UAF Protocol Specification

Specification Set: fido-uaf-v1.0-rd-20140209  REVIEW DRAFT

Editors:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.
Eric Tiffany, FIDO Alliance

Contributors:
Dirk Balfanz, Google

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 for the Relying Party 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:

This Specification has been prepared by FIDO Alliance, Inc. **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.**

Copyright © 2014 FIDO Alliance, Inc. All rights reserved.
# Table of Contents

1 Notation

1.1 Key Words

2 Overview

2.1 Scope

2.2 Architecture

2.3 Protocol Conversation

2.3.1 Registration

2.3.2 Authentication

2.3.3 Transaction Confirmation

2.3.4 Deregistration

3 Protocol Details

3.1 Shared Structures and Types

3.1.1 Version

3.1.2 Operation Header

3.1.3 Type of Authenticator Attestation ID (AAID)

3.1.4 Type of KeyID

3.1.5 Type of ServerChallenge

3.1.6 Type of FinalChallengeParams

3.1.7 Type of TLSData

3.1.8 Type of JwkKey

3.1.9 Type of Extension

3.1.10 Type of TrustedApps

3.1.11 Type of Policy

3.2 Version Negotiation
3.3 Registration Operation

3.3.1 Type of RegisterRequest

3.3.2 Type of RegisterResponse

3.3.3 Processing Rules

3.3.3.1 Registration Request Generation Rules for FIDO Server

3.3.3.2 Registration Request Processing Rules for FIDO Client

3.3.3.3 Registration Request Processing Rules for FIDO Authenticator

3.3.3.4 Registration Response Generation Rules for FIDO Client

3.3.3.5 Registration Response Processing Rules for FIDO Server

3.4 Authentication Operation

3.4.1 Type of AuthenticationRequest

3.4.2 Type of AuthenticationResponse

3.4.3 Processing Rules

3.4.3.1 Authentication Request Generation Rules for FIDO Server

3.4.3.2 Authentication Request Processing Rules for FIDO Client

3.4.3.3 Authentication Request Processing Rules for FIDO Authenticator

3.4.3.4 Authentication Response Generation Rules for FIDO Client

3.4.3.5 Authentication Response Processing Rules for FIDO Server

3.5 Deregistration Operation

3.5.1 Type of DeregistrationRequest

3.5.2 Processing Rules

3.5.2.1 Deregistration Request Generation Rules for FIDO Server

3.5.2.2 Deregistration Request Processing Rules for FIDO Client

3.5.2.3 Deregistration Request Processing Rules for FIDO Authenticator

4 Considerations

4.1 Protocol Core Design Considerations
4.1.1 Authenticator Metadata ......................................................... 43
4.1.2 Authenticator Attestation .................................................... 43
  4.1.2.1 Basic Attestation .......................................................... 44
4.1.3 Error Handling ................................................................. 45
4.1.4 Assertion Schemes ............................................................ 46
4.1.5 Username in Authenticator ............................................... 46
4.1.6 TLS Protected Communication ......................................... 46
4.2 Implementation Considerations ............................................ 47
  4.2.1 Server Challenge and Random Numbers ................................ 47
4.3 Security Considerations ........................................................ 47
  4.3.1 FIDO Authenticator Security ............................................ 50
  4.3.2 Cryptographic Algorithms ............................................... 50
  4.3.3 Application Isolation ....................................................... 50
    4.3.3.1 AppID and FacetID Assertion .................................... 51
    4.3.3.2 Isolation using KHAccessToken .................................. 53
  4.3.4 TLS Binding .................................................................. 54
  4.3.5 Personas ......................................................................... 54
  4.3.6 ServerData and KeyHandle ............................................. 55
  4.3.7 Authenticator Information retrieved through UAF Client API vs. Metadata ........................................ 56
  4.3.8 Policy Verification .......................................................... 56
  4.3.9 Replay Attack Protection ................................................ 56
  4.3.10 Protection against Cloned Authenticators .................................. 57
4.4 Interoperability Considerations ............................................ 57
5 UAF Supported Assertion Schemes .......................................... 60
  5.1 Assertion Scheme “UAFV1TLV” ........................................... 60
    5.1.1 KeyRegistrationData .................................................. 60
1 Notation

Type names, attribute names and element names are written in *italics*. String literals are enclosed in “”, e.g. “UAF-TLV”. In formulas we use “|” to denote byte wise concatenation operations. UAF specific terminology used in this document is defined in [FIDOGlossary].

1.1 Key Words

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

The goal of this Universal Authentication Framework is to provide a unified and extensible authentication mechanism that supplants passwords while avoiding the shortcomings of current alternative authentication approaches. The design goal of the protocol is to enable Relying Parties to leverage the diverse and heterogeneous set of security capabilities available on end users’ devices via a single, unified protocol. This approach is designed to allow the Relying Party to choose the best available authentication mechanism for a particular end user or interaction, while preserving the option for the 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 Client?
- How can FIDO Client communicate to FIDO enabled Authenticators?

The answers to these questions can be found other UAF specifications, e.g. UAFAppAPI&Binding] [UAFASM] [UAFAuthnrCommands].

2.2 Architecture

The following diagram depicts the entities involved in UAF protocol.
Of these entities, only these three directly create and/or process UAF protocol messages:

- **FIDO Server**, running on the Relying Party’s infrastructure
- **FIDO 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 FIDO Authenticator Metadata [UAFAuthnMetadata] describing all the Authenticators it will interact with.

### 2.3 Protocol Conversation

The core UAF protocol consists of four conceptual conversations between FIDO 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. FIDO Client will only register existing FIDO 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 provides support for prompting the user to confirm a specific transaction. This prompt includes the ability to communicate additional information to the client for secure display to the end user. 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.
Deregistration: The Relying Party can trigger the deletion of the 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 User Agent’s request (e.g., an HTTP request) to Relying Party.

The following section gives a brief overview of the protocol conversation for individual operations. More detailed descriptions can be found in the sections Registration Operation, Authentication Operation, Authentication Operation, and Deregistration Operation.

2.3.1 Registration

The following diagram shows the message flows for the Registration operation.

2.3.2 Authentication

The following diagram depicts the message flows for the Authentication operation.
### 2.3.3 Transaction Confirmation

The following figure depicts the Transaction Confirmation message flow.
### 2.3.4 Deregistration

The following diagram depicts the Deregistration message flow.

![Diagram of Deregistration process](image)
3 Protocol Details

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.

- All data used in this protocol MUST be exchanged using a secure protocol (such as TLS/HTTPS) established between FIDO Client and Relying Party; details are specified in section TLS Protected Communication.

- Unless otherwise specified the fields in UAF messages MUST be non-empty and if a list/array is provided it MUST have at least one entry.

- 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].

- The notation string[5] reads as a five-character UTF-8 formatted string of the type indicated in the declaration, typically a WebIDL [WebIDL] DOMString.

- All strings are case-sensitive unless noted otherwise.

Unless explicitly specified the “MUST” keyword applies to all steps described in this document

This document uses WebIDL to define UAF protocol messages. Implementations MUST serialize the UAF protocol messages for transmission using JSON [RFC4627] using UTF8 encoding.

3.1 Shared Structures and Types

This section defines types and structures shared by various operations.

3.1.1 Version

Represents a generic version with major and minor fields.

dictionary Version {
    int mj;   // Mandatory.
    int mn;   // Mandatory.
}
FIDO UAF Protocol Specification

Description:
- \texttt{mj}: Major version
- \texttt{mn}: Minor version

3.1.2 Operation Header

Represents a UAF Message Request and Response header

\begin{verbatim}
dictionary OperationHeader {
  Version upv;  // Mandatory.
  DOMString op;  // Mandatory. Must be "Reg", "Auth" or "Dereg"
  DOMString appID;  // Mandatory. string[1..512].
  DOMString serverData;  // Optional, string[1..1536]
  Extension[] exts;  // Optional.
}
\end{verbatim}

Description:
- \texttt{upv}: UAF protocol version. Major version must be 1 and minor version must be 0.
- \texttt{op}: Name of FIDO operation this message relates to. Note that "Auth" is used for both authentication and transaction confirmation.
- \texttt{appID}: The application id that the RP would like to assert. The new key pair that the UAF Authenticator generates will be associated with this appID. It MUST be an URI with HTTPS protocol as FIDO Client will use it to load the list of FacetIDs using this URI. **Security Relevance**: The AppID is used by the FIDO Client to verify the eligibility of an application to trigger use of a specific Uauth key.
- \texttt{serverData}: A session id created by the RP. The RP can opaquely store things like expiration times for the registration session, protocol version used and other useful information there. This data is opaque to FIDO Client and servers MAY reject a response that lacks or contains unauthorized modifications to this data. Servers that depend on it SHOULD apply and verify a cryptographically secure Message Authentication Code (MAC) to \texttt{serverData} and ensure it is cryptographically bound to other relevant portions of the message such as the ServerChallenge, see also section ServerData and KeyHandle.
- \texttt{exts}: List of UAF Message Extensions.

3.1.3 Type of Authenticator Attestation ID (AAID)

\begin{verbatim}
typedef DOMString AAID;  // string[9]
\end{verbatim}

Description:
AAID: Each Authenticator MUST have an AAID to identify UAF enabled Authenticator models globally. Only Authenticators from the same vendor, of the same Model, and with identical security characteristics may share the same AAID (see Security Considerations).

- The AAID is a string with following format – “V#M”, where
  - “#” is a separator
  - “V” indicates the Authenticator Vendor Code. This code consists of 4 hex digits.
  - “M” indicates the Authenticator Model Code. This code consists of 4 hex digits.

- The Augmented BNF [ABNF] for the 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 model codes to their Authenticators. Authenticator Vendors MUST assign unique AAIDs to Authenticators with different security characteristics.
  - Fixing firmware/software bugs, 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.

### 3.1.4 Type of KeyID

typedef DOMString KeyID; // base64url(byte[32...2048])

**Description:**

- **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 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).

- **KeyID** may be used by Roaming Authenticators which don’t have internal storage and need to store the generated Uauth keys in wrapped form (see also section ServerData and KeyHandle) on a FIDO Server.

- During an authentication operation FIDO Server has to provide the KeyID back to the Authenticator for the latter to unwrap the Uauth.priv key and generate a signature using it.
• The exact structure and content of a KeyID is implementation-specific.

### 3.1.5 Type of ServerChallenge

typedef DOMString ServerChallenge; // base64url(byte[8...64])

**Description:**
- **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. [Server Challenge and Random Numbers]
- The minimum challenge length of 8 bytes follows the requirement in [SP 800-63-1] and is equivalent to the 20 decimal digits as required in [RFC6287].
- The maximum length has been defined such that SHA-512 output can be used without truncation.

### 3.1.6 Type of FinalChallengeParams

dictionary FinalChallengeParams {
  DOMString appID; // Mandatory, string[1..512].
  ServerChallenge challenge; // Mandatory.
  DOMString facetID; // Mandatory, string[1..512].
  TLSData tlsData; // Mandatory
}

**Description:**
- The **appID** is taken from the Operation Header (see section Operation Header).
- The **challenge** is taken from Operation Header (see section Operation Header).
- The **facetID** is determined by FIDO Client and depends on the calling application (see section Type of TrustedApps and section AppID and FacetID Assertion for more details). **Security Relevance:** The **facetID** is determined by the FIDO Client and verified against the TrustedApps retrieved using the **appID**.
- The **tlsData** contains the TLS information to be sent by FIDO Client to the FIDO Server.

### 3.1.7 Type of TLSData

dictionary TLSData {
  DOMString serverEndPoint; // Mandatory, not empty. base64url
DOMString tlsServerCertificate; // Optional, not empty if present
DOMString tlsUnique;      // Mandatory, not empty. base64url
DOMString cid_pubkey;     // Optional, base64url encoded JwsKey

Description:

- **TLSData** contains channel binding information [RFC5056]. **Security Relevance:** The TLSData is verified by the FIDO Server in order to detect and prevent MITM attacks.

- The field **serverEndPoint** must be set
  - to the base64url encoding of the hash of the TLS server certificate if this is available. The hash function is to be selected as follows:
    - 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];
    - if the certificate's signatureAlgorithm uses a single hash function and that hash function neither MD5 nor SHA-1, then use the hash function associated with the certificate's signatureAlgorithm;
    - 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 is channel binding type may occur to address this issue if it ever arises).
  - to "None" if the TLS server certificate is not available to the processing entity (e.g., the FIDO Client) or the hash function cannot be determined as described.

- The field **tlsServerCertificate** is optional.
  - This field must be set to the string “None” if the TLS server certificate is not available to the FIDO Client.
  - This field can be absent if (and only if) the data is available to the FIDO Client, but the FIDO Client decides not to make it available.
  - This field must be set to the base64url encoding of the DER encoded TLS server certificate if this data is available to the FIDO Client and the FIDO Client decides to make this data available.

- The **tlsUnique** field must be set to the base64url encoded TLS channel Finished structure or it must be set to “None” if this data is not available [RFC5929].

- The field **cid_pubkey**
  - is absent if the client TLS stack doesn't provide ChannelID [ChannelID] information to the processing entity (e.g., the web browser or client application).
  - must be set to “None” if ChannelID information is supported by the client-side TLS stack but has not been signaled by the TLS server.
Further requirements:

1. If TLS Channel ID data is accessible to the web browser or client application, it SHALL be relayed to and used by FIDO Client.
2. TLS Channel ID SHALL be supported by FIDO Server. However, it can only be used by FIDO Server if the related Web Server supports it.
3. If TLS binding data according to [RFC5929] is accessible to the FIDO Client, it SHALL be used by FIDO Client. Depending on the constraints given by the operating environment, the FIDO Server may or may not evaluate it.

3.1.8 Type of JwkKey

dictionary JwkKey {
    DOMString kty; // Set key type to “EC”.
    DOMString crv; // Set to “P-256”.
    DOMString x; // Mandatory, not empty. base64url(byte[32])
    DOMString y; // Mandatory, not empty. base64url(byte[32])
}

Description:
- JwkKey is a dictionary representing a JSON Web Key encoding of an Elliptic Curve public key [JWK]. This public key is the Channel ID 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
- The field kty 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].
- The field crv denotes the elliptic curve on which this public key is defined. At this time only P-256 is supported by [ChannelID], so it must be set to “P-256”
- The field x contains the base64url-encoding of the x coordinate of the public key (big-endian, 32-byte value).
- The field y contains the base64url-encoding of the y coordinate of the public key (big-endian, 32-byte value).

3.1.9 Type of Extension

dictionary Extension {
    DOMString id; // Mandatory. string[1..32].
    DOMString data; // Mandatory. base64url(byte[1..8192]).
    boolean fail_if_unknown; // Mandatory.
}

Description:
• Generic extensions used in various operations.
• The field id identifies the extension.
• The field data contains arbitrary data with a semantics agreed between Server and Client.
• The field fail_if_unknown indicates whether unknown extensions should be ignored (fail_if_unknown=false) or should lead to an error (fail_if_unknown=true).

3.1.10 Type of TrustedApps

dictionary TrustedApps {
  Version version;  // Mandatory.
  DOMString[] ids;  // Mandatory. Each list element is string[1..512].
}

Description:
• TrustedApps represents a structure holding a list of FacetIDs trusted by the RP (see section AppID and FacetID Assertion). A HTTP GET query to the AppID (which is a URI) MUST return a JSON object with this structure:

  {"alg": "B64S256",
   "ids": ["https://login.acme.com/",
           "android:apk-key-hash:2jmj7l5rSw0yVb/vlWAYkK/YBwk",
           "ios:bundle-id:com.acme.app"
         ]
  }

• The field version. Major must be set to 1 and minor must be set to 0.
• The field ids contains list of FacetIDs (see also section Type of FinalChallengeParams). Each list element is string[1..512].
  • In the Web case, the facetID is the Web Origin [RFC6454] of the web page triggering the FIDO operation, written as a URI with an empty path. Default ports are omitted. E.g. https://login.mycorp.com/
  • In the Android case, the facetID is derived from the sha1 hash of the APK signing certificate [APK-Signing], i.e. it is the URI android:apk-key-hash:<sha1_hash-of-apk-signing-cert>

The sha1 hash can be computed as follows:
In the iOS case, the facetID is the BundleID [BundleID], i.e. it is the URI
ios:bundle-id:<ios-bundle-id-of-app>

1. This list MUST NOT contain more than one Web Origin facetID.
2. The AppID (i.e. the URL to fetch the TrustedApps object) MUST be a HTTPS URL.
3. The TrustedApps object MUST be directly returned to a HTTPS GET request, i.e. not using any form of redirection.
4. The TrustedApps object MUST be returned as object with MIME-Type “vnd.fido.trusted-apps+json”

### 3.1.11 Type of Policy

```plaintext
dictionary MatchCriteria {
  AAID aaid;       // Optional
  KeyID[] keyIDList; // Optional
  unsigned long long authenticationFactor; // Optional, set of bit flags
  unsigned long long keyProtection;      // Optional, set of bit flags
  unsigned long long attachment;         // Optional, set of bit flags
  unsigned long long secureDisplay;      // Optional, set of bit flags
  unsigned long[] supportedAuthAlgs;     // Optional
  DOMString[] supportedSchemes;           // Optional
  Extension[] exts;                      // Optional
}

dictionary Policy {
  MatchCriteria[][] accepted;  // Mandatory
  MatchCriteria[][] disallowed; // Mandatory
}
```

**Description:**

- The dictionary `MatchCriteria` represents the matching criteria to be used in the server policy:
  - The field `aaid` contains the AAID if matching is restricted to a single AAID.
  - The field `keyIDList` contains a list of the matching Authenticator KeyIDs if matching is restricted to a set of KeyID instances. (see [FIDORegistry])
  - The field `authenticationFactor` contains one or more bit flags if matching is restricted by the authentication factor. (see [FIDORegistry])
  - The field `keyProtection` contains one or more bit flags if matching is restricted by the key protection. (see [FIDORegistry])
○ The field attachment contains one or more bit flags if matching is restricted by the attachment type. (see [FIDORegistry])

○ The field secureDisplay contains one or more bit flags if matching is restricted by the type of the secure display. (see [FIDORegistry])

○ The field supportedAuthAlgs is an array containing values of supported authentication algorithm TAG values (see [FIDORegistry], prefix UAF_ALG_SIGN) if matching is restricted by the supported authentication algorithms.

○ The field supportedSchemes contains a list of supported encoding schemes the authenticators use for KeyRegistrationData and SignedData if matching is restricted by the supported schemes. See section UAF Supported Assertion Schemes for details.

○ The field exts contains a list of extensions.

● The dictionary Policy contains a specification of accepted Authenticators and a specification of disallowed Authenticators.

○ The field accepted is a two dimensional array describing the required authenticator characteristics for the server to accept a registration/authentication 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 registration/authentication.

    ▪ Sub-arrays of MatchCriteria in the second index (i.e. the set) indicate that multiple authenticators (i.e. each set element) must be registered/authenticated to be accepted by the server.

The MatchCriteria array represents ordered preferences by the server. Servers SHOULD put their most preferred authenticators first, and FIDO Clients SHOULD respect those preferences, either by presenting authenticator registration/authentication options to the user in the same order, or by offering to register/authenticate only the most preferred authenticator(s).

○ Any authenticator that matches any of MatchCriteria contained in the field disallowed MUST be excluded from eligibility for registration/authentication, regardless of whether it matches any accepted MatchCriteria or not.

FIDO Client MUST follow the following rules while parsing server policy:

● During registration:

    ○ Policy.accepted is a list of combinations. Each combination indicates a list of criteria for authenticators that the server wants the user to register. A typical combination for registration contains a single criteria.
Follow the priority of items in `Policy.accepted[][]`. The lists are ordered with highest priority first.

Choose the combination who’s criteria matches best with currently available authenticators

- Collect information about available authenticators
- Ignore authenticators which match the `Policy.disallowed` criteria
- Match collected information with the matching criteria imposed in the policy

Guide the user to register the authenticators specified in the chosen combination

During authentication and transaction confirmation:

- Note that `Policy.accepted` is a list of combinations. Each combination indicates a criteria which is enough to completely authenticate the current pending operation
- Follow the priority of items in `Policy.accepted[][]`. The lists are ordered with highest priority first.
- Choose the combination who’s criteria matches best with currently available authenticators
  - Collect information about available authenticators
  - Ignore authenticators which meet the `Policy.disallowed` criteria
  - Match collected information with the matching criteria imposed in the policy
- Guide the user to authenticate with the authenticators specified in chosen combination
  - A pending operation will be approved by Server only after all criteria of a single combination are entirely met

**Example 1: Policy allowing either a FPS based or a Face Recognition based Authenticator**

```json
{
  "accepted": [
    {
      "authenticationFactor": 0x02
    },
    {
      "authenticationFactor": 0x10
    }
  ]
}
```

Note that in the simple example the same result could be achieved by simply combining the `authenticationFactor` bitflags.
Example 2: Policy allowing either a FPS based or a Face Recognition based Authenticator (short)

```json
{  "accepted": [  [{ "authenticationFactor": 0x12}] ] }
```

The next example requires two Authenticators to be used:

Example 3: Policy requiring a FPS based and a Face Recognition based Authenticator (generic)

```json
{  "accepted": [  [{ "authenticationFactor": 0x02},  { "authenticationFactor": 0x10 }] ] }
```

Other criteria can be specified in addition to the authenticationFactor:

Example 4: Policy requiring the combination of two bound Authenticators

```json
{  "accepted": [  [{ "authenticationFactor": 0x02, "attachment": 0x01},  { "authenticationFactor": 0x10, "attachment": 0x01}] ] }
```

### 3.2 Version Negotiation

In the UAF protocol we have the UAF protocol version, the version of KeyRegistrationData and SignedData objects (identified by the respective tags, see [FIDORegistry]), and the ASM version, see [UAFASM].

The KeyRegistrationData and SignedData objects have to be parsed and verified by the FIDO Server. This verification is only possible if the FIDO Server understands the encoding and the content of KeyRegistrationData and SignedData. Each UAF protocol version supports a set of KeyRegistrationData and SignedData versions. Similarly each of the ASM Versions supports a set of KeyRegistrationData and SignedData versions.

As a consequence the FIDO Client must select the Authenticators which will generate the appropriate versions of KeyRegistrationData and SignedData.

Version negotiation is based on the following rules:

1. The FIDO Client creates a set of version pairs ASM Version (av) and UAF Protocol Version (upv) as follows:
   
   1. Add all pairs supported by the FIDO Client into it.
   
   2. Intersect this set of pairs 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).
2. Look into Authenticators available locally which are allowed by the Policy defined in the message. Remove any pair from the set which contains an av not supported by the ASM of any available authenticator.

3. Select the Authenticator to be used from the ones supporting an av included in the set. Remove all pairs from the set which contain an av not supported by the selected Authenticator.

4. Look into remaining set and select "highest" version pair.

Notes:

- "Highest" of two pairs is defined as follows: Take the pair where the upv is highest. In all these pairs look into the one with highest av.
- Each one version in the pair (upv, av) consists of Major and Minor version. The comparison of two versions follows SAML proposal, i.e. compare the Major versions and if they are equal compare the Minor versions.
- Each UAF message contains a version field upv. UAF Protocol Version negotiation is always between FIDO Client and FIDO Server.

3.3 Registration Operation

The Registration operation allows the FIDO Server and the FIDO Authenticator to agree on an Authentication Key.
Figure 3.1: Sequence Diagram of UAF Registration

The following diagram depicts the cryptographic data flow for the Registration sequence.
The FIDO Server sends the AppID (see section AppID and FacetID Assertion), the Authenticator Policy (see section Type of Policy), the server generated Challenge (Chl, see section Type of ServerChallenge) and the Username to the FIDO Client.

The FIDO Client computes the Final Challenge Params (FCH) from the Server Challenge and some other values (see section Type of FinalChallengeParams) and sends the AppID, the FCH and the Username to the Authenticator.

The Authenticator creates the KeyRegistrationData object (KRD, see [UAFAuthnrCommands]) containing the FCH, the newly generated user public key (Uauth.pub) and some other values and signs it using the Attestation private key. This KRD object is cryptographically verified by the FIDO Server.

3.3.1 Type of RegisterRequest

dictionary RegisterRequest {
  OperationHeader header;  // Mandatory, header.op must be “Reg”
  ServerChallenge challenge;  // Mandatory.
  DOMString username;  // Mandatory, string[1..128].
  Policy policy;  // Mandatory
}

object[] uafRegisterRequest;  // Mandatory, not more than one element per version

Description:

- RegisterRequest contains a single registration request:
  - The field header contains the operation header for a registration operation.
  - The field challenge contains the server provided value.
The field username contains a human-readable user name intended to allow the user to distinguish and select from among different accounts – even at the same relying party.

The field policy defines which types of Authenticators are acceptable for this registration operation.

- In general, a single UAF registration message may convey multiple versions of Registration requests. It may not contain more than one element per version.

Example 5: UAF Register Request

```
{
  "header": { "op": "Reg", "upv": { "mj": 1, "mn": 0 }, "appID": "https://mycorp.-com/fido"},
  "challenge": "qwudh827hdbawd8qbdqj3bdq3uq56t324zwasdq4wrt",
  "username": "banking_personal",
  "policy": {
    "accepted": {
      "authenticationFactor": 00000000000001ff,
      "keyProtection": 000000000000000e,
      "attachment": 0000000000000ff,
      "secureDisplay": 0000000000001e,
      "supportedSchemes": "UAFV1TLV"
    },
    "disallowed": {"aaid": "1234#5678"}
  }
}
```

3.3.2 Type of RegisterResponse

```
dictionary AuthenticatorRegistrationAssertion {
  AAD aaid; // Mandatory.
  DOMString attestationCertificateChain; // Optional. base64url(byte[1..])
  DOMString scheme; // Mandatory
  DOMString krd; // Mandatory. base64url(byte[1..4096])
  Extension [] exts; // Optional
}
```

```
dictionary RegisterResponse {
  OperationHeader header; // Mandatory, OperationHeader.op must be "Reg"
  DOMString fcParams; // Mandatory, base64url encoded FinalChallengeParams
  AuthenticatorRegistrationAssertion[] assertions; // Mandatory.
}
```

RegisterResponse uafRegisterResponse;

Description:
• The dictionary `AuthenticatorRegistrationAssertion` contains the Authenticator's response to a `uafRegisterRequest` message:
  ○ The field `aaid` contains the AAID of the specific Authenticator being used for registration.
  ○ The field `attestationCertificateChain` contains the Authenticator’s Attestation certificate chain (excluding the root, which is included in the Authenticator Metadata [UAFAuthnrMetadata]) formatted as defined for the "x5c" type of JSON Web Key [JWK].
  ○ The field `scheme` contains the name of the Assertion Scheme used to encode KRD. See section UAF Supported Assertion Schemes for details.
  ○ The field `krd` contains the `KeyRegistrationData` structure that contains the newly generated Uauth.pub signed with the Attestation Private Key. This structure is produced by Authenticator and is used only in this Registration operation. Its format can vary from one Registration Scheme to another.
  ○ The field `exts` contains Extensions prepared by Authenticator.

• The dictionary `RegisterResponse` contains all fields related to the registration response:
  ○ The field `header` contains the operation header related to a UAF registration operation.
  ○ The field `fcParams` is the base64url encoded serialized [RFC4627] FinalChallengeParams using UTF8 encoding (see section Type of FinalChallengeParams) which contains all parameters required for the server to verify the Final Challenge.
  ○ The field `assertions` contains the response data for each Authenticator being registered.

• `uafRegisterResponse` contains the UAF Registration Response message.

3.3.3 Processing Rules

3.3.3.1 Registration Request Generation Rules for FIDO Server

The policy contains a 2-dimensional array of allowed `MatchCriteria` (see Type of 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, i.e. the list elements are alternatives.

• Construct appropriate registration policy p
For each set of Authenticators a (to be registered simultaneously) do

- Create MatchCriteria object m
- If m.aaid is provided - no other fields, except keyID, attachment and exts, MUST be provided
- If m.aaid is not provided - at least m.supportedAuthAlgs and m.supportedSchemes MUST be provided
- If this set of Authenticators is considered disallowed, append p to m.disallowed. Note: Server MUST include already registered AAIDs and KeyIDs into p.disallowed to hint the Client to not suggest registering these again
- If this set of Authenticators is considered accepted, append p to m.accepted, e.g. m.accepted[n] = set.

- Create a RegisterRequest object r with appropriate r.header
- Generate a random challenge and assign it to r.challenge
- Assign the username of the user to be registered to r.username
- Assign p to r.policy
- Append r to the array o of message with various versions (uafRegisterRequest)
- Send o to the FIDO Client

3.3.3.2 Registration Request Processing Rules for FIDO Client

- Choose the message m with major version 1 and minor version 0
- Parse the message m
  - If a mandatory field in UAF message is not present or a field doesn’t correspond to its type and value - reject the operation
- 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[].keyID
- Follow the priorities in server’s policy and drive user experience based on these priorities.
- Obtain FacetID of the requesting Application. Resolve AppID URI and make sure that this FacetID is listed in TrustedApps.
  - If FacetID is not in TrustedApps – reject the operation
- Obtain TLS data if it is available
- Create a FinalChallengeParams structure fcp and set fcp.appID, fcp.challenge, fcp.facetID, and fcp.tlsData appropriately. Serialize [RFC4627] fcp using UTF8 encoding and base64url encode it.
FinalChallenge = base64url(serialize(utf8encode(fcp)))

For each authenticator that matches UAF protocol version (see section Version Negotiation) and user agrees to register:

- Add AppID, FinalChallenge, and all other required fields to the ASMRequest [UAFASM].
- Send ASMRequest to the ASM

### 3.3.3.3 Registration Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section “Register Command”.

### 3.3.3.4 Registration Response Generation Rules for FIDO Client

- Create a uafRegisterResponse message
- Copy uafRegisterRequest.header into uafRegisterResponse.header
- Set uafRegisterResponse.fcParams to FinalChallenge (base64url encoded serialized and utf8 encoded FinalChallengeParams)
- Append the response from each Authenticator into uafRegisterResponse.assertions
- Send uafRegisterResponse message to FIDO Server

### 3.3.3.5 Registration Response Processing Rules for FIDO Server

**NOTE**

Exact error codes returned by FIDO server are not listed in the current specification. They will be listed in the next revision.

- Parse the message
  - If protocol version (uafRegisterResponse.header.upv) is not supported – reject the operation
  - If a mandatory field in UAF message is not present or a field doesn’t correspond to its type and value - reject the operation
- Verify that uafRegisterResponse.header.serverData, if used, passes any implementation-specific checks against its validity.
- base64url decode uafRegisterResponse.fcParams and convert it into an object (fcp)
Verify each field in \textit{fcp} and make sure it is valid:

\begin{itemize}
  \item Make sure \textit{fcp.appID} corresponds to the one stored in FIDO Server
  \item Make sure \textit{fcp.challenge} has really been generated by FIDO Server for this operation and it is not expired
  \item Make sure \textit{fcp.facetID} is in the local list of “trusted FacetIDs”
  \item Make sure \textit{fcp.tlsData} is as expected [TLS Binding]
  \item Reject the response if any of these checks fails
\end{itemize}

For each assertion \(a\) in \textit{uafRegisterResponse.assertions}

\begin{itemize}
  \item Locate Authenticator specific authentication algorithms from Authenticator Metadata [UAFAuthnrMetadata]
  \item Parse TLV data from \textit{a.krd} and make sure it has all the mandatory fields (indicated in Authenticator Metadata) it is supposed to have
  \item Hash \textit{uafRegisterResponse.fcParams} using hashing algorithm suitable for this authenticator type. Look up the hash algorithm in Authenticator Metadata, field \textit{AuthenticationAlgs}. It is the hash algorithm associated with the first entry related to a constant with prefix UAF\_ALG\_SIGN.
    \begin{itemize}
      \item \(FCHash = \text{hash}(uafRegisterResponse.fcParams)\)
    \end{itemize}
  \item Make sure that \textit{a.krd.FinalChallenge} == \(FCHash\)
    \begin{itemize}
      \item If comparison fails - continue with next assertion
    \end{itemize}
  \item if entry AttestationRootCertificate for this AAID in the Metadata [UAFAuthnrMetadata] contains at least one element:
    \begin{itemize}
      \item Obtain \textit{a.krd.Certificate} and related certificate chain from \textit{a.attestationCertificateChain}
      \item Obtain all entries of \textit{AttestationRootCertificate} for \textit{a.aaid} in Authenticator Metadata, field \textit{AttestationRootCertificate}.
      \item Verify \textit{krd.Certificate} and the entire certificate chain up to the Attestation Root Certificate using Certificate Path Validation as specified in [RFC5280]
        \begin{itemize}
          \item If verification fails – continue with next assertion
        \end{itemize}
      \item Verify \textit{krd.Signature} using \textit{krd.Certificate}
        \begin{itemize}
          \item If verification fails – continue with next assertion
        \end{itemize}
    \end{itemize}
  \item if entry \textit{AttestationRootCertificate} for this AAID in the Metadata is empty
    \begin{itemize}
      \item Verify \textit{krd.Signature} using \textit{krd.PublicKey}
        \begin{itemize}
          \item If verification fails – continue with next assertion
        \end{itemize}
      \item Verify \(a.aaid == krd.AAID\)
    \end{itemize}
\end{itemize}
• If not – continue with next assertion
  • Make sure that the set of successfully verified assertions meets the originally imposed policy
    ○ If they don’t meet the policy - treat the response as insufficient and reject
  • For each positively verified assertion a
    ○ Store a.krd.PublicKey, a.krd.KeyID, a.krd.SignCounter, a.krd.authenticatorVersion and a.krd.AAID 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).

3.4 Authentication Operation

During this operation FIDO Server asks FIDO Client to authenticate user with specified Authenticators and return an authentication response. In order for this operation to succeed the Authenticator and Relying Party MUST have a previously shared registration.
Diagram of cryptographic flow:
The FIDO Server sends the AppID (see section AppID and FacetID Assertion), the Authenticator Policy (see section Type of Policy) and the server generated Challenge (Chl, see section Type of ServerChallenge) to the FIDO Client.

The FIDO Client computes the Final Challenge Params (FCH) from the Server Challenge and some other values (see section Type of FinalChallengeParams) and sends the AppID and FCH to the Authenticator.

The Authenticator creates the SignedData object (see [UAFAuthnrCommands]) containing the FCH and some other values and signs it using the UAuth.priv key. This SignedData object is cryptographically verified by the FIDO Server.

3.4.1 Type of AuthenticationRequest

```plaintext
dictionary Transaction {
    DOMString contentType;  // Mandatory
    DOMString content;       // Mandatory. base64url(byte[1..8192])
}

dictionary AuthenticationRequest {
    OperationHeader header;  // Mandatory, header.op must be “Auth”
    ServerChallenge challenge;  // Mandatory
    Transaction transaction;  // Optional
    Policy policy;            // Mandatory
}
```
object[] uafAuthRequest;    // Mandatory, not more than one element per version

Description:

- The dictionary `Transaction` contains the Transaction Text provided by the FIDO Server:
  - The field `contentType` contains the Content-type according to [RFC2049], should be either “text/plain” or “image/png” [RFC2083].
    - If the type is “text/plain” then the content MUST be ASCII encoded text with a maximum of 200 characters.

  **NOTE**
  
The current specification only supports ASCII encoded transaction text. The next revision will support internationalized text.

- The field `content` contains the content of the transaction according to the content type.

- The dictionary `AuthenticationRequest` contains the UAF Authentication Request Message:
  - The field `header` contains the operation header for an authentication operation.
  - The field `challenge` contains the server-provided value.
  - The field `transaction` contains the transaction data to be explicitly confirmed by the user.
  - The field `policy` defines which types of Authenticators are acceptable for this authentication operation.

- `uafAuthRequest` Represents UAF Authentication request message. In general, a single message may convey multiple versions of Authentication requests. It may not contain more than one element per version.
Example 6: UAF Authentication Request

```
{
  "header": {"op": "Auth", "upv": { "mj": 1, "mn": 0 }, "appID": "https://mycorp.com/fido"},
  "challenge": "triz786ighwer8764g6574234515reg45z",
  "policy": {
    "accepted": [{
      "authenticationFactor": 00000000000001ff,
      "keyProtection": 000000000000000e,
      "attachment": 00000000000000ff,
      "secureDisplay": 000000000000001e,
      "supportedSchemes": "UAFV1TLV"}],
    "disallowed": {"aaid": "1234#5678"}
  }
}
```

3.4.2 Type of AuthenticationResponse

```
dictionary AuthenticatorSignAssertion {
  AAID aaid;          // Mandatory
  KeyID keyID;         // Mandatory
  DOMString scheme;   // Mandatory, e.g. “UAFV1TLV”
  DOMString signedData; // Mandatory, base64url(byte[1..4096])
  Extension[] exts;   // Optional
}

dictionary AuthenticationResponse {
  OperationHeader header; // Mandatory, header.op must be “Auth”
  DOMString fcParams;    // Mandatory, base64url encoded FinalChallengeParams
  AuthenticatorSignAssertion[] assertions; // Mandatory
}
```

Description:
- The dictionary AuthenticatorSignAssertion represents the Authenticator specific response:
  - The field `aaid` contains the Authenticator’s AAID.
  - The field `keyID` contains the unique KeyID related to Uauth.priv.
  - The field `scheme` contains the name of the Assertion Scheme used to encode `signedData` (e.g. “UAFV1TLV”).
  - The field `signedData` is a structure that contains cryptographic signature generated using Uauth.priv. Such a structure is produced by an Authenticator and is used only in Authentication operations.
  - The field `exts` contains extensions prepared by an Authenticator.
The dictionary AuthenticationResponse contains the UAF Authentication Response Message:

- The field header contains the operation header for the authentication operation.
- The field fcParams is the base64url encoded serialized [RFC4627] FinalChallengeParams in UTF8 encoding (see section Type of FinalChallengeParams) which contains all parameters required for the server to verify the Final Challenge.
- The field assertions contains the list of authenticator responses related to this authentication operation.

**Example 7: UAF Authentication Response**

```json
{
    "header": {"op": "Auth", "upv": { "mj": 1, "mn": 0 }},
    "fcParams": "eyJhcHBJRCI6Imh0dHBzOi8vbXljb3JwLmNvbS9maWRvIiwgImNoYWxsZW5nZSI6IjU0Njk4emhmZGtzamdoODc2dWpoZ2hqNyIsICJmYWNldElEiwiYXV0aF9sYXN0b3J5IjoiYW5kcm9pZDphcGsta2V5LWhhc2g6...jdsNXJTdzBSVmIvdmxXQV1rSy92QndrIiwgInRsc0RhdeGEiOiIifQ",
    "assertions": [
        {"AAID":"1234#abcd", "keyID": "1234def...", "scheme": "UAFV1TLV",
            "signedData": "..."},
        {"AAID":"1234#abce", "keyID": "fa73fg...", "scheme": "UAFV1TLV",
            "signedData": "..."}
    ]
}
```

### 3.4.3 Processing Rules

**3.4.3.1 Authentication Request Generation Rules for FIDO Server**

- Generate a random challenge
- Construct appropriate authentication policy
  - If MatchCriteria.aaid is provided then no other fields, except keyID, attachment and exts, MUST be provided
  - If MatchCriteria.aaid is not provided then at least supportedAuthenticationAlgs and supportedSchemes 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.
- Create an authentication request message for each supported version by putting generated data into these, assemble all these messages into an array and send to FIDO Client
3.4.3.2 Authentication Request Processing Rules for FIDO Client

- Choose the message with major version 1 and minor version 0
- 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
- Filter available Authenticators with the given policy and present the filtered list to User.
- Let the user select the preferred Authenticator.
- If \( \text{AuthRequest.policy.accepted} \) list is empty then suggest any registered Authenticator to the user for authentication
- Obtain FacetID of the requesting Application. Resolve AppID URI and make sure that this FacetID is listed in TrustedApps.
  - If FacetID is not in TrustedApps then reject the operation
- Obtain TLS data if its available
- Create a FinalChallengeParams structure \( fcp \) and set \( fcp.AppID, fcp.challenge, fcp.facetID, \) and \( fcp.tlsData \) appropriately. Serialize [RFC4627] \( fcp \) using UTF8 encoding and base64url encode it.
  - \( \text{FinalChallenge} = \text{base64url} (\text{serialize} (\text{utf8encode} (fcp))) \)
- For each authenticator that supports an Authenticator Interface Version AIV compatible with message version \( \text{AuthRequest.header.upv} \) (see section Version Negotiation) and user agrees to authenticate with:
  - Add AppID, FinalChallenge, KeyID, Transaction Text (if present), and all other required fields to the ASMRequest.
  - Send the ASMRequest to the ASM

3.4.3.3 Authentication Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section “Sign Command”.

3.4.3.4 Authentication Response Generation Rules for FIDO Client

- Create an AuthResponse message
- Copy \( \text{AuthRequest.header} \) into \( \text{AuthResponse.header} \)
Fill out `AuthResponse.FinalChallengeParams` with appropriate fields and then stringify it

Append the response from each Authenticator into `AuthResponse.assertions`

Send `AuthResponse` message to the FIDO Server

### 3.4.3.5 Authentication Response Processing Rules for FIDO Server

- Parse the message
  - If protocol version is not supported – reject the operation
  - If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
- Verify that `AuthResponse.header.serverData`, if used, passes any implementation-specific checks against its validity.
- base64url decode `AuthResponse.fcParams` and convert into an object (`fcp`)
- Verify each field in `fcp` and make sure it's valid:
  - Make sure AppID corresponds to the one stored in FIDO Server
  - Make sure FacetID is in “trusted FacetIDs”
  - Make sure TLSData is as expected [TLS Binding]
  - Make sure ServerChallenge is a really generated by FIDO Server and is not expired
  - Reject the response if any of these checks fails
- For each assertions in `AuthResponse.assertions`
  - Locate Uauth.pub public key associated with `AuthResponse.assertions.keyID`
    - If such record doesn't exist - continue with next assertion
  - Verify the `AuthResponse.assertions.aaid` against the AAID stored in the FIDO Server database at time of Registration.
    - If comparison fails – continue with next assertion
  - Locate Authenticator specific authentication algorithms from Authenticator Metadata (field `AuthenticationAlgs`)

**NOTE**

Exact error codes returned by FIDO server are not listed in the current specification. They will be listed in the next revision.
Parse `AuthResponse.assertions.signedData` and make sure it has all the mandatory fields (indicates in Authenticator Metadata) it’s supposed to have.

- Check the Sign Counter and make sure it is either not supported by the Authenticator or it has incremented (compared to the value stored in the user’s record)
  - If didn’t increment - continue with next assertion

- Hash `AuthResponse.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 `UAF_ALG_SIGN`.
  - \( FCHash = \text{hash(AuthResponse.FinalChallengeParams)} \)

- Make sure that `signedData.FinalChallenge == FCHash`
  - If comparison fails – continue with next assertion

- If `authenticationMode == 2`
  - Make sure there is a transaction cached on Relying Party side
    - If not – continue with next assertion
  
  - Hash the cached transaction using hashing algorithm suitable for this authenticator (same hash algorithm as used for FinalChallenge)
    - \( cachedTransHash = \text{hash(cachedTransaction)} \)

  - Make sure that the `cachedTransHash == signedData.TransactionHash`
    - If comparison fails – continue with next assertion

- Use Uauth.pub key and appropriate authentication algorithm to verify the signature included in SignedData
  - If signature verification fails – continue with next assertion

  - Make sure that the set of successfully verified assertions meets the originally imposed policy
    - If they don’t meet the policy – treat the response as insufficient and reject

3.5 Deregistration Operation

This operation allows FIDO Server to ask the FIDO Authenticator to delete keys related to the particular relying party. The FIDO Server should trigger this operation when the user removes his account at the relying party.

Note: there is no deregistration response object.
### 3.5.1 Type of DeregistrationRequest

```plaintext
dictionary DeregisterAuthenticator {
  AAID aaid;    // Mandatory
  KeyID keyID;  // Mandatory
}
dictionary DeregistrationRequest {
  OperationHeader header;  // Mandatory, header.op must be “Dereg”
  DeregisterAuthenticator[] authenticators; // Mandatory
}
```

object[] uafDeregRequest;  // Mandatory, not more than one element per version

**Description:**

- The dictionary `DeregisterAuthenticator` contains the data required to identify the Authenticator to be deregistered:
  - The field `aaid` contains the Authenticator’s AAID.
  - The field `keyID` contains the unique KeyID related to `Uauth.priv`. Note: we assume `keyID` to be unique within the scope of an AAID only.
- The dictionary `DeregistrationRequest` contains the UAF Deregistration Request Message:
  - The field `header` contains the operation header related to the deregistration request”.
  - The field `authenticators` contains the list of the Authenticators to be deregistered.
- `uafDeregRequest` represents UAF Deregistration request message. In general, a single message may convey multiple versions of deregistration requests. It may not contain more than one element per version.
Example 8: UAF Deregistration Request

```
{
    "header": {"op": "Dereg", "upv": { "mj": 1, "mn": 0 }, "appID": "https://mycorp.com/fido"},
    "authenticators": [
        { "aaid": "1234#abcd", "keyID": "14a504423f582727ea15c96d67200727f350dc8cc2289ed8106f3b6b7ee3ebb8" },
        { "aaid": "1234#abce", "keyID": "84a2f881a2ee7866b8fd4b94d00279a2b485b635823fcfacedef79eef0c7771e4" }
    ]
}
```

3.5.2 Processing Rules

3.5.2.1 Deregistration Request Generation Rules for FIDO Server

- Create a deregistration request message m with major version of \(m\.header\.upv\) set to 1 and minor version set to 0
- For each Authenticator to be deregistered
  - Create `DeregisterAuthenticator` object o for Authenticator to be deregistered
  - Set o.aaid and o.keyID appropriately
  - Append o the m.authenticators
- Delete related entry in FIDO Server's account database
- Send message to FIDO Client

3.5.2.2 Deregistration Request Processing Rules for FIDO Client

- Choose the message with major version 1 and minor version 0
- Parse the message
  - If a mandatory field in uafDeregRequest message is not present or a field doesn't correspond to its type and value – reject the operation
- For each Authenticator that supports an Authenticator Interface Version (AIV) compatible with the message version uafDeregRequest.header.upv:
  - Create Deregister Command for Authenticator, containing AppID and uafDeregRequest.keyID.
  - Send Deregister Command to Authenticator
3.5.2.3 Deregistration Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section “Deregister Command”.
4 Considerations

This is the considerations section. In this section the contents is informative by default, normative clauses are clearly marked as follows:

Normative
This is a normative clause

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 [UAFAuthnrMetadata] 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 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”.

Copyright © 2014 FIDO Alliance: REVIEW DRAFT
FIDO UAF Protocol Specification

FIDO Authenticators SHOULD support “Basic Attestation” described below. New Attestation mechanisms MAY be added to the protocol over time.

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 Key Registration Data object as attested Authenticators. This behavior MUST be properly declared in the Authenticator Metadata [UAFAuthnrMetadata].

4.1.2.1 Basic Attestation

FIDO Servers MUST have access to a trust anchor for verifying attestation public keys (i.e. Attestation Certificate trust store) and Authenticators MUST provide its attestation signature during the registration process. The attestation trust anchor is shared with FIDO Servers out of band (as part of the Metadata). This sharing process is out of scope of this document.

**NOTE**

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

The FIDO Server must load the appropriate Authenticator Attestation Root Certificate from its trust store based on the AAID provided in KeyRegistrationData. The remainder of the Attestation Certificate Chain is included in the UAF Registration Response (field AttestationCertificateChain) and potentially the KeyRegistrationData. These two partial chains must be combined. Off-loading portions of the Attestation Certificate Chain from the Authenticator reduces its memory requirements.

In this Basic Attestation model, a large number of Authenticators share the same Attestation certificate and Attestation Private Key in order to provide non-linkability (see section 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.

**Normative**

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.

**Normative**

Either (a) the manufacturer attestation root certificate or (b) the root certificate related to the AAID MUST be specified in the Authenticator Metadata (see section Authentici-
This root certificate MUST be dedicated to the issuance of Authenticator Attestation certificates.

In the case (a), the root certificate might cover multiple Authenticator types (i.e. multiple AAIDs). The AAID MUST be specified in the SubjectDN CommonName (oid 2.5.4.3) of the Attestation Certificate. In the case (b) this is not required as the root certificate only covers a single AAID.

The FIDO Server MUST verify and validate the attestation certificate chain according to [RFC5280], section 6 “Certificate Path Validation”. Certificate revocation status SHOULD be checked (e.g. using OCSP [RFC2560] or CRL based validation [RFC5280]).

### 4.1.3 Error Handling

FIDO Server will inform the calling Relying Party Web Application Server (see Figure 4.4: FIDO Interoperability Overview) about any error conditions encountered when generating or processing UAF messages through their proprietary API.

FIDO Authenticators will inform the FIDO Client (see Figure 4.4: FIDO Interoperability Overview) about any error conditions encountered when processing commands through the Authenticator Specific Module (ASM). See [UAFASM] and [UAFAuthnrCommands] for details.
4.1.4 Assertion Schemes

UAF Protocol is designed to be compatible with a variety of existing Authenticators (TPMs, Fingerprint Sensors, Secure Elements, etc.) and also future Authenticators designed for FIDO. Therefore extensibility is a core capability designed into the protocol.

It is considered that there are two particular aspects that need careful extensibility.

These are:

- Cryptographic key provisioning (Registration Assertions)
- Cryptographic authentication and signature (Authentication Assertion)

The combination of Registration and Authentication Assertion is called an Assertion Scheme.

The UAF protocol allows plugging in new Assertion Schemes. See also section 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 [FIDORegistry].

4.1.5 Username in Authenticator

FIDO UAF supports Authenticators acting as first authentication factor (i.e., replacing username and password). In this case the Authenticator stores the username (uniquely identifying an account at the specific relying party) internally. See [UAFAuthnrCommands], section “Sign Command” for details.

4.1.6 TLS Protected Communication

Normative

[C-General-010] In order to protect the data communication between FIDO Client and FIDO Server a protected TLS channel MUST be used by FIDO Client (or User Agent) and the [S-General-010] Relying Party for all protocol elements.

- The server endpoint of TLS connection MUST be at the Relying Party
- The client endpoint of TLS connection MUST be either FIDO Client or User Agent
- [C-General-010.1] TLS Client and Server [S-General-010.1] SHOULD use TLS v1.2 or newer. The use of TLS v1.1 is recommended 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].

- [C-General-10.3] TLS Client MUST verify and validate the server certificate chain according to [RFC5280], section 6 “Certificate Path Validation”. Certificate revocation status MUST be checked (e.g. using OCSP [RFC2560] or CRL based validation [RFC5280]), as well as via TLS server identity checking [RFC6125].

- [C-General-10.3] TLS Client’s trusted certificate root store MUST be properly maintained and at least require the CAs included in the root store to annually pass Web Trust or ETSI 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

Normative

Server Challenges (see section Type of ServerChallenge) need 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].

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.

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 se-
curity in FIDO, the underlying UAF protocol needs to provide a very high conceptual se-
curity level, so that the protocol isn’t the weakest link.

Relying Parties define Acceptable Assurance Levels FIDO Alliance envisions a
broad range of FIDO 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 Authenti-
cators meeting the security needs of the given business context, to compensate assur-
ance level deficits by adding appropriate implicit authentication measures, and to reject
authenticators not meeting their requirements. FIDO does not mandate a very high as-
surance 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 starts 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 au-
thentication, the application can perform any actions to the Cloud service. 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 authenti-
cation 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 con-
sequences.

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 Confi-
rmation”. 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 hu-
man readable content, i.e. the contract. From a security point, in the case of authentica-
tion 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 secure 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. 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 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 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. Providing the WYSIWYS capability (“Secure Display” component).

Some FIDO Authenticators might implement these functions in software running on the FIDO User Device, others might implement these functions in hardware. Some FIDO Authenticators might even be formally evaluated and accredited to some national scheme. Each FIDO Authenticator model has an attestation ID (AAID), uniquely identifying the related security properties. 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 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, 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 UAF protocol should be acceptable in all regions and hence MUST support the highest level of data protection. As a consequence, UAF doesn’t require transmission of biometric data to the relying party nor does it require the storage of biometric reference data [BioVocab] 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 [FIDO-UAF-Client-API]. 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 [Error: Reference source not found], 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 in combination with SHA-256 / SHA-512 hash algorithms. However, the RSA algorithm is also supported. See [FIDORegistry] “Authentication Algorithms and Key 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. 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 Application Isolation

There are two concepts implemented in UAF to prevent malicious applications from misusing AppID specific keys registered with FIDO Authenticators. First concept is called “FacetID Assertion” and second is based on the “KHAccessToken”.

Copyright © 2014 FIDO Alliance: REVIEW DRAFT
Applications often run on multiple platforms. For example, MyCorp runs a web service (on mycorp.com), has an Android app, and an iOS app. For many use cases, it is desirable to reliably establish that those three aspects of such a service all belong to the same application.

The main idea here is that instead of binding user authentication keys to web origins only, we bind them to a more generic application identity (AppID). So instead of saying...
“this keypair can only be used with mycorp.com”, we say “this keypair can only be used by the MyCorp applications”.

An “application”, for the purpose of this section, can have multiple facets. For example, the various facets of the “MyCorp application” could be:

- The web site mycorp.com
- An Android app signed with a certain public key
- The iOS app with the iOS Bundle ID com.mycorp
- ...

The following diagram depicts the facet architecture.
The calling app passes its AppID (e.g. “https://mycorp.com/app-identity”) to the API. On each platform, the FIDO Client will identify the calling app, and thus determine its FacetID (see section Type of TrustedApps). It then resolves the AppID and checks whether the FacetID is indeed included in the TrustedApps list returned by accessing the AppID (URL) using HTTP GET. For example, the browser is able to see the web origin of the calling app. Similarly, an Android system component like the Account Manager could identify the APK signing key of the Android app making an API call into the Account Manager. There is a similar mechanism in iOS.

Note, that the FIDO Client must correctly determine the FacetID (see section Type of TrustedApps for more details). FIDO Client and ASM vendors should implement vendor specific component verification methods in order to determine whether FIDO Client / ASM are legitimate.

The UAF protocol supports passing FacetID to the FIDO Server and including the FacetID in the computation of the authentication response.

A weakness in the facet identification mechanism results in a security vulnerability, i.e., identity assertions that are issued to facets other than those legitimately belonging to an application. In contrast, a weakness in the application identity matching mechanism results in a privacy (but not the above-mentioned security) vulnerability, causing the authenticator to use an authentication key (in other words, a user identifier) that should have been reserved for a different application.

### 4.3.3.2 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 keys 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 or reliably identify the calling application. Some platforms do not provide a 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 section Type of TLSData for more details.

4.3.5 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 UAF Protocol Specification

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 concept of Personas 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.

The Authenticator will 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 Commands specification [UAFAuthnrCommands] 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.

4.3.6 ServerData and KeyHandle

Data contained in the field serverData (see section Operation Header) of UAF requests is sent to the FIDO Client and will be echoed back to the FIDO Server as part of the related UAF response message.

The FIDO Server should not assume any kind of implicit integrity protection of such data nor any implicit session binding. FIDO Server must explicitly bind the serverData to an active session.

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 on the ciphertext, using a different key for the MAC, e.g. using HMAC [FIPS198-1]. Alternatively, an authenticated encryption scheme such as AES-GCM [SP 800-38D] or AES-CCM [SP 800-38C] could be used. Such a scheme provides both integrity and confidentiality in a single algorithm and using a single key.
If protecting serverData, the MAC should also be over some data that binds the data to its associate message, for example by re-including the challenge value in the authenticated serverData.

### 4.3.7 Authenticator Information retrieved through UAF Client API vs. Metadata

Several Authenticator properties (e.g. UserVerificationMethods, KeyProtection, SecureDisplay, ...) are available in the Metadata [UAFAuthnrMetadata] and through the UAF Client 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 UAF Client API. However, the properties retrieved through UAF Client API provide a good “hint” what to expect from the Authenticator. Such “hints” are well suited to drive and optimize the user experience.

### 4.3.8 Policy Verification

FIDO UAF Response messages do not include all parameters received in the related 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 section Processing Rules).

### 4.3.9 Replay Attack Protection

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 called “Server Challenge”. The FIDO Server only accepts incoming FIDO UAF messages, if the Server Challenge can be verified. This verification is done by recomputing the FinalChallenge included in the signed response object KRD and SignedData. See section Type of FinalChallengeParams.
4.3.10 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 privates 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 section Authentication Response Processing Rules for FIDO Server). 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.

4.4 Interoperability Considerations

FIDO supports Web Applications, Mobile Applications and Native PC Applications. These environments require different bindings in order to achieve interoperability.
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 ECMAScript binding [ECMAScript] is defined in [UAFAppAPI&Binding]. The protocol between the Web App and the Relying Party Web Application Server is typically proprietary. Web Apps SHALL use the UAF message format defined in this document (see section Protocol Details).

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. In order to ensure interoperability, such Apps SHALL use the UAF message format defined in this document (see section Protocol Details).

Native PC Applications play the role of the User Agent, the Web App (Client) and potentially also the FIDO Client. Those applications are typically expected to be independent from any particular Relying Party Web Application Server. These applications should use the UAF HTTP Binding defined in [UAFAppAPI&Binding].
NOTES

The objects KeyRegistrationData and SignedData [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

5.1 Assertion Scheme “UAFV1TLV”

This Assertion Scheme allows the Authenticator and the FIDO Server to exchange an asymmetric authentication key generated by the Authenticator.

The Authenticator MUST generate a key pair (UAuth.pub/UAuth.priv) to be used with authentication algorithms listed in “FIDO Registry of Predefined Values”.

This scheme is using Tag Length Value (TLV) compact encoding to encode KRD and SignedData messages generated by Authenticators. This is the default scheme for UAF protocol.

TAGs and Algorithms are defined in [FIDORegistry].

Normative

[S-Auth-005] Conforming FIDO Servers MUST support all authentication algorithms and key formats listed in document “FIDO Registry of Predefined Values”, section “Authentication Algorithms and Key Formats”.

[A-Auth-002] Conforming Authenticators MUST support at least one Authentication Algorithm and one Key Format listed in in [FIDORegistry], section “Authentication Algorithms and Key Formats”.

5.1.1 KeyRegistrationData

See [UAFAuthnrCommands], section “Register Command”.

5.1.2 SignedData

See [UAFAuthnrCommands], section “Sign Command”.

Copyright © 2014 FIDO Alliance: REVIEW DRAFT
6 Definitions

See [FIDO Glossary].
Bibliography

*FIDO Alliance Documents:*


*Other References:*


[ABNF] Augmented BNF for Syntax Specifications: ABNF (RFC 5234)


[FIPS180-4] Secure Hash Standard (SHS) (FIPS 180-4)


[RFC1321] The MD5 Message-Digest Algorithm (RFC 1321), R. Rivest, April 1992

[RFC1341] MIME (Multipurpose Internet Mail Extensions) (RFC1341)
[RFC2049] Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples (RFC2049)


[RFC2119] Key words for use in RFCs to Indicate Requirement Levels (RFC2119)


[RFC4086] Randomness Requirements for Security (RFC 4086)

[RFC4627] The application/json Media Type for JavaScript Object Notation (JSON), (RFC4627)

[RFC4648] The Base16, Base32, and Base64 Data Encodings (RFC 4648)


[RFC5246] See [TLS]

[RFC5280] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile (RFC 5280)


[RFC5929] Channel Bindings for TLS (RFC 5929)


[RFC6454] The Web Origin Concept (RFC 6454)

[RFC6979] Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) (RFC6979), T. Pornin, August 2013

[SHEFFER-TLS] Recommendations for Secure Use of TLS and DTLS, draft-sheffer-tls-bcp-00

[SP 800-131A] NIST Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths (NIST SP 800-131A)

Copyright © 2014 FIDO Alliance: REVIEW DRAFT
[SP 800-38C] NIST Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality (NIST SP 800-38C).

[SP 800-38D] NIST Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC (NIST SP 800-38D).

[SP 800-63-1] NIST Electronic Authentication Guideline SP 800-63-1 (NIST SP 800-63-1).


[TR-03116-4] eCard-Projekte der Bundesregierung, BSI TR-03116-4