REVIEW DRAFT

FIDO UAF Protocol Specification

FIDO Alliance Review Draft 28 November 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
Editors:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Eric Tiffany, FIDO Alliance
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Dirk Balfanz, Google, Inc.
Brad Hill, PayPal, Inc.
Jeff Hodges, PayPal, Inc.
Ka Yang, Nok Nok Labs, Inc.

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://www.fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Review Draft. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

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.

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 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 [FIDOGlossary].

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

This section is non-normative.

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.

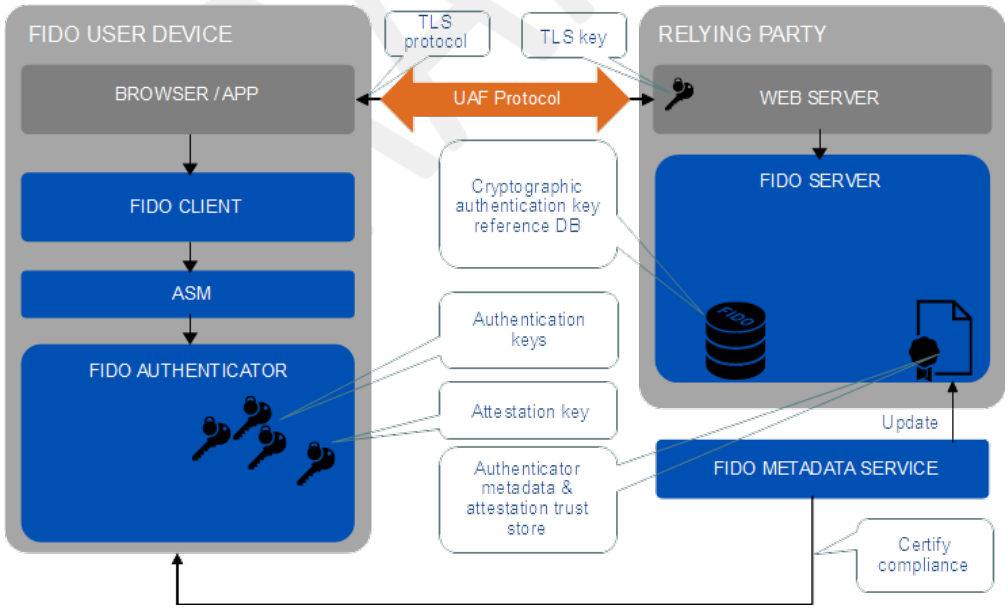

Fig. 1 The UAF Architecture

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.

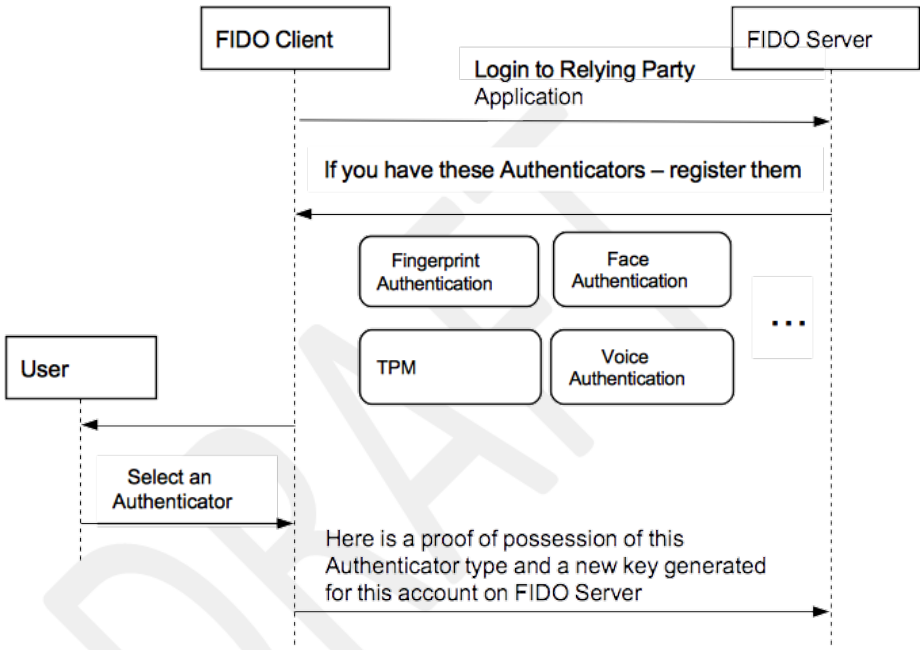

Fig. 2 UAF Registration Message Flow

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 the FIDO UAF Client to do some "housekeeping" tasks.

2.3.2 Authentication

The following diagram depicts the message flows for the authentication operation.

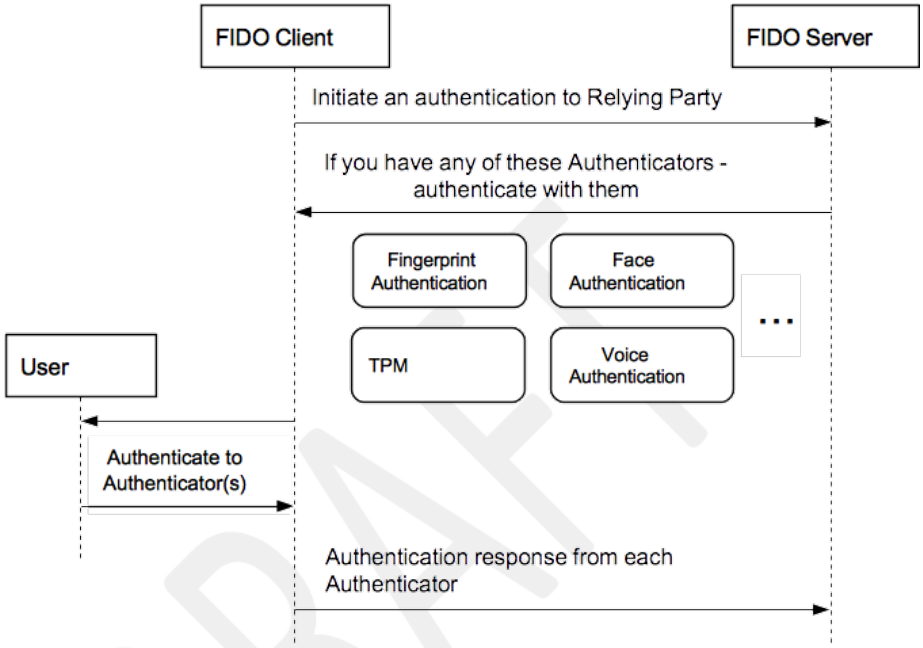

Fig. 3 Authentication Message Flow

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.

2.3.3 Transaction Confirmation

The following figure depicts the transaction confirmation message flow.

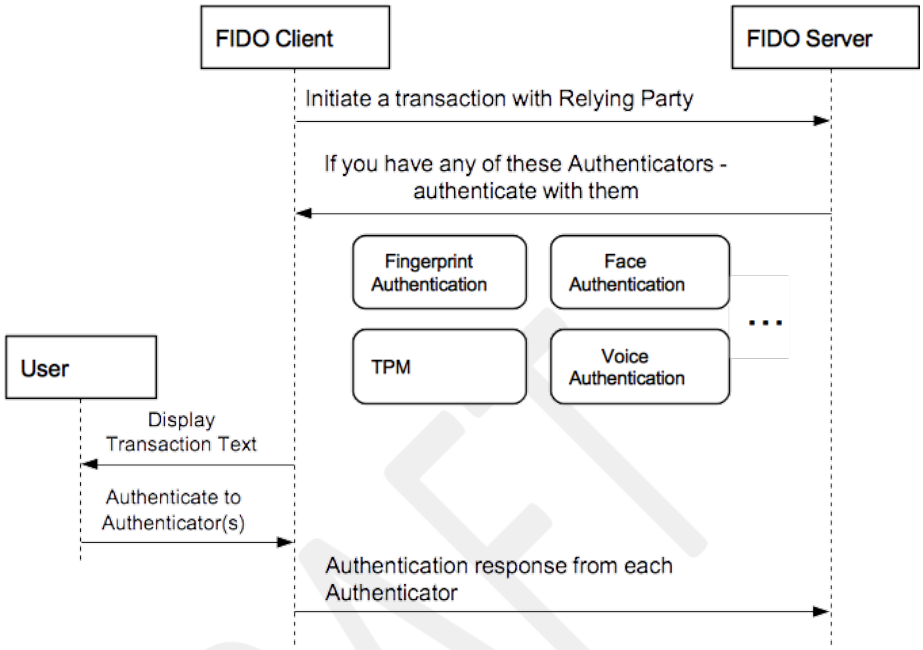

Fig. 4 Transaction Confirmation Message Flow

> 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 the FIDO UAF Client to do some "housekeeping" tasks.

2.3.4 Deregistration

The following diagram depicts the deregistration message flow.

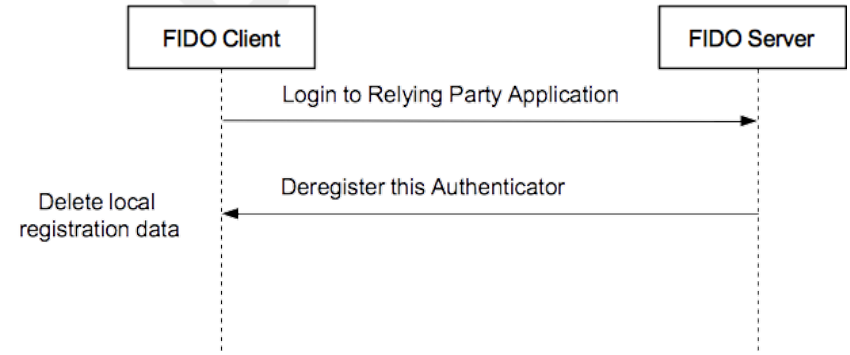

Fig. 5 Deregistration Message Flow

> 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 the FIDO UAF Client to do some "housekeeping" tasks.

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 `U+0000..U+007F`.

Unless otherwise specified, protocol messages are transferred with a UTF-8 content encoding.

> NOTE
>
> 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"
};
```

Enumeration description	
`Reg`	Registration
`Auth`	Authentication or Transaction Confirmation
`Dereg`	Deregistration

3.1.3 OperationHeader dictionary

Represents a UAF message Request and Response header

WebIDL

```
dictionary OperationHeader {
    required Version    upv;
    required Operation  op;
    DOMString           appID;
    DOMString           serverData;
    Extension[]         exts;
};
```

3.1.3.1 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 `UAuth.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 `AuthenticatorRegistrationAssertion.assertion.KeyID` during the registration operation (see also section ServerData and KeyHandle) 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 [UAFAuthnrCommands] 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 ClientData dictionary

ClientData is an alternative to the **`FinalChallengeParams`** structure. It is used by platforms supporting CTAP2 and Web Authentication. The exact definition of clientData can be found in [WebAuthn].

NOTE

WebIDL

```
dictionary ClientData {
    required DOMString           challenge;
    required DOMString           origin;
    required AlgorithmIdentifier hashAlg;
    DOMString                    tokenBinding;
    WebAuthnExtensions           extensions;
};
```

Dictionary `ClientData` Members

`challenge` of type required DOMString

Contains the base64url encoding of the challenge provided by the RP.

This field plays a similar role as the `challenge` field in `FinalChallengeParams`.

`origin` of type required DOMString

The fully qualified origin of the requester, as provided to the authenticator by the client, in the synrax defined by [RFC6454].

This field plays a similar role as the `facetID` field in `FinalChallengeParams`.

`hashAlg` of type required AlgorithmIdentifier

The hash algorithm used to compute the clientDataHash, e.g. "S256", etc.

This field is relevant here as the client can freely select the hash algorithm - unlike `FinalChallengeParams`, where the authenticator must use the same algorithm as for signing the assertion.

`tokenBinding` of type DOMString

Contains the base64url encoding of the Token Binding ID provided by the client. The syntax is equivalent to the cid_pubkey in section ChannelBinding dictionary.

This field plays a similar role as the `channelBinding` field in `FinalChallengeParams`.

`extensions` of type WebAuthnExtensions

Additional parameters generated by processing of extensions passed in by the relying party.

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:

- 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 Servermay reject operations if the channel binding cannot be verified successfully.

NOTE

- If channel binding data 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 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.

WebIDL

```
dictionary ChannelBinding {
    DOMString serverEndPoint;
    DOMString tlsServerCertificate;
    DOMString tlsUnique;
    DOMString cid_pubkey;
};
```

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 [RFC6234], 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.

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):

```
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:

1. if `MetadataStatement.userVerificationDetails` contains multiple entries, then:
    1. if one or more entries `MetadataStatement.userVerificationDetails[i]` contain multiple entries, then: stop, direct derivation is not possible. Must generate `MatchCriteria` object by providing a list of matching AAIDs.
    2. if all entries `MetadataStatement.userVerificationDetails[i]` only contain a single entry, then: combine all entries `MetadataStatement.userVerificationDetails[0][0].userVerification` to `MetadataStatement.userVerificationDetails[N-1][0].userVerification` into a single value using a bitwise OR operation.
2. if `MetadataStatement.userVerificationDetails` contains a single entry, then: combine all entries `MetadataStatement.userVerificationDetails[0][0].userVerification` to `MetadataStatement.userVerificationDetails[0][N-1].userVerification` into a single value using a bitwise OR operation and (if multiple bit flags have been set) additionally set the flag `USER_VERIFY_ALL`.

This method doesn't allow matching authenticators implementing complex combinations of user verification methods, such as `PIN AND (Fingerprint OR Speaker Recognition)` (see above derivation rules). If such specific match rules are required, they need to be specified by providing the AAIDs of the matching authenticators.

`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 auhentication 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

```
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
>
> ```
> {
>   "accepted":
>   [
>  [{ "userVerification": 1023 }]
>   ]
> }
> ```

> NOTE
>
> 1023 = 0x3ff = USER_VERIFY_PRESENCE | USER_VERIFY_FINGERPRINT | ... | USER_VERIFY_NONE

`disallowed` of type array of *MatchCriteria*
Any authenticator that matches any of MatchCriteria contained in the field disallowed must be excluded from eligibility for the operation, regardless of whether it matches any MatchCriteria present in the `accepted` list, or not.

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
>
> ```
> {
> ```

```
    "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.

```
{
  "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.

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"]}] ]
}
```

The next example requires two authenticators to be used:

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"]}
 ]
  ]
}
```

Other criteria can be specified in addition to the `userVerification`:

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"]}
      ]
  ]
}
```

The policy for accepting authenticators of vendor with ID`1234` only is as follows:

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.

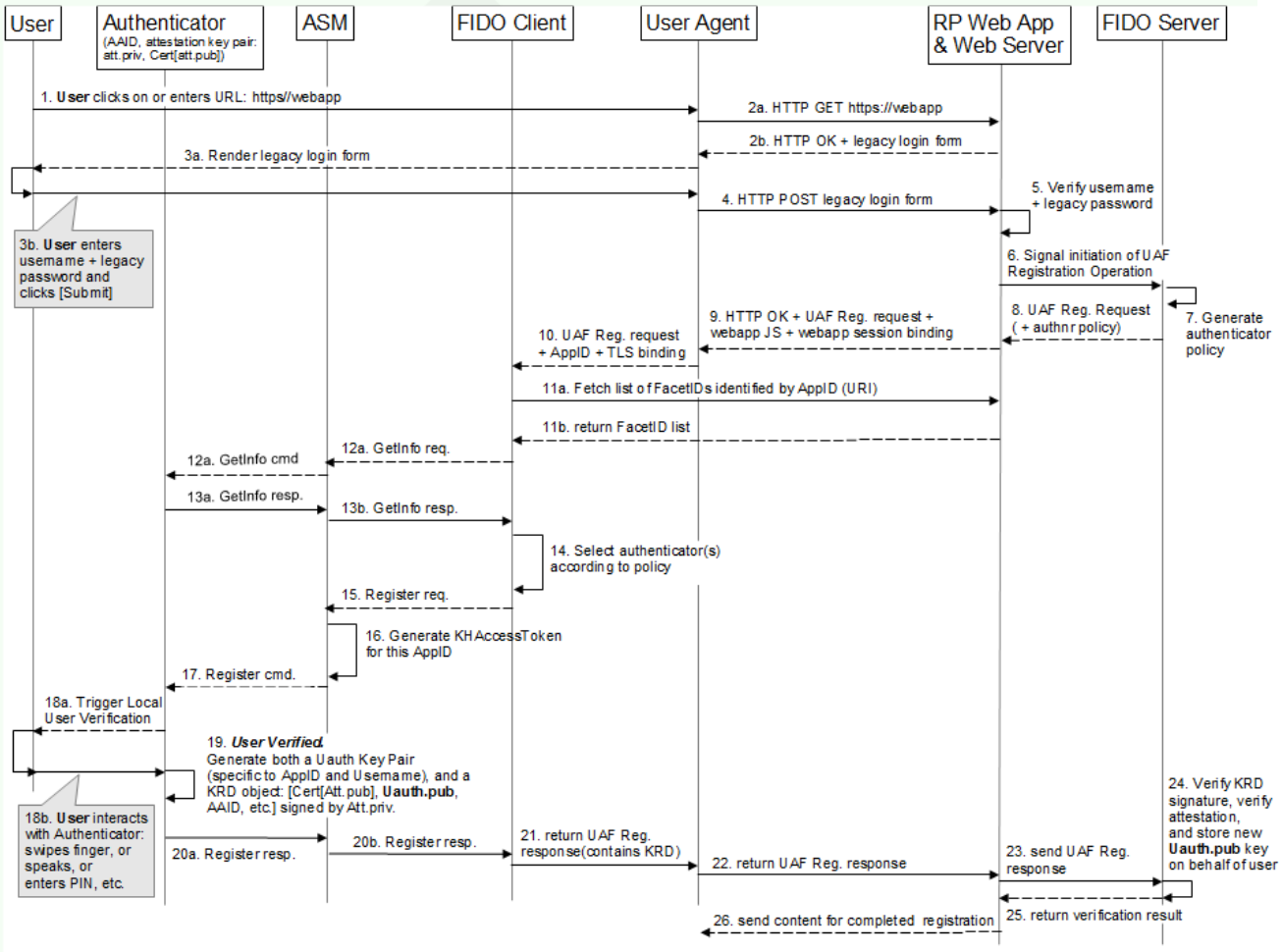

Fig. 6 UAF Registration Sequence Diagram

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.

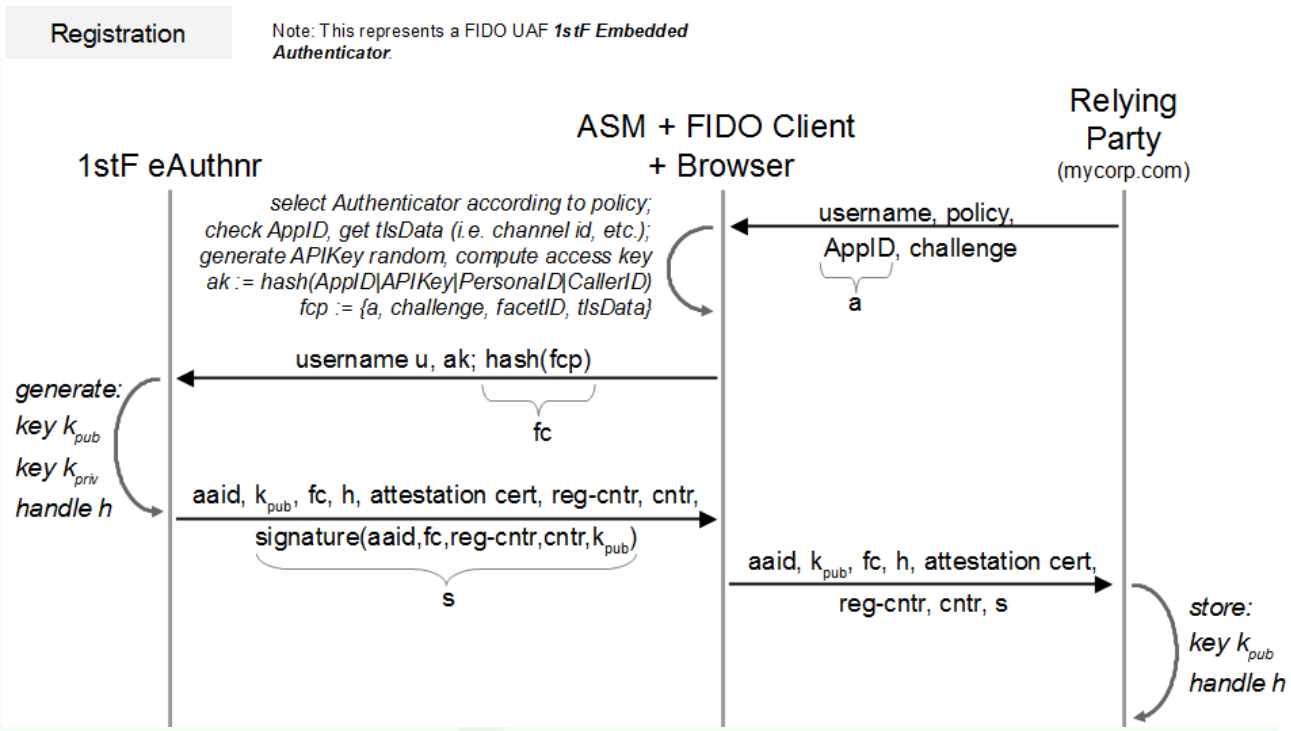

Fig. 7 UAF Registration Cryptographic Data Flow

The FIDO Server sends the `AppID` (see section AppID and FacetID Assertion), the authenticator Policy, the **`ServerChallenge`** and the `Username` to the FIDO UAF Client.

The FIDO UAF Client computes the **`FinalChallengeParams`** (FCP) from the **`ServerChallenge`** and some other values and sends the `AppID`, the `FCH` and the `Username` to the authenticator.

The ASM computes the finalChallengeHash (`FCH`) and calls the authenticator. The authenticator creates a Key Registration Data object (e.g. `TAG_UAFV1_KRD`, see [UAFAuthnrCommands]) containing the hash of `FCH`, the newly generated user public key (UAuth.pub) and some other values and signs it (see section Authenticator Attestation for more details). This KRD object is then cryptographically verified by the FIDO Server.

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

```
[{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Reg",
        "appID": "https://uaf.example.com/facets.json",
        "serverData": "ZQ_fRGDH2ar_LvrTM8JnQcl-wfnaOutiyCmpBgmMcuE"
    },
    "challenge": "Yb39SdUhU2B0089pS5L7VBW8afdlplnvR4B1Ana5vk4",
    "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"]
            }],
            [{
                "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"]
            }]
        ],
        "disallowed": [
            {
                "userVerification": 512,
                "keyProtection": 16,
                "assertionSchemes": ["UAFV1TLV"]
            },
            {
                "userVerification": 256,
                "keyProtection": 16
            },
            {
                "aaid":   ["FFFF#FC02"],
                "keyIDs": ["RfY_RDhsf4z5PCOhnZExMeVloZZmK0hxaSi10tkY_c4"]
            }
        ]
    }
}]
```

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.

`policy` of type required Policy
: Describes which types of authenticators are acceptable for this registration operation

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.

> 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 `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": "eyJmYWNldElEIjoiaHR0cHM6Ly91YWYuZXhhbXBsZS5jb20vaW5kZXguaHRtbCIsImFwcElEIjoiaHR0cH
    M6Ly91YWYuZXhhbXBsZS5jb20vZmFjZXRzLmpzb24iLCJjaGFsbGVuZ2UiOiJZYjM5U2RVaFUyQjAwODlwUzVMN1ZCVzhhZ
    mRscGxudlI0QjFBbmE1dms0IiwiY2hhbm5lbEJpbmRpbmciOnt9fQ",

    "assertions": [{
        "assertionScheme": "UAFV1TLV",
        "assertion": "AT73AgM-sQALLgkARkZGRiNGQzAzDi4HAAEAAQIAAAEKLiAAbkZZjz4ysihP9vVgevgoH8SEV2JITk
        TxKFfsKbAiofQJLiAA2onnfjAyZ0Uc3GL4VyOEdRgIkz7qogqzmITcEPLovP0NLggAAAAAAAEAAAAMLkEABNfRNiA1Hp
        QSfrvD_9Qug55Vw2oaKmjgbC8TdiFXGZ6hjP7jYHV0GtYqO0EvrRRvsNBbnyhXUpq6P_iNg9laDGsHPj4CBi5GADBEAi
        C57WZpOHWCTil_IuAYSEfuj3zgyY6KFp_rgNw5kO5OwwIgiZbTG6ZmY3T6ZqvdeOxcA6FBgn6YLCncK-Wyk0XVY8kFLv
        ABMIIB7DCCAZKgAwIBAgIBBDAKBggqhkjOPQQDAjBwMQswCQYDVQQGEwJOWjEjMCEGA1UEAwwaRklETyBDb25mb3JtYW
        NlIFRlc3QgVG9vbHMxFjAUBgNVBAoMDUZJRE8gQWxsaWFuY2UxJDAiBgNVBAsMG0NlcnRpZmljYXRpb24gV29ya2luZy
        BHcm91cDAeFw0xNzAyMjkxNDMxMTJaFw0yMjAyMjgxNDMxMTJaMHAxCzAJBgNVBAYTAk5aMSMwIQYDVQQDDBpGSURPIE
        NvbmZvcm1hY2UgVGVzdCBUb29sczEWMBQGA1UECgwNRklETyBBbGxpYW5jZTEkMCIGA1UECwwbQ2VydGlmaWNhdGlvbi
        BXb3JraW5nIEdyb3VwMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEZaRKB92Abz8nqEZFf8Xz84ajfA7lLjt4O-i2wq
        lFnD_svIyTyEYm_QbOYJC0GUVE-L6V7OiD8K9Z4PfiBFRO-qMdMBswDAYDVR0TBAUwAwEB_zALBgNVHQ8EBAMCBsAwCg
        YIKoZIzj0EAwIDSAAwRQIgWDy1Oxu8PT6diGXycY0rxb1e16omexfQ-Iv9KOg5p9cCIQCFPPCArmDh3-EyxI_OaZFPvW
        2kG2hQBmi9PnC-bBrfYQ"
    }]
}]
```

NOTE

Line breaks in fcParams have been inserted for improving readability.

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 `ClientData` 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.disallowed`.

            3. If needed - create MatchCriteria`m` for other disallowed criteria (e.g. unsupported authenticationAlgs)
            4. Add all `m` to `p.disallowed`.
2. Create a `RegistrationRequest` object `r` with appropriate `r.header` for each 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. Assign the username of the user to be registered to `r.username`
    4. Assign `p` to `r.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 `RegRequest.policy.disallowed[].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.appID`, `fcp.challenge`, `fcp.facetID`, and `fcp.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:

ASM Status Code	ErrorCode	Comment
UAF_ASM_STATUS_OK	NO_ERROR	Pass-through success status.
UAF_ASM_STATUS_ERROR	UNKNOWN	Map to UNKNOWN.
UAF_ASM_STATUS_ACCESS_DENIED	AUTHENTICATOR_ACCESS_DENIED	Map to AUTHENTICATOR_ACCESS_DENIED
UAF_ASM_STATUS_USER_CANCELLED	USER_CANCELLED	Pass-through status code.
UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT	INVALID_TRANSACTION_CONTENT	Map to INVALID_TRANSACTION_CONTENT. This code indicates a problem to be resolved by the entity providing the transaction text.
UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY	KEY_DISAPPEARED_PERMANENTLY	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.
UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED	NO_SUITABLE_AUTHENTICATOR or WAIT_USER_ACTION	Retry operation with other suitable authenticators and map to NO_SUITABLE_AUTHENTICATOR if the problem persists. Return WAIT_USER_ACTION if being called while retrying.
UAF_ASM_STATUS_USER_NOT_RESPONSIVE	USER_NOT_RESPONSIVE	Pass-through status code. The RP App might want to retry the operation once the user pays attention to the application again.
UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES	INSUFFICIENT_AUTHENTICATOR_RESOURCES	The FIDO Client shall try other authenticators matching the policy. If none exist, pass-through status code.
UAF_ASM_STATUS_USER_LOCKOUT	USER_LOCKOUT	Pass-through status code.
UAF_ASM_STATUS_USER_NOT_ENROLLED	USER_NOT_ENROLLED	Pass-through status code.
Any other status code	UNKNOWN	Map any unknown error code to UNKNOWN. This might happen when a FIDO Client communicates with an ASM implementing a newer UAF specification than the FIDO Client.

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 **`ClientData`** 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.assertion` 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. Retrieve the AAID from the assertion.

    > NOTE
    >
    > The AAID in `TAG_UAFV1_KRD` is contained in `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_AAID`.

    3. Verify that `a.assertionScheme` matches `Metadata(AAID).assertionScheme`
        - If it doesn't match - continue with next assertion
    4. 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
    5. Locate authenticator-specific authentication algorithms from the authenticator metadata [FIDOMetadataStatement] using the AAID.
    6. 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)`
    7. If `fcp` is of type **`ClientData`**, then hash `RegistrationResponse.fcParams` using hashing algorithm specified in `fcp.hashAlg`.
        - `FCHash = hash(RegistrationResponse.fcParams)`
    8. if `a.assertion` contains an object of type `TAG_UAFV1_REG_ASSERTION`, then
        1. 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 `TAG_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.TAG_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.TAG_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 `TAG_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.TAG_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 `TAG_ATTESTATION_ECDAA`
   1. If entry `ecdaaTrustAnchors` for the AAID 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. If `Metadata(AAID).ecdaaTrustAnchors` for this AAID is empty - continue with next assertion
   3. 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).

2. 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.
3. 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.

9. if `a.assertion` doesn't contain an object of type `TAG_UAFV1_REG_ASSERTION`, then skip this assertion (as in this UAF v1 only `TAG_UAFV1_REG_ASSERTION` is defined).

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

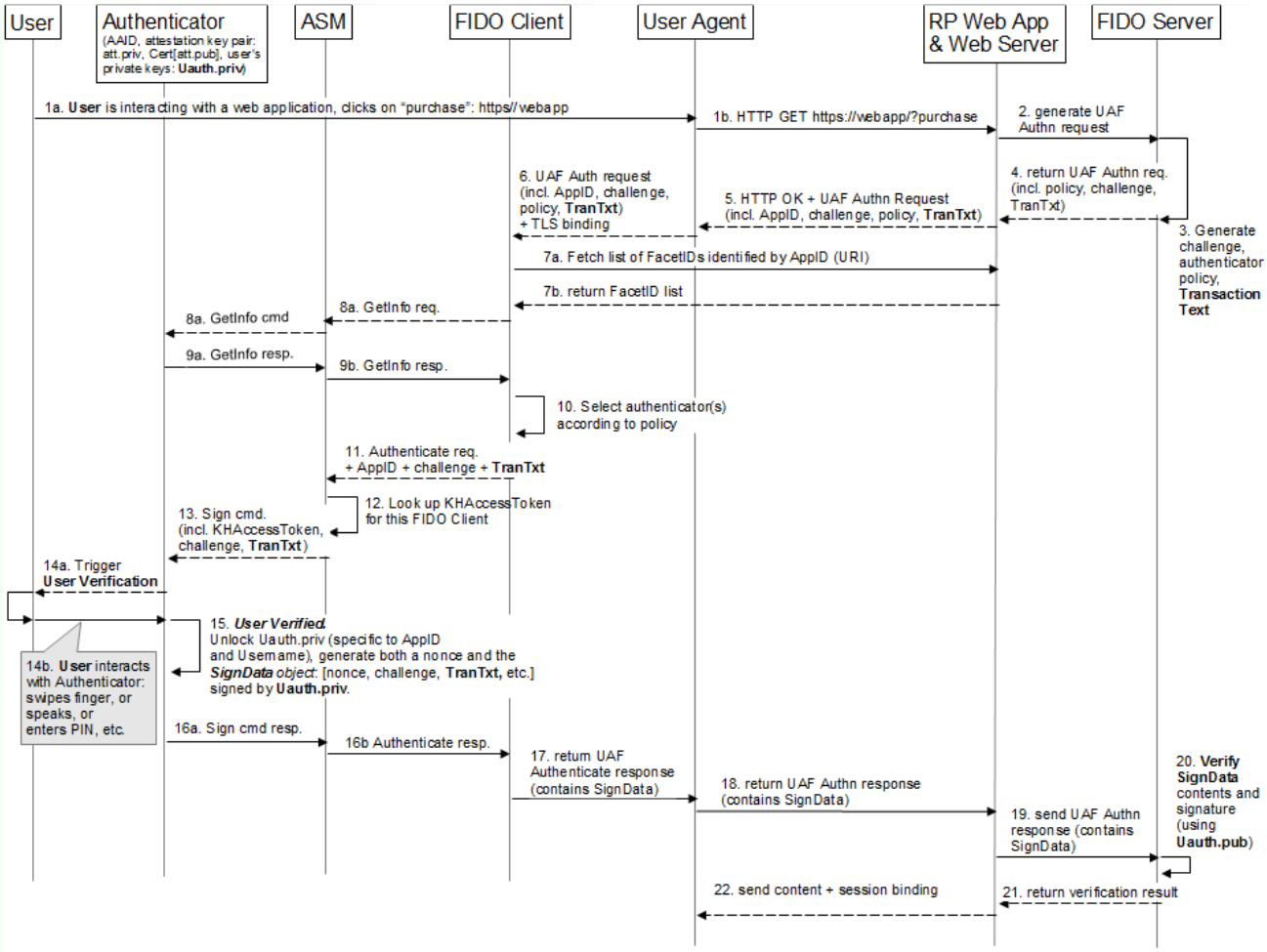

Fig. 8 UAF Authentication Sequence Diagram

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.

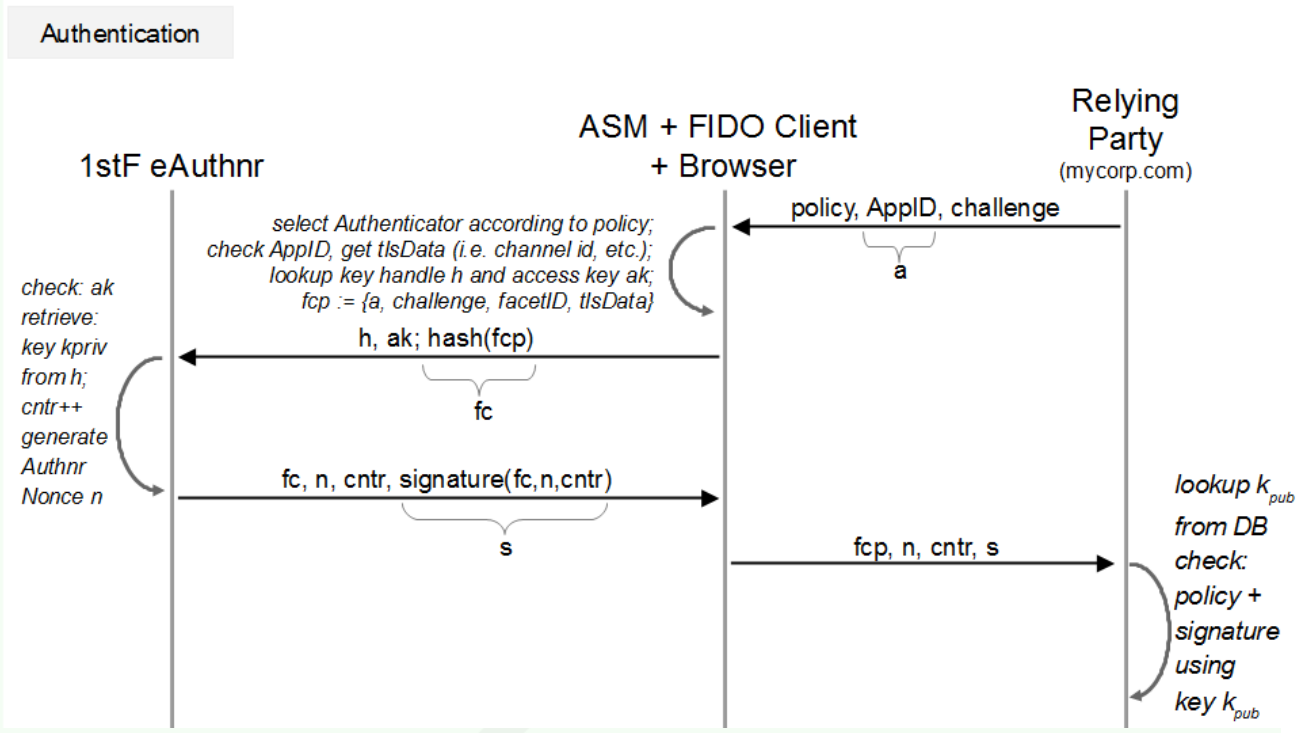

Fig. 9 UAF Authentication Cryptographic Data Flow

Diagram of cryptographic flow:

The FIDO Server sends the `AppID` (see [FIDOAppIDAndFacets]), the authenticator policy and the `ServerChallenge` to the FIDO UAF

Client.

The FIDO UAF Client computes the hash of the `FinalChallengeParams`, produced from the `ServerChallenge` and other values, as described in this document, and sends the `AppID` and hashed `FinalChallengeParams` to the Authenticator.

The authenticator creates the `SignedData` object (see `TAG_UAFV1_SIGNED_DATA` in [UAFAuthnrCommands]) containing the hash of the final challenge parameters, and some other values and signs it using the `UAuth.priv` key. This assertion is then cryptographically verified by the FIDO Server.

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 its metadata statement (see [FIDOMetadataStatement]).

  This version of the specification only supports the values `text/plain` or `image/png`.

`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" then it must be base64url encoded PNG [PNG] image

`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

```
[{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Auth",
        "appID": "https://uaf.example.com/facets.json",
        "serverData": "mz0YSKHLXDd_StbbDINZaRvW3Pa6sxrNMPYp2gOs3-Y"
    },
    "challenge": "4D8eUxdSzQ_Rbk7Gf0SooK7Xr9O2LU-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": [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"]
            }]
        ]
    }
}]
```

EXAMPLE 11: UAF Authentication Request with text/plain Transaction

```
[{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Auth",
        "appID": "https://uaf.example.com/facets.json",
        "serverData": "DLbLt14MdqvuS4fESNCAPJmS8yIKPJ3Ad0xb1cMyu2Q"
    },
    "challenge": "vui9bgJ453N_kWlZbiwMz9q6uPvssjnXjkHYzk-LurY",
    "transaction": [
        {
            "contentType": "text/plain",
            "content": "VHJhbnNmZXIgMjAwMCQgdG8gRXZl"
        }
    ],
    "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": [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 `ClientData` 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

```
[{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Auth",
        "appID": "https://uaf.example.com/facets.json",
        "serverData": "mz0YSKHLXDd_StbbDINZaRvW3Pa6sxrNMPYp2gOs3-Y"
    },

    "fcParams": "eyJmYWNldElEIjoiaHR0cHM6Ly91YWYuZXhhbXBsZS5jb20vaW5kZXguaHRtbCIsImFwcElEIjoiaHR0cH
    M6Ly91YWYuZXhhbXBsZS5jb20vZmFjZXRzLmpzb24iLCJjaGFsbGVuZ2UiOiI0RDhlVXhkU3pRX1JiazdHZjBTb29LN1hyO
    U8yTFUtZzE1MHN0T3BLMGdvIiwiY2hhbm5lbEJpbmRpbmciOnt9fQ",

    "assertions": [{
        "assertionScheme": "UAFV1TLV",
        "assertion": "Aj7EAAQ-dgALLgkARkZGRiNGQzAzDi4FAAEAAQIADy4IAB4gsCir67EvCi4gAMYR1ZSqYuPLiNpYl
        omDJYGZZGQRGSlLlThqf8ZzF-k2EC4AAAkuIADaied-MDJnRRzcYvhXI4R1GAiTPuqiCrOYhNwQ8ui8_Q0uBAABAAAA
        Bi5GADBEAiDDt4-pzmEWZyakWcWGdtBQLIXSf75wL3tEjiCIry_QtQIgjw0oMlQqKOHdG2M26e1Z0bG4wGjfow_vu5z
        p-VkALFo"
    }]
}]
```

EXAMPLE 13: UAF Authentication Response for text/plain Transaction

```
[{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Auth",
        "appID": "https://uaf.example.com/facets.json",
        "serverData": "mz0YSKHLXDd_StbbDINZaRvW3Pa6sxrNMPYp2gOs3-Y"
    },

    "fcParams": "eyJmYWNldElEIjoiaHR0cHM6Ly91YWYuZXhhbXBsZS5jb20vaW5kZXguaHRtbCIsImFwcElEIjoiaHR0cH
    M6Ly91YWYuZXhhbXBsZS5jb20vZmFjZXRzLmpzb24iLCJjaGFsbGVuZ2UiOiI0RDhlVXhkU3pRX1JiazdHZjBTb29LN1hyO
    U8yTFUtZzE1MHN0T3BLMGdvIiwiY2hhbm5lbEJpbmRpbmciOnt9fQ",

    "assertions": [{
        "assertionScheme": "UAFV1TLV",
        "assertion": "Aj7EAAQ-dgALLgkARkZGRiNGQzAzDi4FAAEAAQIADy4IAB4gsCir67EvCi4gAMYR1ZSqYuPLiNpYl
        omDJYGZZGQRGSlLlThqf8ZzF-k2EC4AAAkuIADaied-MDJnRRzcYvhXI4R1GAiTPuqiCrOYhNwQ8ui8_Q0uBAABAAAA
        Bi5GADBEAiDDt4-pzmEWZyakWcWGdtBQLIXSf75wL3tEjiCIry_QtQIgjw0oMlQqKOHdG2M26e1Z0bG4wGjfow_vu5z
        p-VkALFo"
    }]
}]
```

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 `p`
    1. for each set of alternative authenticators do
        1. Create an 1-dimensional array of MatchCriteria objects v containing the set of authenticators to be used for authentication simultaneously that need to be identified by *separate* MatchCriteria objects `m`.
            1. For each collection of authenticators `a` to be used for authentication 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
                - 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
7. Create a FinalChallengeParams structure `fcp` and set `fcp.AppID`, `fcp.challenge`, `fcp.facetID`, and `fcp.channelBinding` appropriately. Serialize [RFC4627] `fcp` using UTF8 encoding and base64url encode it.
   - `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)

        > 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 the above checks fails
5. If this `fcp` object is a **`ClientData`** object, then verify each field in `fcp` and make sure it's 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 the above checks fails
6. For each assertion `a` in `AuthenticationResponse.assertions`
    1. Parse data from `a.assertion` 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. Retrieve the AAID from the assertion.

        > NOTE
        >
        > The AAID in `TAG_UAFV1_SIGNED_DATA` is contained in `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_AAID`.

    3. Verify that `a.assertionScheme` matches `Metadata(AAID).assertionScheme`
        - If it doesn't match - continue with next assertion
    4. Make sure that the AAID indeed matches the policy of the Authentication Request
        - If it doesn't meet the policy – continue with next assertion
    5. if `a.assertion` contains an object of type `TAG_UAFV1_AUTH_ASSERTION`, then
        1. if `a.assertion.TAG_UAFV1_AUTH_ASSERTION` contains `TAG_UAFV1_SIGNED_DATA` as first element:
            1. Obtain `Metadata(AAID).AuthenticatorVersion` for this AAID and make sure that it is lower or equal to `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_ASSERTION_INFO.AuthenticatorVersion`.
                - If `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 [FIDOMetadataService] for more details on this.
            2. Retrieve `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_KEYID` as KeyID
            3. Locate `UAuth.pub` public key associated with (AAID, KeyID) in the user's record.
                - If such record doesn't exist - continue with next assertion
            4. Verify the AAID against the AAID stored in the user's record at time of Registration.
                - If comparison fails – continue with next assertion
            5. Locate authenticator specific authentication algorithms from authenticator metadata (field `AuthenticationAlgs`)
            6. Check the Signature Counter `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).
            7. 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)`

8. If `fcp` is of type **`ClientData`**, then hash `AuthenticationResponse.fcParams` using hashing algorithm specified in `fcp.hashAlg`.
    - `FCHash = hash(AuthenticationResponse.fcParams)`
9. Make sure that `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_FINAL_CHALLENGE_HASH` == `FCHash`
    - If comparison fails – continue with next assertion
10. 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
11. Use `UAuth.pub` key and appropriate authentication algorithm to verify `a.assertion.TAG_UAFV1_AUTH_ASSERTION.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.

6. if `a.assertion` doesn't contain an object of type `TAG_UAFV1_AUTH_ASSERTION`, then skip this assertion (as in this UAF v1 only `TAG_UAFV1_AUTH_ASSERTION` is defined).
7. Treat this assertion `a` as positively verified.

7. Process all positively verified authentication assertions `a`.

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

```
[{
    "header": {
        "upv": {
            "major": 1,
            "minor": 2
        },
        "op": "Dereg",
        "appID": "https://uaf.example.com/facets.json"
    },
    "authenticators": [
        {
            "keyID": "kbufhLYGoFFLJPRCUvwiUu-fr1nh3sX3IjM9i9lcOrQ",
            "aaid": "FFFF#FC03"
        }
    ]
}]
```

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 (`AAID`, `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. For each authenticator compatible with the message version `DeregistrationRequest.header.upv` and having an AAID matching one of the provided `AAID`s (an AAID of an authenticator matches if it is either (a) equal to one of the `AAID`s 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 3.4.6.2.1 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 shouldt be done according to [FIDOMetadataService].

NOTE

The protection measures of the Authenticator's attestation private key depend on the specific authenticator model's implementation.

NOTE

The FIDO Server must load the appropriate Authenticator Attestation Root Certificate from its trust store based on the AAID provided in KeyRegistrationData object.

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.

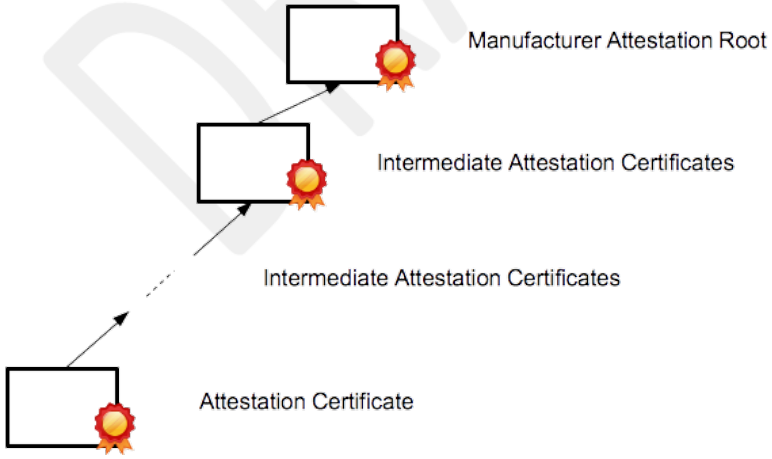

Fig. 10 Attestation Certificate Chain

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].

NORMATIVE

The ECDAA attestation algorithm is used as specified in [FIDOEcdaaAlgorithm].

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 (seeFIDO 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 statelesss 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 [[SP 800-131A]] [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

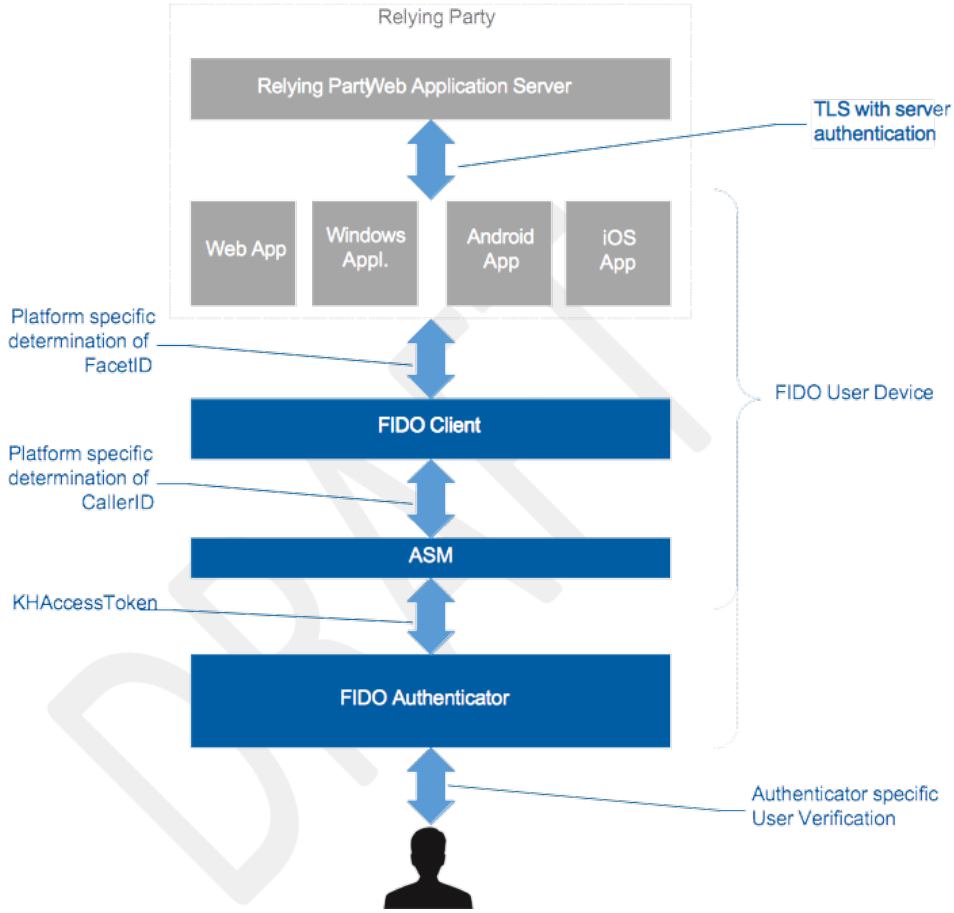

Fig. 11 UAF Client Trust Model

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, PersonaID, ASMToken and the CallerID. See [UAFASM] for more details.

> 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.

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.

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 adding 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.

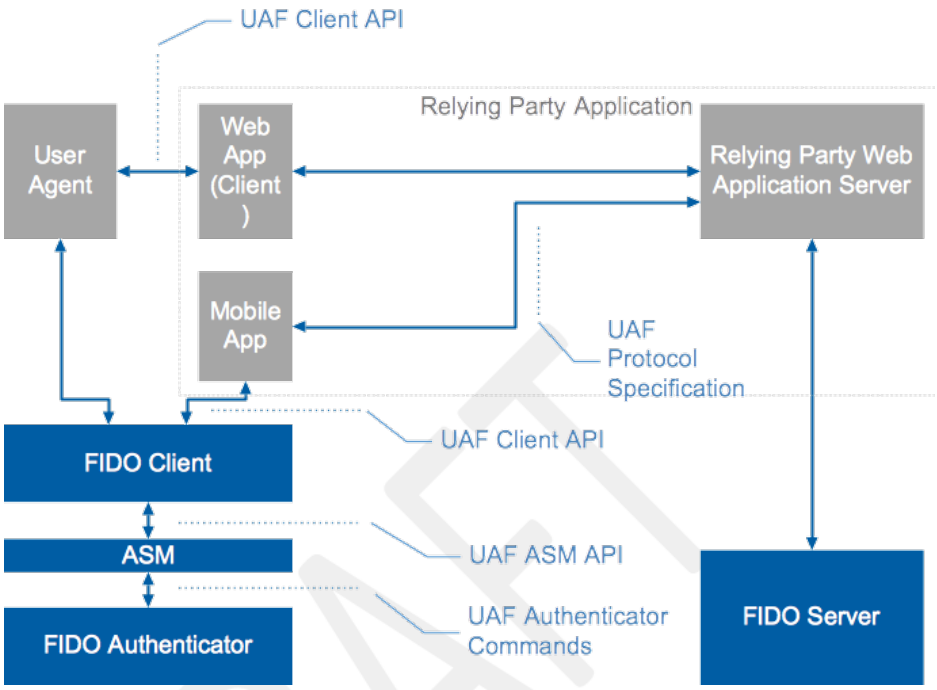

Fig. 12 FIDO Interoperability Overview

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

A. References

A.1 Normative references

[ABNF]
D. Crocker, Ed.; P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234

[ChannelID]
D. Balfanz. *Transport Layer Security (TLS) Channel IDs*. Work In Progress. URL:http://tools.ietf.org/html/draft-balfanz-tls-channelid

[Coron99]
J. Coron; D. Naccache. *An accurate evaluation of Maurer's universal test*. February 1999. URL: http://www.jscoron.fr/publications/universal.pdf

[FIDOAppIDAndFacets]
D. Balfanz; B. Hill; R. Lindemann; D. Baghdasaryan. *FIDO AppID and Facets v1.0* Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-appid-and-facets-v1.2-rd-20171128.html

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian.*FIDO ECDAA Algorithm*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-ecdaa-algorithm-v1.2-rd-20171128.html

[FIDOGlossary]

R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-glossary-v1.2-rd-20171128.html

[FIDOMetadataStatement]
B. Hill; D. Baghdasaryan; J. Kemp. *FIDO Metadata Statements v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-metadata-statement-v1.2-rd-20171128.html

[FIDORegistry]
R. Lindemann; D. Baghdasaryan; B. Hill. *FIDO Registry of Predefined Values*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-registry-v1.2-rd-20171128.html

[FIPS180-4]
FIPS PUB 180-4: Secure Hash Standard (SHS). March 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[JWA]
M. Jones. *JSON Web Algorithms (JWA)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7518

[JWK]
M. Jones. *JSON Web Key (JWK)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7517

[PNG]
Tom Lane. *Portable Network Graphics (PNG) Specification (Second Edition)*. 10 November 2003. W3C Recommendation. URL: https://www.w3.org/TR/PNG/

[RFC1321]
R. Rivest. *The MD5 Message-Digest Algorithm (RFC 1321)*. April 1992. URL: http://www.ietf.org/rfc/rfc1321.txt

[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

[RFC3629]
F. Yergeau. *UTF-8, a transformation format of ISO 10646* November 2003. Internet Standard. URL: https://tools.ietf.org/html/rfc3629

[RFC4086]
D. Eastlake 3rd; J. Schiller; S. Crocker. *Randomness Requirements for Security (RFC 4086)*. June 2005. URL: http://www.ietf.org/rfc/rfc4086.txt

[RFC4627]
D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. July 2006. Informational. URL: https://tools.ietf.org/html/rfc4627

[RFC4648]
S. Josefsson. *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*. October 2006. URL: http://www.ietf.org/rfc/rfc4648.txt

[RFC5056]
N. Williams. *On the Use of Channel Bindings to Secure Channels (RFC 5056)* November 2007. URL: http://www.ietf.org/rfc/rfc5056.txt

[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: http://www.ietf.org/rfc/rfc5280.txt

[RFC5929]
J. Altman; N. Williams; L. Zhu. *Channel Bindings for TLS (RFC 5929)*. July 2010. URL: http://www.ietf.org/rfc/rfc5929.txt

[RFC6234]
D. Eastlake 3rd; T. Hansen. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF) (RFC 6234)* May 2011. URL: http://www.ietf.org/rfc/rfc6234.txt

[RFC6979]
T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) (RFC6979)*. August 2013. URL: http://www.ietf.org/rfc/rfc6979.txt

[SP800-90b]
Elaine Barker; John Kelsey. *NIST Special Publication 800-90b: Recommendation for the Entropy Sources Used for Random Bit Generation*. April 2016. URL: http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf

[UAFASM]
D. Baghdasaryan; J. Kemp; R. Lindemann; B. Hill; R. Sasson. *FIDO UAF Authenticator-Specific Module API*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-asm-api-v1.2-rd-20171128.html

[UAFAppAPIAndTransport]
B. Hill; D. Baghdasaryan; B. Blanke. *FIDO UAF Application API and Transport Binding Specification* Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-client-api-transport-v1.2-rd-20171128.html

[UAFAuthnrCommands]
D. Baghdasaryan; J. Kemp; R. Lindemann; R. Sasson; B. Hill. *FIDO UAF Authenticator Commands v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-authnr-cmds-v1.2-rd-20171128.html

[UAFRegistry]
R. Lindemann; D. Baghdasaryan; B. Hill. *FIDO UAF Registry of Predefined Values* Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-reg-v1.2-rd-20171128.html

[WebAuthn]
Vijay Bharadwaj; Hubert Le Van Gong; Dirk Balfanz; Alexis Czeskis; Arnar Birgisson; Jeff Hodges; Michael B. Jones; Rolf Lindemann; J. C. Jones. *Web Authentication: An API for accessing Scoped Credentials*. September 2016. Draft. URL: https://www.w3.org/TR/webauthn/

[WebIDL-ED]
Cameron McCormack. *Web IDL*. 13 November 2014. Editor's Draft. URL: http://heycam.github.io/webidl/'

A.2 Informative references

[AnonTerminology]
A. Pfitzmann; M. Hansen. *Anonymity, Unlinkability, Unobservability, Pseudonymity, and Identity Management - A Consolidated Proposal for Terminology, Version 0.34*. August 2010. URL: http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf

[BriCamChe2004-DAA]
Ernie Brickell; Jan Camenisch; Liqun Chen. *Direct Anonymous Attestation*. 2004. URL: http://eprint.iacr.org/2004/205.pdf

[CTRMode]
H. Lipmea; P. Rogaway; D. Wagner. *Comments to NIST concerning AES Modes of Operation: CTR-Mode Encryption* URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf

[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

[ECDSA-ANSI]
Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62-2005 November 2005. URL: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005

[FIDOMetadataService]
R. Lindemann; B. Hill; D. Baghdasaryan. *FIDO Metadata Service v1.0*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-metadata-service-v1.2-rd-20171128.html

[FIDOSecRef]
R. Lindemann; D. Baghdasaryan; B. Hill. *FIDO Security Reference*. Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-security-ref-v1.2-rd-20171128.html

[FIPS198-1]
FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC). July 2008. URL: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf

[ISOBiometrics]
ISO/IEC 2382-37 Harmonized Biometric Vocabulary. 15 December 2012. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194_ISOIEC_2382-37_2012.zip

[OWASP2013]
. 2013. OWASP Top 10 - 2013. The Ten Most Critical Web Application Security Risks. URL: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf

[RFC2560]
M. Myers; R. Ankney; A. Malpani; S. Galperin; C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. June 1999. Proposed Standard. URL:https://tools.ietf.org/html/rfc2560

[RFC5746]
E. Rescorla; M. Ray; S. Dispensa; N. Oskov. *Transport Layer Security (TLS) Renegotiation Indication Extension* February 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5746

[RFC6125]
P. Saint-Andre; J. Hodges. *Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS) (RFC 6125)*. March 2011. URL: http://www.ietf.org/rfc/rfc6125.txt

[RFC6287]
D. M'Raihi; J. Rydell; S. Bajaj; S. Machani; D. Naccache. *OCRA: OATH Challenge-Response Algorithm (RFC 6287)*. June 2011. URL: http://www.ietf.org/rfc/rfc6287.txt

[RFC6454]
A. Barth. *The Web Origin Concept (RFC 6454)*. June 2011. URL: http://www.ietf.org/rfc/rfc6454.txt

[RFC7525]
Y. Sheffer; R. Holz; P. Saint-Andre. *Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. May 2015. Best Current Practice. URL:https://tools.ietf.org/html/rfc7525

[RFC7627]
K. Bhargavan, Ed.; A. Delignat-Lavaud; A. Pironti; A. Langley; M. Ray. *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension*. September 2015. Proposed Standard. URL:https://tools.ietf.org/html/rfc7627

[SHEFFER-TLS]
Y. Sheffer; R. Holz; P. Saint-Andre. *Recommendations for Secure Use of TLS and DTLS* Internet-Draft (Work in Progress). URL: https://tools.ietf.org/html/draft-sheffer-tls-bcp

[SP800-38C]
M. Dworkin. *NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. July 2007. URL: http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf

[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: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf

[SP800-63]
W. Burr; D. Dodson; E. Newton; R. Perlner; W.T. Polk; S. Gupta; E. Nabbus. *NIST Special Publication 800-63-2: Electronic Authentication Guideline*. August 2013. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf

[TLS]
T. Dierks; E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2* August 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5246

[TLSAUTH]
Karthikeyan Bhargavan; Antoine Delignat-Lavaud; Cédric Fournet; Alfredo Pironti; Pierre-Yves Strub. *Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS*. February 2014. URL: https://secure-resumption.com/tlsauth.pdf

[TPMv1-2-Part1]
TPM 1.2 Part 1: Design Principles URL: http://www.trustedcomputinggroup.org/files/static_page_files/72C26AB5-1A4B-B294-D002BC0B8C062FF6/TPM%20Main-Part%201%20Design%20Principles_v1.2_rev116_01032011.pdf

[TPMv2-Part1]
Trusted Platform Module Library, Part 1: Architecture URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C56AE3E-1A4B-B294-D0F43097156A55D8/TPM%20Rev%202.0%20Part%201%20-%20Architecture%2001.16.pdf

[TR-03116-4]
Technische Richtlinie TR-03116-4: eCard-Projekte der Bundesregierung: Teil 4 – Vorgaben für Kommunikationsverfahren im eGovernment. 2013. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03116/BSI-TR-03116-4.pdf

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/