FIDO UAF Specification Set Manifest

filename (this file):	fido-specifications-manifest-uaf-v1.0-rd-20131213
FIDO spec family:	UAF
targeted version:	v1.0
status type:	RD (**Review Draft**)
snapshot date:	2013-12-13
version designation:	uaf-v1.0-rd-20131213

Spec Filenames within the "uaf-v1.0-rd-20131213" REVIEW DRAFT spec set	**Description**
fido-uaf-protocol-uaf-v1.0-rd-20131213	UAF protocol spec proper.
fido-authenticator-asm-api-uaf-v1.0-rd-20131213	Authenticator-specific module API
fido-authenticator-commands-uaf-v1.0-rd-20131213	Authenticator commands.
fido-authenticator-metadata-uaf-v1.0-rd-20131213	Authenticator Metadata.
fido-registry-predefined-values-uaf-v1.0-rd-20131213	Pre-defined values registry.
fido-technical-glossary-uaf-v1.0-rd-20131213	Glossary.
fido-client-api-uaf-v1.0-rd-20131213	Non-normative webapp and Android client API.

UAF Protocol Specification Draft v0.12q

Specification Set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

Editors:

Dr. Rolf Lindemann, Nok Nok Labs, Inc.

Contributors:

Abstract:

Table of Contents

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.

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 Internet is expected to continue its breathtaking growth for the foreseeable future, with analysts predicting the online population to exceed 2.7B users by 2015. Unfortunately, this explosive growth has attracted online criminals responsible for over $114B in online fraud against consumers and businesses in 2011 alone. Traditional password-based authentication has failed to provide an effective defense against such sophisticated online attackers; alternative authentication approaches have failed to achieve widespread adoption due to their cumbersome user experience and onerous expense for organizations to deploy.

The goal of this Universal Authentication Framework is to provide a unified and extensible authentication mechanism that supplants passwords while avoiding the shortcoming 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 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 details and defines the UAF protocol as a network protocol. It defines the flow and content of all UAF messages and gives rationale of design choices.

Particular application level bindings are out of scope of this document. This document is not intended to answer questions such as:

- How does HTTP binding look like for UAF?
- How can FIDO Client communicate to FIDO enabled Authenticators?
- How can a web application communicate to FIDO Client?

The answers of these questions can be found in specific application UAF binding specifications.

2.2 Architecture

The following diagram depicts the entities involved in UAF protocol.

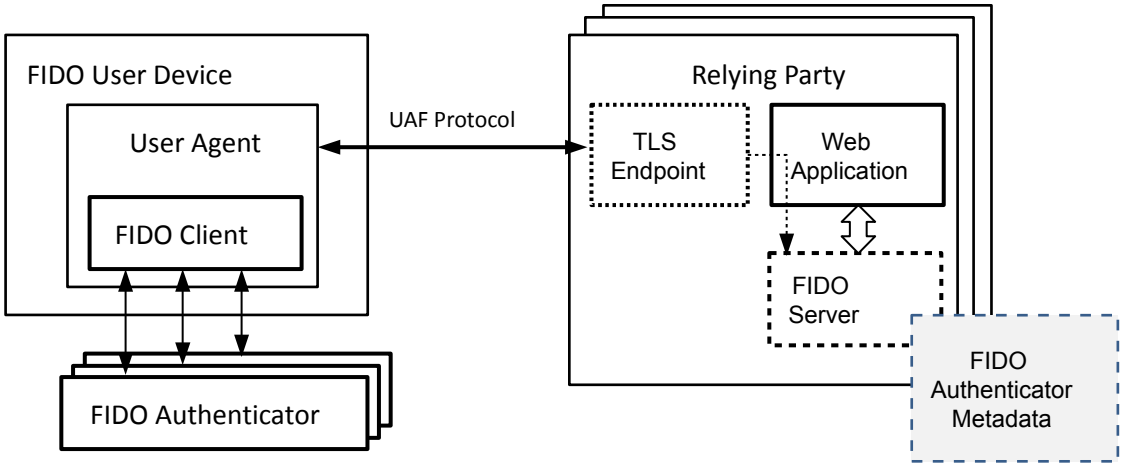

Figure 2.1: The UAF Architecture

There are three actors in the protocol:

- **FIDO Server**, running on Relying Party's infrastructure
- **FIDO Client**, part of User Agent and running on the FIDO user device
- **FIDO Authenticator**, integrated into the FIDO user device

It is assumed that FIDO Server has a built-in list of all supported Authenticator Specifications.

2.3 Protocol Conversation

The core UAF protocol consists of four conceptual conversations between FIDO Client and FIDO Server.

Although this document defines FIDO Server as the initiator of requests, in real world deployment the first UAF operation will always follow User Agent's request (e.g. a web request) to Relying Party.

- **Registration:** UAF allows the Relying Party to register FIDO enabled Authenticators with user's account. 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 a specified set of the transaction details.
- **Deregistration:** The Relying Party can trigger deletion of the Authentication Key material.

2.3.1 Registration

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

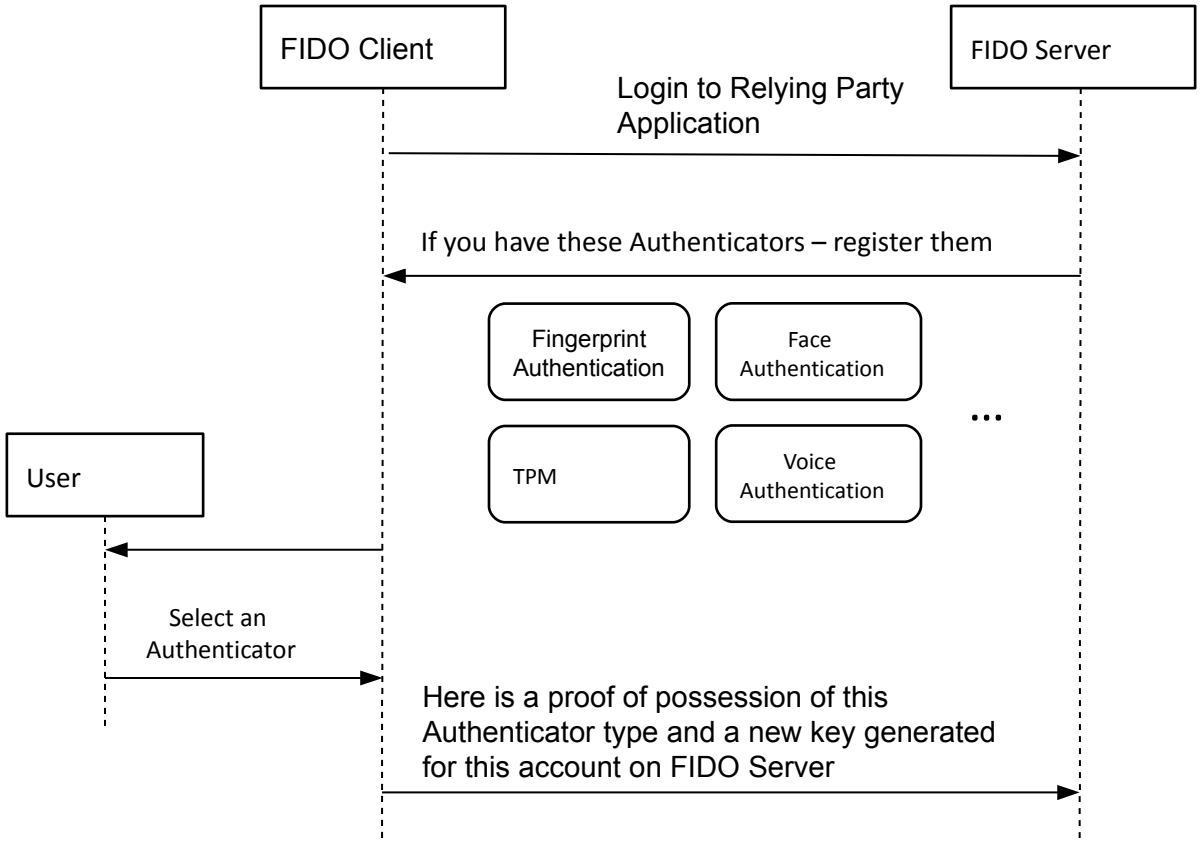

Figure 2.2: Registration

2.3.2 Authentication

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

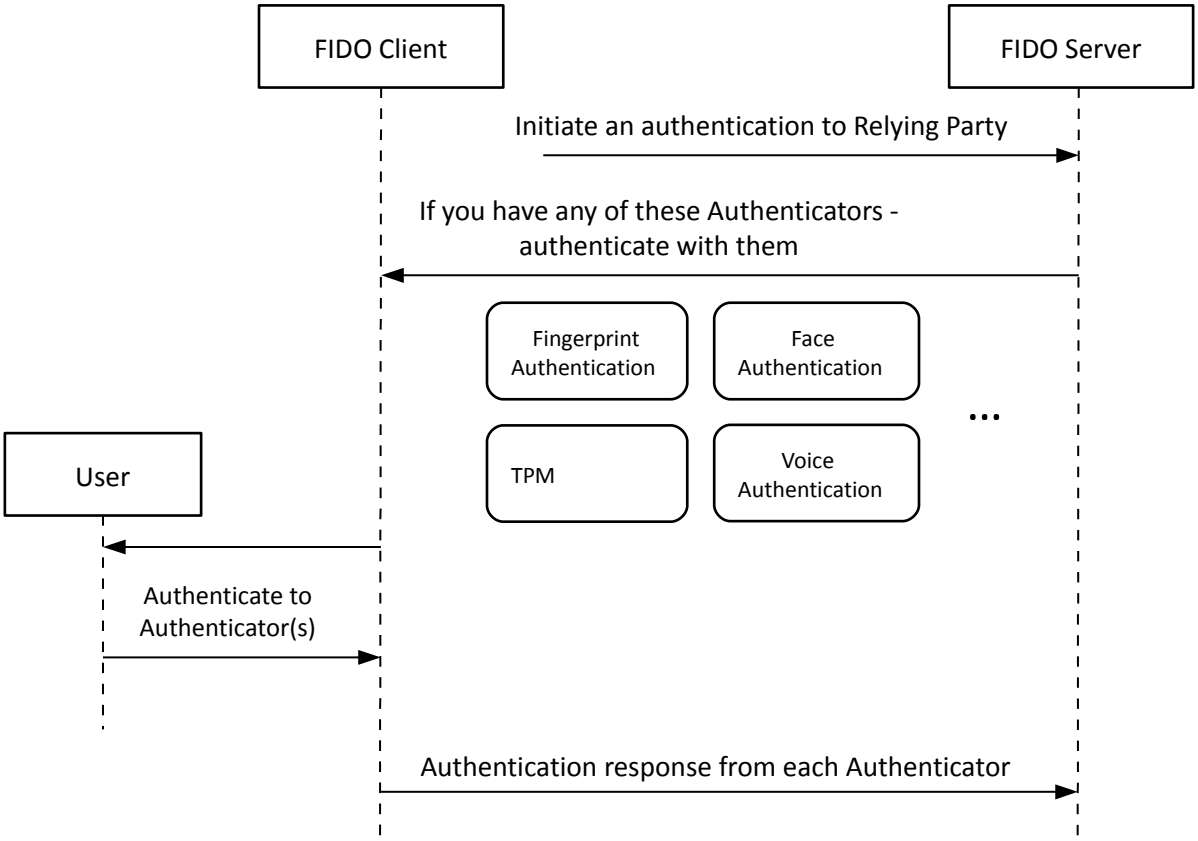

Figure 2.3: Authentication

2.3.3 Transaction Confirmation

The following figure depicts the Transaction Confirmation message flow.

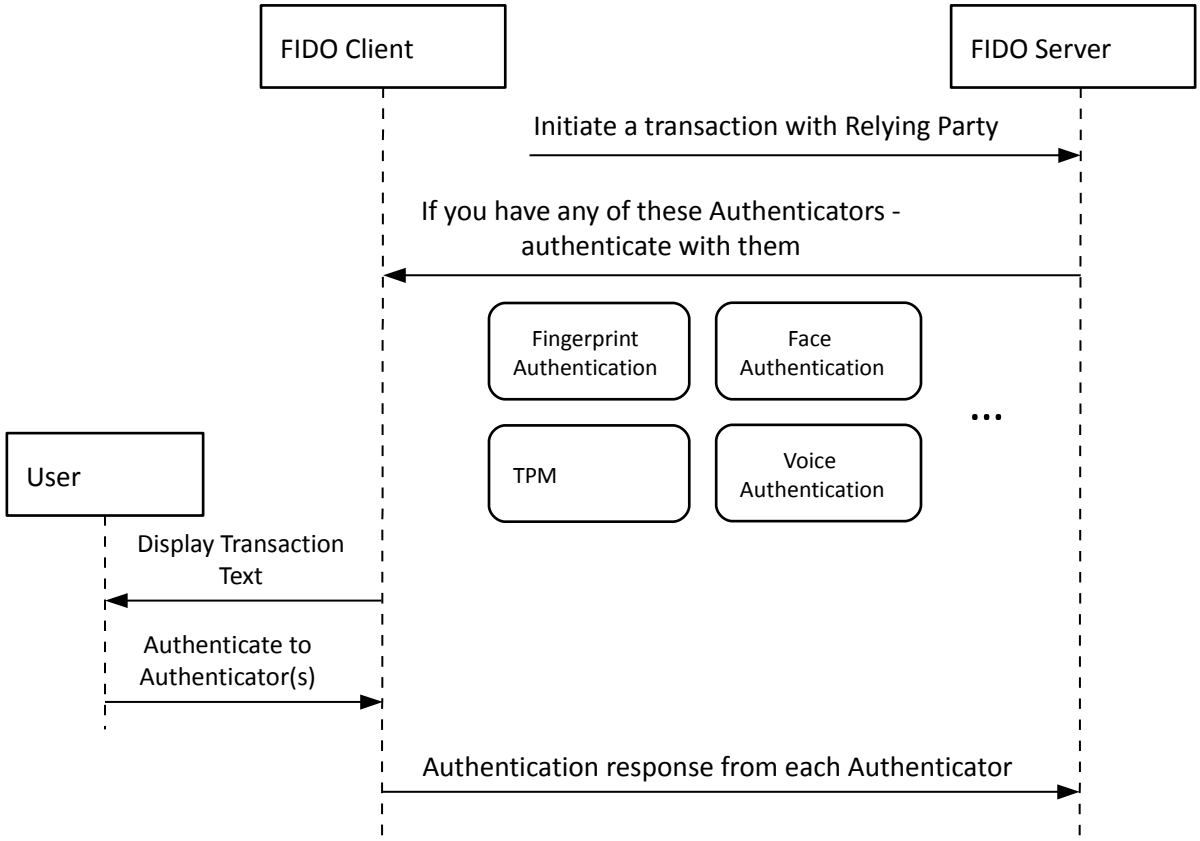

Figure 2.4: Transaction Confirmation

2.3.4 Deregistration

The following diagram depicts the Deregistration message flow.

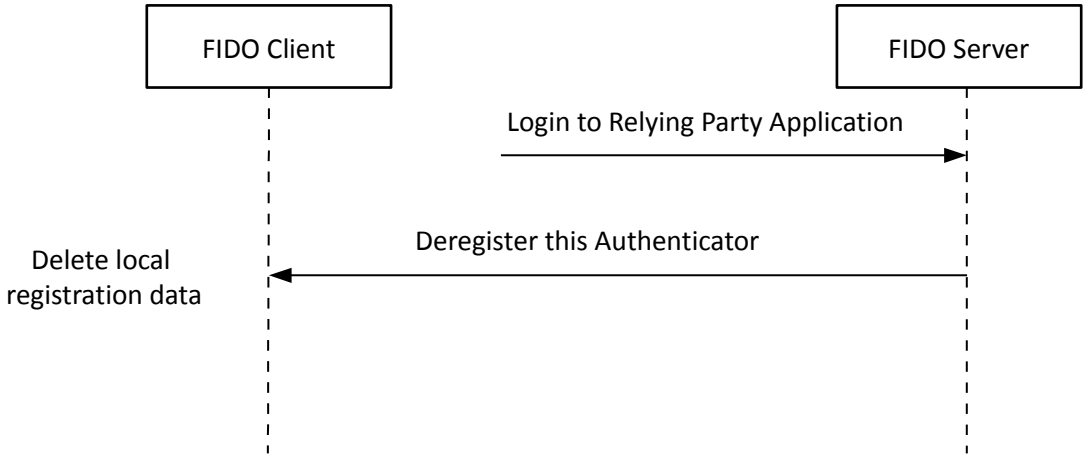

Figure 2.5: Deregistration

3 Protocol Details

This section provides details description of operations supported by UAF Protocol.

Support of all protocol elements is mandatory for conforming software, except if stated otherwise.

- All string literals in this specification are constructed from UNICODE codepoints within the set U+0000..U+007F. All strings are UTF-8 encoded unless otherwise specified
- 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 a 8-64 bytes data encoded in base64url
- The notation *string[5]* reads as a 5 character UTF-8 formatted string

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

The document uses WebIDL for messages structure definition. The protocol MUST use JSON format for delivering messages between FIDO Server and FIDO Client

3.1 Shared Structures and Types

This section defines types and structures shared by various operations.

3.1.1 Operation Header

Represents a UAF Message Request and Response header

```
dictionary OperationHeader {
  DOMString upv;         // Mandatory. string[3..7].
  DOMString aiv;         // Mandatory. string[3..7].
  DOMString op;          // Mandatory. Must be one of “Reg”, “Auth” or “Dereg”
  DOMString appID;       // Mandatory. string[1..512].
  DOMString sessionID;   // Optional, string[1..1536]
  Extension[] exts;      // Optional.
}
```

Description:

- **upv**: UAF protocol version.
- **aiv**: Authenticator Interface version
- **op**: Name of FIDO operation this message relates to. Note that “Auth” is used for authentication and transaction confirmation.
- **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 application id. It MUST be an URI with HTTPS protocol as FIDO Client will use it to load the list of FacetIDs using this URI.
- **sessionID**: 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 must be sent back intact. However, the FIDO Client can modify data. If the FIDO Servers relies on the integrity and relation of sessionID to the Challenge, then the FIDO Server has to provide appropriate security controls for that, e.g. wrapping sessionID with a secret key.
- **exts**: List of UAF Message Extensions.

3.1.2 Type of AAID

```
typedef DOMString AAID;       // string[9]
```

Description:

- **AAID**: Each Authenticator MUST have an AAID to identify UAF enabled Authenticator models globally. Only Authenticators with identical security characteristics may share the same AAID.
  - 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.
  - FIDO Alliance is responsible for assigning Authenticator Vendor Codes.
  - Authenticator Vendors are responsible for assigning model codes to their Authenticators. Authenticator Vendors have to make sure that there is a unique AAID assigned to Authenticators that have different security capabilities and need to be presented as unique FIDO Authenticator.

- Fixing firmware/software bugs, adding new firmware/software features, or changing the underlying hardware protection mechanisms will typically change the security characteristic and hence would require a different AAID to be used.

3.1.3 Type of KeyID

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

Description:

KeyID is a unique identifier used to refer to a specific Uauth.key. It is generated by the Authenticator and registered with a FIDO Server.

- The {AAID, KeyID} pair MUST uniquely identify an Authenticator's registration for a relying party. Whenever FIDO Server wants to provide specific information to a particular Authenticator it MUST use {AAID,KeyID} pair.
- KeyID must be base64url encoded within UAF message.
- KeyID may be used by Removable Authenticators which don't have internal storage and need to store the generated Uauth keys in wrapped form on FIDO Server.
- During authentication operation FIDO Server has to provide the KeyID back to Authenticator for the latter to unwrap Uauth.priv key and generate a signature using it.
- The exact structure and content of KeyID is implementation specific.

3.1.4 Type of ServerChallenge

```
// Mandatory. base64url(byte[8...64]). Server provided random challenge.
// Server Challenge is required to protect against replay
// attacks on an UAF messages. Protection measures against replay
// attacks on TLS level are implemented in TLS [TLS].
//
// 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.
typedef DOMString ServerChallenge;
```

3.1.5 Type of FinalChallengeParams

```
dictionary FinalChallengeParams {

  // Mandatory, string[1..512]. appID taken from request header
  DOMString appID;

  // Mandatory. Challenge taken from request data
  ServerChallenge challenge;

  // Mandatory, string[1..512]. Relying Party Facet ID.
  // facetID is:
  //
  //  For the Web, the facet id is the web origin, written as a
  //  URI with an empty path
  //        (e.g., "https://login.paypal.com/" (default ports are omitted).
  //
  //  For Android, the facet id is the URI
  //        android:apk-key-hash:<hash-of-apk-signing-cert>.
  //
  //  For iOS, the facet id is the URI
  //        ios:bundle-id:<ios-bundle-id-of-app>
  DOMString facetID;

  // TLS information to be sent by FIDO Client and
  // to be verified by FIDO Server
  TLSData tlsData;

}
```

3.1.6 Type of TLSData

```
dictionary TLSData_type {

  // Optional. base64url encoded hash of TLS server certificate.
  // If data is not available - it MUST be the string "None"
```

```
  DOMString serverEndPoint;

  // Optional. base64url encoded TLS channel Finished structure
  // (RFC5929 tls-unique).
  // If data is not available - it MUST be the string "None"
  DOMString tlsUnique;

  // Optional. The Channel ID public key used by this browser to
  // communicate with the above origin.
  // This parameter is optional, and missing if the browser
  // doesn't support Channel ID. It is present and set to the constant
  // 'unused' if the browser supports Channel ID, but is not using
  // Channel ID to talk to the above origin (presumably because the origin
  // server didn't signal support for the Channel ID TLS extension).
  // Otherwise (i.e., both browser and origin server at the above
  // origin support Channel ID), it is present and of type JwkKey
  (DOMString or JwkKey) channelID;
}
```

3.1.7 Type of JwkKey

```
// A dictionary representing the public key used by a browser for the
// Channel ID TLS extension. The current version of the Channel ID draft
// prescribes the algorithm (ECDSA) and curve used, so the dictionary will
// have the following parameters:
dictionary JwkKey {

  // signature algorithm used for Channel ID, i.e., the constant 'EC'
  DOMString alg;

  // Elliptic curve on which this public key is defined, i.e., the constant
  // 'P-256'
  DOMString crv;

  // base64url-encoding of the x coordinate of the public
  // key (big-endian, 32-byte value)
```

```
  DOMString x;

  // base64url-encoding of the y coordinate of the public
  // key (big-endian, 32-byte value)
  DOMString y;
}
```

3.1.8 Type of Extension

```
// Generic extensions used in various operations. ID identifies the extension
// and Data is an arbitrary data with a semantics agreed between Server and
// Client.
dictionary Extension {
  // Mandatory. string[1..32]. Extension ID.
  DOMString id;

  // Mandatory. base64url(byte[1..4096]). Extension data.
  // Interpretation of this data is up to the client
  DOMString data;
}
```

3.1.9 Type of TrustedApps

```
// Represents a structure holding a list of FacetIDs trusted by RP
// A HTTP GET query to the Application Identity URI MUST return a
// JSON object with this structure.
// Example:
//    {
//    'alg' : 'B64S256',
//     'ids' : [
//          // the hash of 'https://login.acme.com/':
//          't6qo8BvAa4LIfA3nuu1SNXWoh5d6ORLT24/h2qhwbn8=',
//          // the hash of
//          // 'android:apk-key-hash:2jmj7l5rSw0yVb/vlWAYkK/YBwk':
//          'Z2hR5jbDh8cTo7ObVP87nNZe5dVCjnBruitd2Cju0zI=',
//          // the hash of 'ios:bundle-id:com.acme.app':
//           '8/ix+Xfz0+6pGyME+tu+quiFd9pVNx6EuwMyBRhn0zw='
```

```
//    ]
//    }
dictionary TrustedApps {
  // Mandatory. Version of the structure. Must be "1.0".
  DOMString version;

  // Mandatory. Hashing and encoding algorithm.
  // For version "1.0" this must be either "B64S256" or "none".
  DOMString alg;

  // List of hashed FacetIDs. Each list element is string[1..512].
  // Each string in this list represents either the Facet ID itself
  // (if alg is "none") or the base64url encoding of the SHA256 hash
  // of the Facet ID (if alg is "B64S256")
  DOMString[] ids;
}
```

3.2 Version Negotiation

Version negotiation is based on the following rules:

- Each UAF message contains a version field (UAFProtocolVersion). UAFProtocolVersion negotiation is always only between FIDO Client and FIDO Server
- The communication between FIDO Client and FIDO Authenticator is also versioned (AuthenticatorInterfaceVersion) and grows independently from UAFProtocolVersion
- There is a clearly defined mapping between UAFProtocolVersion and AuthentiatorInterfaceVersion
  - When UAFProtocolVersion changes - AuthenticatorInterfaceVersion doesn't need to change
  - When AuthenticatorInterfaceVersion changes - UAFProtocolVersion MUST change
- Each time FIDO Server initiates a UAF operation to FIDO Client - it MUST send a list of as many messages as there are supported versions by Server
  - [{'ver': '1.0', ...}, {'ver': '1.2', ...}, ...]
- FIDO Client MUST know about the predefined mapping between different UAFProtocolVersions and AuthenticatorInterfaceVersions
- FIDO Client MUST filter out all Authenticators which it doesn't support

- FIDO Client MUST choose the highest version that both itself and Server support and process the message according to the rules specific to that version
- FIDO Client MUST prepare the response message according to the rules specific to the chosen version

UAFProtocolVersion is fixed as “1.0”. AuthentiatorInterfaceVersion is fixed as “1.0”.

3.3 Policy Generation and Parsing Rules

```
// Represents a matching criteria to be used in server policy
dictionary MatchCriteria {

  // Optional. Authenticator Vendor
  DOMString vendor;

  // Optional. Authenticator AAID
  AAID aaid;

  // Optional. Authenticator KeyID
  KeyID keyID;

  // Optional. A set of bit flags indicating
  // the authentication factor(s) supported by the authenticator.
  unsigned long long authenticationFactor;

  // Optional. A set of bit flags indicating the key protection
  // used by the authenticator.
  unsigned long long keyProtection;

  // Optional. A set of bit flags indicating the attachment type
  // of authenticator unsigned long long attachment;

  // Optional. A set of bit flags indicating the availability and type
  // of secure display
  unsigned long long secureDisplay;
```

```
  // Optional. base64url encoded bytearray of supported
  // authentication suite TAGs
  DOMString supportedAuthSuites;

  // Optional. List of supported encoding schemes the authenticators use for
  // KRD and SignData
  DOMString[] supportedSchemes;

  // Optional. List of Extensions
  Extension[] exts;
}

// Registration policy
dictionary Policy {

  // Mandatory.
  // A two dimensional array describing the required authenticator
  // characteristics for the server to accept a registration/authentication
  // for a particular purpose.
  // The first array index indicates OR conditions. Any authenticator(s)
  // satisfying any set of MatchCriteria in the first index is
  // acceptable to the server for registration/authentication.
  //
  // Sub-arrays of MatchCriteria in the second index indicates that multiple
  // authenticators must be registered/authenticated to be accepted by
  // the server.
  //
  // The MatchCriteria arrays represent 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).
  MatchCriteria[][] accepted;
```

```
  // Mandatory.
  // Any authenticator that matches any of the disallowed MatchCriteria MUST
// be excluded from eligibility for registration/authentication, regardless
  // of whether it matches any accepted MatchCriteria.
  MatchCriteria[] disallowed;
}
```

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

- During registration:
  - *Policy.accepted* is a list of combinations. Each combination indicates list of criteria of 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 the 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:
  - 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

3.4 Registration Operation

Registration operation allows FIDO Server and FIDO Authenticator to agree on Authentication Key (can be symmetric or asymmetric)

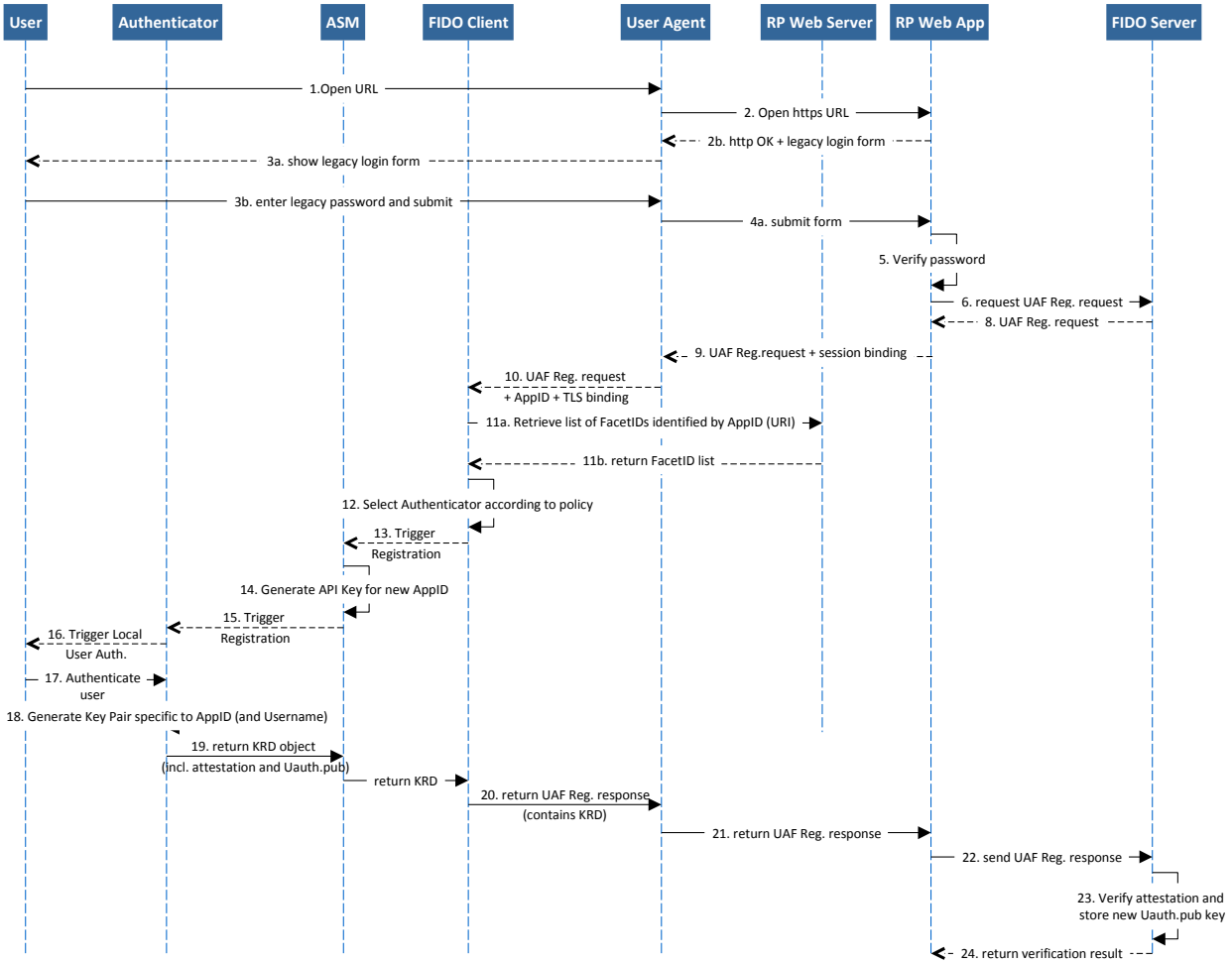

Figure 3.1: Sequence Diagram of UAF Registration

The following diagram depicts the cryptographic data flow for the Registration sequence.

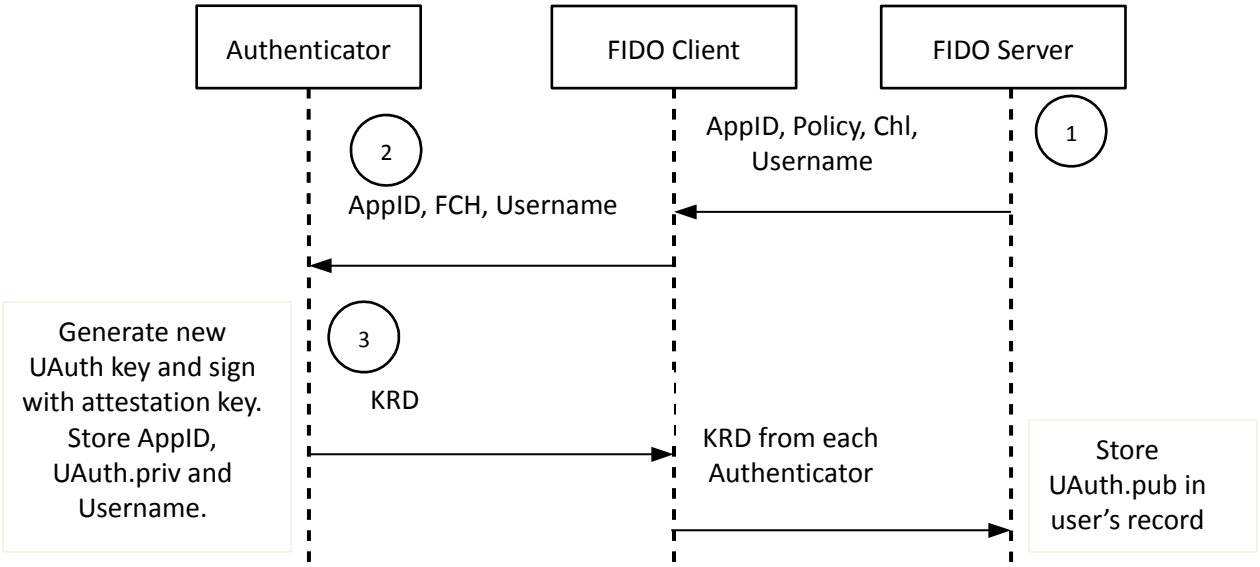

Figure 3.2: Cryptographic data flow of Registration

AppID – App Identity, Chl – Server challenge, FCH – "final" challenge params

3.4.1 Type of RegisterRequest

```
// UAF Registration Request Message
dictionary RegisterRequest {

  // Mandatory, OperationHeader.op must be "Reg"
  OperationHeader header;

  // Mandatory, Server provided challenge
  ServerChallenge challenge;

  // Mandatory, string[1..128]. User's human-readable username
  DOMString username;

  // Mandatory. Registration policy
  Policy policy;
}
```

```
// Represents the Authenticator specific response
dictionary AuthenticatorRegistrationAssertion {

  // Mandatory. Authenticator's AAID
  AAID aaid;

  // Optional. base64url(DER encoded certificate)
  // Authenticator's Attestation certificate chain.
  DOMString attestationCertificateChain;

  // Registration Scheme used to encode KRD (e.g. "UAF-TLV")
  DOMString krdScheme;

  // base64url(byte[1..4096])
  // KeyRegistrationData is a structure that contains newly generated
  // Uauth.pub signed with Attestation Private Key.
  // This structure is produced by Authenticator and is used only in
  // Registration operation.
  //
  // Its format can vary from one Registration Scheme to another. KRD is
  // a byte array converted into string containing KeyRegistrationData
  // as an opaque element.
  DOMString krd;

  // Optional. Extensions prepared by Authenticator
  Extension [] exts;
}
```

3.4.2 Type of RegisterResponse

```
// UAF Registration Response Message
dictionary RegisterResponse {

  // Mandatory, OperationHeader.op must be "Reg"
```

```
  OperationHeader header;

  // Final Challenge data
  FinalChallengeParams fcParams;

  // List of registered authenticators with specific data
  AuthenticatorRegistrationAssertion[] assertions;
}

// Represents UAF Registration request message.
// A single message may convey multiple versions of Registration requests.
// In this document only one version of request message is defined for
// registration - RegisterRequest.
// uafRegisterRequest MUST contain a single object of RegisterRequest type.
object[] uafRegisterRequest;

// Represents UAF Registration Response message.
RegisterResponse uafRegisterResponse;
```

Example 1: UAF Register Request

```
{
    “Op”: “Reg”, “Ver”: “1.0”, “AppId”: “https://mycorp.com/fido”,
    “Challenge”: “qwudh827hddbawd8qbdqj3bduq3duq56t324zwasdq4wrt”,
    “Username” :”banking_personal”,
    “Policy”: {
        “accepted” : [[{
            “authenticationFactor” : 00000000000001ff,
            “keyProtection” : 000000000000000e,
            “attachment” : 00000000000000ff,
            “secureDisplay” : 000000000000001e,
            “supportedSchemas” : “UAF-TLV”}]],
        “disallowed” : {“AAID” : “1234#5678”}
    }
}
```

3.4.3 Processing Rules

3.4.3.1 Registration Request Generation Rules for FIDO Server

- Generate a random challenge and assign a timeout to it.
- Construct appropriate registration policy
  - If MatchCritera.aaid is provided - no other fields, except keyID, attachment and exts, MUST be provided
  - If MatchCritera.aaid is not provided - at least supportedAuthenticationSuites and supportedSchemes MUST be provided
  - Server MUST include already registered AAIDs and KeyIDs into Policy.disallowed to hint the Client to not suggest registering these again
  - In case of step-up authentication every item in Policy.accepted MUST include already registered AAID and KeyID
- Create a registration request message for each supported version by putting generated data into these, assemble all these messages into an array and send to FIDO Client

Registration Request Processing Rules for FIDO Client

- Choose the message with highest supported version from the array provided by FIDO Server

- Parse the message
  - 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 Facet ID of the requesting Application. Resolve AppIdentity URI and make sure that this FacetID is listed in *TrustedApps*.
  - If FacetID is not in RP's trusted list - reject the operation
- Obtain TLS data if its available
- Create a FinalChallengeParams structure and include AppID | ServerChallenge | FacetID | TLSData (optional) into it. Stringify the FinalChallengeParams structure and provide to Authenticator
  - FinalChallenge = JSON.stringify(FinalChallengeParams)
- For each authenticator that matches UAF protocol version and user agrees to register:
  - Provide AppID (mandatory), Username (optional) and FinalChallenge (mandatory) to Authenticator.
  - Instruct the Authenticator to register

Registration Request Processing Rules for FIDO Authenticator

See UAF Authenticator Commands, section "Register Command".

Registration Response Generation Rules for FIDO Client

<TODO>

- Create a RegResponse message
- Copy RegRequest.header into RegResponse.header
- Fill out RegResponse.FinalChallengeParams with appropriate fields
- Append the response from each Authenticator into RegResponse.assertions
- Send RegResponse message to FIDO Server

Registration Response Processing Rules for FIDO Server

<TODO>

- Check AntiFraud signals (depending on set, e.g. ResetCtr and RegCtr)
- Verify Attestation

3.5 Authentication Operation

During this operation FIDO Server asks FIDO Client to authenticate user with specified Authenticators and return authentication response. In order for this operation to succeed Authenticator MUST be already registered with Relying Party.

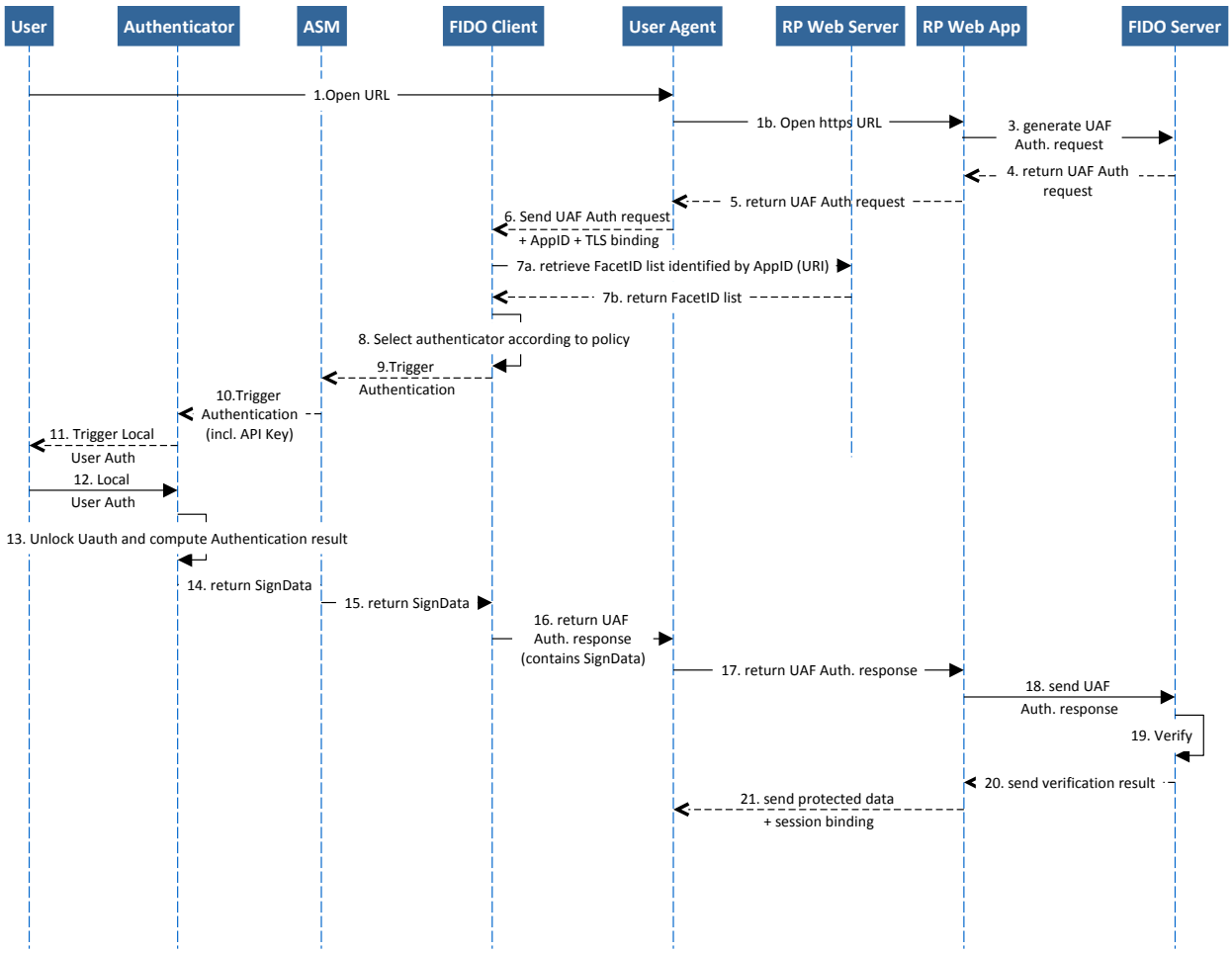

Figure 3.3: Sequence Diagram for UAF Authentication

Diagram of cryptographic flow:

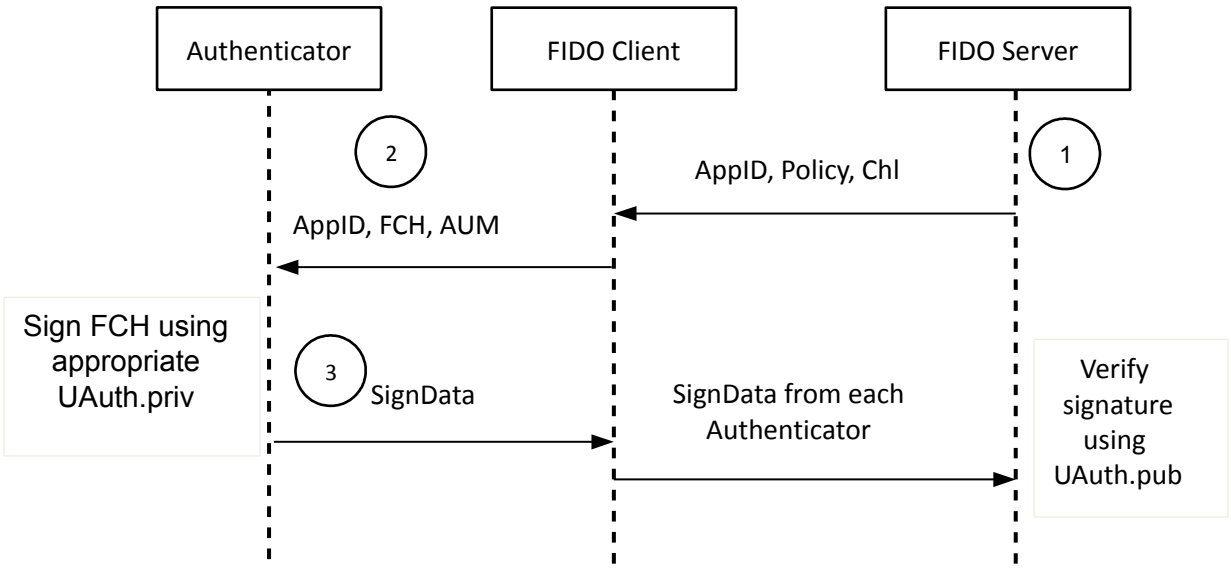

Figure 3.4: Cryptographic data flow of the Authentication message sequence

AppID – App Identity, AUM – Authentication mode, Chl – Server challenge, FCH – "final" challenge params

3.5.1 Type of AuthenticationRequest

```
// Transaction Text provided by FIDO Server.
dictionary Transaction {

  // Mandatory. Content-type of "text" according to [RFC1341]
  DOMString contentType;

  // base64url(byte[1..4096]), Mandatory
  // This is the content of the transaction according to the content type.
  DOMString content;
}

// UAF Authentication Request Message
dictionary AuthenticationRequest {

  // Mandatory, OperationHeader.op must be "Auth"
```

```
  OperationHeader header;

  // Mandatory, Server provided challenge
  ServerChallenge challenge;

  // Mandatory. Authentication Mode indicates the mode of authentication
  // operation at the time of signature generation.
  //
  // authenticationMode is an integer and may have one of the following
  // values:
  //  1 – Authentication with user not present (silent authentication).
  //  2 – Authentication with user present.
  //  3 – User present and transaction shown on a secure
  //  display. For this mode transaction text hash must be included in
  //  the final signature.
  short authenticationMode;

  // Optional. Transaction data.
  Transaction transaction;

  // Mandatory. RP's Authentication Policy
  Policy policy;
}

// Represents the Authenticator specific response
dictionary AuthenticatorSignAssertion_type {

  // Mandatory. Authenticator's AAID
  AAID aaid;

  // Mandatory. UAuth.priv unique KeyID
  KeyID keyID;

  // Authentication Scheme used to encode signData (e.g. "UAF-TLV")
  DOMString signDataScheme;
```

```
  // Mandatory. base64url(byte[1..4096])
  // SignData is a structure that contains cryptographic signature using
  // Uauth.priv. This structure is produced by Authenticator and is used only
  // in Authentication operation.
  // SignData is a byte array converted into string containing SignData
  // as an opaque element.
  DOMString signData;

  // Optional. Extensions prepared by Authenticator
  Extension[] exts;
}
```

Example 2: UAF Authentication Request

```
{“Op”: “Auth”, “Ver”: “1.0”, “AppId”: “https://mycorp.com/fido”, “Challenge”:
“triz786ighwer8764g6574234515reg45z”, “AuthMode” :2, “Policy”: {
      “accepted” : [[{“authenticationFactor” : 00000000000001ff, “keyProtection” :
000000000000000e, “attachment” : 00000000000000ff, “secureDisplay” :
000000000000001e, “supportedSchemas” : “UAF-TLV”}]],
      “disallowed” : {“AAID” : “1234#5678”}
      }
}
```

3.5.2 Type of AuthenticationResponse

```
// UAF Authentication Response Message
dictionary AuthenticationResponse {

  // Mandatory, OperationHeader.op must be “Auth”
  OperationHeader header;

  // Final Challenge data
  FinalChallengeParams fcParams;

  // List of authenticator responses
  AuthenticatorSignAssertion[] assertions;
}
```

```
// Represents UAF Authentication request message.
// A single message may convey multiple versions of Authentication requests.
// In this document only one version of request message is defined for
// authentication - AuthenticationRequest.
// uafAuthRequest MUST contain a single object of AuthenticationRequest type.
object[] uafAuthRequest;
```

Example 3: UAF Authentication Response

```
{“Op”: “Auth”, “Ver”: “1.0”, “FCParams”: {“AppID” “https://mycorp.com/fido”, “Chal-
lenge” :”54698zhfdksjgh876ujhghj7”, “FacetID”:” android:apk-key-
hash:2jmj7l5rSw0yVb/vlWAYkK/YBwk”, “TLSData”:””}, “Auths”: [{“AAID”:”1234#abcd”,
“signDataScheme” : “UAF-TLV”, “SignData”: “…”}, {“AAID”:”1234#abce”, “signDataScheme”
: “UAF-TLV”, “SignData”:”…”}]}
```

3.5.3 Processing Rules

Authentication Request Generation Rules for FIDO Server

- Generate a random challenge and assign a timeout to it
- Construct appropriate authentication policy
  - If MatchCritera.aaid is provided - no other fields, except keyID, attachment and exts, MUST be provided
  - If MatchCritera.aaid is not provided - at least supportedAuthenticationSuites and supportedSchemes MUST be provided
  - In case of step-up authentication every item in Policy.accepted MUST include already registered AAID and KeyID
- 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

Authentication Request Processing Rules for FIDO Client

- Choose the message with highest supported version from the array provided by FIDO Server
- Parse the message

    - 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 available Authenticators with the given policy and present the filtered list to User
- If AuthRequest.policy.accepted list is empty – suggest any registered Authenticator to the user for authentication
- Obtain Facet ID of the requesting Application. Resolve AppIdentity URI and make sure that this FacetID is listed in *TrustedApps*.
    - If FacetID is not in RP's trusted list - reject the operation
- Obtain TLS data if its available
- Create a FinalChallengeParams structure and include AppID | ServerChallenge | FacetID | TLSData (optional) into it. Stringify the FinalChallengeParams structure and provide to Authenticator
    - FinalChallenge = JSON.stringify(FinalChallengeParams)
- For each authenticator that matches message version and user agrees to authenticate with:
    - Provide AppID (mandatory), Final Challenge (mandatory), KeyID (mandatory), Authentication Mode (mandatory) and Transaction Text (mandatory) to Authenticator.
    - Instruct the authenticator to authenticate

Authentication Request Processing Rules for FIDO Authenticator

See UAF Authenticator Commands, section "Sign Command".

Authentication Response Generation Rules for FIDO Client

- Create a AuthResponse message
- Copy AuthRequest.header into AuthResponse.header
- Fill out AuthResponse.FinalChallengeParams with appropriate fields
- Append the response from each Authenticator into AuthResponse.assertions
- Send AuthResponse message to FIDO Server

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
- Make sure that AuthResponse.header.sessionID is intact
- Verify each field in AuthResponse.FinalChallengeParams 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
  - 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 AuthResponse.assertions
  - Locate Uauth.pub public key associated with AuthResponse.assertion-s.keyID
    - If such record doesn't not present - continue with next assertion
  - Locate Authenticator specific authentication suite from Authenticator Metadata
  - Parse AuthResponse.assertions.signData and make sure it has all the manda-tory fields (indicates in Authenticator Metadata) it's supposed to have
  - Check the Sign Counter and make sure it has incremented
    - If didn't increment - continue with next assertion
  - Stringify the contents of AuthResponse.FinalChallengeParams and hash it using hashing algorithm suitable for this authenticator type (look up the algorithm in Authenticator Metadata)
    - stringifiedFC=JSON.stringify(AuthResponse.FinalChallengeParams)
  - If authenticationMode == 3
    - Make sure there is a transaction cached on Relying Party side
      - If no - continue with next assertion
    - Hash the cached transaction using hashing algorithm suitable for this authenticator (look up the algorithm in Authenticator Metadata)

- cachedTransHash = hash(cachedTransaction)
- Make sure that the cachedTransHash=signData.TransactionHash
  - If comparison fails - continue with next assertion
- Append cachedTransHash into final challenge
  - stringifiedFCHash = hash(stringifiedFC | cachedTransHash)

○ If authenticationMode != 3

- stringifiedFCHash = hash(stringifiedFC)

○ Make sure that signData.Challenge = stringifiedFCHash

- If comparison fails - continue with next assertion

○ Use Uauth.pub key and appropriate authentication suite to verify the signature included in SignData

- 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

3.6 Deregistration Operation

This operation allows FIDO Server to delete keys from FIDO Authenticator.

```
// Represents a deregistering authenticator data
```

3.6.1 Type of DeregistrationRequest

```
dictionary DeregisterAuthenticator_type {

  // Optional. Authenticator's AAID
  AAID AAID;

  // Mandatory. UAuth.priv unique KeyID
  KeyID KeyID;
}
```

```
// UAF Deregistration Request Message
dictionary DeregistrationRequest {

  // Mandatory, OperationHeader.op must be "Dereg"
  OperationHeader header;

  // Mandatory. List of authenticators to be deregistered
  DeregisterAuthenticator[] authenticator;
}

// Represents UAF Deregistration request message.
// A single message may convey multiple versions of Deregistration requests.
// In this document only one version of request message is defined for
// authentication – DeregistrationRequest.
// uafDeregRequest MUST contain one object of DeregistrationRequest type.
object[] uafDeregRequest;
```

Example 4: UAF Deregistration Request

```
{"Op": "Dereg", "Ver": "1.0", "AppId": "https://mycorp.com/fido", "Policy": {"Auths":
[{"AAID":"1234#abcd", "KeyID": "14a504423f582727ea15c96d67200727f350dc8c-
c2289ed8106f3b6b7ee3ebb8"}, {"AAID":"1234#abce", "KeyID": "84a2f881a2ee7866b8fd4d-
b94d00279a2b485b635823fcfadef79eef0c7771e4"}]} }
```

3.6.2 Processing Rules

Request Generation Rules for FIDO Server

- Create a de-registration request message for each supported version by putting place AAID and KeyID of the deregistering authenticators into these, assemble all these messages into an array and send to FIDO Client

Message Processing Rules for FIDO Client

- Choose the message with highest supported version from the array provided by FIDO Server
- Parse the message

    - If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
- Find authenticators based on AAID and KeyID
- For each authenticator that matches message version:
    - Provide AppID (mandatory) and KeyID (mandatory) to Authenticator.
    - Instruct the authenticator to deregister

Message Processing Rules for FIDO Authenticator

See UAF Authenticator Commands, 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 a built-in list of all supported Authenticators and their corresponding metadata. Authenticator Metadata contains information such as:

- Supported Registration and Authentication Schemes
- Authentication Factor, Installation type and other supplementary information, etc.

In order to make a decision which Authenticators are appropriate for a specific transaction, FIDO Server looks up the list of Authenticator Metadata and uses this data to make decisions.

Normative

Authenticator Metadata is identified by a unique ID called 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 who 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 a malware sitting on client machine but is really a Authenticator of model “1234#5678”.

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

4.1.2.1 Basic Attestation

FIDO Servers and Authenticators MUST have pre-shared attestation public keys (i.e. Attestation Certificate trust store) and Authenticator MUST provide its attestation signature during registration process.

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

FIDO Server MUST load appropriate Authenticator Attestation (Root-) Certificate from trust store based on 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 have to be combined. The ability to off-load 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). So 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

In the case of basic attestation, the minimum number of authenticators sharing the same Attestation key is 100,000.

Normative

Either (a) the manufacturer attestation root certificate or (b) the root certificate related to the AAID needs to be specified in the Authenticator Metadata (see section Authenticator Metadata).

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.

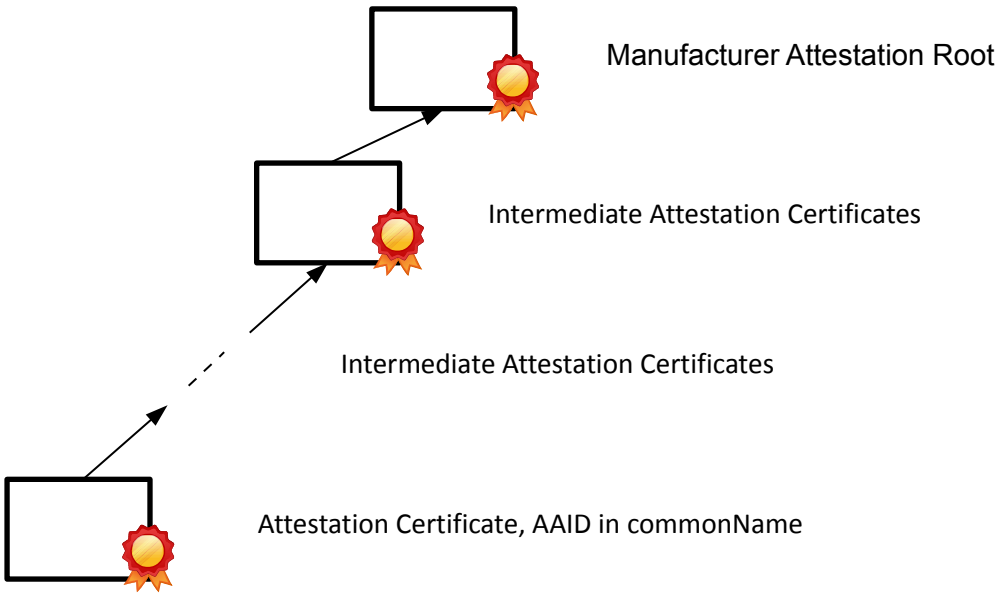

Figure 4.1: Manufacturer Attestation Root

4.1.3 Error Handling

FIDO Server will inform the calling Relying Party Web Application Server (see Error: Reference source not found in section Personas) about any error conditions encountered when generating or processing UAF messages through their proprietary API.

FIDO Authenticators will inform the FIDO Client (see Error: Reference source not found in section Personas) about any error conditions encountered when processing commands through the Authenticator Specific Module (ASM). See document “ASM Plugin and Authenticator API Specification” for details.

For FIDO Client we distinguish two different cases:

1. **Web Applications or Mobile Applications** In this case the FIDO Client will inform the Web App Client or the Mobile App (see Error: Reference source not found in section Personas) about any error conditions encountered when processing UAF messages through the proprietary API. The Web Application or Mobile Application will transmit the error code to the Web Application server through a proprietary protocol.
2. **Native PC Applications** In this case the error code needs to be transmitted through either the UAF message or the HTTP binding.

4.1.4 Registration and Authentication Schemes

UAF Protocol is designed to be compatible with 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" (called Registration Scheme)
- "Cryptographic signature schemes" (called Authentication Scheme)

The UAF protocol allows plugging in new "Registration Schemes" and also supporting new Authentication Schemes, specific to Authenticators.

The Registration Scheme defines how a cryptographic key is exchanged between the Authenticator and the FIDO Server. If in the future one finds a better for key exchange, a new Registration Scheme might be defined and plugged to the protocol.

The Authentication Scheme defines how the Authenticator generates a cryptographic signature.

4.1.5 Username in Authenticator

FIDO UAF supports Authenticators which can be used to replace username and password. In this case the Authenticator stores the username (uniquely identifying an account at the specific relying party) internally and adds it to the UAF SignData message. See UAF Authenticator Commands, 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 v1.1 or newer. Use of TLS v1.2 is RECOMMENDED. The "anon" and "null" crypto suites are not allowed and MUST be rejected; insecure crypto-algorithms in TLS (e.g. MD5, RC4, SHA1) SHOULD be avoided [SP800-131A].

- [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 or CRL based validation).
- [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] 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].

4.2.2 TODO: iOS Implementations of FIDO Clients

see iSec Partner document issue #3

4.3 Security Considerations

There is no “one size fits all” authentication method. The FIDO goal is to decouple the authentication method from the authentication protocol and the authentication server, and to support a broad range of authentication 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 security in FIDO, the underlying UAF protocol needs to provide a very high conceptual security level, so that the protocol isn't the weakest link.

Relying Parties define Acceptable Assurance Levels 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 Authenticators meeting the security needs of the given business context, to compensate assurance level deficits by adding appropriate implicit authentication measures, and to reject authenticators not meeting their requirements. FIDO does not mandate a very high assurance level for FIDO Authenticators, instead it provides the basis for authenticator and authentication 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 authentication, 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 authentication times out. This is a very convenient way as the user doesn't get distracted by manual actions required for the authentication. It is suitable for actions with low risk consequences.

However, in some situations it is important for the relying party to know that a user really has seen and accepted a particular content *before* he authenticates it. This method is typically being used when non-repudiation is required. The resulting requirement for this scenario is called What You See Is What You Sign (WYSIWYS).

UAF supports both methods; they are called "Authentication" and "Transaction Confirmation". The technical difference is, that with Authentication the user confirms a random challenge, where in the case of Transaction Confirmation the user confirms a human readable content, i.e. the contract. From a security point, in the case of authentication the application needs to be trusted as it performs any action once the authenticated communication channel has been established. In the case of Transaction Confirmation only the 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 Authentication Method from Authentication Protocol In order to decouple the user authentication method from the authentication protocol, UAF is based on an extensible set of cryptographic authentication algorithms. The cryptographic secret will be unlocked after authenticating the user (i.e. after user-to-authenticator authentication). 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. 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.

However, some cryptographic hardware Authenticators (e.g. some National Electronic ID cards and legacy PKI smart cards) do not support such a scheme. If such cryptographic hardware is used as FIDO Authenticator, then non-linkability typically cannot be provided.

Relying parties can discover the AAIDs of all enabled FIDO Authenticators on the FIDO User Device using the Java Script interface (see owp-ios). 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 owp-ios, privacy considerations, section 16).

4.3.1 FIDO Authenticator Security

See Authenticator Commands.

4.3.2 Cryptographic Algorithms

In order to keep key sizes small and to make private key operations fast enough for small devices, we prefer ECDSA and suggest the use of either NIST or BrainPool curves (see [RFC5639]) in combination with SHA-256 / SHA-512 hash algorithms.

4.3.3 Application Isolation

There are two concepts implemented in UAF to prevent malicious applications from misusing App identity specific keys registered with FIDO Authenticators. First concept is called "FaceID Assertion" and second is called "API Key Isolation".

4.3.3.1 FacetID Assertion

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. 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
- The web site mycorp-payments.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.

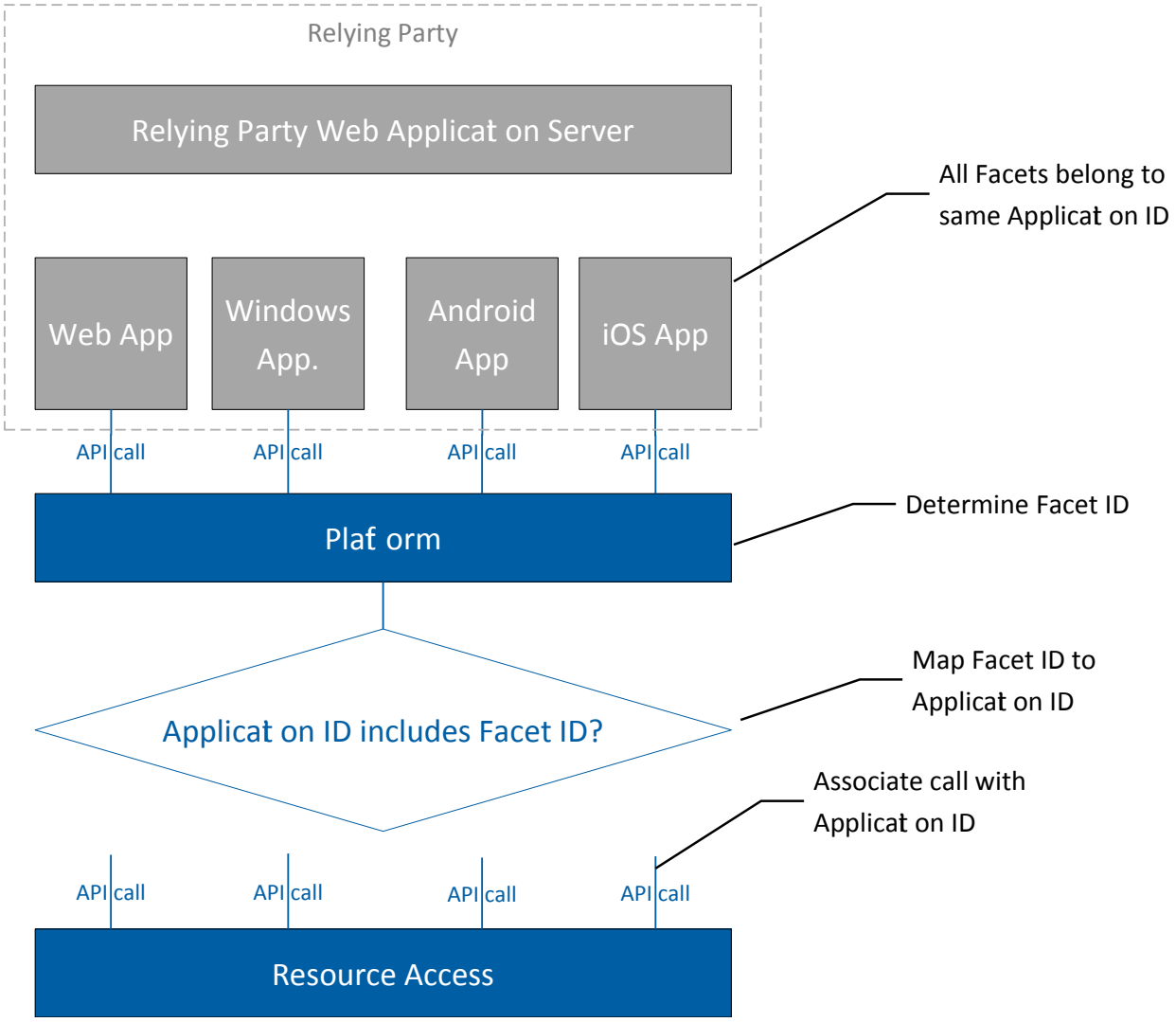

Figure 4.2: Multiple Application Facets

The calling app passes its Application Identity (e.g. “https://mycorp.com/app-identity”) to the API. On each platform, the FIDO Client will identify the calling app, and thus deter-

mine its Facet ID. It then resolves the Application Identity and checks whether the Facet ID is indeed included in the list returned by accessing the Application Identity URL. For example, the browser extension (or, in the future the browser itself) will be 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 needs to be trusted to correctly determine the FacetID.

The UAF protocol supports passing Facet ID to the FIDO Server and including the Facet ID 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.

Normative

In order to properly complement federation protocols, the list of Facet IDs may only include a single web origin facet.

4.3.3.2 Isolation using API Keys

Authenticators might be implemented in dedicated hardware and hence might not be able to verify the caller (i.e. ASM).

This API Key technique allows making sure that only the intended ASM can use the keys generated by the FIDO Authenticator. It is based on Trust On First Use (TOFU) concept.

FIDO Authenticator is capable of binding UAuth keys with a key provided by a caller (e.g. the ASM). This key is called API Key.

Normative

The API key MUST be provided by the ASM both during registration and authentication operations. During registration operation, the API key is stored together with the newly generated key. During authentication operation, the API key is compared against the stored key by FIDO Authenticator. The FIDO Authenticator produces a signature only if the provided API key is valid.

This technique allows making sure that registered keys are only accessible by the caller who have originally registered them. A “dancing pig” 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).

See Authenticator Commands for more details.

4.3.4 TLS Binding

Various channel binding methods have been proposed (e.g. [RFC5929] and [ChannelID]).

UAF relies on TLS server authentication for binding authentication keys to AppIdentities. There are threats:

1. Attackers might fraudulently get a TLS server certificate for the same AppIdentity 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 do load-balancing for TLS endpoints using *different* TLS certificates. As a consequence, "tls-server-end-point" defined in [RFC5929] might be inappropriate.

Normative

1. If TLS Channel ID data is accessible to the FIDO Client, it SHALL be used by FIDO Client.
2. TLS ChannelID 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.

4.3.5 Personas

FIDO supports unlinkability of accounts at different relying parties by using relying party specific keys.

Sometimes users have multiple accounts at a particular relying party and even want to maintain unlinkability between these accounts.

Today, this is difficult and requires certain measures to be strictly applied.

FIDO does not want to add more complexity to maintaining unlinkability between accounts at one relying party.

In the case of Removable Authenticators, it is recommended to use different Authenticators for the various personas (e.g. “business”, “personal”). This is possible as Removable Authenticators typically are small and not excessively expensive.

In the case of Embedded Authenticator, this is different. FIDO supports 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 allows 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).

See Authenticator Commands for more details.

4.3.6 SessionID

TODO: Consensus: AI add explanation to Security Considerations sections that if FS wants to have integrity protection for SessionID it needs to bind SessionID to the challenge (e.g. by adding Challenge to it) and protects its integrity.

4.3.7 Authenticator Information retrieved from client vs. MetaData

TODO: in Security Considerations: talk about "hint" character of the Flags reported by FIDO Client (i.e. not security relevant, Server MUST look into Meta Data using AAID).

4.4 Interoperability Considerations

FIDO supports Web Applications, Mobile Applications and Native PC Applications. These environments require different bindings in order to achieve interoperability.

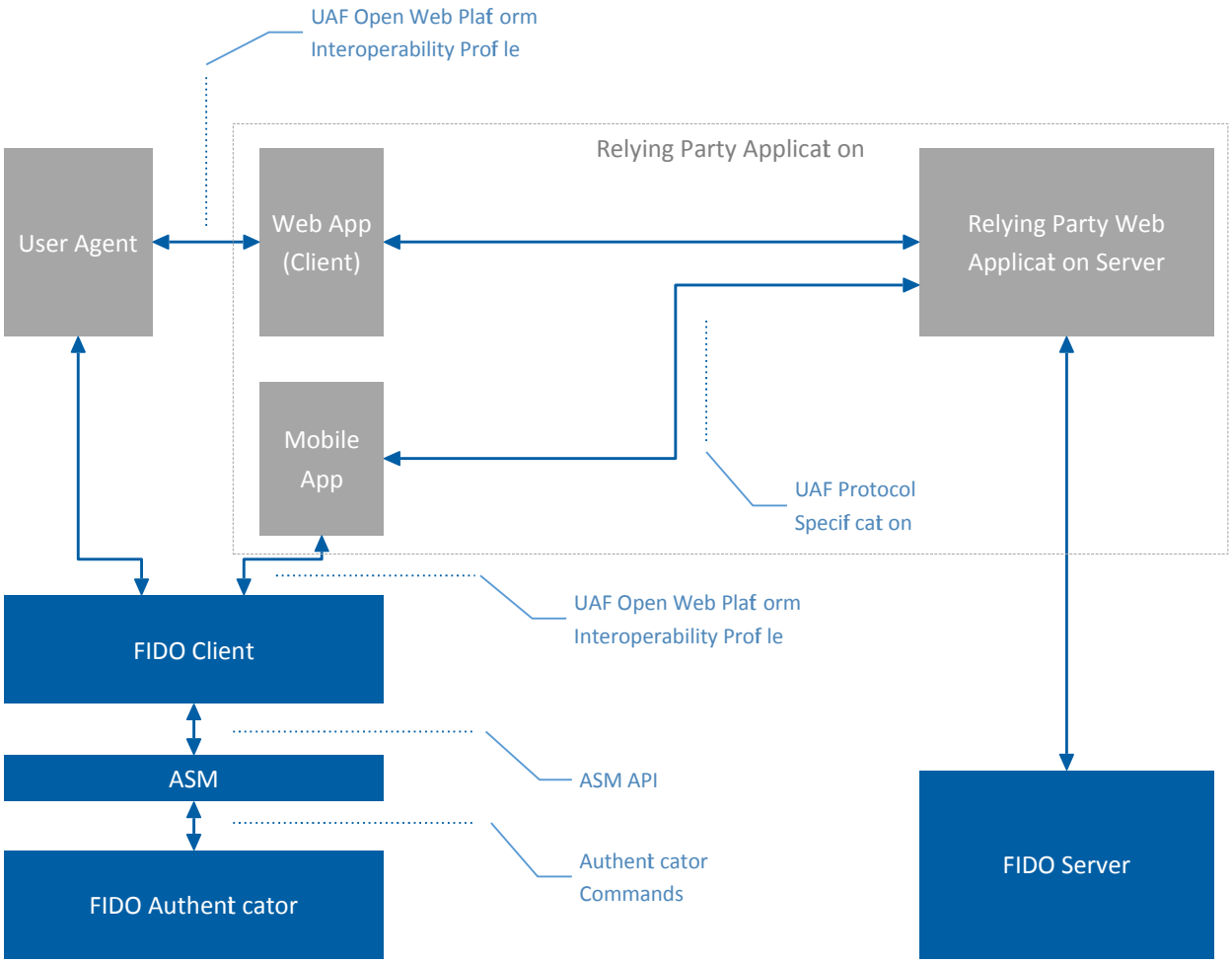

Figure 4.3: FIDO Interoperability Overview

Web applications typically consist of the web application server and the related Web App. The Web App code (e.g. HTML and JavaScript) is rendered and executed on the client side by the User Agent. The Web App code talks to the User Agent via a set of JavaScript APIs, e.g. HTML DOM. The FIDO ECMA Script binding is defined in UAF Open Web Platform Interoperability Profile. 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 pro-

prietary. 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 ”draft-hill-uaf-owp-iop-00”.

NOTES

- *The Objects KeyRegistrationData and SignData (see UAF Authenticator Commands) are generated and signed by FIDO Authenticators and have to be verified by the FIDO Server. Verification will fail if the values are modified.*
- *The ASM API specifies the standardized API to access Authenticator Specific Modules (ASMs) on Desktop PCs and Mobile Devices.*
- *The document Authenticator Commands 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.*

4.5 IANA Considerations

The following identifiers need to be registered with IANA:

<N/A>

5 UAF Supported Schemes

5.1 UAFV1-TLV

This scheme allows Authenticator and FIDO Server to exchange an asymmetric authentication key generated by Authenticator.

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

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

TAGs and Algorithms are defined in FIDO Registry.

Normative

[S-Auth-005] Conforming FIDO Servers MUST support all algorithm suites listed in document “FIDO Registry of Predefined Value”.

[A-Auth-002] Conforming Authenticators MUST support at least one algorithm listed in in document “FIDO Registry of Predefined Value”.

5.1.1 KeyRegistrationData

See UAF Authenticator Commands, section “Register Command”.

5.1.2 SignData

See UAF Authenticator Commands, section “Sign Command”.

6 Definitions

See FIDO Glossary.

Bibliography

[Coron99] Coron, J. and D. Naccache, “An accurate evaluation of Maurer's universal test”, LNCS 1556, February 1999. Download http://www.jscoron.fr/publications/universal.pdf.

[BioVocab] Harmonized Biometric Vocabulary. Text of Standing Document 2 (SD 2) Version 8, WD 2.8, work-in-progress, ISO/IEC JTC 1/SC 37: Biometrics, 2007-08-22. Download: http://isotc.iso.org/livelink/livelink/fetch/2000/2122/327993/327973/654118/6687752/N_3004_JTC_1_SC_37_-_Harmonized_Biometric_Vocabulary_-_for_information.pdf?nodeid=6719683&vernum=0

[Clickj] Clickjacking: Attacks and Defenses, Lin-Shung Huang and Collin Jackson Carnegie Mellon University; Alex Moshchuk, Helen J. Wang, and Stuart Schlechter Microsoft Research. Download https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf

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

[JWA] JSON Web Algorithms (JWA) draft-ietf-jose-json-web-algorithms-08

[JWK] JSON Web Key (JWK) draft-ietf-jose-json-web-key-11

[JWS] JSON Web Signature (JWS) draft-jose-json-web-signature-11

[JPSK] JSON Private and Symmetric Key, see draft-jones-jose-json-private-and-symmetric-key-00

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

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

[FIPS 186-3] NIST DIGITAL SIGNATURE STANDARD (DSS) (FIPS 186-3)

[TLS] The TLS Protocol Version 1.0 (RFC 2246), Version 1.1 (RFC 4346), Version 1.2 (RFC 5246)

[OCSP] Online Certificate Status Protocol (OCSP) (RFC 2560)

[PKCS#1] Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 (RFC 3447)

[RFC1341] MIME (Multipurpose Internet Mail Extensions) (RFC1341)

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

[RFC4086] Randomness Requirements for Security (RFC 4086)

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

[RFC5056] On the Use of Channel Bindings to Secure Channels (RFC 5056).

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

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

[RFC5639] Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation (RFC 5639)

[RFC5929] Channel Bindings for TLS (RFC 5929)

[OCRA] OCRA: OATH Challenge-Response Algorithm (RFC 6287)

[RFC6454] The Web Origin Concept (RFC 6454)

[ChannelID] Transport Layer Security (TLS) Channel IDs (draft-balfanz-tls-channel-id-00)

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

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

Appendix A UAF JSON Schema

```
// UAF Request Schema
{
    "id": "http://fidoalliance.org/UAF_v011#",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "description": "UAF v011 Request",
    "type": "object",
    "required": [ "V", "SeverOrigin", "Req" ],
    "properties": {
        "V": { "type": "string", "pattern": "^0.12$" },
        "SeverOrigin": { "type": "string" },
        "Req": {
            "type": "object",
            "oneOf": [
                { "$ref": "#/definitions/Query" },
                { "$ref": "#/definitions/Reg" },
                { "$ref": "#/definitions/Auth" },
                { "$ref": "#/definitions/Dereg" }
            ]
        }
    },
    "definitions": {
        "Query": {
            "type": "object",
            "properties": {
                "Op": { "type" : "string", "pattern": "^Query$" },
                "Permissions": { "$ref": "#/definitions/StringList" },
                "SupportedExtensions" : { "$ref": "#/definitions/StringList" },
                "Extensions" : { "$ref": "#/definitions/Extensions" }
            },
            "required": [ "Op", "Permissions" ],
            "additionalProperties": false
        },

        "Reg": {
            "type": "object",
            "properties": {
                "Op": { "type" : "string", "pattern": "^Reg$" },
                "Permissions": { "$ref": "#/definitions/StringList" },
                "Policy" : {
                    "type": "object",
                    "properties": {
                        "MatchPolicy": {"type": "string"},
                        "Accepted": {
                            "type": "array",
                            "minItems": 1,
                            "items": {"type": "string"}
                        },
                        "Registered": {
                            "type": "array",
```

```
                    "minItems": 1,
                    "items": {"type": "string"}
                }
            },
            "required": ["MatchPolicy", "Accepted"]
        },
        "UserID": { "type" : "string" },
        "AccountName": { "type" : "string" },
        "RegInfo": {
            "type": "object",
            "properties": {
                "Challenge": { "$ref": "#/definitions/ServerChallenge" },
                "PreferredSuite": {
                    "type": "object",
                    "properties": {
                        "Modes": { "$ref": "#/definitions/StringList" },
                        "AuthSuites": { "$ref": "#/definitions/StringList" }
                    }
                },
                "RegExtensions" : { "$ref": "#/definitions/Extensions" }
            },
            "required": ["Challenge"]
        },
        "SupportedExtensions" : { "$ref": "#/definitions/StringList" },
        "Extensions" : { "$ref": "#/definitions/Extensions" }
    },
    "required": [ "Op", "Policy", "UserID", "RegInfo" ],
    "additionalProperties": false
},

"Auth": {
    "type": "object",
    "properties": {
        "Op": { "type" : "string", "pattern": "^Auth$" },
        "Permissions": { "$ref": "#/definitions/StringList" },
        "Policy" : {
            "type": "object",
            "properties": {
                "MatchPolicy": {"type": "string"},
                "Accepted": {
                    "type": "array",
                    "minItems": 1,
                    "items": {
                        "type": "object",
                        "properties": {
                            "TAID": {
                                "type": "array",
                                "minItems": 1,
                                "items": {"type": "string"}
                            },
                            "TokenID": {
                                "type": "array",
```

```
                                "minItems": 1,
                                "items": {"type": "string"}
                            }
                        }
                    }
                }
            },
            "UserID": { "type" : "string" },
            "AuthInfo": {
                "type": "object",
                "properties": {
                    "Challenge": { "$ref": "#/definitions/ServerChallenge" },
                    "AuthenticatorOperation": {"type": "string"}
                },
                "required": ["Challenge"]
            },
            "Tokens" : {
                "type": "array",
                "minItems": 1,
                "items": {
                    "type": "object",
                    "properties": {
                        "TokenID": {"type": "string"},
                        "TokenData": {"type": "string"},
                        "TokenExtensions" : { "$ref": "#/definitions/Extensions" }
                    },
                    "required": ["TokenID"]
                }
            },
            "Transaction": {
                "type": "object",
                "properties": {
                    "TText": {"type": "string"},
                    "ServerData": {"type": "string"}
                },
                "required": ["TText"]
            },
            "SupportedExtensions" : { "$ref": "#/definitions/StringList" },
            "Extensions" : { "$ref": "#/definitions/Extensions" }
        },
        "required": [ "Op", "Policy", "UserID", "AuthInfo" ],
        "additionalProperties": false
    },

    "Dereg": {
        "type": "object",
        "properties": {
            "Op": { "type" : "string", "pattern": "^Dereg$" },
            "UserID": { "type" : "string" },
            "TokenID": { "type" : "string" },
            "SupportedExtensions" : { "$ref": "#/definitions/StringList" },
```

```
                "Extensions" : { "$ref": "#/definitions/Extensions" }
            },
            "required": [ "Op", "UserID", "TokenID" ],
            "additionalProperties": false
        },

        "Extensions": {
            "type": "array",
            "minItems": 0,
            "items": {
                "type": "object",
                "properties": {
                    "ID": {"type": "string"},
                    "Data": {"type": "string"}
                },
                "required": ["ID", "Data"]
        },
        "StringList": {
            "type": "array",
            "items": { "type": "string" }
        },
        "ServerChallenge": {
            "type": "object",
            "properties": {
                "Challenge": {"type": "string"},
                "ExpirationTime": {"type": "number"}
            },
            "required": ["Challenge", "ExpirationTime"]
        }
    }
}

// UAF Response Schema
{
    "id": "http://fidoalliance.org/UAF_v011#",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "description": "UAF v011 Response",
    "type": "object",
    "required": [ "V", "Op" ],
    "properties": {
        "V": { "type": "string", "pattern": "^0.11$" },
        "Resp": {
            "type": "object",
            "oneOf": [
                { "$ref": "#/definitions/Query" },
                { "$ref": "#/definitions/Reg" },
                { "$ref": "#/definitions/Auth" }
            ]
        }
    },
    "definitions": {
        "Query": {
```

```
        "properties": {
            "Op": { "type" : "string", "pattern": "^Query$" },
            "DeviceInfo" : { "$ref": "#/definitions/DeviceInfo" },
            "Extensions" : { "$ref": "#/definitions/Extensions" }
        },
        "required": [ "Op", "Permissions" ],
        "additionalProperties": false
    },

    "Reg": {
        "type": "object",
        "properties": {
            "Op": { "type" : "string", "pattern": "^Reg$" },
            "UserID": { "type" : "string" },
            "Tokens" : {
                "type": "array",
                "minItems": 1,
                "items": {
                    "type": "object",
                    "properties": {
                        "Info" : { "$ref": "#/definitions/TokenBaseInfo" },
                        "KeyRegistration": {
                            "type": "object",
                            "properties": {
                                "Mode": {"type": "string"},
                                "Scheme": {"type": "string"},
                                "KeyRegistrationData": {"type": "string"}
                            },
                            "required": ["Mode", "Scheme", "KeyRegistrationData"]
                        },
                        "TokenExtensions" : { "$ref": "#/definitions/Extensions" }
                    },
                    "required": ["Info", "KeyRegistration"]
                }
            },
            "DeviceInfo" : { "$ref": "#/definitions/DeviceInfo" },
            "Extensions" : { "$ref": "#/definitions/Extensions" }
        },
        "required": [ "Op" ],
        "additionalProperties": false
    },

    "Auth": {
        "type": "object",
        "properties": {
            "Op": { "type" : "string", "pattern": "^Auth$" },
            "UserID": {"type" : "string"},
            "Tokens" : {
                "type": "array",
                "minItems": 1,
                "items": {
                    "type": "object",
```

```
                    "properties": {
                        "TokenID": {"type": "string"},
                        "TokenAuthData": {
                            "type": "object",
                            "properties": {
                                "UAuthPub": {
                                    "type": "object",
                                    "properties": {
                                        "X509": { "type" : "string" },
                                        "JWK": { "type" : "string" }
                                    }
                                },
                                "AlgSuite": {"type": "string"},
                                "SignScheme": {"type": "string"},
                                "SignData": {"type": "string"}
                            },
                            "required": ["AlgSuite", "SignScheme", "SignData"]
                        },
                        "EncTokenData": {
                            "type": "object",
                            "properties": {
                                "Suite": {"type": "string"},
                                "ESK": {"type": "string"},
                                "Data": {"type": "string"}
                            },
                            "required": ["Suite", "ESK", "Data"]
                        },
                        "TokenExtensions" : { "$ref": "#/definitions/Extensions" }
                    },
                    "required": ["TokenID"]
                }
            },
            "Transaction": {
                "type": "object",
                "properties": {
                    "TText": {"type": "string"}
                },
                "required": ["TText"]
            },
            "DeviceInfo" : { "$ref": "#/definitions/DeviceInfo" },
            "Extensions" : { "$ref": "#/definitions/Extensions" }
        },
        "required": [ "Op", "Policy", "UserID", "AuthInfo" ],
        "additionalProperties": false
    },

    "TokenBaseInfo": {
        "type": "object",
        "properties": {
            "Vendor": { "type" : "string" },
            "VendorURL": { "type" : "string" },
            "TokenID": { "type" : "string" },
```

```
                "TAID": { "type" : "string" },
                "AuthFactor": { "type" : "string" },
                "Install": { "type" : "string" },
                "SecType": { "type" : "string" },
                "SecDisplay": { "type" : "boolean" }
            },
            "required": ["TokenID", "TAID", "AuthFactor", "Install"],
            "additionalProperties": false
        },

        "DeviceInfo": {
            "type": "object",
            "properties": {
                "UAFTokens": {
                    "type": "array",
                    "minItems": 0,
                    "items": { "type": { "$ref": "#/definitions/TokenBaseInfo" } }
                    }
                },
                "OEMInfo": {
                    "type": "object",
                    "properties": {
                        "MachineModel": {
                            "type": "object",
                            "properties": {
                                "OEM": { "type" : "string" },
                                "OS": { "type" : "string" }
                            },
                            "required": [ "OEM", "OS" ],
                            "additionalProperties": false
                        },
                        "UAFInfo": {
                            "type": "object",
                            "properties": {
                                "Vendor": { "type" : "string" },
                                "Version": { "type" : "string" }
                            },
                            "required": [ "Vendor", "Version" ],
                            "additionalProperties": false
                        }
                    },
                    "additionalProperties": true
                }
            }
        }
    }
}
```

UAF Authenticator-specific Module (ASM) API

specification set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

Editors:

Davit Baghdasaryan, Nok Nok Labs
John Kemp, FIDO Alliance

Contributors:

(member companies)

Abstract:

A FIDO authenticator may have a set of authenticator-specific software modules installed in the same environment where the authenticator is installed. This document describes the APIs which may be used by the FIDO client to access the functionality of these modules. The document's intended audience is FIDO Authenticator and FIDO Client vendors.

1. Overview

An *Authenticator Sub-Module* (ASM) is a platform-specific component developed by FIDO authenticator vendors which can be plugged into FIDO Clients and accessed with FIDO protocols.

In its typical form, an ASM is a driver-level component that is directly integrated with the authenticator driver. The ASM implements an API on top of the driver, exposing that API to the FIDO Client.

The intended audience are Authenticator and FIDO Client vendors.

There are two APIs provided by an authenticator sub-module:

- The *ASM Plugin API* is a platform-specific API which allows a platform-specific FIDO Client to enumerate ASMs and work with them.

- The ASM API is a cross-platform JSON API which allows FIDO Clients to communicate with ASMs and exchange FIDO-related data with them.

1.1 Notations

- All data structures and APIs in this document are presented in WebIDL format.

- All examples are provided in Javascript language.

2. Security Requirements

ASM developers must be very careful to protect FIDO data they are working with. There are several security requirements ASMs must meet:

- ASM module MUST implement a mechanism for isolating UAF credentials registered by two different FIDO Clients. One FIDO Client MUST not have access to UAF credentials that have been registered via a different FIDO Client.
  There are several ways to achieve this:
  - If ASM is bundled with a FIDO Client - this isolation mechanism is already built-in.
  - If ASM and FIDO Client are implemented by the same vendor - vendor MAY decide to implement proprietary mechanisms to bind its ASM only with its FIDO Client.
  - On some platforms ASM and FIDO Client may be assigned with a special privilege (or permission) which regular Apps don't have. ASMs built for such platforms MAY avoid supporting isolation of UAF credentials per FIDO Clients since all FIDO Clients will be considered trusted.
  - If FIDO Clients do not have a special privilege (or permission) ASMs MUST implement mechanisms to allow these FIDO Clients to access only UAF credentials that have been registered by themselves.

Rationale: The above requirement ensures that an App pretending to be a FIDO Client won't be able to exercise registered UAF credentials.

- ASM module designed for Internal Authenticators MUST associate an unpredictable identifier with UAF credentials that are registered via itself. This is a security measure that ensures that credentials registered by one ASM cannot be accessed by another ASM.
    - APIKey mechanism described in “UAF Authenticator Commands” document is such a mechanism.

Rationale: The above requirement ensures that an App pretending to be an ASM won’t be able to exercise UAF credentials.

- ASM MUST implement platform provided security best practices for protecting UAF related cached data.

- ASM MUST NOT cache any other UAF sensitive data than the bindings (e.g. FIDO Client IDs, APIKeys), PersonaIDs, KeyIDs, KeyHandles and AppIDs in their local cache. For example ASM MUST never cache Username in its local storage.

Rationale: The above requirement ensures that no sensitive information is stored outside the Authenticator’s security boundaries.

- ASM implementing support for a Silent IAuthnr MUST, during every registration, show a UI which explains what is a silent authenticator and get user’s consent for the registration. Also, such ASM MUST either
    - Run with a special permission/privilege on the system, or
    - Have a built-in binding with the authenticator which ensures that other applications cannot directly communicate with the authenticator by bypassing this ASM.

Rationale: The above requirement ensures that Apps cannot use Silent Authenticator for tracking purposes. User must be aware of each registration and give a consent.

A typical ASM cache would have the following form:

```
var storage = {
       registrations: [
              {
                     personaID: aksd823ndi..., // PersonaID
                     appID: "https://rp1.com",  // AppID
                     keyID: "qiwdm..." // KeyID
                     keyHandle: "aksjd..." // KeyHandle blob
              },
              ...
              ]
       }
```

2.1 Access Control for ASM APIs

The following table depicts access control for UAF ASM API.

Notations:

- NoAuth - No authentication required
- FCAuth - FIDO Client authentication
- UserVerify - explicit user verification by authenticator
- PubKeyHash - hash of public key to be deregistered
- KeyIDList - list of KeyIDs

Table 2-1-1 Access Control for ASM API

Commands	1stF IAuthnr	2ndF IAuthnr	1stF XAuthnr	2ndF XAuthnr
GetInfo	NoAuth	NoAuth	NoAuth	NoAuth
Register	UserVerify	UserVerify	UserVerify	UserVerify
Authenticate	UserVerify AppID FCAuth	UserVerify AppID KeyIDList (provided by FC) FCAuth	UserVerify AppID	UserVerify AppiD KeyIDList (provided by FC)
GetRegistrations	AppID FCAuth	AppID FCAuth	X	X
Deregister	AppID KeyID PubKeyHash FCAuth	AppID KeyID PubKeyHash FCAuth	AppID KeyID PubKeyHash	AppID KeyID PubKeyHash

3. ASM API

3.1. Process Function

All ASM functions are invoked using a generic *process* function with the following signature:

3.1.1. Interface definition

```
interface ASM {
      callback ProcessCallback = void(DOMString response);

      DOMString process(DOMString request);
      void processByCB(DOMString request, ProcessCallback cb);

      const short UAF_STATUS_OK = 0; // Success
      const short UAF_STATUS_ERROR = 1; // Undefined error
      const short UAF_STATUS_INVALID_ARG = 2; // Invalid argument
      const short UAF_STATUS_NO_USER_ENROLLED = 3; // No user enrolled
```

```
      const short UAF_STATUS_VERIFY_FAILED = 4; // User verification failed
      const short UAF_STATUS_ACCESS_DENIED = 5; // Access denied
      const short UAF_STATUS_NOT_REGISTERED = 6; // KeyID is not registered
      const short UAF_STATUS_USER_CANCELLED = 7; // UI cancelled by user
}
```

Depending on the underlying platform, the process function may take two forms.

The first form of the function takes a stringified JSON request and returns a stringified JSON response.
The second form allows a callback function to be passed, and calls the provided callback function when the response from the ASM is ready.

An example invocation of the process function is given below.

3.1.2. Example code

```
var req = {};
req.requestType = "Register”;
req.authenticatorReferenceID = "...”
...
ASM.ProcessByCB(JSON.stringify(req), function(response) {
    var resp = JSON.parse(resp);
    ...
});
```

3.2. GetInfo Request

A *GetInfo* request call returns information about available Authenticators and their sub-modules.

3.2.1. Interface definition

```
interface ASM {
      dictionary Authenticator {
        // Unique ID (unique per device’s scope) associated with Authnr by ASM.
            //   This   ID   is   generated   by   ASM   and   managed   by   it.
        // ASM needs this ID to reference authenticators internally
        long authenticatorReferenceID;

        // Supported UAF versions.
        // List of supported UAF Versions.
        // Each item is an array of two elements from which the first element
        // is the major version and the second is the minor version.
        // Example: [[0, 1], [2, 0]] - indicate two versions “0.1” and “2.0”
        short[][] uafVersions;

        // FIDO characteristics of Authnr
        DOMString aaid;
        long authenticationFactor;
```

```
        long keyProtection;
        long attachmentHint;
        long secureDisplay;
        long authenticationSuite;
        long scheme;
        long additionalInfo;

        // Indicates if the authenticator is 1stF
        boolean is1stF;

        // List of supported UAF extension IDs
        DOMString[] supportedExtensions;
      }

      dictionary GetInfoInput {
        // MUST be "GetInfo"
        DOMString requestType;
      }

      dictionary GetInfoOutput {
        // Mandatory. Status Code. Can be one of these:
        // UAF_STATUS_OK, UAF_STATUS_ERROR
        short statusCode;

        // List of supported ASM Message Versions.
        // Each item is an array of two elements from which the first element
        // is the major version and the second is the minor version.
        // Example: [[0, 1], [2, 0]] - indicate two versions "0.1" and "2.0"
        short[][] asmMessageVersions;

        // ASM Vendor
        DOMString vendor;

        // List of available Authenticators
        Authenticator[] authenticators;
      }
}
```

3.2.2. Example code

3.3. Register Request

Create a FIDO *KeyRegistrationData* and return to the caller.

Unless otherwise specified all steps are NORMATIVE for ASM implementation.

1. If its an IAuthnr
    a. Generate a new binding for this authenticator (e.g. APIKey) if not already generated

    b. Depending on platform (see Security Requirements) ASM MAY identify the FIDO Client and assign a binding ID to it (some form of FIDO Client ID)
    c. Obtain PersonaID from appropriate operational environment
2. Make sure there is a user enrolled with Authenticator. If not - take the user through enrollment. Otherwise locally verify the user
3. Hash the provided finalChallenge using Authnr specific hash function
4. Invoke Register CMD and provide all the necessary arguments
5. If it's an IAuthnr
    a. Cache AppID, KeyHandle, PersonaID, KeyID, and APIKey in ASM's cache
6. Return Authenticator's generated KeyRegistrationData (KRD)

3.3.1. Interface definition

```
interface ASM {

      dictionary RegistrationInput {
        // Mandatory. MUST be "Register"
        DOMString requestType;

        // Mandatory. Authenticator ID returned by GetInfo function
        long authenticatorReferenceID;

        // Mandatory. Application Identity
        DOMString appID;

        // Mandatory. Human-readable Username
        DOMString username;

        // Mandatory. base64url encoded opaque Challenge data
        DOMString finalChallenge;

        // Optional. Extensions
        Extension[] exts;
      }

      dictionary RegistrationOutput {
        // Mandatory. Status Code. Can be one of these:
        // UAF_STATUS_OK, UAF_STATUS_ERROR, UAF_INVALID_ARG,
        // UAF_STATUS_VERIFY_FAILED, UAF_STATUS_CANCELLED,
        // UAF_STATUS_USER_NOT_ENROLLED
        short statusCode;

        // Optional. base64url encoded Key Registration Data
        DOMString krd;

        // Optional. Extensions
        Extension[] exts;
```

```
        }
}
```

An example FIDO Client call to the *register* API is given below.

3.3.2. Example code

```
/* This code assumes that it's a 1stF IAuthnr and that ASM implements isolation of UAF
credentials of FIDO Clients. It's also assumed that ASM and Authenticator implement APIKey.
*/
function Register(authnrID, appID, username, finalChallenge) {
       try {
              // Identify the calling FIDO Client and obtain APIKey associated with it
              var fidoClientID = this.identifyCaller()
              if (this.apiKeyList.hasOwnProperty(fidoClientID) == null) {
                     this.apiKeyList[fidoClientID] = generateNewAPIKey()
                     currentAPIKey = this.apiKeyList[fidoClientID]
              }

              // Select the right authenticator by authnrID
              var authnr = GetAuthenticator(authnrID)

              // Check if any user is enrolled with Authenticator
              if (!authnr.Matcher.IsUserEnrolled()) {
                     // Enroll a new user with Authenticator
                     authToken = authnr.Matcher.EnrollUser()
              }
              else {
                     // Locally verify the user with Authenticator
                     authToken = authnr.Matcher.LocalVerifyUser()
              }

              // Hash the final challenge using a hash function that Authnr supports
              var finalChallengeHash = hash(authnr.supportedHashAlg, finalChallenge)

              // Obtain PersonaID from appropriate operational environment (e.g. OS)
              var personaID = environment.getPersonaID();

              // Invoke REGISTER command
              var tlv = authnr.Register(appID,
                                   username,
                                   finalChallengeHash,
                                   currentAPIKey,
                                   personaID,
                                   authToken)
              // Parse TLV response and extract KeyID, statusCode and KeyHandle
              ...

              if (tlv.errorCode == UAF_STATUS_OK) {
                     // Add a new registration to local storage
                     this.storage.registrations.push({
                                   appID: appID,
```

```
                                keyHandle: tlv.KeyHandle,
                                keyID: tlv.KeyID,
                                personaID: personaID,
                                apiKey: currentAPIKey })

                        return JSON.stringify({
                            statusCode: UAF_STATUS_OK,
                            krd: base64url_encode(TLV.UAFV1_RESPONSE)})
                }
                else {
                        // Determine the right error code and raise it
                        throw errorCode
                }
        }
        catch (err) {
                return JSON.stringify({ statusCode: err })
        }
}
```

3.4. Authenticate Request

Verify the user and return UAF SignData to the caller.

Unless otherwise specified all steps are NORMATIVE for ASM implementation.

1. If its an lAuthnr
    a. Obtain the binding ID specific to this authenticator (e.g. APIKey)
    b. Depending on platform (see Security Requirements) ASM MAY identify the FIDO Client and assign a binding ID to it (some form of FIDO Client ID)
    c. Obtain PersonaID from appropriate operational environment
2. Make sure that there is a registration corresponding to the given AppID and appropriate bindings (PersonaID, APIKey and FIDO Client ID, applies only to lAuthnrs).
3. Make sure that user is enrolled with Authenticator
4. Hash the provided finalChallenge using Authnr specific hash function
5. Invoke Sign CMD and provide all the necessary arguments
6. If it's a 1stF Authenticator and it returns list of Usernames, ASM must
    a. Show all Usernames to the user
    b. Ask the user to choose a single Username
    c. Instruct Authenticator to generate SignData for selected Username
7. Return FIDO formatted SignData

3.4.1. Interface definition

```
interface ASM {
```

```
      dictionary AuthenticateInput {
        // Mandatory. MUST be "Authenticate"
        DOMString requestType;

        // Mandatory. Returned by GetInfo function
        long authenticatorReferenceID;

        // Mandatory. Application Identity
        DOMString appID;

        // Optional. base64url encoded list of KeyIDs
        DOMString[] keyIDList;

        // Mandatory. base64url encoded opaque Challenge data
        DOMString finalChallenge;

        // Optional. base64url encoded Transaction Data. Optional.
        DOMString transactionData;

        // Optional. Extensions
        Extension[] exts;
      }

      dictionary AuthenticateOutput {
        // Mandatory. Status Code. Can be one of these:
        // UAF_STATUS_OK, UAF_STATUS_ERROR, UAF_INVALID_ARG,
        // UAF_AUTH_FAILURE, UAF_STATUS_CANCELLED,
        // UAF_STATUS_USER_NOT_ENROLLED, UAF_STATUS_NOT_REGISTERED
        short statusCode;

        // Optional. base64url encoded Key Registration Data
        DOMString signData;

        // Optional. Extensions
        Extension[] exts;
      }
}
```

3.4.2. Example code

```
/* This code assumes that it's a 1stF IAuthnr and that ASM implements isolation of UAF
credentials of FIDO Clients. It also assumes that ASM and Authenticator implement APIKey.
*/
function Authenticate(authnrID, appID, keyIDList, finalChallenge, transText) {
      try {
            // Identify the calling FIDO Client and obtain APIKey
            // associated with it
```

```
var fidoClientID = this.identifyCaller()
if (this.apiKeyList.hasOwnProperty(fidoClientID) == null) {
      throw UAF_STATUS_ACCESS_DENIED
}
var currentAPIKey = this.apiKeyList[fidoClientID];

// Select the right authenticator by authnrID
var authnr = GetAuthenticator(authnrID)

// Check if any user is enrolled with Authenticator
if (!authnr.Matcher.IsUserEnrolled()) {
      throw UAF_STATUS_NO_USER_ENROLLED
}

// Hash the final challenge using a hash function that Authnr supports
var finalChallengeHash = hash(authnr.supportedHashAlg, finalChallenge)

// Obtain PersonaID from appropriate operational environment (e.g. OS)
var personaID = environment.getPersonaID();

// Filter KeyHandles
khList = this._filterKeyHandles(appID, keyIDList, personaID, currentAPIKey);

// Invoke SIGN Command
var tlv = authnr.Sign(finalChallengeHash,
                  appID,
                  currentAPIKey,
                  personaID,
                  transText,
                  khList,
                  authToken);

// Parse TLV response and extract KeyID, errorCode and KeyHandle
...
if (tlv.StatusCode != UAF_STATUS_OK) {
      // Determine the right error code and raise it
      throw errorCode
}

// If Authnr returned list of Usernames -
// we need to show the list to user and ask to pick one
if (tlv.UsernameList != null) {
      // Show to user and ask them select a Username
      var finalKeyHandle = this._showToUser(tlv.UsernameList);

      // Invoke SIGN command again with the final KeyHandle
      tlv = authnr.Sign(finalChallengeHash,
                        appID,
                        currentAPIKey,
                        personaID,
                        transText,
                        finalKeyHandle,
                        authToken);
```

```
                    return JSON.stringify({
                        statusCode: UAF_STATUS_OK,
                        signData: base64url_encode(TLV.UAFV1_RESPONSE)}
                           )
             }

             return JSON.stringify({
                    statusCode: UAF_STATUS_OK,
                    signData: base64url_encode(TLV.UAFV1_RESPONSE)
                    })
      }
      catch (err) {
             return JSON.stringify({ statusCode: err })
      }
}
```

3.5. Deregister Function

Delete registered UAF data.

Unless otherwise specified all steps are NORMATIVE for ASM implementation.

1. If its an lAuthnr
    a. Obtain the binding ID specific to this authenticator (e.g. APIKey)
    b. Depending on platform (see Security Requirements) ASM MAY identify the FIDO Client and assign a binding ID to it (some form of FIDO Client ID)
    c. Obtain PersonaID from appropriate operational environment
    d. Delete the registration data associated with FIDO Client ID, APIKey, AppID, KeyID, PersonaID and publicKeyHash
2. If it's an XAuthnr
    a. Delete the registration data associated with AppID, KeyID and publicKeyHash

3.5.1. Interface definition

```
interface ASM {

      dictionary DeregisterInput {
        // Mandatory. MUST be "Deregister"
        DOMString requestType;

        // Mandatory. Returned by GetInfo function
        long authenticatorReferenceID;
```

```
        // Optional. Application Identity
        DOMString appID;

        // Mandatory. AAID
        DOMString aaid;

        // Mandatory. base64url encoded Transaction Data.
        DOMString keyID;

        // Mandatory. base64url encoded Transaction Data.
        DOMString publicKeyHash;

        // Optional. Extensions
        Extension[] exts;
      }

      dictionary DeregisterOutput {
        // Mandatory. Status Code. Can be one of these:
        // UAF_STATUS_OK, UAF_STATUS_ERROR, UAF_INVALID_ARG,
        // UAF_STATUS_VERIFY_FAILED, UAF_STATUS_ACCESS_DENIED
        short statusCode;

        // Optional. Extensions
        Extension[] exts;
      }
}
```

3.5.2. Example code

```
/* This code assumes that it's a 1stF IAuthnr and that ASM implements isolation of UAF
credentials of FIDO Clients. It also assumes that ASM and Authenticator implement APIKey.
*/
function Deregister(authnrID, appID, aaid, keyID, publicKeyHash) {
       try {
              // Identify the calling FIDO Client and obtain APIKey
              // associated with it
              var fidoClientID = this.identifyCaller()
              if (this.apiKeyList.hasOwnProperty(fidoClientID) == null) {
                     throw UAF_STATUS_ACCESS_DENIED
              }
              var currentAPIKey = this.apiKeyList[fidoClientID];

              // Obtain PersonaID from appropriate operational environment (e.g. OS)
              var personaID = environment.getPersonaID();

              // Select the right authenticator by authnrID
              var authnr = GetAuthenticator(authnrID)

              // Invoke SIGN Command
```

```
            var tlv = authnr.Deregister(appID,
                                keyID,
                                currentAPIKey,
                                personaID,
                                publicKeyHash);

            // Delete KeyHandle from local storage
            this.storage.deleteRegistration(authnrID, appID, keyID,
                                            currentAPIKey, personaID, publicKeyHash)

            return JSON.stringify({ statusCode: UAF_STATUS_OK })
      }
      catch (err) {
            return JSON.stringify({ statusCode: err })
      }
}
```

3.6. GetRegistrations Function

Return registrations corresponding to the given AppID.

Unless otherwise specified all steps are NORMATIVE for ASM implementation.

1. If it's an IAuthnr:
    a. Obtain the binding ID specific to this authenticator (e.g. APIKey)
    b. Depending on platform (see Security Requirements) ASM MAY identify the FIDO Client and assign a binding ID to it (some form of FIDO Client ID)
    c. Obtain PersonaID from appropriate operational environment
    d. Collect the list of registrations for given authenticatorReferenceID
    e. Filter this list with FIDO Client ID, APIKey, PersonaID and AppID.
2. If it's an XAuthnr:
    a. Create an empty list
3. Return the final list to caller

3.6.1. Interface Definition

```
interface ASM {

      dictionary GetRegistrationsInput {
        // Mandatory. MUST be "GetRegistrations"
        DOMString requestType;

        // Mandatory. Returned by GetInfo function
        long authenticatorReferenceID;
```

```
        // Mandatory. Application Identity
        DOMString appID;
      }

      dictionary Registration {
        // Mandatory. Application Identity
        DOMString appID;

        // Mandatory. List of base64url encoded KeyID
        DOMString[] keyIDs;
      }

      dictionary GetRegistrationsOutput {
        // Mandatory. Status Code. Can be one of these:
        // UAF_STATUS_OK, UAF_STATUS_ERROR, UAF_INVALID_ARG,
        // UAF_STATUS_VERIFY_FAILED, UAF_STATUS_NOT_REGISTERED,
        // UAF_STATUS_ACCESS_DENIED
        short statusCode;

        // Mandatory. List of registrations
        Registration[] regs;
      }
}
```

3.6.2. Example code

```
/* This code assumes that it's a 1stF IAuthnr and that ASM implements isolation of UAF
credentials of FIDO Clients. It also assumes that ASM and Authenticator implement APIKey.
*/
function GetRegistrations(authnrID, appID) {
       try {
              // Identify the calling FIDO Client and obtain APIKey
              // associated with it
              var fidoClientID = this.identifyCaller()
              if (this.apiKeyList.hasOwnProperty(fidoClientID) == null) {
                     throw UAF_STATUS_ACCESS_DENIED
              }
              var currentAPIKey = this.apiKeyList[fidoClientID];

              // Obtain PersonaID from appropriate operational environment (e.g. OS)
              var personaID = environment.getPersonaID();

              var regs = []
              // Iterate over registrations in the cache and match with provided AppID
              foreach(reg in this.storage.registrations) {
                     if (reg.appID != appID || personaID != reg.personaID ||
                            currentAPIKey != reg.apiKey)
                            continue;
```

```
                    if (!result.hasOwnProperty(reg.appID)) {
                          result[reg.appID] = []
                    }
                    result[reg.appID].push(reg.keyID)
              }
              return JSON.stringify({ statusCode: UAF_STATUS_OK, result: result })
       }
       catch (err) {
              return JSON.stringify({ statusCode: err })
       }
}
```

4. Plugin API

ASM Plugins are operating system platform-specific components which offer implementations of the ASM *process* interface described above, accepting and returning JSON-formatted values.

4.1. Android ASM Plugin API

On Android systems, an ASM Plugin may be implemented either as a Java library, statically linked with a FIDO Client application, or it may be implemented as a separate APK-packaged application. The Java library approach is not described in this document.

In order to be recognized by a FIDO Client, an ASM Plugin application must be designed as an Android Service, implementing the *IASMService* interface (http://developer.android.com/guide/components/aidl.html) described below .

```
/** ASM Service AIDL interface. This interface is used by FIDO Clients */

interface IASMService
{
    int process(in String req, out String resp);
}
```

The following code demonstrates how to register a concrete implementation of the *IASMService* interface to receive the Android-defined *IBinder::onBind* event within an ASM.

```
public class SampleASMService extends Service {

    @Override
    public IBinder onBind (Intent intent) {

        try {
             // Return an object of SampleASMServiceImpl which
             // implements IASMService AIDL
            return new SampleASMServiceImpl();
        }
        catch(Exception ex) {
```

```
        }

        return null;
    }
}
```

Additionally each ASM Plugin APK must include the following entry to its manifest file:

```
<service android:name=".exampleASMPluginName">
    <intent-filter>
        <action android:name=
            "com.fido.android.framework.FIDO_INTENT_ENUM_ASM" />
    </intent-filter>
</service>
```

The FIDO Client will find ASM packages installed on the system by looking for packages which have the *FIDO_INTENT_ENUM_ASM* intent registered.

The following code demonstrates how FIDO Clients can find ASMs and invoke the “process” function.

```
public class ASMBinder() {
      public void FindAndBind() {
            PackageManager pm = getApplicationContext().getPackageManager();
            List<ResolveInfo> asmList = pm.queryIntentServices(
                  new Intent("com.fido.android.framework.FIDO_INTENT_ASM_ENUM"),
                  PackageManager.GET_INTENT_FILTERS);

            Iterator<ResolveInfo> iter = asmList.iterator();
            while (iter.hasNext()) {
                  ResolveInfo info = iter.next();
                  ASMAgent agent = new ASMAgent();
                  agent.bind(info);
            }
      }
}

public class ASMAgent

    @Override
    public boolean bind() {
        ComponentName name = name();
        return mContext.bindService(
            new Intent().setClassName(name.getPackageName(), name.getClassName()),
                                      serConn,
                                      Context.BIND_AUTO_CREATE);
    }

    private ServiceConnection serConn = new ServiceConnection() {
        @Override
        public void onServiceConnected (ComponentName name, IBinder service) {
```

```
            IASMService asmService = IASMService.Stub.asInterface(service);
            if (asmService != null) {
                // Create an ASM JSON request and call Process function
                asmService.process(jsonRequest, jsonResponse);
            }
        }
        @Override
        public void onServiceDisconnected (ComponentName name) {
                    // ...
        }
    };
}
```

4.2. Windows ASM Plugin API

ASM Plugins are implemented in form of DLLs on Windows. Refer to ***asmplugin doxygen*** documentation for details on the API exposed by the DLL.

A Windows-based FIDO Client looks for ASM Plugin DLLs in the following registry paths:

HKCU\Software\FIDO\ASM

HKLM\Software\FIDO\ASM

The FIDO Client iterates over all keys under this path and looks for "string values" named "path":

```
[HK**\Software\FIDO\ASM\<exampleASMName>]
"path"="...\\<asmDLLName>.dll"
```

"path" must point to the exact filesystem location of ASM Plugin DLL.

Note that there may be FIDO Clients which can't access HKLM registry entries and therefore it's recommended to put ASM plugin entries in HKCU location.

References

Normative

FIDO UAF Authenticator Commands specification, Davit Baghdasaryan, Nok Nok Labs - https://docs.google.com/document/d/1HxZtBuKuloXdkjGw1N-ld44UMjAV7mqdsn9FhyBch7s

FIDO Technical Glossary, Baghdasaryan, Hill, Lindemann - https://docs.google.com/document/d/10qoZNA47QZSEiZ0YiG4HLSvHsBLDqsFIJUtZCAyLYYk

WebIDL, Cameron McCormack, Mozilla Corporation <cam@mcc.id.au> - http://dev.w3.org/2006/webapi/WebIDL/

Informative

FIDO Architecture Overview

UAF Authenticator Commands

specification set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

Glossary

Authnr	FIDO Authenticator
IAuthnr	Internal FIDO Authenticator
XAuthnr	External FIDO Authenticator
User Verification	Authenticators have different ways to locally verify a user. User Verification refers to the process of locally verifying a user by Authnr.
User Enrollment	User Enrollment refers to the process of creating association (also called a template) between user and Authenticator so that it can be further used for user verification.
Matcher	Matcher Component - a component which is able to locally verify a user (biometric matching, PIN verification, etc)
1stF Authnr	A FIDO authenticator which may act as first factor as well as second factor. By definition 1stF Authnrs must have a Matcher. An important characteristics of 1stF Authnrs is that they can be used in Login scenarios (without requiring user to enter username or password) where user is not known by the Application.
2ndF Authnr	A FIDO authenticator which acts only as second factor. 2ndF Authnrs always require a single KeyID to be provided before responding to Sign command. They might or might not have a Matcher.
Username	Username is a human readable string identifying user's account on RP site. It is provided by Relying Party as part of UAF Registration operation. All 1stF Authnrs MUST be able to store Username inside KeyHandle.
APIKey	APIKey acts as a guard for various commands. It's generated and provided by the ASM.
Uauth.pub/Uauth.priv	User authentication public/private key is generated during Register command by the Authenticator. Uauth.pub is shared with the web application while Uauth.priv is wrapped by the Authenticator. This key is specific to a particular account.
Wrap.sym	Internal symmetric wrapping key in the Authenticator used to wrap/unwrap the KeyHandle.
KeyHandle	A key container created by the Authenticator. KeyHandle is wrapped by Wrap.sym and contains the UAuth.priv key (and other data) inside.
RawKeyHandle	KeyHandle in un-wrapped form
Att.priv	Attestation Private Key as defined in UAF protocol
KRD	Key Registration Data as defined in UAF protocol.
SIGNDATA	Signing Data as defined in UAF protocol

UINT16	A 16 bit (2 bytes) unsigned integer
UINT32	A 32 bit (4 bytes) unsigned integer
UINT64	A 64 bit (8 bytes) unsigned integer
PersonaID	An identifier provided by ASM. PersonaID is used to associate different personas with registrations. It can be used to create virtual personas on an Authenticator e.g. for business and private accounts. PersonaID provides ways for users to manage their privacy settings.
Authentication Token/ AuthToken	AuthToken is a mechanism that allows the Authenticator to implement User Verification and Register/Sign functions in two distinct commands. Refer to example codes for more details.

1. Overview

This document specifies low-level functionality which UAF Authenticators should implement in order to support the UAF protocol. The document contains informative and normative notes. While implementing the set of commands specified in this document is not mandatory nevertheless implementors must follow all the normative notes.

The audience of this document is UAF Authenticator Vendors.

1.1 UAF Authenticator

The UAF Authenticator is an Authentication component that meets the UAF protocol requirements. The main functions to be provided by UAF Authenticators are:

1. [Mandatory] Verifying the User using verification technology built into the authenticator
2. [Mandatory] Performing cryptographic operations defined in UAF protocol
3. [Mandatory] Attesting itself to the UAF Server if there is a built-in support for attestation
4. [Optional] Securely displaying the transaction text to the User

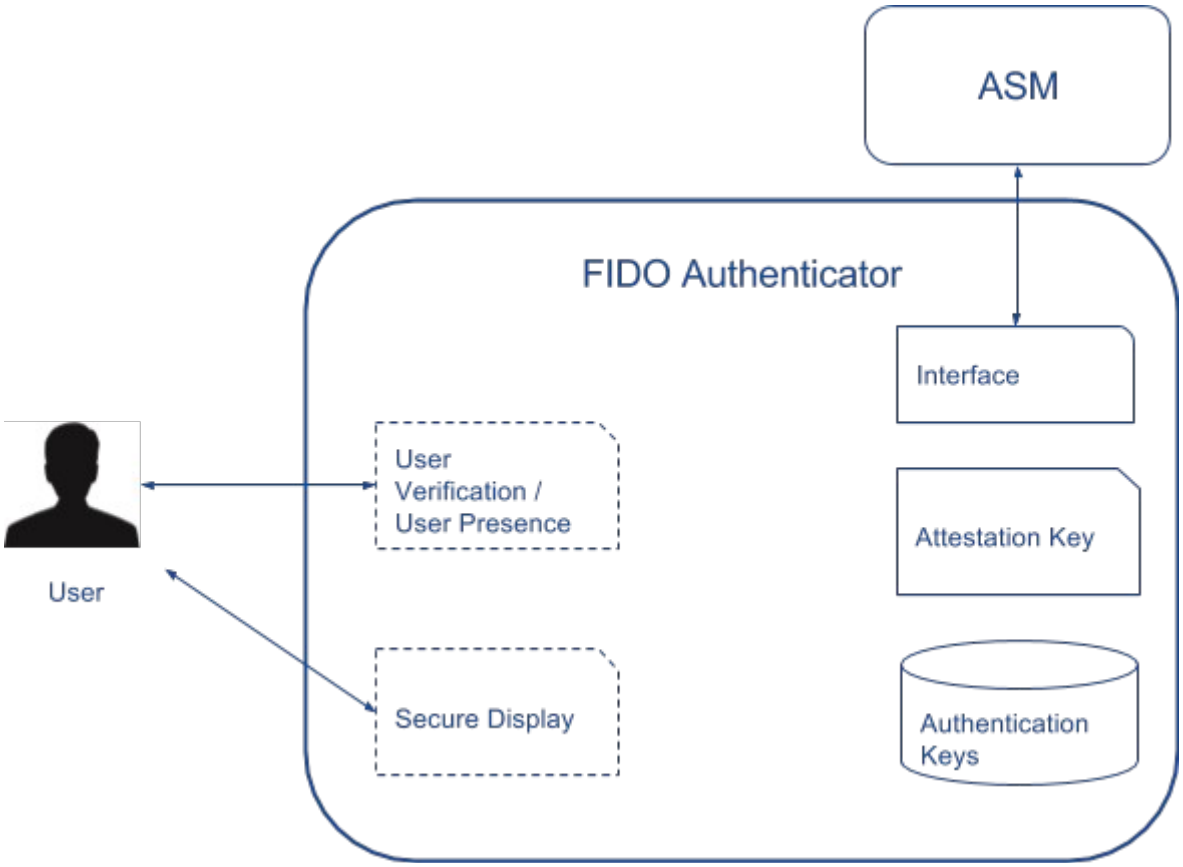

Figure 1: FIDO Authenticator Logical Sub-Components

Some examples of UAF Authenticators:

- A fingerprint sensor built into a mobile device
- PIN authentication implemented inside a secure element
- A mobile phone acting as an authenticator to a different machine
- A USB token with built-in authentication capability

1.2 Types of Authenticators

There are three types of authenticators defined in this document.

- 1stF IAuthnr - Internal Authenticator with 1st Factor capability
    - It is assumed that these authenticators MAY or MAY NOT store RawKeyHandles in their own internal storage. If they don't - they return KeyHandle to ASM.

- 1stF XAuthnr - External Authenticator with 1st Factor capability
    - It is assumed that these authenticators are designed to store RawKeyHandles in their own internal secure storage and don't provide these to ASM.

    - Note that in some deployments IAuthnrs can act as XAuthnr. When it happens this Authenticator MUST follow requirements of Internal Authenticators within the boundary of the system it is internally connected to and follow requirements of External Authenticator to the system it connects externally.

- 2ndF Authnr - Internal or External Authenticator with 2nd Factor capability
    - It is assumed that these authenticators MAY or MAY NOT store RawKeyHandles in their own internal storage. If they don't - they return KeyHandle to ASM.
    - These authenticators can only act as 2ndF

Throughout the document there will be special conditions applying to some of these types of authenticators.

1.3 Notations

- All data described in this document MUST be encoded in **big-endian** format.

- The following is an example of a buffer presented in form of Table. It reads as:
    - A variable length buffer which has a 2 bytes Tag in the beginning with value of TAG_CUSTOM_KHANDLE_LIST.
    - Tag follows by 2 bytes of overall Length
    - What follows is a list of KeyHandles. The sum of all KeyHandles equals to the Length.
        - Length = N * length(UINT16) + N * KeyHandleSize where N is the number of KeyHandles

1.	UINT16	Tag	TAG_CUSTOM_KHANDLE_LIST
1.1	UINT16	Length	Entire Length of list of KeyHandles
1.2	UINT16	KeyHandleSize	Each entry is a KeyHandle with length and content. This is the length part.
1.3	BYTE	KeyHandle[KeyHandleSize]	Each entry is a KeyHandle with length and content. This is the content part.

- Details of commands is described using a pseudo-code based on Javascript syntax.

2. Access Control for Commands

The following table summarizes access control requirements for each command.

Table 2-1 Access Control for Commands

Commands	1stF IAuthnr	2ndF IAuthnr	1stF XAuthnr	2ndF XAuthnr
GetInfo	NoAuth	NoAuth	NoAuth	NoAuth

Register	UserVerify	UserVerify	UserVerify	UserVerify
Sign	UserVerify PersonaID AppID KeyHandleList (provided by ASM) APIKey	UserVerify PersonaID AppID KeyHandleList (provided by ASM) APIKey	UserVerify AppID	UserVerify AppID KeyHandleList (provided by ASM)
Deregister	PersonaID AppID APIKey KeyID PubKeyHash (the command should never reveal any information whether key was registered or not)	PersonaID AppID APIKey KeyID PubKeyHash (the command should never reveal any information whether key was registered or not)	AppID KeyID PubKeyHash (the command should never reveal any information whether key was registered or not)	AppID KeyID PubKeyHash (the command should never reveal any information whether key was registered or not)

Normative Note:
All UAF Authenticators MUST support the access control mechanism defined above.

If Authenticator vendors want to add additional security mechanisms - they MAY do it.

2.1 API Key Concept

From Authenticator perspective APIKey guards the access to KeyHandles from unauthorized ASMs. APIKey uses Trust On First Use (TOFU) trust model to establish a link between ASM, which instructs Authenticator to create a KeyHandle, and the KeyHandle itself.

The concept requires the following steps to be performed by ASM and Authenticator:

- During Registration command
  - ASM generates a random APIKey and provides to Authenticator
  - Authenticator puts the APIKey into RawKeyHandle, wraps it and returns
  - ASM stores APIKey in its own storage and makes sure to protect it from other Apps on the system

- During Sign command
  - ASM provides the APIKey to Authenticator along with other arguments
  - Authenticator unwraps provided KeyHandle(s) and proceeds with Sign command only if RawKeyHandle.APIKey equals to provided APIKey

Normative Notes:

IAuthnrs MUST support a mechanism for binding generated KeyHandles with ASMs. The mechanism MUST have at least the same security characteristics as APIKeys described above.

It is RECOMMENDED that XAuthnrs ignore APIKeys since it would not allow using these authenticators on multiple machines.

3. Types and Tags

Informative Notes:

In this document UAF Authenticators use TLV (Type Length Value) format to communicate with outside world. All requests and response packets are encoded via TLV.
While it is not mandatory for implementors to follow this formatting it is recommended to do so. Future versions of UAF specifications may make this mandatory.

Commands and existing predefined TAGs can be extended with appending other TAGs (custom or predefined).

Refer to "FIDO Registry of Pre-defined Values" document for information about predefined TAGs.

TAG values defined in this section are custom and non-normative.

TLV formatted data has the following simple structure:

2 bytes	2 bytes	"Length" bytes
Tag	Length	Data

Table 3-1 Custom non-normative TAGs used in this document only (0x0100 - 0x01FF)

Name	Value	Description
TAG_CUSTOM_KEYHANDLE	0x1001	Custom tag for representing a KeyHandle
TAG_CUSTOM_KHANDLE_LIST	0x1002	Custom tag for representing a list of KeyHandles.
TAG_CUSTOM_UNAME_LIST	0x1003	Custom tag for representing a list of Usernames.
TAG_CUSTOM_AUTHTOKEN	0x1004	Custom tag for representing an Authentication Token

Table 3-2 UAF Predefined Authenticator Command TAGs (0x1000 - 0x10FF)

Name	Value	Description

TAG_UAFV1_GETINFO_CMD	0x2001	Tag for GetInfo command.
TAG_UAFV1_GETINFO_CMD_RESP	0x2101	Tag for GetInfo command response.
TAG_UAFV1_REG_CMD	0x2002	Tag for Register command.
TAG_UAFV1_REG_CMD_RESP	0x2102	Tag for Register command response.
TAG_UAFV1_SIGN_CMD	0x2003	Tag for Sign command.
TAG_UAFV1_SIGN_CMD_RESP	0x2103	Tag for Sign command response.
TAG_UAFV1_DEREG_CMD	0x2004	Tag for Deregister command.
TAG_UAFV1_DEREG_CMD_RESP	0x2104	Tag for Deregister command response.

Table 3-3 UAF Authenticator Error Codes

Name	Value	Description
UAF_STATUS_OK	0x0000	Indicates success
UAF_STATUS_ERR_UNKNOWN	0x0001	Indicates a generic error
UAF_STATUS_ERR_ACCESS_DENIED	0x0002	Indicates that access to command is denied
UAF_STATUS_ERR_INVALID_KEYHANDLE	0x0003	Indicates that the provided KeyHandle is invalid
UAF_STATUS_ERR_INVALID_PARAM	0x0004	Indicates that one of the input params is not valid
UAF_STATUS_ERR_USER_CANCELLED	0x0005	User cancelled the operation
UAF_STATUS_ERR_UNSUPPORTED_CMD	0x0006	Unsupported command

4. Structures

4.1 RawKeyHandle

RawKeyHandle is a structure generated and parsed by Authenticator. Authenticators may define RawKeyHandle in different ways and its internal structure is relevant only to Authenticator implementation. This section provides a sample reference definition for it.

Table 4-1 RawKeyHandle Structure

Hash of	Size depends on	Hash of uauth.pub.	32 bytes	32 bytes	max 128 bytes

AppID	Algorithm used	Size depend on Algorithm used			(mandatory for 1stF Authnrs)
AppIDHash	Uauth.priv	UauthPubHash	APIKey	PersonaI D	Username

Normative Note:
1stF Authnrs MUST store Username and 2ndF Authnrs MUST NOT store it. Ability to support Username is a key difference between 1stF and 2ndF Authenticators. Refer to "Sign" command section for more details on this difference.

It is RECOMMENDED that XAuthnrs do not store APIKey and PersonaID since otherwise users won't be able to use them on different machines. However if Authenticator vendor decides to do that to address a specific use case - they MAY do it.

RawKeyHandle MUST be wrapped before leaving Authenticator boundary since it contains the user authentication private key (uauth.priv).

Supporting PersonaID is optional for all types of authenticators. However an authenticator designed for multi-user systems will likely have to support.

5. Commands

All UAF Authenticator commands and responses are semantically similar - they are all represented as TLV encoded blobs. The first 2 bytes of each command is the command code. After receiving a command - Authenticator MUST parse the first TAG and figure out what command is being issued.

Informative Note:
Supporting exactly the same semantics of commands is not a requirement but is recommended. This applies to all commands described in this section.
UAF v1.0 doesn't attempt to standardize Authenticator Commands layer however next versions may standardize it and therefore it is recommended to follow the structures and notations used in this specification.

5.1 GetInfo Command

General Description

This command returns a subset of Authenticator's UAF metadata.

Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_GETINFO_CMD
1.1	UINT16	Length	Entire Command Length - must be 0 for this command

Command Response

Informative Note:
Refer to "FIDO Registry of Pre-defined Values" document for bitflag definitions of AuthFactor, KeyProtection, SecureDisplay and AuthSuite.

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_GETINFO_CMD_RESP
1.1	UINT16	Length	TLV Length
1.2	UINT16	APIVersion	Specifies API version. MAJOR (1 byte) and MINOR (1 byte). MUST be 0x0001
1.2	BYTE	NumUAFVersion	Number of supported UAF Versions
1.3	UINT16	UAFVersion[NumUAFVersion]	List of supported UAF Protocol Versions. Each item is encoded as UINT16: MAJOR (1 byte) and MINOR (1 byte).
1.4	UINT32	AAID	Vendor assigned AAID
1.5	UINT64	AuthFactor	Authentication Factor (as defined in *"FIDO Registry of Pre-defined Values"* document)
1.6	UINT64	KeyProtection	Key Protection type (as defined in *"FIDO Registry of Pre-defined Values"* document)
1.7	UINT64	SecureDisplay	Secure Display type (as defined in *"FIDO Registry of Pre-defined Values"* document)
1.8	UINT64	AuthSuite	Authentication Suite (as defined in *"FIDO Registry of Pre-defined Values"* document)

Non-Normative Example Code

```
function GetInfo()
{
	var response = []

	write16(response, TAG_UAFV1_GETINFO_CMD);
	write16(response, GETINFO_LENGTH);
	write16(response, NUM_UAF_VERSIONS);
	for (i = 0; i < NUM_UAF_VERSIONS; ++i)
		write16(response, supportedUAFVersions[i]);
	write32(response, MY_AAID);
	...

	return response;
}
```

5.2 Register Command

General Description

This command generates a UAF registration assertion. This assertion can be used to register the Authenticator with a UAF server.

Authenticator MUST:

1. If User is already locally enrolled (such as biometric enrollment, PIN setup, etc.) - locally verify the user.
    a. If verification fails - fail the command
2. If User is not locally enrolled with Authenticator – take the User through enrollment process.
    a. If enrollment fails - fail the command
3. Generate a new User Authentication Key (Uauth.pub/Uauth.priv)
4. Create TAG_UAFV1_KRD structure (see below table) and attest it with Attestation Private Key (Att.priv)
5. Create a RawKeyHandle
6. If it's an IAuthnr
    a. Add APIKey and PersonaID into RawKeyHandle
7. Wrap RawKeyHandle with Wrap.sym key
8. Return the command response

Error Codes:

- UAF_STATUS_ERR_ACCESS_DENIED
- UAF_STATUS_ERR_INVALID_PARAM
- UAF_STATUS_ERR_UNKNOWN

Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_REG_CMD
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	ChallengeSize	Final Challenge size - max 32 bytes
1.3	BYTE	FinalChallenge[ChallengeSize]	Final Challenge provided by ASM
1.4	UINT16	AppIDSize	AppID size - (max 256 bytes)
1.5	BYTE	AppID[AppIDSize]	AppID provided by ASM
1.6	UINT16	UsernameSize	Username size - max 128 bytes
1.7	BYTE	Username[UsernameSize]	Username provided by ASM
1.8	UINT16	APIKeySize	APIKey size - max 32 bytes
1.9	BYTE	APIKey[APIKeySize]	APIKey provided by ASM
1.10	UINT16	PersonaIDSize	PersonaID size - max 32 bytes
1.11	BYTE	PersonaID[PersonaIDSize]	PersonaID provided by ASM
1.12	UINT16	Tag	TAG_CUSTOM_AUTHTOKEN (optional)
1.12.1	UINT16	Length	Entire Length of Authentication Token
1.12.2	UINT32	AuthToken	Authentication Token. Used only by Authenticators which have a Matcher.

Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_REG_CMD_RESP
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	StatusCode	Error Code returned by Authnr
1.3	UINT16	Tag	TAG_UAFV1_RESPONSE (optional)
1.3.1	UINT16	Length	Entire Length of TAG_UAFV1_RESPONSE structure
1.3.2	UINT16	Tag	TAG_UAFV1_KRD

1.3.2.1	UINT16	Length	Entire Length of TAG_UAFV1_KRD structure
1.3.2.2	UINT16	AAIDSize	AAID size
1.3.2.3	BYTE	AAID[AAIDSize]	Authenticator AAID
1.3.2.4	UINT16	SignatureAlgAndEncoding	Signature Algorithm and Encoding. Refer to *FIDO Registry of Predefined Values* document for information on supported algorithms and their values.
1.3.2.5	UINT16	ChallengeSize	Final Challenge size
1.3.2.6	BYTE	FinalChallenge[ChallengeSize]	Final Challenge provided in the Command
1.3.2.7	UINT16	KeyIDSize	KeyID size
1.3.2.8	BYTE	KeyID[KeyIDSize]	KeyID generated by Authenticator
1.3.2.9	BYTE	AuthenticatorVersion	Vendor assigned authenticator version
1.3.2.10	UINT32	RegCounter	Registration Counter. Indicates how many times this Authenticator has performed registrations in the past.
1.3.2.11	UINT32	SignCounter	Sign Counter. Indicates how many times this Authenticator has performed signatures with any user authentication key in the past.
1.3.2.12	UINT16	PublicKeyAlgAndEnc	Public Key algorithm and encoding. Refer to *FIDO Registry of Predefined Values* document for information on supported algorithms and their values.
1.3.2.13	UINT16	PublicKeySize	Size of uauth.pub
1.3.2.14	BYTE	PublicKey[PublicKeySize]	User authentication public key (uauth.pub) newly generated by Authenticator
1.3.3	UINT16	SignatureSize	Signature size
1.3.4	BYTE	Signature[SignatureSize]	Signature calculated with Att.priv over TAG_UAFV1_KRD content. Note that entire TAG_UAFV1_KRD content, including the tag and it's length field, MUST be included during signature computation.
1.3.5	UINT16	Tag	TAG_ATTESTATION_CERT
1.3.5.1	UINT16	Length	Entire Length of Attestation Cert

1.3.5.2	BYTE	Certificate[Length]	Attestation Certificate bytestream
1.3.6	UINT16	Tag	TAG_CUSTOM_KEYHANDLE (optional)
1.3.6.1	UINT16	Length	Entire Length of KeyHandle
1.3.6.2	BYTE	KeyHandle[Length]	KeyHandle

Normative Note:
Independent on whether Authenticator supports the exact semantics of this command or not, the content of TAG_UAFV1_RESPONSE MUST precisely correspond to the structure defined above since it will be parsed and verified by the UAF Server.

For 2ndF Authnr KeyID MAY be the KeyHandle. This is useful for situations where KeyHandle must be stored on UAF Server.

For Silent Authenticators KeyHandle MUST never be stored on UAF Server otherwise this would enable tracking users without providing ability to users to clear KeyHandles from local device.

KeyID MUST be a unique and unguessable 32 bytes bytestream. The uniqueness MUST be within the scope of AAID.

If an Authenticator is not able to protect an attestation private key - it's RECOMMENDED to not support attestation at all. If an Authenticator doesn't support attestation - final TAG_UAFV1_KRD object MUST be signed with newly generated Uauth.priv key. In addition the content of TAG_ATTESTATION_CERT MUST have 0 length.

If Authenticator doesn't support Sign Counter or Reg Counter it MUST set these to 0 in TAG_UAFV1_KRD.

Non-Normative Example Code

```
/* It is assumed that prior to calling this function ASM also called a function
   which verifies the user locally using authenticator's matcher. After successful verification
a
   special Authentication Token (authToken) is stored in cache of authenticator and returned
   to the caller. The caller must provide authToken back to this command.
   This code assumes that it's a 1stF IAuthnr and it doesn't store RawKeyHandles internally.
*/
function Register(finalChallenge, appID, username, apiKey, personaID, authToken)
{
       // Compare authToken with the one in cache and make sure it's valid
       if (!isValidAuthToken(authToken)) {
              return UAF_STATUS_ERR_ACCESS_DENIED;
       }

       try {
              // Generate a new Uauth key pair
              var uauth = genKeyPair();
```

```
            // Construct a RawKeyHandle and wrap with Wrap.sym key
            var kh = createKeyHandle(appID, apiKey, personaID, username, uauth.priv,
uauth.pub);

            // Use SHA256 hash of KeyHandle as KeyID. This KeyID derivation
            // mechanism is not mandatory.
            var keyID = SHA256(kh);

            // Create UAF Registration to-be-signed object
            var tbs = prepareTLVRegistrationTBS(aaid, finalChallenge, keyID, uauth.pub);

            // Calculate attestation signature
            var signature = sign(attestation.priv, tbs);

            // Prepare TLV response buffer and return
            return prepareTLVWithKRD(UAF_STATUS_OK, tbs, signature, kh);
      }
      catch (err) {
            // Prepare TLV response buffer, put error code and return
            return prepareTLVWithError(err);
      }
}
```

5.3 Sign Command

General Description

This command generates a UAF assertion. This assertion can be further verified by a UAF server which has a prior registration with this Authenticator.

Informative Note:

1stF Authenticators MUST implement this command in two stages.

1. *The first stage will be executed only if Authenticator finds out that there are more than one keyHandles after filtering with AppID, APIKey and PersonaID. In this stage Authenticator must return a list of usernames along with corresponding keyHandles*
2. *In the second stage, after user selects a username, this command will be called with a single keyHandle and will return a UAF assertion based on this keyHandle*

2ndF Authenticators do not need to support the first stage.

Authenticator MUST:

1. Locally verify the User
    a. If verification fails - fail the command

2. Filter KeyHandles with provided AppID, APIKey and PersonaID
    a. If number of filtered KeyHandles is 0 - fail
3. If number of filtered KeyHandles is 1:
    a. If TransactionText is not empty:
        - If this is a Silent Authenticator - fail
        - If Authenticator doesn't have a built-in Secure Display - fail
        - Show TransactionText and AppID on Authenticator's Secure Display and wait for the user to confirm it
            - Fail if user cancels transaction
        - Compute hash of Transaction Text Hash (TTHash)
    b. Create a UAFV1_SIGNDATA package
    c. Sign the UAFV1_SIGNDATA with uauth.priv
    d. Create a UAFV1_SIGN_RESPONSE and return to caller
4. If number of filtered KeyHandles is greater than 1:
    a. If it's not a 1stF Authnr - fail
    b. Return list of usernames (TAG_CUSTOM_UNAME_LIST) from filtered KeyHandles

Error Codes:

- UAF_STATUS_ERR_ACCESS_DENIED
- UAF_STATUS_ERR_USER_CANCELLED

Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_SIGN_CMD
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	ChallengeSize	Final Challenge size (max 32 bytes)
1.3	BYTE	FinalChallenge[ChallengeSize]	Final Challenge provided by ASM
1.4	UINT16	AppIDSize	AppID size (max 256 bytes)
1.5	BYTE	AppID[AppIDSize]	AppIdentity provided by ASM
1.6	UINT16	APIKeySize	APIKey size (max 32 bytes)
1.7	BYTE	APIKey[APIKeySize]	APIKey provided by ASM
1.8	UINT16	PersonaIDSize	PersonaID size - max 32 bytes
1.9	BYTE	PersonaID[PersonaIDSize]	PersonaID provided by ASM

1.10	UINT16	TTSize	Transaction Text Size - max 1024 bytes
1.11	BYTE	TransactionText[TTSize]	Transaction Text provided by ASM
1.12	UINT16	Tag	TAG_CUSTOM_KHANDLE_LIST (optional)
1.12.1	UINT16	Length	Entire Length of list of KeyHandles. This TAG contains multiple (>=1) KeyHandle entries. Each entry has a size and content. Length = Sum(KeyHandleSizes) + NumberOfKeyHandles * sizeof(UINT16)
1.12.2	UINT16	KeyHandleSize	Size of KeyHandle
1.12.3	BYTE	KeyHandle[KeyHandleSize]	KeyHandle
1.13	UINT16	Tag	TAG_CUSTOM_AUTHTOKEN (optional)
1.13.1	UINT16	Length	Entire Length of UAFV1_RESPONSE structure
1.13.2	UINT32	AuthToken	Authentication Token. Used only by Authenticators which have a Matcher.

Informative Note:

Authenticators which do not expose KeyHandles should expect KeyIDs to be provided to Sign command within TAG_CUSTOM_KHANDLE_LIST tag.

Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_SIGN_CMD_RESP
1.1	UINT16	Length	Entire Length of Command Response
1.2	UINT16	StatusCode	StatusCode returned by Authenticator
1.3	UINT16	Tag	TAG_CUSTOM_UNAME_LIST (optional)
1.3.1	UINT16	Length	Entire Length of list of Usernames. This TAG contains multiple (>=1) Username entries. Each entry has a size and content.

			Length = Sum(UsernameSizes) + Sum(KeyHandleSizes) + NumberOfUsernames * sizeof(UINT16) * 2
1.3.2	UINT16	UsernameSize	Size of Username
1.3.3	BYTE	Username[UsernameSize]	Username
1.3.4	UINT16	KeyHandleSize	Size of KeyHandle
1.3.5	BYTE	KeyHandle[KeyHandleSize]	KeyHandle
1.4	UINT16	Tag	TAG_UAFV1_SIGN_RESPONSE (optional)
1.4.1	UINT16	Length	Entire Length of UAFV1_SIGN_RESPONSE structure.
1.4.2	UINT16	Tag	TAG_UAFV1_SIGNDATA
1.4.2.1	UINT16	Length	Entire Length of UAFV1_SIGNDATA structure.
1.4.2.2	BYTE	AuthenticationMode	Authentication Mode indicating whether user explicitly verified or not and indicating if there is a transaction text or not. ● 1 means that user has been explicitly verified ● 2 means that transaction text has been shown on secure display and user confirmed it by explicitly verifying with authenticator
1.4.2.3	UINT16	SignatureAlgAndEncoding	Signature algorithm and encoding scheme. Refer to *FIDO Registry of Predefined Values* document for information on supported algorithms and their values.
1.4.2.4	UINT16	AuthnrNonceSize	Authenticator Nonce size - MUST be at least 8 bytes
1.4.2.5	BYTE	AuthnrNonce[NonceSize]	A nonce randomly generated by Authenticator
1.4.2.6	UINT16	ChallengeSize	Final Challenge size
1.4.2.7	BYTE	FinalChallenge[ChallengeSize]	Final Challenge provided in the Command
1.4.2.8	UINT16	TTHashSize	Transaction Hash size
1.4.2.9	BYTE	TTHash[TTHashSize]	Transaction Hash
1.4.2.10	BYTE	AuthenticatorVersion	Vendor assigned authenticator version.
1.4.2.11	UINT32	SignCounter	Sign Counter.

			Indicates how many times this Authenticator has performed signatures with any user authentication keys in the past.
1.4.3	UINT16	SignatureSize	Signature size
1.4.4	BYTE	Signature[SignatureSize]	Signature calculated using Uauth.priv over UAFV1_SIGNDATA structure. Note that entire UAFV1_SIGNDATA content, including the tag and it's length field, MUST be included during signature computation.

Normative Note:
Independent on whether Authenticator supports the exact semantics of this command or not, the content of TAG_UAFV1_SIGN_RESPONSE MUST precisely correspond to the structure defined above since it will be verified by the FIDO Server.

Silent Authenticators MUST only behave as a 2ndF Authnrs.

If Authenticator doesn't support Sign Counter - it MUST set it to 0 in TAG_UAFV1_SIGNDATA.

Non-Normative Example Code

```
/* It is assumed that prior to calling this function ASM also called a function
   which verifies the user locally using authenticator's matcher. After successful verification
a
   special Authentication Token (authToken) is stored in cache of authenticator and returned
   to the caller. The caller must provide authToken back to this command.
   This code assumes that it's a 1stF IAuthnr and it doesn't store RawKeyHandles internally.
*/
function Sign(authMode, finalChl, appID, apiKey, personaID, trans, keyHandleList,
             authToken)
{
       // Make sure that provided authToken equals the one stored in cache.
       if (!isValidAuthToken(authToken)) {
              return UAF_STATUS_ERR_ACCESS_DENIED;
       }

       try {
              var rawkh = null;
              var usernameList = [];
              foreach(kh in keyHandleList) {
                     rawkh = unwrap(kh);
                     if (rawkh.personaID != personaID ||
                         rawkh.appIDHash != SHA256(appID) ||
                         rawkh.apiKey != apiKey) {
                            continue;
```

```
                }
                usernameList.push({"uname": rawkh.username, "kh": kh});
            }

            // If number of keyHandles is 1 - prepare SIGNDATA
            if (usernameList.length == 1) {
                if (trans != null) {
                    // Show transaction on Secure Display.
                    ret = showOnDisplay(trans)
                    // If user cancelled - return an error
                    if (ret == false) {
                        return UAF_STATUS_ERR_USER_CANCELLED;
                    }
                    transHash = SHA256(trans)
                }

                // Create UAF to-be-signed object
                var tbs = prepareTLVSignTBS(finalChl, transHash);

                // Calculate signature
                var signature = sign(rawkh.uauth_priv, tbs)

                // Prepare response buffer and return
                return prepareTLVWithSIGNDATA(UAF_STATUS_OK, tbs, signature)
            }

            // Need to return usernames and ask the user select a single username.
            // After user selects it - Sign function will be called again with
            // a single keyHandle
            return prepareTLVWithUsernames(UAF_STATUS_OK, usernameList);
        }
        catch (err) {
            // Prepare TLV response buffer, put error code and return
            return prepareTLVWithError(err)
        }
}
```

5.4 Deregister Command

General Description

This command deletes a registered UAF credential from Authenticator. Only Authenticators which store RawKeyhandle (or KeyHandle) in internal storage must support this command.

Authenticator MUST:

1. If it's an IAuthnr
    a. Delete KeyHandle associated with KeyID, AppID, KeyID, PersonaID and PubKeyHash
2. If it's an XAuthnr
    a. Delete KeyHandle associated with KeyID, AppID and PubKeyHash

Error Codes:

- UAF_STATUS_ERR_ACCESS_DENIED

Command Structure

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_DEREG_CMD
1.1	UINT16	Length	Entire Command Length
1.2	UINT16	AppIDSize	AppID size (max 256 bytes)
1.3	BYTE	AppID[AppIDSize]	AppIdentity provided by ASM
1.4	UINT16	KeyIDSize	KeyID size (max 32 bytes)
1.5	BYTE	KeyID[KeyIDSize]	KeyID provided by ASM
1.6	UINT16	APIKeySize	APIKey size (max 32 bytes)
1.7	BYTE	APIKey[APIKeySize]	APIKey provided by ASM
1.8	UINT16	PersonaIDSize	PersonaID size - max 32 bytes
1.9	BYTE	PersonaID[PersonaIDSize]	PersonaID provided by ASM
1.10	UINT16	HashSize	Hash of uauth public key hash size - max 32 bytes
1.11	BYTE	PublicKeyHash[HashSize]	Hash of uauth public key hash

Command Response

	TLV Structure		Description
1	UINT16	CmdTag	TAG_UAFV1_SIGN_CMD_RESP
1.1	UINT16	Length	Entire Length of Command Response
1.2	UINT16	StatusCode	StatusCode returned by Authenticator

Non-Normative Example Code

```
/*
    This code assumes that it's a 1stF IAuthnr and it doesn't store RawKeyHandles internally.
*/
function Deregister(appID, keyID, apiKey, personaID, publicKeyHash)
{
       try {
           var rawkh = null;
              var khList = storage.getKeyHandleList();
              foreach(kh in khList) {
                     rawkh = unwrap(kh);
                     if (rawkh.appIDhash == SHA256(appID) &&
                            rawkh.personaID == personaID &&
                            rawkh.apiKey == apiKey &&
                            rawkh.keyID == keyID &&
                            rawkh.publicKeyHash == publicKeyHash) {
                                storage.removeKeyHandle(kh);
                     }
              }

              // Return status
              return prepareTLVWithSuccess(UAF_STATUS_OK);
       }
       catch (err) {
              // Prepare TLV response buffer, put error code and return
              return prepareTLVWithError(err)
       }
}
```

Appendix: Security Guidelines

This section is informative only.

Category	Guidelines
Wrap.sym	If the Authenticator uses a wrapping key Wrap.sym, then the Authenticator must protect Wrap.sym as its most sensitive asset. The overall security of Authenticator highly depends on the protection level of this key. Wrap.sym strength MUST be equal or higher than the strength of secrets stored in RawKeyHandle. Refer to "*NIST Special Publication 800-57*" and "*NIST Special Publication 800-38F*" publications for more information about choosing the right wrapping algorithm and implementing it correctly. It is highly recommended to generate, store and operate this key inside a trusted execution environment. In situations where physical attacks and side channel attacks are considered in the threat model

	it is highly recommended to use a tamper-resistant hardware module such as a Smart Card. If the Authenticator uses a wrapping key to unwrap a data, it must ensure that unwrapping invalid data (e.g. KeyHandle) and successfully unwrapping data which has invalid contents (e.g. KeyHandle from invalid origin) are indistinguishable.
Private Keys (Uauth.priv and Att.priv)	This document requires (a) the attestation key to be used for attestation purposes only and (b) the authentication keys to be used for FIDO authentication purposes only. The related to-be-signed objects (i.e. Key Registration Data and SignData) are designed to reduce the likelihood of such attacks: 1. They start with a tag marking them as specific FIDO objects 2. They include an Authenticator generated random value. As a consequence all to-be-signed objects are unique with very high probability. 3. They have a structure allowing only very few fields containing uncontrolled values, i.e. value which are neither generated nor verified by the Authenticator
Att.priv	Authenticator must protect Att.priv as a very sensitive asset. The overall security of Authenticator depends on the protection level of this key. It is highly recommended to store and operate this key inside a tamper-resistant hardware module such as a Smart Card. Authenticators must ensure that the Attestation Private Key Att.priv 1. Is *only* used to attest Authentication Keys generated and protected by the FIDO Authenticator using the FIDO defined data structures, KeyRegistrationData. 2. Never is accessible outside the FIDO Authenticator boundary. Attestation must be implemented in a way that two different relying parties cannot link registrations, authentications or other transactions.
Uauth.priv	Authenticator must protect all Uauth.priv keys as its most sensitive assets. The overall security of Authenticator highly depends on the protection level of these keys. It is highly recommended to generate, store and operate this key inside a trusted execution environment. In situations where physical attacks and side channel attacks are considered in the threat model it is highly recommended to use a tamper-resistant hardware module such as a Smart Card. FIDO Authenticators must ensure that Authentication Private Keys (Uauth.priv) 1. are specific to the particular account at one relying party (relying party is identified by an App Identity) 2. are generated based on good random numbers generated using sufficient entropy. The challenge provided by FIDO Server SHOULD be mixed into the entropy pool in order to add additional entropy. 3. are never directly revealed, i.e. always remain in exclusive control of the FIDO Authenticator 4. are only being used for the defined Authentication Modes, i.e. a. authenticating to the AppIdentity they have been generated for, or

	b. confirming transaction to the AppIdentity they have been generated for, or 5. are only being used to create the FIDO defined data structures, i.e. SignData. 6. Authentication tokens and Transaction tokens only: a. are only being used after requiring the user to authenticate to the FIDO Authenticator. This authenticator shall be resistant against software based attacks. b. the confidentiality of the Authentication data is protected (e.g. malware cannot access a PIN entered by the user).
Username	Username MUST NOT be returned in plaintext in any condition other than the conditions described for SIGN command. In all other conditions Usernames MUST be stored inside KeyHandle.
AppIDs and KeyIDs	Registered AppIDs and KeyIDs MUST NOT be returned by Authenticator in plaintext. Additionally if attacker gets physical access to the authenticator - it should not be easy to read out AppIDs and KeyIDs.
Crypto Kernel	Crypto Kernel is a module of the Authenticator implementing crypto functions (key generation, signing, wrapping, etc) necessary for UAF and having access to Uauth.priv, Att.priv and Wrap.sym. This module must reside within the same security boundaries as Uauth.priv, Att.priv and Wrap.sym keys are residing. If it resides in a different module than the implementation must guarantee the same level of security as if they would reside within the same module. It is highly recommended to generate, store and operate this key inside a trusted execution environments. In situations where physical attacks and side channel attacks are considered in the threat model it is highly recommended to use a tamper-resistant hardware module such as a Smart Card. “Software” based Authenticators must make sure to use state of the art code protection and obfuscation techniques to protect this module and whitebox encryption techniques to protect the associated keys. Authenticators need good Random Number Generators using good entropy source enough for 1. generating authentication keys 2. generating signatures 3. computing Authenticator generated challenges If the Authenticator doesn’t have sufficient entropy for generating strong random numbers, it should fail safe.
Matcher	Tampering the Matcher module may have significant security consequences. It is highly recommended for this module to reside within integrity boundaries of Authenticator and detect tampering of itself. It is highly recommended to run this module inside a trusted execution environment (TEE). Authenticators which have separated Matcher and CryptoKernel modules should implement mechanisms which would allow CryptoKernel to securely receive assertions from Matcher module indicating user’s local verification status.

	Software based Authenticators (if not in trusted execution environment) must make sure to use state of the art code protection and obfuscation techniques to protect this module. When Authenticator receives a wrong AuthToken it should treat it as an attack and invalidate the cached AuthToken. AuthToken should have a lifetime not longer than 10 seconds. PIN based Authenticators MUST implement anti-hammering on PIN. Biometrics based authenticators MUST protect the captured biometrics data (such as fingerprints) as well as the reference data (templates) and make sure that they never leave the security boundaries of authenticators.
Random Numbers	The FIDO Authenticator uses its random number generator to generate authentication key pairs, client side challenges and potentially for creating ECDSA signatures. Weak random numbers will make FIDO vulnerable to certain attacks. It is important for the FIDO Authenticator to work with good random numbers only.
Secure Display	Secure Display MUST ensure that the user is presented with the provided Transaction Text, e.g. not overlaid by other display elements and clearly recognizable. See [Clickj] for some examples of threats and potential counter-measures For more guidelines refer to Global Platform “Trusted User Interface API” (GPD_SPE_020).
Certifications	Vendors must strive passing security standard certifications with Authenticators, such as FIPS 140-2, Common Criteria and similar. Passing such certifications will positively impact the UAF implementation inside Authenticator.
Signature Counter	Good protection measures of the Attestation key (Att.priv) is one method to prevent cloning authenticators. In some situations the protection measures might not be sufficient. If the Authenticator maintains a signature counter, then the FIDO Server would have an additional method to detect cloned authenticators. If Signature-Counter is implemented: ensure that the Signature-Counter 1. is increased by any authentication / transaction confirmation operation and 2. cannot be manipulated/modified otherwise (e.g. API calls, etc.)

References

[Clickj] Clickjacking: Attacks and Defenses, Lin-Shung Huang and Collin Jackson Carnegie Mellon University; Alex Moshchuk, Helen J. Wang, and Stuart Schlechter Microsoft Research. Download https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf

Specification set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

FIDO: Fast IDentity Online Authenticator Metadata

Draft v0.2

CONTENTS

1 Overview

FIDO Authenticators may have different forms and capabilities. It's important to formally define these entities and categorize. This will allow adding semantics in the communication of FIDO Servers and Clients.

This document defines metadata for FIDO Authenticators.

2 Metadata

Name	Description
AAID	String: the Authenticator Attestation ID
AttestationCertificate	String: Base64url encoded representation of the DER encoded Attestation Certificate or Certificate Chain. For non-attested authenticators, this value MUST be left empty.
Description	String: human readable short description of Authenticator
AuthenticationFactor	A 64 bit number representing the bit fields defined by the USER_VERIFY constants in the FIDO Registry of Predefined Values. Any number of the relevant bits may be set.
ValidAttachmentTypes	A 64 bit number representing the bit fields defined by the ATTACHMENT_HINT constants in the FIDO Registry of Predefined Values. The connection state and topology of an authenticator may be transient and cannot be relied on as authoritative by a Relying Party, but the medadata field should have all the bit flags set for the topologies possible for the authenticator. For example, an authenticator instantiated as a single-purpose hardware token that can communicate over bluetooth should set ATTACHMENT_HINT_EXTERNAL but not ATTACHMENT_HINT_INTERNAL
KeyProtection	A 64 bit number representing the bit fields defined by the KEY_PROTECTION constants in the FIDO Registry of Predefined Values.

SecurityType	Specifies security class. The following classes are defined: • "SecureModuleWithAttestation" - A hardware security module that both securely stores key material, performs FIDO crypto operations inside the module and has the Attestation Private Key securely stored in the module, e.g. Secure Element, TPM, etc. • "SecureModule" – The same as "SecureModuleWithAttestation" but Attestation Private Key stored outside hardware boundary, e.g. TEE or TPM without Attestation enabled. • "Software" - An Authenticator that is exclusively built in software; keys are stored only in software (although they may be protected using operating system-based security mechanisms), and all FIDO crypto operations are performed in software.
SecureDisplay	A 64 bit number representing the bit fields defined by the SECURE_DISPLAY constants in the FIDO Registry of Predefined Values.
SecondFactorOnly	Boolean: Indicates if the Authenticator is designed to be used only as a second factor
Logo	String: Authenticator Logo, encoded in base64.
RegInfo	Describes what capabiliites the Authenticator supports for Registration operation. • RegMode – Registration Mode • RegScheme – Registration Scheme See FIDO OSTP Protocol Specification for more details on supported modes and schemes.
AuthInfo	Describes what capabiliites the Authenticator supports for Authentication operation. • AuthSuite – Supported Authentication Suite • AuthScheme – Authentication Scheme

	See FIDO OSTP Protocol Specification for more details on supported suites and schemes.

3 Schema

The Authenticator Metadata document is a JSON structure containing the above information. This forms the Payload portion of a JSON Web Signature (JWS) signed with the X.509 certificate that issued the Attestation Certificate, or the software vendor's issuing certificate in the case of a non-attestable implementation.

TDB [need to define JSON schema]

FIDO Registry of Predefined Values - revision 01

Specification set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

Editors:
Rolf Lindeman, Nok Nok Labs
Davit Baghdasaryan, Nok Nok Labs
Brad Hill, PayPal

Contributors:

Abstract:
This document defines all the strings and constants reserved by FIDO protocols.

This document's purpose is to define values that may be extensible and referenced from additional specifications.

Contents

1. Introduction

This document is the FIDO Alliance registry of predefined values that are referenced by different components in FIDO architecture.

This registry is expected to evolve along with the FIDO Alliance specifications and documents.

2. Authenticator Characteristics

2.1. Authentication Factors

The USER_VERIFY constants are flags in a bitfield represented as a 64 bit long. They describe the methods and capabilities of an UAF authenticator for locally verifying a user. In most cases, the operational details of these methods are opaque to the server, but in some cases verification may involve transmission of attested measurement, such as for USER_VERIFY_LOCATION. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form Authenticator policies in UAF protocol messages.

USER_VERIFY_PRESENCE 0x01
: This flag will be set if the authenticator is able to confirm user presence in any fashion. If this flag and no other is set for user verification, the guarantee is only that the authenticator cannot be operated without some human intervention, not necessarily that the presence verification provides any level of authentication of the human's identity. (e.g. a device that requires a touch to activate)

USER_VERIFY_FINGERPRINT 0x02
: This flag will be set if the authenticator uses any type of measurement of a fingerprint for User-to-Authenticator authentication.

USER_VERIFY_PASSCODE 0x04
: This flag will be set if the authenticator uses a local-only passcode for User-to-Authenticator authentication.

USER_VERIFY_VOICEPRINT 0x08
: This flag will be set if the authenticator uses a voiceprint for User-to-Authenticator authentication.

USER_VERIFY_FACEPRINT 0x10
: This flag will be set if the authenticator uses any manner of face recognition to locally authenticate the user.

USER_VERIFY_LOCATION 0x20

This flag will be set if the authenticator uses any form of location sensor or measurement for User-to-Authenticator authentication and/or returns a location measurement to the Relying Party as an additional user verification.

USER_VERIFY_EYEPRINT 0x40

This flag will be set if the authenticator uses any form of eye biometrics for User-to-Authenticator authentication.

USER_VERIFY_PATTERN 0x80

This flag will be set if the authenticator uses a drawn pattern for User-to-Authenticator authentication.

USER_VERIFY_HANDPRINT 0x100

This flag will be set if the authenticator uses any measurement of a full hand (including palmprint, hand geometry or vein geometry) for User-to-Authenticator authentication.

USER_VERIFY_NONE 0x200

This flag will be set if the authenticator will respond without any user interaction.

USER_VERIFY_OTHER 0x400

Some method other than those defined here but not none.

2.2. Key Protection Types

The KEY_PROTECTION constants are flags in a bit field represented as a 64 bit long. They describe the method an authenticator uses to protect the private key material for FIDO registrations. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form Authenticator policies in UAF protocol messages. When used in metadata describing an authenticator, several of these flags are exclusive with others - the certified metadata may have at most one of the mutually exclusive bits set to 1. When used in authenticator policy, any bit may be set to 1, e.g. to indicate that a server is willing to accept authenticators using either KEY_PROTECTION_SOFTWARE and KEY_PROTECTION_HARDWARE.

KEY_PROTECTION_SOFTWARE 0x01

This flag will be set if the authenticator uses software-based key management.

Exclusive in authenticator metadata with KEY_PROTECTION_HARDWARE, KEY_PROTECTION_TEE, KEY_PROTECTION_SECURE_ELEMENT

KEY_PROTECTION_HARDWARE 0x02

This flag will be set if the authenticator uses hardware-based key management.

Exclusive in authenticator metadata with KEY_PROTECTION_SOFTWARE

KEY_PROTECTION_TEE 0x04

This flag will be set if the authenticator uses the Trusted Execution Environment for key management. In authenticator metadata, this flag should be set in conjunction with KEY_PROTECTION_HARDWARE.

Exclusive in authenticator metadata with KEY_PROTECTION_SOFTWARE, KEY_PROTECTION_SECURE_ELEMENT

KEY_PROTECTION_SECURE_ELEMENT 0x08

This flag will be set if the authenticator uses a Secure Element for key management. In authenticator metadata, this flag should be set in conjunction with KEY_PROTECTION_HARDWARE.

Exclusive in authenticator metadata with KEY_PROTECTION_TEE, KEY_PROTECTION_SOFTWARE

KEY_PROTECTION_REMOTE_HANDLE 0x10

This flag will be set if the authenticator does not store per-Origin keys at the client, but relies on a server-provided key handle.

This flag MUST be set in conjunction with one of the other KEY_PROTECTION flags to indicate how the local key handle unwrapping key and operations are protected.

Servers can unset this flag in authenticator policy if they are unprepared to store and return key handles, for example, if they have a requirement to respond indistinguishably to authentication attempts against userIDs that do and do not exist.

2.3. Authenticator Attachment Hints

The ATTACHMENT_HINT constants are flags in a bit field represented as a 64 bit long. They describe the method an authenticator uses to communicate with the system on which the FIDO client software is executing. These constants are reported and queried through the UAF Discovery APIs, and used to form Authenticator policies in UAF protocol messages.

Because the connection state and topology of an authenticator may be transient, these values are only hints that can be used by server-supplied policy to guide the user experience, e.g. to prefer a device that is connected and ready for authenticating or confirming a low-value transaction, rather than one that is more secure but requires more user effort. These values are not reflected in authenticator metadata and cannot be relied on by the relying party, although some models of authenticator may provide attested measurements of similar data as part of UAF response messages.

ATTACHMENT_HINT_INTERNAL 0x01

This flag indicates that the authenticator is permanently attached to the system on which the FIDO client software is running.

A device such as a smartphone may have authenticator functionality that is able to be used both locally and remotely. In such a case, the FIDO client MUST filter and exclusively report only the relevant bit during Discovery and when performing policy matching.

ATTACHMENT_HINT_EXTERNAL 0x02

This flag indicates, for a hardware-based authenticator, that it is removable or remote from the system on which the FIDO client software is running.

A device such as a smartphone may have authenticator functionality that is able to be used both locally and remotely. In such a case, the FIDO client MUST filter and exclusively report only the relevant bit during Discovery and when performing policy matching.

ATTACHMENT_HINT_WIRED 0x04

Indicates that an external authenticator currently has an exclusive wired connection, e.g. through USB, Firewire or similar, to the system on which the FIDO client software is executing.

ATTACHMENT_HINT_WIRELESS 0x08

Indicates that an external authenticator communicates with the system on which the FIDO client software is executing through a personal area or otherwise non-routed wireless protocol, such as Bluetooth or NFC.

ATTACHMENT_HINT_NFC 0x10

Indicates that an external authenticator is able to communicate by NFC to the FIDO client software. As part of authenticator metadata, or when reporting characteristics through Discovery, if this flag is set, the ATTACHMENT_HINT_WIRELESS flag SHOULD also be set.

ATTACHMENT_HINT_BLUETOOTH 0x20

Indicates that an external authenticator is able to communicate using Bluetooth to the FIDO client software. As part of authenticator metadata, or when reporting characteristics through Discovery, if this flag is set, the ATTACHMENT_HINT_WIRELESS flag SHOULD also be set.

ATTACHMENT_HINT_NETWORK 0x40

Indicates that the authenticator is not on the same system as the FIDO client software but communicates with it over a non-exclusive network. (e.g. over a TCP/IP LAN or WAN, as opposed to a point-to-point Bluetooth connection)

ATTACHMENT_HINT_READY 0x80

Indicates that an external authenticator is in a ready state. e.g. a Bluetooth connected device that is currently paired and connected or a USB device that is plugged in.

2.4. Secure Display Types

The SECURE_DISPLAY constants are flags in a bit field represented as a 64 bit long. They describe the availability and implementation of a secure display capability required for the Transaction Confirmation operation. These constants are used in the authoritative metadata for an authenticator, reported and queried through the UAF Discovery APIs, and used to form Authenticator policies in UAF protocol messages.

SECURE_DISPLAY_ANY 0x01

This flag indicates, that some form of secure display is available on this authenticator.

SECURE_DISPLAY_PRIVILEGED_SOFTWARE 0x02

This flag indicates, that a software-based secure display operating in a privileged context is available on this authenticator.

Software based displays are typically provided by the FIDO client software rather than the authenticator itself. A FIDO client that is capable of providing this capability MAY set this bit for all authenticators of type ATTACHMENT_EMBEDDED, even if the authoritative metadata for the authenticator does not indicate this capability.

SECURE_DISPLAY_TEE 0x04

This flag indicates that the authenticator implements a secure display in the Trusted Execution Environment.

SECURE_DISPLAY_HARDWARE 0x08
This flag indicates, that a secure display based on hardware assisted capabilities is available on this authenticator.

SECURE_DISPLAY_REMOTE 0x10
This flag indicates, that the secure display is provided on a distinct device from the system the FIDO client software is executing on.

3. UAF Authenticator Command TLV Tags (0x0001-0x1000)

The internal structure of UAF authenticator commands is a "Tag-Length-Value" (TLV) sequence. The **Tag** is a 2-byte unsigned value describing the type of field the data represents, the **Length** is a 2-byte unsigned value indicating the size of the value in bytes and **Value** is the variable-sized series of bytes which contain data for this item in the sequence. The following tags have been allocated for data types in UAF protocol messages:

TAG_UAFV1_RESPONSE 0x0001
The content of this TAG is Authenticator Response for Registration (TAG_UAFV1_KRD) or Sign (TAG_UAFV1_SIGNDATA) operations.

TAG_UAFV1_KRD 0x0002
Indicates Key Registration Data.

TAG_UAFV1_SIGNDATA 0x0003
Indicates SignData.

TAG_DER_ATTESTATION_CERT 0x0004
Indicates DER encoded Attestation Batch Certificate.

TAG_KEYHANDLE 0x0005
Indicates a wrapped KeyHandle.

TAG_AUTHENTICATOR_VERSION 0x0006
Indicates Authenticator version.

TAG_REG_RESET_CTR 0x0007
Indicates Registration Reset Counter.

TAG_REGISTRATION_CTR 0x0008
Indicates Registration Counter.

TAG_SIGN_CTR 0x0009
Indicates Signing Counter.

4. Crypto Suites

These constant strings and values indicate the specific cipher, mode of operation and encoding of encrypted data.

UAF_ALG_SIGN_ECDSA_SHA256_RAW 0x01
ECDSA signature MUST have raw R and S buffers, encoded in big endian.
For example for ECC-P256 curve the signature MUST have the following form
[R (32 bytes), S (32 bytes)]

UAF_ALG_SIGN_ECDSA_SHA256_DER 0x02
DER encoded ECDSA signature,
i.e. DER encoded SEQUENCE { r INTEGER, s INTEGER }

UAF_ALG_KEY_ECC_NISTP256R1_X962_RAW 0x50
Raw ANSI X.9.62 formatted public key

UAF_ALG_KEY_ECC_NISTP256R1_X962_DER 0x51
DER encoded ANSI X.9.62 formatted public key

UAF_ALG_SIGN_RSASSA-PSS_SHA256_RAW 0x03
RSASSA-PSS signature MUST have raw S buffers, encoded in big endian. For example for RSA 2048 the signature have the following form [S (256 bytes)]

UAF_ALG_SIGN_RSASSA-PSS_SHA256_DER 0x04

DER encoded RSASSA-PSS signature

UAF_ALG_KEY_RSA_2048_PSS_RAW 0x52

Raw RSASSA-PSS formatted public key

UAF_ALG_KEY_RSA_2048_PSS_DER 0x53

ASN.1 DER encoded RSASSA-PSS formatted public key

5. UAF Server Response Codes

When responding to a UAF client request message, the UAF server indicates status by the following codes, encoded as a JSON number. These codes are part of the JSON that constitutes the HTTP response body and indicate the status of the UAF protocol operation on the server, NOT the HTTP status code.

Code	Meaning
200	OK. Registration completed
202	Accepted. Registration message accepted, but not completed at this time. The RP may need time to process the attestation, run risk scoring, etc. The server SHOULD NOT send an authenticationToken with a 202 response
400	Bad Request. The server did not understand the message
401	Unauthorized. The userid must be authenticated to register a FIDO Authenticator, or this KeyID is not associated with this UserID.
403	Forbidden. The userid is not allowed to register a FIDO Authenticator. Client SHOULD NOT retry
408	Request Timeout
480	Unknown AAID. The server was unable to locate authoritative metadata for the AAID.
481	Unknown KeyID. The server was unable to locate a registration for the given

	UserID and KeyID combination.
490	Channel Binding Refused. The server refused to service the request due to a missing or mismatched channel binding(s).
491	Server Challenge Invalid. The server refused to service the request because the challenge was unknown, expired or the server has previously serviced a message with the same challenge and user ID.
492	Unacceptable Authenticator. The authenticator is not acceptable according to the server's policy, for example because the capability registry used by the server reported different capabilities than client-side discovery.
493	Revoked Authenticator. The authenticator is considered revoked by the server.
494	Unacceptable Key. The key being registered is unacceptable. Perhaps it is on a list of known weak keys or uses insecure parameter choices.
495	Unacceptable Algorithm. The server believes the authenticator to be capable of using a stronger mutually-agreeable algorithm than was presented in the request.
496	Unacceptable Attestation. The attestation(s) provided were not accepted by the server for registration.
497	Unacceptable Client Capabilities. The server was unable or unwilling to use required capabilities provided supplementally to the authenticator by the client software.
498	Unacceptable Content. There was a problem with the contents of the message and the server was unwilling or unable to process it.
500	Internal Server Error

6. UAF Server Response Token Types

When responding to a UAF client request message, the UAF server may include additional authentication or authorization tokens. These are described by the following WebIDL enum:

```
enum TokenType {
    "HTTP_COOKIE",
    "OAUTH",
    "OAUTH2",
    "SAML1_1",
```

```
    "SAML2",
    "JWT"
};
```

The values of the enum refer to the following token types:

HTTP_COOKIE
: If the user agent is a standard web browser or other HTTP native client with a cookie store, this TokenType SHOULD NOT be used. Cookies should be set directly with the Set-Cookie HTTP header. For non- HTTP or non-browser contexts this indicates a token intended to be set as an HTTP cookie. [RFC 6525] (for example, a native VPN client on Microsoft Windows that authenticates with UAF might use this TokenType to add a cookie to the browser cookie jar.)

OAUTH
: Indicates an OAuth token

OAUTH2
: Indicates an OAuth2 token

SAML1_1
: Indicates a SAML 1.1 token

SAML2
: Indicates a SAML 2.0 token

JWT
: Indicates a JSON Web Token (JWT)

7. UAF Client Callback Error Codes

When an application using UAF invokes an asynchronous operation targeting the FIDO client software, the following error codes are used to indicate the status of the operation. Values are of the WebIDL type short.

NO_ERROR 0
: No error condition encountered.

WAIT_USER_ACTION 1
: Waiting on user action to proceed. (selecting an authenticator in the FIDO client user interface, performing local User-to-Authenticator authentication, or completing an enrollment step with an authenticator)

USER_CANCELLED 2

The user declined any necessary part of the interaction to complete the registration or no registered authenticator exists for authenticate, confirm or deregister calls.

UNSUPPORTED_VERSION 3

The UAFProtocolMessage indicates protocol versions not supported by this FIDO Client.

NO_SUITABLE_AUTHENTICATOR 4

No authenticator matching the AuthenticatorPolicy internal to the UAFProtocolMessage is available to service the request, or the user declined to consent to the use of a suitable authenticator.

INSECURE_TRANSPORT 5

window.location.protocol is not https or the DOM contains *insecure mixed content*.

PROTOCOL_ERROR 6

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

UNKNOWN 99

An error condition not described by other codes.

FIDO Technical Glossary

specification set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

Editors:
Rolf Lindeman, Nok Nok Labs
Davit Baghdasaryan, Nok Nok Labs
Brad Hill, PayPal

Contributors:

Abstract:
This document defines many of the technical terms and phrases used in FIDO Alliance specifications and documents.

Introduction:

This document is the FIDO Alliance glossary of normative technical terms.

This document is not an exhaustive compendium of all FIDO technical terminology because the FIDO terminology is built upon existing terminology. Thus many terms that are commonly used within this context are not listed. They may be found in the glossaries/documents/specifications referenced in the bibliography. Terms defined here that are not attributed to other glossaries/documents/specifications are being defined here.

This glossary is expected to evolve along with the FIDO Alliance specifications and documents.

Definitions:

AAID
: Authenticator Attestation ID. See *Attestation ID.*

APIKey
: A secret value that acts as a guard for Authenticator Commands. APIKeys are generated and provided by an ASM.

Application
: A set of functionality provided by a common entity (the application owner, aka the *Relying Party*), and perceived by the user as belonging together. For example, “PayPal”

is an application that allows users to make payments and send money.

Application Facet

An (application) facet is how an application is implemented on various platforms. For example, the application MyBank may have an Android app, an iOS app, and a Web app. These are all facets of the MyBank application.

Application Facet ID

A platform-specific identifier (URI) for an application facet.

- For Web applications, the facet id is the RFC 6454 origin.
- For Android applications, the facet id is the URI android:apk-key-hash:<*hash-of-apk-signing-cert*>
- For iOS, the facet id is the URI ios:bundle-id:<*ios-bundle-id-of-app*>

APP_IDENTITY

The RFC 6454 Web Origin of the Relying Party. [http://www.ietf.org/rfc/rfc6454.txt]
This identity is shared across all Facets of an Application.

APP_IDENTITY_HASH

The SHA256 hash of the APP_IDENTITY.

Attestation

In the FIDO context, attestation is how Authenticators make claims to a Relying Party that the keys they generate, and/or certain measurements they report, originate from genuine devices with certified characteristics.

Attestation Certificate

A public key certificate related to an Attestation Key.

Attestation ID

A unique identifier assigned to a model, class or batch of FIDO Authenticators that all share the same characteristics, and which a Relying Party can use to look up an Attestation Public Key and Authenticator Metadata for the device.

Attestation [Public / Private] Key

A key used for FIDO Authenticator attestation.

Attestation Root Certificate

A root certificate explicitly trusted by the FIDO Alliance, to which Attestation Certificates chain to.

Authentication

In the FIDO protocols “Authentication” may refer to several actions depending on the context and qualifications with which it is used:

a. User-to-Authenticator Authentication, also known as Local Authentication, in which a user authenticates directly to a FIDO Authenticator with a biometric measurement, PIN or other means. Local Authentication is used by the FIDO Authenticator to authorize the use of keys or query state from the device.
b. Authenticator-to-Relying Party Authentication, in which a user, having gained access to use a key managed by a FIDO Authenticator, uses that key to perform a cryptographic authentication with a Relying Party over a network using one of the FIDO protocols.
c. Authentication also refers to a fundamental operation of the FIDO family of protocols which encompasses the entire ceremony of a registered user locally authenticating to their FIDO Authenticator(s) and the FIDO Client leveraging that to complete the authentication to the Relying Party.

Authentication Algorithm

The combination of signature and hash algorithms used for authenticator-to-relying party authentication.

Authentication Factor

The means by which User-to-Authenticator authentication is accomplished. e.g. fingerprint, voiceprint, or PIN.

Authentication Scheme

The combination of an Authentication Algorithm with a message syntax or framing that is used by an Authenticator when constructing a response.

ASI (Authentication Scheme Input)

Additional input that must be provided by the Relying Party server and is specific to a given Authentication Scheme.

Authenticator

See *FIDO Authenticator*.

Authenticator, 1stF / First Factor

A FIDO Authenticator that transactionally provides a username and at least two authentication factors: cryptographic key material (something you have) plus user verification (something you know / something you are) and so can be used by itself to complete an authentication. 1stF Authnrs must have a user verification method more specific than presence/touch.

Authenticator, 2ndF / Second Factor

A FIDO Authenticator which acts only as a second factor. 2ndF Authnrs always require a single Key Handle to be provided before responding to a Sign command. They might or might not have a user verification method.

Authenticator Attestation

The process of communicating a cryptographic assertion to a Relying Party that a key presented during Registration was created and protected by a genuine Authenticator with verified characteristics.

Authenticator Metadata

Verified information about the characteristics of a certified Authenticator, associated with an AAID and available from the FIDO Alliance. FIDO Servers are expected to have access to up-to-date metadata to be able to interact with a given Authenticator.

Authenticator Policy

A JSON data structure that allows a Relying Party to communicate to a FIDO Client the capabilities or specific authenticators that are allowed or disallowed for use in a given operation.

ASM / Authenticator Specific Module

Software associated with a FIDO Authenticator that provides a uniform interface between the hardware and FIDO Client software.

Certificate

An X.509v3 certificate defined by the profile specified in RFC5280 and its successors. [http://www.ietf.org/rfc/rfc5280.txt]

Channel Binding

See: http://tools.ietf.org/html/rfc5056

A channel binding allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication to the higher layer to the channel at the lower layer.

Correlation Handle

Any piece of information that may allow, in the context of FIDO protocols, implicit or explicit association and or attribution of multiple actions, believed by the user to be distinct and unrelated, back to a single unique entity. An example of a correlation handle outside of the FIDO context is a client certificate used in traditional TLS mutual authentication: because it sends the same data to multiple Relying Parties, they can therefore collude to uniquely identify and track the user across unrelated activities.

Deregistration

A phase of a FIDO protocol in which a Relying Party tells a FIDO Authenticator to forget a specified piece of (or all) locally managed key material associated with a specific Relying Party account, in case such keys are no longer considered valid by the Relying Party.

Discovery

A phase of a FIDO protocol in which a Relying Party is able to determine the availability of FIDO capabilities at the client's device, including metadata about the available authenticators.

E(K,D)

Designates Encryption of data *D* with key *K*

Enrollment

The process of making a User known to an Authenticator. This might be a Biometric Enrollment as defined in (http://biometrics.gov/Documents/Glossary.pdf) or involve processes such as taking ownership of and setting a PIN or password for a non-biometric cryptographic storage device. Enrollment may happen as part of a FIDO protocol ceremony, or it may happen outside of the FIDO context for multi-purpose authenticators.

External Authenticator (--> Roaming Authenticator?)

A FIDO Authenticator configured for potential use with FIDO Clients in multiple computing contexts. It uses its own local storage for registrations in a 1st Factor scenario and is not required to use an access control mechanism to restrict access to key material by multiple FIDO Clients. (compare to Internal Authenticator)

Facet

See *Application Facet*

Facet ID

See *Application Facet ID*

FIDO Authenticator

An Authentication entity that meets the FIDO Alliance's requirements and which has published metadata.

A FIDO Authenticator is responsible for User-to-Authenticator Authentication and maintaining the cryptographic material required for the Authenticator-to-Relying Party Authentication.

It is important to note that a FIDO Authenticator is only considered such for and in relation to its participation in FIDO Alliance protocols. Because the FIDO Alliance aims to utilize a diversity of existing and future hardware, many devices used for FIDO may have other primary or secondary uses. To the extent that a device is used for non-FIDO purposes such as local operating system login or network login with non-FIDO protocols, it is not considered a FIDO Authenticator and its operation in such modes is NOT subject to FIDO Alliance guidelines or restrictions, including those related to security and privacy.

A FIDO Authenticator may be referred to as simply an Authenticator or abbreviated as "Authnr". Important distinctions in an Authenticator's capabilities and user experience may be experienced depending on whether it is an "Internal" or 'External" authenticator, and whether it is a "First Factor" or "Second Factor" authenticator.

FIDO Client

This is the software entity processing the UAF protocol messages on client side and meeting UAF requirements. FIDO Clients MAY have two forms:

- A software component implemented in a User Agent (either web browser or native application).
- A standalone piece of software shared by several User Agents. (Web browsers or native applications)

FIDO Data / FIDO Information

Any information gathered or created as part of completing a FIDO transaction. This includes but is not limited to, biometric measurements of or templates for the user and FIDO transaction history.

FIDO Plugin

The implementation of the interface in a web browser that brokers messages between a client side web application and FIDO client. This component is referred to as a "plugin" even if the APIs are built natively into the web browser or injected into a hosted browser component.

FIDO Server

Server software typically deployed in Relying Party's infrastructure that meets the UAF protocol's server requirements

FIDO User Device

The computing device where the FIDO Client operates and from which the user initiates an action that utilizes FIDO.

Internal Authenticator (--> Associated Authenticator?)

A FIDO Authenticator which uses an access control mechanism with FIDO Clients to restrict the use of Authentication Keys and which might also use local system storage. (compare to External Authenticator)

JWA

JSON Web Algorithm, see [JWA]

JWK

JSON Web Key, see [JWK]

JWS
: JSON Web Signature, see [JWS]

JPSK
: JSON Private and Symmetric Key, see [JPSK]

KeyID
: KeyID identifies a registered key between an Authenticator and a FIDO Server for 1F Authenticators. It is used in concert with AAID to identify a particular Authenticator that holds the necessary key. KeyID is the SHA256 hash of the KeyHandle managed by the ASM.

KeyHandle
: A key container created by a FIDO Authenticator, containing a private key and (optionally) other data (such as Username). A key handle may be wrapped (encrypted with a key known only to the authenticator) or unwrapped. In the unwrapped form it is referred to as a Raw Key Handle. 2F Authenticators must retrieve their Key Handles from the Relying Party to function, 1F Authenticators manage the storage of their own Key Handles, either internally (for External Authenticators) or at the ASM layer. (for Internal Authenticators)

Key Registration
: The process of securely establishing a key between FIDO Server and FIDO Authenticator.

Key Registration Data (KRD)
: {TBD}

Matcher
: A component of a FIDO Authenticator which is able to perform local User Verification. (biometric matching, PIN verification, etc.)

TTEXT
: Transaction Text, i.e. text to be confirmed in the Transaction Confirmation operation.

TTEXT_HASH
: Hash of Transaction Text based on the following formula: TTEXT_HASH = SHA256(TTEXT)

Open Web Platform
: The interoperable application environment available in most web browsers and several app platforms encompassing a family of standards defined at the W3C, the IETF and ECMA. Key pieces of the Open Web Platform technology stack for FIDO include: HTTP 1.1, HTML5, XMLHttpRequest, and ECMAScript (JavaScript).

PersonaID
: An identifier provided by an ASM, PersonaID is used to associate different registrations. It can be used to create virtual identities on a single authenticator, for example to differentiate “personal” and “business” accounts. PersonaIDs can be used to manage privacy settings on the Authenticator.

PV
: Protocol Version

Registration
: A phase of a FIDO protocol in which a user generates and associates new key material with an account at the Relying Party, subject to policy set by the server and acceptable attestation that the authenticator and registration matches that policy.

Registration Scheme
: The Registration Scheme defines how the authentication key is being exchanged between the FIDO Server and the FIDO Authenticator.

RSI
: Registration Scheme specific server Input. Different Registration schemes may require different input. For each of them a specific RSI might be required to be defined.

Relying Party
: A web site or other entity that uses a FIDO protocol to authenticate users

S(K, D)
: Signing of data D with key K

Secure Display
: This is a feature of FIDO Authenticators able to show content of a message to a user and protect the integrity of this message.

Server Challenge
: A random value provided by the FIDO Server in the UAF protocol requests.

SignData
: {TBD}

Silent Authenticator
: FIDO Authenticator not requiring any user interaction for the authentication operation.

TLS
: Transport Layer Security

Transaction Confirmation

An operation in the FIDO protocol that allows a Relying Party to request that a FIDO Client and Authenticator with the appropriate capabilities display some information to the user, request that the user authenticate locally to their FIDO Authenticator to confirm it, and provide proof of possession of previously registered key material an attestation of the confirmation back to the Relying Party.

U2F

Universal 2nd Factor. The FIDO protocol and family of Authenticators to enable a cloud service to offer its users the options of using an easy–to–use, strongly–secure open standards–based 2nd factor device for authentication.

UAF

Universal Authentication Framework. The FIDO Protocol and family of Authenticators to enable a service to offer its users flexible and interoperable authentication.

UAuth.pub / UAuth.priv / UAuth.key / UAuth.sym

User authentication keys generated by FIDO Authenticator. UAuth.pub is the public part of key pair. UAuth,priv is the private part of the key. UAuth.sym indicates a symmetric key. UAuth.key is the more generic notation to refer to UAuth.priv.

UINT16

A 16 bit (2 bytes) unsigned integer.

UINT32

A 32 bit (4 bytes) unsigned integer.

UINT64

A 64 bit (8 bytes) unsigned integer.

Use Counter

A monotonically increasing counter maintained by the Authenticator. It is increased on every use of the Uauth (private) key. This value can be used by the FIDO Server to detect cloned Authenticators.

User

Relying Party's user and owner of the FIDO Authenticator.

User Agent

The user agent is a client application that is acting on behalf of a user in a client-server system. Examples of user agents include web browsers and mobile apps.

Username

Username is a human-readable string identifying a user's account at a Relying Party.

Web Application, Client-Side

The portion of a Relying Party application built on the Open Web Platform which executes in the User Agent. When the term "Web Application" appears unqualified or without specific context in FIDO documents, it generally refers to either the client-side portion or the combination of both client-side and server-side pieces of such an application.

Web Application, Server-Side

The portion of a Relying Party application that executes server-side and responds to HTTP requests. When the term "Web Application" appears unqualified or without specific context in FIDO documents, it generally refers to either the client-side portion or the combination of both client-side and server-side pieces of such an application.

FIDO Alliance UAF TWG Draft Specification

B. Hill

PayPal, Inc.

November 2013

FIDO UAF Client API and Bindings Interoperability Profile (rev 00)

draft-hill-uaf-client-api-00

Specification Set: uaf-v1.0-rd-20131213 (REVIEW DRAFT)

Abstract

NOTE: This is a DRAFT in-progress revision of this document.

This document describes a client API and bindings interoperability profile for FIDO (Fast IDentity Online) UAF (Universal Authentication Factor) version 1.0. This profile is defined for Web Browser applications in terms of:

An HTTPS protcol binding
HTTP 1.1 message framing
HTML5
XMLHttpRequest
ECMAScript

with these technologies collectively falling under the umbrella term "The Open Web Platform".

For Android apps, the profile is defined in Java sample code and the Java language bindings for WebIDL. (http://www.w3.org/TR/WebIDL-Java/)

This docuoment specifically addresses implementing the portion of FIDO UAF v1.0 interactions that occur in both the client and server side of a web application executing in the context of a general purpose web browser, and for applications on the Android platform, including the security and privacy considerations thereof.

Table of Contents

1. Conventions

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 BCP 14, RFC 2119 .

2. Introduction

FIDO (Fast IDentity Online) describes a set of technologies to enable standards-based interoperability of strong authentication technologies such as secure cryptographic hardware and biometric devices. The FIDO UAF (Universal Authentication Framework) standards enable use cases where FIDO replaces a password as the primary or sole authentication credential for a user account, or as a step-up credential without the use of a password.

The FIDO UAF protocol consists of four operations:

Discovery
: Discovery allows the relying party server to determine the availability of FIDO capabilities at the client, including metadata about the available authenticators.

Registration
: Registration allows the client to generate and associate new key material with an account at the relying party server, subject to policy set by the server and acceptable attestation that the authenticator and registration matches that policy.

Authentication
: Authentication allows a user to provide proof of possession of previously registered key material, and potentially other attested data, to the relying party server.

Transaction Confirmation
: Transaction Confirmation allows a server to request that a FIDO client and authenticator with the appropriate capabilities display some information to the user, request that the user authenticate locally to their FIDO authenticator to confirm it, and provide proof of possession of previously registered key material and an attestation of the confirmation back to the relying party server.

Deregistration
: Deregistration allows a relying party server to tell an authenticator to forget selected locally managed key material associated with that relying party in case such keys are no longer considered valid by the relying party.

The majority of the operations performed as part of FIDO UAF are out of scope for this document. Conceptually, the FIDO UAF client and server exchange UAF protocol messages tunnelled over HTTP through the browser, relying on the web application to provide context, initial authentication (where needed) and portions of the user experience necessary to complete the various ceremonies in the protocol.

For Web applications, support and protocol tunnelling features are accomplished by means of ECMAScript APIs exposed to client-side web applications by the browser or a browser plugin. These APIs are described by this document. The implementation and instantiation of the full FIDO client that these APIs connect to are not specified here - the client might be implemented entirely within the context of the browser or a browser plugin, as a distinct application, a system service or even on a remote device.

Android applications may either implement the UI and logic to initiate FIDO activities and communicate with the relying party server directly, or they may instantiate a WebView to do so. If the WebView technology supports the FIDO UAF APIs and bindings described in this document natively, the Android application may need to do little work. If not, the application may need to use addJavascriptInterface() to inject these APIs into the WebView and provide an implementation. This document describes the necessary APIs, bindings and communication patterns between an Android application and a FIDO client

implemented as an Android Bound Service necessary to complete the link between either a native UI or APIs injected into a WebView.

3. Definitions and Terminology

See the FIDO Techncial Glossary

4. API Notation Conventions

The API notation in this document describes ECMAScript bindings in WebIDL for applications executing in the context of the W3C Document Object Model (DOM). They are intended to be suitable for a web application to participate in UAF, and are designed to enforce and support existing notions of privacy and security in that context.

Where appropriate, the WebIDL bindings from the Web API notation apply to Android objects as well, using the Java Language Binding for Web IDL. (http://www.w3.org/TR/WebIDL-Java/)

Android-specific constructs are described and sample code for Android is provided in Java.

5. UAF HTTP TLS Binding

A secure channel MUST be established to convey UAF protcool messages, regardless of whichever intermediate protocol the UAF protcool is bound to.

5.1 Security Requirements

The following requirements apply for HTTP over TLS:

1. The HTTP client initiates the TLS connection to the HTTP server in the typical fashion.
2. If there are any TLS errors, whether "warning" or "fatal" or any other error level with the TLS connection, the HTTP client MUST terminate the connection without prompting the user. For example, this includes any errors found in certificate validity checking that HTTP clients employ, such as via TLS server identity checking [RFC6124], Certificate Revocation Lists (CRLs) [RFC4280], or via the Online Certificate Status Protocol (OCSP) [RFC2460].
3. Whenever comparisons are made between the presented TLS server identity (as presented during the TLS handshake, typically within the server certificate) and the intended source TLS server identity (e.g., as entered by a user, or embedded in a link), [RFC6124] server identity checking MUST be employed. The client MUST terminate the connection without prompting the user on any error condition.
4. The TLS server certificate MUST either be provisioned explicitly out-of-band (e.g. packaged with an app as a "pinned certificate") or be trusted by chaining to a root included in the certificate store of the operating system or a major browser by virtue of being currently in compliance with their root store program requirements. The client MUST terminate the connection without prompting the user if there are any error conditions when building the chain of trust.
5. The "anon" and "null" crypto suites are not allowed and insecure cryptographic algorithms in TLS (e.g. MD4, RC4, SHA1) SHOULD be avoided (see NIST SP800-131A).
6. The client and server SHOULD use the latest practicable TLS version.
7. The client MUST NOT follow HTTP redirects that are not within the same Origin. [RFC6454]
8. Otherwise, the TLS connection is now established and ready for use.

5.1.1 Insecure Mixed Content

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

The FIDO plugin MUST immediately throw an `InsecureTransportException` if any APIs defined in this document are invoked by a document context containing insecure mixed content.

5.1.2 XMLHttpRequest

This document describes an interoperability profile for UAF that can be executed by a ECMAScript operating in a document context inside a web browser and gives non-normative examples using the XMLHttpRequest object defined in [XHR]. A conformant implementation of this XMLHttpRequest with https URLs can be assumed to meet the TLS binding requirements of UAF.

If any non-standard features or alternative transports (e.g. plugin-based asynchronous HTTP requests or the Android HttpsURLConnection) are used by the UAF client-side web application, it must meet the requirements described above.

Android applications implementing FIDO for WebViews with addJavascriptInterface() MUST only add these interfaces to resources that meet these requirements.

6. Locating FIDO APIs

For Web applications, FIDO UAF APIs are rooted in a new `uaf` object, a property of a new `fido` object under to the `window.navigator` object, the existence and properties of which can be used for feature detection.

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

For Android apps, the FIDO Client is implemented as a Bound Service with the name `org.fidoalliance.uaf.FidoClient`. This contains a local class `FidoBinder` that implements `getServerInstance()` to return an instance of `org.fidoalliance.uaf.FidoClient`, as shown in the following example code. All other interfaces defined in this document are rooted in the `org.fidoalliance.uaf` package.

```
import org.fidoalliance.uaf.FidoClient;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;


public class FidoAwareApp extends Activity {
```

```
 boolean     mFidoClientBound;
 FidoClient  mFidoClient;

 @Override
 protected void onStart() {
  super.onStart();
  Intent intent = new Intent(this, FidoClient.class);
  bindService(intent, mConnection, BIND_AUTO_CREATE);
 };

 ServiceConnection mConnection = new ServiceConnection() {

  public void onServiceDisconnected(ComponentName name) {
   mFidoClientBound = false;
   mFidoClient = null;
  }

  public void onServiceConnected(ComponentName name, IBinder binder) {
   mFidoClientBound = true;
   mFidoClient = ((FidoClient.FidoBinder) binder).getServerInstance();
  }
 };

 @Override
 protected void onStop() {
  super.onStop();
  if(mFidoClientBound) {
   unbindService(mConnection);
   mFidoClientBound = false;
  }
 };
}
```

Once an Android app has obtained a `FidoClient` server instance, the additional inferfaces can be accessed through that object using the following Java-specific interface:

```
package org.fidoalliance.uaf;

public interface FidoClient {

    // equivalent to DOM api at window.navigator.fido.uaf.discovery
    Discovery discovery;

     // equivalent to DOM api at window.navigator.fido.uaf.registration
    Registration registration;

    // equivalent to DOM api at window.navigator.fido.uaf.authentication
    Authentication authentication;

    // equivalent to DOM api at window.navigator.fido.uaf.confirmation
    Confirmation confirmation;

    // equivalent to DOM api at window.navigator.fido.uaf.deregistration
    Deregistration deregistration;

    // used to notify the FIDO client of the results of a POST of
    // a UAF message, should be called to allow client to do
    // housekeeping for a good user experience.
    public void notifyUAFResult(String uafResult);
}
```

7. UAF API

The following interfaces and definitions are shared among several phases of the UAF protocol.

7.1 UAFClientMessage Dictionary

This dictionary allows arbitrary addtional information from the client web app to be included with the uafResponse protocol message in the POST body sent to the Relying Party server.

```
dictionary UAFClientMessage {

    DOMString uafResponse;
    Object    additionalData;

}
```

uafResponse of type DOMString
: This string contains the UAF protocol client response message that will be processed by the FIDO server at the Relying Party. UAF protocol messages are JSON objects. Their internal structure is described in [UAF_Protocol_Specification].

  In general, the client web application should not need to read or modify the `uafResponse`, and modifying cetain parts of the message will cause the transaction to fail. It is recommended that this field be treated as opaque by the client side web application.

additionalData
: This key allows the client web app to attach any additional data it wishes to the POST body sent to the Relying Party server.

7.2 UAFResponseCallback

A `UAFResponseCallback` is used to indicate the successful completion of asynchronous operations that invoke the FIDO client with a `uafChallenge`.

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

Arguments

`uafResponse` of type `DOMString`. The UAF protcol message representing the FIDO Authenticator and Client's response to the `uafChallenge`. This should be delivered to the Relying Party server by the client web application.

7.3 ErrorCallback

The `ErrorCallback` is used to return progress and error codes from asynchronous operations that invoke the FIDO client.

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

interface ErrorCode {

  const short NO_ERROR                  = 0;
  const short WAIT_USER_ACTION          = 1;
  const short INSECURE_TRANSPORT        = 2;
  const short USER_CANCELLED            = 3;
  const short UNSUPPORTED_VERSION       = 4;
  const short NO_SUITABLE_AUTHENTICATOR = 5;
  const short PROTOCOL_ERROR            = 6;
  const short UNKNOWN                   = 99;
}
```

7.3.1 ErrorCode Values

NO_ERROR
: No error condition encountered.

WAIT_USER_ACTION
: Waiting on user action to procede. (selecting an authenticator in the FIDO client user interface, performing local User-to-Authenticator authentication, or completing an enrollment step with an authenticator)

USER_CANCELLED
: The user declined any necessary part of the interaction to complete the registration or no registered authenticator exists for authenticate, confirm or deregister calls.

UNSUPPORTED_VERSION
: The `UAFProtocolMessage` indicates protocol versions not supported by this FIDO Client.

NO_SUITABLE_AUTHENTICATOR
: No authenticator matching the `AuthenticatorPolicy` internal to the UAFProtocolMessage is available to service the request, or the user declined to consent to the use of a suitable authenticator.

INSECURE_TRANSPORT
: `window.location.protocol` is not `https` or the DOM contains *insecure mixed content*.

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

UNKNOWN
: An error condition not described by other codes.

The `ErrorCallback` may be called multiple times, for example with the `WAIT_USER_ACTION`.

7.4 ServerResponse Interface

In response to a Registration or Authentication message, the Relying Party server may return additional information to the client web application, some of which can be handled locally in an application-specific manner, some of which SHOULD be delivered to the FIDO client.

```
interface ServerResponse {

    readonly int        statusCode;
    readonly DOMString  description;
    readonly Token[]    tokens;
    readonly DOMString  location;
    readonly DOMString  postData;

    readonly DOMString uafResult;

    interface Token {
      enum TokenType {
        "HTTP_COOKIE",
        "OAUTH",
        "OAUTH2",
        "SAML1_1",
        "SAML2",
        "JWT"
      };

      readonly TokenType type;
      readonly DOMString value;

    }
}
```

7.4.1 Attributes

statusCode of type int, readonly
: The FIDO UAF response status code.

description of type DOMString, readonly
: A detailed message describing the status code and providing additioanl information.

tokens of type ServerResponse.Token[], readonly
: New authentication or authorization token(s) for the client not natively handled by the HTTP transport. Tokens SHOULD be processed prior to processing of `location`.

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

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

uafResult of type DOMString, readonly
: This is the UAF protocol message indicating the result of the processing of a `UAFResponse` message. It should be returned to the FIDO Client using the supplied `UAFResultCallback`

7.4.2 Interface Token

type of type TokenType, readonly
: The type of the additional authentication / authorization token.

value of type DOMString, readonly
: The value of the additional authentication / authorization token.

enum TokenType

HTTP_COOKIE
: If the user agent is a standard web browser or other HTTP native client with a cookie store, this

`TokenType` SHOULD NOT be used. Cookies should be set directly with the `Set-Cookie` HTTP header. For non- HTTP or non-browser contexts this indicates a token indended to be set as an HTTP cookie. [RFC 6525] (for example, a native VPN client on Microsoft Windows that authenticates with UAF might use this `TokenType` to add a cookie to the browser cookie jar.)

OAUTH
: Indicates that the token is of type OAUTH as defined in [RFC ????].

OAUTH2
: Indicates that the token is of type OAUTH2 as defined in [RFC ????].

SAML1_1
: Indicates that the token is of type SAML 1.1 as defined in [OASIS ????].

SAML2
: Indicates that the token is of type SAML 2.0 as defined in [OASIS ?????]

JWT
: Indicates that the token is of type JSON Web Token (JWT) as defined in [draft-jones-jwt-????]

7.4.3 ServerResponse Status Codes

This table lists UAF protocol status codes. These indicate the result of the transaction at the FIDO server, not the Relying Party's HTTP layer. These codes are intended for consumption by the client-side web app to inform error messaging or retry behavior.

Table 1

Code	Meaning
200	OK. Registration completed
202	Accepted. Registration message accepted, but not completed at this time. The RP may need time to process the attestation, run risk scoring, etc. The server SHOULD NOT send an authenticationToken with a 202 response
400	Bad Request. The server did not understand the message
401	Unauthorized. The userid must be authenticated to register a FIDO Authenticator, or this KeyID is not associated with this UserID.
403	Forbidden. The userid is not allowed to register a FIDO Authenticator. Client SHOULD NOT retry
408	Request Timeout
480	Unknown AAID. The server was unable to locate authoritative metadata for the AAID.
481	Unknown KeyID. The server was unable to locate a registration for the given UserID and KeyID combination.
490	Channel Binding Refused. The server refused to service the request due to a missing or mismatched channel binding(s).

Code	Meaning
491	Server Challenge Invalid. The server refused to service the request because the challenge was unknown, expired or the server has previously serviced a message with the same challenge and user ID.
492	Unacceptable Authenticator. The authenticator is not acceptable according to the server's policy, for example because the capability registry used by the server reported different capabilities than client-side discovery.
493	Revoked Authenticator. The authenticator is considered revoked by the server.
494	Unacceptable Key. The key being registered is unacceptable. Perhaps it is on a list of known weak keys or uses insecure parameter choices.
495	Unacceptable Algorithm. The server believes the authenticator to be capable of using a stronger mutually-agreeable algorithm than was presented in the request.
496	Unacceptable Attestation. The attestation(s) provided were not accepted by the server for registration.
497	Unacceptable Client Capabilities. The server was unable or unwilling to use required capabilities provided supplementally to the authenticator by the client software.
498	Unacceptable Content. There was a problem with the contents of the message and the server was unwilling or unable to process it.
500	Internal Server Error

7.5 Authenticator Interface

```
interface Authenticator {

    readonly attribute DOMString AAID;
    readonly attribute long userVerification;
    readonly attribute long keyProtection;
    readonly attribute long attachmentHint;
    readonly attribute long secureDisplay;

    readonly attribute DOMString authenticationSuite;
    readonly attribute DOMString scheme;

    // for future use
    readonly attribute long additionalInfo;


    const long USER_VERIFY_PRESENCE    =  0x01;
    const long USER_VERIFY_FINGERPRINT =  0x02;
    const long USER_VERIFY_PIN         =  0x04;
    const long USER_VERIFY_VOICEPRINT  =  0x08;
    const long USER_VERIFY_FACEPRINT   =  0x10;
    const long USER_VERIFY_LOCATION    =  0x20;
    const long USER_VERIFY_EYEPRINT    =  0x40;
```

```
    const long USER_VERIFY_PATTERN     =  0x80;
    const long USER_VERIFY_HANDPRINT   = 0x100;
    const long USER_VERIFY_NONE        = 0x200;


    const long KEY_PROTECTION_SOFTWARE         = 0x01;
    const long KEY_PROTECTION_HARDWARE         = 0x02;
    const long KEY_PROTECTION_TEE              = 0x04;
    const long KEY_PROTECTION_SECURE_ELEMENT   = 0x08;
    const long KEY_PROTECTION_REMOTE_HANDLE    = 0x10;


    const long ATTACHMENT_HINT_INTERNAL   = 0x01;
    const long ATTACHMENT_HINT_EXTERNAL   = 0x02;
    const long ATTACHMENT_HINT_WIRED      = 0x04;
    const long ATTACHMENT_HINT_WIRELESS   = 0x08;
    const long ATTACHMENT_HINT_NFC        = 0x10;
    const long ATTACHMENT_HINT_BLUETOOTH  = 0x20;
    const long ATTACHMENT_HINT_NETWORK    = 0x40;
    const long ATTACHMENT_HINT_READY      = 0x80;


    const long SECURE_DISPLAY_ANY                    = 0x01;
    const long SECURE_DISPLAY_PRIVILEGED_SOFTWARE    = 0x02;
    const long SECURE_DISPLAY_HARDWARE               = 0x04;
    const long SECURE_DISPLAY_TEE                    = 0x08;
    // bhill - I still think we need this one...
    const long SECURE_DISPLAY_REMOTE                 = 0x10;

}
```

7.5.1 Constants

USER_VERIFY_PRESENCE
: This flag will be set if the authenticator is able to confirm user presence in any fashion. If this flag and no other is set for the `userVerification`, the guarantee is only that the authenticator cannot be operated without some human intervention, not necessarily that the presence verifiction provides any level of authentication of the human's identity. (e.g. a device that requires a touch to activate)

USER_VERIFY_FINGERPRINT
: This flag will be set if the authenticator uses any type of measurement of a fingerprint for User-to-Authenticator authentication.

USER_VERIFY_PIN
: This flag will be set if the authenticator uses a PIN code for User-to-Authenticator authentication.

USER_VERIFY_VOICEPRINT
: This flag will be set if the authenticator uses a voiceprint for User-to-Authenticator authentication.

USER_VERIFY_FACEPRINT
: This flag will be set if the authenticator uses any manner of face recognition to locally authenticate the user.

USER_VERIFY_LOCATION
: This flag will be set if the authenticator uses any form of location sensor or measurement for User-to-Authenticator authentication and/or returns a location measurement to the Relying Party as an additional user verification.

USER_VERIFY_EYEPRINT
: This flag will be set if the authenticator uses any form of eye biometrics for User-to-Authenticator authentication.

USER_VERIFY_PATTERN
: This flag will be set if the authenticator uses a drawn pattern for User-to-Authenticator authentication.

USER_VERIFY_HANDPRINT

This flag will be set if the authenticator uses any measurement of a full hand (including palmprint, hand geometry or vein geometry) for User-to-Authenticator authentication.

USER_VERIFY_NONE
: This flag will be set if the authenticator will respond without any user interaction.

KEY_PROTECTION_SOFTWARE
: This flag will be set if the authenticator uses software-based key management.

Exclusive with KEY_PROTECTION_HARDWARE, KEY_PROTECTION_TEE, KEY_PROTECTION_SECURE_ELEMENT

KEY_PROTECTION_HARDWARE
: This flag will be set if the authenticator uses hardware-based key management.

Exclusive with KEY_PROTECTION_SOFTWARE

KEY_PROTECTION_TEE
: This flag will be set if the authenticator uses the Trusted Execution Environment [ref:TODO] for key management.

Exclusive with KEY_PROTECTION_HARDWARE, KEY_PROTECTION_SOFTWARE, KEY_PROTECTION_SECURE_ELEMENT

KEY_PROTECTION_SECURE_ELEMENT
: This flag will be set if the authenticator uses a Secure Elemlent [ref:TODO] for key management.

Exclusive with KEY_PROTECTION_HARDWARE, KEY_PROTECTION_TEE, KEY_PROTECTION_SOFTWARE

KEY_PROTECTION_REMOTE_HANDLE
: This flag will be set if the authenticator does not store per-Origin keys locally, but relies on a server-provided `key handle`.

  This flag MUST be set in conjunction with one of the other KEY_PROTECTION flags to indicate how local key unwrapping operations are protected.

  Servers can use this value in authenticator policy to exclude such devices if they are unprepared to store and return key handles, for example, if they have a requirement to respond indistinguishably to authentication attempts for userIDs that do and do not exist.

ATTACHMENT_HINT_INTERNAL
: This flag indicates that the authenticator is permanently attached to the system on which the FIDO client software is running.

  A device such as a smartphone may have authenticator functionality that is able to be used both locally and remotely. In such a case, the device metadata may have both of these flags set, but the FIDO client MUST filter and exclusively report only the relevant bit during Discovery and when performing policy matching.

ATTACHMENT_HINT_EXTERNAL
: This flag indicates, for a hardware-based authenticator, that it is removable or remote from the system on which the FIDO client software is running.

  A device such as a smartphone may have authenticator functionality that is able to be used both locally and remotely. In such a case, the device metadata may have both of these flags set, but the FIDO client MUST filter and exclusively report only the relevant bit during Discovery and when performing policy matching.

ATTACHMENT_HINT_WIRED
: Indicates that an external authenticator currently has a wired connection to the FIDO client software.

ATTACHMENT_HINT_WIRELESS
: Indicates that an external authenticator is able to communicate wirelessly to the FIDO client software.

ATTACHMENT_HINT_NFC
: Indicates that an external authenticator is able to communicate by NFC to the FIDO client software.

ATTACHMENT_HINT_BLUETOOTH

Indicates that an external authenticator is able to communicate using Bluetooth to the FIDO client software.

ATTACHMENT_HINT_NETWORK
: Indicates that an external authenticator is able to communite to the FIDO client software over a packet-switched network.

ATTACHMENT_HINT_READY
: Indicates that an external authenticator is in a ready state. e.g. a Bluetooth conencted device that is currently paired and connected or a USB device that is plugged in.

SECURE_DISPLAY_ANY
: This flag indicates that some form of secure display is available on this authenticator.

SECURE_DISPLAY_PRIVILEGED_SOFTWARE
: This flag indicates that a software-based secure display operating in a privileged context is available on this authenticator.

  Software-based displays are typically provided by the FIDO client software rather than the authenticator itself. A FIDO client that is capable of providing this capability MAY set this bit for all authenticators of type `ATTACHMENT_HINT_INTERNAL`, even if the authoritative metadata for the authenticator does not indicate this capability.

  A FIDO client that is capable of providing this capability MUST NOT set this bit for authenticators of type `ATTACHMENT_HINT_EXTERNAL` as it may not be uniformly available at all systems to which the authenticator may roam.

SECURE_DISPLAY_TEE
: This flag indicates that the authenticator implements a secure display in the Trusted Execution Environment. [ref: TODO]

SECURE_DISPLAY_HARDWARE
: This flag indicates that an integrated and distinct hardware secure display is available on this authenticator.

SECURE_DISPLAY_REMOTE
: This flag indicates that the secure display is provided on a distinct device from the system the FIDO client software is operating on.

7.5.2 Attributes

AAID of type DOMString, readonly
: The Authenticatior Attestation ID, which identifies the type and batch of the authenticator.

  No exceptions.

userVerification of type long, readonly
: A set of bit flags indicating the user verification methods(s) supported by the authenticator. The values are defined by the `USER_VERIFY_` constants.

  No exceptions.

keyProtection of type long, readonly
: A set of bit flags indicating the key PROTECTION used by the authenticator. The values are defined by the `KEY_PROTECTION_` constants.

  No exceptions.

attachmentHint of type long, readonly
: A set of bit flags indicating how they are currently connected to the system hosting the FIDO client software. The values are defined by the `ATTACHMENT_HINT` constants.

  Because the connection state and topology of an authenticator may be transient, these values are only hints that can be used by server-supplied policy to guide the user experience, e.g. to prefer a device that is connected and ready for authenticating or confirming a l ow-value transaction, rather than one that is more secure but requires more user effort. These values are not reflected in authenticator metadata and cannot be relied on by the relying party, although some models of authenticator may provide attested measurements of similar data as part of OSTP response

messages. *No exceptions.*

secureDisplay of type long, readonly
: A set of bit flags indicating the availability and type of secure display. The values are defined by the `SECURE_DISPLAY_` constants.

  No exceptions.

authenticationSuite of type DOMString, readonly
: Indicates the authentication suite the authenticator uses.

  Authentication suite identifiers are defined in the OSTP specification. [ref]

  No exceptions.

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

  Scheme identifiers are defined in the OSTP specification. [ref]

  No exceptions.

additionalInfo of type DOMString, readonly
: RESERVED FOR FUTURE USE

  No exceptions.

7.6 notifyUAFResult Operation

A `notifyUAFResult` call is used to indicate to the FIDO Client the completion status of a `UAFResponse` delivered to the Relying Party server. Applications SHOULD make this call to allow the client to do housekeeping for a better user experience.

```
    void UAFResultCallback(DOMString uafResult);
```

Arguments

`uafResult` of type `DOMString`. The UAF protcol message representing the result of the Relying Party server's processing of a `uafResponse`.

8. Discovery

To discover if the user's client software and devices support UAF and if Authenticator capabilities are available that it may be willing to accept for authentication, Relying Party code in the browser can use the following interface.

8.1 Discovery Interface

```
interface Discovery {

    readonly attribute long[][] version
```

```
    readonly attribute DOMString clientVendor;
    readonly attribute long[] clientVersion;
    readonly attribute Authenticator[] availableAuthenticators;

    void checkPolicy(DOMString uafChallenge, ErrorCallback cb);

}
```

8.1.1 Attributes

version of type long[], readonly
: An array of arrays. The first index is the list of the FIDO UAF protocol versions supported by the client, most- preferred first.

  The second index is always a three element array indicating the detailed version supported by the FIDO UAF Client. The first element indiates the major version, the second the minor version.

  No exceptions.

clientVendor of type DOMString, readonly
: The vendor of the FIDO UAF Client.

  No exceptions.

clientVersion of type long[], readonly
: The version of the FIDO UAF Client, ordered by version number significance.

  No exceptions.

availableAuthenticators of type Authenticator[], readonly
: An array containing `Authenticator` dictionaries describing the available UAF authenticators. The order is not significant.

  No exceptions.

8.1.2 Operations

checkPolicy(DOMString uafChallenge, ErrorCallback cb) of return type void
: Asks the FIDO plugin if it would be able to process the supplied UAF server challenge message based on cached state only. Unlike other operations using an `ErrorCallback`, this operation MUST always trigger the callback and return `NO_ERROR` if it belives that the message can be processed and a suitable authenticator matching the embedded policy is available, or the appropriate `ErrorCode` value otherwise. Because this call should use cached information only, it should not incur a potentially disrupting context-switch even if the FIDO client is implemented out-of-process.

8.2 Example

This section is non-normative.

A Relying Party could employ a client-side web application similar to the following to do FIDO UAF Discovery and show a message to the user that they are eligible to register a FIDO Authenticator with their account if an authenticator with the required capabilities is available.

```
<html>
```

```
<head></head>
<body>
<script>
      var uaf = window.navigator.fido.uaf;

      if(!!uaf) // check if UAF is available in user agent
      {

        var disco = uaf.Discovery;
        var aa    = disco.availableAuthenticators;
        var done  = false;

        // get a server challenge message, this is app specific...
        var uafChallenge = getRegistrationChallenge();

        disco.checkPolicy(uafChallenge,
                          function(ec) {
                            if(ec != 0) { // if no error, show the registration div
                                document.getElementById("fidoDiv").style.visibility =
"visible";
                                done = true;
                            }
                          }
                         );
      }

</script>
<div id="fidoDiv" style="visibility:hidden">
 <a onclick="beginFIDOReg()">
   Click here to register your account with FIDO!
 </a>
</div>
</body>
</html>
```

The following Java example code demonstrates the same functionality for an Android app.

```
import org.fidoalliance.uaf.FidoClient;
import org.fidoalliance.uaf.Discovery;
import org.fidoalliance.uaf.ErrorCallback;
import org.fidialliance.uaf.ErrorCallback.ErrorCode;

import java.lang.String;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;


public class FidoAwareApp extends Activity {

 boolean     mFidoClientBound;
 FidoClient  mFidoClient;

 @Override
 protected void onStart() {
  super.onStart();
  Intent intent = new Intent(this, FidoClient.class);
  bindService(intent, mConnection, BIND_AUTO_CREATE);
```

```
};

ServiceConnection mConnection = new ServiceConnection() {

 public void onServiceDisconnected(ComponentName name) {
  mFidoClientBound = false;
  mFidoClient = null;
 }

 public void onServiceConnected(ComponentName name, IBinder binder) {
  mFidoClientBound = true;
  mFidoClient = ((FidoClient.FidoBinder) binder).getServerInstance();

  Discovery disco = mFidoClient.discovery;

  // implementation to get a server challenge an exercise for the reader
  String uafChallenge = getServerChallenge();

  discoCheckPolicy(uafChallenge,
                   new ErrorCallback() {
                           public void call(ErrorCode code) {
                               if(code == ErrorCode.NO_ERROR) {
                                   // implementation of register
                                   doFidoRegistration(mFidoClient, uafChallenge);
                               } else if (code == ErrorCode.WAIT_USER_ACTION) {
                                  // shouldn't happen, but do nothing
                               } else {
                                   // fallback to user/pass
                                   doUserPassRegistration();
                               }
                           };
                       };
                   );

   }
};

@Override
protected void onStop() {
 super.onStop();
 if(mFidoClientBound) {
  unbindService(mConnection);
  mFidoClientBound = false;
 }
};
}
```

9. Registration

In the Registration phase, a user who is newly creating or has already demonstrated control of an account through a non-UAF authentication scheme (presumably username + password, but possibly with additional or alternate factors) can associate a FIDO UAF Authenticator with their account.

This section is non-normative. Web applications might make distinct HTTP request/response cycles to deliver client-side code and data implementing the Discovery and Registration phase, but it is expected that both actions will be able to be performed in a single HTTP round-trip. For example, the response to

a successful legacy username + password authentication might contain all the code and information necessary to both dicover the availability of suitable authenticator(s) as well as the fine grained registration policy and server challenge information needed to immediately get a `RegistrationResponseMessage` ready to be sent asynchronously to the server.

9.1 Registration Interface

```
interface Registration {

    void register(DOMString           uafChallenge,
                  UAFResponseCallback completionCallback,
                  ErrorCallback       errorCallback);

}
```

9.1.1 Operations

register(DOMString uafChallenge, UAFResponseCallback completionCallback, ErrorCallback errorCallback) of return type void

Invokes the registration logic in the FIDO Client and prompts the user to register an authenticator(s) matching the provided policy, using the supplied challenge, and returning to the callback a message in one of the supported protocol versions indicated by the `uafChallenge`.

Arguments

uafChallenge of type DOMString
: The UAF Server Challenge message to be delivered to the FIDO client software.

completionCallback of type RegistrationCallback
: The callback that receives the UAF Client Response message from the FIDO client software, to be delivered to the Relying Party server.

errorCallback of type ErrorCallback
: A callback function to recieve error and progress events from the FIDO Client.

checkOnly of type boolean, optional
: This defaults to 'false'. If set to 'true', this instructs the FIDO client to evaluate the operation using it's cached

9.2 Example

This section is non-normative.

A Relying Party could employ a client-side web application similar to the following to do FIDO UAF Discovery and show a message to the user that they are eligible to register a FIDO Authenticator with their account if an authenticator with the required capabilities is available.

```
<html>
<head></head>
<body>
<script>
    function registerFIDO() {
```

```
      var fidoReg = window.navigator.uaf.Registration;

      function deliverResponse(uafResponse, jsonCallback) {

        var clientMessage = {};
        clientMessage.uafResponse = uafResponse;
        clientMessage.additionalData = {};

        // TODO: do not follow off-origin redirects!
        var client = new XMLHttpRequest();
        client.onreadystatechange = function () {
            if(this.readyState == this.DONE) {
                  jsonCallback(JSON.parse(this.responseText));
             }
        };


        client.setRequestHeader("Content-Type", "application/fido.uaf+json;
charset=utf-8");
        client.setRequestHeader("Accept", "application/fido.uaf+json");
        client.open("POST", "https://fido.example.com/register");
        client.send(clientMessage);


      };

      function completionCallback(uafResponse) {

         // invoke XHR to deliver the response from the FIDO client
         // back to the server
         deliverResponse(uafResponse, function(jsonResults) {

               // return the result to the FIDO client for housekeeping
               window.navigator.fido.uaf.notifyUAFResult(jsonResults.uafResult);

               // invoke app-specific logic to process the web-app layer
               // result semantics (new auth tokens, redirects, etc.)
               handleResult(jsonResults);
           };
         );

      };

      function errorCallback(code) {
           if(code == code.WAIT_USER_ACTION) {
               // do nothing...
           }
           else {
              // app specific error handling...
             logError(code);
           }
      };

      // get a server challenge message, this is app specific...
      var serverChallenge = getRegistrationChallenge();

      // call the register operation
      fidoReg.register(serverChallenge.uafChallenge, completionCallback,
errorCallback);


    };
</script>
```

```
   <button onClick="registerFIDO()">Register your FIDO Authenticator</button>
</body>
</html>
```

The following Java example code demonstrates equivalent functionality for an Android app.

```
/** only partial class implemntation shown; implicit imports:
* import org.fidoalliance.uaf.FidoClient;
* import org.fidoalliance.uaf.ErrorCallback;
* import org.fidoalliance.uaf.ErrorCallback.ErrorCode;
* import org.fidoalliance.uaf.UAFResponseCallback;
* import org.fidoalliance.uaf.UAFResultCallback;
* import java.net.*;
* import java.io.*;
* import javax.net.ssl.HttpsUrlConnection;
* import org.apache.commons.io.IOUtils;
* import org.json.JSONObject;
**/

public void doFidoRegistration(FidoClient fidoClient, String uafChallenge) {

    // inner class implementation of UAFResponseCallback handler
    class RegCompletionCallback implements UAFResponseCallback {

        public void call(String uafResponse) {

            URL url = new URL("https://fido.example.com/register");
            HttpsUrlConnection huc = new HttpsUrlConnection(url);
            huc.setFollowRedirects(false);
            huc.addRequestProperty("Content-Type", "application/fido.uaf+json;
charset=utf-8");
            huc.addRequestPrpoerty("Accept", "application/fido.uaf+json");
            huc.setRequestMethod("POST");
            huc.setDoInput(true);
            huc.setDoOutput(true);

            OutputStreamWriter out = new OutputStreamWriter(
                                        huc.getOutputStream());

            // build the JSON structure with (in this case no) additional client data
            String clientMessage = "{";
            clientMessage += "uafResponse : {" + uafResponse;
            clientMessage += "}, additionalData: {} }";

            out.write(URLEncoder.encode(clientMessage, "UTF-8");
            out.close();

            String response = IOUtils.toString(huc.getInputStream(), "UTF-8");
            JSONObject jsonResults = new JSONObject(response);

            fidoClient.notifyUAFResult(jsonResults.getString("uafResult"));

            // invoke app-specific logic to process the web-app layer
            // result semantics (new auth tokens, redirects, etc.)
            handleResults(jsonResults);

            }

        };
```

```
    };

    // inner class implemenation of ErroCallback handler
    class RegErrorCallback implements ErrorCallback {
        public void call(ErrorCode code) {
            if(code == ErrorCode.WAIT_USER_ACTION
            || code == ErrorCode.NO_ERROR){
                // do nothing
            }
            else {
                // app specific error handling...
                logError(code);
            }
        };
    }

    fidoClient.Registration.register(uafChallenge,
                                     new RegCompletionCallback(),
                                     new RegErrorCallback());

}
```

10. Authentication

In the Authentication phase, a user who has previously completed Registration can Authenticate to the Relying Party with a registered authenticator.

This section is non-normative. Like the Registration phase, Authentication is policy-driven. A Relying Party may ask a user to authenticate in order to access personal account information, to authorize a transaction, to access or change sensitive information, or to confirm a transaction, and the Relying Party may have different requirements about which registered authenticators are acceptable, and in what combination, for each circumstance.

10.1 Authentication Interface

```
interface Authentication {

    void authenticate(DOMString           uafChallenge,
                      UAFResponseCallback completionCallback,
                      ErrorCallback       errorCallback);

}
```

10.1.1 Operations

authenticate(DOMString uafChallenge, UAFResponseCallback completionCallback, ErrorCallback errorCallback) of return type void

> Invokes the authentication logic in the FIDO Client and prompts the user to register an authenticator(s) matching the provided policy, using the supplied challenge, and returning to the callback a message in one of the supported protocol versions indicated by the `uafChallenge`.

Arguments

uafChallenge of type DOMString
: The UAF Server Challenge message to be delivered to the FIDO client software.

completionCallback of type RegistrationCallback
: The callback that receives the UAF Client Response message from the FIDO client software, to be delivered to the Relying Party server.

errorCallback of type ErrorCallback
: A callback function to recieve error and progress events from the FIDO Client.

checkOnly of type boolean, optional
: This defaults to 'false'. If set to 'true', this instructs the FIDO client to evaluate the operation using it's cached

11. Transaction Confirmation

In the Transaction Confirmation phase, a Relying Party can send text to a user with a registered authenticator and request the user confirm the text by authenticating. This phase is essentially identical to Authentication but with the addition of the transactional text to be displayed and confirmed. The FIDO Client is responsble for displaying the text in a secure manner, the client web app SHOULD NOT display the transactional text.

This section is non-normative. Like other phases, Transaction Confirmation is policy-driven. A Relying Party may ask a user to confirm transaction, changes of sensitive information, or receipt of a message and the Relying Party may have different requirements about which registered authenticators are acceptable, and in what combination, for each circumstance.

11.1 Confirmation Interface

```
interface Confirmation {

    void confirm(DOMString           uafChallenge,
                 UAFResponseCallback completionCallback,
                 ErrorCallback       errorCallback);

}
```

11.1.1 Operations

confirm(DOMString uafChallenge, UAFResponseCallback completionCallback, ErrorCallback errorCallback) of return type void

Invokes the transaction confirmation logic in the FIDO Client and prompts the user to register an authenticator(s) matching the provided policy, using the supplied challenge, and returning to the callback a message in one of the supported protocol versions indicated by the `uafChallenge`.

Arguments

uafChallenge of type DOMString
: The UAF Server Challenge message to be delivered to the FIDO client software.

completionCallback of type RegistrationCallback
: The callback that receives the UAF Client Response message from the FIDO client software, to be delivered to the Relying Party server.

errorCallback of type ErrorCallback

A callback function to recieve error and progress events from the FIDO Client.

checkOnly of type boolean, optional

This defaults to 'false'. If set to 'true', this instructs the FIDO client to evaluate the operation using it's cached

11.2 Example

This section is non-normative.

A Relying Party could employ a client-side web application similar to the following to do FIDO UAF Transaction Confirmation.

```
<html>
<head>…</head>
<body>
<script>
    // TODO
</script>
   <!-- TODO -->
</body>
</html>
```

12. Deregistration

In the Deregistration phase, a Relying Party can instruct the FIDO client to delete a registration(s) at the specified authenticator(s). Unlike the other phases of UAF, a deregistration is a one-way, server-to-client message.

This section is non-normative. Because key registrations in UAF are always Relying Party-specific, the Relying Party may declare them as invalid at any time. Deregistration is primarily a 'housekeeping' function to improve the user experience by avoiding prompting to authenticate with keys that the Relying Party no longer considers valid.

12.1 Deregistration Interface

```
interface Deregistration {

    void deregister(DOMString uafChallenge);

}
```

12.1.1 Operations

deregister(DOMString uafChallenge) of return type void

Invokes the deregistration logic in the FIDO Client and prompts the user to register an authenticator(s) matching the provided policy, using the supplied challenge, and returning to the callback a message in one of the supported protocol versions indicated by the `uafChallenge`.

Arguments
uafChallenge of type DOMString
The UAF Server Challenge message to be delivered to the FIDO client software.

12.2 Example

This section is non-normative.

A Relying Party could employ a client-side web application similar to the following to do FIDO UAF Deregistration.

```
<html>
<head>…</head>
<body>
<script>
    // TODO
</script>
   <!-- TODO -->
</body>
</html>
```

13. Delivery of the uafResponse

After calling `register()`, `authenticate()`, or `confirm()`, it is the responsiblity of the Relying Party client-side web application to deliver the `uafResponse` returned to the `UAFResponseCallback` back to the Relying Party server to complete the operation. This is typically accomplished in an asynchronous manner using the `XMLHttpRequest` object.

When sending the message to the server, the following conditions apply:

1. The HTTP Method MUST be POST
2. A `UAFClientMessage` dictionary MUST comprise the entire POST body.
3. The `Content-Type` HTTP header MUST be "`application/fido.uaf+json; charset=utf-8`"
4. The `Accept` HTTP header MUST include "`application/fido.uaf+json`"
5. The server SHOULD maintain its own notion of the user's authenticated identity associated with the `uafChallenge`, but when possible the request SHOULD include the user's original cookies and other ambient autority to assist the server in verifying the registration. For `XMLHttpRequest`, this means that the `Anonymous` flag SHOULD NOT be set.

The Relying Party server SHOULD, after processing the request, return an HTTP response with a Content-Type of `application/fido.uaf+json` of which the entire HTTP Response Body is a JSON representation of a `ServerResponse`. Processing of the `ServerResponse` is at the discretion of the client web application, but the `UAFResultCallback` supplied by the FIDO Client SHOULD be called with the `uafResult`.

The Relying Party server MAY also inlcude additional information to be processed by the browser's HTTP

client stack, such as the Set-Cookie HTTP header to provide additional authentication or authorization context for subsequent operations.

14. Considerations for Cross-Origin Requests

In some circumstances, a client-side web application may be able to interact asynchronously over HTTP with servers other than the application's own Origin, for example by using CORS [ref CORS@W3C] or Adobe(R) Flash. [ref crossdomain.xml] These capabilities will generally not allow for UAF to be used cross-origin because the FIDO client software will receive the Origin of the executing Document context from the plugin and the Origin of the Relying Party server inside the OSTP message and terminate the protocol if a mismatch is detected.

15. Extensibility

This section is non-normative. So long as they meet the requirements in the security and privacy considerations sections and the relevant overall security and privacy requirements of FIDO, FIDO clients may add to the interface.

Non-standard additions SHOULD be vendor-prefixed, e.g.

```
readonly attribute DOMString vendorName-extendedAttrName;
```

The `additionalData` field of the `UAFClientMessage` also can serve as an extensibility point between the client web app and Relying Party server.

16. Security Considerations

This section is non-normative. It is RECOMMENED that Relying Parties prevent resources that use the FIDO UAF API from being displayed in an `iframe` by a different Origin, to avoid confusing users who might rely on the browser's address bar to determine with which Relying Party they are interacting. This can be accomplished by setting the following HTTP headers:

```
X-Frame-Options: SAMEORIGIN
Content-Security-Policy: frame-options 'self'
```

This protection is most important during Registration. In some circumstances, it may be acceptable to invoke other phases of the UAF in a cross-origin context. Relying Parties that choose to expose endpoints cross-origin should be careful to restrict the semantics to operations that make sense. (e.g. allowing an Authentication phase resource to be embedded cross-origin for making payments, but not allowing a resource that allows changing an address to be so embedded)

An error-free TLS connection and Document context free of insecure mixed content MUST be established before sending any OSTP protocol messages, as defined in the TLS Binding section.

17. Privacy Considerations

This section is non-normative. Differences in the FIDO capabilities of a Web User Agent may (among many other characteristics) allow for a remote server to "fingerprint" a remote client and attempt to

persistently identify it in the absence of any explicit session state maintenance mechanism. Although it may contribute some amount of signal to servers attempting to fingerprint clients, the attributes exposed by the FIDO UAF Disocvery API are designed to have a large anonymity set size and should present little or no qualitatively new privacy risk. Nonetheless, an unusual configuration of FIDO Authenticators may be sufficient to uniquely identify a user. It is recommended that user agents expose the Discovery API to all applications without requiring explicit user consent by default, but user agents or FIDO Client implementers should provide users with the means to opt-out of discovery if they wish to do so for privacy reasons.

18. Normative References

Author's Address

Brad Hill HillBradPayPal, Inc.EMail: bhill@paypal.comURI: http://fidoalliance.org/