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 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
- 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

This section is normative.

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>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>Length of the structure.</td>
</tr>
<tr>
<td>1.2.2</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>Length of AAID</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Authenticator Attestation ID</td>
</tr>
<tr>
<td>1.4</td>
<td>TAG_KEYID</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Length of KeyID</td>
</tr>
<tr>
<td>1.4.2</td>
<td>(binary value of) KeyID</td>
</tr>
</tbody>
</table>
### KeyID

<table>
<thead>
<tr>
<th>1.5</th>
<th>UINT16 Tag</th>
<th>TAG_SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5.1</td>
<td>UINT16 Length</td>
<td>Length of Signature</td>
</tr>
<tr>
<td>1.5.2</td>
<td>UINT8[] Signature</td>
<td>Signature calculated using UAuth.priv over tbsData - not including any TAGs nor the KeyID and AAID.</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 [UFAuthnrCommands] 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 [UFAuthnrCommands]), 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. 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](https://developer.android.com/training/articles/security-android-protected-confirmation).

6. Create a `TAGAUTHENTICATORASSERTION` structure containing a `TAG_UAFV1_REGASSERTION` 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 `TAGAUTHENTICATORASSERTION` (i.e. `TAG_UAFV1_REGASSERTION`) 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_REGASSERTION` contains `TAG_UAFV1_KRD` as first element:
   1. Obtain `Metadata(AAID).AttestationType` for the AAID and make sure that `a.assertion.TAG_UAFV1_REGASSERTION` contains the most preferred attestation tag specified in field `MatchCriteria.attestationTokenTypes` in `RegistrationRequest.policy` (if this field is present).
     - If `a.assertion.TAG_UAFV1_REGASSERTION` 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_REGASSERTION.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_REGASSERTION.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 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.
Return if no entry has been found.

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.contentType 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.
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.
   
   \[
   FCHash = \text{hash(AuthenticationResponse.FinalChallengeParams)}
   \]

6. If fcp is of type ClientData, then hash AuthenticationResponse.fcParams using hashing algorithm specified in fcp.hashAlg.
   
   \[
   FCHash = \text{hash(AuthenticationResponse.fcParams)}
   \]

7. Make sure that tbsData.dataThatWasConfirmed.extraData == FHash
   
   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

### NOTE

The **promptText** 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.

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]

[UAF ASM]
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

[UAF Authn Commands]

[UAF Protocol]

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

A.2 Informative references

[FIDOMetadataService]