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.

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 Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

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

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

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

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

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

1.1 Key Words

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

2. Overview

This section is non-normative.

This document 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.

	TLV Structure	Description
1	UINT16 Tag	TAG_WAV1CBOR_REG_ASSERTION
1.1	UINT16 Length	Length of the structure.
1.2	UINT16 Tag	TAG_WAV1CBOR_REG_DATA
1.2.1	UINT16 Length	Length of the structure.
1.2.2	UINT8 tbsData	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`.

1.3	UINT16 Tag	TAG_ATTESTATION_FORMAT
1.3.1	UINT16 Length	Length of Attestation Format
1.3.2	UINT8[] Attestation Format	Authenticator Attestation Format, see field "fmt" in section sctn-attestation in [WebAuthn]
1.4	UINT16 Tag	TAG_ATTESTATION_STATEMENT
1.4.1	UINT16 Length	Length of Attestation Statement
1.4.2	UINT8[] 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.

TLV Structure		Description
1	UINT16 Tag	TAG_WAV1CBOR_AUTH_ASSERTION
1.1	UINT16 Length	Length of the structure.
1.2	UINT16 Tag	TAG_WAV1CBOR_SIGNED_DATA
1.2.1	UINT16 Length	Length of the structure.
1.2.2	UINT8 tbsData	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`.
1.3	UINT16 Tag	TAG_AAGUID
1.3.1	UINT16 Length	Length of AAGUID
1.3.2	UINT8[] AAGUID	Authenticator Attestation GUID, see section 6.4.1 in [WebAuthn]
1.4	UINT16 Tag	TAG_KEYID
1.4.1	UINT16 Length	Length of KeyID
1.4.2	UINT8[] KeyID	(binary value of) Credential ID (see definition of CredentialID in [WebAuthn])
1.5	UINT16 Tag	TAG_SIGNATURE
1.5.1	UINT16 Length	Length of Signature

1.5.2	UINT8[] Signature	Signature calculated using UAuth.priv over tbsData - *not* including any TAGs nor the KeyID and AAGUID.

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.

2. fmt (required, String). The attestation statement format identifier.
3. attStmt (required, sequence of bytes). The attestation statement.

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_PASSCODE`, `USER_VERIFY_VOICEPRINT`, `USER_VERIFY_FACEPRINT`, `USER_VERIFY_LOCATION`, `USER_VERIFY_EYEPRINT`, `USER_VERIFY_PATTERN`, 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 eequal 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`

- If it doesn't match - continue with next assertion

4. Verify that the `claimedAAGUID` 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 AAGUID matches the policy.

   - If it doesn't match the policy - continue with next assertion

5. Locate authenticator-specific authentication algorithms from the authenticator metadata [FIDOMetadataStatement] identified by `claimedAAGUID` (field `authenticationAlgs`).
6. If `fcp` is of type FinalChallengeParams [UAFProtocol], 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)`
7. If `fcp` is of type CollectedClientData [UAFProtocol], then hash `RegistrationResponse.fcParams` using hashing algorithm specified in `fcp.hashAlg`.
   - `FCHash = hash(RegistrationResponse.fcParams)`
8. Obtain `Metadata(claimedAAGUID).AttestationType` for the `claimedAAGUID` and make sure that `a.assertion.TAG_WAV1CBOR_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_WAV1CBOR_REG_ASSERTION` doesn't contain the preferred attestation - it is RECOMMENDED to skip this assertion and continue with next one
9. set `tbsData` to the data contained in `a.assertion.tbsData`.
10. set `authenticatorData` to the CBOR object `tbsData` starts with. Use the "length" field of the CBOR object to determine its end.
11. set `clientDataHash` to the remaining bytes of the `tbsData` (i.e. the bytes following the CBOR object).
12. Make sure that `clientDataHash` == `FCHash`
    - If comparison fails - continue with next assertion
13. 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 check, 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.

14. 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.
15. If `a.assertion.TAG_WAV1CBOR_REG_ASSERTION.TAG_ATTESTATION_STATEMENT` contains `ATTESTATION_BASIC_FULL` tag
    1. If entry `AttestationRootCertificates` for the claimedAAGUID in the metadata [FIDOMetadataStatement] contains at least one element:
       1. Obtain contents of all `TAG_ATTESTATION_CERT` tags from `a.assertion.TAG_WAV1CBOR_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 claimedAAGUID 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_WAV1CBOR_REG_ASSERTION.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 `a.assertion.TAG_WAV1CBOR_REG_ASSERTION.TAG_ATESTATION_STATEMENT` contains an object of type `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 `a.assertion.TAG_WAV1CBOR_REG_ASSERTION` contains an object of type `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 `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).
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 `UAF_ASM_STATUS_ACCESS_DENIED` if no entry has been found.

2. If this is a roaming authenticator, then for each entry in `AuthenticateIn.keyIDs` add an entry in `KeyIDRecords` with `entry.KeyID` and `entry.KeyHandle` set to the respective keyID in `AuthenticateIn.keyIDs`. Set `entry.userName` to empty.

6. If `AuthenticateIn.keyIDs` is empty, lookup all `KeyHandles` matching this request and add an entry in `KeyIDRecords` with `entry.KeyID` and `entry.KeyHandle` set to the respective `KeyHandles`. Set `entry.userName` the related userName.
7. If `KeyIDRecords` containes multiple entries, show the related distinct usernames and ask the user to choose a single username. Remember the `KeyHandle` and the related `KeyID` to this key.
8. If `AuthenticateIn.transaction` is NOT empty then select the entry `n` with the content type best matching the authenticator capabilities.
   1. if `AuthenticateIn.transaction[n].contentType` == "text/plain"

      then create a corresponding `txAuthSimple` extension in `extensionsCBOR`.

   2. if `AuthenticateIn.transaction[n].contentType` != "text/plain"

      then create a corresponding `txAuthGeneric` extension in `extensionsCBOR`.

9. for each extension included in `ASMRequest.exts`

   create a corresponding WebAuthn/FIDO2 extension (see [WebAuthn]) extension in `extensionsCBOR`. If no corrsponding 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.

    > NOTE
    >
    > authenticatorGetAssertion 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` for user presence and `uv` for user verification).
    > 6. pinAuth (optional, byte array).
    > 7. pinProtocol (optional, unsigned integer).
    >
    > 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).

    Use the following values for the respective parameters:

    - Set `rpId` to the `ASMRequest.args.AppID`
    - Set `clientDataHash` to `FinalChallengeHash`
    - Set `allowList` to the `KeyHandle` remembered earlier
    - 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_PASSCODE`, `USER_VERIFY_VOICEPRINT`,

USER_VERIFY_FACEPRINT, USER_VERIFY_LOCATION, USER_VERIFY_EYEPRINT, USER_VERIFY_PATTERN, or USER_VERIFY_HANDPRINT set options.uv to true and set options.up to true.

2. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and uvm.userVerificationMethod is equal to USER_VERIFY_CLIENTPIN set options.uv to true and set options.up to false. Remember to provide the clientPIN to the authenticator.
3. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and uvm.userVerificationMethod is equal to USER_VERIFY_PRESENCE set options.uv to false and set options.up to true.
4. If extension "UVM" (userVerificationMethod, see [UAFRegistry]) is present and uvm.userVerificationMethod is equal to USER_VERIFY_NONE set options.uv to false and set options.up 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.

11. 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.
12. If the numberOfCredentials in the response is > 1, then follow the rules in section "Client Logic" [FIDOCTAP] to receive and process the remaining (numberOfCredentials-1) responses (see authenticatorGetNextAssertion in [FIDOCTAP]).
13. Create TAG_WAV1CBOR_AUTH_ASSERTION structure.
    1. Copy AAGUID (if known) into the respective TLV fields. Otherwise set the field to an empty value (zero length).

       > NOTE
       >
       > In the case of a platform authenticator, the AAGUID value can be remembered at registration time. In the case of a roaming authenticator, it might be possible to call authenticatorGetInfo [FIDOCTAP] which provides the AAGUID in the response.

    2. Copy the remembered KeyID into the respective TLV field.
    3. Copy result.authData into the value of the TAG_WAV1CBOR_SIGNED_DATA field.
    4. Copy result.signature into the value of the TAG_SIGNATURE field.
14. Create the AuthenticateOut object
    1. Set AuthenticateOut.assertionScheme to "WAV1CBOR"
    2. Encode the content of TAG_WAV1CBOR_AUTH_ASSERTION in base64url format and set as AuthenticateOut.assertion
15. set ASMResponse.responseData to AuthenticateOut object.
16. set ASMResponse.statusCode to the correct status code corresponding to the result received earlier.
17. set ASMResponse.exts to empty
18. Return 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 a.assertionScheme == "WAV1CBOR" AND a.assertion starts with a valid structure as defined in section 3.2 Authentication Assertion, then
    1. set tbsData to the data contained in a.assertion.tbsData.
    2. set authenticatorData to the CBOR object tbsData starts with. Use the "length" field of the CBOR object to determine its end.
    3. set clientDataHash to the remaining bytes of the tbsData (i.e. the bytes following the CBOR object).
    4. read claimedAAGUID from a.assertion.AAGUID (note that it might be empty).
    5. read claimedKeyID from a.assertion.KeyID.
    6. Locate UAuth.pub associated with (claimedAAGUID, claimedKeyID) in the user's record. If claimedAAGUID is empty, search for a matching 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 `hash(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 `UAuth.pub` key found in step 1.9 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
    2. Update `SignCounter` in user's record with `authenticatorData.SignCounter`.

> NOTE
>
> The values of `claimedAAGUID` and `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:

CTAP2 Code	CTAP2 Name	ASM Error Name
0x00	CTAP1_ERR_SUCCESS, CTAP2_OK	UAF_ASM_STATUS_OK
0x01	CTAP1_ERR_INVALID_COMMAND	UAF_ASM_STATUS_ERROR
0x02	CTAP1_ERR_INVALID_PARAMETER	UAF_ASM_STATUS_ERROR
0x03	CTAP1_ERR_INVALID_LENGTH	UAF_ASM_STATUS_ERROR
0x04	CTAP1_ERR_INVALID_SEQ	UAF_ASM_STATUS_ERROR
0x05	CTAP1_ERR_TIMEOUT	UAF_ASM_STATUS_USER_NOT_RESPONSIVE
0x06	CTAP1_ERR_CHANNEL_BUSY	UAF_ASM_STATUS_ERROR
0x0A	CTAP1_ERR_LOCK_REQUIRED	UAF_ASM_STATUS_ERROR
0x0B	CTAP1_ERR_INVALID_CHANNEL	UAF_ASM_STATUS_ERROR
0x11	CTAP2_ERR_CBOR_UNEXPECTED_TYPE	UAF_ASM_STATUS_ERROR
0x12	CTAP2_ERR_INVALID_CBOR	UAF_ASM_STATUS_ERROR
0x14	CTAP2_ERR_MISSING_PARAMETER	UAF_ASM_STATUS_ERROR
0x15	CTAP2_ERR_LIMIT_EXCEEDED	UAF_ASM_STATUS_ERROR
0x16	CTAP2_ERR_UNSUPPORTED_EXTENSION	UAF_ASM_STATUS_ERROR
0x19	CTAP2_ERR_CREDENTIAL_EXCLUDED	UAF_ASM_STATUS_ERROR
0x21	CTAP2_ERR_PROCESSING	UAF_ASM_STATUS_ERROR
0x22	CTAP2_ERR_INVALID_CREDENTIAL	UAF_ASM_STATUS_ERROR
0x23	CTAP2_ERR_USER_ACTION_PENDING	UAF_ASM_STATUS_USER_NOT_RESPONSIVE
0x24	CTAP2_ERR_OPERATION_PENDING	UAF_ASM_STATUS_ERROR
0x25	CTAP2_ERR_NO_OPERATIONS	UAF_ASM_STATUS_ERROR

0x26	CTAP2_ERR_UNSUPPORTED_ALGORITHM	UAF_ASM_STATUS_ERROR
0x27	CTAP2_ERR_OPERATION_DENIED	UAF_ASM_STATUS_ACCESS_DENIED
0x28	CTAP2_ERR_KEY_STORE_FULL	UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES
0x2A	CTAP2_ERR_NO_OPERATION_PENDING	UAF_ASM_STATUS_ERROR
0x2B	CTAP2_ERR_UNSUPPORTED_OPTION	UAF_ASM_STATUS_ERROR
0x2C	CTAP2_ERR_INVALID_OPTION	UAF_ASM_STATUS_ERROR
0x2D	CTAP2_ERR_KEEPALIVE_CANCEL	UAF_ASM_STATUS_ERROR
0x2E	CTAP2_ERR_NO_CREDENTIALS	UAF_ASM_STATUS_ERROR
0x2F	CTAP2_ERR_USER_ACTION_TIMEOUT	UAF_ASM_STATUS_USER_NOT_RESPONSIVE
0x30	CTAP2_ERR_NOT_ALLOWED	UAF_ASM_STATUS_ERROR
0x31	CTAP2_ERR_PIN_INVALID	UAF_ASM_STATUS_ACCESS_DENIED
0x32	CTAP2_ERR_PIN_BLOCKED	UAF_ASM_STATUS_USER_LOCKOUT
0x33	CTAP2_ERR_PIN_AUTH_INVALID	UAF_ASM_STATUS_ACCESS_DENIED
0x34	CTAP2_ERR_PIN_AUTH_BLOCKED	UAF_ASM_STATUS_USER_LOCKOUT
0x35	CTAP2_ERR_PIN_NOT_SET	UAF_ASM_STATUS_USER_NOT_ENROLLED
0x36	CTAP2_ERR_PIN_REQUIRED	UAF_ASM_STATUS_ACCESS_DENIED
0x37	CTAP2_ERR_PIN_POLICY_VIOLATION	UAF_ASM_STATUS_ACCESS_DENIED
0x38	CTAP2_ERR_PIN_TOKEN_EXPIRED	UAF_ASM_STATUS_ACCESS_DENIED
0x39	CTAP2_ERR_REQUEST_TOO_LARGE	UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES
0x3A	CTAP2_ERR_ACTION_TIMEOUT	UAF_ASM_STATUS_USER_NOT_RESPONSIVE
0x3B	CTAP2_ERR_UP_REQUIRED	UAF_ASM_STATUS_ACCESS_DENIED
0x7F	CTAP1_ERR_OTHER	UAF_ASM_STATUS_ERROR
0xDF	CTAP2_ERR_SPEC_LAST	UAF_ASM_STATUS_ERROR
0xE0	CTAP2_ERR_EXTENSION_FIRST	UAF_ASM_STATUS_ERROR
0xEF	CTAP2_ERR_EXTENSION_LAST	UAF_ASM_STATUS_ERROR
0xF0	CTAP2_ERR_VENDOR_FIRST	UAF_ASM_STATUS_ERROR
0xFF	CTAP2_ERR_VENDOR_LAST	UAF_ASM_STATUS_ERROR

A. References

A.1 Normative references

[FIDOCTAP]
C. Brand; A. Czeskis; J. Ehrensvärd; M. Jones; A. Kumar; R. Lindemann; A. Powers; J. Verrept. *FIDO 2.0: Client To Authenticator Protocol*. 30 January 2019. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAA Algorithm*. 28 November 2017.

Review Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.html

[FIDOGlossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Review Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-glossary-v2.0-id-20180227.html

[FIDOMetadataStatement]
B. Hill; D. Baghdasaryan; J. Kemp. *FIDO Metadata Statements*. Review Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-metadata-statement-v2.0-id-20180227.html

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC5280]
D. Cooper; S. Santesson; S. Farrell; S. Boeyen; R. Housley; W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. URL: https://tools.ietf.org/html/rfc5280

[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

[UAFAuthnrCommands]
D. Baghdasaryan; J. Kemp; R. Lindemann; R. Sasson; B. Hill; J. Hodges; K. Yang. *FIDO UAF Authenticator Commands*. Review Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-authnr-cmds-v1.2-ps-20201020.html

[UAFProtocol]
R. Lindemann; D. Baghdasaryan; E. Tiffany; D. Balfanz; B. Hill; J. Hodges; K. Yang. *FIDO UAF Protocol Specification v1.2*. Review Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-protocol-v1.2-ps-20201020.html

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

[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/