAxas9/Bmpj7/R2ehL/FMgsFdiDkg9VAGoPZ3ajhnu94FQ7
      aMzH33nUSBr8fHYDrqOBb58pxGg3JXpRQw/6NGME4wQYDVQBDDEBkEFFOA3CLhXpFb
      C0ITz7E4w8hsk5EJ/MB8GA1UDwiyQYMBaAFPoHAC3LhxFbCOITz7E4w8hsk5EJ/MAwG
      A10d6w8fPMABAg8wqYfYKoPZiJ0EAwIgAAwRQIhA0J06gSXTt9ihbGFKK1sKPrk1
      /VdLgi6fGouS7Ecro4AIbqyYCF0+i55aQeABJzA9x63irx8b39ps2ZXN
      1Q=="}, "fail_if_unknown": false }
}]
```

5.5 User Verification Caching

In several cases it is good enough for the relying party to know whether the user was verified by the authenticator "some time" ago. This extension allows an app to specify such user verification caching time, i.e. the time for which the user verification status can be "cached" by the authenticator.

For example: Do not ask the user for a fresh user verification to authorize a payment of 4€ if the user was verified by this authenticator within the past 300 seconds.

This extension allows the authenticator to bridge the gap between a "silent" authenticator, i.e. an authenticator never verifying the user and a "traditional" authenticator, i.e. an authenticator always asking for fresh user verification.

We formally define one extension for the request and a separate extension for the response as the request extension can be safely ignored, but the response extension cannot.

Authenticator supporting this extension must truthfully specify both, the UVC Request and UVC Response extension in the supportedExtensions list of the related Metadata Statement [FIDOMetadataStatement]. The TAG of the UVC Response extension must be specified in that list.

5.5.1 UVC Request

This extension can be added by FIDO Servers to the UAF Request object (request extension) in the OperationHeader in order to trigger generation of the related response extension.

**Extension Identifier**

fido.uaf.uvc-req

**Extension fail-if-unknown flag**

false, i.e. the request extension can safely be ignored by all entities.

**UVC Extension data value**

A (base64url-encoded) TLV object as defined in the description of TAG_USER_VERIFICATION_CACHING. In the UVC Extension provided through the DOM API [UAFAppAPIAndTransport], the field verifyIfExceeded MAY NOT be present. The FIDO Client MAY add the field verifyIfExceeded in order to improve processing.

**FIDO Client processing**

- In a registration request: Simple pass-through to the platform preferred authenticator.
- In a sign request: Simple pass-through to an authenticator which would not require fresh user verification and still meets all other authentication selection criteria (if such authenticator exists). If this is not possible, then use the preferred authenticator (as normal) but pass-through this extension.

**Authenticator argument**

Same TLV object as defined in "Extension data value", but as binary object included in the Registration / Authentication command.
**Authenticator processing**

**In a registration request:**

The Authenticator **MUST** always freshly verify the user and create a key marked with the maximum user verification caching time as specified (referred to as `regMaxUVC`), i.e. in `signAssertion` the acceptable maximum user verification time can never exceed this value. The field `(verifyIfExceeded)` is **not** allowed in a registration request.

**In a sign request:**

If the authenticator supports specifying user verification caching time in a sign request:

1. Compute \( \text{maxUVC} = \min(\text{maxUVC}, \text{regMaxUVC}) \)
2. Compute \( \text{elapsedTime} \), i.e. the time since last user verification.
3. If \( (\text{elapsedTime} \geq \text{maxUVC}) \) AND \( \text{verifyIfExceeded} = \text{false} \) then return error
4. If \( (\text{elapsedTime} \geq \text{maxUVC}) \) AND ((\( \text{verifyIfExceeded} = \text{true} \)) OR (\( \text{verifyIfExceeded} \) is NOT PRESENT)) then verify user
5. If \( (\text{elapsedTime} \leq \text{maxUVC}) \) then Sign the assertion as normal
6. Add the **UVC Response** extension to the assertion.

If the authenticator does not support specifying user verification caching time in a sign request, this extension will be ignored by the authenticator. This will be detected by the server since no extension output will be generated by the authenticator.

**Authenticator data**

N/A

**Server processing**

N/A

5.5.2 **UVC Response**

This extension can be added by the Authenticator to the `AuthenticatorRegistrationAssertion`, or the `AuthenticatorSignAssertion` of the UAF Response object (response extension).

**Extension Identifier**

fido.uaf.uvc-resp (TAG_USER_VERIFICATION_CACHING)

**Extension fail-if-unknown flag**

`true`, i.e. the `response` extension (included in the UAF assertion) **MAY** NOT be ignored if unknown. If the server is not prepared to process the UVC response extension, it **MUST** fail.

**Extension data value**

N/A

**FIDO Client processing**

N/A

**Authenticator argument**

N/A

**Authenticator processing**

N/A

**Authenticator data**

If the extension is supported and the request extension was received and evaluated during the respective call, then the binary TLV object as described in the description of `TAG_USER_VERIFICATION_CACHING` will be included in the assertion generated by the Authenticator.

Where the field `maxUVC` contains an upper bound of `trueUVC` and where the field `verifyIfExceeded` **will not** be present.

The upper bound value is to be computed as follows:

1. Compute the elapsed seconds since last user verification (= `trueUVC`).
2. Compute some upper bound of `trueUVC`, must not exceed \( \min(\text{command.maxUVC}, \text{regMaxUVC}) \).

   Where `command.maxUVC` refers to the `maxUVC` value of the related **UVC Request**.

   Where `regMaxUVC` is the `maxUVC` value specified in the related registration call (see above) or 0 if no such value was provided at registration time.

For example, use \( \min(\text{maxUVC}, \text{createMaxUVC}) \) or \( \min(\text{round(trueUVC to 5 seconds)}, \text{maxUVC}, \text{createMaxUVC}) \).

**Server processing**
If the FIDO Server requested the UVC extension,

1. Verify that the Metadata Statement related to this Authenticator indicates support for this extension in the field `supportedExtensions`
2. Verify that assertion.maxUVC is less or equal to request.maxUVC, fail if it isn't.
3. Verify that assertion.maxUVC is acceptable, fail if it isn't.

If the FIDO Server did not request the UVC extension (but encounters it in the response) or if the server doesn't understand the UVC response extension, it **MUST** fail.

### 5.5.3 Privacy Considerations

Using the UVC Request extension with `verifyIfExceeded` set to `FALSE` might allow the caller to triage the last time the user was verified without requiring any input from the user and without notifying the user. We do not allow this field to be set through the DOM API (i.e. by web pages). However, native applications can use this field and hence could be able to determine the last time the user was verified. Native applications have substantially more permissions and hence can have more detailed knowledge about the user's behavior than web pages (e.g. track whether the device is used by evaluating accelerometers).

In the UVC Response extension the Authenticator can provide an upper bound of the `trueUVC` value in order to prevent disclosure of exact time of user verification.

### 5.5.4 Security Considerations

FIDO Servers not expecting user verification being used, might expect a fresh user verification and an explicit user consent being provided. Authenticators supporting this extension shall only use it when they are asked for that (i.e. UVC Request extension is present). Additionally the authenticator must indicate if the user was **not** freshly verified using the UVC Response extension. This response extension is marked with "fail-if-unknown" set to true, to make sure that servers receiving this extension know that the user might not have been freshly verified.

### 5.6 Require Resident Key extension

This extension is intended to simplify the integration of authenticators implementing [FIDOCTAP](#) with FIDO UAF [UAFProtocol](#).

**Extension Identifier**

`fido.uaf.rk` (TAG_RESIDENT_KEY)

**Extension fail-if-unknown flag**

`false`, i.e. the extension **MAY** be ignored if unknown.

**Extension data value**

boolean, i.e. `rk=true` or `rk=false`.

**FIDO Client processing**

N/A

**Authenticator argument**

boolean, i.e. `rk=true` or `rk=false`.

**Authenticator processing**

If the authenticator supports this extension, it should

1. persistently store the credential's cryptographic key material internally is `rk=true`
2. **NOT** persistently store the credential's cryptographic key material internally is `rk=false`

**NOTE**

It is expected that

1. authenticators with `isSecondFactorOnly=false` in their Metadata Statement will persistently store the credential's cryptographic key material internally if the extension is missing.
2. authenticators with `isSecondFactorOnly=true` in their Metadata Statement will **NOT** persistently store the credential's cryptographic key material internally if the extension is missing.

**Authenticator data**

boolean, i.e. `rk=true` or `rk=false` in an assertion, indicating whether the current credential is resident in the authenticator or not.

**Server processing**
A response extension `fido.uaf.rk` set to false indicates that the FIDO Server needs to provide a keyHandle for triggering authentication. This means that the authenticator can only be used as a second factor (see also `isSecondFactorOnly` in [FIDOMetadataStatement]).

If the FIDO Server did not request the `fido.uaf.rk` extension (but encounters it in the response) or if the server doesn't understand the `fido.uaf.rk` response extension, it can silently ignore the extension.

5.7 Attestation Conveyance Extension

This extension is intended to simplify the integration of authenticators implementing [FIDOCTAP] with FIDO UAF [UAFProtocol].

Extension Identifier

`fido.uaf.ac`

Extension fail-if-unknown flag

`false`, i.e. the extension **may** be ignored if unknown.

Extension data value

String, i.e. `ac='direct'`, `ac='indirect'`, or `ac='none'`.

FIDO Client processing

If the ac value is

- **direct**
  - the FIDO Client **shall** pass-through the attestation statement as received from the Authenticator.

- **indirect**
  - the FIDO Client **shall** either
    1. pass-through the attestation statement as received from the Authenticator or
    2. replace the attestation statement received from the Authenticator using some anonymization CA.

- **none**
  - the FIDO Client **shall** remove the attestation statement received from the Authenticator.

Authenticator argument

N/A

Authenticator processing

If the authenticator supports this extension, it should

1. return an attestation statement according to the conveyance indicated.

Authenticator data

N/A (only indirectly through the generated attestation statement)

Server processing

The server should verify the attestation statement if it asked for it (i.e. `ac='direct'` or `ac='indirect'`).

If the FIDO Server specified `ac='none'`, but received an attestation statement, it can silently ignore it.

6. Other Identifiers specific to FIDO UAF

6.1 FIDO UAF Application Identifier (AID)

This AID [ISOIEC-7816-5] is used to identify FIDO UAF authenticator applications in a Secure Element.

The FIDO UAF AID consists of the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>RID</th>
<th>AC</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0xA000000647</td>
<td>0xAF</td>
<td>0x0001</td>
</tr>
</tbody>
</table>

Table 1: FIDO UAF Applet AID

A. References

A.1 Normative references

[FIDOGlossary]
FIDO UAF WebAuthentication Assertion Format

FIDO Alliance Proposed Standard 20 October 2020

This version:
https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-webauthn-v1.2-ps-20201020.html

Editor:
Dr. Rolf Lindemann, 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

This document defines the assertion format "WAV1CBOR" in order to use Web Authentication assertions through the FIDO UAF protocol.

Status of This Document

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

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

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

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

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

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

1.1 Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

This document defines the assertion format "WAV1CBOR" in order to use Web Authentication assertions through the FIDO UAF protocol.

3. Data Structures for WAV1CBOR

This section is normative.

3.1 Registration Assertion

The registration assertion for the assertion format "WAV1CBOR" is a TLV encoded object containing the CBOR encoded authenticatorData, the name of the attestation format, and the atestation statement itself.

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UINT16 Tag</td>
<td>TAG_WAV1CBOR_REG_ASSERTION</td>
</tr>
<tr>
<td>1.1 UINT16 Length</td>
<td>Length of the structure.</td>
</tr>
<tr>
<td>1.2 UINT16 Tag</td>
<td>TAG_WAV1CBOR_REG_DATA</td>
</tr>
<tr>
<td>1.2.1 UINT16 Length</td>
<td>Length of the structure.</td>
</tr>
<tr>
<td>1.2.2 UINT8 tbsData</td>
<td>The binary authenticatorData structure as specified in section 6.1 in [WebAuthn] with non-empty attestedCredentialData field being present followed by (i.e. binary concatenation) the clientDataHash.</td>
</tr>
</tbody>
</table>
1.3 INT16 Tag  TAG_ATTESTATION_FORMAT
1.3.1 INT16 Length  Length of Attestation Format
1.3.2 INT8[] Attestation Format  Authenticator Attestation Format, see field "fmt" in section sctn-attestation in [WebAuthn]

1.4 INT16 Tag  TAG_ATTESTATION_STATEMENT
1.4.1 INT16 Length  Length of Attestation Statement
1.4.2 INT8[] Attestation Statement  Authenticator Attestation Statement, see field "stmt" in section sctn-attestation in [WebAuthn]. This field contains the signature in sub-field "sig".

3.2 Authentication Assertion

The authentication assertion is a TLV structure containing the CBOR encoded authenticatorData object, the authenticator model name (AAGUID), the key identifier and the signature of the authenticatorData object.

<table>
<thead>
<tr>
<th>TLV Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INT16 Tag</td>
<td>TAG_WAV1CBOR_AUTH_ASSERTION</td>
</tr>
<tr>
<td>1.1 INT16 Length</td>
<td>Length of the structure.</td>
</tr>
<tr>
<td>1.2 INT16 Tag</td>
<td>TAG_WAV1CBOR_SIGNED_DATA</td>
</tr>
<tr>
<td>1.2.1 INT16 Length</td>
<td>Length of the structure.</td>
</tr>
<tr>
<td>1.2.2 INT8 tbsData</td>
<td>As described in step 11 in section 6.3.3 in [WebAuthn]: The binary authenticatorData structure as specified in section 6.1 in [WebAuthn] with empty attestedCredentialData field being present followed by (i.e. binary concatenation) the clientDataHash.</td>
</tr>
<tr>
<td>1.3 INT16 Tag</td>
<td>TAG_AAGUID</td>
</tr>
<tr>
<td>1.3.1 INT16 Length</td>
<td>Length of AAGUID</td>
</tr>
<tr>
<td>1.3.2 INT8[] AAGUID</td>
<td>Authenticator Attestation GUID, see section 6.4.1 in [WebAuthn]</td>
</tr>
<tr>
<td>1.4 INT16 Tag</td>
<td>TAG_KEYID</td>
</tr>
<tr>
<td>1.4.1 INT16 Length</td>
<td>Length of KeyID</td>
</tr>
<tr>
<td>1.4.2 INT8[] KeyID</td>
<td>(binary value of) Credential ID (see definition of CredentialID in [WebAuthn])</td>
</tr>
<tr>
<td>1.5 INT16 Tag</td>
<td>TAG_SIGNATURE</td>
</tr>
<tr>
<td>1.5.1 INT16 Length</td>
<td>Length of Signature</td>
</tr>
</tbody>
</table>
4. Processing Rules

This section is normative.

4.1 Registration Response Processing Rules for ASM

See [UAFASM] for details of the ASM API.

Refer to [UAFAuthnrCommands] document for more information about the TAGs and structure mentioned in this paragraph.

1. Locate authenticator using authenticatorIndex. If the authenticator cannot be located, then fail with error code UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED.

2. Connect to the Authenticator and call authenticatorGetInfo [FIDOCTAP]. Remember whether the authenticator supports residentKeys (rk), clientPin, User Presence (up), User Verification (uv). Also remember whether the authenticator is a roaming authenticator (plat=false), or a platform authenticator (plat=true). If the connection fails, then fail with error code UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED.

3. If clientPin is the requested user verification method (see UVM extension), but step 2 indicated that clientPin is not yet set (i.e. clientPin present but set to false), then ask user to set (enroll) clientPin.
   - If neither the ASM nor the Authenticator can trigger the enrollment process, return UAF_ASM_STATUS_USER_NOT_ENROLLED.
   - If enrollment fails, return UAF_ASM_STATUS_ACCESS_DENIED.

4. Hash the provided ASMRequest.args.finalChallenge using the authenticator-specific hash function and store the result in FinalChallengeHash.

An authenticator's preferred hash function information must meet the algorithm defined in the AuthenticatorInfo.authenticationAlgorithm field.

5. For each extension included in ASMRequest.exts
   - If the extension "fido.uaf.rk" is found, set parameter rk to the value of that extension and continue with the next extension.
   - If the extension "fido.uaf.ac" is found, set parameter ac to the value of that extension and continue with the next extension.
   - If the extension was not handled before, create a corresponding WebAuthn/FIDO2 extension (see [WebAuthn]) extension in extensionsCBOR. If no corresponding WebAuthn/FIDO2 extension is specified, ignore this extension (if fail_if_unknown is false) or return UAF_ASM_STATUS_ERROR (if fail_if_unknown is true).

6. Call authenticatorMakeCredential [FIDOCTAP] (either via CTAP or via a platform proprietary API), send the required information and receive result containing the error code of that operation.

NOTE

This interface has the following input parameters (see [FIDOCTAP]):

1. clientDataHash (required, byte array).
2. rp (required, PublicKeyCredentialRpEntity). Identity of the relying party.
3. user (required, PublicKeyCredentialUserEntity).
4. pubKeyCredParams (required, CBOR array).
5. excludeList (optional, sequence of PublicKeyCredentialDescriptors).
6. extensions (optional, CBOR map). Parameters to influence authenticator operation.
7. options (optional, sequence of authenticator options, i.e. parameters rk, uv, and up).
8. pinAuth (optional, byte array).
9. pinProtocol (optional, unsigned integer).

The output parameters are (see [FIDOCTAP]):

1. authData (required, sequence of bytes). The authenticator data object.
Use the following values for the respective parameters:

- Set `rp.rpId` to the `ASMRequest.args.AppID`.
- Set `user.Id` to the `fido.uaf.userid` extension retrieved from `ASMRequest.exts`; set `user.displayName` to `ASMRequest.args.username`. Fail if the `fido.uaf.userid` extension is missing in `ASMRequest.exts`.
- Set `clientDataHash` to `FinalChallengeHash`.
- Set `pubKeyCredParams.type` to "public-key" and `pubKeyCredParams.alg` to the preferred algorithm, e.g. "ES256".
- Set `excludeList` to an empty list.
- Set `extensions` to the CBOR map `extensionsCBOR`.
- Set `pinAuth` and `pinProtocol` to the respective values supported by this ASM (to the extent the underlying platform allows specifying these values).
- Set `options` to an empty object and add items as follows:
  1. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and `uvm.userVerificationMethod` includes one or more of the flags `USER_VERIFY_FINGERPRINT`, `USER_VERIFY_FACEPRINT`, `USER_VERIFY_LOCATION`, `USER_VERIFY_EYEPATTERN`, or `USER_VERIFY_HANDPRINT` set `options.userVerification` to `true` and set `options.userPresence` to `true`.
  2. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and `uvm.userVerificationMethod` is equal to `USER_VERIFY_CLIENTPIN` set `options.userVerification` to `true` and set `options.userPresence` to `false`.
  3. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and `uvm.userVerificationMethod` is equal to `USER_VERIFY_PRESENCE` set `options.userVerification` to `false` and set `options.userPresence` to `true`.
  4. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and `uvm.userVerificationMethod` is equal to `USER_VERIFY_NONE` set `options.userVerification` to `false` and set `options.userPresence` to `false`.

NOTE

If the authenticator uses clientPin but the clientPin was not set (indicated by `CTAP2_ERR_PIN_NOT_SET`), the ASM should ask the user for the clientPin and provide it to the authenticator.

7. If `result` is not equal to `CTAP2_OK` and retry cannot fix the problem, then map the CTAP error code to a UAF ASM error code using the table in section 5. Mapping CTAP2 error codes to ASM error codes and return the resulting error code.

8. Create a `TAG_WAV1CBOR_REG_ASSERTION` structure:
   1. Copy `result.AuthData` concatenated with the `finalChallengeHash` into field `TAG_WAV1CBOR_SIGNED_DATA`.
   2. Copy `result.fmt` into field `TAG_ATTESTATION_FORMAT`.
   3. Copy `result.stmt` into field `TAG_ATTESTATION_STATEMENT`.

9. Create a `RegisterOut` object:
   1. Set `RegisterOut.assertionScheme` to "WAV1CBOR".
   2. Encode the content of `TAG_WAV1CBOR_REG_ASSERTION` in base64url format and set as `RegisterOut.assertion`.

10. Set `ASMResponse.responseData` to `RegisterOut`.

11. Set `ASMResponse.statusCode` to the correct status code corresponding to the `result` received earlier.

12. Set `ASMResponse.exts` to empty.

13. Return `ASMResponse` object.

4.2 Registration Response Processing Rules for FIDO Server

Instead of skipping the assertion as described in step 6.8 in section 3.4.6.5 [UAFProtocol], follow these rules:

1. if `a.assertionScheme == "WAV1CBOR" AND a.assertion.TAG_WAV1CBOR_REG_ASSERTION contains TAG_WAV1CBOR_SIGNED_DATA as first element:
   1. extract `authenticatorData` from `TAG_WAV1CBOR_SIGNED_DATA.tbsData`.
   2. read `claimedAAGUID` from `authenticatorData.attestedCredentialData.AAGUID`.
   3. Verify that `a.assertionScheme` matches `Metadata(claimedAAGUID).assertionScheme`.
4. Verify that the \texttt{claimedAAGUID} indeed matches the policy specified in the registration request.

\begin{note}
Depending on the policy (e.g. in the case of AND combinations), it might be required to evaluate other assertions included in this \texttt{RegistrationResponse} in order to determine whether this AAGUID matches the policy.
\end{note}

5. Locate authenticator-specific authentication algorithms from the authenticator metadata [\texttt{FIDOMetadataStatement}] identified by \texttt{claimedAAGUID} (field \texttt{authenticationAlgs}).

6. If \texttt{fcp} is of type \texttt{FinalChallengeParams} [\texttt{UAFProtocol}], then hash \texttt{RegistrationResponse.fcParams} using hashing algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator metadata, field \texttt{AuthenticationAlgs}. It is the hash algorithm associated with the first entry related to a constant with prefix \texttt{ALG\_SIGN}.

\begin{itemize}
    \item \texttt{FCHash = \textbf{hash}(RegistrationResponse.fcParams)}
\end{itemize}

7. If \texttt{fcp} is of type \texttt{CollectedClientData} [\texttt{UAFProtocol}], then hash \texttt{RegistrationResponse.fcParams} using hashing algorithm specified in \texttt{fcp.hashAlg}.

\begin{itemize}
    \item \texttt{FCHash = \textbf{hash}(RegistrationResponse.fcParams)}
\end{itemize}

8. Obtain \texttt{Metadata(claimedAAGUID).AttestationType} for the \texttt{claimedAAGUID} and make sure that \texttt{a.assertion.TAG\_NACIBOR\_REG\_ASSERTION} contains the most preferred attestation tag specified in field \texttt{MatchCriteria.attestationTypes} in \texttt{RegistrationRequest.policy} (if this field is present).

\begin{itemize}
    \item If \texttt{a.assertion.TAG\_NACIBOR\_REG\_ASSERTION} doesn't contain the preferred attestation - it is \textbf{RECOMMENDED} to skip this assertion and continue with next one
\end{itemize}

9. \texttt{set tbsData to the data contained in a.assertion.tbsData.}

10. \texttt{set authenticatorData to the CBOR object tbsData starts with. Use the "length" field of the CBOR object to determine its end.}

11. \texttt{set clientDataHash to the remaining bytes of the tbsData (i.e. the bytes following the CBOR object).}

12. \texttt{Make sure that clientDataHash == FCHash}

\begin{itemize}
    \item \texttt{If comparison fails - continue with next assertion}
\end{itemize}

13. Extract the \texttt{up} and \texttt{uv} bits from \texttt{authenticatorData}. Verify whether these bits match the \texttt{UVM} extension sent in the request. Fail if the verification result is not acceptable.

\begin{note}
\begin{itemize}
    \item \texttt{up=false and uv=false} means silent authentication (\texttt{USER\_VERIFY\_NONE})
    \item \texttt{up=true and uv=false} means user presence check only (\texttt{USER\_VERIFY\_PRESENCE})
    \item \texttt{up=false and uv=true} means user verification that doesn't provide user presence check, e.g. client Pin or some other user verification method not necessarily implemented fully inside the authenticator boundary (\texttt{USER\_VERIFY\_CLIENTPIN})
    \item \texttt{up=true and uv=true} means user verification using a user verification method implemented inside the authenticator boundary (e.g. \texttt{USER\_VERIFY\_FINGERPRINT, ...}) or client Pin plus user presence check (\texttt{USER\_VERIFY\_CLIENTPIN} \textbf{AND} \texttt{USER\_VERIFY\_PRESENCE}) - depending on the authenticator capabilities as declared in the related Metadata Statement.
\end{itemize}
\end{note}

14. If a \texttt{UVM} extension is included in the response, extract this value and compare it verify whether it matches the extension from the request. Fail if the verification result is not acceptable.

15. If \texttt{a.assertion.TAG\_NACIBOR\_REG\_ASSERTION.TAG\_ATTESTATION\_STATEMENT} contains \texttt{ATTESTATION\_BASIC\_FULL} tag

\begin{enumerate}
    \item If entry \texttt{AttestationRootCertificates} for the \texttt{claimedAAGUID} in the metadata [\texttt{FIDOMetadataStatement}] contains at least one element:
        \begin{enumerate}
            \item Obtain \texttt{contents of all TAG\_ATTESTATION\_CERT tags from a.assertion.TAG\_NACIBOR\_REG\_ASSERTION.ATTESTATION\_BASIC\_FULL object}. The occurrences are ordered (see \texttt{[UAFAuthnrCommands]}) and represent the attestation certificate followed by the related certificate chain.
            \item Obtain all entries of \texttt{AttestationRootCertificates} for the \texttt{claimedAAGUID} in authenticator Metadata, field \texttt{AttestationRootCertificates}.
        \end{enumerate}
\end{enumerate}
3. Verify the attestation certificate and the entire certificate chain up to the Attestation Root Certificate using Certificate Path Validation as specified in [RFC5280]
   - If verification fails – continue with next assertion
4. Verify \texttt{a.assertion.TAG_WAV1CBOR_REG_ASSERTION.\texttt{TAG_ATTESTATION_STATEMENT}.sig} using the attestation certificate (obtained before).
   - If verification fails – continue with next assertion
2. If Metadata(claimedAAGUID).AttestationRootCertificates for this claimedAAGUID is empty - continue with next assertion
3. Mark assertion as positively verified
16. If \texttt{a.assertion.TAG_WAV1CBOR_REG_ASSERTION.\texttt{TAG_ATTESTATION_STATEMENT}} contains an object of type \texttt{ATTESTATION_BASIC_SURROGATE}
   1. There is no real attestation for the AAGUID, so we just assume the claimedAAGUID is the real one.
   2. If entry AttestationRootCertificates for the claimedAAGUID in the metadata is not empty - continue with next assertion (as the AAGUID obviously is expecting a different attestation method).
   3. Verify that extension "fido.uaf.android.key_attestation" is present and check whether it is positively verified according to its server processing rules as specified [UAFRegistry].
      - If verification fails – continue with next assertion
4. Mark assertion as positively verified
17. If \texttt{a.assertion.TAG_WAV1CBOR_REG_ASSERTION} contains an object of type \texttt{ATTESTATION_ECDAA}
   1. If entry ecdaaTrustAnchors for the claimedAAGUID in the metadata [FIDOMetadataStatement] contains at least one element:
      1. For each of the ecdaaTrustAnchors entries, perform the ECDAA Verify operation as specified in [FIDOEcdaaAlgorithm].
         - If verification fails – continue with next ecdaaTrustAnchors entry
      2. If no ECDAA Verify operation succeeded – continue with next assertion
   2. Mark assertion as positively verified and the authenticator indeed is of model as indicated by the claimedAAGUID.
   3. If Metadata(claimedAAID).ecdaaTrustAnchors for this claimedAAGUID is empty - continue with next assertion
4. Mark assertion as positively verified and the authenticator indeed is of model as indicated by the claimedAAGUID.
18. If \texttt{a.assertion.TAG_UAFV1_REG_ASSERTION} contains another \texttt{TAG_ATTESTATION} tag - verify the attestation by following appropriate processing rules applicable to that attestation. Currently this document defines the processing rules for Basic Attestation and direct anonymous attestation (ECDAA).
19. Extract authenticatorData.attestedCredentialData.credentialPubKey into PublicKey, authenticatorData.attestedCredentialData.credentialID into KeyID, authenticatorData.counter into SignCounter, authenticatorData.attestedCredentialData.AAGUID into AAGUID.
20. Set AuthenticatorVersion to 0 (as it is not included in the message).

4.3 Authentication Response Generation Rules for ASM

See [UAFASM] for details of the ASM API.

1. Locate the authenticator using authenticatorIndex. If the authenticator cannot be located, then fail with UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED.
2. if this is a bound authenticator, verify callerid against the one stored at registration time and return UAF_ASM_STATUS_ACCESS_DENIED if it doesn’t match.
3. Hash the provided AuthenticateIn.finalChallenge using the preferred authenticator-specific hash function (FinalChallengeHash).

   The authenticator’s preferred hash function information MUST meet the algorithm defined in the AuthenticatorInfo.authenticationAlgorithm field.

4. Create an empty list KeyIDRecords of KeyID, related KeyHandle and related username
5. If AuthenticateIn.keyIDs is not empty,
   1. If this is a bound authenticator, then look up ASM’s database with AuthenticateIn.appID and AuthenticateIn.keyIDs and matching entry into KeyIDRecords
      - Return UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY if the related key disappeared permanently from the
authenticator.

- Return \texttt{UAF\_ASM\_STATUS\_ACCESS\_DENIED} if no entry has been found.

2. If this is a roaming authenticator, then for each entry in \texttt{AuthenticateIn.keyIDs} add an entry in \texttt{KeyIDRecords} with \texttt{entry.KeyID} and \texttt{entry.KeyHandle} set to the respective keyID in \texttt{AuthenticateIn.keyIDs}. Set \texttt{entry.userName} to empty.

6. If \texttt{AuthenticateIn.keyIDs} is empty, lookup all \texttt{KeyHandles} matching this request and add an entry in \texttt{KeyIDRecords} with \texttt{entry.KeyID} and \texttt{entry.KeyHandle} set to the respective \texttt{KeyHandles}. Set \texttt{entry.userName} the related \texttt{userName}.

7. If \texttt{KeyIDRecords} contains multiple entries, show the related distinct usernames and ask the user to choose a single username. Remember the \texttt{KeyHandle} and the related \texttt{KeyID} to this key.

8. If \texttt{AuthenticateIn.transaction} is NOT empty then select the entry \texttt{n} with the content type best matching the authenticator capabilities.

   1. if \texttt{AuthenticateIn.transaction[n].contentType == "text/plain"}

      then create a corresponding \texttt{txAuthSimple} extension in \texttt{extensionsCBOR}.

   2. if \texttt{AuthenticateIn.transaction[n].contentType != "text/plain"}

      then create a corresponding \texttt{txAuthGeneric} extension in \texttt{extensionsCBOR}.

9. for each extension included in \texttt{ASMRequest.exts}

   create a corresponding WebAuthn/FIDO2 extension (see \[WebAuthn\]) extension in \texttt{extensionsCBOR}. If no corresponding WebAuthn/FIDO2 extension is specified, ignore this extension.

10. Call authenticatorGetAssertion (either via CTAP or via a platform proprietary API), send the require information and receive the expected result containing the error code of that operation.

\begin{note}
authenticatorGetAssertion has the following input parameters (see \[FIDOCTAP\]):

   1. \texttt{rpId} (required, String). Identity of the relying party.
   2. \texttt{clientDataHash} (required, byte array).
   3. \texttt{allowList} (optional, sequence of PublicKeyCredentialDescriptors).
   4. \texttt{extensions} (optional, CBOR map).
   5. \texttt{options} (optional, sequence of authenticator options, i.e. \texttt{up} for user presence and \texttt{uv} for user verification).
   6. \texttt{pinAuth} (optional, byte array).
   7. \texttt{pinProtocol} (optional, unsigned integer).

The output parameters are (see \[FIDOCTAP\]):

   1. \texttt{credential} (optional, PublicKeyCredentialDescriptor).
   2. \texttt{authData} (required, byte array).
   3. \texttt{signature} (required, byte array).
   4. \texttt{user} (required, PublicKeyCredentialUserEntity).
   5. \texttt{numberOfCredentials} (optional, integer).

Use the following values for the respective parameters:

- Set \texttt{rpId} to the \texttt{ASMRequest.args.AppID}
- Set \texttt{clientDataHash} to \texttt{FinalChallengeHash}
- Set \texttt{allowList} to the \texttt{KeyHandle} remembered earlier
- Set \texttt{extensions} to the CBOR map \texttt{extensionsCBOR}
- Set \texttt{pinAuth} and \texttt{pinProtocol} to the respective values supported by this ASM (to the extent the underlying platform allows specifying these values).
- Set \texttt{options} to an empty object and add items as follows

   1. If extension "UVM" (userVerificationMethod, see \[UAFRegistry\]) is present and \texttt{uvm.userVerificationMethod} includes one or more of the flags \texttt{USER\_VERIFY\_FINGERPRINT, USER\_VERIFY\_PASSCODE, USER\_VERIFY\_VOICEPRINT},
1. If `result` is not equal to \texttt{CTAP2\_OK} and retry cannot fix the problem, then map the CTAP error code to a UAF ASM error code using the table in section 5. Mapping CTAP2 error codes to ASM error codes and return the resulting error code.

2. If the `numberOfCredentials` in the response is \textgreater{} 1, then follow the rules in section "Client Logic" [FIDOCTAP] to receive and process the remaining \texttt{(numberOfCredentials-1)} responses (see \texttt{authenticatorGetNextAssertion} in [FIDOCTAP]).

3. Create \texttt{TAG\_WAV1CBOR\_AUTH\_ASSERTION} structure.
   1. Copy \texttt{AAGUID} (if known) into the respective TLV fields. Otherwise set the field to an empty value (zero length).
   2. Copy the remembered \texttt{KeyID} into the respective TLV field.
   3. Copy \texttt{result.authData} into the value of the \texttt{TAG\_WAV1CBOR\_SIGNED\_DATA} field.
   4. Copy \texttt{result.signature} into the value of the \texttt{TAG\_SIGNATURE} field.

4. Create the \texttt{AuthenticateOut} object
   1. Set \texttt{AuthenticateOut.assertionScheme} to "WAV1CBOR"
   2. Encode the content of \texttt{TAG\_WAV1CBOR\_AUTH\_ASSERTION} in base64url format and set as \texttt{AuthenticateOut.assertion}

5. Set \texttt{ASMResponse.responseData} to \texttt{AuthenticateOut} object.
6. Set \texttt{ASMResponse.statusCode} to the correct status code corresponding to the \texttt{result} received earlier.
7. Set \texttt{ASMResponse.exts} to empty
8. Return \texttt{ASMResponse} object

### 4.4 Authentication Response Processing Rules for FIDO Server

Instead of skipping the assertion according to step 6.5. in section 3.5.7.5 [UAFProtocol], follow these rules:

1. if \texttt{a.assertionScheme} \texttt{==} "WAV1CBOR" AND \texttt{a.assertion} starts with a valid structure as defined in section 3.2 Authentication Assertion, then
   1. set \texttt{tbsData} to the data contained in \texttt{a.assertion.tbsData}.
   2. set \texttt{authenticateData} to the CBOR object \texttt{tbsData} starts with. Use the "length" field of the CBOR object to determine its end.
   3. set \texttt{clientDataHash} to the remaining bytes of the \texttt{tbsData} (i.e. the bytes following the CBOR object).
   4. read \texttt{claimedAAGUID} from \texttt{a.assertion.AAGUID} (note that it might be empty).
   5. read \texttt{claimedKeyID} from \texttt{a.assertion.KeyID}.
   6. Locate \texttt{UAuth.pub} associated with \texttt{(claimedAAGUID, claimedKeyID)} in the user's record. If \texttt{claimedAAGUID} is empty, search for a matching \texttt{claimedKeyID}.
If such record doesn’t exist - continue with next assertion
If multiple records match the search criteria - use the first one
7. if claimedAAGUID is empty, set it to the AAGUID stored along with UAuth.pub
8. Verify that a.assertionScheme matches Metadata(claimedAAGUID).assertionScheme
   - If it doesn’t match - continue with next assertion
9. Verify whether the claimedAAGUID indeed matches the policy of the Authentication Request.
   - If it doesn’t meet the policy – continue with next assertion
10. Check the Signature Counter authenticatorData.SignCounter and make sure it is either not supported by the
    authenticator (i.e. the value provided and the value stored in the user's record are both 0 or the value isKeyRestricted is
    set to ‘false’ in the related Metadata Statement) or it has been incremented (compared to the value stored in the user's
    record)
    - If it is greater than 0, but didn’t increment - continue with next assertion (as this is a cloned authenticator or a cloned
      authenticator has been used previously).
11. Locate authenticator specific authentication algorithms from authenticator metadata (field AuthenticationAlgs)
12. If fcp is of type FinalChallengeParams, then hash AuthenticationResponse.FinalChallengeParams using the hashing
    algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator Metadata, field
    AuthenticationAlgs. It is the hash algorithm associated with the first entry related to a constant with prefix ALG_SIGN.
    - FCHash = hash(AuthenticationResponse.FinalChallengeParams)
13. If fcp is of type CollectedClientData [UAFProtocol], then hash AuthenticationResponse.fcParams using hashing algorithm
    specified in fcp.hashAlg.
    - FCHash = hash(AuthenticationResponse.fcParams)
14. Make sure that clientDataHash == FCHash
    - If comparison fails – continue with next assertion
15. Extract the up and uv bits from authenticatorData. Verify whether these bits match the UVM extension sent in the request.
    Fail if the verification result is not acceptable.

**NOTE**
- up=false and uv=false means silent authentication (USER_VERIFY_NONE)
- up=true and uv=false means user presence check only (USER_VERIFY_PRESENCE)
- up=false and uv=true means user verification that doesn't provide user presence, e.g. client Pin or some other
  user verification method not necessarily implemented fully inside the authenticator boundary
  (USER_VERIFY_CLIENTPIN)
- up=true and uv=true means user verification using a user verification method implemented inside the
  authenticator boundary (e.g. USER_VERIFY_FINGERPRINT, ...) or client Pin plus user presence check
  (USER_VERIFY_CLIENTPIN) AND USER_VERIFY_PRESENCE - depending on the authenticator capabilities as
  declared in the related Metadata Statement.

16. If a UVM extension is included in the response, extract this value and compare it verify whether it matches the extension
    from the request. Fail if the verification result is not acceptable.
17. If authenticatorData contains "txAuthSimple" (see section 10.2 [WebAuthn]) or "txAuthGeneric" (see section 10.3
    [WebAuthn]) extension(s),

**NOTE**
The transaction/transaction hash included in this AuthenticationResponse must match the transaction content
specified in the related AuthenticationRequest. As FIDO doesn’t mandate any specific FIDO Server API, the
transaction content could be cached by any relying party software component, e.g. the FIDO Server or the relying
party Web Application.

1. Make sure there is a transaction cached on Relying Party side.
   - If not – continue with next assertion
2. Go over all cached forms of the transaction content (potentially multiple cached PNGs for the same transaction) and
  calculate their hashes using hashing algorithm suitable for this authenticator (same hash algorithm as used for
FinalChallenge).

- For each cachedTransaction add \(\text{hash}(\text{cachedTransaction})\) into cachedTransactionHashList

3. Make sure that the transaction ("txAuthSimple") or the transaction hash ("txAuthGeneric") included in the extension is in cachedTransactionHashList
- If it's not in the list – continue with next assertion

18. Use the \(\text{UAuth.pub}\) key found in step 1.9 and the appropriate authentication algorithm to verify the signature \(a . \text{assertion}. \text{Signature}\) of the to-be-signed object \(\text{tbsData}\).
   1. If signature verification fails – continue with next assertion
   2. Update \(\text{SignCounter}\) in user's record with \(\text{authenticatorData}. \text{SignCounter}\).

### NOTE
The values of \(\text{claimedAAGUID}\) and \(\text{claimedKeyID}\) are now confirmed since the public key we looked up using those values was the correct one.

5. Mapping CTAP2 error codes to ASM error codes

In many cases the status code returned via [FIDOCTAP] needs to be processed and handled by the ASM. If the communication to the authenticator via [FIDOCTAP] finally failed with an error, the following error code mapping rules apply:

<table>
<thead>
<tr>
<th>CTAP2 Code</th>
<th>CTAP2 Name</th>
<th>ASM Error Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>CTAP1_ERR_SUCCESS, CTAP2_OK</td>
<td>UAF_ASM_STATUS_OK</td>
</tr>
<tr>
<td>0x01</td>
<td>CTAP1_ERR_INVALID_COMMAND</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x02</td>
<td>CTAP1_ERR_INVALID_PARAMETER</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x03</td>
<td>CTAP1_ERR_INVALID_LENGTH</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x04</td>
<td>CTAP1_ERR_INVALID_SEQ</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x05</td>
<td>CTAP1_ERR_TIMEOUT</td>
<td>UAF_ASM_STATUS_USER_NOT_RESPONSIVE</td>
</tr>
<tr>
<td>0x06</td>
<td>CTAP1_ERR_CHANNEL_BUSY</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x0A</td>
<td>CTAP1_ERR_LOCK_REQUIRED</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x0B</td>
<td>CTAP1_ERR_INVALID_CHANNEL</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x11</td>
<td>CTAP2_ERR_CBOR_UNEXPECTED_TYPE</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x12</td>
<td>CTAP2_ERR_INVALID_CBOR</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x14</td>
<td>CTAP2_ERR_MISSING_PARAMETER</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x15</td>
<td>CTAP2_ERR_LIMIT_EXCEEDED</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x16</td>
<td>CTAP2_ERR_UNSUPPORTED_EXTENSION</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x19</td>
<td>CTAP2_ERR_CREDENTIAL_EXCLUDED</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x21</td>
<td>CTAP2_ERR_PROCESSING</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x22</td>
<td>CTAP2_ERR_INVALID_CREDENTIAL</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x23</td>
<td>CTAP2_ERR_USER_ACTION_PENDING</td>
<td>UAF_ASM_STATUS_USER_NOT_RESPONSIVE</td>
</tr>
<tr>
<td>0x24</td>
<td>CTAP2_ERR_OPERATION_PENDING</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x25</td>
<td>CTAP2_ERR_NO_OPERATIONS</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>Code</td>
<td>Error Code</td>
<td>Status Code</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>0x26</td>
<td>CTAP2_ERR_UNSUPPORTED_ALGORITHM</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x27</td>
<td>CTAP2_ERR_OPERATION_DENIED</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x28</td>
<td>CTAP2_ERR_KEY_STORE_FULL</td>
<td>UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES</td>
</tr>
<tr>
<td>0x2A</td>
<td>CTAP2_ERR_NO_OPERATION_PENDING</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x2B</td>
<td>CTAP2_ERR_UNSUPPORTED_OPTION</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x2C</td>
<td>CTAP2_ERR_INVALID_OPTION</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x2D</td>
<td>CTAP2_ERR_KEEPALIVE_CANCEL</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x2E</td>
<td>CTAP2_ERR_NO_CREDENTIALS</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x2F</td>
<td>CTAP2_ERR_USER_ACTION_TIMEOUT</td>
<td>UAF_ASM_STATUS_USER_NOT_RESPONSIVE</td>
</tr>
<tr>
<td>0x30</td>
<td>CTAP2_ERR_NOT_ALLOWED</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0x31</td>
<td>CTAP2_ERR_PIN_INVALID</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x32</td>
<td>CTAP2_ERR_PIN_BLOCKED</td>
<td>UAF_ASM_STATUS_USER_LOCKOUT</td>
</tr>
<tr>
<td>0x33</td>
<td>CTAP2_ERR_PIN_AUTH_INVALID</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x34</td>
<td>CTAP2_ERR_PIN_AUTH_BLOCKED</td>
<td>UAF_ASM_STATUS_USER_LOCKOUT</td>
</tr>
<tr>
<td>0x35</td>
<td>CTAP2_ERR_PIN_NOT_SET</td>
<td>UAF_ASM_STATUS_USER_NOT_ENROLLED</td>
</tr>
<tr>
<td>0x36</td>
<td>CTAP2_ERR_PIN_REQUIRED</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x37</td>
<td>CTAP2_ERR_PIN_POLICY_VIOLATION</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x38</td>
<td>CTAP2_ERR_PIN_TOKEN_EXPIRED</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x39</td>
<td>CTAP2_ERR_REQUEST_TOO_LARGE</td>
<td>UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES</td>
</tr>
<tr>
<td>0x3A</td>
<td>CTAP2_ERR_ACTION_TIMEOUT</td>
<td>UAF_ASM_STATUS_USER_NOT_RESPONSIVE</td>
</tr>
<tr>
<td>0x3B</td>
<td>CTAP2_ERR_UP_REQUIRED</td>
<td>UAF_ASM_STATUS_ACCESS_DENIED</td>
</tr>
<tr>
<td>0x7F</td>
<td>CTAP1_ERR_OTHER</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0xDF</td>
<td>CTAP2_ERR_SPEC_LAST</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0xE0</td>
<td>CTAP2_ERR_EXTENSION_FIRST</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0xEF</td>
<td>CTAP2_ERR_EXTENSION_LAST</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0xF0</td>
<td>CTAP2_ERR_VENDOR_FIRST</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
<tr>
<td>0xFF</td>
<td>CTAP2_ERR_VENDOR_LAST</td>
<td>UAF_ASM_STATUS_ERROR</td>
</tr>
</tbody>
</table>

A. References

A.1 Normative references

[FIDOCTAP]

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. FIDO ECDAA Algorithm. 28 November 2017.