Table of Contents

IMPLEMENTATION DRAFT

FIDO UAF Architectural Overview

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-overview-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-overview-v1.0-ps-20141208.html
Editors:
Salah Machani, RSA, the Security Division of EMC
Rob Philpott, RSA, the Security Division of EMC
Sampath Srinivas, Google, Inc.
John Kemp, FIDO Alliance
Jeff Hodges, PayPal, Inc.

Abstract

The FIDO UAF strong authentication framework enables online services and websites, whether on the open Internet or within enterprises, to transparently leverage native security features of end-user computing devices for strong user authentication and to reduce the problems associated with creating and remembering many online credentials. The FIDO UAF Reference Architecture describes the components, protocols, and interfaces that make up the FIDO UAF strong authentication ecosystem.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Introduction

This section is non-normative.

This document describes the FIDO Universal Authentication Framework (UAF) Reference Architecture. The target audience for this document is decision makers and technical architects who need a high-level understanding of the FIDO UAF strong authentication solution and its relationship to other relevant industry standards.

The FIDO UAF specifications are as follows:

- FIDO UAF Protocol
- FIDO UAF Application API and Transport Binding
- FIDO UAF Authenticator Commands
- FIDO UAF Authenticator-Specific Module API
- FIDO UAF Registry of Predefined Values
- FIDO UAF APDU

The following additional FIDO documents provide important information relevant to the UAF specifications:

- FIDO AppID and Facets Specification
- FIDO Metadata Statements
- FIDO Metadata Service
- FIDO Registry of Predefined Values
- FIDO ECDAA Algorithm
- FIDO Security Reference
- FIDO Glossary

These documents may all be found on theFIDO Alliance website at http://fidoalliance.org/specifications/download/

1.1 Background

This section is non-normative.

The FIDO Alliance mission is to change the nature of online strong authentication by:

- Developing technical specifications defining open, scalable, interoperable mechanisms that supplant reliance on passwords to securely authenticate users of online services.
- Operating industry programs to help ensure successful worldwide adoption of the specifications.
- Submitting mature technical specifications to recognized standards development organization(s) for formal standardization.

The core ideas driving the FIDO Alliance's efforts are 1) ease of use, 2) privacy and security, and 3) standardization. The primary objective is to enable online services and websites, whether on the open Internet or within enterprises, to leverage native security features of end-user computing devices for strong user authentication and to reduce the problems associated with creating and remembering many online credentials.

There are two key protocols included in the FIDO architecture that cater to two basic options for user experience when dealing with Internet services. The two protocols share many of underpinnings but are tuned to the specific intended use cases.

Universal Authentication Framework (UAF) Protocol

The UAF protocol allows online services to offer password-less and multi-factor security. The user registers their device to the online service by selecting a local authentication mechanism such as swiping a finger, looking at the camera, speaking into the mic, entering a PIN, etc. The UAF protocol allows the service to select which mechanisms are presented to the user.

Once registered, the user simply repeats the local authentication action whenever they need to authenticate to the service. The user no longer needs to enter their password when authenticating from that device. UAF also allows experiences that combine multiple authentication mechanisms such as fingerprint + PIN.

This document that you are reading describes the UAF reference architecture.

Universal 2nd Factor (U2F) Protocol

The U2F protocol allows online services to augment the security of their existing password infrastructure by adding a strong second factor to user login. The user logs in with a username and password as before. The service can also prompt the user to present a second factor device at any time it chooses. The strong second factor allows the service to simplify its passwords (e.g. 4-digit PIN) without compromising security.

During registration and authentication, the user presents the second factor by simply pressing a button on a USB device or tapping over NFC. The user can use their FIDO U2F device across all online services that support the protocol leveraging built-in support in web browsers.

Please refer to the FIDO website for an overview and documentation set focused on the U2F protocol.

1.2 FIDO UAF Documentation

This section is non-normative.

To understand the FIDO UAF protocol, it is recommended that new audiences start by reading this architecture overview document and become familiar with the technical terminology used in the specifications (the glossary). Then they should proceed to the individual UAF documents in the recommended order listed below.

- **FIDO UAF Overview**: This document. Provides an introduction to the FIDO UAF architecture, protocols, and specifications.
- **FIDO Technical Glossary**: Defines the technical terms and phrases used in FIDO Alliance specifications and documents.
- **Universal Authentication Framework (UAF)**
  - **UAF Protocol Specification** : Message formats and processing rules for all UAF protocol messages.

    - **UAF Application API and Transport Binding Specification**: APIs and interoperability profile for client applications to utilize FIDO UAF.
    - **UAF Authenticator Commands**: Low-level functionality that UAF Authenticators should implement to support the UAF protocol.
    - **UAF Authenticator-specific Module API**: Authenticator-specific Module API provided by an ASM to the FIDO client.
    - **UAF Registry of Predefined Values**: defines all the strings and constants reserved by UAF protocols.
    - **UAF APDU**: defines a mapping of FIDO UAF Authenticator commands to Application Protocol Data Units (APDUs).
- **FIDO AppID and Facet Specification** : Scope of user credentials and how a trusted computing base which supports application isolation may make access control decisions about which keys can be used by which applications and web origins.
- **FIDO Metadata Statements**: Information describing form factors, characteristics, and capabilities of FIDO Authenticators used to inform interactions with and make policy decisions about the authenticators.
- **FIDO Metadata Service** : Baseline method for relying parties to access the latest Metadata statements.
- **FIDO ECDAA Algorithm** : Defines the direct anonymous attestation algorithm for FIDO Authenticators.
- **FIDO Registry of Predefined Values**: defines all the strings and constants reserved by FIDO protocols with relevance to multiple FIDO protocol families.
- **FIDO Security Reference**: Provides an analysis of FIDO security based on detailed analysis of security threats pertinent to the FIDO protocols based on its goals, assumptions, and inherent security measures.

The remainder of this Overview section of the reference architecture document introduces the key drivers, goals, and principles which inform the design of FIDO UAF.

Following the Overview, this document describes:

- A high-level look at the components, protocols, and APIs defined by the architecture
- The main FIDO UAF use cases and the protocol message flows required to implement them.
- The relationship of the FIDO protocols to other relevant industry standards.

1.3 FIDO UAF Goals

This section is non-normative.

In order to address today's strong authentication issues and develop a smoothly-functioning low-friction ecosystem, a comprehensive, open, multi-vendor solution architecture is needed that encompasses:

- User devices, whether personally acquired, enterprise-issued, or enterprise BYOD, and the device's potential operating environment, e.g. home, office, in the field, etc.
- Authenticators[1]
- Relying party applications and their deployment environments
- Meeting the needs of both end users and Relying Parties
- Strong focus on both browser- and native-app-based end-user experience

This solution architecture must feature:

- FIDO UAF Authenticator discovery, attestation, and provisioning
- Cross-platform strong authentication protocols leveraging FIDO UAF Authenticators
- A uniform cross-platform authenticator API
- Simple mechanisms for Relying Party integration

The FIDO Alliance envisions an open, multi-vendor, cross-platform reference architecture with these goals:

- **Support strong, multi-factor authentication**: Protect Relying Parties against unauthorized access by supporting end user authentication using two or more strong authentication factors ("something you know", "something you have", "something you are").
- **Build on, but not require, existing device capabilities**: Facilitate user authentication using built-in platform authenticators or capabilities (fingerprint sensors, cameras, microphones, embedded TPM hardware), but do not preclude the use of discrete additional authenticators.
- **Enable selection of the authentication mechanism**: Facilitate Relying Party and user choice amongst supported authentication mechanisms in order to mitigate risks for their particular use cases.
- **Simplify integration of new authentication capabilities**: Enable organizations to expand their use of strong authentication to address new use cases, leverage new device's capabilities, and address new risks with a single authentication approach.
- **Incorporate extensibility for future refinements and innovations**: Design extensible protocols and APIs in order to support the future emergence of additional types of authenticators, authentication methods, and authentication protocols, while maintaining reasonable backwards compatibility.
- **Leverage existing open standards where possible, openly innovate and extend where not**: An open, standardized, royalty-free specification suite will enable the establishment of a virtuous-circle ecosystem, and decrease the risk, complexity, and costs associated with deploying strong authentication. Existing gaps -- notably uniform authenticator provisioning and attestation, a uniform cross-platform authenticator API, as well as a flexible strong authentication challenge-response protocol leveraging the user's authenticators will be addressed.
- **Complement existing single sign-on, federation initiatives**: While industry initiatives (such as OpenID, OAuth, SAML, and others) have created mechanisms to reduce the reliance on passwords through single sign-on or federation technologies, they do not directly address the need for an initial strong authentication interaction between end users and Relying Parties.
- **Preserve the privacy of the end user**: Provide the user control over the sharing of device capability information with Relying Parties, and mitigate the potential for collusion amongst Relying Parties.
- **Unify end-User Experience**: Create easy, fun, and unified end-user experiences across all platforms and across similar Authenticators.

2. FIDO UAF High-Level Architecture

This section is non-normative.

The FIDO UAF Architecture is designed to meet the FIDO goals and yield the desired ecosystem benefits. It accomplishes this by filling in the status-quo's gaps using standardized protocols and APIs.

The following diagram summarizes the reference architecture and how its components relate to typical user devices and Relying Parties.

The FIDO-specific components of the reference architecture are described below.

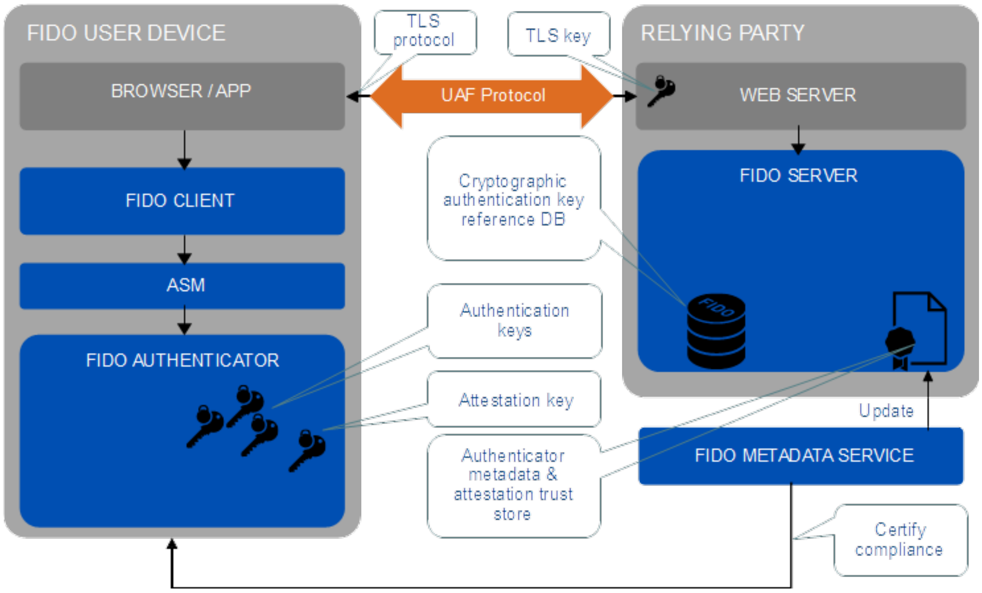


Fig. 1 FIDO UAF High-Level Architecture

2.1 FIDO UAF Client

A FIDO UAF Client implements the client side of the FIDO UAF protocols, and is responsible for:

- Interacting with specific FIDO UAF Authenticators using the FIDO UAF Authenticator Abstraction layer via the FIDO UAF Authenticator API.
- Interacting with a user agent on the device (e.g. a mobile app, browser) using user agent-specific interfaces to communicate with the FIDO UAF Server. For example, a FIDO-specific browser plugin would use existing browser plugin interfaces or a mobile app may use a FIDO-specific SDK. The user agent is then responsible for communicating FIDO UAF messages to a FIDO UAF Server at a Relying Party.

The FIDO UAF architecture ensures that FIDO client software can be implemented across a range of system types, operating systems, and Web browsers. While FIDO client software is typically platform-specific, the interactions between the components should ensure a consistent user experience from platform to platform.

2.2 FIDO UAF Server

A FIDO UAF server implements the server side of the FIDO UAF protocols and is responsible for:

- Interacting with the Relying Party web server to communicate FIDO UAF protocol messages to a FIDO UAF Client via a device user agent.
- Validating FIDO UAF authenticator attestations against the configured authenticator metadata to ensure only trusted authenticators are registered for use.
- Manage the association of registered FIDO UAF Authenticators to user accounts at the Relying Party.
- Evaluating user authentication and transaction confirmation responses to determine their validity.

The FIDO UAF server is conceived as being deployable as an on-premise server by Relying Parties or as being outsourced to a FIDO-enabled third-party service provider.

2.3 FIDO UAF Protocols

The FIDO UAF protocols carry FIDO UAF messages between user devices and Relying Parties. There are protocol messages addressing:

- Authenticator Registration: The FIDO UAF registration protocol enables Relying Parties to:
  - Discover the FIDO UAF Authenticators available on a user's system or device. Discovery will convey FIDO UAF Authenticator attributes to the Relying Party thus enabling policy decisions and enforcement to take place.
  - Verify attestation assertions made by the FIDO UAF Authenticators to ensure the authenticator is authentic and trusted. Verification occurs using the attestation public key certificates distributed via authenticator metadata.
  - Register the authenticator and associate it with the user's account at the Relying Party. Once an authenticator attestation has been validated, the Relying Party can provide a unique secure identifier that is specific to the Relying Party and the FIDO UAF Authenticator. This identifier can be used in future interactions between the pair {RP, Authenticator} and is not known to any other devices.
- User Authentication: Authentication is typically based on cryptographic challenge-response authentication protocols and will facilitate user choice regarding which FIDO UAF Authenticators are employed in an authentication event.
- Secure Transaction Confirmation: If the user authenticator includes the capability to do so, a Relying Party can present the user with a secure message for confirmation. The message content is determined by the Relying Party and could be used in a variety of contexts such as confirming a financial transaction, a user agreement ,or releasing patient records.
- Authenticator Deregistration: Deregistration is typically required when the user account is removed at the Relying Party. The Relying Party can trigger the deregistration by requesting the Authenticator to delete the associated UAF credential with the user account.

2.4 FIDO UAF Authenticator Abstraction Layer

The FIDO UAF Authenticator Abstraction Layer provides a uniform API to FIDO Clients enabling the use of authenticator-based cryptographic services for FIDO-supported operations. It provides a uniform lower-layer "authenticator plugin" API facilitating the deployment of multi-vendor FIDO UAF Authenticators and their requisite drivers.

2.5 FIDO UAF Authenticator

A FIDO UAF Authenticator is a secure entity, connected to or housed within FIDO user devices, that can create key material associated to a Relying Party. The key can then be used to participate in FIDO UAF strong authentication protocols. For example, the FIDO UAF Authenticator can provide a response to a cryptographic challenge using the key material thus authenticating itself to the Relying Party.

In order to meet the goal of simplifying integration of trusted authentication capabilities, a FIDO UAF Authenticator will be able to attest to its particular type (e.g., biometric) and capabilities (e.g., supported crypto algorithms), as well as to its provenance. This provides a Relying Party with a high degree of confidence that the user being authenticated is indeed the user that originally registered with the site.

2.6 FIDO UAF Authenticator Metadata Validation

In the FIDO UAF context, attestation is how Authenticators make claims to a Relying Party during registration that the keys they generate, and/or certain measurements they report, originate from genuine devices with certified characteristics. An attestation signature, carried in a FIDO UAF registration protocol message is validated by the FIDO UAF Server. FIDO UAF Authenticators are created with attestation private keys used to create the signatures and the FIDO UAF Server validates the signature using that authenticator's attestation public key certificate located in the authenticator metadata. The metadata holding attestation certificates is shared with FIDO UAF Servers out of band.

3. FIDO UAF Usage Scenarios and Protocol Message Flows

This section is non-normative.

The FIDO UAF ecosystem supports the use cases briefly described in this section.

3.1 FIDO UAF Authenticator Acquisition and User Enrollment

It is expected that users will acquire FIDO UAF Authenticators in various ways: they purchase a new system that comes with embedded FIDO UAF Authenticator capability; they purchase a device with an embedded FIDO UAF Authenticator, or they are given a FIDO Authenticator by their employer or some other institution such as their bank.

After receiving a FIDO UAF Authenticator, the user must go through an authenticator-specific enrollment process, which is outside the scope of the FIDO UAF protocols. For example, in the case of a fingerprint sensing authenticator, the user must register their fingerprint(s) with the authenticator. Once enrollment is complete, the FIDO UAF Authenticator is ready for registration with FIDO UAF enabled online services and websites.

3.2 Authenticator Registration

Given the FIDO UAF architecture, a Relying Party is able to transparently detect when a user begins interacting with them while possessing an initialized FIDO UAF Authenticator. In this initial introduction phase, the website will prompt the user regarding any detected FIDO UAF Authenticator(s), giving the user options regarding registering it with the website or not.

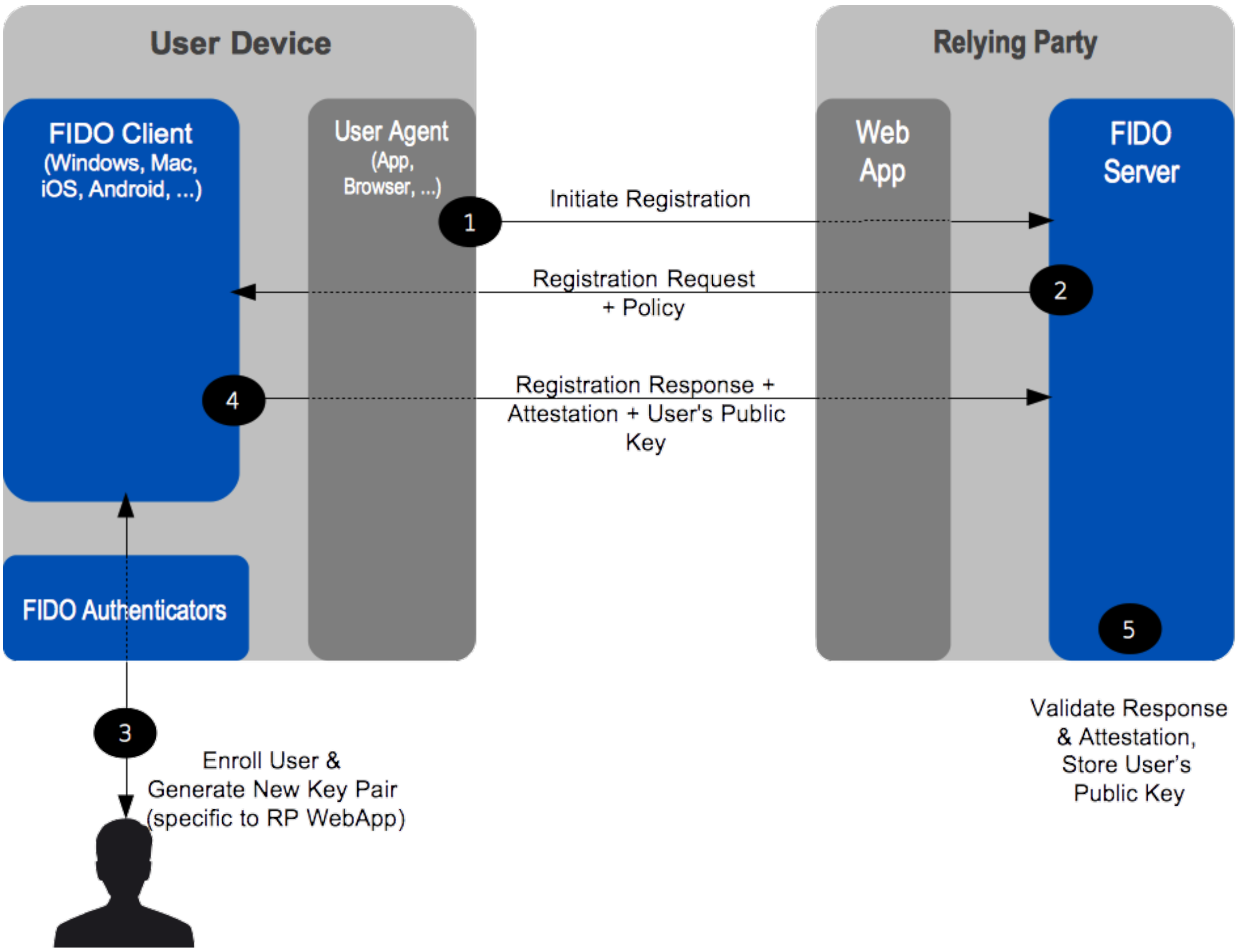


Fig. 2 Registration Message Flow

3.3 Authentication

Following registration, the FIDO UAF Authenticator will be subsequently employed whenever the user authenticates with the website (and the authenticator is present). The website can implement various fallback strategies for those occasions when the FIDO Authenticator is not present. These might range from allowing conventional login with diminished privileges to disallowing login.

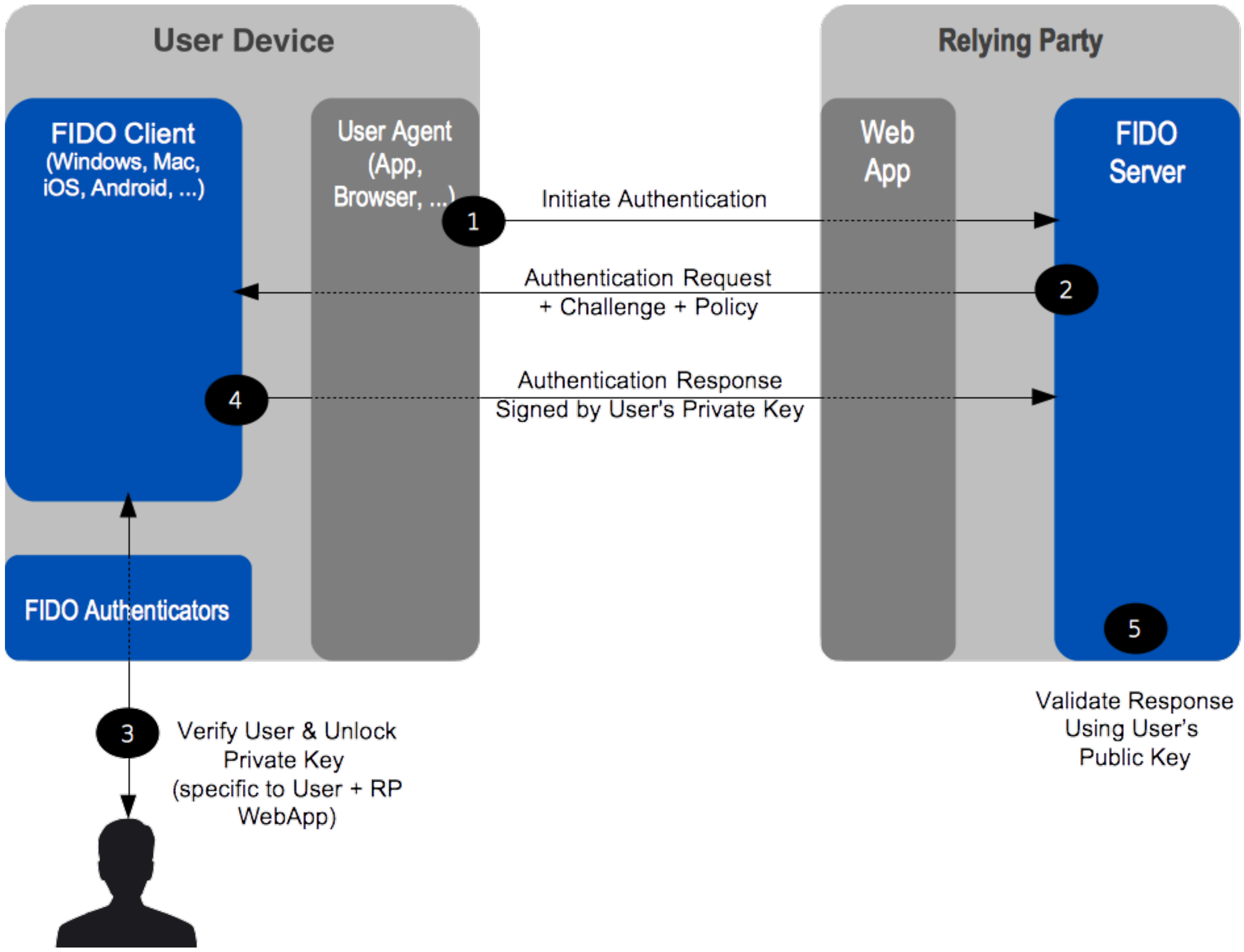


Fig. 3 Authentication Message Flow

This overall scenario will vary slightly depending upon the type of FIDO UAF Authenticator being employed. Some authenticators may sample biometric data such as a face image, fingerprint, or voice print. Others will require a PIN or local authenticator-specific passphrase entry. Still others may simply be a hardware bearer authenticator. Note that it is permissible for a FIDO Client to interact with external services as part of the authentication of the user to the authenticator as long as the FIDO Privacy Principles are adhered to.

3.4 Step-up Authentication

Step-up authentication is an embellishment to the basic website login use case. Often, online services and websites allow unauthenticated, and/or only nominally authenticated use -- for informational browsing, for example. However, once users request more valuable interactions, such as entering a members-only area, the website may request further higher-assurance authentication. This could proceed in several steps if the user then wishes to purchase something, with higher-assurance steps with increasing transaction value.

FIDO UAF will smoothly facilitate this interaction style since the website will be able to discover which FIDO UAF Authenticators are available on FIDO-wielding users' systems, and select incorporation of the appropriate one(s) in any particular authentication interaction. Thus online services and websites will be able to dynamically tailor initial, as well as step-up authentication interactions according to what the user is able to wield and the needed inputs to website's risk analysis engine given the interaction the user has requested.

3.5 Transaction Confirmation

There are various innovative use cases possible given FIDO UAF-enabled Relying Parties with end-users wielding FIDO UAF Authenticators. Website login and step-up authentication are relatively simple examples. A somewhat more advanced use case is secure transaction processing.

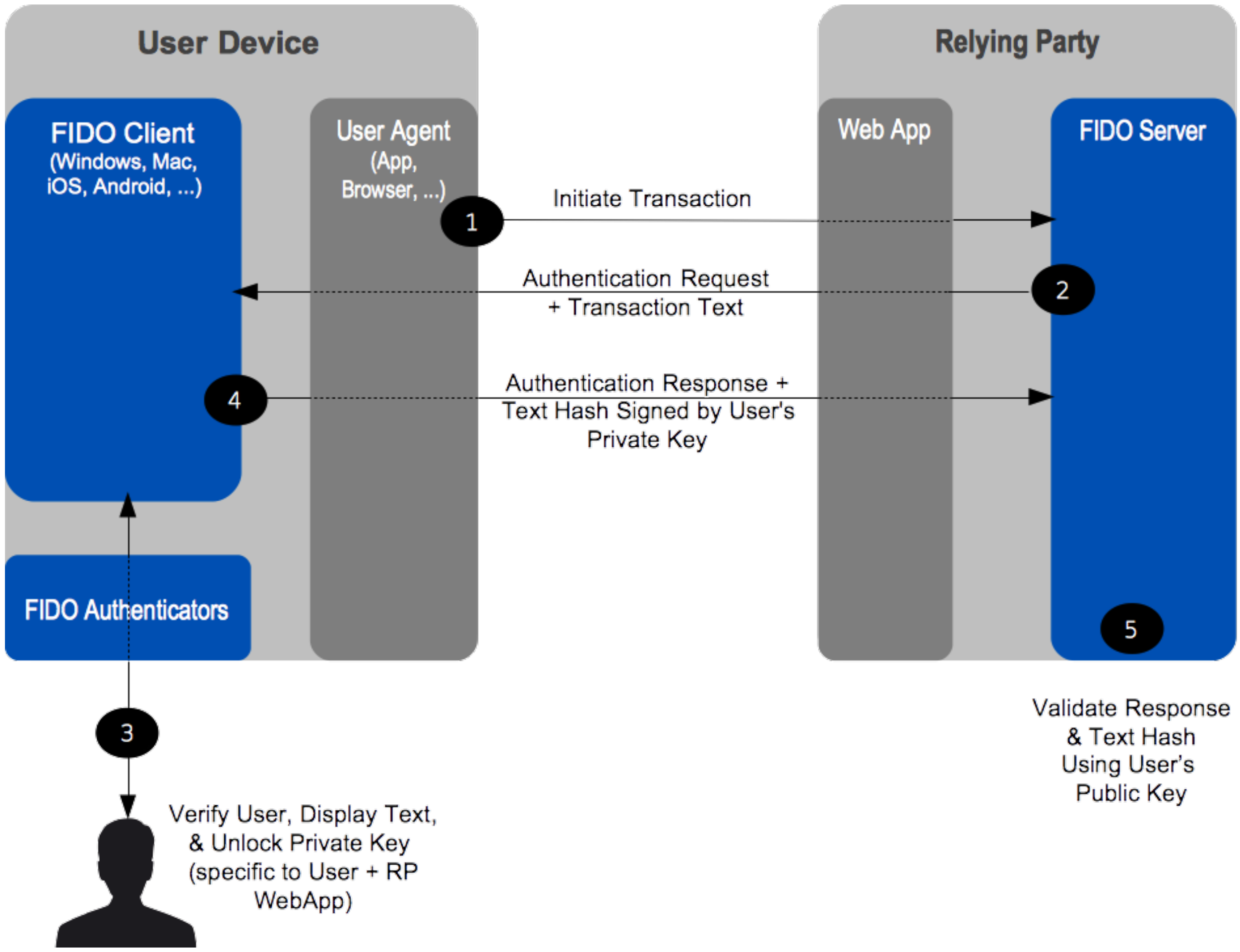


Fig. 4 Confirmation Message Flow

Imagine a situation in which a Relying Party wants the end-user to confirm a transaction (e.g. financial operation, privileged operation, etc) so that any tampering of a transaction message during its route to the end device display and back can be detected. FIDO architecture has a concept of "secure transaction" which provides this capability. Basically if a FIDO UAF Authenticator has a transaction confirmation display capability, FIDO UAF architecture makes sure that the system supports What You See is What You Sign mode (WYSIWYS). A number of different use cases can derive from this capability -- mainly related to authorization of transactions (send money, perform a context specific privileged action, confirmation of email/address, etc).

3.6 Authenticator Deregistration

There are some situations where a Relying Party may need to remove the UAF credentials associated with a specific user account in FIDO Authenticator. For example, the user's account is cancelled or deleted, the user's FIDO Authenticator is lost or stolen, etc. In these situations, the RP may request the FIDO Authenticator to delete authentication keys that are bound to user account.

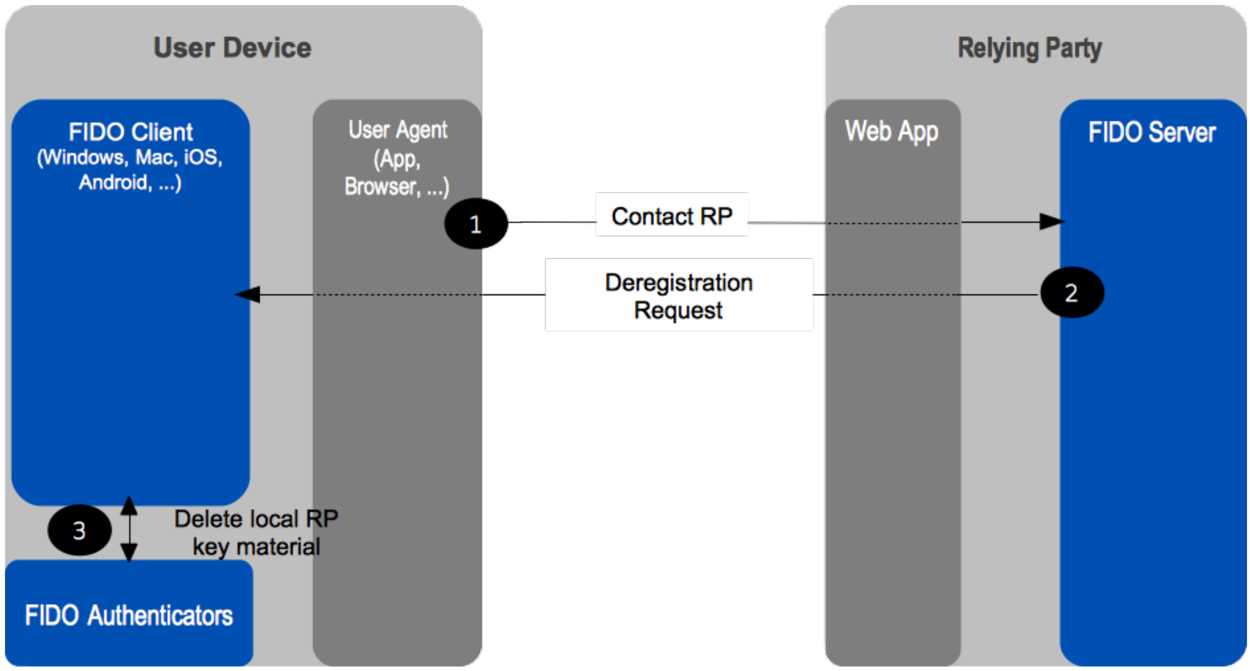


Fig. 5 Deregistration Message Flow

3.7 Adoption of New Types of FIDO UAF Authenticators

Authenticators will evolve and new types are expected to appear in the future. Their adoption on the part of both users and Relying Parties is facilitated by the FIDO architecture. In order to support a new FIDO UAF Authenticator type, Relying Parties need only to add a new entry to their configuration describing the new authenticator, along with its FIDO Attestation Certificate. Afterwards, end users will be able to use the new FIDO UAF Authenticator type with those Relying Parties.

4. Privacy Considerations

This section is non-normative.

User privacy is fundamental to FIDO and is supported in UAF by design. Some of the key privacy-aware design elements are summarized here:

- A UAF device does not have a global identifier visible across relying parties and does not have a global identifier within a particular relying party. If for example, a person looses their UAF device, someone finding it cannot "point it at a relying party" and discover if the original user had any accounts with that relying party. Similarly, if two users share a UAF device and each has registered their account with the same relying party with this device, the relying party will not be able to discern that the two accounts share a device, based on the UAF protocol alone.
- The UAF protocol generates unique asymmetric cryptographic key pairs on a per-device, per-user account, and per-relying party basis. Cryptographic keys used with different relying parties will not allow any one party to link all the actions to the same user, hence the unlinkability property of UAF.
- The UAF protocol operations require minimal personal data collection: at most they incorporate a user's relying party username. This personal data is only used for FIDO purposes, for example to perform user registration, user verification, or authorization. This personal data does not leave the user's computing environment and is only persisted locally when necessary.
- In UAF, user verification is performed locally. The UAF protocol does not convey biometric data to relying parties, nor does it require the storage of such data at relying parties.
- Users explicitly approve the use of a UAF device with a specific relying party. Unique cryptographic keys are generated and bound to a relying party during registration only after the user's consent.
- UAF authenticators can only be identified by their attestation certificates on a production batch-level or on manufacturer- and device model-level. They cannot be identified individually. The UAF specifications require implementers to ship UAF authenticators with the same attestation certificate and private key in batches of 100,000 or more in order to provide unlinkability.

5. Relationship to Other Technologies

This section is non-normative.

OpenID, SAML, and OAuth

FIDO protocols (both UAF and U2F) complement Federated Identity Management (FIM) frameworks, such as OpenID and SAML, as well as web authorization protocols, such as OAuth. FIM Relying Parties can leverage an initial authentication event at an identity provider (IdP). However, OpenID and SAML do not define specific mechanisms for direct user authentication at the IdP.

When an IdP is integrated with a FIDO-enabled authentication service, it can subsequently leverage the attributes of the strong authentication with its Relying Parties. The following diagram illustrates this relationship. FIDO-based authentication (1) would logically occur first, and the FIM protocols would then leverage that authentication event into single sign-on events between the identity provider and its federated Relying Parties (2).[2]

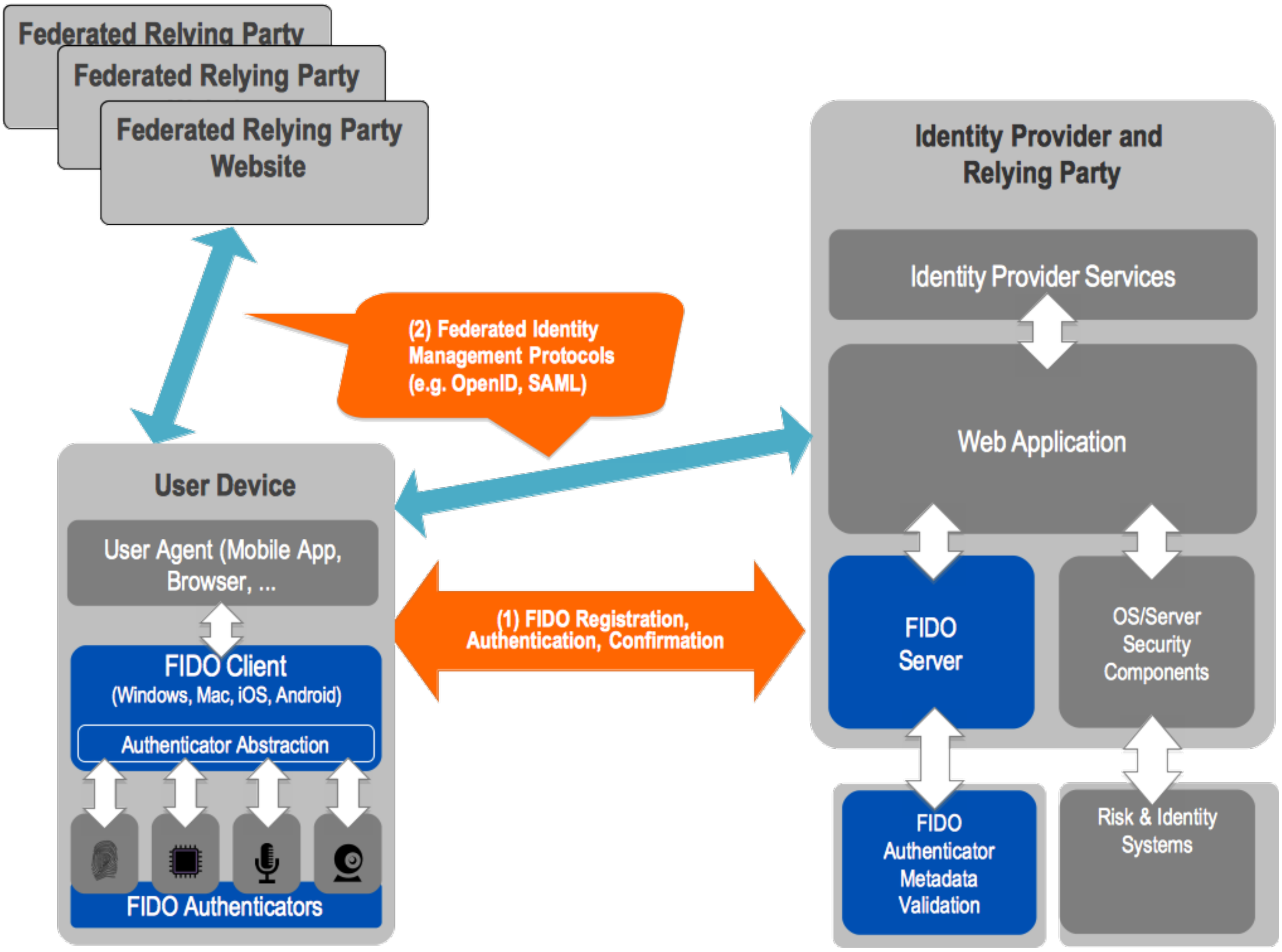


Fig. 6 FIDO UAF & Federated Identity Frameworks

6. OATH, TCG, PKCS#11, and ISO 24727

These are either initiatives (OATH, Trusted Computing Group (TCG)), or industry standards (PKCS#11, ISO 24727). They all share an underlying focus on hardware authenticators.

PKCS#11 and ISO 24727 define smart-card-based authenticator abstractions.

TCG produces specifications for the Trusted Platform Module, as well as networked trusted computing.

OATH, the "Initiative for Open AuTHentication", focuses on defining symmetric key provisioning protocols and authentication algorithms for hardware One-Time Password (OTP) authenticators.

The FIDO framework shares several core notions with the foregoing efforts, such as an authentication abstraction interface, authenticator attestation, key provisioning, and authentication algorithms. FIDO's work will leverage and extend some of these specifications.

Specifically, FIDO will complement them by addressing:

- Authenticator discovery
- User experience
- Harmonization of various authenticator types, such as biometric, OTP, simple presence, smart card, TPM, etc.

7. Table of Figures

---

1. Also known as: Authentication Tokens, Security Tokens, etc.↩

2. FIM protocols typically convey IdP <-> RP interactions through the browser via HTTP redirects and POSTs.↩

IMPLEMENTATION DRAFT

FIDO UAF Protocol Specification

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html
Editors:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Eric Tiffany, FIDO Alliance
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Dirk Balfanz, Google, Inc.
Brad Hill, PayPal, Inc.
Jeff Hodges, PayPal, Inc.
Ka Yang, Nok Nok Labs, Inc.

Abstract

The goal of the Universal Authentication Framework is to provide a unified and extensible authentication mechanism that supplants passwords while avoiding the shortcomings of current alternative authentication approaches.

This approach is designed to allow the relying party to choose the best available authentication mechanism for a particular end user or interaction, while preserving the option to leverage emerging device security capabilities in the future without requiring additional integration effort.

This document describes the FIDO architecture in detail, it defines the flow and content of all UAF protocol messages and presents the rationale behind the design choices.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

The notation base64url refers to "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as `required`.

WebIDL dictionary members must not have a value of null — i.e., there are no declarations of nullable dictionary members in this specification.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

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

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

NOTE

Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

The goal of this Universal Authentication Framework is to provide a unified and extensible authentication mechanism that supplants passwords while avoiding the shortcomings of current alternative authentication approaches.

The design goal of the protocol is to enable Relying Parties to leverage the diverse and heterogeneous set of security capabilities available on end users' devices via a single, unified protocol.

This approach is designed to allow the FIDO Relying Parties to choose the best available authentication mechanism for a particular end user or interaction, while preserving the option for a relying party to leverage emerging device security capabilities in the future, without requiring additional integration effort.

2.1 Scope

This document describes FIDO architecture in detail and defines the UAF protocol as a network protocol. It defines the flow and content of all UAF messages and presents the rationale behind the design choices.

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

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

The answers to these questions can be found in other UAF specifications, e.g. [UAFAppAPIAndTransport] [UAFASM] [UAFAuthnrCommands].

2.2 Architecture

The following diagram depicts the entities involved in UAF protocol.

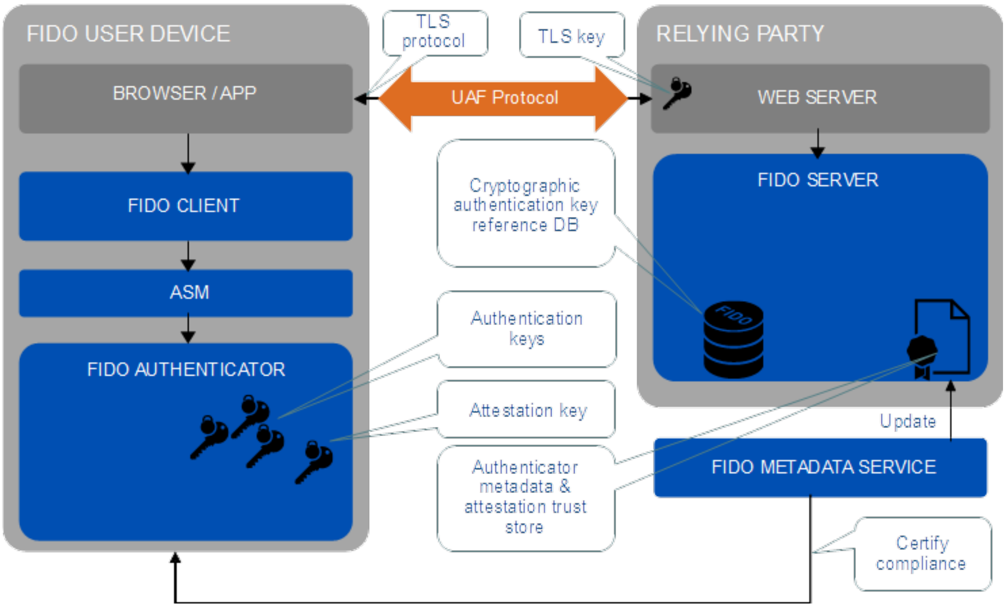


Fig. 1 The UAF Architecture

Of these entities, only these three directly create and/or process UAF protocol messages:

- FIDO Server, running on the relying party's infrastructure
- FIDO UAF Client, part of the user agent and running on the FIDO user device
- FIDO Authenticator, integrated into the FIDO user device

It is assumed in this document that a FIDO Server has access to the UAF Authenticator Metadata [FIDOMetadataStatement] describing all the authenticators it will interact with.

2.3 Protocol Conversation

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

- **Registration:** UAF allows the relying party to register a FIDO Authenticator with the user's account at the relying party. The relying party can specify a policy for supporting various FIDO Authenticator types. A FIDO UAF Client will only register existing authenticators in accordance with that policy.

- **Authentication:** UAF allows the relying party to prompt the end user to authenticate using a previously registered FIDO Authenticator. This authentication can be invoked any time, at the relying party's discretion.

- **Transaction Confirmation:** In addition to providing a general authentication prompt, UAF offers support for prompting the user to confirm a specific transaction.

  This prompt includes the ability to communicate additional information to the client for display to the end user, using the client's transaction confirmation display. The goal of this additional authentication operation is to enable relying parties to ensure that the user is confirming a specified set of the transaction details (instead of authenticating a session to the user agent).

- **Deregistration:** The relying party can trigger the deletion of the account-related authentication key material.

Although this document defines the FIDO Server as the initiator of requests, in a real world deployment the first UAF operation will always follow a user agent's (e.g. HTTP) request to a relying party.

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

2.3.1 Registration

The following diagram shows the message flows for registration.

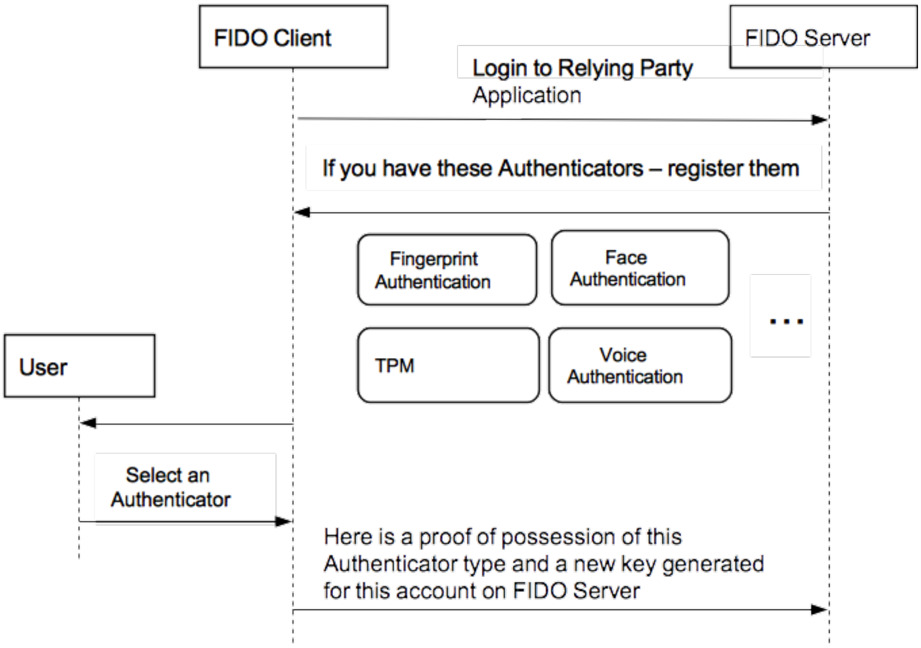


Fig. 2 UAF Registration Message Flow

> NOTE
>
> The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow the FIDO UAF Client to do some "housekeeping" tasks.

2.3.2 Authentication

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

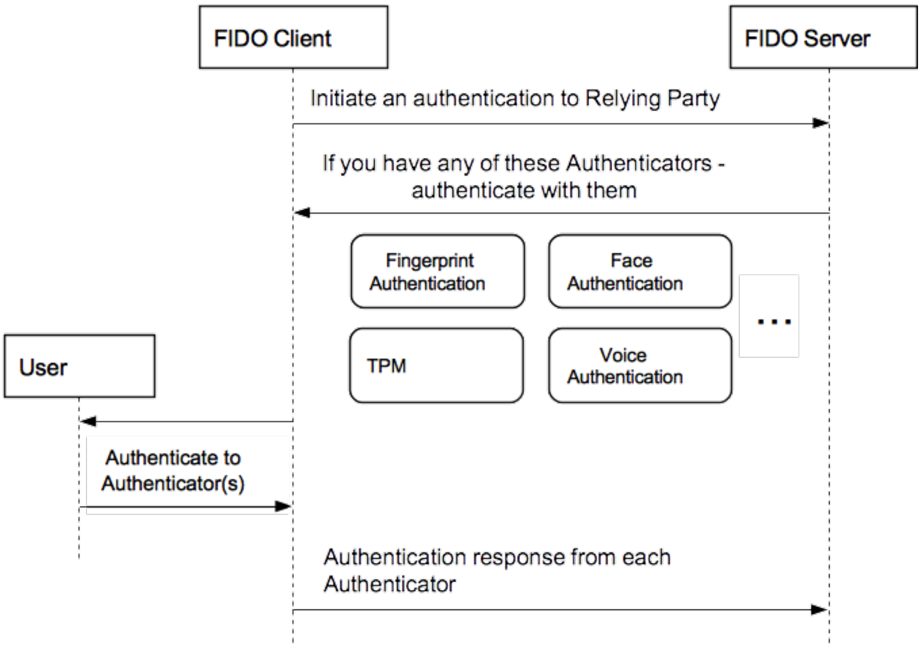


Fig. 3 Authentication Message Flow

> NOTE
>
> The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow FIDO UAF Client to do some "housekeeping" tasks.

2.3.3 Transaction Confirmation

The following figure depicts the transaction confirmation message flow.

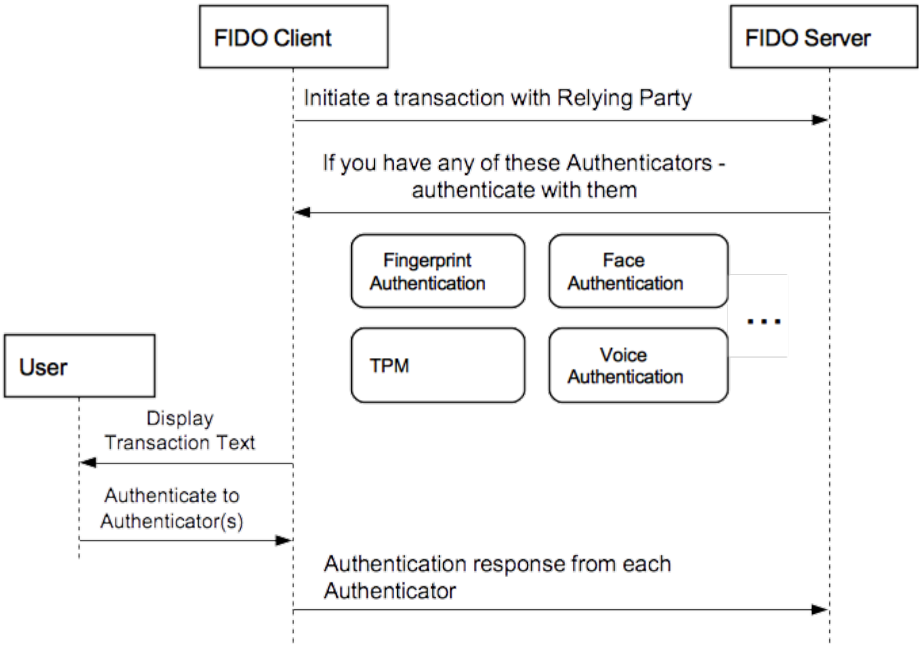


Fig. 4 Transaction Confirmation Message Flow

NOTE

The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow the FIDO UAF Client to do some "housekeeping" tasks.

2.3.4 Deregistration

The following diagram depicts the deregistration message flow.

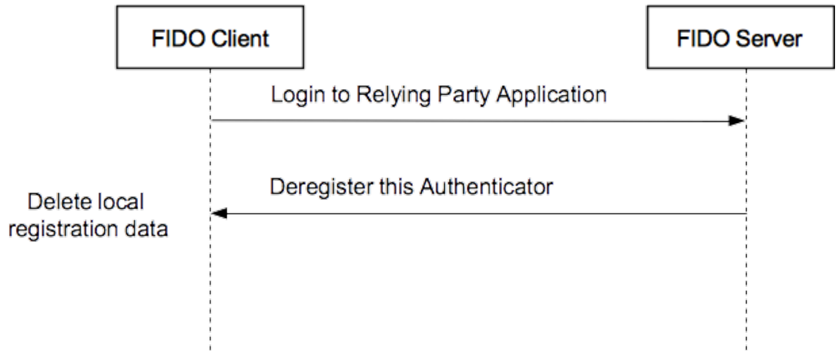


Fig. 5 Deregistration Message Flow

NOTE

The client application should use the appropriate API to inform the FIDO UAF Client of the results of the operation (see section 2.3.1 in [UAFAppAPIAndTransport]) in order to allow the FIDO UAF Client to do some "housekeeping" tasks.

3. Protocol Details

This section is normative.

This section provides a detailed description of operations supported by the UAF Protocol.

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

All string literals in this specification are constructed from Unicode codepoints within the set `U+0000`..`U+007F`.

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

NOTE

All data used in this protocol must be exchanged using a secure transport protocol (such as TLS/HTTPS) established between the FIDO UAF Client and the relying party in order to follow the assumptions made in [FIDOSecRef]; details are specified in section 4.1.7 TLS Protected Communication.

The notation `base64url(byte[8..64])` reads as 8-64 bytes of data encoded in base64url, "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

The notation `string[5]` reads as five unicode characters, represented as a UTF-8 [RFC3629] encoded string of the type indicated in the declaration, typically a WebIDL [WebIDL-ED] DOMString.

As the UTF-8 representation has variable length, the *maximum* byte length of `string[5]` is `string[4*5]`.

All strings are case-sensitive unless stated otherwise.

This document uses WebIDL [WebIDL-ED] to define UAF protocol messages.

Implementations must serialize the UAF protocol messages for transmission using UTF-8 encoded JSON [RFC4627].

3.1 Shared Structures and Types

This section defines types and structures shared by various operations.

3.1.1 Version Interface

Represents a generic version with major and minor fields.

WebIDL

```
interface Version {
    readonly    attribute unsigned short major;
    readonly    attribute unsigned short minor;
};
```

3.1.1.1 Attributes

`major` of type unsigned short, readonly
Major version.

`minor` of type unsigned short, readonly
Minor version.

3.1.2 Operation enumeration

Describes the operation type of a UAF message or request for a message.

WebIDL

```
enum Operation {
    "Reg",
    "Auth",
    "Dereg"
};
```

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

3.1.3 OperationHeader dictionary

Represents a UAF message Request and Response header

WebIDL

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

3.1.3.1 Dictionary `OperationHeader` Members

`upv` of type required Version
UAF protocol version (`upv`). To conform with this version of the UAF spec set, the `major` value must be `1` and the `minor` value must be `1`.

`op` of type required Operation
Name of FIDO operation (`op`) this message relates to.

> NOTE
>
> "Auth" is used for both authentication and transaction confirmation.

`appID` of type DOMString
`string[0..512]`.

The application identifier that the relying party would like to assert.

There are three ways to set the `AppID` [FIDOAppIDAndFacets]:

1. If the element is missing or empty in the request, the FIDO UAF Client must set it to the `FacetID` of the caller.

2. If the `appID` present in the message is identical to the `FacetID` of the caller, the FIDO UAF Client must accept it.
3. If it is an URI with HTTPS protocol scheme, the FIDO UAF Client must use it to load the list of trusted facet identifiers from the specified URI. The FIDO UAF Client must only accept the request, if the facet identifier of the caller matches one of the trusted facet identifiers in the list returned from dereferencing this URI.

> NOTE
>
> The new key pair that the authenticator generates will be associated with this application identifier.
>
> *Security Relevance:* The application identifier is used by the FIDO UAF Client to verify the eligibility of an application to trigger the use of a specific `UAuth.Key`. See [FIDOAppIDAndFacets]

`serverData` of type DOMString
`string[1..1536]`.

A session identifier created by the relying party.

> NOTE
> The relying party can opaquely store things like expiration times for the registration session, protocol version used and other useful information in `serverData`. This data is opaque to FIDO UAF Clients. FIDO Servers may reject a response that is lacking this data or is containing unauthorized modifications to it.
>
> Servers that depend on the integrity of `serverData` should apply appropriate security measures, as described in Registration Request Generation Rules for FIDO Server and section ServerData and KeyHandle.

`exts` of type array of *Extension*
List of UAF Message Extensions.

3.1.4 Authenticator Attestation ID (AAID) typedef

WebIDL

```
typedef DOMString AAID;
```

`string[9]`

Each authenticator must have an `AAID` to identify UAF enabled authenticator models globally. The `AAID` must uniquely identify a specific authenticator model within the range of all UAF-enabled authenticator models made by all authenticator vendors, where authenticators of a specific model must share identical security characteristics within the model (see Security Considerations).

The `AAID` is a string with format "V#M", where

"#" is a separator

"V" indicates the authenticator Vendor Code. This code consists of 4 hexadecimal digits.

"M" indicates the authenticator Model Code. This code consists of 4 hexadecimal digits.

The Augmented BNF [ABNF] for the `AAID` is:

```
AAID = 4(HEXDIG) "#" 4(HEXDIG)
```

> NOTE
>
> *HEXDIG* is case insensitive, i.e. "03EF" and "03ef" are identical.

The FIDO Alliance is responsible for assigning authenticator vendor Codes.

Authenticator vendors are responsible for assigning authenticator model codes to their authenticators. Authenticator vendors must assign unique `AAIDs` to authenticators with different security characteristics.

AAIDs are unique and each of them must relate to a distinct authentication metadata file ([FIDOMetadataStatement])

> NOTE
>
> Adding new firmware/software features, or changing the underlying hardware protection mechanisms will typically change the security characteristics of an authenticator and hence would require a new `AAID` to be used. Refer to ([FIDOMetadataStatement]) for more details.

3.1.5 KeyID typedef

WebIDL

```
typedef DOMString KeyID;
```

`base64url(byte[32...2048])`

`KeyID` is a unique identifier (within the scope of an `AAID`) used to refer to a specific `UAuth.Key`. It is generated by the authenticator and registered with a FIDO Server.

The (`AAID`, `KeyID`) tuple must uniquely identify an authenticator's registration for a relying party. Whenever a FIDO Server wants to provide specific information to a particular authenticator it must use the (`AAID`, `KeyID`) tuple.

KeyID must be base64url encoded within the UAF message (see above).

During step-up authentication and deregistration operations, the FIDO Server should provide the KeyID back to the authenticator for the latter to locate the appropriate user authentication key, and perform the necessary operation with it.

Roaming authenticators which don't have internal storage for, and cannot rely on any ASM to store, generated key handles should provide the key handle as part of the AuthenticatorRegistrationAssertion.assertion.KeyID during the registration operation (see also section ServerData and KeyHandle) and get the key handle back from the FIDO Server during the step-up authentication (in the MatchCriteria dictionary which is part of the policy) or deregistration operations (see [UAFAuthnrCommands] for more details).

> NOTE
>
> The exact structure and content of a KeyID is specific to the authenticator implementation.

3.1.6 ServerChallenge typedef

WebIDL

```
typedef DOMString ServerChallenge;
```

base64url(byte[8...64])

ServerChallenge is a server-provided random challenge. *Security Relevance:* The challenge is used by the FIDO Server to verify whether an incoming response is new, or has already been processed. See section Replay Attack Protection for more details.

The ServerChallenge should be mixed into the entropy pool of the authenticator. *Security Relevance:* The FIDO Server should provide a challenge containing strong cryptographic randomness whenever possible. See section Server Challenge and Random Numbers.

> NOTE
>
> The minimum challenge length of 8 bytes follows the requirement in [SP800-63] and is equivalent to the 20 decimal digits as required in [RFC6287].

> NOTE
>
> The maximum length has been defined such that SHA-512 output can be used without truncation.

> NOTE
>
> The mixing of multiple sources of randomness is recommended to improve the quality of the random numbers generated by the authenticator, as described in [RFC4086].

3.1.7 FinalChallengeParams dictionary

WebIDL

```
dictionary FinalChallengeParams {
    required DOMString       appID;
    required ServerChallenge challenge;
    required DOMString       facetID;
    required ChannelBinding  channelBinding;
};
```

*3.1.7.1 Dictionary **FinalChallengeParams** Members*

appID of type required DOMString
string[1..512]

The value must be taken from the appID field of the **OperationHeader**

challenge of type required ServerChallenge

The value must be taken from the challenge field of the request (e.g. RegistrationRequest.challenge, AuthenticationRequest.challenge).

facetID of type required DOMString
string[1..512]

The value is determined by the FIDO UAF Client and it depends on the calling application. See [FIDOAppIDAndFacets] for more details. *Security Relevance:* The facetID is determined by the FIDO UAF Client and verified against the list of trusted facets retrieved by dereferencing the appID of the calling application.

channelBinding of type required ChannelBinding

Contains the TLS information to be sent by the FIDO Client to the FIDO Server, binding the TLS channel to the FIDO operation.

3.1.8 TLS ChannelBinding dictionary

ChannelBinding contains channel binding information [RFC5056].

> NOTE

*Security Relevance:*The channel binding may be verified by the FIDO Server in order to detect and prevent MITM attacks.

At this time, the following channel binding methods are supported:

- TLS ChannelID (`cid_pubkey`) [ChannelID]
- serverEndPoint [RFC5929]
- tlsServerCertificate
- tlsUnique [RFC5929]

Further requirements:

1. If data related to any of the channel binding methods, described here, is available to the FIDO UAF Client (i.e. included in this dictionary), it must be used according to the relevant specification .
2. All channel binding methods described here must be supported by the FIDO Server. The FIDO Server may reject operations if the channel binding cannot be verified successfully.

NOTE

- If channel binding data is accessible to the web browser or client application, it must be relayed to the FIDO UAF Client in order to follow the assumptions made in [FIDOSecRef].
- If channel binding data is accessible to the web server, it must be relayed to the FIDO Server in order to follow the assumptions made in [FIDOSecRef]. The FIDO Server relies on the web server to provide accurate channel binding information.

WebIDL

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

3.1.8.1 Dictionary `ChannelBinding` Members

`serverEndPoint` of type DOMString

The field `serverEndPoint` must be set to the base64url-encoded hash of the TLS server certificate if this is available. The hash function must be selected as follows:

1. if the certificate's `signatureAlgorithm` uses a single hash function and that hash function is either MD5 [RFC1321] or SHA-1 [RFC6234], then use SHA-256 [FIPS180-4];
2. if the certificate's `signatureAlgorithm` uses a single hash function and that hash function is neither MD5 nor SHA-1, then use the hash function associated with the certificate's`signatureAlgorithm`;
3. if the certificate's `signatureAlgorithm` uses no hash functions, or uses multiple hash functions, then this channel binding type's channel bindings are undefined at this time (updates to this channel binding type may occur to address this issue if it ever arises)

This field must be absent if the TLS server certificate is not available to the processing entity (e.g., the FIDO UAF Client) or the hash function cannot be determined as described.

`tlsServerCertificate` of type DOMString

This field must be absent if the TLS server certificate is not available to the FIDO UAF Client.

This field must be set to the base64url-encoded, DER-encoded TLS server certificate, if this data is available to the FIDO UAF Client.

`tlsUnique` of type DOMString

must be set to the base64url-encoded TLS channel `Finished` structure. It must, however, be absent, if this data is not available to the FIDO UAF Client [RFC5929].

The use of the tlsUnique is deprecated as the security of the`tls-unqiue` channel binding type [RFC5929] is broken, see [TLSAUTH].

`cid_pubkey` of type DOMString

must be absent if the client TLS stack doesn't provide TLS ChannelID [ChannelID] information to the processing entity (e.g., the web browser or client application).

must be set to "unused" if TLS ChannelID information is supported by the client-side TLS stack but has not been signaled by the TLS (web) server.

Otherwise, it must be set to the base64url-encoded serialized [RFC4627] `JwkKey` structure using UTF-8 encoding.

3.1.9 JwkKey dictionary

`JwkKey` is a dictionary representing a JSON Web Key encoding of an elliptic curve public key [JWK].

This public key is the ChannelID public key minted by the client TLS stack for the particular relying party. [ChannelID] stipulates using only a particular elliptic curve, and the particular coordinate type.

WebIDL

```
dictionary JwkKey {
    required DOMString kty = "EC";
    required DOMString crv = "P-256";
    required DOMString x;
```

```
    required DOMString y;
};
```

3.1.9.1 Dictionary `JwkKey` Members

`kty` of type required DOMString, defaulting to `"EC"`
: Denotes the key type used for Channel ID. At this time only elliptic curve is supported by [ChannelID], so it must be set to "EC" [JWA].

`crv` of type required DOMString, defaulting to `"P-256"`
: Denotes the elliptic curve on which this public key is defined. At this time only the NIST curve `secp256r1` is supported by [ChannelID], so the `crv` parameter must be set to "P-256".

`x` of type required DOMString
: Contains the base64url-encoding of the x coordinate of the public key (big-endian, 32-byte value).

`y` of type required DOMString
: Contains the base64url-encoding of the y coordinate of the public key (big-endian, 32-byte value).

3.1.10 Extension dictionary

FIDO extensions can appear in several places, including the UAF protocol messages, authenticator commands, or in the assertion signed by the authenticator.

Each extension has an identifier, and the namespace for extension identifiers is FIDO UAF global (i.e. doesn't depend on the message where the extension is present).

Extensions can be defined in a way such that a processing entity which doesn't understand the meaning of a specific extension must abort processing, or they can be specified in a way that unknown extension can (safely) be ignored.

Extension processing rules are defined in each section where extensions are allowed.

Generic extensions used in various operations.

WebIDL

```
dictionary Extension {
    required DOMString id;
    required DOMString data;
    required boolean   fail_if_unknown;
};
```

3.1.10.1 Dictionary `Extension` Members

`id` of type required DOMString
: `string[1..32]`.

  Identifies the extension.

`data` of type required DOMString
: Contains arbitrary data with a semantics agreed between server and client. Binary data is base64url-encoded.

  This field may be empty.

`fail_if_unknown` of type required boolean
: Indicates whether unknown extensions must be ignored (`false`) or must lead to an error (`true`).

  - A value of `false` indicates that unknown extensions must be ignored
  - A value of `true` indicates that unknown extensions must result in an error.

> NOTE
>
> The FIDO UAF Client might (a) process an extension or (b) pass the extension through to the ASM. Unknown extensions must be passed through.
>
> The ASM might (a) process an extension or (b) pass the extension through to the FIDO authenticator. Unknown extensions must be passed through.
>
> The FIDO authenticator must handle the extension or ignore it (only if it doesn't know how to handle it *and* `fail_if_unknown` is not set). If the FIDO authenticator doesn't understand the meaning of the extension and `fail_if_unknown` is set, it must generate an error (see definition of `fail_if_unknown` above).
>
> When passing through an extension to the next entity, the`fail_if_unknown` flag must be preserved (see [UAFASM] [UAFAuthnrCommands]).
>
> FIDO protocol messages are not signed. If the security depends on an extension being known or processed, then such extension should be accompanied by a related (and signed) extension in the authenticator assertion (e.g. `TAG_UAFV1_REG_ASSERTION`, `TAG_UAFV1_AUTH_ASSERTION`). If the security has been increased (e.g. the FIDO authenticator according to the description in the metadata statement accepts multiple fingers but in this specific case indicates that the finger used at registration was also used for authentication) there is no need to mark the extension as `fail_if_unknown` (i.e. tag 0x3E12 should be used [UAFAuthnrCommands]). If the security has been degraded (e.g. the FIDO authenticator according to the description in the metadata statement accepts only the finger used at registration for authentication but in this specific case indicates that a different finger was used for authentication) the extension must be marked as `fail_if_unknown` (i.e. tag 0x3E11 must be used [UAFAuthnrCommands]).

3.1.11 MatchCriteria dictionary

Represents the matching criteria to be used in the server policy.

The MatchCriteria object is considered to match an authenticator, if *all* fields in the object are considered to match (as indicated in the particular fields).

WebIDL

```
dictionary MatchCriteria {
    AAID[]            aaid;
    DOMString[]       vendorID;
    KeyID[]           keyIDs;
    unsigned long     userVerification;
    unsigned short    keyProtection;
    unsigned short    matcherProtection;
    unsigned long     attachmentHint;
    unsigned short    tcDisplay;
    unsigned short[]  authenticationAlgorithms;
    DOMString[]       assertionSchemes;
    unsigned short[]  attestationTypes;
    unsigned short    authenticatorVersion;
    Extension[]       exts;
};
```

3.1.11.1 Dictionary **MatchCriteria** *Members*

aaid of type array of *AAID*
List of AAIDs, causing matching to be restricted to certain AAIDs.

The field m.aaid may be combined with (one or more of) m.keyIDs, m.attachmentHint, m.authenticatorVersion, and m.exts, but m.aaid must not be combined with any other match criteria field.

If m.aaid is not provided - at least m.authenticationAlgorithms and m.assertionSchemes must be provided.

The match succeeds if at least one AAID entry in this array matches AuthenticatorInfo.aaid [UAFASM].

NOTE

This field corresponds to MetadataStatement.aaid [FIDOMetadataStatement].

vendorID of type array of DOMString
The vendorID causing matching to be restricted to authenticator models of the given vendor. The first 4 characters of the AAID are the vendorID (see **AAID**)).

The match succeeds if at least one entry in this array matches the first 4 characters of the AuthenticatorInfo.aaid [UAFASM].

NOTE

This field corresponds to the first 4 characters of MetadataStatement.aaid [FIDOMetadataStatement].

keyIDs of type array of *KeyID*
A list of authenticator KeyIDs causing matching to be restricted to a given set of KeyID instances. (see TAG_KEYID in [UAFRegistry]).

This match succeeds if at least one entry in this array matches.

NOTE

This field corresponds to AppRegistration.keyIDs [UAFASM].

userVerification of type unsigned long
A set of 32 bit flags which may be set if matching should be restricted by the user verification method (see [FIDORegistry]).

NOTE
The match with AuthenticatorInfo.userVerification ([UAFASM]) succeeds, if the following condition holds (written in Java):

```
if (
        // They are equal
        (AuthenticatorInfo.userVerification == MatchCriteria.userVerification) ||

        // USER_VERIFY_ALL is not set in both of them and they have at least one common bit set
        (
            ((AuthenticatorInfo.userVerification & USER_VERIFY_ALL) == 0) &&
            ((MatchCriteria.userVerification & USER_VERIFY_ALL) == 0) &&
            ((AuthenticatorInfo.userVerification & MatchCriteria.userVerification) != 0)
        )
   )
```

NOTE
This field value can be derived from MetadataStatement.userVerificationDetails as follows:

1. if MetadataStatement.userVerificationDetails contains multiple entries, then:
    1. if one or more entries MetadataStatement.userVerificationDetails[i] contain multiple entries, then: stop, direct derivation is not possible. Must generate MatchCriteria object by providing a list of matching AAIDs.
    2. if all entries MetadataStatement.userVerificationDetails[i] only contain a single entry, then: combine all entries MetadataStatement.userVerificationDetails[0][0].userVerification to MetadataStatement.userVerificationDetails[N-1][0].userVerification into a single value using a bitwise OR

operation.

2. if `MetadataStatement.userVerificationDetails` contains a single entry, then: combine all entries `MetadataStatement.userVerificationDetails[0][0].userVerification` to `MetadataStatement.userVerificationDetails[0][N-1].userVerification` into a single value using a bitwise OR operation and (if multiple bit flags have been set) additionally set the flag `USER_VERIFY_ALL`.

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

keyProtection of type unsigned short
A set of 16 bit flags which may be set if matching should be restricted by the key protections used (see [FIDORegistry]).

This match succeeds, if at least one of the bit flags matches the value of `AuthenticatorInfo.keyProtection` [UAFASM].

NOTE

This field corresponds to `MetadataStatement.keyProtection` [FIDOMetadataStatement].

matcherProtection of type unsigned short
A set of 16 bit flags which may be set if matching should be restricted by the matcher protection (see [FIDORegistry]).

The match succeeds if at least one of the bit flags matches the value of `AuthenticatorInfo.matcherProtection` [UAFASM].

NOTE

This field corresponds to the `MetadataStatement.matcherProtection` metadata statement. See [FIDOMetadataStatement].

attachmentHint of type unsigned long
A set of 32 bit flags which may be set if matching should be restricted by the authenticator attachment mechanism (see [FIDORegistry]).

This field is considered to match, if at least one of the bit flags matches the value of `AuthenticatorInfo.attachmentHint` [UAFASM].

NOTE

This field corresponds to the `MetadataStatement.attachmentHint` metadata statement.

tcDisplay of type unsigned short
A set of 16 bit flags which may be set if matching should be restricted by the transaction confirmation display availability and type. (see [FIDORegistry]).

This match succeeds if at least one of the bit flags matches the value of `AuthenticatorInfo.tcDisplay` [UAFASM].

NOTE

This field corresponds to the `MetadataStatement.tcDisplay` metadata statement. See [FIDOMetadataStatement].

authenticationAlgorithms of type array of unsigned short
An array containing values of supported authentication algorithm TAG values (see [FIDORegistry], prefix `ALG_SIGN`) if matching should be restricted by the supported authentication algorithms. This field must be present, if field `aaid` is missing.

This match succeeds if at least one entry in this array matches the `AuthenticatorInfo.authenticationAlgorithm` [UAFASM].

NOTE

This field corresponds to the `MetadataStatement.authenticationAlgorithm` metadata statement. See [FIDOMetadataStatement].

assertionSchemes of type array of DOMString
A list of supported assertion schemes if matching should be restricted by the supported schemes. This field must be present, if field `aaid` is missing.

See section UAF Supported Assertion Schemes for details.

This match succeeds if at least one entry in this array matches `AuthenticatorInfo.assertionScheme` [UAFASM].

NOTE

This field corresponds to the `MetadataStatement.assertionScheme` metadata statement. See [FIDOMetadataStatement].

attestationTypes of type array of unsigned short
An array containing the preferred attestation TAG values (see [UAFRegistry], prefix `TAG_ATTESTATION`). The order of items must be preserved. The most-preferred attestation type comes first.

This match succeeds if at least one entry in this array matches one entry in `AuthenticatorInfo.attestationTypes` [UAFASM].

NOTE

This field corresponds to the `MetadataStatement.attestationTypes` metadata statement. See [FIDOMetadataStatement].

`authenticatorVersion` of type unsigned short
Contains an authenticator version number, if matching should be restricted by the authenticator version in use.

This match succeeds if the value is *lower or equal* to the field `AuthenticatorVersion` included in `TAG_UAFV1_REG_ASSERTION` or `TAG_UAFV1_AUTH_ASSERTION` or a corresponding value in the case of a different assertion scheme.

NOTE

Since the semantic of the `authenticatorVersion` depends on the AAID, the field `authenticatorVersion` should always be combined with a single `aaid` in `MatchCriteria`.

This field corresponds to the `MetadataStatement.authenticatorVersion` metadata statement. See [FIDOMetadataStatement].

The use of authenticatorVersion in the policy is deprecated since there is no standardized way for the FIDO Client to learn the authenticatorVersion. The authenticatorVersion is included in the auhentication assertion and hence can still be evaluated in the FIDO Server.

`exts` of type array of *Extension*
Extensions for matching policy.

3.1.12 Policy dictionary

Contains a specification of accepted authenticators and a specification of disallowed authenticators.

WebIDL

```
dictionary Policy {
    required MatchCriteria[][] accepted;
    MatchCriteria[]            disallowed;
};
```

3.1.12.1 Dictionary `Policy` Members

`accepted` of type array of array of required MatchCriteria

This field is a two-dimensional array describing the required authenticator characteristics for the server to accept either a FIDO registration, or authentication operation for a particular purpose.

This two-dimensional array can be seen as a list of sets. List elements (i.e. the sets) are alternatives (OR condition).

All elements within a set must be combined:

The first array index indicates OR conditions (i.e. the list). Any set of authenticator(s) satisfying these `MatchCriteria` in the first index is acceptable to the server for this operation.

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

The MatchCriteria array represents ordered preferences by the server. Servers must put their preferred authenticators first, and FIDO UAF Clients should respect those preferences, either by presenting authenticator options to the user in the same order, or by offering to perform the operation using only the highest-preference authenticator(s).

NOTE
This list must not be empty. If the FIDO Server accepts any authenticator, it can follow the example below.

EXAMPLE 1: Example for an 'any' policy

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

NOTE

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

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

3.2 Processing Rules for the Server Policy

This section is normative.

The FIDO UAF Client must follow the following rules while parsing server policy:

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

3. Choose the combination whose criteria best match the features of the currently available authenticators
4. Collect information about available authenticators
5. Ignore authenticators which match the `Policy.disallowed` criteria
6. Match collected information with the matching criteria imposed in the policy (see MatchCriteria dictionary for more details on matching)
7. Guide the user to register the authenticators specified in the chosen combination

2. During authentication and transaction confirmation:

> NOTE
>
> `Policy.accepted` is a list of combinations. Each combination indicates a set of criteria which is enough to completely authenticate the current pending operation

1. Follow the priority of items in`Policy.accepted[][]`. The lists are ordered with highest priority first.
2. Choose the combination whose criteria best match the features of the currently available authenticators
3. Collect information about available authenticators
4. Ignore authenticators which meet the `Policy.disallowed` criteria
5. Match collected information with the matching criteria described in the policy
6. Guide the user to authenticate with the authenticators specified in the chosen combination
7. A pending operation will be approved by the server only after all criteria of a single combination are entirely met

3.2.1 Examples

This section is non-normative.

EXAMPLE 2: Policy matching either a FPS-, or Face Recognition-based Authenticator

```
{
  "accepted":
   [
 [{ "userVerification": 2, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}],
     [{ "userVerification": 16, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}]
   ]
}
```

EXAMPLE 3: Policy matching authenticators implementing FPS and Face Recognition as alternative combination of user verification methods.

```
{
  "accepted":
   [
 [{ "userVerification": 18, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}]
   ]
}
```

Combining these two bit-flags and the flag `USER_VERIFY_ALL` (USER_VERIFY_ALL = 1024) into a single `userVerification` value would match authenticators implementing FPS and Face Recognition as a *mandatory* combination of user verification methods.

EXAMPLE 4: Policy matching authenticators implementing FPS and Face Recognition as mandatory combination of user verification methods.

```
{
  "accepted": [ [{ "userVerification": 1042, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}] ]
}
```

The next example requires two authenticators to be used:

EXAMPLE 5: Policy matching the combination of a FPS based and a Face Recognition based authenticator

```
{
  "accepted":
   [
  [
   { "userVerification": 2, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]},
   { "userVerification": 16, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}
  ]
   ]
}
```

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

EXAMPLE 6: Policy requiring the combination of a bound FPS based and a bound Face Recognition based authenticator

```
{
  "accepted":
   [
  [
   { "userVerification": 2, "attachmentHint": 1, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]},
   { "userVerification": 16, "attachmentHint": 1, "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}
        ]
   ]
}
```

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

EXAMPLE 7: Policy accepting all authenticators from vendor with ID 1234

```
{
  "accepted":
  [ [{ "vendorID": "1234", "authenticationAlgorithms": [1, 2, 5, 6], "assertionSchemes": ["UAFV1TLV"]}] ]
}
```

3.3 Version Negotiation

The UAF protocol includes multiple versioned constructs: UAF protocol version, the version of Key Registration Data and Signed Data objects (identified by their respective tags, see [UAFRegistry]), and the ASM version, see [UAFASM].

> NOTE
>
> The Key Registration Data and Signed Data objects have to be parsed and verified by the FIDO Server. This verification is only possible if the FIDO Server understands their encoding and the content. Each UAF protocol version supports a set of Key Registration Data and SignedData object versions (called Assertion Schemes). Similarly each of the ASM versions supports a set Assertion Scheme versions.

As a consequence the FIDO UAF Client must select the authenticators which will generate the appropriately versioned constructs.

For version negotiation the FIDO UAF Client must perform the following steps:

1. Create a set (`FC_Version_Set`) of version pairs, ASM version (`asmVersion`) and UAF Protocol version (`upv`) and add all pairs supported by the FIDO UAF Client into `FC_Version_Set`
   - e.g. `[{upv1, asmVersion1}, {upv2, asmVersion1}, ...]`

   > NOTE
   >
   > The ASM versions are retrieved from the `AuthenticatorInfo.asmVersion` field. The UAF protocol version is derived from the related `AuthenticatorInfo.assertionScheme` field.

2. Intersect `FC_Version_Set` with the set of `upv` included in UAF Message (i.e. keep only those pairs where the `upv` value is also contained in the UAF Message).
3. Select authenticators which are allowed by the UAF Message Policy. For each authenticator:
   - Construct a set (`Authnr_Version_Set`) of version pairs including authenticator supported `asmVersion` and the compatible `upv(s)`.
     - e.g. `[{upv1, asmVersion1}, {upv2, asmVersion1}, ...]`
   - Intersect `Authnr_Version_Set` with `FC_Version_Set` and select highest version pair from it.
     - Take the pair where the `upv` is highest. In all these pairs leave only the one with highest `asmVersion`.
   - Use the remaining version pair with this authenticator

> NOTE
>
> Each version consists of `major` and `minor` fields. In order to compare two versions - compare the Major fields and if they are equal compare the Minor fields.
>
> Each UAF message contains a version field `upv`. UAF Protocol version negotiation is always between FIDO UAF Client and FIDO Server.
>
> > A possible implementation optimization is to have the RP web application itself preemptively convey to the FIDO Server the UAF protocol version(s) (UPV) supported by the FIDO Client. This allows the FIDO Server to craft its UAF messages using the UAF version most preferred by both the FIDO client and server.

3.4 Registration Operation

> NOTE
>
> The Registration operation allows the FIDO Server and the FIDO Authenticator to agree on an authentication key.

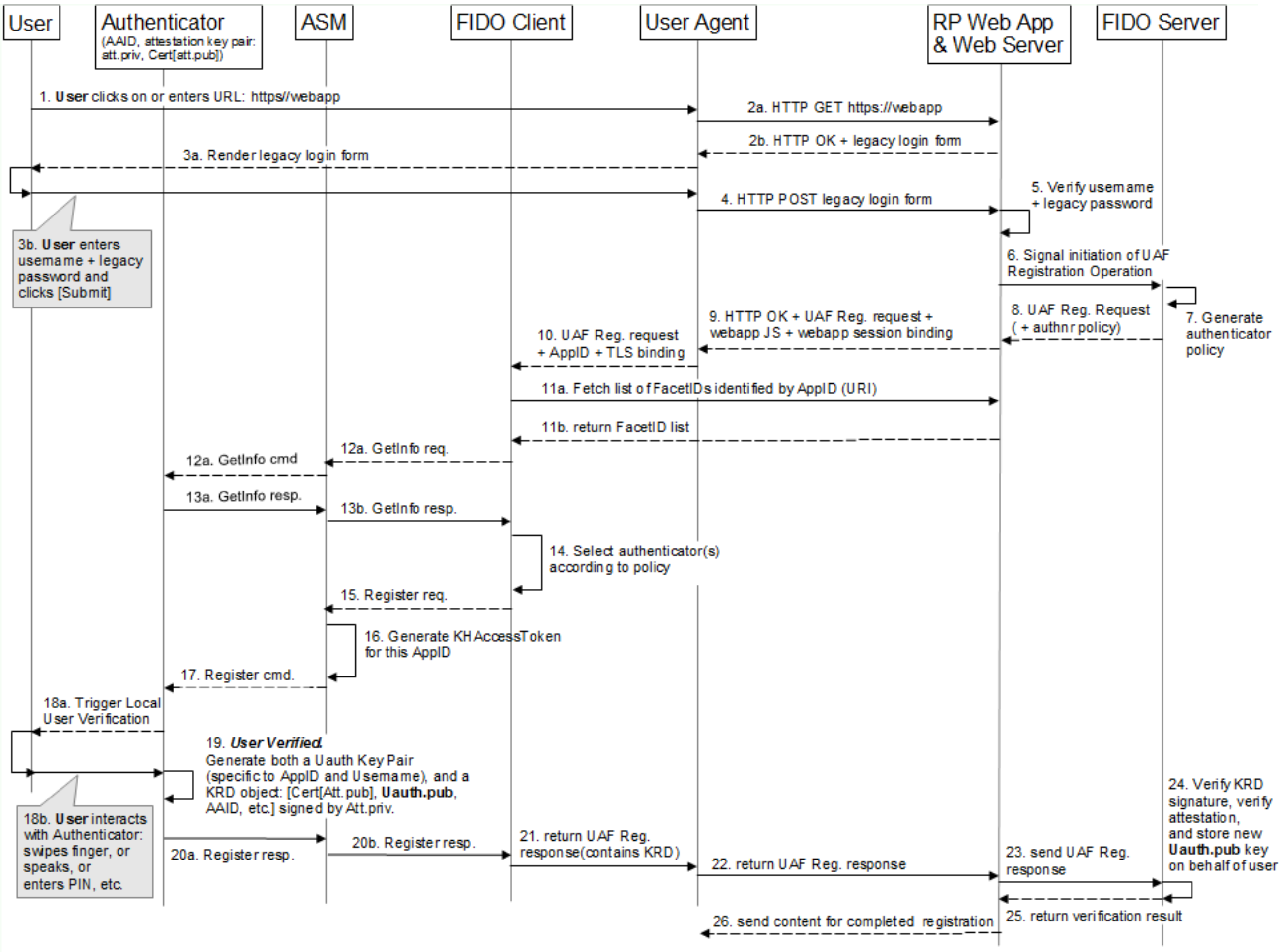


Fig. 6 UAF Registration Sequence Diagram

The steps 11a and 11b and 12 to 13 are not always necessary as the related data could be cached.

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

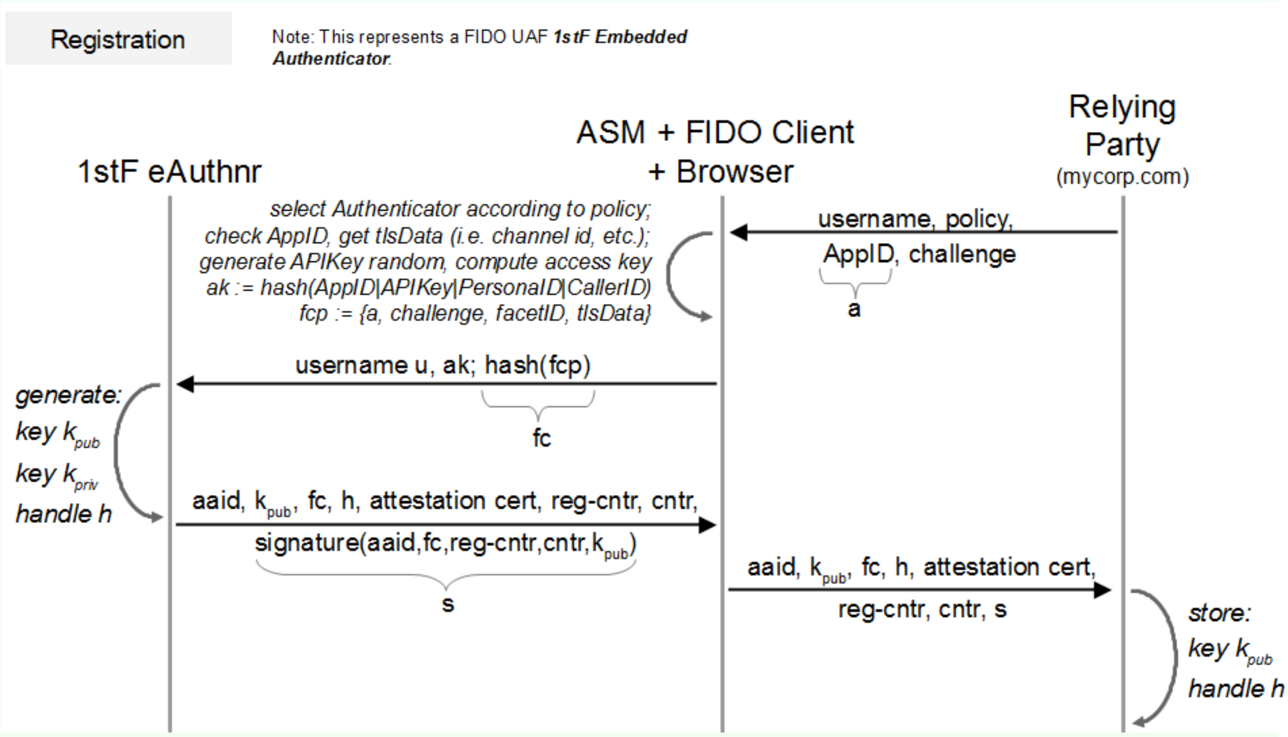


Fig. 7 UAF Registration Cryptographic Data Flow

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

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

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

3.4.1 Registration Request Message

UAF Registration request message is represented as an array of dictionaries. The array must contain exactly one dictionary. The request is defined as RegistrationRequest dictionary.

EXAMPLE 8: UAF Registration Request


```
[{
    "header": {
      "upv": {
        "major": 1,
        "minor": 1
      },
      "op": "Reg",
 "appID": "https://uaf-test-1.noknoktest.com:8443/SampleApp/uaf/facets",
      "serverData": "IjycjPZYiWMaQ1tKLrJROiXQHmYG0tSSYGjP5mgjsDaM17RQgq0
dl3NNDDTx9d-aSR_6hGgclrU2F2Yj-12S67v5VmQHj4eWVseLulHdpk2v_hHtKSvv_DFqL4n
2IiUY6XZWVbOnvg"
    },
    "challenge": "H9iW9yA9aAXF_lelQoi_DhUk514Ad8Tqv0zCnCqKDpo",
    "username": "apa",
    "policy": {
      "accepted": [
        [
          {
            "userVerification": 512,
            "keyProtection": 1,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              1
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          }
        ],
        [
          {
            "userVerification": 4,
            "keyProtection": 1,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              1
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          }
        ],
        [
          {
            "userVerification": 4,
            "keyProtection": 1,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              2
            ]
          }
        ],
        [
          {
            "userVerification": 2,
            "keyProtection": 4,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              2
            ]
          }
        ],
        [
          {
            "userVerification": 4,
            "keyProtection": 2,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              1,
              3
            ]
          }
        ],
        [
          {
            "userVerification": 2,
            "keyProtection": 2,
            "authenticationAlgorithms": [
              2
            ]
          }
        ],
        [
          {
            "userVerification": 32,
            "keyProtection": 2,
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          },
          {
            "userVerification": 2,
            "authenticationAlgorithms": [
              1,
              3
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          },
          {
            "userVerification": 2,
            "authenticationAlgorithms": [
              1,
              3
            ],
```

```
                "assertionSchemes": [
                  "UAFV1TLV"
                ]
              },
              {
                "userVerification": 4,
                "keyProtection": 1,
                "authenticationAlgorithms": [
                  1,
                  3
                ],
                "assertionSchemes": [
                  "UAFV1TLV"
                ]
              }
            ]
          ],
          "disallowed": [
            {
              "userVerification": 512,
              "keyProtection": 16,
              "assertionSchemes": [
                "UAFV1TLV"
              ]
            },
            {
              "userVerification": 256,
              "keyProtection": 16
            },
            {
              "aaid": [
                "ABCD#ABCD"
              ],
              "keyIDs": [
                "RfY_RDhsf4z5PCOhnZExMeVloZZmK0hxaSi10tkY_c4"
              ]
            }
          ]
        }
    }]
```


3.4.2 RegistrationRequest dictionary

RegistrationRequest contains a single, versioned, registration request.

WebIDL

```
dictionary RegistrationRequest {
    required OperationHeader header;
    required ServerChallenge challenge;
    required DOMString       username;
    required Policy          policy;
};
```

3.4.2.1 Dictionary `RegistrationRequest` Members

`header` of type required OperationHeader
: Operation header. `Header.op` must be "Reg"

`challenge` of type required ServerChallenge
: Server-provided challenge value

`username` of type required DOMString
: `string[1..128]`

  A human-readable user name intended to allow the user to distinguish and select from among different accounts at the same relying party.

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

3.4.3 AuthenticatorRegistrationAssertion dictionary

Contains the authenticator's response to a RegistrationRequest message:

WebIDL

```
dictionary AuthenticatorRegistrationAssertion {
    required DOMString                   assertionScheme;
    required DOMString                   assertion;
    DisplayPNGCharacteristicsDescriptor[] tcDisplayPNGCharacteristics;
    Extension[]                          exts;
};
```

3.4.3.1 Dictionary `AuthenticatorRegistrationAssertion` Members

`assertionScheme` of type required DOMString
: The name of the Assertion Scheme used to encode the`assertion`. See UAF Supported Assertion Schemes for details.

  NOTE

  This assertionScheme is not part of a signed object and hence considered the *suspected* assertionScheme.

`assertion` of type required DOMString
: `base64url(byte[1..4096])` Contains the `TAG_UAFV1_REG_ASSERTION` object containing the assertion scheme specific

KeyRegistrationData (KRD) object which in turn contains the newly generated `UAuth.pub` and is signed by the Attestation Private Key.

This assertion must be generated by the authenticator and it must be used only in this Registration operation. The format of this assertion can vary from one assertion scheme to another (e.g. for "UAFV1TLV" assertion scheme it must be `TAG_UAFV1_KRD`).

`tcDisplayPNGCharacteristics` of type array of DisplayPNGCharacteristicsDescriptor
: Supported transaction PNG type [FIDOMetadataStatement]. For the definition of the DisplayPNGCharacteristicsDescriptor structure See [FIDOMetadataStatement].

`exts` of type array of *Extension*
: Contains Extensions prepared by the authenticator

3.4.4 Registration Response Message

A UAF Registration response message is represented as an array of dictionaries. Each dictionary contains a registration response for a specific protocol version. The array must not contain two dictionaries of the same protocol version. The response is defined as RegistrationResponse dictionary.

EXAMPLE 9: Registration Response


```
[{
    "assertions": [
      {
        "assertion": "AT7uAgM-sQALLgkAQUJDRCNBQkNEDi4HAAABAQEAAAEKLiAA9t
BzZC64ecgVQBGSQb5QtEIPC8-Vav4HsHLZDflLaugJLiAAZMCPn92yHv1Ip-iCiBb6i4ADq6
ZOv569KFQCvYSJfNgNLggAAQAAAAEAAAAMLkEABJsvEtUsVKh7tmYHhJ2FBm3kHU-OCdWiUY
VijgYa8lMfkjQ1z6UiHbKP9_nRzIN9anprHqDGcR6q7O20q_yctZAHPjUCBi5AACv8L7YlRM
x10gPnszGO6rLFqZFmmRkhtV0TIWuWqYxd1jO0wxam7i5qdEa19u4sfpHFZ9RGI_WHxINkH8
FfvAwFLu0BMIIB6TCCAY8CAQEwCQYHKoZIzj0EATB7MQswCQYDVQQGEwJVUzELMAkGA1UECA
wCQ0ExCzAJBgNVBAcMAlBBMRAwDgYDVQQKDAdOTkwsSW5jMQ0wCwYDVQQLDAREQU4xMRMwEQ
YDVQQDDApOTkwsSW5jIENBMRwwGgYJKoZIhvcNAQkBFg1ubmxAZ21haWwuY29tMB4XDTE0MD
gyODIxMzU0MFoXDTE3MDUyNDIxMzU0MFowgYYxCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJDQT
EWMBQGA1UEBwwNU2FuIEZyYW5jaXNjbzEQMA4GA1UECgwHTk5MLEluYzENMAsGA1UECwwERE
FOMTETMBEGA1UEAwwKTk5MLEluYyBDQTEcMBoGCSqGSIb3DQEJARYNbm5sQGdtYWlsLmNvbT
BZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCGBt3CIjnDowzSiF68C2aErYXnDUsWXOYxqIP
im0OWg9FFdUYCa6AgKjn1R99Ek2d803sGKROivnavmdVH-SnEwCQYHKoZIzj0EAQNJADBGAi
EAzAQujXnSS9AIAh6lGz6ydypLVTsTnBzqGJ4ypIqy_qUCIQCFsuOEGcRV-o4GHPBph_VMrG
3NpYh2GKPjsAim_cSNmQ",
        "assertionScheme": "UAFV1TLV"
      }
    ],
    "fcParams": "eyJhcHBJRCI6Imh0dHBzOi8vdWFmLXRlc3QtMS5ub2tub2t0ZXN0LmN
vbTo4NDQzL1NhbXBsZUFwcC91YWYvZmFjZXRzIiwiY2hhbGxlbmdlIjoiSDlpVzl5QTlhQVh
GX2xlbFFvaV9EaFVrNTE0QWQ4VHF2MHpDbkNxS0RwbyIsImNoYW5uZWxCaW5kaW5nIjp7fSw
iZmFjZXRJRCI6ImNvbS5ub2tub2suYW5kcm9pZC5zYW1wbGVhcHAifQ",
    "header": {
  "appID": "https://uaf-test-1.noknoktest.com:8443/SampleApp/uaf/facets",
      "op": "Reg",
      "serverData": "IjycjPZYiWMaQ1tKLrJROiXQHmYG0tSSYGjP5mgjsDaM17RQgq0
dl3NNDDTx9d-aSR_6hGgclrU2F2Yj-12S67v5VmQHj4eWVseLulHdpk2v_hHtKSvv_DFqL4n
2IiUY6XZWVbOnvg",
      "upv": {
        "major": 1,
        "minor": 1
      }
    }
}]
```


NOTE

Line breaks in fcParams have been inserted for improving readability.

3.4.5 RegistrationResponse dictionary

Contains all fields related to the registration response.

WebIDL

```
dictionary RegistrationResponse {
    required OperationHeader                    header;
    required DOMString                          fcParams;
    required AuthenticatorRegistrationAssertion[] assertions;
};
```

3.4.5.1 Dictionary `RegistrationResponse` Members

`header` of type required OperationHeader
: `Header.op` must be "Reg".

`fcParams` of type required DOMString
: The base64url-encoded serialized [RFC4627] `FinalChallengeParams` using UTF8 encoding (see FinalChallengeParams dictionary) which contains all parameters required for the server to verify the Final Challenge.

`assertions` of type array of required AuthenticatorRegistrationAssertion
: Response data for each Authenticator being registered.

3.4.6 Registration Processing Rules

3.4.6.1 Registration Request Generation Rules for FIDO Server

The policy contains a two-dimensional array of allowed `MatchCriteria` (see Policy). This array can be considered a list (first dimension) of sets (second dimension) of authenticators (identified by `MatchCriteria`). All authenticators in a specific set must be registered simultaneously in order to match the policy. But any of those sets in the list are valid, as the list elements are alternatives.

The FIDO Server must follow the following steps:

1. Construct appropriate authentication policy `p`
    1. for each set of alternative authenticators do
        1. Create an array of MatchCriteria objects, containing the set of authenticators to be registered simultaneously that need to be identified by *separate* MatchCriteria objects `m`.
            1. For each collection of authenticators `a` to be registered simultaneously that can be identified by the *same rule*, create a MatchCriteria object `m`, where
                - `m.aaid` may be combined with (one or more of)`m.keyIDs`, `m.attachmentHint`, `m.authenticatorVersion`, and `m.exts`, but `m.aaid` must not be combined with any other match criteria field.
                - If `m.aaid` is not provided - at least `m.authenticationAlgorithms` and `m.assertionSchemes` must be provided
            2. Add `m` to `v`, e.g. `v[j+1]=m`.
        2. Add `v` to `p.allowed`, e.g. `p.allowed[i+1]=v`
    2. Create MatchCriteria objects `m[]` for all disallowed Authenticators.
        1. For each already registered AAID for the current user
            1. Create a MatchCriteria object `m` and add AAID and corresponding KeyIDs to `m.aaid` and `m.KeyIDs`.

               The FIDO Server must include already registered AAIDs and KeyIDs into field `p.disallowed` to hint that the client should not register these again.
        2. Create a MatchCriteria object `m` and add the AAIDs of all disallowed Authenticators to `m.aaid`.

           The status (as provided in the metadata TOC (Table-of-Contents file) [FIDOMetadataService]) of some authenticators might be unacceptable. Such authenticators should be included in `p.disallowed`.
        3. If needed - create MatchCriteria `m` for other disallowed criteria (e.g. unsupported authenticationAlgs)
        4. Add all `m` to `p.disallowed`.
2. Create a `RegistrationRequest` object `r` with appropriate `r.header` for each supported version, and
    1. FIDO Servers should not assume any implicit integrity protection of `r.header.serverData`.

       FIDO Servers that depend on the integrity of `r.header.serverData` should apply and verify a cryptographically secure Message Authentication Code (MAC) to serverData and they should also cryptographically bind serverData to the related message, e.g. by re-including `r.challenge`, see also section ServerData and KeyHandle.

       > NOTE
       >
       > All other FIDO components (except the FIDO server) will treat `r.header.serverData` as an opaque value. As a consequence the FIDO server can implement any suitable cryptographic protection method.
    2. Generate a random challenge and assign it to `r.challenge`
    3. Assign the username of the user to be registered to `r.username`
    4. Assign `p` to `r.policy`.
    5. Append `r` to the array `o` of message with various versions (`RegistrationRequest`)
3. Send `o` to the FIDO UAF Client

3.4.6.2 Registration Request Processing Rules for FIDO UAF Clients

The FIDO UAF Client must perform the following steps:

1. Choose the message `m` with upv set to the appropriate version number.
2. Parse the message `m`
3. If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
4. Filter the available authenticators with the given policy and present the filtered authenticators to User. Make sure to not include already registered authenticators for this user specified in `RegRequest.policy.disallowed[].keyIDs`
5. Obtain `FacetID` of the requesting Application. If the `AppID` is missing or empty, set the `AppID` to the `FacetID`.

   Verify that the `FacetID` is authorized for the `AppID` according to the algorithms in [FIDOAppIDAndFacets].

    - If the `FacetID` of the requesting Application is not authorized, reject the operation
6. Obtain TLS data if it is available
7. Create a **`FinalChallengeParams`** structure `fcp` and set `fcp.appID`, `fcp.challenge`, `fcp.facetID`, and `fcp.channelBinding` appropriately. Serialize [RFC4627] fcp using UTF8 encoding and base64url encode it.
    - `FinalChallenge = base64url(serialize(utf8encode(fcp)))`
8. For each authenticator that matches UAF protocol version (see section Version Negotiation) and user agrees to register:
    1. Add `AppID`, `Username`, `FinalChallenge`, `AttestationType` and all other required fields to the ASMRequest [UAFASM].

       The FIDO UAF Client must follow the server policy and find the single preferred attestation type. A single attestation type must be provided to the ASM.
    2. Send the ASMRequest to the ASM. If the ASM returns an error, handle that error appropriately. The status code returned by the ASM [UAFASM] must be mapped to a status code defined in [UAFAppAPIAndTransport] as specified in section 3.4.6.2.1 Mapping ASM Status Codes to ErrorCode.

3.4.6.2.1 Mapping ASM Status Codes to ErrorCode

ASMs are returning a status code in their responses to the FIDO Client. The FIDO Client needs to act on those responses and also map the status code returned the ASM [UAFASM] to an ErrorCode specified in [UAFAppAPIAndTransport].

The mapping of ASM status codes to ErrorCode is specified here:

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

3.4.6.3 Registration Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section "Register Command".

3.4.6.4 Registration Response Generation Rules for FIDO UAF Client

The FIDO UAF Client must follow the steps:

1. Create a `RegistrationResponse` message
2. Copy `RegistrationRequest.header` into `RegistrationResponse.header`

   > NOTE
   >
   > When the `appID` provided in the request was empty, the FIDO Client must set the `appID` in this header to the facetID (see [FIDOAppIDAndFacets]).

3. Set `RegistrationResponse.fcParams` to `FinalChallenge` (base64url encoded serialized and utf8 encoded FinalChallengeParams)
4. Append the response from each Authenticator into `RegistrationResponse.assertions`
5. Send `RegistrationResponse` message to FIDO Server

3.4.6.5 Registration Response Processing Rules for FIDO Server

> NOTE
>
> The following processing rules assume that Authenticator supports "UAFV1TLV" assertion scheme. Currently "UAFV1TLV" is the only defined and supported assertion scheme. When a new assertion scheme is added to UAF protocol - this section will be extended with corresponding processing rules.

The FIDO Server must follow the steps:

1. Parse the message
   1. If protocol version (`RegistrationResponse.header.upv`) is not supported – reject the operation
   2. If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
2. Verify that `RegistrationResponse.header.serverData`, if used, passes any implementation-specific checks against its validity. See also

section ServerData and KeyHandle.

3. base64url decode `RegistrationResponse.fcParams` and convert it into an object (`fcp`)
4. Verify each field in fcp and make sure it is valid:
    1. Make sure `fcp.appID` corresponds to the one stored by the FIDO Server

        NOTE

        When the `appID` provided in the request was empty, the FIDO Client must set the `appID` to the facetID (see [FIDOAppIDAndFacets]). In this case, the Uauth key cannot be used by other application facets.

    2. Make sure `fcp.facetID` is in the list of trusted FacetIDs [FIDOAppIDAndFacets]
    3. Make sure `fcp.channelBinding` is as expected (see section ChannelBinding dictionary)

        NOTE

        There might be legitimate situations in which some methods of channel binding fail (see section 4.3.4 TLS Binding).

    4. Reject the response if any of these checks fails
    5. Make sure `fcp.challenge` has really been generated by the FIDO Server for this operation and it is not expired
5. For each assertion `a` in `RegistrationResponse.assertions`
    1. Parse TLV data from `a.assertion` assuming it is encoded according to the suspected assertion scheme `a.assertionScheme` and make sure it contains all mandatory fields (indicated in Authenticator Metadata) it is supposed to have and has a valid syntax.
        - If it doesn't - continue with next assertion
    2. Retrieve the AAID from the assertion.

        NOTE

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

    3. Verify that `a.assertionScheme` matches `Metadata(AAID).assertionScheme`
        - If it doesn't match - continue with next assertion
    4. Verify that the AAID indeed matches the policy specified in the registration request.

        NOTE

        Depending on the policy (e.g. in the case of AND combinations), it might be required to evaluate other assertions included in this `RegistrationResponse` in order to determine whether this AAID matches the policy.

        - If it doesn't match the policy - continue with next assertion
    5. Locate authenticator-specific authentication algorithms from the authenticator metadata [FIDOMetadataStatement] using the AAID.
    6. Hash `RegistrationResponse.fcParams` using hashing algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator metadata, field `AuthenticationAlgs`. It is the hash algorithm associated with the first entry related to a constant with prefix ALG_SIGN.
        - `FCHash = hash(RegistrationResponse.fcParams)`
    7. if `a.assertion` contains an object of type `TAG_UAFV1_REG_ASSERTION`, then
        1. if `a.assertion.TAG_UAFV1_REG_ASSERTION` contains `TAG_UAFV1_KRD` as first element:
            1. Obtain `Metadata(AAID).AttestationType` for the AAID and make sure that `a.assertion.TAG_UAFV1_REG_ASSERTION` contains the most preferred attestation tag specified in field `MatchCriteria.attestationTypes` in `RegistrationRequest.policy` (if this field is present).
                - If `a.assertion.TAG_UAFV1_REG_ASSERTION` doesn't contain the preferred attestation - it is recommended to skip this assertion and continue with next one
            2. Make sure that `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.FinalChallengeHash` == `FCHash`
                - If comparison fails - continue with next assertion
            3. Obtain `Metadata(AAID).AuthenticatorVersion` for the AAID and make sure that it is lower or equal to `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.AuthenticatorVersion`.
                - If `Metadata(AAID).AuthenticatorVersion` is higher (i.e. the authenticator firmware is outdated), it is recommended to assume increased risk. See sections "StatusReport dictionary" and "Metadata TOC object Processing Rules" in [FIDOMetadataService] for more details on this.
            4. Check whether `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.RegCounter` is acceptable, i.e. it is either not supported (value is 0 or the field isKeyRestricted is set to 'false' in the related Metadata Statement) or it is not exceedingly high
                - If `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.RegCounter` is exceedingly high, this assertion might be skipped and processing will continue with next one
            5. If `a.assertion.TAG_UAFV1_REG_ASSERTION` contains `TAG_ATTESTATION_BASIC_FULL` tag
                1. If entry `AttestationRootCertificates` for the AAID in the metadata [FIDOMetadataStatement] contains at least one element:
                    1. Obtain contents of all `TAG_ATTESTATION_CERT` tags from `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_ATTESTATION_BASIC_FULL` object. The occurrences are ordered (see [UAFAuthnrCommands]) and represent the attestation certificate followed by the related certificate chain.
                    2. Obtain all entries of `AttestationRootCertificates` for the AAID in authenticator Metadata, field `AttestationRootCertificates`.
                    3. Verify the attestation certificate and the entire certificate chain up to the Attestation Root Certificate using Certificate Path Validation as specified in [RFC5280]

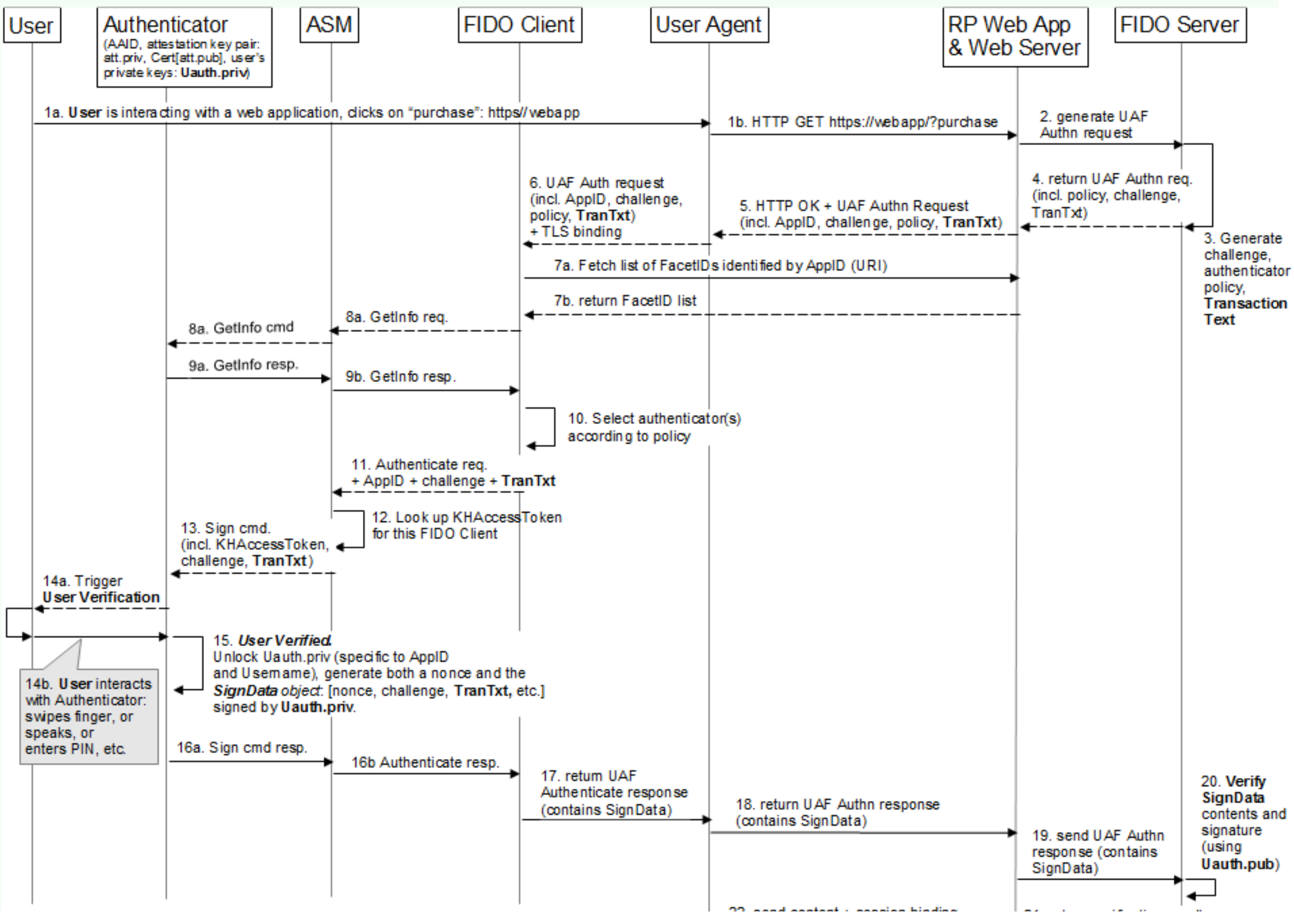
        - If verification fails – continue with next assertion
      4. Verify `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_ATTESTATION_BASIC_FULL.Signature` using the attestation certificate (obtained before).
        - If verification fails – continue with next assertion
    2. If `Metadata(AAID).AttestationRootCertificates` for this AAID is empty - continue with next assertion
    3. Mark assertion as positively verified
  6. If `a.assertion.TAG_UAFV1_REG_ASSERTION` contains an object of type `TAG_ATTESTATION_BASIC_SURROGATE`
    1. There is no real attestation for the AAID, so we just assume the AAID is the real one.
    2. If entry `AttestationRootCertificates` for the AAID in the metadata is empty
      - Verify `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_ATTESTATION_BASIC_SURROGATE.Signature` using `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_PUB_KEY`
        - If verification fails – continue with next assertion
    3. If entry `AttestationRootCertificates` for the AAID in the metadata is not empty - continue with next assertion (as the AAID obviously is expecting a different attestation method).
    4. Mark assertion as positively verified
  7. If `a.assertion.TAG_UAFV1_REG_ASSERTION` contains an object of type `TAG_ATTESTATION_ECDAA`
    1. If entry `ecdaaTrustAnchors` for the AAID in the metadata [FIDOMetadataStatement] contains at least one element:
      1. For each of the `ecdaaTrustAnchors` entries, perform the ECDAA Verify operation as specified in [FIDOEcdaaAlgorithm].
        - If verification fails – continue with next `ecdaaTrustAnchors` entry
      2. If no ECDAA Verify operation succeeded – continue with next assertion
    2. If `Metadata(AAID).ecdaaTrustAnchors` for this AAID is empty - continue with next assertion
    3. Mark assertion as positively verified and the authenticator indeed is of model as indicated by the AAID.
  8. If `a.assertion.TAG_UAFV1_REG_ASSERTION` contains another `TAG_ATTESTATION` tag - verify the attestation by following appropriate processing rules applicable to that attestation. Currently this document defines the processing rules for Basic Attestation and direct anonymous attestation (ECDAA).
2. if `a.assertion.TAG_UAFV1_REG_ASSERTION` contains a different object than `TAG_UAFV1_KRD` as first element, then follow the rules specific to that object.
3. Extract `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.PublicKey` into PublicKey, `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.KeyID` into KeyID, `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.SignCounter` into SignCounter, `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_ASSERTION_INFO.authenticatorVersion` into AuthenticatorVersion, `a.assertion.TAG_UAFV1_REG_ASSERTION.TAG_UAFV1_KRD.TAG_AAID` into AAID.

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

6. For each positively verified assertion `a`
   - Store PublicKey, KeyID, SignCounter, AuthenticatorVersion, AAID and `a.tcDisplayPNGCharacteristics` into a record associated with the user's identity. If an entry with the same pair of AAID and KeyID already exists then fail (should never occur).

3.5 Authentication Operation

NOTE



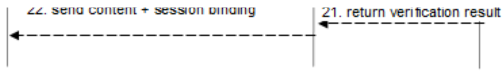


Fig. 8 UAF Authentication Sequence Diagram

The steps 7a and 7a and 8 to 9 are not always necessary as the related data could be cached.

The TransactionText (TranTxt) is only required in the case of Transaction Confirmation (see section 3.5.1 Transaction dictionary), it is absent in the case of a pure Authenticate operation.

During this operation, the FIDO Server asks the FIDO UAF Client to authenticate user with server-specified authenticators, and return an authentication response.

In order for this operation to succeed, the authenticator and the relying party must have a previously shared registration.

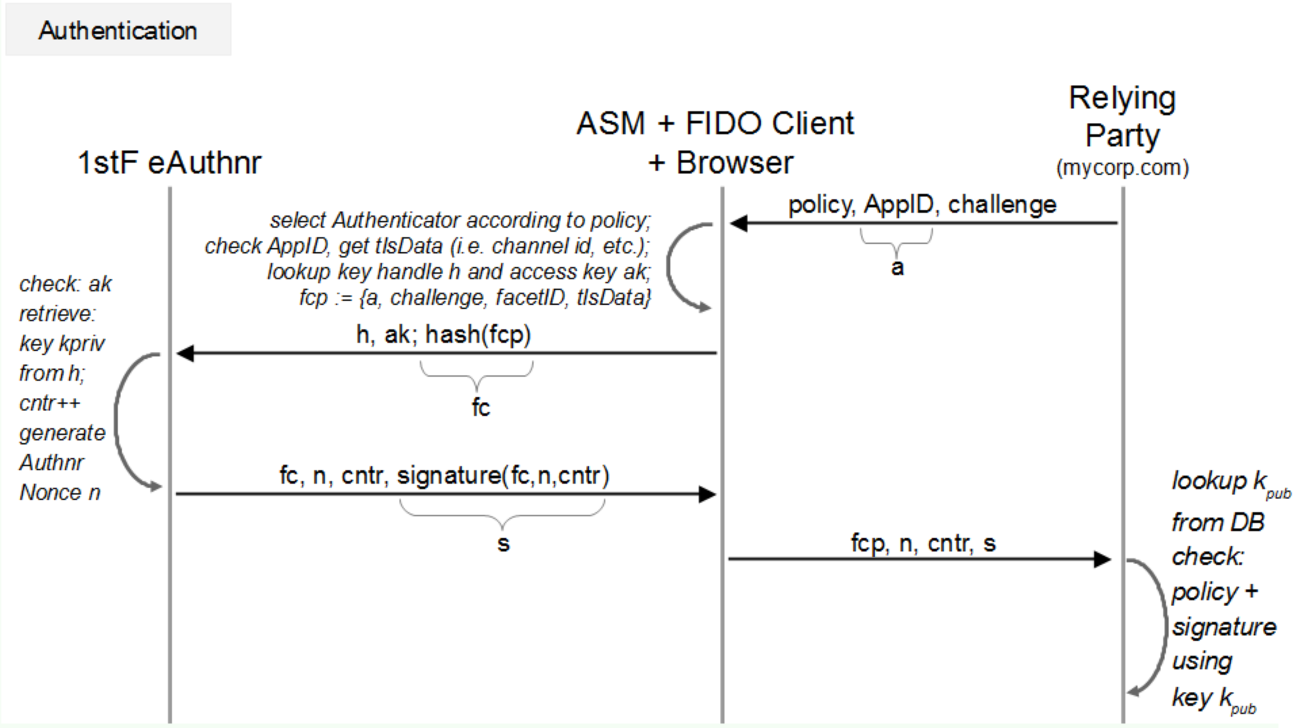


Fig. 9 UAF Authentication Cryptographic Data Flow

Diagram of cryptographic flow:

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

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

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

3.5.1 Transaction dictionary

Contains the Transaction Content provided by the FIDO Server:

WebIDL

```
dictionary Transaction {
    required DOMString                   contentType;
    required DOMString                   content;
    DisplayPNGCharacteristicsDescriptor tcDisplayPNGCharacteristics;
};
```

3.5.1.1 Dictionary **`Transaction`** *Members*

`contentType` of type required DOMString

Contains the MIME Content-Type supported by the authenticator according its metadata statement (see [FIDOMetadataStatement]).

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

`content` of type required DOMString

`base64url(byte[1...])`

Contains the base64-url encoded transaction content according to the `contentType` to be shown to the user.

If `contentType` is "text/plain" then the content must be the base64-url encoding of the ASCII encoded text with a maximum of 200 characters.

`tcDisplayPNGCharacteristics` of type DisplayPNGCharacteristicsDescriptor

Transaction content PNG characteristics. For the definition of the DisplayPNGCharacteristicsDescriptor structure See

[FIDOMetadataStatement]. This field must be present if the contentType is "image/png".

3.5.2 Authentication Request Message

UAF Authentication request message is represented as an array of dictionaries. The array must contain exactly one dictionary. The request is defined as AuthenticationRequest dictionary.

EXAMPLE 10: UAF Authentication Request

```
[{
    "header": {
      "upv": {
        "major": 1,
        "minor": 1
      },
      "op": "Auth",
 "appID": "https://uaf-test-1.noknoktest.com:8443/SampleApp/uaf/facets",
      "serverData": "5s7n8-7_LDAtRIKKYqbAtTTOezVKCjl2mPorYzbpxRrZ-_3wWro
MXsF_pLYjNVm_l7bplAx4bkEwK6ibil9EHGfdfKOQ1q0tyEkNJFOgqdjVmLioroxgThlj8Is
tpt7q"
    },
    "challenge": "HQ1VkTUQC1NJDOo6OOWdxewrb9i5WthjfKIehFxpeuU",
    "policy": {
      "accepted": [
        [
          {
            "userVerification": 512,
            "keyProtection": 1,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              1
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          }
        ],
        [
          {
            "userVerification": 4,
            "keyProtection": 1,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              1
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          }
        ],
        [
          {
            "userVerification": 4,
            "keyProtection": 1,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              2
            ]
          }
        ],
        [
          {
            "userVerification": 2,
            "keyProtection": 4,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              2
            ]
          }
        ],
        [
          {
            "userVerification": 4,
            "keyProtection": 2,
            "tcDisplay": 1,
            "authenticationAlgorithms": [
              1,
              3
            ]
          }
        ],
        [
          {
            "userVerification": 2,
            "keyProtection": 2,
            "authenticationAlgorithms": [
              2
            ]
          }
        ],
        [
          {
            "userVerification": 32,
            "keyProtection": 2,
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          },
          {
            "userVerification": 2,
            "authenticationAlgorithms": [
              1,
              3
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          },
          {
            "userVerification": 2,
            "authenticationAlgorithms": [
              1,
              3
            ],
            "assertionSchemes": [
```

```
              "UAFV1TLV"
            ]
          },
          {
            "userVerification": 4,
            "keyProtection": 1,
            "authenticationAlgorithms": [
              1,
              3
            ],
            "assertionSchemes": [
              "UAFV1TLV"
            ]
          }
        ]
      ],
      "disallowed": [
        {
          "userVerification": 512,
          "keyProtection": 16,
          "assertionSchemes": [
            "UAFV1TLV"
          ]
        },
        {
          "userVerification": 256,
          "keyProtection": 16
        }
      ]
    }
}]
```


3.5.3 AuthenticationRequest dictionary

Contains the UAF Authentication Request Message:

WebIDL

```
dictionary AuthenticationRequest {
    required OperationHeader header;
    required ServerChallenge challenge;
    Transaction[]            transaction;
    required Policy          policy;
};
```

3.5.3.1 Dictionary `AuthenticationRequest` Members

`header` of type required OperationHeader
: `Header.op` must be "Auth"

`challenge` of type required ServerChallenge
: Server-provided challenge value

`transaction` of type array of *Transaction*
: Transaction data to be explicitly confirmed by the user.

  The list contains the same transaction content in various content types and various image sizes. Refer to [FIDOMetadataStatement] for more information about Transaction Confirmation Display characteristics.

`policy` of type required Policy
: Server-provided policy defining what types of authenticators are acceptable for this authentication operation.

3.5.4 AuthenticatorSignAssertion dictionary

Represents a response generated by a specific Authenticator:

WebIDL

```
dictionary AuthenticatorSignAssertion {
    required DOMString assertionScheme;
    required DOMString assertion;
    Extension[]        exts;
};
```

3.5.4.1 Dictionary `AuthenticatorSignAssertion` Members

`assertionScheme` of type required DOMString
: The name of the Assertion Scheme used to encode `assertion`. See UAF Supported Assertion Schemes for details.

  NOTE

  This assertionScheme is not part of a signed object and hence considered the *suspected* assertionScheme.

`assertion` of type required DOMString
: `base64url(byte[1..4096])` Contains the assertion containing a signature generated by `UAuth.priv`, i.e. `TAG_UAFV1_AUTH_ASSERTION`.

`exts` of type array of *Extension*
: Any extensions prepared by the Authenticator

3.5.5 AuthenticationResponse dictionary

Represents the response to a challenge, including the set of signed assertions from registered authenticators.

```
WebIDL

dictionary AuthenticationResponse {
    required OperationHeader               header;
    required DOMString                     fcParams;
    required AuthenticatorSignAssertion[]  assertions;
};
```

3.5.5.1 Dictionary `AuthenticationResponse` Members

`header` of type required OperationHeader
: `Header.op` must be "Auth"

`fcParams` of type required DOMString
: The field fcParams is the base64url-encoded serialized [RFC4627] FinalChallengeParams in UTF8 encoding (see FinalChallengeParams dictionary) which contains all parameters required for the server to verify the Final Challenge.

`assertions` of type array of required AuthenticatorSignAssertion
: The list of authenticator responses related to this operation.

3.5.6 Authentication Response Message

UAF Authentication response message is represented as an array of dictionaries. The array must contain exactly one dictionary. The response is defined as AuthenticationResponse dictionary.

EXAMPLE 11: UAF Authentication Response


```
[{
    "assertions": [
      {
        "assertion": "Aj7WAAQ-jgALLgkAQUJDRCNBQkNEDi4FAAABAQEADy4gAHwyJA
EX8t1b2wOxbaKOC5ZL7ACqbLo_TtiQfK3DzDsHCi4gAFwCUz-dOuafXKXJLbkUrIzjAU6oDb
P8B9iLQRmCf58fEC4AAAkuIABkwI-f3bIe_Uin6IKIFvqLgAOrpk6_nr0oVAK9hIl82A0uBA
ACAAAABi5AADwDOcBvPslX2bRNy4SvFhAwhEAoBSGUitgMUNChgUSMxss3K3ukekq1paG7Fv
1v5mBmDCZVPt2NCTnjUxrjTp4",
        "assertionScheme": "UAFV1TLV"
      }
    ],
    "fcParams": "eyJhcHBJRCI6Imh0dHBzOi8vdWFmLXRlc3QtMS5ub2tub2t0ZXN0LmN
vbTo4NDQzL1NhbXBsZUFwcC91YWYvZmFjZXRzIiwiY2hhbGxlbmdlIjoiSFExVmtUVVFDMU5
KRE9vNk9PV2R4ZXdyYjlpNVd0aGpmS0llaEZ4cGV1VSIsImNoYW5uZWxCaW5kaW5nIjp7fSw
iZmFjZXRJRCI6ImNvbS5ub2tub2suYW5kcm9pZC5zYW1wbGVhcHAifQ",
    "header": {
  "appID": "https://uaf-test-1.noknoktest.com:8443/SampleApp/uaf/facets",
      "op": "Auth",
      "serverData": "5s7n8-7_LDAtRIKKYqbAtTTOezVKCjl2mPorYzbpxRrZ-_3wWro
MXsF_pLYjNVm_l7bplAx4bkEwK6ibil9EHGfdfKOQ1q0tyEkNJFOgqdjVmLioroxgThlj8Is
tpt7q",
      "upv": {
        "major": 1,
        "minor": 1
      }
    }
}]
```


NOTE

Line breaks in fcParams have been inserted for improving readability.

3.5.7 Authentication Processing Rules

3.5.7.1 Authentication Request Generation Rules for FIDO Server

The policy contains a 2-dimensional array of allowed MatchCriteria (see Policy). This array can be considered a list (first dimension) of sets (second dimension) of authenticators (identified by MatchCriteria). All authenticators in a specific set must be used for authentication simultaneously in order to match the policy. But any of those sets in the list are valid, i.e. the list elements are alternatives.

The FIDO Server must follow the steps:

1. Construct appropriate authentication policy `p`
    1. for each set of alternative authenticators do
        1. Create an 1-dimensional array of MatchCriteria objects v containing the set of authenticators to be used for authentication simultaneously that need to be identified by *separate* MatchCriteria objects `m`.
            1. For each collection of authenticators `a` to be used for authentication simultaneously that can be identified by the *same rule*, create a MatchCriteria object `m`, where
                - `m.aaid` may be combined with (one or more of) `m.keyIDs`, `m.attachmentHint`, `m.authenticatorVersion`, and `m.exts`, but `m.aaid` must not be combined with any other match criteria field.
                - If `m.aaid` is not provided - at least `m.authenticationAlgorithms` and `m.assertionSchemes` must be provided
                - In case of step-up authentication (i.e. in the case where it is expected the user is already known due to a previous authentication step) every item in `Policy.accepted` must include the `AAID` and `KeyID` of the authenticator registered for this account in order to avoid ambiguities when having multiple accounts at this relying party.
            2. Add `m` to `v`, e.g. `v[j+1]=m`.
        2. Add `v` to `p.allowed`, e.g. `p.allowed[i+1]=v`
    2. Create MatchCriteria objects `m[]` for all disallowed authenticators.
        1. Create a MatchCriteria object `m` and add AAIDs of all disallowed authenticators to `m.aaid`.

           The status (as provided in the metadata TOC [FIDOMetadataService]) of some authenticators might be unacceptable. Such authenticators should be included in `p.disallowed`.

   2. If needed - create MatchCriteria m for other disallowed criteria (e.g. unsupported authenticationAlgs)
   3. Add all m to p.disallowed.
2. Create an AuthenticationRequest object r with appropriate r.header for the supported version, and
   1. FIDO Servers should not assume any implicit integrity protection of r.header.serverData. FIDO Servers that depend on the integrity of r.header.serverData should apply and verify a cryptographically secure Message Authentication Code (MAC) to serverData and they should also cryptographically bind serverData to the related message, e.g. by re-including r.challenge, see also section ServerData and KeyHandle.

      > NOTE
      >
      > All other FIDO components (except the FIDO server) will treat r.header.serverData as an opaque value. As a consequence the FIDO server can implement any suitable cryptographic protection method.

   2. Generate a random challenge and assign it to r.challenge
   3. If this is a transaction confirmation operation - look up TransactionConfirmationDisplayContentTypes/ TransactionConfirmationDisplayPNGCharacteristics from authenticator metadata of every participating AAID, generate a list of corresponding transaction content and insert the list into r.transaction.
      - If the authenticator reported (a dynamic) AuthenticatorRegistrationAssertion.tcDisplayPNGCharacteristics during Registration - it must be preferred over the (static) value specified in the authenticator Metadata.
   4. Set r.policy to our new policy object p created above, e.g. r.policy = p.
   5. Add the authentication request message the array
3. Send the array of authentication request messages to the FIDO UAF Client

3.5.7.2 Authentication Request Processing Rules for FIDO UAF Client

The FIDO UAF Client must follow the steps:

1. Choose the message m with upv set to the appropriate version number.
2. Parse the message m
   - If a mandatory field in the UAF message is not present or a field doesn't correspond to its type and value then reject the operation
3. Obtain FacetID of the requesting Application. If the AppID is missing or empty, set the AppID to the FacetID.

   Verify that the FacetID is authorized for the AppID according to the algorithms in [FIDOAppIDAndFacets].

   - If the FacetID of the requesting Application is not authorized, reject the operation
4. Filter available authenticators with the given policy and present the filtered list to User.
5. Let the user select the preferred Authenticator.
6. Obtain TLS data if its available
7. Create a FinalChallengeParams structure fcp and set fcp.AppID, fcp.challenge, fcp.facetID, and fcp.channelBinding appropriately. Serialize [RFC4627] fcp using UTF8 encoding and base64url encode it.
   - FinalChallenge = base64url(serialize(utf8encode(fcp)))
8. For each authenticator that supports an Authenticator Interface Version AIV compatible with message version AuthenticationRequest.header.upv (see Version Negotiation) and user agrees to authenticate with:
   1. Add AppID, FinalChallenge, Transactions (if present), and all other fields to the ASMRequest.
   2. Send the ASMRequest to the ASM. If the ASM returns an error, handle that error appropriately. The status code returned by the ASM [UAFASM] must be mapped to a status code defined in [UAFAppAPIAndTransport] as specified in section 3.4.6.2.1 Mapping ASM Status Codes to ErrorCode.

3.5.7.3 Authentication Request Processing Rules for FIDO Authenticator

See [UAFAuthnrCommands], section "Sign Command".

3.5.7.4 Authentication Response Generation Rules for FIDO UAF Client

The FIDO UAF Client must follow the steps:

1. Create an AuthenticationResponse message
2. Copy AuthenticationRequest.header into AuthenticationResponse.header

   > NOTE
   >
   > When the appID provided in the request was empty, the FIDO Client must set the appID in this header to the facetID (see [FIDOAppIDAndFacets]).

3. Fill out AuthenticationResponse.FinalChallengeParams with appropriate fields and then stringify it
4. Append the response from each authenticator into AuthenticationResponse.assertions
5. Send AuthenticationResponse message to the FIDO Server

3.5.7.5 Authentication Response Processing Rules for FIDO Server

> NOTE
>
> The following processing rules assume that authenticator supports "UAFV1TLV" assertion scheme. Currently "UAFV1TLV" is the only defined and supported assertion scheme. When a new assertion scheme is added to UAF protocol - this section will be extended with

corresponding processing rules.

The FIDO Server must follow the steps:

1. Parse the message
    1. If protocol version (`AuthenticationResponse.header.upv`) is not supported – reject the operation
    2. If a mandatory field in UAF message is not present or a field doesn't correspond to its type and value - reject the operation
2. Verify that `AuthenticationResponse.header.serverData`, if used, passes any implementation-specific checks against its validity. See also section ServerData and KeyHandle.
3. base64url decode `AuthenticationResponse.fcParams` and convert into an object (`fcp`)
4. Verify each field in fcp and make sure it's valid:
    1. Make sure `fcp.appID` corresponds to the one stored by the FIDO Server

       NOTE

       When the `appID` provided in the request was empty, the FIDO Client must set the `appID` to the facetID (see [FIDOAppIDAndFacets]). In this case, the Uauth key cannot be used by other application facets.

    2. Make sure `fcp.facetID` is in the list of trusted FacetIDs [FIDOAppIDAndFacets]
    3. Make sure `channelBinding` is as expected (see section ChannelBinding dictionary)

       NOTE

       There might be legitimate situations in which some methods of channel binding fail (see section 4.3.4 TLS Binding).

    4. Make sure `fcp.challenge` has really been generated by the FIDO Server for this operation and it is not expired
    5. Reject the response if any of the above checks fails
5. For each assertion `a` in `AuthenticationResponse.assertions`
    1. Parse TLV data from `a.assertion` assuming it is encoded according to the suspected assertion scheme `a.assertionScheme` and make sure it contains all mandatory fields (indicated in authenticator Metadata) it is supposed to have and has a valid syntax.
        - If it doesn't - continue with next assertion
    2. Retrieve the AAID from the assertion.

       NOTE

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

    3. Verify that `a.assertionScheme` matches `Metadata(AAID).assertionScheme`
        - If it doesn't match - continue with next assertion
    4. Make sure that the AAID indeed matches the policy of the Authentication Request
        - If it doesn't meet the policy – continue with next assertion
    5. if `a.assertion` contains an object of type `TAG_UAFV1_AUTH_ASSERTION`, then
        1. if `a.assertion.TAG_UAFV1_AUTH_ASSERTION` contains `TAG_UAFV1_SIGNED_DATA` as first element:
            1. Obtain `Metadata(AAID).AuthenticatorVersion` for this AAID and make sure that it is lower or equal to `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_ASSERTION_INFO.AuthenticatorVersion`.
                - If `Metadata(AAID).AuthenticatorVersion` is higher (i.e. the authenticator firmware is outdated), it is recommended to assume increased authentication risk. See "StatusReport dictionary" and "Metadata TOC object Processing Rules" in [FIDOMetadataService] for more details on this.
            2. Retrieve `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_KEYID` as KeyID
            3. Locate `UAuth.pub` public key associated with (AAID, KeyID) in the user's record.
                - If such record doesn't exist - continue with next assertion
            4. Verify the AAID against the AAID stored in the user's record at time of Registration.
                - If comparison fails – continue with next assertion
            5. Locate authenticator specific authentication algorithms from authenticator metadata (field `AuthenticationAlgs`)
            6. Check the Signature Counter `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.SignCounter` and make sure it is either not supported by the authenticator (i.e. the value provided and the value stored in the user's record are both 0 or the value isKeyRestricted is set to 'false' in the related Metadata Statement) or it has been incremented (compared to the value stored in the user's record)
                - If it is greater than 0, but didn't increment - continue with next assertion (as this is a cloned authenticator or a cloned authenticator has been used previously).
            7. Hash `AuthenticationResponse.FinalChallengeParams` using the hashing algorithm suitable for this authenticator type. Look up the hash algorithm in authenticator Metadata, field `AuthenticationAlgs`. It is the hash algorithm associated with the first entry related to a constant with prefix ALG_SIGN.
                - `FCHash = hash(AuthenticationResponse.FinalChallengeParams)`
            8. Make sure that `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_FINAL_CHALLENGE_HASH` == `FCHash`
                - If comparison fails – continue with next assertion
            9. If `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.TAG_ASSERTION_INFO.authenticationMode` == `2`

               NOTE

               The transaction hash included in this `AuthenticationResponse` must match the transaction content specified in the related `AuthenticationRequest`. As FIDO doesn't mandate any specific FIDO Server API, the transaction content

> could be cached by any relying party software component, e.g. the FIDO Server or the relying party Web Application.

1. Make sure there is a transaction cached on Relying Party side.
   - If not – continue with next assertion
2. Go over all cached forms of the transaction content (potentially multiple cached PNGs for the same transaction) and calculate their hashes using hashing algorithm suitable for this authenticator (same hash algorithm as used for FinalChallenge).
   - For each `cachedTransaction` add `hash(cachedTransaction)` into `cachedTransactionHashList`
3. Make sure that `a.TransactionHash` is in `cachedTransactionHashList`
   - If it's not in the list – continue with next assertion

10. Use `UAuth.pub` key and appropriate authentication algorithm to verify `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_SIGNATURE`
    1. If signature verification fails – continue with next assertion
    2. Update `SignCounter` in user's record with `a.assertion.TAG_UAFV1_AUTH_ASSERTION.TAG_UAFV1_SIGNED_DATA.SignCounter`

2. if `a.assertion.TAG_UAFV1_AUTH_ASSERTION` contains a different object than `TAG_UAFV1_SIGNED_DATA` as first element, then follow the rules specific to that object.

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

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

3.6 Deregistration Operation

This operation allows FIDO Server to ask the FIDO Authenticator to delete keys related to the particular relying party.

The FIDO Server may explicitly enumerate the keys to be deleted, or the FIDO server may signal deregistration of all keys on all authenticators managed by the FIDO UAF Client and relating to a given appID.

> NOTE
> There are various deregistration use cases that both FIDO Server and FIDO Client implementations should allow for. Two in particular are:
>
> 1. FIDO Servers should trigger this operation in the event a user removes their account at the relying party.
> 2. FIDO Clients should ensure that relying party application facets -- e.g., mobile apps, web pages -- have means to initiate a deregistration operation without having necessarily received a UAF protocol message with an `op` value of "Dereg". This allows the relying party app facet to remove a user's keys from authenticators during events such as relying party app removal or installation.

3.6.1 Deregistration Request Message

The FIDO UAF Deregistration request message is represented as an array of dictionaries. The array must contain exactly one dictionary. The request is defined as DeregistrationRequest dictionary.

EXAMPLE 12: UAF Deregistration Request

```
[{
    "header": {
      "op": "Dereg",
      "upv": {
        "major": 1,
        "minor": 1
      },
  "appID": "https://uaf-test-1.noknoktest.com:8443/SampleApp/uaf/facets"
    },
    "authenticators": [
      {
        "aaid": "ABCD#ABCD",
        "keyID": ""
      }
    ]
}]
```

The example above contains a version 1.1 deregistration request. This request will deregister all keys registered in authenticator with `aaid` "ABCD#ABCD" for the given `appID`.

> NOTE
>
> There is no deregistration response object.

3.6.2 DeregisterAuthenticator dictionary

WebIDL

```
dictionary DeregisterAuthenticator {
    required AAID   aaid;
    required KeyID keyID;
};
```

3.6.2.1 Dictionary `DeregisterAuthenticator` Members

aaid of type required AAID
: AAID of the authenticator housing the `UAuth.priv` key to deregister, or an empty string if all keys related to the specified `appID` are to be de-registered.

keyID of type required KeyID
: The unique KeyID related to `UAuth.priv`. KeyID is assumed to be unique within the scope of an AAID only. If `aaid` is not an empty string, then:

    1. `keyID` may contain a value of type KeyID, or,
    2. `keyID` may be an empty string.

    (1) signals deletion of a particular `UAuth.priv` key mapped to the (`AAID`, `KeyID`) tuple.

    (2) signals deletion of all KeyIDs associated with the specified `aaid`.

    If `aaid` is an empty string, then `keyID` must also be an empty string. This signals deregistration of all keys on all authenticators that are mapped to the specified `appID`.

3.6.3 DeregistrationRequest dictionary

WebIDL

```
dictionary DeregistrationRequest {
    required OperationHeader           header;
    required DeregisterAuthenticator[] authenticators;
};
```

3.6.3.1 Dictionary `DeregistrationRequest` Members

header of type required OperationHeader
: `Header.op` must be "Dereg".

authenticators of type array of required DeregisterAuthenticator
: List of authenticators to be deregistered.

3.6.4 Deregistration Processing Rules

3.6.4.1 Deregistration Request Generation Rules for FIDO Server

The FIDO Server must follow the steps:

1. Create a **`DeregistrationRequest`** message `m` with `m.header.upv` set to the appropriate version number.
2. If the FIDO Server intends to deregister all keys on all authenticators managed by the FIDO UAF Client for this `appID`, then:
    1. create one and only one **`DeregisterAuthenticator`** object `o`
    2. Set `o.aaid` and `o.keyID` to be empty string values
    3. Append `o` to `m.authenticators`, and go to step 5
3. If the FIDO Server intends to deregister all keys on all authenticators with a given AAID managed by the FIDO UAF Client for this `appID`, then:
    1. create one and only one **`DeregisterAuthenticator`** object `o`
    2. Set `o.aaid` to the intended AAID and set `o.keyID` to be an empty string.
    3. Append `o` to `m.authenticators`, and go to step 5
4. Otherwise, if the FIDO Server intends to deregister specific (`AAID`, `KeyID`) tuples, then for each tuple to be deregistered:
    1. create a **`DeregisterAuthenticator`** object `o`
    2. Set `o.aaid` and `o.keyID` appropriately
    3. Append `o` to `m.authenticators`
5. delete related entry (or entries) in FIDO Server's account database
6. Send message to FIDO UAF Client

3.6.4.2 Deregistration Request Processing Rules for FIDO UAF Client

The FIDO UAF Client must follow the steps:

1. Choose the message `m` with upv set to the appropriate version number.
2. Parse the message
    - If a mandatory field in **`DeregistrationRequest`** message is not present or a field doesn't correspond to its type and value – reject the operation
    - Empty string values for `o.aaid` and `o.keyID` must occur in the first and only DeregisterAuthenticator object o, otherwise reject the operation
3. Obtain `FacetID` of the requesting Application. If the `AppID` is missing or empty, set the `AppID` to the `FacetID`.

    Verify that the `FacetID` is authorized for the `AppID` according to the algorithms in [FIDOAppIDAndFacets].

    - If the `FacetID` of the requesting Application is not authorized, reject the operation
4. For each authenticator compatible with the message version `DeregistrationRequest.header.upv` and having an AAID matching one of the provided `AAID`s (an AAID of an authenticator matches if it is either (a) equal to one of the `AAID`s in the `DeregistrationRequest` or if (b) the `AAID` in the `DeregistrationRequest` is an empty string):
    1. Create appropriate `ASMRequest` for Deregister function and send it to the ASM. If the ASM returns an error, handle that error appropriately. The status code returned by the ASM [UAFASM] must be mapped to a status code defined in [UAFAppAPIAndTransport] as specified in section 3.4.6.2.1 Mapping ASM Status Codes to ErrorCode.

3.6.4.3 Deregistration Request Processing Rules for FIDO Authenticator

See [UAFASM] section "Deregister request".

4. Considerations

This section is non-normative.

4.1 Protocol Core Design Considerations

This section describes the important design elements used in the protocol.

4.1.1 Authenticator Metadata

It is assumed that FIDO Server has access to a list of all supported authenticators and their corresponding Metadata. Authenticator metadata [FIDOMetadataStatement] contains information such as:

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

In order to make a decision about which authenticators are appropriate for a specific transaction, FIDO Server looks up the list of authenticator metadata by AAID and retrieves the required information from it.

NORMATIVE

Each entry in the authenticator metadata repository must be identified with a unique authenticator Attestation ID (AAID).

4.1.2 Authenticator Attestation

Authenticator Attestation is the process of validating authenticator model identity during registration. It allows Relying Parties to cryptographically verify that the authenticator reported by FIDO UAF Client is really what it claims to be.

Using authenticator Attestation, a relying party "example-rp.com" will be able to verify that the authenticator model of the "example-Authenticator", reported with AAID "1234#5678", is not malware running on the FIDO User Device but is really a authenticator of model "1234#5678".

NORMATIVE

FIDO Authenticators should support "Basic Attestation" or "ECDAA" described below. New Attestation mechanisms may be added to the protocol over time.

NORMATIVE

FIDO Authenticators not providing sufficient protection for Attestation keys (non-attested authenticators) must use the UAuth.priv key in order to formally generate the same KeyRegistrationData object as attested authenticators. This behavior must be properly declared in the Authenticator Metadata.

4.1.2.1 Basic Attestation

NORMATIVE

There are two different flavors of Basic Attestation:

Full Basic Attestation
: Based on an attestation private key shared among a class of authenticators (e.g. same model).

Surrogate Basic Attestation
: Just syntactically a Basic Attestation. The attestation object self-signed, i.e. it is signed using the UAuth.priv key, i.e. the key corresponding to the UAuth.pub key included in the attestation object. As a consequence it **does not** provide a cryptographic proof of the security characteristics. But it is the best thing we can do if the authenticator is not able to have an attestation private key.

4.1.2.1.1 Full Basic Attestation

NOTE

FIDO Servers must have access to a trust anchor for verifying attestation public keys (i.e. Attestation Certificate trust store) in order to follow the assumptions made in [FIDOSecRef]. Authenticators must provide its attestation signature during the registration process for the same reason. The attestation trust anchor is shared with FIDO Servers out of band (as part of the Metadata). This sharing process shouldt be done according to [FIDOMetadataService].

NOTE

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

NOTE

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

In this Full Basic Attestation model, a large number of authenticators must share the same Attestation certificate and Attestation Private Key in order to provide non-linkability (see Protocol Core Design Considerations). Authenticators can only be identified on a production batch level or an AAID level by their Attestation Certificate, and not individually. A large number of authenticators sharing the same Attestation Certificate provides better privacy, but also makes the related private key a more attractive attack target.

NOTE

When using Full Basic Attestation: A given set of authenticators sharing the same manufacturer and essential characteristics must not be issued a new Attestation Key before at least 100,000 devices are issued the previous shared key.

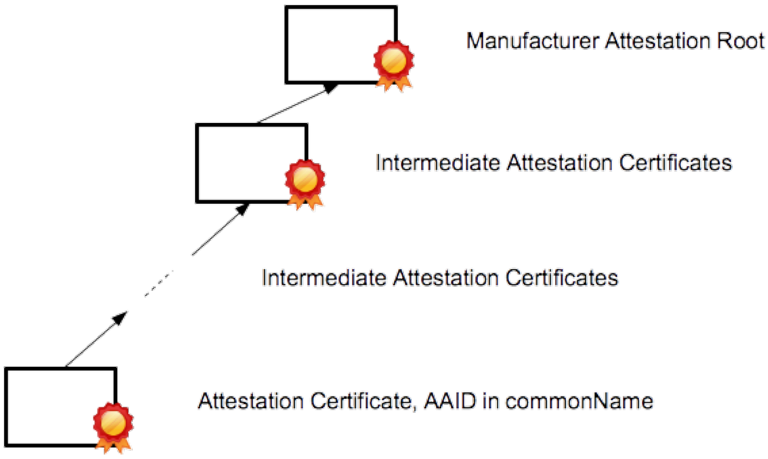


Fig. 10 Attestation Certificate Chain

4.1.2.1.2 Surrogate Basic Attestation

NORMATIVE

In this attestation method, the UAuth.priv key must be used to sign the Registration Data object. This behavior must be properly declared in the Authenticator Metadata.

NOTE

FIDO Authenticators not providing sufficient protection for Attestation keys (non-attested authenticators) must use this attestation method.

4.1.2.2 Direct Anonymous Attestation (ECDAA)

The FIDO Basic Attestation scheme uses attestation "group" keys shared across a set of authenticators with identical characteristics in order to preserve privacy by avoiding the introduction of global correlation handles. If such an attestation key is extracted from one single authenticator, it is possible to create a "fake" authenticator using the same key and hence indistinguishable from the original authenticators by the relying party. Removing trust for registering new authenticators with the related key would affect the entire set of authenticators sharing the same "group" key. Depending on the number of authenticators, this risk might be unacceptable high.

This is especially relevant when the attestation key is primarily protected against malware attacks as opposed to targeted physical attacks.

An alternative approach to "group" keys is the use of individual keys combined with a Privacy-CA [TPMv1-2-Part1]. Translated to FIDO, this approach would require one Privacy-CA interaction for each Uauth key. This means relatively high load and high availability requirements for the Privacy-CA. Additionally the Privacy-CA aggregates sensitive information (i.e. knowing the relying parties the user interacts with). This might make the Privacy-CA an interesting attack target.

Another alternative is the Direct Anonymous Attestation [BriCamChe2004-DAA]. Direct Anonymous Attestation is a cryptographic scheme combining privacy with security. It uses the Authenticator specific secret once to communicate with a single DAA Issuer (either at manufacturing time or after being sold before first use) and uses the resulting DAA credential in the DAA-Sign protocol with each relying party. The (original) DAA scheme has been adopted by the Trusted Computing Group for TPM v1.2 [TPMv1-2-Part1].

ECDAA (see [FIDOEcdaaAlgorithm] for details) is an improved DAA scheme based on elliptic curves and bilinear pairings [CheLi2013-ECDAA]. This scheme provides significantly improved performance compared with the original DAA and it is part of the TPMv2 specification [TPMv2-Part1].

NORMATIVE

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

4.1.3 Error Handling

NOTE

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

NORMATIVE

FIDO Authenticators must inform the FIDO UAF Client (see FIDO Interoperability Overview) about any error conditions encountered when processing commands through the Authenticator Specific Module (ASM). See [UAFASM] and [UAFAuthnrCommands] for details.

4.1.4 Assertion Schemes

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

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

- Cryptographic key provisioning (KeyRegistrationData)
- Cryptographic authentication and signature (SignedData)

The combination of KeyRegistrationData and SignedData schemes is called an Assertion Scheme.

The UAF protocol allows plugging in new Assertion Schemes. See also UAF Supported Assertion Schemes.

The Registration Assertion defines how and in which format a cryptographic key is exchanged between the authenticator and the FIDO Server.

The Authentication Assertion defines how and in which format the authenticator generates a cryptographic signature.

The generally-supported Assertion Schemes are defined in [UAFRegistry].

4.1.5 Username in Authenticator

FIDO UAF supports authenticators acting as first authentication factor (i.e. replacing username and password). As part of the FIDO UAF Registration, the Uauth key is registered (linked) to the related user account at the RP. The authenticator stores the username (allowing the user to select a specific account at the RP in the case he has multiple ones). See [UAFAuthnrCommands], section "Sign Command" for details.

4.1.6 Silent Authenticators

FIDO UAF supports authenticators not requiring any types of user verification or user presence check. Such authenticators are called **Silent Authenticators**.

In order to meet user's expectations, such Silent Authenticators need specific properties:

- It must be possible for a user to effectively remove a Uauth key maintained by a Silent Authenticator (in order to avoid being tracked) at the user's discretion (see [UAFAuthnrCommands]). This is not compatible with statelesss implementations storing the Uauth private key wrapped inside a KeyHandle on the FIDO Server.
- TransactionConfirmation is not supported (as it would require user input which is not intended), see [UAFAuthnrCommands].
- They might not operate in first factor mode (see [UAFAuthnrCommands]) as this might violate the privacy principles.

The MetadataStatement has to truthfully reflect the Silent Authenticator, i.e. field userVerification needs to be set to USER_VERIFY_NONE.

4.1.7 TLS Protected Communication

NOTE

In order to protect the data communication between FIDO UAF Client and FIDO Server a protected TLS channel must be used by FIDO UAF Client (or User Agent) and the Relying Party for all protocol elements.

1. The server endpoint of the TLS connection must be at the Relying Party
2. The client endpoint of the TLS connection must be either the FIDO UAF Client or the User Agent / App
3. TLS Client and Server should use TLS v1.2 or newer and should only use TLS v1.1 if TLS v1.2 or higher are not available. The "anon" and "null" TLS crypto suites are not allowed and must be rejected; insecure crypto-algorithms in TLS (e.g. MD5, RC4, SHA1) should be avoided [[SP 800-131A]] [RFC7525].
4. TLS Extended Master Secret Extension [RFC7627] and TLS Renegotiation Indication Extension [RFC5746] should be used to protect against MITM attacks.
5. The use of the tls-unique method is deprecated as its security is broken, see [TLSAUTH].

We recommend, that the

1. TLS Client verifies and validates the server certificate chain according to [RFC5280], section 6 "Certificate Path Validation". The certificate revocation status should be checked (e.g. using OCSP [RFC2560] or CRL based validation [RFC5280]) and the TLS server identity should be checked as well [RFC6125].
2. TLS Client's trusted certificate root store is properly maintained and at least requires the CAs included in the root store to annually pass Web Trust or ETSI (ETSI TS 101 456, or ETSI TS 102 042) audits for SSL CAs.

See [TR-03116-4] and [SHEFFER-TLS] for more recommendations on how to use TLS.

4.2 Implementation Considerations

4.2.1 Server Challenge and Random Numbers

NOTE

A `ServerChallenge` needs appropriate random sources in order to be effective (see [RFC4086] for more details). The (pseudo-)random numbers used for generating the Server Challenge should successfully pass the randomness test specified in [Coron99] and they should follow the guideline given in [SP800-90b].

4.3 Security Considerations

There is no "one size fits all" authentication method. The FIDO goal is to decouple the user verification method from the authentication protocol and the authentication server, and to support a broad range of user verification methods and a broad range of assurance levels. FIDO authenticators should be able to leverage capabilities of existing computing hardware, e.g. mobile devices or smart cards.

The overall assurance level of electronic user authentications highly depends (a) on the security and integrity of the user's equipment involved and (b) on the authentication method being used to authenticate the user.

When using FIDO, users should have the freedom to use any available equipment and a variety of authentication methods. The relying party needs reliable information about the security relevant parts of the equipment and the authentication method itself in order to determine whether the overall risk of an electronic authentication is acceptable in a particular business context. The FIDO Metadata Service [FIDOMetadataService] is intended to provide such information.

It is important for the UAF protocol to provide this kind of reliable information about the security relevant parts of the equipment and the authentication method itself to the FIDO server.

The overall security is determined by the weakest link. In order to support scalable security in FIDO, the underlying UAF protocol needs to provide a very high conceptual security level, so that the protocol isn't the weakest link.

Relying Parties define Acceptable Assurance Levels. The FIDO Alliance envisions a broad range of FIDO UAF Clients, FIDO Authenticators and FIDO Servers to be offered by various vendors. Relying parties should be able to select a FIDO Server providing the appropriate level of security. They should also be in a position to accept FIDO Authenticators meeting the security needs of the given business context, to compensate assurance level deficits by adding appropriate implicit authentication measures, and to reject authenticators not meeting their requirements. FIDO does not mandate a very high assurance level for FIDO Authenticators, instead it provides the basis for authenticator and user verification method competition.

Authentication vs. Transaction Confirmation. Existing Cloud services are typically based on authentication. The user launches an application (i.e. User Agent) assumed to be trusted and authenticates to the Cloud service in order to establish an authenticated communication channel between the application and the Cloud service. After this authentication, the application can perform any actions to the Cloud service using the authenticated channel. The service provider will attribute all those actions to the user. Essentially the user authenticates all actions performed by the application in advance until the service connection or authentication times out. This is a very convenient way as the user doesn't get distracted by manual actions required for the authentication. It is suitable for actions with low risk consequences.

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

UAF supports both methods; they are called "Authentication" and "Transaction Confirmation". The technical difference is, that with Authentication the user confirms a random challenge, where in the case of Transaction Confirmation the user also confirms a human readable content, i.e. the contract. From a security point, in the case of authentication the application needs to be trusted as it performs any action once the authenticated communication channel has been established. In the case of Transaction Confirmation only the transaction confirmation display component implementing WYSIWYS needs to be trusted, not the entire application.

Distinct Attestable Security Components. For the relying party in order to determine the risk associated with an authentication, it is important to know details about some components of the user's environment. Web Browsers typically send a "User Agent" string to the web server. Unfortunately any application could send any string as "User Agent" to the relying party. So this method doesn't provide strong security. FIDO UAF is based on a concept of cryptographic attestation. With this concept, the component to be attested owns a cryptographic secret and authenticates its identity with this cryptographic secret. In FIDO UAF the cryptographic secret is called "Authenticator Attestation Key". The relying party gets access to reference data required for verifying the attestation.

In order to enable the relying party to appropriately determine the risk associated with an authentication, all components performing significant security functions need to be attestable.

In FIDO UAF significant security functions are implemented in the "FIDO Authenticators". Security functions are:

1. Protecting the attestation key.
2. Generating and protecting the Authentication key(s), typically one per relying party and user account on relying party.
3. Verifying the user.
4. Providing the WYSIWYS capability ("Transaction Confirmation Display" component).

Some FIDO Authenticators might implement these functions in software running on the FIDO User Device, others might implement these functions in "hardware", i.e. software running on a hardware segregated from the FIDO User Device. Some FIDO Authenticators might even be formally evaluated and accredited to some national or international scheme. Each FIDO Authenticator model has an attestation ID (AAID), uniquely identifying the related security characteristics. Relying parties get access to these security properties of the FIDO Authenticators and the reference data required for verifying the attestation.

Resilience to leaks from other verifiers. One of the important issues with existing authentication solutions is a weak server side implementation, affecting the security of authentication of typical users to other relying parties. It is the goal of the FIDO UAF protocol to decouple the security of different relying parties.

Decoupling User Verification Method from Authentication Protocol. In order to decouple the user verification method from the authentication protocol, FIDO UAF is based on an extensible set of cryptographic authentication algorithms. The cryptographic secret will be unlocked after user verification by the Authenticator. This secret is then used for the authenticator-to-relying party authentication. The set of cryptographic algorithms is chosen according to the capabilities of existing cryptographic hardware and computing devices. It can be extended in order to support new cryptographic hardware.

Privacy Protection. Different regions in the world have different privacy regulations. The FIDO UAF protocol should be acceptable in all regions and hence must support the highest level of data protection. As a consequence, FIDO UAF doesn't require transmission of biometric data to the relying party nor does it require the storage of biometric reference data [ISOBiometrics] at the relying party. Additionally, cryptographic secrets used for different relying parties shall not allow the parties to link actions to the same user entity. UAF supports this concept, known as non-linkability. Consequently, the UAF protocol doesn't require a trusted third party to be involved in every transaction.

Relying parties can interactively discover the AAIDs of all enabled FIDO Authenticators on the FIDO User Device using the Discovery interface [UAFAppAPIAndTransport]. The combination of AAIDs adds to the entropy provided by the client to relying parties. Based on such information, relying parties can fingerprint clients on the internet (see Browser Uniqueness at eff.org and https://wiki.mozilla.org/Fingerprinting). In order to minimize the entropy added by FIDO, the user can enable/disable individual authenticators – even when they are embedded in the device (see [UAFAppAPIAndTransport], section "privacy considerations").

4.3.1 FIDO Authenticator Security

See [UAFAuthnrCommands].

4.3.2 Cryptographic Algorithms

In order to keep key sizes small and to make private key operations fast enough for small devices, it is suggested that implementers prefer ECDSA [ECDSA-ANSI] in combination with SHA-256 / SHA-512 hash algorithms. However, the RSA algorithm is also supported. See [FIDORegistry] "Authentication Algorithms" and "Public Key Representation Formats" for a list of generally supported cryptographic algorithms.

One characteristic of ECDSA is that it needs to produce, for each signature generation, a fresh random value. For effective security, this value must be chosen randomly and uniformly from a set of modular integers, using a cryptographically secure process. Even slight biases in that process may be turned into attacks on the signature schemes.

NOTE

If such random values cannot be provided under all possible environmental conditions, then a deterministic version of ECDSA should be used (see [RFC6979]).

4.3.3 FIDO Client Trust Model

The FIDO environment on a FIDO User Device comprises 4 entities:

- User Agents (a native app or a browser)
- FIDO UAF Clients (a shared service potentially used by multiple User Agents)
- Authenticator Specific Modules (ASMs)
- Authenticators

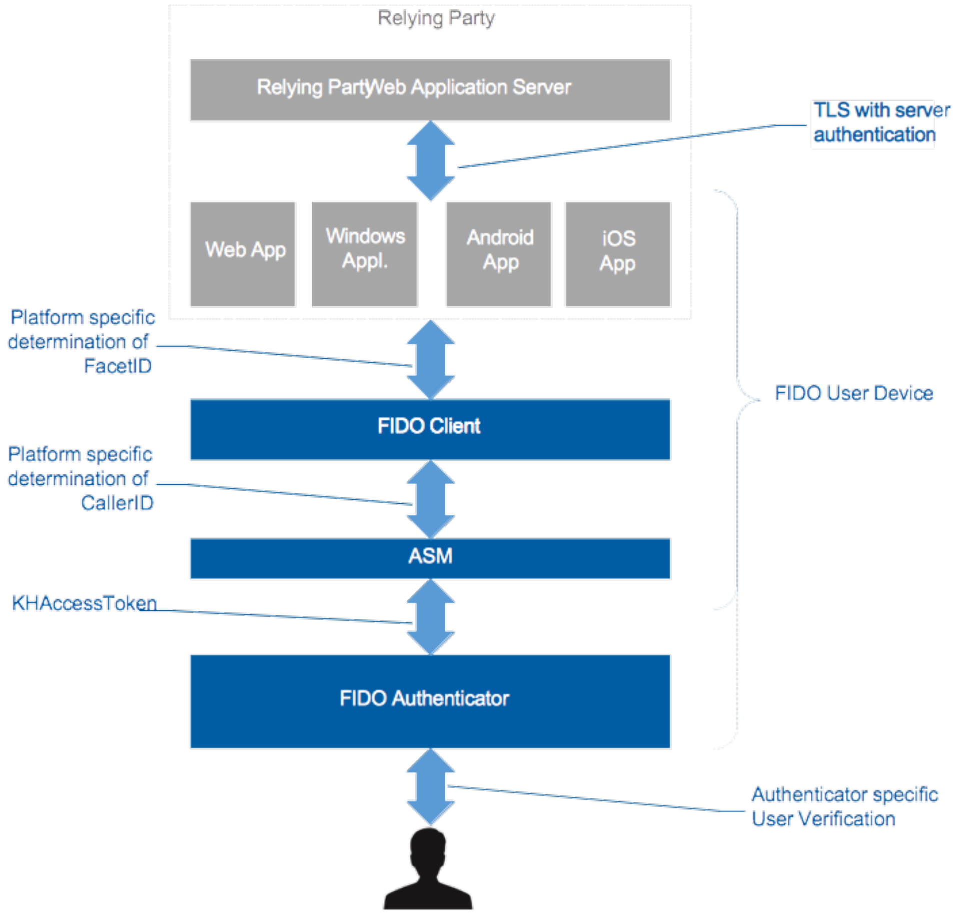


Fig. 11 UAF Client Trust Model

The security and privacy principles that underpin mobile operating systems require certain behaviours from apps. FIDO must uphold those principles wherever possible. This means that each of these components has to enforce specific trust relationships with the others to avoid the risk of rogue components subverting the integrity of the solution.

One specific requirement on handsets is that apps originating from different vendors must not be allowed directly to view or edit each other's data (e.g. FIDO UAF credentials).

Given that FIDO UAF Clients are intended to provide a shared service, the principle of siloed app data has been applied to the FIDO UAF Client, rather than individual apps. This means that if two or more FIDO UAF Clients are present on a device, then each FIDO UAF Client is unable to access authentication keys created by another FIDO UAF Client. A given FIDO UAF Client may however provide services to multiple User Agents, so that the same authentication key can authenticate to different facets of the same Relying Party, even if one facet is a 3rd party browser.

This exclusive access restriction is enforced through the KHAccessToken. When a FIDO UAF Client communicates with an ASM, the ASM reads the identity of the FIDO UAF Client caller1 and includes that Client ID in the KHAccessToken that it sends to the authenticator. Subsequent calls to the authenticator must include the same Client ID in the KHAccessToken. Each authentication key is also bound to the ASM that created it, by means of an ASMToken (a random unique ID for the ASM) that is also included in the KHAccessToken.

Finally, the User Agents that a FIDO UAF Client will recognise are determined by the Relying Party itself. The FIDO UAF Client requests a list of Trusted Apps from the RP as part of the Registration and Authentication protocols. This prevents User Agents that have not been explicitly authorized by the Relying Party from using the FIDO credentials.

In this manner, in a compliant FIDO installation, UAF credentials can only be accessed via apps that the relying party explicitly trusts and through the same client and ASM that performed the original registration.

It should be noted that the specification allows for FIDO UAF Clients to be built directly into User Agents. However, such implementations will restrict the ability to support multiple facets for relying party applications unless they also expose the UAF Client API for other User Agents to consume.

4.3.3.1 Isolation using KHAccessToken

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

The KHAccessToken allows restricting access to the keys generated by the FIDO Authenticator to the intended ASM. It is based on a Trust On First Use (TOFU) concept.

FIDO Authenticators are capable of binding UAuth.Key with a key provided by the caller (i.e. the ASM). This key is called KHAccessToken.

This technique allows making sure that registered keys are only accessible by the caller that originally registered them. A malicious App on a mobile platform won't be able to access keys by bypassing the related ASM (assuming that this ASM originally registered these keys).

The KHAccessToken is typically specific to the AppID, PersonaID, ASMToken and the CallerID. See [UAFASM] for more details.

> NOTE
>
> On some platforms, the ASM additionally might need special permissions in order to communicate with the FIDO Authenticator. Some platforms do not provide means to reliably enforce access control among applications.

4.3.4 TLS Binding

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

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

1. Attackers might fraudulently get a TLS server certificate for the same AppID as the relying party and they might be able to manipulate the DNS system.
2. Attackers might be able to steal the relying party's TLS server private key and certificate and they might be able to manipulate the DNS system.

And there are functionality requirements:

1. UAF transactions might span across multiple TLS sessions. As a consequence, "tls-unique" defined in [RFC5929] might be difficult to implement.
2. Data centers might use SSL concentrators.
3. Data centers might implement load-balancing for TLS endpoints using different TLS certificates. As a consequence, "tls-server-end-point" defined in [RFC5929], i.e. the hash of the TLS server certificate might be inappropriate.
4. Unfortunately, hashing of the TLS server certificate (as in "tls-server-end-point") also limits the usefulness of the channel binding in a particular, but quite common circumstance. If the client is operated behind a trusted (to that client) proxy that acts as a TLS man-in-the-middle, your client will see a different certificate than the one the server is using. This is actually quite common on corporate or military networks with a high security posture that want to inspect all incoming and outgoing traffic. If the FIDO Server just gets a hash value, there's no way to distinguish this from an attack. If sending the entire certificate is acceptable from a performance perspective, the server can examine it and determine if it is a certificate for a valid name from a non-standard issuer (likely administratively trusted) or a certificate for a different name (which almost certainly indicates a forwarding attack).

See ChannelBinding dictionary for more details.

4.3.5 Session Management

FIDO does not define any specific session management methods. However, several FIDO functions rely on a robust session management being implemented by the relying party's web application:

FIDO Registration
: A web application might trigger FIDO Registration after authenticating an existing user via legacy credentials. So the session is used to maintain the authentication state until the FIDO Registration is completed.

FIDO Authentication
: After success FIDO Authentication, the session is used to maintain the authentication state during the operations performed by the user agent or mobile app.

Best practices should be followed to implement robust session management (e.g. [OWASP2013]).

4.3.6 Personas

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

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

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

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

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

In the case of bound authenticators, this is different. FIDO recommends the "Persona" concept for this situation.

All relevant data in an authenticator are related to one Persona (e.g. "business" or "personal"). Some administrative interface (not standardized by FIDO) of the authenticator may allow maintaining and switching Personas.

> NORMATIVE
>
> The authenticator must only "know" / "recognize" data (e.g. authentication keys, usernames, KeyIDs, …) related to the Persona being active at that time.

With this concept, the User can switch to the "Personal" Persona and register new accounts. After switching back to "Business" Persona, these accounts will not be recognized by the authenticator (until the User switches back to "Personal" Persona again).

In order to support the persona feature, the FIDO Authenticator-specific Module API [UAFASM] supports the use of a 'PersonaID' to identify the persona in use by the authenticator. How Personas are managed or communicated with the user is out of scope for FIDO.

4.3.7 ServerData and KeyHandle

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

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

> NOTE
>
> In some situations, it is desirable to protect sensitive data such that it can be stored in arbitrary places (e.g. in serverData or in the KeyHandle). In such situations, the confidentiality and integrity of such sensitive data must be protected. This can be achieved by using a suitable encryption algorithm, e.g. AES with a suitable cipher mode, e.g. CBC or CTR [CTRMode]. This cipher mode needs to be used correctly. For CBC, for example, a fresh random IV for each encryption is required. The data might have to be padded first in order to obtain an integral number of blocks in length. The integrity protection can be achieved by adding a MAC or a digital signature on the ciphertext, using a different key than for the encryption, e.g. using HMAC [FIPS198-1]. Alternatively, an authenticated encryption scheme such as AES-GCM [SP800-38D] or AES-CCM [SP800-38C] could be used. Such a scheme provides both integrity and confidentiality in a single algorithm and using a single key.

> NOTE
>
> When protecting serverData, the MAC or digital signature computation should include some data that binds the data to its associated message, for example by re-including the challenge value in the authenticated serverData.

4.3.8 Authenticator Information retrieved through UAF Application API vs. Metadata

Several authenticator properties (e.g. UserVerificationMethods, KeyProtection, TransactionConfirmationDisplay, ...) are available in the metadata [FIDOMetadataStatement] and through the FIDO UAF Application API. The properties included in the metadata are authoritative and are provided by a trusted source. When in doubt, decisions should be based on the properties retrieved from the Metadata as opposed to the data retrieved through the FIDO UAF Application API.

However, the properties retrieved through the FIDO UAF Application API provide a good "hint" what to expect from the Authenticator. Such "hints" are well suited to drive and optimize the user experience.

4.3.9 Policy Verification

FIDO UAF Response messages do not include all parameters received in the related FIDO UAF request message into the to-be-signed object. As a consequence, any MITM could modify such entries.

FIDO Server will detect such changes if the modified value is unacceptable.

For example, a MITM could replace a generic policy by a policy specifying only the weakest possible FIDO Authenticator. Such a change will be detected by FIDO Server if the weakest possible FIDO Authenticator does not match the initial policy (see Registration Response Processing Rules and Authentication Response Processing Rules).

4.3.10 Replay Attack Protection

The FIDO UAF protocol specifies two different methods for replay-attack protection:

1. Secure transport protocol (TLS)
2. Server Challenge.

The TLS protocol by itself protects against replay-attacks when implemented correctly [TLS].

Additionally, each protocol message contains some random bytes in the `serverChallenge` field. The FIDO server should only accept incoming FIDO UAF messages which contain a valid `serverChallenge` value. This is done by verifying that the `serverChallenge` value, sent by the client, was previously generated by the FIDO server. See **`FinalChallengeParams`**.

It should also be noted that under some (albeit unlikely) circumstances, random numbers generated by the FIDO server may not be unique, and in such cases, the same `serverChallenge` may be presented more than once, making a replay attack harder to detect.

4.3.11 Protection against Cloned Authenticators

FIDO UAF relies on the UAuth.Key to be protected and managed by an authenticator with the security characteristics specified for the model (identified by the AAID). The security is better when only a single authenticator with that specific UAuth.Key instance exists. Consequently FIDO UAF specifies some protection measures against cloning of authenticators.

Firstly, if the UAuth private keys are protected by appropriate measures then cloning should be hard as such keys cannot be extracted easily.

Secondly, UAF specifies a Signature Counter (see Authentication Response Processing Rules and [UAFAuthnrCommands]). This counter is increased by every signature operation. If a cloned authenticator is used, then the subsequent use of the original authenticator would include a signature counter lower to or equal to the previous (malicious) operation. Such an incident can be detected by the FIDO Server.

4.3.12 Anti-Fraud Signals

There is the potential that some attacker misuses a FIDO Authenticator for committing fraud, more specifically they would:

1. Register the authenticator to some relying party for one account
2. Commit fraud
3. Deregister the Authenticator
4. Register the authenticator to some relying party for another account
5. Commit fraud
6. Deregister the Authenticator
7. and so on...

> NOTE
>
> Authenticators might support a Registration Counter (`RegCounter`). The `RegCounter` will be incremented on each registration and hence might become exceedingly high in such fraud scenarios. See [UAFAuthnrCommands] for more details.

4.4 Interoperability Considerations

FIDO supports Web Applications, Mobile Applications and Native PC Applications. Such applications are referred to as FIDO enabled applications.

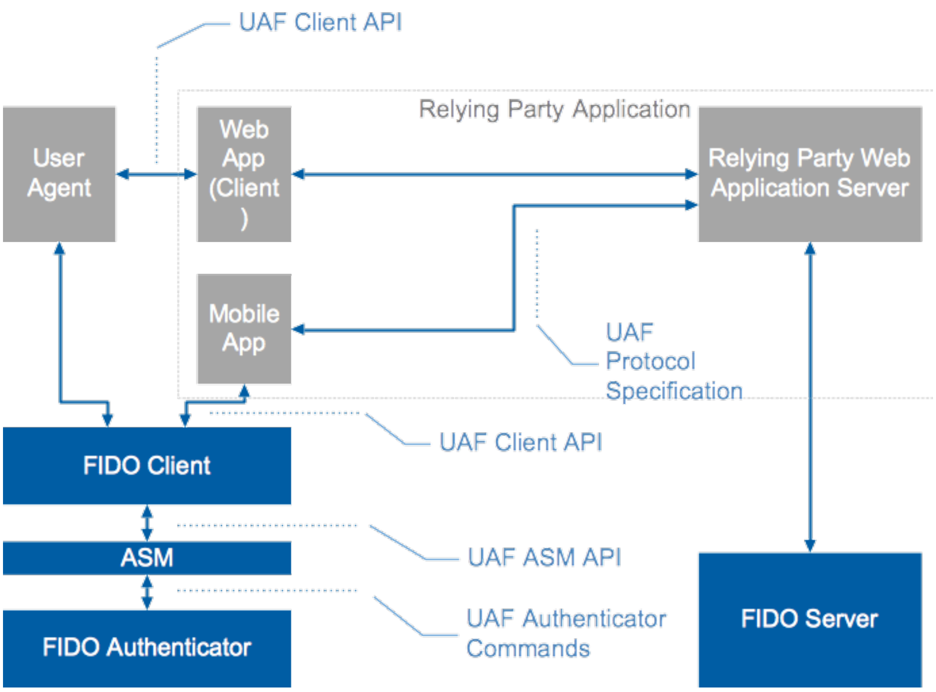


Fig. 12 FIDO Interoperability Overview

Web applications typically consist of the web application server and the related Web App. The Web App code (e.g. HTML and JavaScript) is rendered and executed on the client side by the User Agent. The Web App code talks to the User Agent via a set of JavaScript APIs, e.g. HTML DOM. The FIDO DOM API is defined in [UAFAppAPIAndTransport]. The protocol between the Web App and the Relying Party Web Application Server is typically proprietary.

Mobile Apps play the role of the User Agent and the Web App (Client). The protocol between the Mobile App and the Relying Party Web Application Server is typically proprietary.

Native PC Applications play the role of the User Agent, the Web App (Client). Those applications are typically expected to be independent from any particular Relying Party Web Application Server.

It is recommended for FIDO enabled applications to use the FIDO messages according to the format specified in this document.

It is recommended for FIDO enabled application to use the UAF HTTP Binding defined in [UAFAppAPIAndTransport].

> NOTE
>
> The KeyRegistrationData and SignedData objects [UAFAuthnrCommands] are generated and signed by the FIDO Authenticators and have to be verified by the FIDO Server. Verification will fail if the values are modified during transport.
>
> The ASM API [UAFASM] specifies the standardized API to access authenticator Specific Modules (ASMs) on Desktop PCs and Mobile Devices.
>
> The document [UAFAuthnrCommands] does not specify a particular protocol or API. Instead it lists the minimum data set and a specific message format which needs to be transferred to and from the FIDO Authenticator.

5. UAF Supported Assertion Schemes

This section is normative.

5.1 Assertion Scheme "UAFV1TLV"

This scheme is mandatory to implement for FIDO Servers. This scheme is mandatory to implement for FIDO Authenticators.

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

Authenticator.

This assertion scheme is using Tag Length Value (TLV) compact encoding to encode registration and authentication assertions generated by authenticators. This is the default assertion scheme for UAF protocol.

TAGs and Algorithms are defined in [UAFRegistry].

The authenticator must use a dedicated key pair (UAuth.pub/UAuth.priv) suitable for the authentication algorithm specified in the metadata statement [FIDOMetadataStatement] for each relying party. This key pair should be generated as part of the registration operation.

Conforming FIDO Servers must implement all authentication algorithms and key formats listed in document [FIDORegistry] unless they are explicitly marked as optional in [FIDORegistry].

Conforming FIDO Servers must implement all attestation types (`TAG_ATTESTATION_*`) listed in document [UAFRegistry] unless they are explicitly marked as optional in [UAFRegistry].

Conforming authenticators must implement (at least) one attestation type defined in [UAFRegistry], as well as one authentication algorithm and one key format listed in [FIDORegistry].

5.1.1 KeyRegistrationData

See [UAFAuthnrCommands], section "TAG_UAFV1_KRD".

5.1.2 SignedData

See [UAFAuthnrCommands], section "TAG_UAFV1_SIGNED_DATA".

6. Definitions

See [FIDOGlossary].

7. Table of Figures

A. References

A.1 Normative references

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

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

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

[FIDOAppIDAndFacets]
D. Balfanz, B. Hill, R. Lindemann, D. Baghdasaryan, *FIDO AppID and Facets v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-appid-and-facets-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-appid-and-facets-v1.1-id-20170202.pdf

[FIDOEcdaaAlgorithm]
R. Lindemann, J. Camenisch, M. Drijvers, A. Edgington, A. Lehmann, R. Urian, *FIDO ECDAA Algorithm*. FIDO Alliance Implementation Draft. URLs:
HTML: fido-ecdaa-v1.1-id-20170202.html
PDF: fido-ecdaa-v1.1-id-20170202.pdf.

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[FIDOMetadataStatement]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO Metadata Statements v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.pdf

[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.pdf

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

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

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

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

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119
[RFC3629]
F. Yergeau. *UTF-8, a transformation format of ISO 10646* November 2003. Internet Standard. URL:https://tools.ietf.org/html/rfc3629
[RFC4086]
D. Eastlake 3rd, J. Schiller, S. Crocker *Randomness Requirements for Security (RFC 4086)*, IETF, June 2005, URL: http://www.ietf.org/rfc/rfc4086.txt
[RFC4627]
D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. July 2006. Informational. URL: https://tools.ietf.org/html/rfc4627
[RFC4648]
S. Josefsson, *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*, IETF, October 2006, URL: http://www.ietf.org/rfc/rfc4648.txt
[RFC5056]
N. Williams, *On the Use of Channel Bindings to Secure Channels (RFC 5056)* IETF, November 2007, URL: http://www.ietf.org/rfc/rfc5056.txt
[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL:http://www.ietf.org/rfc/rfc5280.txt
[RFC5929]
J. Altman, N. Williams, L. Zhu,*Channel Bindings for TLS (RFC 5929)*, IETF, July 2010, URL:http://www.ietf.org/rfc/rfc5929.txt
[RFC6234]
D. Eastlake 3rd, T. Hansen, *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF) (RFC 6234)* IETF, May 2011, URL: http://www.ietf.org/rfc/rfc6234.txt
[RFC6979]
T. Pornin, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) (RFC6979)*, IETF, August 2013, URL:http://www.ietf.org/rfc/rfc6979.txt
[SP800-90b]
Elaine Barker and John Kelsey, *NIST Special Publication 800-90b: Recommendation for the Entropy Sources Used for Random Bit Generation*. National Institute of Standards and Technology, April 2016, URL:http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf
[UAFASM]
D. Baghdasaryan, J. Kemp, R. Lindemann, B. Hill, R. Sasson,*FIDO UAF Authenticator-Specific Module API*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.pdf
[UAFAppAPIAndTransport]
B. Hill, D. Baghdasaryan, B. Blanke, *FIDO UAF Application API and Transport Binding Specification* FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.pdf
[UAFAuthnrCommands]
D. Baghdasaryan, J. Kemp, R. Lindemann, R. Sasson, B. Hill,*FIDO UAF Authenticator Commands v1.0* FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.pdf
[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values* FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.pdf
[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[AnonTerminology]
"Anonymity, Unlinkability, Unobservability, Pseudonymity, and Identity Management - A Consolidated Proposal for Terminology", Version 0.34,. A. Pfitzmann and M. Hansen, August 2010. URL:http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf
[BriCamChe2004-DAA]
Ernie Brickell, Intel Corporation; Jan Camenisch, IBM Research; Liqun Chen, HP Laboratories,*Direct Anonymous Attestation*, 2004, URL: http://eprint.iacr.org/2004/205.pdf
[CTRMode]
H. Lipmea, P. Rogaway, D. Wagner,*Comments to NIST concerning AES Modes of Operation: CTR-Mode Encryption* National Institute of Standards and Technology, accessed March 11, 2014, URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf
[CheLi2013-ECDAA]
Liqun Chen, HP Laboratories and Jiangtao Li, Intel Corporation,*Flexible and Scalable Digital Signatures in TPM 2.0* 2013. URL: http://dx.doi.org/10.1145/2508859.2516729
[ECDSA-ANSI]
Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), ANSI X9.62-2005 American National Standards Institute, November 2005, URL: http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005
[FIDOMetadataService]
R. Lindemann, B. Hill, D. Baghdasaryan, *FIDO Metadata Service v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-service-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-service-v1.1-id-20170202.pdf
[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.pdf
[FIPS198-1]
FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC) National Institute of Standards and Technology, July 2008, URL: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf
[ISOBiometrics]
Project Editor, *Harmonized Biometric Vocabulary*. ISO/IEC 2382-37. 15 December 2012, URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194_ISOIEC_2382-37_2012.zip
[OWASP2013]
OWASP 2013. OWASP Top 10 - 2013. The Ten Most Critical Web Application Security Risks
[RFC2560]
M. Myers; R. Ankney; A. Malpani; S. Galperin; C. Adams.*X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. June 1999. Proposed Standard. URL:https://tools.ietf.org/html/rfc2560
[RFC5746]
E. Rescorla; M. Ray; S. Dispensa; N. Oskov.*Transport Layer Security (TLS) Renegotiation Indication Extension* February 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5746
[RFC6125]
P. Saint-Andre, J. Hodges, *Representation and Verification of Domain-Based Application Service Identity within Internet Public Key*

Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS) (RFC 6125), IETF, March 2011, URL: http://www.ietf.org/rfc/rfc6125.txt

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

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

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

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

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

[SP800-38D]
M. Dworkin. *NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. November 2007 URL: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf

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

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

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

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

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

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

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Working Draft. URL: https://www.w3.org/TR/WebIDL-1/

FIDO UAF Application API and Transport Binding Specification

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-client-api-transport-v1.0-ps-20141208.html
Editor:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Brad Hill, PayPal, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.
Bill Blanke, Nok Nok Labs, Inc.
Jeff Hodges, PayPal, Inc.
Ka Yang, Nok Nok Labs, Inc.

Abstract

Describes APIs and an interoperability profile for client applications to utilize FIDO UAF. This includes methods of communicating with a FIDO UAF Client for both Web platform and Android applications, transport requirements, and an HTTPS interoperability profile for sending FIDO UAF messages to a compatible server.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

The notation base64url refers to "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

DOM APIs are described using the ECMAScript [ECMA-262] bindings for WebIDL [WebIDL-ED].

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as `required`.

WebIDL dictionary members must not have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

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

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

> NOTE
>
> Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

The FIDO UAF technology replaces traditional username and password-based authentication solutions for online services, with a stronger and simpler alternative. The core UAF protocol consists of four conceptual conversations between a FIDO UAF Client and FIDO Server: Registration, Authentication, Transaction Confirmation, and Deregistration. As specified in the core protocol, these messages do not have a defined network transport, or describe how application software that a user interfaces with can use UAF. This document describes the API surface that a client application can use to communicate with FIDO UAF Client software, and transport patterns and security requirements for delivering UAF Protocol messages to a remote server.

The reader should also be familiar with the FIDO Glossary of Terms [FIDOGlossary] and the UAF Protocol specification [UAFProtocol].

2.1 Audience

This document is of interest to client-side application authors that wish to utilize FIDO UAF, as well as implementers of web browsers, browser plugins and FIDO clients, in that it describes the API surface they need to expose to application authors.

2.2 Scope

This document describes:

- The local ECMAScript [ECMA-262] API exposed by a FIDO UAF-enabled web browser to client-side web applications.
- The mechanisms and APIs for Android [ANDROID] applications to discover and utilize a shared FIDO UAF Client service.
- The general security requirements for applications initiating and transporting UAF protocol exchanges.
- An interoperability profile for transporting FIDO UAF messages over HTTPS [RFC2818].

The following are out of scope for this document:

- The format and details of the underlying UAF Protocol messages
- APIs for, and any details of interactions between FIDO Server software and the server-side application stack.

> NOTE
>
> The goal of describing standard APIs and an interoperability profile for the transport of FIDO UAF messages here is to provide an example of how to develop a FIDO-enabled application and to promote the ease of integrating interoperable layers from different vendors to build a complete FIDO UAF solution. For any given application instance, these particular patterns may not be ideal and are not mandatory. Applications may use alternate transports, bundle UAF Protocol messages with other network data, or discover and utilize alternative APIs as they see fit.

2.3 Architecture

The overall architecture of the UAF protocol and its various operations is described in the FIDO UAF Protocol Specification [UAFProtocol]. The following simplified architecture diagram illustrates the interactions and actors this document is concerned with:

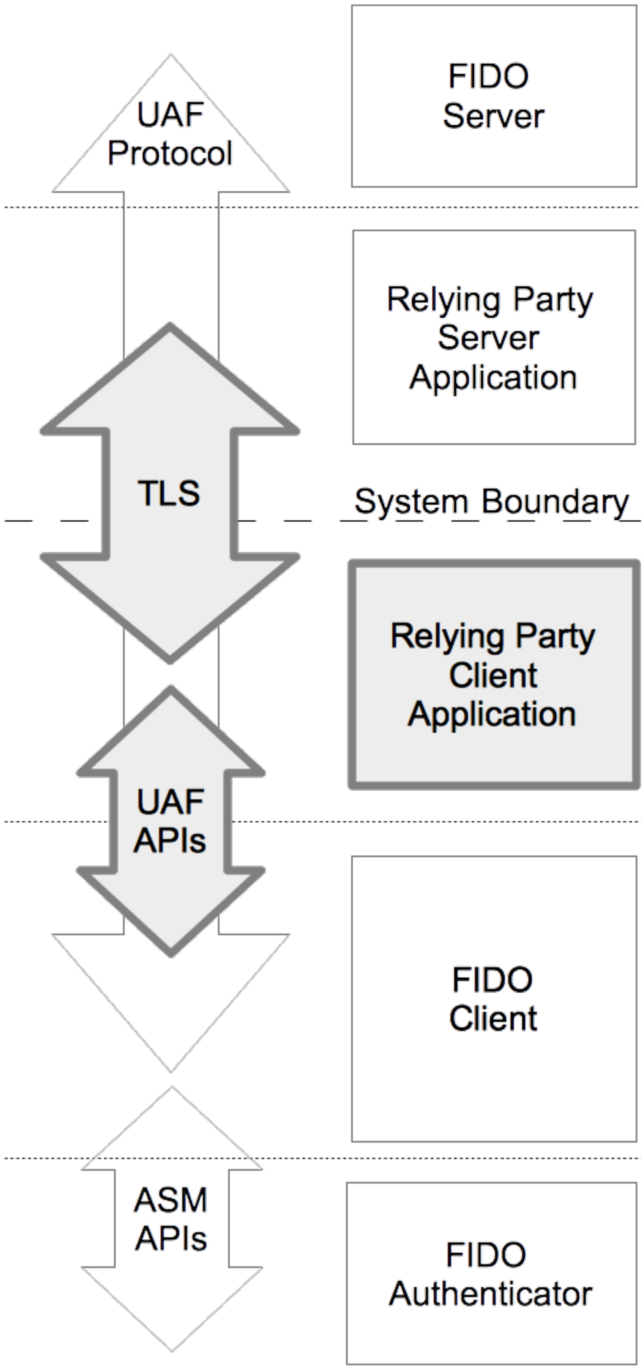


Fig. 1 UAF Application API Architecture and Transport Layers

This document describes the shaded components in Fig 1.

2.3.1 Protocol Conversation

The core UAF protocol consists of five conceptual phases:

- **Discovery** allows the relying party server to determine the availability of FIDO capabilities at the client, including metadata about the available authenticators.
- **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** allows a user to provide an account identifier, proof-of-possession of previously registered key material associated with that identifier, and potentially other attested data, to the relying party server.
- **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** 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.

Discovery does not involve a protocol exchange with the FIDO Server. However, the information available through the discovery APIs might be communicated back to the server in an application-specific manner, such as by obtaining a UAF protocol request message containing an authenticator policy tailored to the specific capabilities of the FIDO user device.

Although the UAF protocol abstractly defines the FIDO server as the initiator of requests, UAF client applications working as described in this document will always transport UAF protocol messages over a client-initiated request/response protocol such as HTTP.

The protocol flow from the point of view of the relying party client application for registration, authentication, and transaction confirmation is as follows:

1. The client application either explicitly contacts the server to obtain a UAF Protocol Request Message, or this message is delivered along with other client application content.

2. The client application invokes the appropriate API to pass the UAF protocol request message asynchronously to the FIDO UAF Client, and receives a set of callbacks.
3. The FIDO UAF Client performs any necessary interactions with the user and authenticator(s) to complete the request and uses a callback to either notify the client application of an error, or to return a UAF response message.
4. The client application delivers the UAF response message to the server over a transport protocol such as HTTP.
5. The server optionally returns an indication of the results of the operation and additional data such as authorization tokens or a redirect.
6. The client application optionally uses the appropriate API to inform the FIDO UAF Client of the results of the operation. This allows the FIDO UAF Client to perform "housekeeping" tasks for a better user experience, e.g. by not attempting to use again later a key that the server refused to register.
7. The client application optionally processes additional data returned to it in an application-specific manner, e.g. processing new authorization tokens, redirecting the user to a new resource or interpreting an error code to determine if and how it should retry a failed operation.

Deregister does not involve a UAF protocol round-trip. If the relying party server instructs the client application to perform a deregistration, the client application simply delivers the UAF protocol Request message to the FIDO UAF Client using the appropriate API. The FIDO UAF Client does not return the results of a deregister operation to the relying party client application or FIDO Server.

UAF protocol Messages are JSON [ECMA-404] structures, but client applications are discouraged from modifying them. These messages may contain embedded cryptographic integrity protections and any modifications might invalidate the messages from the point of view of the FIDO UAF Client or Server.

3. Common Definitions

This section is normative.

These elements are shared by several APIs and layers.

3.1 UAF Status Codes

This table lists UAF protocol status codes.

> NOTE
> These codes indicate the result of the UAF operation at the FIDO Server. They do not represent the HTTP [RFC7230] layer or other transport layers. These codes are intended for consumption by both the client-side web app and FIDO UAF Client to inform application-specific error reporting, retry and housekeeping behavior.

Code	Meaning
1200	OK. Operation completed
1202	Accepted. 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 1202 response
1400	Bad Request. The server did not understand the message
1401	Unauthorized. The userid must be authenticated to perform this operation, or this KeyID is not associated with this UserID.
1403	Forbidden. The userid is not allowed to perform this operation. Client should not retry
1404	Not Found.
1408	Request Timeout.
1480	Unknown AAID. The server was unable to locate authoritative metadata for the AAID.
1481	Unknown KeyID. The server was unable to locate a registration for the given UserID and KeyID combination. This error indicates that there is an invalid registration on the user's device. It is recommended that FIDO UAF Client deletes the key from local device when this error is received.
1490	Channel Binding Refused. The server refused to service the request due to a missing or mismatched channel binding(s).
1491	Request Invalid. The server refused to service the request because the request message nonce was unknown, expired or the server has previously serviced a message with the same nonce and user ID.
1492	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.
1493	Revoked Authenticator. The authenticator is considered revoked by the server.
1494	Unacceptable Key. The key used is unacceptable. Perhaps it is on a list of known weak keys or uses insecure parameter choices.
1495	Unacceptable Algorithm. The server believes the authenticator to be capable of using a stronger mutually-agreeable algorithm than was presented in the request.
1496	Unacceptable Attestation. The attestation(s) provided were not accepted by the server.
1497	Unacceptable Client Capabilities. The server was unable or unwilling to use required capabilities provided supplementally to the authenticator by the client software.
1498	Unacceptable Content. There was a problem with the contents of the message and the server was unwilling or unable to process it.
1500	Internal Server Error

4. Shared Definitions

This section is normative.

> NOTE
>
> This section defines a number of JSON structures, specified with WebIDL [WebIDL-ED]. These structures are shared among APIs for multiple target platforms.

4.1 UAFMessage Dictionary

The UAFMessage dictionary is a wrapper object that contains the raw UAF protocol Message and additional JSON data that may be used to carry application-specific data for use by either the client application or FIDO UAF Client.

WebIDL

```
dictionary UAFMessage {
    required DOMString uafProtocolMessage;
    Object             additionalData;
};
```

4.1.1 Dictionary `UAFMessage` Members

`uafProtocolMessage` of type required DOMString
: This key contains the UAF protocol Message that will be processed by the FIDO UAF Client or Server. Modification by the client application may invalidate the message. A client application may examine the contents of a message, for example, to determine if a message is still fresh. Details of the structure of the message can be found in the UAF protocol Specification [UAFProtocol].

`additionalData` of type Object
: This key allows the FIDO Server or client application to attach additional data for use by the FIDO UAF Client as a JSON object, or the FIDO UAF Client or client application to attach additional data for use by the client application.

4.2 Version interface

Describes a version of the UAF protocol or FIDO UAF Client for compatibility checking.

WebIDL

```
interface Version {
    readonly    attribute unsigned short major;
    readonly    attribute unsigned short minor;
};
```

4.2.1 Attributes

`major` of type unsigned short, readonly
: Major version number.

`minor` of type unsigned short, readonly
: Minor version number.

4.3 Authenticator interface

Used by several phases of UAF, the `Authenticator` interface exposes a subset of both verified metadata [FIDOMetadataStatement] and transient information about the state of an available authenticator.

WebIDL

```
interface Authenticator {
    readonly    attribute DOMString                             title;
    readonly    attribute AAID                                  aaid;
    readonly    attribute DOMString                             description;
    readonly    attribute Version[]                             supportedUAFVersions;
    readonly    attribute DOMString                             assertionScheme;
    readonly    attribute unsigned short                        authenticationAlgorithm;
    readonly    attribute unsigned short[]                      attestationTypes;
    readonly    attribute unsigned long                         userVerification;
    readonly    attribute unsigned short                        keyProtection;
    readonly    attribute unsigned short                        matcherProtection;
    readonly    attribute unsigned long                         attachmentHint;
    readonly    attribute boolean                               isSecondFactorOnly;
    readonly    attribute unsigned short                        tcDisplay;
    readonly    attribute DOMString                             tcDisplayContentType;
    readonly    attribute DisplayPNGCharacteristicsDescriptor[] tcDisplayPNGCharacteristics;
    readonly    attribute DOMString                             icon;
    readonly    attribute DOMString[]                           supportedExtensionIDs;
};
```

4.3.1 Attributes

`title` of type DOMString, readonly
: A short, user-friendly name for the authenticator.

> NOTE
>
> This text must be localized for current locale.
>
> If the ASM doesn't return a title in the `AuthenticatorInfo` object [UAFASM], the FIDO UAF Client must generate a title based on the other fields in `AuthenticatorInfo`, because `title` must not be empty (see section 1. Notation).

`aaid` of type AAID, readonly
: The *Authenticator Attestation ID*, which identifies the type and batch of the authenticator. See [UAFProtocol] for the definition of the AAID structure.

`description` of type DOMString, readonly
: A user-friendly description string for the authenticator.

> NOTE
>
> This text must be localized for current locale.
>
> It is intended to be displayed to the user. It might deviate from the description specified in the authenticator's metadata statement [FIDOMetadataStatement].
>
> If the ASM doesn't return a description in the `AuthenticatorInfo` object [UAFASM], the FIDO UAF Client must generate a meaningful description to the calling App based on the other fields in `AuthenticatorInfo`, because `description` must not be empty (see section 1. Notation).

supportedUAFVersions of type array of *Version*, readonly
: Indicates the UAF protocol Versions supported by the authenticator.

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

  Assertion scheme identifiers are defined in the UAF Registry of Predefined Values. [UAFRegistry]

authenticationAlgorithm of type unsigned short, readonly
: Supported Authentication Algorithm. The value must be related to constants with prefix `ALG_SIGN`.

attestationTypes of type array of unsigned short, readonly
: A list of supported attestation types. The values are defined in [UAFRegistry] by the constants with the prefix `TAG_ATTESTATION`.

userVerification of type unsigned long, readonly
: A set of bit flags indicating the user verification methods supported by the authenticator. The values are defined by the constants with the prefix `USER_VERIFY`.

keyProtection of type unsigned short, readonly
: A set of bit flags indicating the key protection used by the authenticator. The values are defined by the constants with the prefix `KEY_PROTECTION`.

matcherProtection of type unsigned short, readonly
: A set of bit flags indicating the matcher protection used by the authenticator. The values are defined by the constants with the prefix `MATCHER_PROTECTION`.

attachmentHint of type unsigned long, readonly
: A set of bit flags indicating how the authenticator is *currently* connected to the FIDO User Device. The values are defined by the constants with the prefix `ATTACHMENT_HINT`.

  > NOTE
  >
  > Because the connection state and topology of an authenticator may be transient, these values are only hints that can be used in applying server-supplied policy to guide the user experience. This can be used to, for example, 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 upon by the relying party, although some models of authenticator may provide attested measurements with similar semantics as part of UAF protocol messages.

isSecondFactorOnly of type boolean, readonly
: Indicates whether the authenticator can only be used as a second-factor.

tcDisplay of type unsigned short, readonly
: A set of bit flags indicating the availability and type of transaction confirmation display. The values are defined by the constants with the prefix `TRANSACTION_CONFIRMATION_DISPLAY`.

  This value must be 0 if transaction confirmation is not supported by the authenticator.

tcDisplayContentType of type DOMString, readonly
: The MIME content-type [RFC2045] supported by the transaction confirmation display, such as `text/plain` or `image/png`.

  This value must be non-empty if transaction confirmation is supported (`tcDisplay` is non-zero).

tcDisplayPNGCharacteristics of type array of DisplayPNGCharacteristicsDescriptor, readonly
: The set of PNG characteristics *currently* supported by the transaction confirmation display (if any).

  > NOTE
  >
  > See [FIDOMetadataStatement] for additional information on the format of this field and the definition of the `DisplayPNGCharacteristicsDescriptor` structure.

  This list must be non-empty if PNG-image based transaction confirmation is supported, i.e. `tcDisplay` is non-zero and `tcDisplayContentType` is `image/png`.

icon of type DOMString, readonly
: A PNG [PNG] icon for the authenticator, encoded as a `data:` url [RFC2397].

  > NOTE
  >
  > If the ASM doesn't return an icon in the `AuthenticatorInfo` object [UAFASM], the FIDO UAF Client must set a default icon, because `icon` must not be empty (see section 1. Notation).

supportedExtensionIDs of type array of DOMString, readonly
: A list of supported UAF protocol extension identifiers. These may be vendor-specific.

4.3.2 Authenticator Interface Constants

A number of constants are defined for use with the bit flag fields `userVerification`, `keyProtection`, `attachmentHint`, and `tcDisplay`. To avoid duplication and inconsistencies, these are defined in the FIDO Registry of Predefined Values [FIDORegistry].

4.4 DiscoveryData dictionary

WebIDL

```
dictionary DiscoveryData {
    required Version[]       supportedUAFVersions;
    required DOMString       clientVendor;
    required Version         clientVersion;
    required Authenticator[] availableAuthenticators;
};
```

4.4.1 Dictionary `DiscoveryData` Members

`supportedUAFVersions` of type array of required Version
: A list of the FIDO UAF protocol versions supported by the client, most-preferred first.

`clientVendor` of type required DOMString
: The vendor of the FIDO UAF Client.

`clientVersion` of type required Version
: The version of the FIDO UAF Client. This is a vendor-specific version for the client software, not a UAF version.

`availableAuthenticators` of type array of required Authenticator
: An array containing Authenticator dictionaries describing the available UAF authenticators. The order is not significant. The list may be empty.

4.5 ErrorCode interface

WebIDL

```
interface ErrorCode {
    const short NO_ERROR = 0x0;
    const short WAIT_USER_ACTION = 0x01;
    const short INSECURE_TRANSPORT = 0x02;
    const short USER_CANCELLED = 0x03;
    const short UNSUPPORTED_VERSION = 0x04;
    const short NO_SUITABLE_AUTHENTICATOR = 0x05;
    const short PROTOCOL_ERROR = 0x06;
    const short UNTRUSTED_FACET_ID = 0x07;
    const short KEY_DISAPPEARED_PERMANENTLY = 0x09;
    const short AUTHENTICATOR_ACCESS_DENIED = 0x0c;
    const short INVALID_TRANSACTION_CONTENT = 0x0d;
    const short USER_NOT_RESPONSIVE = 0x0e;
    const short INSUFFICIENT_AUTHENTICATOR_RESOURCES = 0x0f;
    const short USER_LOCKOUT = 0x10;
    const short USER_NOT_ENROLLED = 0x11;
    const short UNKNOWN = 0xFF;
};
```

4.5.1 Constants

`NO_ERROR` of type short
: The operation completed with no error condition encountered. Upon receipt of this code, an application should no longer expect an associated `UAFResponseCallback` to fire.

`WAIT_USER_ACTION` of type short
: Waiting on user action to proceed. For example, selecting an authenticator in the FIDO client user interface, performing user verification, or completing an enrollment step with an authenticator.

`INSECURE_TRANSPORT` of type short
: `window.location.protocol` is not "https" or the DOM contains insecure mixed content.

`USER_CANCELLED` of type short
: The user declined any necessary part of the interaction to complete the registration.

`UNSUPPORTED_VERSION` of type short
: The `UAFMessage` does not specify a protocol version supported by this FIDO UAF Client.

`NO_SUITABLE_AUTHENTICATOR` of type short
: No authenticator matching the authenticator policy specified in the `UAFMessage` is available to service the request, or the user declined to consent to the use of a suitable authenticator.

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

`UNTRUSTED_FACET_ID` of type short
: The client declined to process the operation because the caller's calculated facet identifier was not found in the trusted list for the application identifier specified in the request message.

`KEY_DISAPPEARED_PERMANENTLY` of type short
: The UAuth key disappeared from the authenticator and canot be restored.

> NOTE
>
> The RP App might want to re-register the authenticator in this case.

`AUTHENTICATOR_ACCESS_DENIED` of type short
: The authenticator denied access to the resulting request.

> NOTE
>
> TODO: when does that occur and what should RP app do?

`INVALID_TRANSACTION_CONTENT` of type short
: Transaction content cannot be rendered, e.g. format doesn't fit authenticator's need.

> NOTE
>
> The transaction content format requirements are specified in the authenticator's metadata statement.

`USER_NOT_RESPONSIVE` of type short
: The user took too long to follow an instruction, e.g. didn't swipe the finger within the accepted time.

`INSUFFICIENT_AUTHENTICATOR_RESOURCES` of type short

Insufficient resources in the authenticator to perform the requested task.

USER_LOCKOUT of type short
The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that. For example, an authenticator could allow the user to enter an alternative password to re-enable the use of fingerprints after too many failed finger verification attempts. This error will be reported if such method either doesn't exist or the ASM / authenticator cannot automatically trigger it.

USER_NOT_ENROLLED of type short
The operation failed because the user is not enrolled to the authenticator and the authenticator cannot automatically trigger user enrollment.

UNKNOWN of type short
An error condition not described by the above-listed codes.

5. DOM API

This section is normative.

This section describes the API details exposed by a web browser or browser plugin to a client-side web application executing in a `Document` [DOM] context.

5.1 Feature Detection

FIDO's UAF DOM APIs are rooted in a new `fido` object, a property of `window.navigator` code; the existence and properties of which may be used for feature detection.

Example 1: Feature Detection of UAF APIs

EXAMPLE 1

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

5.2 uaf Interface

The `window.navigator.fido.uaf` interface is the primary means of interacting with the FIDO UAF Client. All operations are asynchronous.

WebIDL

```
interface uaf {
    void discover (DiscoveryCallback completionCallback, ErrorCallback errorCallback);
    void checkPolicy (UAFMessage message, ErrorCallback cb);
    void processUAFOperation (UAFMessage message, UAFResponseCallback completionCallback, ErrorCallback errorCallback);
    void notifyUAFResult (int responseCode, UAFMessage uafResponse);
};
```

5.2.1 Methods

discover

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.

Parameter	Type	Nullable	Optional	Description
completionCallback	DiscoveryCallback	✘	✘	The callback that receives DiscoveryData from the FIDO UAF Client.
errorCallback	ErrorCallback	✘	✘	A callback function to receive error and progress events.

Return type: `void`

checkPolicy

Ask the browser or browser plugin if it would be able to process the supplied request message without prompting the user.

Unlike other operations using an `ErrorCallback`, this operation must always trigger the callback and return `NO_ERROR` if it believes that the message can be processed and a suitable authenticator matching the embedded policy is available, or the appropriate `ErrorCode` value otherwise.

NOTE

Because this call should not prompt the user, it should not incur a potentially disrupting context-switch even if the FIDO UAF Client is implemented out-of-process.

Parameter	Type	Nullable	Optional	Description
message	UAFMessage	✘	✘	A UAFMessage containing the policy and operation to be tested.
cb	ErrorCallback	✘	✘	The callback function which receives the status of the operation.

Return type: `void`

processUAFOperation
Invokes the FIDO UAF Client, transferring control to prompt the user as necessary to complete the operation, and returns to the callback a message in one of the supported protocol versions indicated by the UAFMessage.

Parameter	Type	Nullable	Optional	Description
message	UAFMessage	✘	✘	The UAFMessage to be used by the FIDO client software.
completionCallback	UAFResponseCallback	✘	✘	The callback that receives the client response UAFMessage from the FIDO UAF Client, to be delivered to the relying party server.
errorCallback	ErrorCallback	✘	✘	A callback function to receive error and progress events from the FIDO UAF Client.

Return type: `void`

notifyUAFResult

Used to indicate the status code resulting from a FIDO UAF message delivered to the remote server. Applications must make this call when they receive a UAF status code from a server. This allows the FIDO UAF Client to perform housekeeping for a better user experience, for example not attempting to use keys that a server refused to register.

> NOTE
>
> If, and how, a status code is delivered by the server, is application and transport specific. A non-normative example can be found below in the HTTPS Transport Interoperability Profile.

Parameter	Type	Nullable	Optional	Description
responseCode	`int`	✘	✘	The `uafResult` field of a `ServerResponse`.
uafResponse	`UAFMessage`	✘	✘	The `UAFMessage` to which this `responseCode` applies.

Return type: `void`

5.3 UAFResponseCallback

A `UAFResponseCallback` is used upon successful completion of an asynchronous operation by the FIDO UAF Client to return the protocol response message to the client application for transport to the server.

> NOTE
>
> This callback is also called in the case of deregistration completion, even though the response object is empty then.

WebIDL

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

5.3.1 Callback `UAFResponseCallback` Parameters

`uafResponse` of type `UAFMessage`
The message and any additional data representing the FIDO UAF Client's response to the server's request message.

5.4 DiscoveryCallback

A `DiscoveryCallback` is used upon successful completion of an asynchronous discover operation by the FIDO UAF Client to return the `DiscoveryData` to the client application.

WebIDL

```
callback DiscoveryCallback = void (DiscoveryData data);
```

5.4.1 Callback `DiscoveryCallback` Parameters

`data` of type `DiscoveryData`
Describes the current state of FIDO UAF client software and authenticators available to the application.

5.5 ErrorCallback

An ErrorCallback is used to return progress and error codes from asynchronous operations performed by the FIDO UAF Client.

WebIDL

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

5.5.1 Callback `ErrorCallback` Parameters

`code` of type `ErrorCode`
A value from the `ErrorCode` interface indicating the result of the operation.

For certain operations, an ErrorCallback may be called multiple times, for example with the `WAIT_USER_ACTION` code.

5.6 Privacy Considerations for the DOM API

This section is non-normative.

Differences in the FIDO capabilities on a user device may (among many other characteristics) allow a server to "fingerprint" a remote client and attempt to persistently identify it, even 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 Discovery 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.

5.7 Security Considerations for the DOM API

This section is non-normative.

5.7.1 Insecure Mixed Content

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

The UAF APIs must immediately trigger the`ErrorCallback` with the `INSECURE_TRANSPORT` code and cease any further processing if any APIs defined in this document are invoked by a Document context that was not loaded over a secure transport and/or which contains insecure mixed content.

5.7.2 The Same Origin Policy, HTTP Redirects and Cross-Origin Content

When retrieving or transporting UAF protocol messages over HTTP, it is important to maintain consistency among the web origin of the document context and the origin embedded in the UAF protocol message. Mismatches may cause the protocol to fail or enable attacks against the protocol. Therefore:

FIDO UAF messages should not be transported using methods that opt-out of the Same Origin Policy [SOP], for example, using `<script src="url">` to non-same-origin URLs or by setting the `Access-Control-Allow-Origin` header at the server.

When transporting FIDO UAF messages using XMLHttpRequest [XHR] the client should not follow redirects that are to URLs with a different origin than the requesting document.

FIDO UAF messages should not be exposed in HTTP responses where the entire response body parses as valid ECMAScript. Resources exposed in this manner may be subject to unauthorized interactions by hostile applications hosted at untrusted origins through cross-origin embedding using `<script src="url">`.

Web applications should not share FIDO UAF messages across origins through channels such as `postMessage()` [webmessaging].

5.8 Implementation Notes for Browser/Plugin Authors

This section is non-normative.

Web applications utilizing UAF depend on services from the web browser as a trusted platform. The APIs for web applications do not provide a means to assert an origin as an application identity for the purposes of FIDO operations as this will be provided to the FIDO UAF Client by the browser based on its privileged understanding of the actual origin context.

The browser must enforce that the web origin communicated to the FIDO UAF Client as the application identity is accurate

The browser must also enforce that resource instances containing insecure mixed-content cannot utilize the UAF DOM APIs.

6. Android Intent API

This section is normative.

This section describes how an Android [ANDROID] client application can locate and communicate with a conforming FIDO Client installation operating on the host device.

> NOTE
>
> As with web applications, a variety of integration patterns are possible on the Android platform. The API described here allows an app to communicate with a shared FIDO UAF Client on the user device in a loosely-coupled fashion using Android *Intents*.

6.1 Android-specific Definitions

6.1.1 org.fidoalliance.uaf.permissions.FIDO_CLIENT

FIDO UAF Clients running on Android versions prior to Android 5must declare the `org.fidoalliance.uaf.permissions.FIDO_CLIENT` permission and they also must declare the related "uses-permission". See the below example of this permission expressed in an Android app manifest file `<permission/>` and `<uses-permission/>` element [AndroidAppManifest].

FIDO UAF Clients running on Android version 5 or latershould not declare this permission and they also should not declare the related "uses-permission".

EXAMPLE 2

```
<permission
    android:name="org.fidoalliance.uaf.permissions.FIDO_CLIENT"
    android:label="Act as a FIDO Client."
    android:description="This application acts as a FIDO Client. It may
       access authentication devices available on the system, create and
       delete FIDO registrations on behalf of other applications."
    android:protectionLevel="dangerous"
/>
<uses-permission android:name="org.fidoalliance.uaf.permissions.FIDO_CLIENT"/>
```

> NOTE
>
> - Since FIDO Clients perform security relevant tasks (e.g. verifying the AppID/FacetID relation and asking for user consent), users should carefully select the FIDO Clients they use. Requiring apps acting as FIDO Clients to declare and use this permission allows them to be identified as such to users.
> - There are not any FIDO Client resources needing "protection" based upon the FIDO_CLIENT permission. The reason for having FIDO Client declare the FIDO_CLIENT permission is solely that users should be able to carefully decide which FIDO Clients to install.
> - Android version 5 changed the way it handles the case where multiple apps declare the same permission [Android5Changes]; it blocks the installation of all subsequent apps declaring that permission.
> - *The best way to flag the fact that an app may act as a FIDO Client needs to be determined for Android version 5.*

6.1.2 org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER

Android applications requesting services from the FIDO UAF Client can do so under their own identity, or they can act as the user's agent by explicitly declaring an RFC6454 [RFC6454] serialization of the remote server's origin when invoking the FIDO UAF Client.

An application that is operating on behalf of a single entity must not set an explicit origin. Omitting an explicit origin will cause the FIDO UAF Client to determine the caller's identity as `android:apk-key-hash:<hash-of-public-key>`. The FIDO UAF Client will then compare this with the list of authorized application facets for the target AppID and proceed if it is listed as trusted.

> NOTE
>
> See the UAF Protocol Specification [UAFProtocol] for more information on application and facet identifiers.

If the application is explicitly intended to operate as the user's agent in the context of an arbitrary number of remote applications (as when implementing a full web browser) it may set its origin to the RFC6454 [RFC6454] Unicode serialization of the remote application's Origin. The application must satisfy the necessary conditions described in Transport Security Requirements for authenticating the remote server before setting the origin.

Use of the origin parameter requires the application to declare the `org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER` permission, and the FIDO UAF Client must verify that the calling application has this permission before processing the operation.

EXAMPLE 3

```
<permission
   android:name="org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER"
   android:label="Act as a browser for FIDO registrations."
   android:description="This application may act as a web browser,
       creating new and accessing existing FIDO registrations for any domain."
   android:protectionLevel="dangerous"
/>
```

6.1.3 channelBindings

This section is non-normative.

In the DOM API, the browser or browser plugin is responsible for supplying any available channel binding information to the FIDO Client, but an Android application, as the direct owner of the transport channel, must provide this information itself.

The `channelBindings` data structure is:

`Map<String,String>`

with the keys as defined for the`ChannelBinding` structure in the UAF Protocol Specification. [UAFProtocol]

The use of channel bindings for TLS helps assure the server that the channel over which UAF protocol messages are transported is the same channel the legitimate client is using and that messages have not been forwarded through a malicious party.

UAF defines support for the `tls-unique` and `tls-server-end-point` bindings from [RFC5929], as well as server certificate and ChannelID [ChannelID] bindings. The client should supply all channel binding information available to it.

Missing or invalid channel binding information may cause a relying party server to reject a transaction.

6.1.4 UAFIntentType enumeration

This enumeration describes the type of operation for the intent implementing the Android API.

NOTE

UAF uses only a single intent to simplify behavior in the situation even where multiple FIDO clients may be installed. In such a case, the user will be prompted which of the installed FIDO UAF clients should be used to handle an implicit intent.

If the user selected to make different FIDO UAF Clients the default for different intents representing different phases, it could produce inconsistent results or fail to function at all.

If the application workflow requries multiple calls to the client (and it usually does) the application should read the `componentName` from the intent extras it receives from `startActivityForResult()` and pass it to `setComponent()` for subsequent intents to be sure they are explicitly resolved to the same FIDO UAF Client.

WebIDL

```
enum UAFIntentType {
    "DISCOVER",
    "DISCOVER_RESULT",
    "CHECK_POLICY",
    "CHECK_POLICY_RESULT",
    "UAF_OPERATION",
    "UAF_OPERATION_RESULT",
    "UAF_OPERATION_COMPLETION_STATUS"
};
```

Enumeration description	
`DISCOVER`	Discovery
`DISCOVER_RESULT`	Discovery results
`CHECK_POLICY`	Perform a no-op check if a message could be processed.
`CHECK_POLICY_RESULT`	Check Policy results.
`UAF_OPERATION`	Process a Registration, Authentication, Transaction Confirmation or Deregistration message.
`UAF_OPERATION_RESULT`	UAF Operation results.
`UAF_OPERATION_COMPLETION_STATUS`	Inform the FIDO UAF Client of the completion status of a Registration, Authentication, Transaction Confirmation or Deregistration message.

6.2 org.fidoalliance.intent.FIDO_OPERATION Intent

All interactions between a FIDO UAF Client and an application on Android takes place via a single Android intent:

`org.fidoalliance.intent.FIDO_OPERATION`

The specifics of the operation are carried by the MIME media type and various extra data included with the intent.

The operations described in this document are of MIME media type `application/fido.uaf_client+json` and this must be set as the `type` attribute of any intent.

NOTE

Client applications can discover if a FIDO UAF Client (or several) is available on the system by using `PackageManager.queryIntentActivities(Intent intent, int flags)` with this intent to see if any activities are available.

Extra	Type	Description
UAFIntentType	String	One of the **UAFIntentType** enumeration values describing the intent.
discoveryData	String	**DiscoveryData** JSON dictionary.
componentName	String	The component name of the responding FIDO UAF Client. It must be serialized using ComponentName.flattenString()
errorCode	short	**ErrorCode** value for operation
message	String	**UAFMessage** request to test or process, depending on **UAFIntentType**.
origin	String	An RFC6454 Web Origin [RFC6454] string for the request, if the caller has the org.fidoalliance.permissions.ACT_AS_WEB_BROWSER permission.
channelBindings	String	The JSON dictionary of channel bindings for the operation.
responseCode	short	The uafResult field of a ServerResponse.

The following table shows what intent extras are expected, depending on the value of the UAFIntentType extra:

UAFIntentType value	discoveryData	componentName	errorCode	message	origin	channelBindings	responseCode
"DISCOVER"							
"DISCOVER_RESULT"	optional	required	required				
"CHECK_POLICY"				required	optional		
"CHECK_POLICY_RESULT"		required	required				
"UAF_OPERATION"				required	optional	required	
"UAF_OPERATION_RESULT"		required	required	optional			
"UAF_OPERATION_COMPLETION_STATUS"				required			required

6.2.1 UAFIntentType.DISCOVER

This Android intent invokes the FIDO UAF Client to discover the available authenticators and capabilties. The FIDO UAF Client generally will not show a UI associated with the handling of this intent, but immediately return the JSON structure. The calling application cannot depend on this however, as the FIDO UAF Client may show a UI for privacy purposes, allowing the user to choose whether and which authenticators to disclose to the calling application.

This intent must be invoked with startActivityForResult().

6.2.2 UAFIntentType.DISCOVER_RESULT

An intent with this type is returned by the FIDO UAF Client as an argument toonActivityResult() in response to receiving an intent of type DISCOVER.

If the resultCode passed to onActivityResult() is RESULT_OK, and the intent extra errorCode is NO_ERROR, this intent has an extra, discoveryData, containing a String representation of a **DiscoveryData** JSON dictionary with the available authenticators and capabilities.

6.2.3 UAFIntentType.CHECK_POLICY

This intent invokes the FIDO UAF Client to discover if it would be able to process the supplied message without prompting the user. The action handling this intent should not show a UI to the user.

This intent requires the following extras:

- message, containing a String representation of a **UAFMessage** representing the request message to test.
- origin, an optional extra that allows a caller with the org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER permission to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

This intent must be invoked with startActivityForResult().

6.2.4 UAFIntentType.CHECK_POLICY_RESULT

This Android intent is returned by the FIDO UAF Client as an argument toonActivityResult() in response to receiving a CHECK_POLICY intent.

In addition to the resultCode passed to onActivityResult(), this intent has an extra, errorCode, containing an **ErrorCode** value indicating the specific error condition or NO_ERROR if the FIDO UAF Client could process the message.

6.2.5 UAFIntentType.UAF_OPERATION

This Android intent invokes the FIDO UAF Client to process the supplied request message and return a response message ready for delivery to the FIDO UAF Server.

The sender should assume that the FIDO UAF Client will display a user interface allowing the user to handle this intent, for example, prompting the user to complete their verification ceremony.

This intent requires the following extras:

- message, containing a String representation of a **UAFMessage** representing the request message to process.
- channelBindings, containing a String representation of a JSON dictionary as defined by the ChannelBinding structure in the FIDO UAF Protocol Specification [UAFProtocol].
- origin, an optional parameter that allows a caller with the org.fidoalliance.uaf.permissions.ACT_AS_WEB_BROWSER permission to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

This intent must be invoked with startActivityForResult().

6.2.6 UAFIntentType.UAF_OPERATION_RESULT

This intent is returned by the FIDO UAF Client as an argument to `onActivityResult()`, in response to receiving a `UAF_OPERATION` intent.

If the `resultCode` passed to `onActivityResult()` is `RESULT_CANCELLED`, this intent will have an extra, `errorCode` parameter, containing an **`ErrorCode`** value indicating the specific error condition.

If the `resultCode` passed to `onActivityResult()` is `RESULT_OK`, and the `errorCode` is NO_ERROR, this intent has a `message`, containing a `String` representation of a **`UAFMessage`**, being the UAF protocol response message to be delivered to the FIDO Server.

6.2.7 UAFIntentType.UAF_OPERATION_COMPLETION_STATUS

This intent must be delivered to the FIDO UAF Client to indicate the processing status of a FIDO UAF message delivered to the remote server. This is especially important as a new registration may be considered by the client to be in a pending state until it is communicated that the server accepted it.

6.3 Alternate Android AIDL Service UAF Client Implementation

The Android Intent API can also be implemented using Android AIDL services as an alternative transport mechanism to Android Intents. While Android Intents work at the UI layer, Android AIDL services are performed at a lower level. This can ease integration with relying party apps, since UAF requests can be fulfilled without interfering with existing relying party app UI and application lifecycle behavior.

The UAF Android AIDL service needs to be defined in the UAF client manifest. This is done using the <service> tag for an Android AIDL service instead of the <activity> tag in Android Intents. Just as with Android intents, the manifest definition for the AIDL service uses an intent filter (note `org.fidoalliance.aidl.FIDO_OPERATION` versus `org.fidoalliance.intent.FIDO_OPERATION`) to identify itself as a FIDO UAF client to the relying party app:

EXAMPLE 4

```
<service android:name="foo" >
<intent-filter>
<action android:name="org.fidoalliance.aidl.FIDO_OPERATION" />
<category android:name="android.intent.category.DEFAULT" />
<data android:mimeType="application/fido.uaf_client+json" />
</intent-filter>
</service>
```

Once the relying party app choses a UAF client from the list discovered by `PackageManager.queryIntentServices()`, the relying party app and the FIDO UAF client share the following AIDL interface to service UAF requests:

EXAMPLE 5

```
package org.fidoalliance.aidl

oneway interface IUAFOperation
{
 void process(in Intent uafRequest, in IUAFResponseListener uafResponseListener);
}
```

Note that Android AIDL services use `Binder.getCallingUid()` instead of `Activity.getCallingActivity()` with Android Intents to identify the caller and obtain FacetID information.

For consistency, the Intents for the Android AIDL service are the same as defined in the Android Intent specification in the UAF standard. In `process()`, the `uafRequest` parameter is the Intent that would be passed to `startActivityForResult()`. The `uafResponseListener` parameter is a listener interface that receives the result. The following AIDL defines this interface:

EXAMPLE 6

```
package org.fidoalliance.aidl

interface IUAFResponseListener
{
 void onResult(in Intent uafResponse);
}
```

In the listener, the `uafResponse` parameter is the Intent that would be passed to `onActivityResult`.

6.4 Security Considerations for Android Implementations

This section is non-normative.

Android applications may choose to implement the user-interactive portion of FIDO in at least two ways:

- by authoring an Android Activity using Android-native user interface components, or
- with an HTML-based experience by loading an Android WebView and injecting the UAF DOM APIs with `addJavaScriptInterface()`.

An application that chooses to inject the UAF interface into a WebView must follow all appropriate security considerations that apply to usage of the DOM APIs, *and* those that apply to user agent implementers.

In particular, the content of a WebView into which an API will be injected must be loaded only from trusted local content or over a secure channel as specified in Transport Security Requirements and must not contain insecure mixed-content.

Applications should not declare the `ACT_AS_WEB_BROWSER` permission unless they need to act as the user's agent for an un-predetermined number of third party applications. Where an Android application has an explicit relationship with a relying party application(s), the preferred method of access control is for those applications to list the Android application's identity as a trusted facet. See the UAF Protocol Specification [UAFProtocol] for more information on application and facet identifiers.

To protect against a malicious application registering itself as a FIDO UAF Client, relying party applications can obtain the identity of the responding application, and utilize it in risk management decisions around the authentication or transaction events.

For example, a relying party might maintain a list of application identities known to belong to malware and refuse to accept operations completed with such clients, or a list of application identities of known-good clients that receive preferred risk-scoring.

Relying party applications running on Android versions prior to Android 5 must make sure that a FIDO UAF Client has the "uses-permission" for `org.fidoalliance.uaf.permissions.FIDO_CLIENT`. Relying party applications running on Android 5 should not implement this check.

NOTE

Relying party applications should implement the check on Android prior to 5 by using the package manager to verify that the FIDO Client indeed declared the `org.fidoalliance.uaf.permissions.FIDO_CLIENT` permission (see example below). Relying party applications should not use a "uses-permission" for `FIDO_CLIENT`.

EXAMPLE 7

```
boolean checkFIDOClientPermission(String packageName)
                                  throws NameNotFoundException {
  for (String requestedPermission :
         getPackageManager().getPackageInfo(packageName,
             PackageManager.GET_PERMISSIONS).requestedPermissions) {
      if (requestedPermission.matches(
           "org.fidoalliance.uaf.permissions.FIDO_CLIENT"))
        return true;
  }
 return false;
}
```

7. iOS Custom URL API

This section is normative.

This section describes how an iOS relying party application can locate and communicate with a conforming FIDO UAF Client installed on the host device.

NOTE

Because of sandboxing and no true multitasking support, the iOS operating system offers very limited ways to do interprocess communication (IPC).

Any IPC solution for a FIDO UAF Client must be able to:

1. Identify the calling app in order to provide FacetID approval.
2. Allow transition to another app without user intervention

Currently the only IPC method on iOS that satisfies both of these requirements is custom URL handlers.

Custom URL handlers use the iOS operating system to handle URL requests from the sender, launch the receiving app, and then pass the request to the receiving app for processing. By enabling custom URL handlers for two different applications, it is possible to achieve bidirectional IPC between them--one custom URL handler to send data from app A to app B and another custom URL handler to send data from app B to app A.

Because iOS has no true multitasking, there must be an app transition to process each request and response. Too many app transitions can negatively affect the user experience, so relying party applications must carefully choose when it is necessary to query the FIDO UAF Client.

7.1 iOS-specific Definitions

7.1.1 X-Callback-URL Transport

When the relying party application communicates with the FIDO UAF Client, it sends a URL with the standard `x-callback-url` format (see x-callback-url.com):

EXAMPLE 8

```
FidoUAFClient1://x-callback-url/[UAFxRequestType]?x-success=[RelyingPartyURL]://x-callback-url/
        [UAFxResponseType]&
        key=[SecretKey]&
        state=[STATE]&
        json=[Base64EncodedJSON]
```

- `FidoUAFClient1` is the iOS custom URL scheme used by FIDO UAF Clients. As specified in the `x-callback-url` standard, version information for the transport layer is encoded in the URL scheme itself (in this case, `FidoUAFClient1`). This is so other applications can check for support for the 1.0 version by using the `canOpenURL` call.
- `[UAFxRequestType]` is the type that should be used for request operations, which are described later in this document.
- `[RelyingPartyURL]` is the URL that the relying party app has registered in order to receive the response. According to the `x-callback-url` standard, this is defined using the `x-success` parameter.
- `[UAFxResponseType]` is the type that should be used for response operations, which are described later in this document.
- `[SecretKey]` is a base64url-encoded, without padding, random key generated for each request by the calling application.

  The response from the FIDO UAF Client will be encrypted with this key in order to prevent rogue applications from obtaining information by spoofing the return URL.

- `[STATE]` is data that can be used to match the request with the response.
- Finally `[Base64EncodedJSON]` contains the message to be sent to the FIDO UAF Client.

  Items are stored in JSON format and then base64url-encoded without padding.

For FIDO UAF Clients, the custom URL scheme handler entrypoint is the openURL() function:

EXAMPLE 9

```
(BOOL)application:(UIApplication *)application openURL:(NSURL *)url sourceApplication:(NSString *)sourceApplication annotation:(id)ann
```

Here, the URL above is received via the `url` parameter. For security considerations, the `sourceApplication` parameter contains the iOS bundle ID of the relying party application. This bundle ID must be used to verify the application `FacetID`.

Conversely, when the FIDO UAF Client responds to the request, it sends the following URL back in standard `x-callback-url` format:

EXAMPLE 10

```
[RelyingPartyURL]://x-callback-url/
      [UAFxResponseType]?
```

```
state=[STATE]&
json=[Base64EncodedJWE]
```

The parameters in the response are similar to those of the request, except that the `[Base64EncodedEncryptedJSON]` parameter is encrypted with the public key before being base64url-encoded without padding. `[STATE]` is the same `STATE` as was sent in the request--it is echoed back to the sender to verify the matched response.

In the relying party application's `openURL()` handler, the `url` parameter will be the URL listed above and the `sourceApplication` parameter will be the iOS bundle ID for the FIDO client application.

7.1.2 Secret Key Generation

A new secret encryption key must be generated by the calling application every time it sends a request to FIDO UAF Client. The FIDO UAF Client must then use this key to encrypt the response message before responding to the caller.

JSON Web Encryption (JWE), JSON Serialization (JWE Section 7.2) format must be used to represent the encrypted response message.

The encryption algorithm is that specified in "A128CBC-HS256" where the JWE "Key Management Mode" employed is "Direct Encryption" and the JWE "Content Encryption Key (CEK)" is the secret key generated by the calling application and passed to the FIDO UAF Client in the `key` parameter of the request.

EXAMPLE 11

```
{
   "unprotected": {"alg": "dir", "enc": "A128CBC-HS256"},
   "iv": "...",
   "ciphertext": "...",
   "tag": "..."
}
```

7.1.3 Origin

iOS applications requesting services from the FIDO Client can do so under their own identity, or they can act as the user's agent by explicitly declaring an RFC6454 [RFC6454] serialization of the remote server's origin when invoking the FIDO UAF Client.

An application that is operating on behalf of a single entity must not set an explicit origin. Omitting an explicit origin will cause the FIDO UAF Client to determine the caller's identity as `"ios:bundle-id"`. The FIDO UAF Client will then compare this with the list of authorized application facets for the target AppID and proceed if it is listed as trusted.

See the UAF Protocol Specification [UAFProtocol] for more information on application and facet identifiers.

If the application is explicitly intended to operate as the user's agent in the context of an arbitrary number of remote applications (as when implementing a full web browser) it may set origin to the RFC6454 [RFC6454] Unicode serialization of the remote application's Origin. The application must satisfy the necessary conditions described in Transport Security Requirements for authenticating the remote server before setting origin.

7.1.4 channelBindings

This section is non-normative.

In the DOM API, the browser or browser plugin is responsible for supplying any available channel binding information to the FIDO Client, but an iOS application, as the direct owner of the transport channel, must provide this information itself.

The channelBindings data structure is `Map<String,String>` with the keys as defined for the `ChannelBinding` structure in the FIDO UAF Protocol Specification. [UAFProtocol]

The use of channel bindings for TLS helps assure the server that the channel over which UAF protocol messages are transported is the same channel the legitimate client is using and that messages have not been forwarded through a malicious party. UAF defines support for the `tls-unique` and `tls-server-end-point` bindings from [RFC5929], as well as server certificate and `ChannelID` [ChannelID] bindings. The client should supply all channel binding information available to it.

Missing or invalid channel binding information may cause a relying party server to reject a transaction.

7.1.5 UAFxType

This value describes the type of operation for the `x-callback-url` operations implementing the iOS API.

WebIDL

```
enum UAFxType {
    "DISCOVER",
    "DISCOVER_RESULT",
    "CHECK_POLICY",
    "CHECK_POLICY_RESULT",
    "UAF_OPERATION",
    "UAF_OPERATION_RESULT",
    "UAF_OPERATION_COMPLETION_STATUS"
};
```

Enumeration description	
`DISCOVER`	Discovery
`DISCOVER_RESULT`	Discovery results
`CHECK_POLICY`	Perform a no-op check if a message could be processed.
`CHECK_POLICY_RESULT`	Check Policy results.
`UAF_OPERATION`	The UAF message operation type (for example `Registration`).
`UAF_OPERATION_RESULT`	UAF Operation results.
`UAF_OPERATION_COMPLETION_STATUS`	Inform the FIDO UAF Client of the completion status of a UAF operation (such as `Registration`.

7.2 JSON Values

The specifics of the UAFxType operation are carried by various JSON values encoded in the `json` `x-callback-url` parameter.

JSON value	Type	Description

JSON value	Type	Description
discoveryData	String	DiscoveryData JSON dictionary.
errorCode	short	ErrorCode value for operation
message	String	UAFMessage request to test or process, depending on UAFxType.
origin	String	An RFC6454 Web Origin [RFC6454] string for the request.
channelBindings	String	The channel bindings JSON dictionary for the operation.
responseCode	short	The uafResult field of a ServerResponse.

The following table shows what JSON values are expected, depending on the value of the UAFxType x-callback-url operation:

UAFxType operation	discoveryData	errorCode	message	origin	channelBindings	responseCode
"DISCOVER"						
"DISCOVER_RESULT"	optional	required				
"CHECK_POLICY"			required	optional		
"CHECK_POLICY_RESULT"		required				
"UAF_OPERATION"			required	optional	required	
"UAF_OPERATION_RESULT"		required	optional			
"UAF_OPERATION_COMPLETION_STATUS"			required			required

7.2.1 DISCOVER

This operation invokes the FIDO UAF Client to discover the available authenticators and capabilities. The FIDO UAF Client generally will not show a user interface associated with the handling of this operation, but will simply return the resulting JSON structure.

The calling application cannot depend on this however, as the client may show a user interface for privacy purposes, allowing the user to choose whether and which authenticators to disclose to the calling application.

> NOTE
>
> iOS custom URL scheme handlers always require an application switch for every request and response, even if no user interface is displayed.

7.2.2 DISCOVER_RESULT

An operation with this type is returned by the FIDO UAF Client in response to receiving an x-callback-url operation of type DISCOVER.

If the resultCode is RESULT_OK, and the JSON value errorCode is NO_ERROR, then this operation has a JSON value, discoveryData, containing a String representation of a DiscoveryData JSON dictionary listing the available authenticators and their capabilities.

7.2.3 CHECK_POLICY

This operation invokes the FIDO UAF Client to discover if the client would be able to process the supplied message, without prompting the user.

The related Action handling this operation should not show an interface to the user.

> NOTE
>
> iOS custom URL scheme handlers always require an application switch for every request and response, even if no UI is displayed.

This x-callback-url operation requires the following JSON values:

- message, containing a String representation of a UAFMessage representing the request message to test.
- origin, an optional JSON value that allows a caller to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

7.2.4 CHECK_POLICY_RESULT

This operation is returned by the FIDO UAF Client in response to receiving a CHECK_POLICY x-callback-url operation.

In addition to the resultCode, this x-callback-url operation has a JSON value, errorCode, containing an ErrorCode value indicating the specific error condition or NO_ERROR if the FIDO UAF Cliet could process the message.

7.2.5 UAF_OPERATION

This operation invokes the FIDO UAF Client to process the supplied request message and return a result message ready for delivery to the FIDO UAF Server. The sender should assume that the FIDO UAF Client will display a UI to the user to handle this x-callback-url operation, e.g. prompting the user to complete their verification ceremony.

This x-callback-url operation requires the following JSON values:

- message, containing a String representation of a UAFMessage representing the request message to process.
- channelBindings, containing a String representation of a JSON dictionary as defined by the ChannelBinding structure in the UAF Protocol Specification [UAFProtocol].
- origin, an optional JSON value that allows a caller to supply an RFC6454 Origin [RFC6454] string to be used instead of the application's own identity.

7.2.6 UAF_OPERATION_RESULT

This x-callback-url operation is returned by the FIDO UAF Client in response to receiving a UAF_OPERATION x-callback-url operation.

If the resultCode is RESULT_CANCELLED, this x-callback-url operation has a JSON value, errorCode, containing an **ErrorCode** value indicating the specific error condition.

If the resultCode is RESULT_OK,, and the x-callback-url JSON value errorCode is NO_ERROR, this x-callback-url operation has a JSON value, message, containing a String representation of a **UAFMessage**, being the UAF protocol response message to be delivered to the FIDO Server.

7.2.7 UAF_OPERATION_COMPLETION_STATUS

This x-callback-url operation must be delivered to the FIDO UAF Client to indicate the completion status of a FIDO UAF message delivered to the remote server. This is especially important as, e.g. a new registration may be considered in a pending status until it is known the server accepted it.

7.3 Implementation Guidelines for iOS Implementations

Each iOS Custom URL based request results in a human-noticeable context switch between the App and FIDO UAF Client and vice versa. This will be most noticeable when invoking DISCOVER and CHECK_POLICY requests since typically these requests will be invoked automatically, without user's involvement. Such a context switch impacts the User Experience and therefore it's recommended to avoid making these two requests and integrate FIDO without using them.

7.4 Security Considerations for iOS Implementations

This section is non-normative.

A security concern with custom URLs under iOS is that any app can register any custom URL. If multiple applications register the same custom URL, the behavior for handling the URL call in iOS is undefined.

On the FIDO UAF Client side, this issue with custom URL scheme handlers is solved by using the sourceApplication parameter which provides the bundle ID of the URL originator. This is effective as long as the device has not been jailbroken and as long as Apple has done due diligence vetting submissions to the app store for malware with faked bundle IDs. The sourceApplication parameter can be matched with the FacetID list to ensure that the calling app is approved to use the credentials for the relying party.

On the relying party app side, encryption is used to prevent a rogue app from spoofing the relying party app's response URL. The relying party app generates a random encryption key on every request and sends it to the FIDO client. The FIDO client then encrypts the response to this key. In this manner, only the relying party app can decrypt the response. Even in the event that malware is able to spoof the relying party app's URL and intercept the response, it would not be able to decode it.

To protect against potentially malicious applications registering themselves to handle the FIDO UAF Client custom URL scheme, relying party Applications can obtain the bundle-id of the responding app and utilize it in risk management decisions around the authentication or transaction events. For example, a relying party might maintain a list of bundle-ids known to belong to malware and refuse to accept operations completed with such clients, or a list of bundle-ids of known-good clients that receive preferred risk-scoring.

8. Transport Binding Profile

This section is normative.

This section describes general normative security requirements for how a client application transports FIDO UAF protocol messages, gives specific requirements for Transport Layer Security [TLS], and describes an interoperability profile for using HTTP over TLS [RFC2818] with the FIDO UAF protocol.

8.1 Transport Security Requirements

This section is non-normative.

The UAF protocol contains no inherent means of identifying a relying party server, or for end-to-end protection of UAF protocol messages. To perform a secure UAF protocol exchange, the following abstract requirements apply:

1. The client application must securely authenticate the server endpoint as authorized, from that client's viewpoint, to represent the Web origin [RFC6454] (scheme:host:port tuple) reported to the FIDO UAF Client by the client application. Most typically this will be done by using TLS and verifying the server's certificate is valid, asserts the correct DNS name, and chains up to a root trusted by the client platform. Clients may also utilize other means to authenticate a server, such as via a pre-provisioned certificate or key that is distributed with an application, or alternative network authentication protocols such as Kerberos [RFC4120].
2. The transport mechanism for UAF protocol messages must provide confidentiality for the message, to prevent disclosure of their contents to unauthorized third parties. These protections should be cryptographically bound to proof of the server's identity as described above.
3. The transport mechanism for UAF protocol messages must protect the integrity of the message from tampering by unauthorized third parties. These protections should be cryptographically bound to proof of the server's identity in as described above.

8.2 TLS Security Requirements

This section is non-normative.

If using HTTP over TLS ([RFC2246] [RFC4346], [RFC5246] or [TLS13draft02]) to transport an UAF protocol exchange, the following specific requirements apply:

1. 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 [RFC6125], Certificate Revocation Lists (CRLs) [RFC5280], or via the Online Certificate Status Protocol (OCSP) [RFC2560].
2. 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), [RFC6125] server identity checking must be employed. The client must terminate the connection without prompting the user upon any error condition.
3. 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 user recourse if there are any error conditions when building the chain of trust.
4. 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 [SP800-131A]).
5. The client and server should use the latest practicable TLS version.
6. The client should supply, and the server should verify whatever practicable channel binding information is available, including a ChannelID [ChannelID] public key, the tls-unique and tls-server-end-point bindings [RFC5929], and TLS server certificate binding [UAFProtocol]. This information provides protection against certain classes of network attackers and the forwarding of protocol messages, and a server may reject a message that lacks or has channel binding data that does not verify correctly.

8.3 HTTPS Transport Interoperability Profile

This section is normative.

Conforming applications may support this profile.

Complex and highly-optimized applications utilizing UAF will often transport UAF protocol messages in-line with other application protocol messages. The profile defined here for transporting UAF protocol messages over HTTPS is intended to:

- Provide an interoperability profile to enable easier composition of client-side application libraries and server-side implementations for FIDO UAF-enabled products from different vendors.
- Provide detailed illustration of specific necessary security properties for the transport layer and HTTP interfaces, especially as they may interact with a browser-hosted application.
- This profile is also utilized in the examples that constitute the appendices of this document. This profile is optional to implement. RFC 2119 key words are used in this section to indicate necessary security and other properties for implementations that intend to use this profile to interoperate.
- NOTE

  Certain FIDO UAF operations, in particular, transaction confirmation, will always require an application-specific implementation. This interoperability profile only provides a skeleton framework suitable for replacing username/password authentication.

8.3.1 Obtaining a UAF Request message

A UAF-enabled web application might typically deliver request messages as part of a response body containing other application content, e.g in a script block as such:

EXAMPLE 12

```
...

<script type="application/json">
{
"initialRequest": {
// initial request message here
},

"lifetimeMillis": 60000; // hint: this initial request is valid for 60 seconds
}
</script>

...
```

However, request messages have a limited lifetime, and an installed application cannot be delivered with a request, so client applications generally need the ability to retrieve a fresh request.

When sending a request message over HTTPS with XMLHttpRequest [XHR] or another HTTP API:

1. The URI of the server endpoint, and how it is communicated to the client, is application-specific.
2. The client must set the HTTP method to POST. [RFC7231]
3. The client must set the HTTP "Content-Type" header to `"application/fido+uaf; charset=utf-8".` [RFC7231]
4. The client should include `"application/fido+uaf"` as a media type in the HTTP "Accept" header. [RFC7231]
5. The client may need to supply additional headers, such as a HTTP Cookie [RFC6265], to demonstrate, in an application-specific manner, their authorization to perform a request.
6. The entire POST body must consist entirely of a JSON [ECMA-404] structure described by the `GetUAFRequest` dictionary.
7. The server's response should set the HTTP "Content-Type" to `"application/fido+uaf; charset=utf-8"`
8. The client should decode the response byte string as UTF-8 with error handling. [HTML5]
9. The decoded body of the response must consist entirely of a JSON structure described by the `ReturnUAFRequest` interface.

8.3.2 Operation enum

Describes the operation type of a FIDO UAF message or request for a message.

WebIDL

```
enum Operation {
    "Reg",
    "Auth",
    "Dereg"
};
```

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

8.3.3 GetUAFRequest dictionary

WebIDL

```
dictionary GetUAFRequest {
    Operation op;
    DOMString previousRequest;
    DOMString context;
};
```

8.3.3.1 Dictionary `GetUAFRequest` Members

`op` of type *Operation*
: The type of the UAF request message desired. Allowable string values are defined by the Operation enum. This field is optional but must be set if the operation is not known to the server through other context, e.g. an operation-specific URL endpoint.

previousRequest of type DOMString
: If the application is requesting a new UAF request message because a previous one has expired, this optional key can include the previous one to assist the server in locating any state that should be re-associated with a new request message, should one be issued.

context of type DOMString
: Any additional contextual information that may be useful or necessary for the server to generate the correct request message. This key is optional and the format and nature of this data is application-specific.

8.3.4 ReturnUAFRequest dictionary

WebIDL

```
dictionary ReturnUAFRequest {
    required unsigned long statusCode;
    DOMString              uafRequest;
    Operation              op;
    long                   lifetimeMillis;
};
```

8.3.4.1 Dictionary `ReturnUAFRequest` Members

statusCode of type required unsigned long
: The UAF Status Code for the operation (see section 3.1 UAF Status Codes).

uafRequest of type DOMString
: The new UAF Request Message, optional, if the server decided to issue one

op of type *Operation*
: An optional hint to the client of the operation type of the message, useful if the server might return a different type than was requested. For example, a server might return a deregister message if an authentication request referred to a key it no longer considers valid. Allowable string values are defined by the Operation enum.

lifetimeMillis of type long
: If the server returned a `uafRequest`, this is an optional hint informing the client application of the lifetime of the message in milliseconds.

8.3.5 SendUAFResponse dictionary

WebIDL

```
dictionary SendUAFResponse {
    required DOMString uafResponse;
    DOMString          context;
};
```

8.3.5.1 Dictionary `SendUAFResponse` Members

uafResponse of type required DOMString
: The UAF Response Message. It must be set to `UAFMessage.uafProtocolMessage` returned by FIDO UAF Client.

context of type DOMString
: Any additional contextual information that may be useful or necessary for the server to process the response message. This key is optional and the format and nature of this data is application-specific.

8.3.6 Delivering a UAF Response

Although it is not the only pattern possible, an asynchronous HTTP request is a useful way of delivering a UAF Response to the remote server for either web applications or standalone applications.

When delivering a response message over HTTPS with XMLHttpRequest [XHR] or another API:

1. The URI of the server endpoint and how it is communicated to the client is application-specific.
2. The client must set the HTTP method to POST. [RFC7231]
3. The client must set the HTTP “Content-Type” header to `"application/fido+uaf; charset=utf-8"`. [RFC7231]
4. The client should include `"application/fido+uaf"` as a media type in the HTTP “Accept” header. [RFC7231]
5. The client may need to supply additional headers, such as a HTTP Cookie [RFC6265], to demonstrate, in an application-specific manner, their authorization to perform an operation.
6. The entire POST body must consist entirely of a JSON [ECMA-404] structure described by the `SendUAFResponse`.
7. The server's response should set the “Content-Type” to `"application/fido+uaf; charset=utf-8"` and the body of the response must consist entirely of a JSON structure described by the `ServerResponse` interface.

8.3.7 ServerResponse Interface

The `ServerResponse` interface represents the completion status and additional application-specific additional data that results from successful processing of a Register, Authenticate, or Transaction Confirmation operation. This message is not formally part of the UAF protocol, but the `statusCode` should be posted to the FIDO UAF Client, for housekeeping, using the `notifyUAFResult()` operation.

WebIDL

```
interface ServerResponse {
    readonly    attribute int       statusCode;
    [Optional]
    readonly    attribute DOMString description;
    [Optional]
    readonly    attribute Token[]   additionalTokens;
    [Optional]
    readonly    attribute DOMString location;
    [Optional]
    readonly    attribute DOMString postData;
    [Optional]
    readonly    attribute DOMString newUAFRequest;
};
```

8.3.7.1 Attributes

`statusCode` of type int, readonly
: The FIDO UAF response status code. Note that this status code describes the result of processing the tunneled UAF operation, not the status code for the outer HTTP transport.

`description` of type DOMString, readonly
: A detailed message describing the status code or providing additional information to the user.

`additionalTokens` of type array of *Token*, readonly
: This key contains new authentication or authorization token(s) for the client that are 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 specified location after processing any tokens.

`newUAFRequest` of type DOMString, readonly
: The server may use this to return a new UAF protocol message. This might be used to supply a fresh request to retry an operation in response to a transient failure, to request additional confirmation for a transaction, or to send a deregistration message in response to a permanent failure.

8.3.8 Token interface

> NOTE
>
> The UAF Server is not responsible for creating additional tokens returned as part of a UAF response. Such tokens exist to provide a means for the relying party application to update the authentication/authorization state of the client in response to a successful UAF operation. For example, these fields could be used to allow UAF to serve as the initial authentication leg of a federation protocol, but the scope and details of any such federation are outside of the scope of UAF.

WebIDL

```
interface Token {
    readonly    attribute TokenType type;
    readonly    attribute DOMString value;
};
```

8.3.8.1 Attributes

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

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

8.3.9 TokenType enum

WebIDL

```
enum TokenType {
    "HTTP_COOKIE",
    "OAUTH",
    "OAUTH2",
    "SAML1_1",
    "SAML2",
    "JWT",
    "OPENID_CONNECT"
};
```

Enumeration description	
`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 processing by the user agent. For non-HTTP or non-browser contexts this indicates a token intended to be set as an HTTP cookie. [RFC6265] For example, a native VPN client that authenticates with UAF might use this TokenType to automatically add a cookie to the browser cookie jar.
`OAUTH`	Indicates that the token is of type OAUTH. [RFC5849].
`OAUTH2`	Indicates that the token is of type OAUTH2. [RFC6749].
`SAML1_1`	Indicates that the token is of type SAML 1.1. [SAML11].
`SAML2`	Indicates that the token is of type SAML 2.0. [SAML2-CORE]
`JWT`	Indicates that the token is of type JSON Web Token (JWT). [JWT]
`OPENID_CONNECT`	Indicates that the token is an OpenID Connect “id_token”. [OpenIDConnect]

8.3.10 Security Considerations

This section is non-normative.

It is important that the client set, and the server require, the method be POST and the “Content-Type” HTTP header be the correct values. Because the response body is valid ECMAScript, to protect against unauthorized cross-origin access, a server must not respond to the type of request that can be generated by a script tag, e.g. `<script src="https://example.com/fido/uaf/getRequest">`. The request a user agent generates with this kind of embedding cannot set custom headers.

Likewise, by requiring a custom “Content-Type” header, cross-origin requests cannot be made with an XMLHttpRequest [XHR] without triggering a CORS preflight access check. [CORS]

As FIDO UAF messages are only valid when used same-origin, servers should not supply an “Access-Control-Allow-Origin” [CORS] header with responses that would allow them to be read by non-same-origin content.

To protect from some classes of cross-origin, browser-based, distributed denial-of-service attacks, request endpoints should ignore, without performing additional processing, all requests with an "Access-Control-Request-Method" [CORS] HTTP header or an incorrect "Content-Type" HTTP header.

If a server chooses to respond to requests made with the GET method and without the custom "Content-Type" header, it should apply a prefix string such as "`while(1);`" or "`&&&BEGIN_UAF_RESPONSE&&&`" to the body of all replies and so prevent their being read through cross-origin <script> tag embedding. Legitimate same-origin callers will need to (and alone be able to) strip this prefix string before parsing the JSON content.

A. References

A.1 Normative references

[AndroidAppManifest]
Android App Manifest. Google, Inc., the Open Handset Alliance and the Android Open Source Project (Work in progress) URL: http://developer.android.com/guide/topics/manifest/manifest-intro.html
[ChannelID]
D. Balfanz *Transport Layer Security (TLS) Channel IDs*. (Work In Progress) URL:http://tools.ietf.org/html/draft-balfanz-tls-channelid
[DOM]
Anne van Kesteren; Aryeh Gregor; Ms2ger; Alex Russell; Robin Berjon. *W3C DOM4*. 19 November 2015. W3C Recommendation. URL: https://www.w3.org/TR/dom/
[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/
[ECMA-404]
The JSON Data Interchange Format. 1 October 2013. Standard. URL:http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf
[FIDOMetadataStatement]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO Metadata Statements v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.pdf
[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.pdf
[HTML5]
I. Hickson; R.Berjon; S. Faulkner; T. Leithead; E. D. Navara; E. O'Connor; S. Pfeiffer. *HTML5: A vocabulary and associated APIs for HTML and XHTML*. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/
[JWT]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Token (JWT)*. May 2015. RFC. URL:https://tools.ietf.org/html/rfc7519
[OpenIDConnect]
OpenID Connect. OpenID Foundation (Work in Progress) URL:http://openid.net/connect/
[PNG]
Tom Lane. *Portable Network Graphics (PNG) Specification (Second Edition)*. 10 November 2003. W3C Recommendation. URL: https://www.w3.org/TR/PNG
[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL:https://tools.ietf.org/html/rfc2119
[RFC2397]
L. Masinter. *The "data" URL scheme*. August 1998. Proposed Standard. URL:https://tools.ietf.org/html/rfc2397
[RFC2818]
E. Rescorla. *HTTP Over TLS*. May 2000. Informational. URL:https://tools.ietf.org/html/rfc2818
[RFC4648]
S. Josefsson, *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*, IETF, October 2006, URL:http://www.ietf.org/rfc/rfc4648.txt
[RFC5849]
E. Hammer-Lahav, *The OAuth 1.0 Protocol (RFC 5849)*, IETF, April 2010, URL:http://www.ietf.org/rfc/rfc5849.txt
[RFC5929]
J. Altman, N. Williams, L. Zhu, *Channel Bindings for TLS (RFC 5929)*, IETF, July 2010, URL:http://www.ietf.org/rfc/rfc5929.txt
[RFC6125]
P. Saint-Andre, J. Hodges, *Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS) (RFC 6125)*, IETF, March 2011, URL: http://www.ietf.org/rfc/rfc6125.txt
[RFC6265]
A. Barth. *HTTP State Management Mechanism*. April 2011. Proposed Standard. URL:https://tools.ietf.org/html/rfc6265
[RFC6454]
A. Barth, *The Web Origin Concept (RFC 6454)*, IETF, June 2011, URL:http://www.ietf.org/rfc/rfc6454.txt
[RFC6749]
D. Hardt, Ed., *The OAuth 2.0 Authorization Framework (RFC 6749)*, IETF, October 2012, URL:http://www.ietf.org/rfc/rfc6749.txt
[RFC7230]
R. Fielding, Ed.; J. Reschke, Ed.. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7230
[RFC7231]
R. Fielding, Ed.; J. Reschke, Ed.. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7231
[SAML11]
E. Maler, P. Mishra and R. Philpott, *The Security Assertion Markup Language (SAML) v1.1*. OASIS, October 2003, URL:https://www.oasis-open.org/standards#samlv1.1
[SAML2-CORE]
Scott Cantor; John Kemp; Rob Philpott; Eve Maler. *Assertions and Protocols for SAML V2.0* 15 March 2005. URL:http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf
[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges, *FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf
[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.pdf
[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[ANDROID]
The Android™ Operating System. Google, Inc., the Open Handset Alliance and the Android Open Source Project (Work in progress) URL: http://developer.android.com/
[Android5Changes]
Android 5.0 Changes. Google, Inc., the Open Handset Alliance and the Android Open Source Project (Work in progress) URL: http://developer.android.com/about/versions/android-5.0-changes.html

[CORS]
Anne van Kesteren. *Cross-Origin Resource Sharing*. 16 January 2014. W3C Recommendation. URL: https://www.w3.org/TR/cors/
[RFC2045]
N. Freed; N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* November 1996. Draft Standard. URL: https://tools.ietf.org/html/rfc2045
[RFC2246]
T. Dierks, E. Rescorla, *The TLS Protocol Version 1.0*, IETF, January 1999, URL: http://www.ietf.org/rfc/rfc2246.txt
[RFC2560]
M. Myers; R. Ankney; A. Malpani; S. Galperin; C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. June 1999. Proposed Standard. URL: https://tools.ietf.org/html/rfc2560
[RFC4120]
C. Neuman, T. Yu, S. Hartman, K. Raeburn, *The Kerberos Network Authentication Protocol (V5) (RFC 4120)*, IETF, July 2005, URL: http://www.ietf.org/rfc/rfc4120.txt
[RFC4346]
T. Dierks, E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.1*, IETF, April 2006, URL: http://www.ietf.org/rfc/rfc4346.txt
[RFC5246]
T. Dierks, E. Rescorla, *The Transport Layer Security (TLS) Protocol*, IETF, August 2008, URL: http://www.ietf.org/rfc/rfc5246.txt
[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk; *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL: http://www.ietf.org/rfc/rfc5280.txt
[SOP]
Same Origin Policy for JavaScript. Mozilla Developer Network, January 2014 URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript
[SP800-131A]
E. Barker, A. Roginsky, *NIST Special Publication 800-131A: Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. National Institute of Standards and Technology, January 2011, URL: http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf
[TLS13draft02]
T. Dierks, E. Rescorla, *The Transport Layer Security (TLD) Protocol Version 1.3 (draft 02)*, IETF, July, 2014, URL: https://tools.ietf.org/html/draft-ietf-tls-tls13-02
[UAFASM]
D. Baghdasaryan, J. Kemp, R. Lindemann, B. Hill, R. Sasson, *FIDO UAF Authenticator-Specific Module API*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.pdf
[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Working Draft. URL: https://www.w3.org/TR/WebIDL-1/
[XHR]
Anne van Kesteren. *XMLHttpRequest Standard*. Living Standard. URL: https://xhr.spec.whatwg.org/
[webmessaging]
Ian Hickson. *HTML5 Web Messaging*. 19 May 2015. W3C Recommendation. URL: https://www.w3.org/TR/webmessaging/

FIDO UAF Authenticator-Specific Module API

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-asm-api-v1.0-ps-20141208.html
Editors:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
John Kemp, FIDO Alliance
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal, Inc.
Roni Sasson, Discretix, Inc.
Jeff Hodges, PayPal, Inc.
Ka Yang, Nok Nok Labs, Inc.

Abstract

UAF authenticators may be connected to a user device via various physical interfaces (SPI, USB, Bluetooth, etc). The UAF Authenticator-Specific Module (ASM) is a software interface on top of UAF authenticators which gives a standardized way for FIDO UAF Clients to detect and access the functionality of UAF authenticators and hides internal communication complexity from FIDO UAF Client.

This document describes the internal functionality of ASMs, defines the UAF ASM API and explains how FIDO UAF Clients should use the API.

This document's intended audience is FIDO authenticator and FIDO FIDO UAF Client vendors.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

DOM APIs are described using the ECMAScript [ECMA-262] bindings for WebIDL [WebIDL-ED].

The notation base64url refers to "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as `required`.

WebIDL dictionary members must not have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

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

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

NOTE

Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

UAF authenticators may be connected to a user device via various physical interfaces (SPI, USB, Bluetooth, etc). The UAF Authenticator-Specific module (ASM) is a software interface on top of UAF authenticators which gives a standardized way for FIDO UAF Clients to detect and access the functionality of UAF authenticators, and hides internal communication complexity from clients.

The ASM is a platform-specific software component offering an API to FIDO UAF Clients, enabling them to discover and communicate with one or more available authenticators.

A single ASM may report on behalf of multiple authenticators.

The intended audience for this document is FIDO UAF authenticator and FIDO UAF Client vendors.

> NOTE
>
> Platform vendors might choose to not expose the ASM API defined in this document to applications. They might instead choose to expose ASM functionality through some other API (such as, for example, the Android KeyStore API, or iOS KeyChain API). In these cases it's important to make sure that the underlying ASM communicates with the FIDO UAF authenticator in a manner defined in this document.

The FIDO UAF protocol and its various operations is described in the FIDO UAF Protocol Specification [UAFProtocol]. The following simplified architecture diagram illustrates the interactions and actors this document is concerned with:

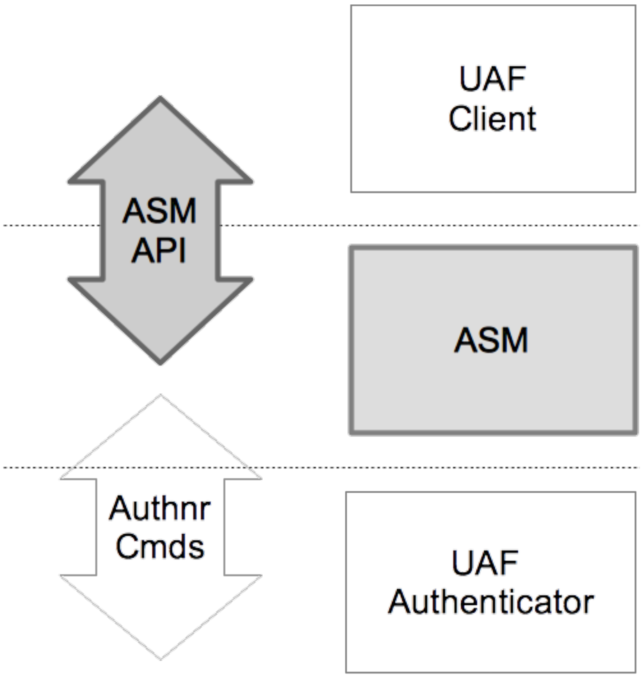


Fig. 1 UAF ASM API Architecture

2.1 Code Example format

ASM requests and responses are presented in WebIDL format.

3. ASM Requests and Responses

This section is normative.

The ASM API is defined in terms of JSON-formatted [ECMA-404] request and reply messages. In order to send a request to an ASM, a FIDO UAF Client creates an appropriate object (e.g., in ECMAscript), "stringifies" it (also known as serialization) into a JSON-formated string, and sends it to the ASM. The ASM de-serializes the JSON-formatted string, processes the request, constructs a response, stringifies it, returning it as a JSON-formatted string.

> NOTE
>
> The ASM request processing rules in this document explicitly assume that the underlying authenticator implements the "UAFV1TLV" assertion scheme (e.g. references to TLVs and tags) as described in [UAFProtocol]. If an authenticator supports a different assertion scheme then the corresponding processing rules must be replaced with appropriate assertion scheme-specific rules.

Authenticator implementers may create custom authenticator command interfaces other than the one defined in [UAFAuthnrCommands]. Such implementations are not required to implement the exact message-specific processing steps described in this section. However,

1. the command interfaces must present the ASM with external behavior equivalent to that described below in order for the ASM to properly respond to the client request messages (e.g. returning appropriate UAF status codes for specific conditions).
2. all authenticator implementations must support an assertion scheme as defined [UAFRegistry] and must return the related objects, i.e. `TAG_UAFV1_REG_ASSERTION` and `TAG_UAFV1_AUTH_ASSERTION` as defined in [UAFAuthnrCommands].

3.1 Request enum

WebIDL

```
enum Request {
    "GetInfo",
    "Register",
    "Authenticate",
    "Deregister",
    "GetRegistrations",
    "OpenSettings"
};
```

Enumeration description	
`GetInfo`	GetInfo
`Register`	Register
`Authenticate`	Authenticate
`Deregister`	Deregister
`GetRegistrations`	GetRegistrations
`OpenSettings`	OpenSettings

3.2 StatusCode Interface

If the ASM needs to return an error received from the authenticator, it shall map the status code received from the authenticator to the appropriate ASM status code as specified here.

If the ASM doesn't understand the authenticator's status code, it shall treat it as `UAF_CMD_STATUS_ERR_UNKNOWN` and map it to `UAF_ASM_STATUS_ERROR` if it cannot be handled otherwise.

If the caller of the ASM interface (i.e. the FIDO Client) doesn't understand a status code returned by the ASM, it shall treat it as `UAF_ASM_STATUS_ERROR`. This might occur when new error codes are introduced.

WebIDL

```
interface StatusCode {
    const short UAF_ASM_STATUS_OK = 0x00;
    const short UAF_ASM_STATUS_ERROR = 0x01;
    const short UAF_ASM_STATUS_ACCESS_DENIED = 0x02;
    const short UAF_ASM_STATUS_USER_CANCELLED = 0x03;
    const short UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT = 0x04;
    const short UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY = 0x09;
    const short UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED = 0x0b;
    const short UAF_ASM_STATUS_USER_NOT_RESPONSIVE = 0x0e;
    const short UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES = 0x0f;
    const short UAF_ASM_STATUS_USER_LOCKOUT = 0x10;
    const short UAF_ASM_STATUS_USER_NOT_ENROLLED = 0x11;
};
```

3.2.1 Constants

`UAF_ASM_STATUS_OK` of type short
: No error condition encountered.

`UAF_ASM_STATUS_ERROR` of type short
: An unknown error has been encountered during the processing.

`UAF_ASM_STATUS_ACCESS_DENIED` of type short
: Access to this request is denied.

`UAF_ASM_STATUS_USER_CANCELLED` of type short
: Indicates that user explicitly canceled the request.

`UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT` of type short
: Transaction content cannot be rendered, e.g. format doesn't fit authenticator's need.

`UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY` of type short
: Indicates that the UAuth key disappeared from the authenticator and canot be restored.

`UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED` of type short
: Indicates that the authenticator is no longer connected to the ASM.

`UAF_ASM_STATUS_USER_NOT_RESPONSIVE` of type short
: The user took too long to follow an instruction, e.g. didn't swipe the finger within the accepted time.

`UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES` of type short
: Insufficient resources in the authenticator to perform the requested task.

`UAF_ASM_STATUS_USER_LOCKOUT` of type short
: The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that. Typically the user would have to enter an alternative password (formally: undergo some other alternative user verification method) to re-enable the use of the main user verification method.

> NOTE
>
> Any method the user can use to (re-) enable the main user verification method is considered an alternative user verification method and must be properly declared as such. For example, if the user can enter an alternative password to re-enable the use of fingerprints or to add additional fingers, the authenticator obviously supports fingerprint *or* password based user verification.

`UAF_ASM_STATUS_USER_NOT_ENROLLED` of type short
: The operation failed because the user is not enrolled to the authenticator and the authenticator cannot automatically trigger user enrollment.

3.2.2 Mapping Authenticator Status Codes to ASM Status Codes

Authenticators are returning a status code in their responses to the ASM. The ASM needs to act on those responses and also map the status code returned by the authenticator to an ASM status code.

The mapping of authenticator status codes to ASM status codes is specified here:

Authenticator Status Code	ASM Status Code	Comment
`UAF_CMD_STATUS_OK`	`UAF_ASM_STATUS_OK`	Pass-through success status.
`UAF_CMD_STATUS_ERR_UNKNOWN`	`UAF_ASM_STATUS_ERROR`	Pass-through unspecific error status.
`UAF_CMD_STATUS_ACCESS_DENIED`	`UAF_ASM_STATUS_ACCESS_DENIED`	Pass-through status code.
		According to [UAFAuthnrCommands], t

Authenticator Status Code	ASM Status Code	Comment
UAF_CMD_STATUS_USER_NOT_ENROLLED	UAF_ASM_STATUS_USER_NOT_ENROLLED (or UAF_ASM_STATUS_ACCESS_DENIED in some situations)	might occur at the Sign command and or at *Register* command if the authenticator automatically trigger user enrollment. T mapping depends on the command as In the case of "Register" command, the mapped to UAF_ASM_STATUS_USER_NOT_EN in order to tell the calling FIDO Client tl is an authenticator present but the user enrollment needs to be triggered outsic authenticator. In the case of the "Sign" command, the key needs to be protected by one of th authenticator's user verification methoc times. So if this error occurs it is consic internal error and hence mapped to UAF_ASM_STATUS_ACCESS_DENIE
UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT	UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT	Pass-through status code as it indicate problem to be resolved by the entity pr the transaction text.
UAF_CMD_STATUS_USER_CANCELLED	UAF_ASM_STATUS_USER_CANCELLED	Map to UAF_ASM_STATUS_USER_CANCELLEI
UAF_CMD_STATUS_CMD_NOT_SUPPORTED	UAF_ASM_STATUS_OK or UAF_ASM_STATUS_ERROR	If the ASM is able to handle that comm behalf of the authenticator (e.g. removi key handle in the case of *Dereg* comm bound authenticator), the UAF_ASM_STAT must be returned. Map the status code UAF_ASM_STATUS_ERROR otherwise.
UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED	UAF_ASM_STATUS_ERROR	Indicates an ASM issue as the ASM ha obviously not requested one of the sup attestation types indicated in the authe response to the *GetInfo* command.
UAF_CMD_STATUS_PARAMS_INVALID	UAF_ASM_STATUS_ERROR	Indicates an ASM issue as the ASM ha obviously not provided the correct para to the authenticator when sending the command.
UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY	UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY	Pass-through status code. It indicates t Uauth key disappeared permanently ar RP App might want to trigger re-registr the authenticator.
UAF_STATUS_CMD_TIMEOUT	UAF_ASM_STATUS_ERROR	Retry operation and map to UAF_ASM_STATUS_ERROR if the problem pe
UAF_CMD_STATUS_USER_NOT_RESPONSIVE	UAF_ASM_STATUS_USER_NOT_RESPONSIVE	Pass-through status code. The RP App want to retry the operation once the us attention to the application again.
UAF_CMD_STATUS_INSUFFICIENT_RESOURCES	UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES	Pass-through status code.
UAF_CMD_STATUS_USER_LOCKOUT	UAF_ASM_STATUS_USER_LOCKOUT	Pass-through status code.
Any other status code	UAF_ASM_STATUS_ERROR	Map any unknown error code to UAF_ASM_STATUS_ERROR. This might happ an ASM communicates with an authen implementing a newer UAF specificatic the ASM.

3.3 ASMRequest Dictionary

All ASM requests are represented as ASMRequest objects.

WebIDL

```
dictionary ASMRequest {
    required Request requestType;
    Version          asmVersion;
    unsigned short   authenticatorIndex;
    object           args;
    Extension[]      exts;
};
```

3.3.1 Dictionary ASMRequest Members

requestType of type required Request
    Request type

asmVersion of type Version
    ASM message version to be used with this request. For the definition of the Version dictionary see [UAFProtocol]. The **asmVersion** must be 1.1 (i.e. major version is 1 and minor version is 1) for this version of the specification.

authenticatorIndex of type unsigned short
    Refer to the GetInfo request for more details. Field authenticatorIndex must not be set for GetInfo request.

args of type object
    Request-specific arguments. If set, this attribute may take one of the following types:

- RegisterIn
- AuthenticateIn
- DeregisterIn

exts of type array of Extension
: List of UAF extensions. For the definition of the `Extension` dictionary see [UAFProtocol].

3.4 ASMResponse Dictionary

All ASM responses are represented as `ASMResponse` objects.

WebIDL

```
dictionary ASMResponse {
    required short statusCode;
    object         responseData;
    Extension[]    exts;
};
```

3.4.1 Dictionary `ASMResponse` Members

statusCode of type required short
: must contain one of the values defined in the `StatusCode` interface

responseData of type object
: Request-specific response data. This attribute must have one of the following types:

- `GetInfoOut`
- `RegisterOut`
- `AuthenticateOut`
- `GetRegistrationOut`

exts of type array of Extension
: List of UAF extensions. For the definition of the `Extension` dictionary see [UAFProtocol].

3.5 GetInfo Request

Return information about available authenticators.

1. Enumerate all of the authenticators this ASM supports
2. Collect information about all of them
3. Assign indices to them (`authenticatorIndex`)
4. Return the information to the caller

NOTE

Where possible, an `authenticatorIndex` should be a persistent identifier that uniquely identifies an authenticator over time, even if it is repeatedly disconnected and reconnected. This avoids possible confusion if the set of available authenticators changes between a `GetInfo` request and subsequent ASM requests, and allows a FIDO client to perform caching of information about removable authenticators for a better user experience.

NOTE

It is up to the ASM to decide whether authenticators which are disconnected temporarily will be reported or not. However, if disconnected authenticators are reported, the FIDO Client might trigger an operation via the ASM on those. The ASM will have to notify the user to connect the authenticator and report an appropriate error if the authenticator isn't connected in time.

For a GetInfo request, the following `ASMRequest` member(s) must have the following value(s). The remaining `ASMRequest` members should be omitted:

- `ASMRequest.requestType` must be set to `GetInfo`

For a GetInfo response, the following `ASMResponse` member(s) must have the following value(s). The remaining `ASMResponse` members should be omitted:

- `ASMResponse.statusCode` must have one of the following values
  - `UAF_ASM_STATUS_OK`
  - `UAF_ASM_STATUS_ERROR`
- `ASMResponse.responseData` must be an object of type `GetInfoOut`. In the case of an error the values of the fields might be empty (e.g. array with no members).

See section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes for details on the mapping of authenticator status codes to ASM status codes.

3.5.1 GetInfoOut Dictionary

WebIDL

```
dictionary GetInfoOut {
    required AuthenticatorInfo[] Authenticators;
};
```

3.5.1.1 Dictionary `GetInfoOut` Members

Authenticators of type array of required AuthenticatorInfo
: List of authenticators reported by the current ASM. may be empty an empty list.

3.5.2 AuthenticatorInfo Dictionary

WebIDL

```
dictionary AuthenticatorInfo {
    required unsigned short                 authenticatorIndex;
    required Version[]                      asmVersions;
    required boolean                        isUserEnrolled;
    required boolean                        hasSettings;
    required AAID                           aaid;
    required DOMString                      assertionScheme;
    required unsigned short                 authenticationAlgorithm;
    required unsigned short[]               attestationTypes;
    required unsigned long                  userVerification;
    required unsigned short                 keyProtection;
    required unsigned short                 matcherProtection;
    required unsigned long                  attachmentHint;
    required boolean                        isSecondFactorOnly;
    required boolean                        isRoamingAuthenticator;
    required DOMString[]                    supportedExtensionIDs;
    required unsigned short                 tcDisplay;
    DOMString                               tcDisplayContentType;
    DisplayPNGCharacteristicsDescriptor[]   tcDisplayPNGCharacteristics;
    DOMString                               title;
    DOMString                               description;
    DOMString                               icon;
};
```

3.5.2.1 Dictionary `AuthenticatorInfo` Members

`authenticatorIndex` of type required unsigned short
: Authenticator index. Unique, within the scope of all authenticators reported by the ASM, index referring to an authenticator. This index is used by the UAF Client to refer to the appropriate authenticator in further requests.

`asmVersions` of type array of required Version
: A list of ASM Versions that this authenticator can be used with. For the definition of the `Version` dictionary see [UAFProtocol].

`isUserEnrolled` of type required boolean
: Indicates whether a user is enrolled with this authenticator. Authenticators which don't have user verification technology must always return true. Bound authenticators which support different profiles per operating system (OS) user must report enrollment status for the current OS user.

`hasSettings` of type required boolean
: A boolean value indicating whether the authenticator has its own settings. If so, then a FIDO UAF Client can launch these settings by sending a `OpenSettings` request.

`aaid` of type required AAID
: The "Authenticator Attestation ID" (AAID), which identifies the type and batch of the authenticator. See [UAFProtocol] for the definition of the AAID structure.

`assertionScheme` of type required DOMString
: The assertion scheme the authenticator uses for attested data and signatures.

AssertionScheme identifiers are defined in the UAF Protocol specification [UAFProtocol].

`authenticationAlgorithm` of type required unsigned short
: Indicates the authentication algorithm that the authenticator uses. Authentication algorithm identifiers are defined in are defined in [FIDORegistry] with `ALG_` prefix.

`attestationTypes` of type array of required unsigned short
: Indicates attestation types supported by the authenticator. Attestation type TAGs are defined in [UAFRegistry] with `TAG_ATTESTATION` prefix

`userVerification` of type required unsigned long
: A set of bit flags indicating the user verification method(s) supported by the authenticator. The values are defined by the `USER_VERIFY` constants in [FIDORegistry].

`keyProtection` of type required unsigned short
: A set of bit flags indicating the key protections used by the authenticator. The values are defined by the `KEY_PROTECTION` constants in [FIDORegistry].

`matcherProtection` of type required unsigned short
: A set of bit flags indicating the matcher protections used by the authenticator. The values are defined by the `MATCHER_PROTECTION` constants in [FIDORegistry].

`attachmentHint` of type required unsigned long
: A set of bit flags indicating how the authenticator is currently connected to the system hosting the FIDO UAF Client software. The values are defined by the `ATTACHMENT_HINT` constants defined in [FIDORegistry].

> NOTE
>
> 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 with similar semantics as part of UAF protocol messages.

`isSecondFactorOnly` of type required boolean
: Indicates whether the authenticator can be used only as a second factor.

`isRoamingAuthenticator` of type required boolean
: Indicates whether this is a roaming authenticator or not.

`supportedExtensionIDs` of type array of required DOMString
: List of supported UAF extension Ids. may be an empty list.

`tcDisplay` of type required unsigned short
: A set of bit flags indicating the availability and type of the authenticator's transaction confirmation display. The values are defined by the `TRANSACTION_CONFIRMATION_DISPLAY` constants in [FIDORegistry].

This value must be 0 if transaction confirmation is not supported by the authenticator.

`tcDisplayContentType` of type DOMString
: Supported transaction content type [FIDOMetadataStatement].

This value must be present if transaction confirmation is supported, i.e. `tcDisplay` is non-zero.

`tcDisplayPNGCharacteristics` of type array of DisplayPNGCharacteristicsDescriptor
Supported transaction Portable Network Graphic (PNG) type [FIDOMetadataStatement]. For the definition of the `DisplayPNGCharacteristicsDescriptor` structure see [FIDOMetadataStatement].

This list must be present if PNG-image based transaction confirmation is supported, i.e. `tcDisplay` is non-zero and `tcDisplayContentType` is `image/png`.

`title` of type DOMString
A human-readable short title for the authenticator. It should be localized for the current locale.

> NOTE
>
> If the ASM doesn't return a title, the FIDO UAF Client must provide a title to the calling App. See section "Authenticator interface" in [UAFAppAPIAndTransport].

`description` of type DOMString
Human-readable longer description of what the authenticator represents.

> NOTE
>
> This text should be localized for current locale.
>
> The text is intended to be displayed to the user. It might deviate from the description specified in the metadata statement for the authenticator [FIDOMetadataStatement].
>
> If the ASM doesn't return a description, the FIDO UAF Client will provide a description to the calling application. See section "Authenticator interface" in [UAFAppAPIAndTransport].

`icon` of type DOMString
Portable Network Graphic (PNG) format image file representing the icon encoded as a data: url [RFC2397].

> NOTE
>
> If the ASM doesn't return an icon, the FIDO UAF Client will provide a default icon to the calling application. See section "Authenticator interface" in [UAFAppAPIAndTransport].

3.6 Register Request

Verify the user and return an authenticator-generated UAF registration assertion.

For a Register request, the following **`ASMRequest`** member(s) must have the following value(s). The remaining **`ASMRequest`** members should be omitted:

- `ASMRequest.requestType` must be set to `Register`
- `ASMRequest.asmVersion` must be set to the desired version
- `ASMRequest.authenticatorIndex` must be set to the target authenticator index
- `ASMRequest.args` must be set to an object of type `RegisterIn`

For a Register response, the following **`ASMResponse`** member(s) must have the following value(s). The remaining **`ASMResponse`** members should be omitted:

- `ASMResponse.statusCode` must have one of the following values:
  - `UAF_ASM_STATUS_OK`
  - `UAF_ASM_STATUS_ERROR`
  - `UAF_ASM_STATUS_ACCESS_DENIED`
  - `UAF_ASM_STATUS_USER_CANCELLED`
  - `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`
  - `UAF_ASM_STATUS_USER_NOT_RESPONSIVE`
  - `UAF_ASM_STATUS_INSUFFICIENT_AUTHENTICATOR_RESOURCES`
  - `UAF_ASM_STATUS_USER_LOCKOUT`
  - `UAF_ASM_STATUS_USER_NOT_ENROLLED`
- `ASMResponse.responseData` must be an object of type `RegisterOut`. In the case of an error the values of the fields might be empty (e.g. empty strings).

3.6.1 RegisterIn Object

WebIDL

```
dictionary RegisterIn {
    required DOMString      appID;
    required DOMString      username;
    required DOMString      finalChallenge;
    required unsigned short attestationType;
};
```

3.6.1.1 Dictionary `RegisterIn` Members

`appID` of type required DOMString
The FIDO server Application Identity.

`username` of type required DOMString
Human-readable user account name

`finalChallenge` of type required DOMString
base64url-encoded challenge data [RFC4648]

attestationType of type required unsigned short
Single requested attestation type

3.6.2 RegisterOut Object

WebIDL

```
dictionary RegisterOut {
    required DOMString assertion;
    required DOMString assertionScheme;
};
```

3.6.2.1 Dictionary `RegisterOut` Members

assertion of type required DOMString
FIDO UAF authenticator registration assertion, base64url-encoded

assertionScheme of type required DOMString
Assertion scheme.

AssertionScheme identifiers are defined in the UAF Protocol specification [UAFProtocol].

3.6.3 Detailed Description for Processing the Register Request

Refer to [UAFAuthnrCommands] document for more information about the TAGs and structure mentioned in this paragraph.

1. Locate authenticator using `authenticatorIndex`. If the authenticator cannot be located, then fail with `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`.
2. If a user is already enrolled with this authenticator (such as biometric enrollment, PIN setup, etc. for example) then the ASM must request that the authenticator verifies the user.

   NOTE

   If the authenticator supports `UserVerificationToken` (see [UAFAuthnrCommands]), then the ASM must obtain this token in order to later include it with the `Register` command.

   If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return `UAF_ASM_STATUS_USER_LOCKOUT`.

   - If verification fails, return `UAF_ASM_STATUS_ACCESS_DENIED`
3. If the user is not enrolled with the authenticator then take the user through the enrollment process.
   - If neither the ASM nor the Authenticator can trigger the enrollment process, return `UAF_ASM_STATUS_USER_NOT_ENROLLED`.
   - If enrollment fails, return `UAF_ASM_STATUS_ACCESS_DENIED`
4. Construct `KHAccessToken` (see section KHAccessToken for more details)
5. Hash the provided `RegisterIn.finalChallenge` using the authenticator-specific hash function (`FinalChallengeHash`)

   An authenticator's preferred hash function information must meet the algorithm defined in the `AuthenticatorInfo.authenticationAlgorithm` field.
6. Create a `TAG_UAFV1_REGISTER_CMD` structure and pass it to the authenticator
   1. Copy `FinalChallengeHash`, `KHAccessToken`, `RegisterIn.Username`, `UserVerificationToken`, `RegisterIn.AppID`, `RegisterIn.AttestationType`
      1. Depending on `AuthenticatorType` some arguments may be optional. Refer to [UAFAuthnrCommands] for more information on authenticator types and their required arguments.
7. Invoke the command and receive the response. If the authenticator returns an error, handle that error appropriately. If the connection to the authenticator gets lost and cannot be restored, return `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`. If the operation finally fails, map the authenticator error code to the the appropriate ASM error code (see section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codesfor details).
8. Parse `TAG_UAFV1_REGISTER_CMD_RESP`
   1. Parse the content of `TAG_AUTHENTICATOR_ASSERTION` (e.g. `TAG_UAFV1_REG_ASSERTION`) and extract `TAG_KEYID`
9. If the authenticator is a bound authenticator
   1. Store `CallerID`, `AppID`, `TAG_KEYHANDLE`, `TAG_KEYID` and `CurrentTimestamp` in the ASM's database.

      NOTE

      What data an ASM will store at this stage depends on underlying authenticator's architecture. For example some authenticators might store AppID, KeyHandle, KeyID inside their own secure storage. In this case ASM doesn't have to store these data in its database.
10. Create a `RegisterOut` object
    1. Set `RegisterOut.assertionScheme` according to `AuthenticatorInfo.assertionScheme`
    2. Encode the content of `TAG_AUTHENTICATOR_ASSERTION` (e.g. `TAG_UAFV1_REG_ASSERTION`) in base64url format and set as `RegisterOut.assertion`.
    3. Return `RegisterOut` object

3.7 Authenticate Request

Verify the user and return authenticator-generated UAF authentication assertion.

For an Authenticate request, the following **`ASMRequest`** member(s) must have the following value(s). The remaining **`ASMRequest`** members should be omitted:

- `ASMRequest.requestType` must be set to `Authenticate`.
- `ASMRequest.asmVersion` must be set to the desired version.
- `ASMRequest.authenticatorIndex` must be set to the target authenticator index.
- `ASMRequest.args` must be set to an object of type `AuthenticateIn`

For an Authenticate response, the following **`ASMResponse`** member(s) must have the following value(s). The remaining **`ASMResponse`** members should be omitted:

- `ASMResponse.statusCode` must have one of the following values:
  - `UAF_ASM_STATUS_OK`
  - `UAF_ASM_STATUS_ERROR`
  - `UAF_ASM_STATUS_ACCESS_DENIED`
  - `UAF_ASM_STATUS_USER_CANCELLED`
  - `UAF_ASM_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT`
  - `UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY`
  - `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`
  - `UAF_ASM_STATUS_USER_NOT_RESPONSIVE`
  - `UAF_ASM_STATUS_USER_LOCKOUT`
  - `UAF_ASM_STATUS_USER_NOT_ENROLLED`
- `ASMResponse.responseData` must be an object of type `AuthenticateOut`. In the case of an error the values of the fields might be empty (e.g. empty strings).

3.7.1 AuthenticateIn Object

WebIDL

```
dictionary AuthenticateIn {
    required DOMString appID;
    DOMString[]        keyIDs;
    required DOMString finalChallenge;
    Transaction[]      transaction;
};
```

3.7.1.1 Dictionary `AuthenticateIn` Members

`appID` of type required DOMString
: appID string

`keyIDs` of type array of DOMString
: base64url [RFC4648] encoded keyIDs

`finalChallenge` of type required DOMString
: base64url [RFC4648] encoded final challenge

`transaction` of type array of *Transaction*
: An array of transaction data to be confirmed by user. If multiple transactions are provided, then the ASM must select the one that best matches the current display characteristics.

> NOTE
>
> This may, for example, depend on whether user's device is positioned horizontally or vertically at the moment of transaction.

3.7.2 Transaction Object

WebIDL

```
dictionary Transaction {
    required DOMString                    contentType;
    required DOMString                    content;
    DisplayPNGCharacteristicsDescriptor tcDisplayPNGCharacteristics;
};
```

3.7.2.1 Dictionary `Transaction` Members

`contentType` of type required DOMString
: Contains the MIME Content-Type supported by the authenticator according to its metadata statement (see [FIDOMetadataStatement])

`content` of type required DOMString
: Contains the base64url-encoded [RFC4648] transaction content according to the `contentType` to be shown to the user.

`tcDisplayPNGCharacteristics` of type DisplayPNGCharacteristicsDescriptor
: Transaction content PNG characteristics. For the definition of the `DisplayPNGCharacteristicsDescriptor` structure See [FIDOMetadataStatement].

3.7.3 AuthenticateOut Object

WebIDL

```
dictionary AuthenticateOut {
    required DOMString assertion;
    required DOMString assertionScheme;
};
```

3.7.3.1 Dictionary `AuthenticateOut` Members

`assertion` of type required DOMString
: Authenticator UAF authentication assertion.

`assertionScheme` of type required DOMString
: Assertion scheme

3.7.4 Detailed Description for Processing the Authenticate Request

Refer to the [UAFAuthnrCommands] document for more information about the TAGs and structure mentioned in this paragraph.

1. Locate the authenticator using `authenticatorIndex`. If the authenticator cannot be located, then fail with `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`.
2. If no user is enrolled with this authenticator (such as biometric enrollment, PIN setup, etc.), return `UAF_ASM_STATUS_ACCESS_DENIED`
3. The ASM must request the authenticator to verify the user.
   - If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return `UAF_ASM_STATUS_USER_LOCKOUT`.
   - If verification fails, return `UAF_ASM_STATUS_ACCESS_DENIED`

   > NOTE
   >
   > If the authenticator supports `UserVerificationToken` (see [UAFAuthnrCommands]), the ASM must obtain this token in order to later pass to `Sign` command.

4. Construct `KHAccessToken` (see section KHAccessToken for more details)
5. Hash the provided `AuthenticateIn.finalChallenge` using an authenticator-specific hash function (`FinalChallengeHash`).

   The authenticator's preferred hash function information must meet the algorithm defined in the `AuthenticatorInfo.authenticationAlgorithm` field.

6. If this is a Second Factor authenticator and `AuthenticateIn.keyIDs` is empty, then return `UAF_ASM_STATUS_ACCESS_DENIED`
7. If AuthenticateIn.keyIDs is not empty,
   1. If this is a bound authenticator, then look up ASM's database with `AuthenticateIn.appID` and `AuthenticateIn.keyIDs` and obtain the KeyHandles associated with it.
      - Return `UAF_ASM_STATUS_KEY_DISAPPEARED_PERMANENTLY` if the related key disappeared permanently from the authenticator.
      - Return `UAF_ASM_STATUS_ACCESS_DENIED` if no entry has been found.
   2. If this is a roaming authenticator, then treat `AuthenticateIn.keyIDs` as KeyHandles
8. Create `TAG_UAFV1_SIGN_CMD` structure and pass it to the authenticator.
   1. Copy `AuthenticateIn.AppID`, `AuthenticateIn.Transaction.content` (if not empty), `FinalChallengeHash`, `KHAccessToken`, `UserVerificationToken`, `KeyHandles`
      - Depending on AuthenticatorType some arguments may be optional. Refer to [UAFAuthnrCommands] for more information on authenticator types and their required arguments.
      - If multiple transactions are provided, select the one that best matches the current display characteristics.

        > NOTE
        >
        > This may, for example, depend on whether user's device is positioned horizontally or vertically at the moment of transaction.

      - Decode the base64url encoded `AuthenticateIn.Transaction.content` before passing it to the authenticator
9. Invoke the command and receive the response. If the authenticator returns an error, handle that error appropriately. If the connection to the authenticator gets lost and cannot be restored, return `UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED`. If the operation finally fails, map the authenticator error code to the appropriate ASM error code (see section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codesfor details).
10. Parse `TAG_UAFV1_SIGN_CMD_RESP`
    - If it's a first-factor authenticator and the response includes `TAG_USERNAME_AND_KEYHANDLE`, then
      1. Extract usernames from `TAG_USERNAME_AND_KEYHANDLE` fields
      2. If two or more equal usernames are found, then choose the one which has registered most recently

         > NOTE
         >
         > After this step, a first-factor bound authenticator which stores KeyHandles inside the ASM's database may delete the redundant KeyHandles from the ASM's database. This avoids having unusable (old) private key in the authenticator which (surprisingly) might become active after deregistering the newly generated one.

      3. Show remaining distinct usernames and ask the user to choose a single username
      4. Set `TAG_UAFV1_SIGN_CMD.KeyHandles` to the single KeyHandle associated with the selected username.
      5. Go to step #8 and send a new `TAG_UAFV1_SIGN_CMD command`
11. Create the `AuthenticateOut` object
    1. Set `AuthenticateOut.assertionScheme` as `AuthenticatorInfo.assertionScheme`
    2. Encode the content of `TAG_AUTHENTICATOR_ASSERTION` (e.g. `TAG_UAFV1_AUTH_ASSERTION`) in base64url format and set as `AuthenticateOut.assertion`
    3. Return the `AuthenticateOut` object

> NOTE
>
> Some authenticators might support "Transaction Confirmation Display" functionality not inside the authenticator but within the boundaries of the ASM. Typically these are software based Transaction Confirmation Displays. When processing the `Sign` command with a given transaction such ASM should show transaction content in its own UI and after user confirms it -- pass the content to authenticator so that the authenticator includes it in the final assertion.
>
> See [FIDORegistry] for flags describing Transaction Confirmation Display type.

The authenticator metadata statement must truly indicate the type of transaction confirmation display implementation. Typically the "Transaction Confirmation Display" flag will be set to `TRANSACTION_CONFIRMATION_DISPLAY_ANY` (bitwise) or `TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE`.

3.8 Deregister Request

Delete registered UAF record from the authenticator.

For a Deregister request, the following **`ASMRequest`** member(s) must have the following value(s). The remaining **`ASMRequest`** members should be omitted:

- `ASMRequest.requestType` must be set to `Deregister`

- ASMRequest.asmVersion must be set to the desired version
- ASMRequest.authenticatorIndex must be set to the target authenticator index
- ASMRequest.args must be set to an object of type DeregisterIn

For a Deregister response, the following **ASMResponse** member(s) must have the following value(s). The remaining **ASMResponse** members should be omitted:

- ASMResponse.statusCode must have one of the following values:
  - UAF_ASM_STATUS_OK
  - UAF_ASM_STATUS_ERROR
  - UAF_ASM_STATUS_ACCESS_DENIED
  - UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED

3.8.1 DeregisterIn Object

WebIDL

```
dictionary DeregisterIn {
    required DOMString appID;
    required DOMString keyID;
};
```

*3.8.1.1 Dictionary **DeregisterIn** Members*

appID of type required DOMString
FIDO Server Application Identity

keyID of type required DOMString
Base64url-encoded [RFC4648] key identifier of the authenticator to be de-registered. The keyID can be an empty string. In this case all keyIDs related to this appID must be deregistered.

3.8.2 Detailed Description for Processing the Deregister Request

Refer to [UAFAuthnrCommands] for more information about the TAGs and structures mentioned in this paragraph.

1. Locate the authenticator using authenticatorIndex
2. Construct KHAccessToken (see section KHAccessToken for more details).
3. If this is a bound authenticator, then
   - If the value of DeregisterIn.keyID is an empty string, then lookup all pairs of this appID and any keyID mapped to this authenticatorIndex and delete them. Go to step 4.
   - Otherwise, lookup the authenticator related data in the ASM database and delete the record associated with DeregisterIn.appID and DeregisterIn.keyID. Go to step 4.
4. Create the TAG_UAFV1_DEREGISTER_CMD structure, copy KHAccessToken and DeregisterIn.keyID and pass it to the authenticator.

> NOTE
>
> In the case of roaming authenticators, the keyID passed to the authenticator might be an empty string. The authenticator is supposed to deregister all keys related to this appID in this case.

5. Invoke the command and receive the response. If the authenticator returns an error, handle that error appropriately. If the connection to the authenticator gets lost and cannot be restored, return UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED. If the operation finally fails, map the authenticator error code to the appropriate ASM error code (see section 3.2.2 Mapping Authenticator Status Codes to ASM Status Codes for details). Return proper ASMResponse.

3.9 GetRegistrations Request

Return all registrations made for the calling FIDO UAF Client.

For a GetRegistrations request, the following **ASMRequest** member(s) must have the following value(s). The remaining **ASMRequest** members should be omitted:

- ASMRequest.requestType must be set to GetRegistrations
- ASMRequest.asmVersion must be set to the desired version
- ASMRequest.authenticatorIndex must be set to corresponding ID

For a GetRegistrations response, the following **ASMResponse** member(s) must have the following value(s). The remaining **ASMResponse** members should be omitted:

- ASMResponse.statusCode must have one of the following values:
  - UAF_ASM_STATUS_OK
  - UAF_ASM_STATUS_ERROR
  - UAF_ASM_STATUS_AUTHENTICATOR_DISCONNECTED
- The ASMResponse.responseData must be an object of type GetRegistrationsOut. In the case of an error the values of the fields might be empty (e.g. empty strings).

3.9.1 GetRegistrationsOut Object

WebIDL

```
dictionary GetRegistrationsOut {
    required AppRegistration[] appRegs;
};
```

*3.9.1.1 Dictionary **GetRegistrationsOut** Members*

appRegs of type array of required AppRegistration
List of registrations associated with an appID (see AppRegistration below). may be an empty list.

3.9.2 AppRegistration Object

WebIDL

```
dictionary AppRegistration {
    required DOMString   appID;
    required DOMString[] keyIDs;
};
```

3.9.2.1 Dictionary AppRegistration Members

appID of type required DOMString
FIDO Server Application Identity.

keyIDs of type array of required DOMString
List of key identifiers associated with the appID

3.9.3 Detailed Description for Processing the GetRegistrations Request

1. Locate the authenticator using authenticatorIndex
2. If this is bound authenticator, then
   - Lookup the registrations associated with CallerID and AppID in the ASM database and construct a list of AppRegistration objects

> NOTE
>
> Some ASMs might not store this information inside their own database. Instead it might have been stored inside the authenticator's secure storage area. In this case the ASM must send a proprietary command to obtain the necessary data.

3. Create GetRegistrationsOut object and return

3.10 OpenSettings Request

Display the authenticator-specific settings interface. If the authenticator has its own built-in user interface, then the ASM must invoke TAG_UAFV1_OPEN_SETTINGS_CMD to display it.

For an OpenSettings request, the following **ASMRequest** member(s) must have the following value(s). The remaining **ASMRequest** members should be omitted:

- ASMRequest.requestType must be set to OpenSettings
- ASMRequest.asmVersion must be set to the desired version
- ASMRequest.authenticatorIndex must be set to the target authenticator index

For an OpenSettings response, the following **ASMResponse** member(s) must have the following value(s). The remaining **ASMResponse** members should be omitted:

- ASMResponse.statusCode must have one of the following values:
  - UAF_ASM_STATUS_OK

4. Using ASM API

This section is non-normative.

In a typical implementation, the FIDO UAF Client will call GetInfo during initialization and obtain information about the authenticators. Once the information is obtained it will typically be used during FIDO UAF message processing to find a match for given FIDO UAF policy. Once a match is found the FIDO UAF Client will send the appropriate request (Register/Authenticate/Deregister...) to this ASM.

The FIDO UAF Client may use the information obtained from a GetInfo response to display relevant information about an authenticator to the user.

5. Using the ASM API on various platforms

This section is normative.

5.1 Android ASM Intent API

On Android systems FIDO UAF ASMs may be implemented as a separate APK-packaged application.

The FIDO UAF Client invokes ASM operations via Android Intents. All interactions between the FIDO UAF Client and an ASM on Android takes place through the following intent identifier:

```
org.fidoalliance.intent.FIDO_OPERATION
```

To carry messages described in this document, an intent must also have its type attribute set to application/fido.uaf_asm+json.

ASMs must register that intent in their manifest file and implement a handler for it.

FIDO UAF Clients must append an extra, message, containing a String representation of a **ASMRequest**, before invoking the intent.

FIDO UAF Clients must invoke ASMs by calling startActivityForResult()

FIDO UAF Clients should assume that ASMs will display an interface to the user in order to handle this intent, e.g. prompting the user to complete the verification ceremony. However, the ASM should not display any user interface when processing a GetInfo request.

After processing is complete the ASM will return the response intent as an argument to onActivityResult(). The response intent will have an extra, message, containing a String representation of a **ASMResponse**.

5.1.1 Discovering ASMs

FIDO UAF Clients can discover the ASMs available on the system by using `PackageManager.queryIntentActivities(Intent intent, int flags)` with the FIDO Intent described above to see if any activities are available.

A typical FIDO UAF Client will enumerate all ASM applications using this function and will invoke the `GetInfo` operation for each one discovered.

5.1.2 Alternate Android AIDL Service ASM Implementation

The Android Intent API can also be implemented using Android AIDL services as an alternative transport mechanism to Android Intents. Please see Android Intent API section [UAFAppAPIAndTransport] for differences between the Android AIDL service and Android Intent implementation.

5.2 Windows ASM API

On Windows, an ASM is implemented in the form of a Dynamic Link Library (DLL). The following is an example `asmplugin.h` header file defining a Windows ASM API:

EXAMPLE 1

```
/*! @file asm.h
*/

#ifndef __ASMH_
#define __ASMH_
#ifdef _WIN32
#define ASM_API __declspec(dllexport)
#endif

#ifdef _WIN32
#pragma warning ( disable : 4251 )
#endif

#define ASM_FUNC extern "C" ASM_API
#define ASM_NULL 0

/*! \brief Error codes returned by ASM Plugin API.
*  Authenticator specific error codes are returned in JSON form.
*  See JSON schemas for more details.
*/

enum asmResult_t
{
  Success = 0, /**< Success */
  Failure /**< Generic failure */
};

/*! \brief Generic structure containing JSON string in UTF-8
*  format.
*  This structure is used throughout functions to pass and receives
*  JSON data.
*/

struct asmJSONData_t
{
  int length; /**< JSON data length */
  char pData; /*< JSON data */
};

/*! \brief Enumeration event types for authenticators.
These events will be fired when an authenticator becomes
  available (plugged) or unavailable (unplugged).
*/

enum asmEnumerationType_t
{
  Plugged = 0, /**< Indicates that authenticator Plugged to system */
  Unplugged /**< Indicates that authenticator Unplugged from system */
};

namespace ASM
{
  /*! \brief Callback listener.
  FIDO UAF Client must pass an object implementating this interface to
  Authenticator::Process function. This interface is used to provide
  ASM JSON based response data.*/
  class ICallback
  {
    public
      virtual ~ICallback() {}
      /**
      This function is called when ASM's response is ready.
      *
      @param response JSON based event data
      @param exchangeData must be provided by ASM if it needs some
      data back right after calling the callback function.
      The lifecycle of this parameter must be managed by ASM. ASM must
      allocate enough memory for getting the data back.
      */

      virtual void Callback(const asmJSONData_t &response,
      asmJSONData_t &exchangeData) = 0;
  };

  /*! \brief Authenticator Enumerator.
  FIDO UAF Client must provide an object implementing this
  interface. It will be invoked when a new authenticator is plugged or
  when an authenticator has been unplugged. */

  class IEnumerator
  {
    public
      virtual ~IEnumerator() {}
      /**
        This function is called when an authenticator is plugged or
     unplugged.
      * @param eventType event type (plugged/unplugged)
        @param AuthenticatorInfo JSON based GetInfoResponse object
      */

      virtual void Notify(const asmEnumerationType_t eventType, const
      asmJSONData_t &AuthenticatorInfo) = 0;
  };
}

/**
Initializes ASM plugin. This is the first function to be
     called.
*
@param pEnumerationListener caller provided Enumerator
*/

ASM_FUNC asmResult_t asmInit(ASM::IEnumerator
```

```
    *pEnumerationListener);
/**
Process given JSON request and returns JSON response.
*
If the caller wants to execute a function defined in ASM JSON
     schema then this is the function that must be called.
*
@param pInData input JSON data
@param pListener event listener for receiving events from ASM
*/
ASM_FUNC asmResult_t asmProcess(const asmJSONData_t *pInData,
    ASM::ICallback *pListener);
/**
Unitializes ASM plugin.
*
*/
ASM_FUNC asmResult_t asmUninit();
#endif // __ASMPLUGINH_
```

A Windows-based FIDO UAF Client must look for ASM DLLs in the following registry paths:

`HKCU\Software\FIDO\UAF\ASM`

`HKLM\Software\FIDO\UAF\ASM`

The FIDO UAF Client iterates over all keys under this path and looks for "path" field:

`[HK**\Software\FIDO\UAF\ASM\<exampleASMName>]`

`"path"="<ABSOLUTE_PATH_TO_ASM>.dll"`

`path` must point to the absolute location of the ASM DLL.

6. Security and Privacy Guidelines

This section is normative.

ASM developers must carefully protect the FIDO UAF data they are working with. ASMs must follow these security guidelines:

- ASMs must implement a mechanism for isolating UAF credentials registered by two different FIDO UAF Clients from one another. One FIDO UAF Client must not have access to FIDO UAF credentials that have been registered via a different FIDO UAF Client. This prevents malware from exercising credentials associated with a legitimate FIDO Client.

  > NOTE
  >
  > ASMs must properly protect their sensitive data against malware using platform-provided isolation capabilities in order to follow the assumptions made in [FIDOSecRef]. Malware with root access to the system or direct physical attack on the device are out of scope for this requirement.

  > NOTE
  >
  > The following are examples for achieving this:
  >
  > - If an ASM is bundled with a FIDO UAF Client, this isolation mechanism is already built-in.
  > - If the ASM and FIDO UAF Client are implemented by the same vendor, the vendor may implement proprietary mechanisms to bind its ASM exclusively to its own FIDO UAF Client.
  > - On some platforms ASMs and the FIDO UAF Clients may be assigned with a special privilege or permissions which regular applications don't have. ASMs built for such platforms may avoid supporting isolation of UAF credentials per FIDO UAF Clients since all FIDO UAF Clients will be considered equally trusted.

- An ASM designed specifically for bound authenticators must ensure that FIDO UAF credentials registered with one ASM cannot be accessed by another ASM. This is to prevent an application pretending to be an ASM from exercising legitimate UAF credentials.
  - Using a KHAccessToken offers such a mechanism.
- An ASMs must implement platform-provided security best practices for protecting UAF related stored data.
- ASMs must not store any sensitive FIDO UAF data in its local storage, except the following:
  - `CallerID`, `ASMToken`, `PersonaID`, `KeyID`, `KeyHandle`, `AppID`

  > NOTE
  >
  > An ASM, for example, must never store a username provided by a FIDO Server in its local storage in a form other than being decryptable exclusively by the authenticator.

- ASMs should ensure that applications cannot use silent authenticators for tracking purposes. ASMs implementing support for a silent authenticator must show, during every registration, a user interface which explains what a silent authenticator is, asking for the users consent for the registration. Also, it is recommended that ASMs designed to support roaming silent authenticators 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.

6.1 KHAccessToken

`KHAccessToken` is an access control mechanism for protecting an authenticator's FIDO UAF credentials from unauthorized use. It is created by the ASM by mixing various sources of information together. Typically, a `KHAccessToken` contains the following four data items in it: `AppID`, `PersonaID`, `ASMToken` and

CallerID.

AppID is provided by the FIDO Server and is contained in every FIDO UAF message.

PersonaID is obtained by the ASM from the operational environment. Typically a different PersonaID is assigned to every operating system user account.

ASMToken is a randomly generated secret which is maintained and protected by the ASM.

> NOTE
>
> In a typical implementation an ASM will randomly generate an ASMToken when it is launched the first time and will maintain this secret until the ASM is uninstalled.

CallerID is the ID the platform has assigned to the calling FIDO UAF Client (e.g. "bundle ID" for iOS). On different platforms the *CallerID* can be obtained differently.

> NOTE
>
> For example on Android platform ASM can use the hash of the caller's apk-signing-cert.

The ASM uses the KHAccessToken to establish a link between the ASM and the key handle that is created by authenticator on behalf of this ASM.

The ASM provides the KHAccessToken to the authenticator with every command which works with key handles.

> NOTE
>
> The following example describes how the ASM constructs and uses KHAccessToken.
>
> - During a Register request
>   - Set KHAccessToken to a secret value only known to the ASM. This value will always be the same for this ASM.
>   - Append AppID
>   - KHAccessToken = AppID
>   - If a bound authenticator, append ASMToken, PersonaID and CallerID
>   - KHAccessToken |= ASMToken | PersonaID | CallerID
>   - Hash KHAccessToken
>   - Hash KHAccessToken using the authenticator's hashing algorithm. The reason of using authenticator specific hash function is to make sure of interoperability between ASMs. If interoperability is not required, an ASM can use any other secure hash function it wants.
>   - KHAccessToken=hash(KHAccessToken)
>   - Provide KHAccessToken to the authenticator
>   - The authenticator puts the KHAccessToken into RawKeyHandle (see [UAFAuthnrCommands] for more details)
> - During other commands which require KHAccessToken as input argument
>   - The ASM computes KHAccessToken the same way as during the Register request and provides it to the authenticator along with other arguments.
>   - The authenticator unwraps the provided key handle(s) and proceeds with the command only if RawKeyHandle.KHAccessToken is equal to the provided KHAccessToken.

Bound authenticators must support a mechanism for binding generated key handles to ASMs. The binding mechanism must have at least the same security characteristics as mechanism for protcting KHAccessToken described above. As a consequence it is recommended to securely derive KHAccessToken from AppID, ASMToken, PersonaID and the CallerID.

> NOTE
>
> It is recommended for roaming authenticators that the KHAccessToken contains only the AppID since otherwise users won't be able to use them on different machines (PersonaID, ASMToken and CallerID are platform specific). If the authenticator vendor decides to do that in order to address a specific use case, however, it is allowed.
>
> Including PersonaID in the KHAccessToken is optional for all types of authenticators. However an authenticator designed for multi-user systems will likely have to support it.

If an ASM for roaming authenticators doesn't use a KHAccessToken which is different for each AppID, the ASM must include the AppID in the command for a deregister request containing an empty KeyID.

6.2 Access Control for ASM APIs

The following table summarizes the access control requirements for each API call.

ASMs must implement the access control requirements defined below. ASM vendorsmay implement additional security mechanisms.

Terms used in the table:

- NoAuth -- no access control
- CallerID -- FIDO UAF Client's platform-assigned ID is verified
- UserVerify -- user must be explicitly verified
- KeyIDList -- must be known to the caller

Commands	First-factor bound authenticator	Second-factor bound authenticator	First-factor roaming authenticator	Second-factor roaming authenticator
GetInfo	NoAuth	NoAuth	NoAuth	NoAuth
OpenSettings	NoAuth	NoAuth	NoAuth	NoAuth
Register	UserVerify	UserVerify	UserVerify	UserVerify

Authenticate	UserVerify AppID CallerID PersonalID	UserVerify AppID KeyIDList CallerID PersonalID	UserVerify AppID	UserVerify AppID KeyIDList
GetRegistrations*	CallerID PersonalID	CallerID PersonalID	X	X
Deregister	AppID KeyID PersonalID CallerID	AppID KeyID PersonalID CallerID	AppID KeyID	AppID KeyID

A. References

A.1 Normative references

[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[FIDOMetadataStatement]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO Metadata Statements v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.pdf

[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.pdf

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

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

[UAFAuthnrCommands]
D. Baghdasaryan, J. Kemp, R. Lindemann, R. Sasson, B. Hill, *FIDO UAF Authenticator Commands v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.pdf

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges, *FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.pdf

[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[ECMA-404]
The JSON Data Interchange Format. 1 October 2013. Standard. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.pdf

[RFC2397]
L. Masinter. *The "data" URL scheme*. August 1998. Proposed Standard. URL: https://tools.ietf.org/html/rfc2397

[UAFAppAPIAndTransport]
B. Hill, D. Baghdasaryan, B. Blanke, *FIDO UAF Application API and Transport Binding Specification*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.pdf

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Working Draft. URL: https://www.w3.org/TR/WebIDL-1/

IMPLEMENTATION DRAFT

FIDO UAF Authenticator Commands

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-authnr-cmds-v1.0-ps-20141208.html
Editors:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
John Kemp, FIDO Alliance
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Roni Sasson, Discretix
Brad Hill, PayPal, Inc.
Jeff Hodges, PayPal, Inc.
Ka Yang, Nok Nok Labs, Inc.

Abstract

UAF Authenticators may take different forms. Implementations may range from a secure application running inside tamper-resistant hardware to software-only solutions on consumer devices.

This document defines normative aspects of UAF Authenticators and offers security and implementation guidelines for authenticator implementors.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

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

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

Unless otherwise specified all data described in this document must be encoded in **little-endian** format.

All TLV structures can be parsed using a "recursive-descent" parsing approach. In some cases multiple occurrences of a single tag may be allowed within a structure, in which case all values must be preserved.

All fields in TLV structures are *mandatory*, unless explicitly mentioned as otherwise.

1.1 Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

This document specifies low-level functionality which UAF Authenticators should implement in order to support the UAF protocol. It has the following goals:

- Define normative aspects of UAF Authenticator implementations
- Define a set of commands implementing UAF functionality that may be implemented by different types of authenticators
- Define `UAFV1TLV` assertion scheme-specific structures which will be parsed by a FIDO Server

NOTE

The UAF Protocol supports various assertion schemes. Commands and structures defined in this document assume that an authenticator supports the `UAFV1TLV` assertion scheme. Authenticators implementing a different assertion scheme do not have to follow requirements specified in this document.

The overall architecture of the UAF protocol and its various operations is described in [UAFProtocol]. The following simplified architecture diagram illustrates the interactions and actors this document is concerned with:

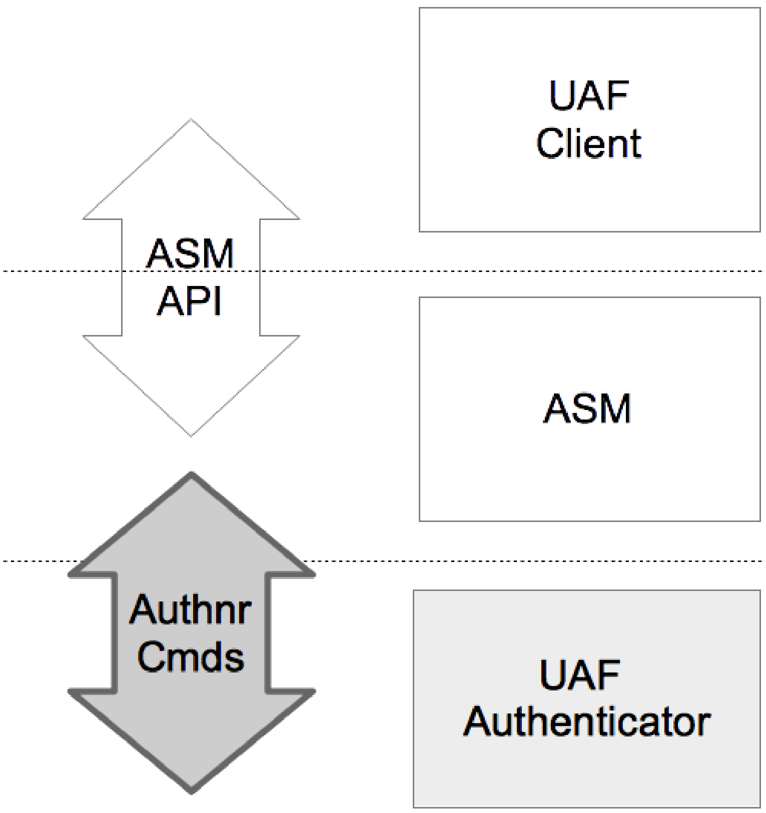


Fig. 1 UAF Authenticator Commands

3. UAF Authenticator

This section is non-normative.

The UAF Authenticator is an authentication component that meets the UAF protocol requirements as described in [UAFProtocol]. The main functions to be provided by UAF Authenticators are:

1. [Mandatory] Verifying the user with the verification mechanism built into the authenticator. The verification technology can vary, from biometric verification to simply verifying physical presence, or no user verification at all (the so-called *Silent Authenticator*).
2. [Mandatory] Performing the cryptographic operations defined in [UAFProtocol]
3. [Mandatory] Creating data structures that can be parsed by FIDO Server.
4. [Mandatory] Attesting itself to the FIDO Server if there is a built-in support for attestation
5. [Optional] Displaying the transaction content to the user using the transaction confirmation display

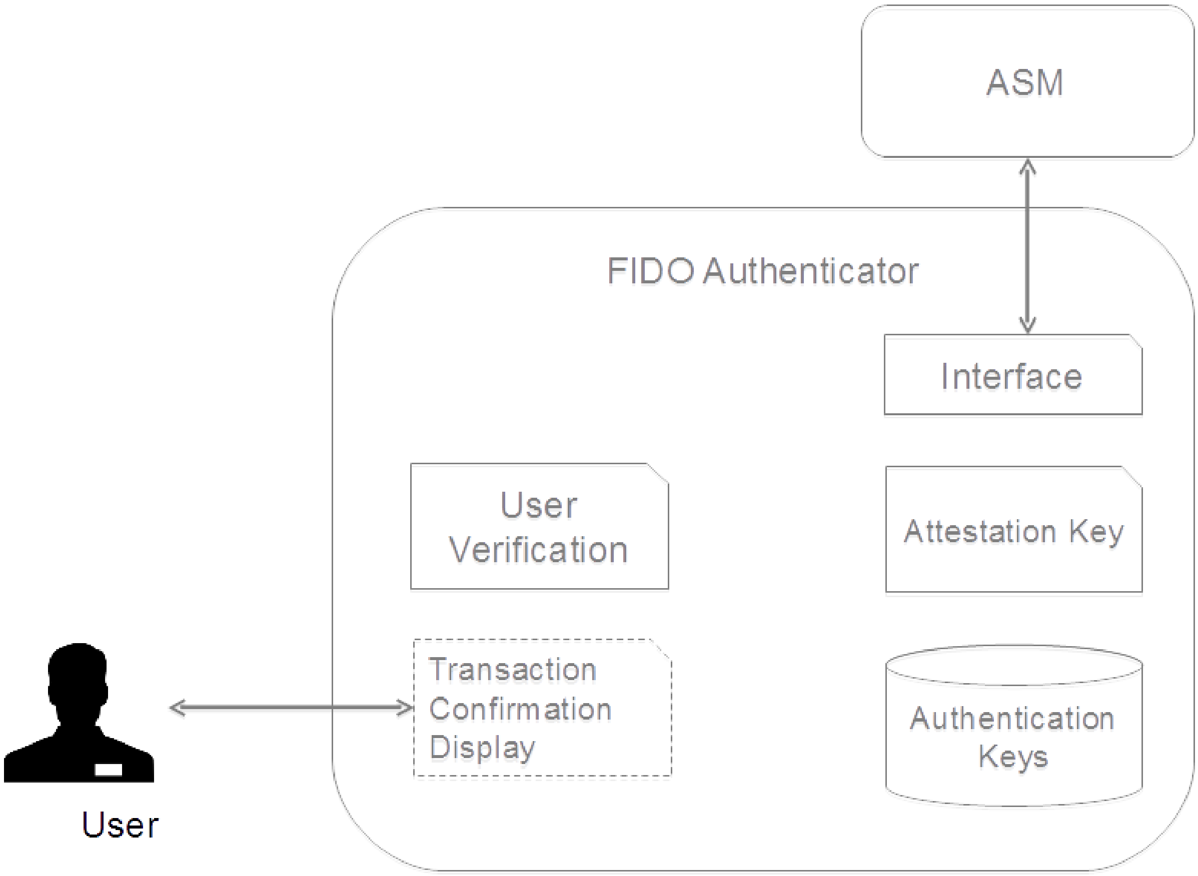


Fig. 2 FIDO Authenticator Logical Sub-Components

Some examples of UAF Authenticators:

- A fingerprint sensor built into a mobile device
- PIN authenticator implemented inside a *secure element*
- A mobile phone acting as an authenticator to a different device
- A USB token with built-in user presence verification
- A voice or face verification technology built into a device

3.1 Types of Authenticators

There are four types of authenticators defined in this document. These definitions are not normative (unless otherwise stated) and are provided merely for simplifying some of the descriptions.

> NOTE
>
> The following is the rationale for considering only these 4 types of authenticators:
>
> - Bound authenticators are typically embedded into a user's computing device and thus can utilize the host's storage for their needs. It makes more sense from an economic perspective to utilize the host's storage rather than have embedded storage. Trusted Execution Environments (TEE), Secure Elements and Trusted Platform Modules (TPM) are typically designed in this manner.
> - First-factor roaming authenticators must have an internal storage for key handles.
> - Second-factor roaming authenticators can store their key handles on an associated server, in order to avoid the need for internal storage.
> - Defining such constraints makes the specification simpler and clearer for defining the mainstream use-cases.

Vendors, however, are not limited to these constraints. For example a bound authenticator which has internal storage for storing key handles is possible. Vendors are free to design and implement such authenticators as long as their design follows the normative requirements described in this document.

- **First-factor Bound Authenticator**
  - These authenticators have an internal matcher. The matcher is able to verify an already enrolled user. If there is more than one user enrolled - the matcher can also identify a user.
  - There is a logical binding between this authenticator and the device it is attached to (the binding is expressed through a concept called KeyHandleAccessToken). This authenticator cannot be bound with more than one device.
  - These authenticators do not store key handles in their own internal storage. They always return the key handle to the ASM and the latter stores it in its local database.
  - Authenticators of this type may also work as a second factor.
  - Examples
    - A fingerprint sensor built into a laptop, phone or tablet
    - Embedded secure element in a mobile device
    - Voice verification built into a device
- **Second-factor (2ndF) Bound Authenticator**
  - This type of authenticator is similar to first-factor bound authenticators, except that it can operate only as the second-factor in a multi-factor authentication
  - Examples
    - USB dongle with a built-in capacitive touch device for verifying user presence
    - A "Trustlet" application running on the trusted execution environment of a mobile phone, and leveraging a secure keyboard to verify user presence
- **First Factor (1stF) Roaming Authenticator**
  - These authenticators are not bound to any device. User can use them with any number of devices.
  - It is assumed that these authenticators have an internal matcher. The matcher is able to verify an already enrolled user. If there is more than one user enrolled - the matcher can also identify a user.
  - It is assumed that these authenticators are designed to store key handles in their own internal secure storage and not expose externally.
  - These authenticators may also work as a second factor.
  - Examples
    - A Bluetooth LE based hardware token with built-in fingerprint sensor
    - PIN protected USB hardware token
    - A first-factor bound authenticator acting as a roaming authenticator for a different device on the user's behalf
- **Second-factor Roaming Authenticator**
  - These authenticators are not bound to any device. A user may use them with any number of devices.
  - These authenticators may have an internal matcher. The matcher is able to verify an already enrolled user. If there is more than one user enrolled then the matcher can also identify a particular specific user.
  - It is assumed that these authenticators do not store key handles in their own internal storage. Instead they push key handles to the FIDO Server and receive them back during the authentication operation.
  - These authenticators can only work as second factors.
  - Examples
    - USB dongle with a built-in capacitive touch device for verifying user presence
    - A "Trustlet" application running on the trusted execution environment of a mobile phone, and leveraging a secure keyboard to verify user presence

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

NORMATIVE

In some deployments, the combination of ASM and a bound authenticator can act as a roaming authenticator (for example when an ASM with an embedded authenticator on a mobile device acts as a roaming authenticator for another device). When this happens such an authenticator must follow the requirements applying to bound authenticators within the boundary of the system the authenticator is bound to, and follow the requirements that apply to roaming authenticators in any other system it connects to externally.

Conforming authenticators must implement at least one attestation type defined in [UAFRegistry], as well as one authentication algorithm and one key format listed in [FIDORegistry].

NOTE

As stated above, the bound authenticator does not store key handles and roaming authenticators do store them. In the example above the ASM would store the key handles of the bound authenticator and hence meets these assumptions.

4. Tags

This section is normative.

In this document UAF Authenticators use "Tag-Length-Value" (TLV) format to communicate with the outside world. All requests and response data must be encoded as TLVs.

Commands and existing predefined TLV tags can be extended by appending other TLV tags (custom or predefined).

Refer to [UAFRegistry] for information about predefined TLV tags.

TLV formatted data has the following simple structure:

2 bytes	2 bytes	Length bytes
Tag	Length in bytes	Data

All lengths are in bytes. e.g. a UINT32[4] will have length 16.

Although 2 bytes are allotted for the tag, only the first 14 bits (values up to 0x3FFF) should be used to accommodate the limitations of some hardware platforms.

Arrays are implicit. The description of some structures indicates where multiple values are permitted, and in these cases, if same tag appears more than once, all values are signifanct and should be treated as an array.

For convenience in decoding TLV-formatted messages, all composite tags - those with values that must be parsed by recursive descent - have the 13th bit (0x1000) set.

A tag that has the 14th bit (0x2000) set indicates that it is critical and a receiver must abort processing the entire message if it cannot process that tag.

Since UAF Authenticators may have extremely constrained processing environments, an ASM must follow a normative ordering of structures when sending commands.

It is assumed that ASM and Server have sufficient resources to handle parsing tags in any order so structures send from authenticator may use tags in any order.

4.1 Command Tags

Name	Value	Description
TAG_UAFV1_GETINFO_CMD	0x3401	Tag for GetInfo command.
TAG_UAFV1_GETINFO_CMD_RESPONSE	0x3601	Tag for GetInfo command response.
TAG_UAFV1_REGISTER_CMD	0x3402	Tag for Register command.
TAG_UAFV1_REGISTER_CMD_RESPONSE	0x3602	Tag for Register command response.
TAG_UAFV1_SIGN_CMD	0x3403	Tag for Sign command.
TAG_UAFV1_SIGN_CMD_RESPONSE	0x3603	Tag for Sign command response.
TAG_UAFV1_DEREGISTER_CMD	0x3404	Tag for Deregister command.
TAG_UAFV1_DEREGISTER_CMD_RESPONSE	0x3604	Tag for Deregister command response.
TAG_UAFV1_OPEN_SETTINGS_CMD	0x3406	Tag for OpenSettings command.
TAG_UAFV1_OPEN_SETTINGS_CMD_RESPONSE	0x3606	Tag for OpenSettings command response.

Table 4.1.1: UAF Authenticator Command TLV tags (0x3400 - 0x34FF, 0x3600-0x36FF)

4.2 Tags used only in Authenticator Commands

Name	Value	Description
TAG_KEYHANDLE	0x2801	Represents key handle. Refer to [FIDOGlossary] for more information about key handle.
TAG_USERNAME_AND_KEYHANDLE	0x3802	Represents an associated Username and key handle. This is a composite tag that contains a TAG_USERNAME and TAG_KEYHANDLE that identify a registration valid oin the authenticator. Refer to [FIDOGlossary] for more information about username.

Name	Value	Description
TAG_USERVERIFY_TOKEN	0x2803	Represents a User Verification Token. Refer to [FIDOGlossary] for more information about user verification tokens.
TAG_APPID	0x2804	A full AppID as a UINT8[] encoding of a UTF-8 string. Refer to [FIDOGlossary] for more information about AppID.
TAG_KEYHANDLE_ACCESS_TOKEN	0x2805	Represents a key handle Access Token.
TAG_USERNAME	0x2806	A Username as a UINT8[] encoding of a UTF-8 string.
TAG_ATTESTATION_TYPE	0x2807	Represents an Attestation Type.
TAG_STATUS_CODE	0x2808	Represents a Status Code.
TAG_AUTHENTICATOR_METADATA	0x2809	Represents a more detailed set of authenticator information.
TAG_ASSERTION_SCHEME	0x280A	A UINT8[] containing the UTF8-encoded Assertion Scheme as defined in [UAFRegistry]. ("UAFV1TLV")
TAG_TC_DISPLAY_PNG_CHARACTERISTICS	0x280B	If an authenticator contains a PNG-capable transaction confirmation display that is not implemented by a higher-level layer, this tag is describing this display. See [FIDOMetadataStatement] for additional information on the format of this field.
TAG_TC_DISPLAY_CONTENT_TYPE	0x280C	A UINT8[] containing the UTF-8-encoded transaction display content type as defined in [FIDOMetadataStatement]. ("image/png")
TAG_AUTHENTICATOR_INDEX	0x280D	Authenticator Index
TAG_API_VERSION	0x280E	API Version
TAG_AUTHENTICATOR_ASSERTION	0x280F	The content of this TLV tag is an assertion generated by the authenticator. Since authenticators may generate assertions in different formats - the content format may vary from authenticator to authenticator.
TAG_TRANSACTION_CONTENT	0x2810	Represents transaction content sent to the authenticator.
TAG_AUTHENTICATOR_INFO	0x3811	Includes detailed information about authenticator's capabilities.
TAG_SUPPORTED_EXTENSION_ID	0x2812	Represents extension ID supported by authenticator.
TAG_TRANSACTIONCONFIRMATION_TOKEN	0x2813	Represents a token for transaction confirmation. It might be returned by the authenticator to the ASM and given back to the authenticator at a later stage. The meaning of it is similar to TAG_USERVERIFY_TOKEN, except that it is used for the user's approval of a displayed transaction text.

Table 4.2.1: Non-Command Tags (0x2800 - 0x28FF, 0x3800 - 0x38FF)

4.3 Tags used in UAF Protocol

Name	Value	Description
TAG_UAFV1_REG_ASSERTION	0x3E01	Authenticator response to Register command.
TAG_UAFV1_AUTH_ASSERTION	0x3E02	Authenticator response to Sign command.
TAG_UAFV1_KRD	0x3E03	Key Registration Data
TAG_UAFV1_SIGNED_DATA	0x3E04	Data signed by authenticator with the UAuth.priv key
TAG_ATTESTATION_CERT	0x2E05	Each entry contains a single X.509 DER-encoded [ITU-X690-2008] certificate. Multiple occurrences are allowed and form the attestation certificate chain. Multiple occurrences must be ordered. The attestation certificate itself must occur first. Each subsequent occurrence (if exists) must be the issuing certificate of the previous occurrence.
TAG_SIGNATURE	0x2E06	A cryptographic signature
TAG_ATTESTATION_BASIC_FULL	0x3E07	Full Basic Attestation as defined in [UAFProtocol]

Name	Value	Description
TAG_ATTESTATION_BASIC_SURROGATE	0x3E08	Surrogate Basic Attestation as defined in [UAFProtocol]
TAG_ATTESTATION_ECDAA	0x3E09	Elliptic curve based direct anonymous attestation as defined in [UAFProtocol]. In this case the signature in TAG_SIGNATURE is a ECDAA signature as specified in [FIDOEcdaaAlgorithm].
TAG_KEYID	0x2E09	Represents a KeyID.
TAG_FINAL_CHALLENGE_HASH	0x2E0A	Represents a Final Challenge Hash. Refer to [UAFProtocol] for more information about the Final Challenge.
TAG_AAID	0x2E0B	Represents an authenticator Attestation ID. Refer to [UAFProtocol] for more information about the AAID.
TAG_PUB_KEY	0x2E0C	Represents a Public Key.
TAG_COUNTERS	0x2E0D	Represents a use counters for the authenticator.
TAG_ASSERTION_INFO	0x2E0E	Represents assertion information necessary for message processing.
TAG_AUTHENTICATOR_NONCE	0x2E0F	Represents a nonce value generated by the authenticator.
TAG_TRANSACTION_CONTENT_HASH	0x2E10	Represents a hash of transaction content.
TAG_EXTENSION	0x3E11, 0x3E12	This is a composite tag indicating that the content is an extension. If the tag is 0x3E11 - it's a critical extension and if the recipient does not understand the contents of this tag, it must abort processing of the entire message. This tag has two embedded tags - TAG_EXTENSION_ID and TAG_EXTENSION_DATA. For more information about UAF extensions refer to [UAFProtocol] NOTE This tag can be appended to any command and response. Using tag 0x3E11 (as opposed to tag 0x3E12) has the same meaning as the flag `fail_if_unknown` in [UAFProtocol].
TAG_EXTENSION_ID	0x2E13	Represents extension ID. Content of this tag is a UINT8[] encoding of a UTF-8 string.
TAG_EXTENSION_DATA	0x2E14	Represents extension data. Content of this tag is a UINT8[] byte array.

Table 4.3.1: Tags used in the UAF Protocol (0x2E00 - 0x2EFF, 0x3E00 - 0x3EFF). Normatively defined in [UAFRegistry]

4.4 Status Codes

Name	Value	Description
UAF_CMD_STATUS_OK	0x00	Success.
UAF_CMD_STATUS_ERR_UNKNOWN	0x01	An unknown error.
UAF_CMD_STATUS_ACCESS_DENIED	0x02	Access to this operation is denied.
UAF_CMD_STATUS_USER_NOT_ENROLLED	0x03	User is not enrolled with the authenticator and the authenticator cannot automatically trigger enrollment.
UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT	0x04	Transaction content cannot be rendered.
UAF_CMD_STATUS_USER_CANCELLED	0x05	User has cancelled the operation.
UAF_CMD_STATUS_CMD_NOT_SUPPORTED	0x06	Command not supported.

Name	Value	Description
UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED		Required attestation not supported.
UAF_CMD_STATUS_PARAMS_INVALID	0x08	The parameters for the command received by the authenticator are malformed/invalid.
UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY	0x09	The UAuth key which is relevant for this command disappeared from the authenticator and cannot be restored. On some authenticators this error occurs when the user verification reference data set was modified (e.g. new fingerprint template added).
UAF_CMD_STATUS_TIMEOUT	0x0a	The operation in the authenticator took longer than expected (due to technical issues) and it was finally aborted.
UAF_CMD_STATUS_USER_NOT_RESPONSIVE	0x0e	The user took too long to follow an instruction, e.g. didn't swipe the finger within the accepted time.
UAF_CMD_STATUS_INSUFFICIENT_RESOURCES	0x0f	Insufficient resources in the authenticator to perform the requested task.
UAF_CMD_STATUS_USER_LOCKOUT	0x10	The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that. Typically the user would have to enter an alternative password (formally: undergo some other alternative user verification method) to re-enable the use of the main user verification method. NOTE Any method the user can use to (re-) enable the main user verification method is considered an alternative user verification method and must be properly declared as such. For example, if the user can enter an alternative password to re-enable the use of fingerprints or to add additional fingers, the authenticator obviously supports fingerprint *or* password based user verification.

Table 4.4.1: UAF Authenticator Status Codes (0x00 - 0xFF)

5. Structures

This section is normative.

5.1 RawKeyHandle

RawKeyHandle is a structure generated and parsed by the authenticator. Authenticators may define RawKeyHandle in different ways and the internal structure is relevant only to the specific authenticator implementation.

RawKeyHandle for a typical **first-factor bound authenticator** has the following structure.

Depends on hashing algorithm (e.g. 32 bytes)	Depends on key type. (e.g. 32 bytes)	Username Size (1 byte)	Max 128 bytes
KHAccessToken	UAuth.priv	Size	Username

Table 5.1: RawKeyHandle Structure

First Factor authenticators must store Usernames in the authenticator and they must link the Username to the related key. This may be achieved by storing the Username inside the RawKeyHandle. Second Factor authenticators must not store the Username.

The ability to support Usernames is a key difference between first-, and second-factor authenticators.

The RawKeyHandle must be cryptographically wrapped before leaving the authenticator boundary since it typically contains sensitive information, e.g. the user authentication private key (UAuth.priv).

5.2 Structures to be parsed by FIDO Server

The structures defined in this section are created by UAF Authenticators and parsed by FIDO Servers.

Authenticators must generate these structures if they implement "UAFV1TLV" assertion scheme.

> NOTE
>
> "UAFV1TLV" assertion scheme assumes that the authenticator has exclusive control over all data included inside TAG_UAFV1_KRD and TAG_UAFV1_SIGNED_DATA.

The nesting structure must be preserved, but the order of tags within a composite tag is not normative. FIDO Servers must be prepared to handle tags appearing in any order.

5.2.1 TAG_UAFV1_REG_ASSERTION

The following TLV structure is generated by the authenticator during processing of a Register command. It is then delivered to FIDO Server intact, and parsed by the server. The structure embeds a TAG_UAFV1_KRD tag which among other data contains the newly generated UAuth.pub.

If the authenticator wants to append custom data to TAG_UAFV1_KRD structure (and thus sign with Attestation Key) - this data must be included as TAG_EXTENSION_DATA in a TAG_EXTENSION object inside TAG_UAFV1_KRD.

If the authenticator wants to send additional data to FIDO Server without signing it - this data must be included as TAG_EXTENSION_DATA in a TAG_EXTENSION object inside TAG_UAFV1_REG_ASSERTION and not inside TAG_UAFV1_KRD.

Currently this document only specifies TAG_ATTESTATION_BASIC_FULL, TAG_ATTESTATION_BASIC_SURROGATE and TAG_ATTESTATION_ECDAA. In case if the authenticator is required to perform "Some_Other_Attestation" on TAG_UAFV1_KRD - it must use the TLV tag and content defined for "Some_Other_Attestation" (defined in [UAFRegistry]).

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_REG_ASSERTION
1.1	UINT16 Length	Length of the structure
1.2	UINT16 Tag	TAG_UAFV1_KRD
1.2.1	UINT16 Length	Length of the structure
1.2.2	UINT16 Tag	TAG_AAID
1.2.2.1	UINT16 Length	Length of AAID
1.2.2.2	UINT8[] AAID	Authenticator Attestation ID
1.2.3	UINT16 Tag	TAG_ASSERTION_INFO
1.2.3.1	UINT16 Length	Length of Assertion Information
1.2.3.2	UINT16 AuthenticatorVersion	Vendor assigned authenticator version
1.2.3.3	UINT8 AuthenticationMode	For Registration this must be 0x01 indicating that the user has explicitly verified the action.
1.2.3.4	UINT16 SignatureAlgAndEncoding	Signature Algorithm and Encoding of the attestation signature. Refer to [FIDORegistry] for information on supported algorithms and their values.
1.2.3.5	UINT16 PublicKeyAlgAndEncoding	Public Key algorithm and encoding of the newly generated `UAuth.pub` key. Refer to [FIDORegistry] for information on supported algorithms and their values.
1.2.4	UINT16 Tag	TAG_FINAL_CHALLENGE_HASH
1.2.4.1	UINT16 Length	Final Challenge Hash length
1.2.4.2	UINT8[] FinalChallengeHash	(binary value of) Final Challenge Hash provided in the Command
1.2.5	UINT16 Tag	TAG_KEYID

TLV Structure		Description
1.2.5.1	UINT16 Length	Length of KeyID
1.2.5.2	UINT8[] KeyID	(binary value of) KeyID generated by Authenticator
1.2.6	UINT16 Tag	TAG_COUNTERS
1.2.6.1	UINT16 Length	Length of Counters
1.2.6.2	UINT32 SignCounter	Signature Counter. Indicates how many times this authenticator has performed signatures in the past.
1.2.6.3	UINT32 RegCounter	Registration Counter. Indicates how many times this authenticator has performed registrations in the past.
1.2.7	UINT16 Tag	TAG_PUB_KEY
1.2.7.1	UINT16 Length	Length of UAuth.pub
1.2.7.2	UINT8[] PublicKey	User authentication public key (UAuth.pub) newly generated by authenticator
1.3 **(choice 1)**	UINT16 Tag	TAG_ATTESTATION_BASIC_FULL
1.3.1	UINT16 Length	Length of structure
1.3.2	UINT16 Tag	TAG_SIGNATURE
1.3.2.1	UINT16 Length	Length of signature
1.3.2.2	UINT8[] Signature	Signature calculated with Basic Attestation Private Key over TAG_UAFV1_KRD content. The entire TAG_UAFV1_KRD content, including the tag and it's length field, must be included during signature computation.
1.3.3	UINT16 Tag	TAG_ATTESTATION_CERT (multiple occurrences possible) Multiple occurrences must be ordered. The attestation certificate must occur first. Each subsequent occurrence (if exists) must be the issuing certificate of the previous occurrence. The last occurence must be chained to one of the certificates included in field `attestationRootCertificate` in the related Metadata Statement [FIDOMetadataStatement].
1.3.3.1	UINT16 Length	Length of Attestation Cert
1.3.3.2	UINT8[] Certificate	Single X.509 DER-encoded [ITU-X690-2008] Attestation Certificate or a single certificate from the attestation certificate chain (see description above).
1.3 **(choice 2)**	UINT16 Tag	TAG_ATTESTATION_BASIC_SURROGATE
1.3.1	UINT16 Length	Length of structure
1.3.2	UINT16 Tag	TAG_SIGNATURE
1.3.2.1	UINT16 Length	Length of signature
1.3.2.2	UINT8[] Signature	Signature calculated with newly generated UAuth.priv key over TAG_UAFV1_KRD content. The entire TAG_UAFV1_KRD content, including the tag and it's length field, must be included during signature computation.
1.3 **(choice 3)**	UINT16 Tag	TAG_ATTESTATION_ECDAA
1.3.1	UINT16 Length	Length of structure
1.3.2	UINT16 Tag	TAG_SIGNATURE
1.3.2.1	UINT16 Length	Length of signature

TLV Structure		Description
1.3.2.2	UINT8[] Signature	The binary ECDAA signature as specified in [FIDOEcdaaAlgorithm].

5.2.2 TAG_UAFV1_AUTH_ASSERTION

The following TLV structure is generated by an authenticator during processing of a Sign command. It is then delivered to FIDO Server intact and parsed by the server. The structure embeds a TAG_UAFV1_SIGNED_DATA tag.

If the authenticator wants to append custom data to TAG_UAFV1_SIGNED_DATA structure (and thus sign with Attestation Key) - this data must be included as an additional tag inside TAG_UAFV1_SIGNED_DATA.

If the authenticator wants to send additional data to FIDO Server without signing it - this data must be included as an additional tag inside TAG_UAFV1_AUTH_ASSERTION and not inside TAG_UAFV1_SIGNED_DATA.

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_AUTH_ASSERTION
1.1	UINT16 Length	Length of the structure.
1.2	UINT16 Tag	TAG_UAFV1_SIGNED_DATA
1.2.1	UINT16 Length	Length of the structure.
1.2.2	UINT16 Tag	TAG_AAID
1.2.2.1	UINT16 Length	Length of AAID
1.2.2.2	UINT8[] AAID	Authenticator Attestation ID
1.2.3	UINT16 Tag	TAG_ASSERTION_INFO
1.2.3.1	UINT16 Length	Length of Assertion Information
1.2.3.2	UINT16 AuthenticatorVersion	Vendor assigned authenticator version.
1.2.3.3	UINT8 AuthenticationMode	Authentication Mode indicating whether user explicitly verified or not and indicating if there is a transaction content or not. • 0x01 means that user has been explicitly verified • 0x02 means that transaction content has been shown on the display and user confirmed it by explicitly verifying with authenticator
1.2.3.4	UINT16 SignatureAlgAndEncoding	Signature algorithm and encoding format. Refer to [FIDORegistry] for information on supported algorithms and their values.
1.2.4	UINT16 Tag	TAG_AUTHENTICATOR_NONCE
1.2.4.1	UINT16 Length	Length of authenticator Nonce -must be at least 8 bytes
1.2.4.2	UINT8[] AuthnrNonce	(binary value of) A nonce randomly generated by Authenticator
1.2.5	UINT16 Tag	TAG_FINAL_CHALLENGE_HASH
1.2.5.1	UINT16 Length	Length of Final Challenge Hash
1.2.5.2	UINT8[] FinalChallengeHash	(binary value of) Final Challenge Hash provided in the Command
1.2.6	UINT16 Tag	TAG_TRANSACTION_CONTENT_HASH
1.2.6.1	UINT16 Length	Length of Transaction Content Hash. This length is 0 if AuthenticationMode == 0x01, i.e. authentication, not transaction confirmation.
1.2.6.2	UINT8[] TCHash	(binary value of) Transaction Content Hash
1.2.7	UINT16 Tag	TAG_KEYID
1.2.7.1	UINT16 Length	Length of KeyID
1.2.7.2	UINT8[] KeyID	(binary value of) KeyID
1.2.8	UINT16 Tag	TAG_COUNTERS
1.2.8.1	UINT16 Length	Length of Counters
1.2.8.2	UINT32 SignCounter	Signature Counter. Indicates how many times this authenticator has performed signatures in the past.

	TLV Structure	Description
1.3	UINT16 Tag	TAG_SIGNATURE
1.3.1	UINT16 Length	Length of Signature
1.3.2	UINT8[] Signature	Signature calculated using UAuth.priv over TAG_UAFV1_SIGNED_DATA structure. The entire TAG_UAFV1_SIGNED_DATA content, including the tag and it's length field, must be included during signature computation.

5.3 UserVerificationToken

This specification doesn't specify how exactly user verification must be performed inside the authenticator. Verification is considered to be an authenticator, and vendor, specific operation.

This document provides an example on how the "vendor_specific_UserVerify" command (a command which verifies the user using Authenticator's built-in technology) could be securely bound to UAF Register and Sign commands. This binding is done through a concept called `UserVerificationToken`. Such a binding allows decoupling "vendor_specific_UserVerify" and "UAF Register/Sign" commands from each other.

Here is how it is defined:

- The ASM invokes the "vendor_specific_UserVerify" command. The authenticator verifies the user and returns a `UserVerificationToken` back.
- The ASM invokes UAF.Register/Sign command and passes `UserVerificationToken` to it. The authenticator verifies the validity of `UserVerificationToken` and performs the FIDO operation if it is valid.

The concept of UserVerificationToken is non-normative. An authenticator might decide to implement this binding in a very different way. For example an authenticator vendor may decide to append a UAF Register request directly to their "vendor_specific_UserVerify" command and process both as a single command.

If `UserVerificationToken` binding is implemented, it should either meet one of the following criteria or implement a mechanism providing similar, or better security:

- `UserVerificationToken` must allow performing only a single UAF Register or UAF Sign operation.
- `UserVerificationToken` must be time bound, and allow performing multiple UAF operations within the specified time.

6. Commands

This section is non-normative.

NORMATIVE

UAF Authenticators which are designed to be interoperable with ASMs from different vendors must implement the command interface defined in this section. Examples of such authenticators:

- Bound Authenticators in which the core authenticator functionality is developed by one vendor, and the ASM is developed by another vendor
- Roaming Authenticators

NORMATIVE

UAF Authenticators which are tightly integrated with a custom ASM (typically bound authenticators) may implement a different command interface.

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, the authenticator must parse the first TLV tag and figure out which command is being issued.

6.1 GetInfo Command

6.1.1 Command Description

This command returns information about the connected authenticators. It may return 0 or more authenticators. Each authenticator has an assigned `authenticatorIndex` which is used in other commands as an authenticator reference.

6.1.2 Command Structure

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

TLV Structure		Description

6.1.3 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_GETINFO_CMD_RESPONSE
1.1	UINT16 Length	Response length
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 Value	Status Code returned by Authenticator
1.3	UINT16 Tag	TAG_API_VERSION
1.3.1	UINT16 Length	Length of API Version (must be 0x0001)
1.3.2	UINT8 Version	Authenticator API Version (must be 0x01). This version indicates the types of commands, and formatting associated with them, that are supported by the authenticator.
1.4	UINT16 Tag	TAG_AUTHENTICATOR_INFO (multiple occurrences possible)
1.4.1	UINT16 Length	Length of Authenticator Info
1.4.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.4.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.4.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.4.3	UINT16 Tag	TAG_AAID
1.4.3.1	UINT16 Length	Length of AAID
1.4.3.2	UINT8[] AAID	Vendor assigned AAID
1.4.4	UINT16 Tag	TAG_AUTHENTICATOR_METADATA
1.4.4.1	UINT16 Length	Length of Authenticator Metadata
1.4.4.2	UINT16 AuthenticatorType	Indicates whether the authenticator is bound or roaming, and whether it is first-, or second-factor only. The ASM must use this information to understand how to work with the authenticator. Predefined values: • 0x0001 - Indicates second-factor authenticator (first-factor when the flag is not set) • 0x0002 - Indicates roaming authenticator (bound authenticator when the flag is not set) • 0x0004 - Key handles will be stored inside authenticator and won't be returned to ASM • 0x0008 - Authenticator has a built-in UI for enrollment and verification. ASM should not show its custom UI • 0x0010 - Authenticator has a built-in UI for settings, and supports OpenSettings command. • 0x0020 - Authenticator expects TAG_APPID to be passed as an argument to commands where it is defined as an optional argument • 0x0040 - At least one user is enrolled in the authenticator. Authenticators which don't support the concept of user enrollment (e.g. USER_VERIFY_NONE, USER_VERIFY_PRESENCE) must always have this bit set. • 0x0080 - Authenticator supports user verification tokens (UVTs) as described in this document. See section 5.3 UserVerificationToken. • 0x0100 - Authenticator only accepts TAG_TRANSACTION_TEXT_HASH in Sign command. This flag may ONLY be set if TransactionConfirmationDisplay is set to 0x0003 (see section 6.3 Sign Command).
1.4.4.3	UINT8 MaxKeyHandles	Indicates maximum number of key handles this authenticator can receive and process in a single command. This information will be used by the ASM when invoking SIGN command with multiple key handles.
1.4.4.4	UINT32 UserVerification	User Verification method (as defined in [FIDORegistry])
1.4.4.5	UINT16 KeyProtection	Key Protection type (as defined in [FIDORegistry]).

TLV Structure		Description
1.4.4.6	UINT16 MatcherProtection	Matcher Protection type (as defined in [FIDORegistry]).
1.4.4.7	UINT16 TransactionConfirmationDisplay	Transaction Confirmation type (as defined in [FIDORegistry]). NOTE If Authenticator doesn't support Transaction Confirmation - this value must be set to 0.
1.4.4.8	UINT16 AuthenticationAlg	Authentication Algorithm (as defined in [FIDORegistry]).
1.4.5	UINT16 Tag	TAG_TC_DISPLAY_CONTENT_TYPE (optional)
1.4.5.1	UINT16 Length	Length of content type.
1.4.5.2	UINT8[] ContentType	Transaction Confirmation Display Content Type. See [FIDOMetadataStatement] for additional information on the format of this field.
1.4.6	UINT16 Tag	TAG_TC_DISPLAY_PNG_CHARACTERISTICS (optional,multiple occurrences permitted)
1.4.6.1	UINT16 Length	Length of display characteristics information.
1.4.6.2	UINT32 Width	See [FIDOMetadataStatement] for additional information.
1.4.6.3	UINT32 Height	See [FIDOMetadataStatement] for additional information.
1.4.6.4	UINT8 BitDepth	See [FIDOMetadataStatement] for additional information.
1.4.6.5	UINT8 ColorType	See [FIDOMetadataStatement] for additional information.
1.4.6.6	UINT8 Compression	See [FIDOMetadataStatement] for additional information.
1.4.6.7	UINT8 Filter	See [FIDOMetadataStatement] for additional information.
1.4.6.8	UINT8 Interlace	See [FIDOMetadataStatement] for additional information.
1.4.6.9	UINT8[] PLTE	See [FIDOMetadataStatement] for additional information.
1.4.7	UINT16 Tag	TAG_ASSERTION_SCHEME
1.4.7.1	UINT16 Length	Length of Assertion Scheme
1.4.7.2	UINT8[] AssertionScheme	Assertion Scheme (as defined in [UAFRegistry])
1.4.8	UINT16 Tag	TAG_ATTESTATION_TYPE (multiple occurrences possible)
1.4.8.1	UINT16 Length	Length of AttestationType
1.4.8.2	UINT16 AttestationType	Attestation Type values are defined in [UAFRegistry] by the constants with the prefix `TAG_ATTESTATION`.
1.4.9	UINT16 Tag	TAG_SUPPORTED_EXTENSION_ID (optional, multiple occurrences possible)
1.4.9.1	UINT16 Length	Length of SupportedExtensionID
1.4.9.2	UINT8[] SupportedExtensionID	SupportedExtensionID as a UINT8[] encoding of a UTF-8 string

6.1.4 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ERR_UNKNOWN`
- `UAF_CMD_STATUS_PARAMS_INVALID`

6.2 Register Command

This command generates a UAF registration assertion. This assertion can be used to register the authenticator with a FIDO Server.

6.2.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_REGISTER_CMD
1.1	UINT16 Length	Command Length

TLV Structure		Description
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.3	UINT16 Tag	TAG_APPID (optional)
1.3.1	UINT16 Length	Length of AppID
1.3.2	UINT8[] AppID	AppID (max 512 bytes)
1.4	UINT16 Tag	TAG_FINAL_CHALLENGE_HASH
1.4.1	UINT16 Length	Final Challenge Hash Length
1.4.2	UINT8[] FinalChallengeHash	Final Challenge Hash provided by ASM (max 32 bytes)
1.5	UINT16 Tag	TAG_USERNAME
1.5.1	UINT16 Length	Length of Username
1.5.2	UINT8[] Username	Username provided by ASM (max 128 bytes)
1.6	UINT16 Tag	TAG_ATTESTATION_TYPE
1.6.1	UINT16 Length	Length of AttestationType
1.6.2	UINT16 AttestationType	Attestation Type to be used
1.7	UINT16 Tag	TAG_KEYHANDLE_ACCESS_TOKEN
1.7.1	UINT16 Length	Length of KHAccessToken
1.7.2	UINT8[] KHAccessToken	KHAccessToken provided by ASM (max 32 bytes)
1.8	UINT16 Tag	TAG_USERVERIFY_TOKEN (optional)
1.8.1	UINT16 Length	Length of VerificationToken
1.8.2	UINT8[] VerificationToken	User verification token

6.2.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_REGISTER_CMD_RESPONSE
1.1	UINT16 Length	Command Length
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 Value	Status code returned by Authenticator
1.3	UINT16 Tag	TAG_AUTHENTICATOR_ASSERTION
1.3.1	UINT16 Length	Length of Assertion
1.3.2	UINT8[] Assertion	Registration Assertion (see section TAG_UAFV1_REG_ASSERTION).
1.4	UINT16 Tag	TAG_KEYHANDLE (optional)
1.4.1	UINT16 Length	Length of key handle
1.4.2	UINT8[] Value	(binary value of) key handle

6.2.3 Status Codes

- UAF_CMD_STATUS_OK
- UAF_CMD_STATUS_ERR_UNKNOWN
- UAF_CMD_STATUS_ACCESS_DENIED
- UAF_CMD_STATUS_USER_NOT_ENROLLED
- UAF_CMD_STATUS_USER_CANCELLED
- UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED
- UAF_CMD_STATUS_PARAMS_INVALID
- UAF_CMD_STATUS_TIMEOUT

- UAF_CMD_STATUS_USER_NOT_RESPONSIVE
- UAF_CMD_STATUS_INSUFFICIENT_RESOURCES
- UAF_CMD_STATUS_USER_LOCKOUT

6.2.4 Command Description

The authenticator must perform the following steps (see below table for command structure):

If the command structure is invalid (e.g. cannot be parsed correctly), return UAF_CMD_STATUS_PARAMS_INVALID.

1. If this authenticator has a transaction confirmation display and is able to display AppID, then make sure Command.TAG_APPID is provided, and show its content on the display when verifying the user. Return UAF_CMD_STATUS_PARAMS_INVALID if Command.TAG_APPID is not provided in such case. Update Command.KHAccessToken with TAG_APPID:
    - Update Command.KHAccessToken by mixing it with Command.TAG_APPID. An example of such mixing function is a cryptographic hash function.

      NOTE

      This method allows us to avoid storing the AppID separately in the RawKeyHandle.

        - For example: Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_APPID)
2. If the user is already enrolled with this authenticator (via biometric enrollment, PIN setup or similar mechanism) - verify the user. If the verification has been already done in a previous command - make sure that Command.TAG_USERVERIFY_TOKEN is a valid token.

   If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return UAF_CMD_STATUS_USER_LOCKOUT.

    1. If the user doesn't respond to the request to get verified - return UAF_CMD_STATUS_USER_NOT_RESPONSIVE
    2. If verification fails - return UAF_CMD_STATUS_ACCESS_DENIED
    3. If user explicitly cancels the operation - return UAF_CMD_STATUS_USER_CANCELLED
3. If the user is not enrolled with the authenticator then take the user through the enrollment process. If the enrollment process cannot be triggered by the authenticator, return UAF_CMD_STATUS_USER_NOT_ENROLLED.
    1. If the authenticator can trigger enrollment, but the user doesn't respond to the request to enroll - return UAF_CMD_STATUS_USER_NOT_RESPONSIVE
    2. If the authenticator can trigger enrollment, but enrollment fails - return UAF_CMD_STATUS_ACCESS_DENIED
    3. If the authenticator can trigger enrollment, but the user explicitly cancels the enrollment operation - return UAF_CMD_STATUS_USER_CANCELLED
4. Make sure that Command.TAG_ATTESTATION_TYPE is supported. If not - return UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED
5. Generate a new key pair (UAuth.pub/UAuth.priv) If the process takes longer than accepted - return UAF_CMD_STATUS_TIMEOUT
6. Create a RawKeyHandle, for example as follows
    1. Add UAuth.priv to RawKeyHandle
    2. Add Command.KHAccessToken to RawKeyHandle
    3. If a first-factor authenticator, then add Command.Username to RawKeyHandle

   If there are not enough resources in the authenticator to perform this task - return UAF_CMD_STATUS_INSUFFICIENT_RESOURCES.
7. Wrap RawKeyHandle with Wrap.sym key
8. Create TAG_UAFV1_KRD structure
    1. If this is a second-factor roaming authenticator - place key handle inside TAG_KEYID. Otherwise generate a random KeyID and place it inside TAG_KEYID.
    2. Copy all the mandatory fields (see section TAG_UAFV1_REG_ASSERTION)
9. Perform attestation on TAG_UAFV1_KRD based on provided Command.AttestationType.
10. Create TAG_AUTHENTICATOR_ASSERTION
    1. Create TAG_UAFV1_REG_ASSERTION
        1. Copy all the mandatory fields (see section TAG_UAFV1_REG_ASSERTION)
        2. If this is a first-factor roaming authenticator - add KeyID and key handle into internal storage
        3. If this is a bound authenticator - return key handle inside TAG_KEYHANDLE
    2. Put the entire TLV structure for TAG_UAFV1_REG_ASSERTION as the value of TAG_AUTHENTICATOR_ASSERTION
11. Return TAG_UAFV1_REGISTER_CMD_RESPONSE
    1. Use UAF_CMD_STATUS_OK as status code
    2. Add TAG_AUTHENTICATOR_ASSERTION
    3. Add TAG_KEY_HANDLE if the key handle must be stored outside the Authenticator

NORMATIVE

The authenticator must not process a `Register` command without verifying the user (or enrolling the user, if this is the first time the user has used the authenticator).

The authenticator must generate a unique UAuth key pair each time the Register command is called.

The authenticator should either store key handle in its internal secure storage or cryptographically wrap it and export it to the ASM.

For silent authenticators, the key handle must never be stored on a FIDO Server, otherwise this would enable tracking of users without providing the ability for users to clear key handles from the local device.

If KeyID is not the key handle itself (e.g. such as in case of a second-factor roaming authenticator) - it must be a unique and unguessable byte array with a maximum length of 32 bytes. It must be unique within the scope of the AAID.

> NOTE
>
> If the KeyID is generated randomly (instead of, for example, being derived from a key handle) - it should be stored inside RawKeyHandle so that it can be accessed by the authenticator while processing the Sign command.

If the authenticator doesn't support `SignCounter` or `RegCounter` it must set these to 0 in TAG_UAFV1_KRD. The `RegCounter` must be set to 0 when a factory reset for the authenticator is performed. The `SignCounter` must be set to 0 when a factory reset for the authenticator is performed.

6.3 Sign Command

This command generates a UAF assertion. This assertion can be further verified by a FIDO Server which has a prior registration with this authenticator.

6.3.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_SIGN_CMD
1.1	UINT16 Length	Length of Command
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.3	UINT16 Tag	TAG_APPID (optional)
1.3.1	UINT16 Length	Length of AppID
1.3.2	UINT8[] AppID	AppID (max 512 bytes)
1.4	UINT16 Tag	TAG_FINAL_CHALLENGE_HASH
1.4.1	UINT16 Length	Length of Final Challenge Hash
1.4.2	UINT8[] FinalChallengeHash	(binary value of) Final Challenge Hash provided by ASM (max 32 bytes)
1.5	UINT16 Tag	TAG_TRANSACTION_CONTENT (optional)
1.5.1	UINT16 Length	Length of Transaction Content
1.5.2	UINT8[] TransactionContent	(binary value of) Transaction Content provided by the ASM
1.5	UINT16 Tag	TAG_TRANSACTION_CONTENT_HASH (optional and mutually exclusive with TAG_TRANSACTION_CONTENT). This TAG is only allowed for authenticators not able to display the transaction text, i.e. authenticator with `tcDisplay=0x0003` (i.e. flags `TRANSACTION_CONFIRMATION_DISPLAY_ANY` and `TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE` are set).
1.5.1	UINT16 Length	Length of Transaction Content Hash
1.5.2	UINT8[] TransactionContentHash	(binary value of) Transaction Content Hash provided by the ASM
1.6	UINT16 Tag	TAG_KEYHANDLE_ACCESS_TOKEN
1.6.1	UINT16 Length	Length of KHAccessToken

	TLV Structure	Description
1.6.2	UINT8[] KHAccessToken	(binary value of) KHAccessToken provided by ASM (max 32 bytes)
1.7	UINT16 Tag	TAG_USERVERIFY_TOKEN (optional)
1.7.1	UINT16 Length	Length of the User Verification Token
1.7.2	UINT8[] VerificationToken	User Verification Token
1.8	UINT16 Tag	TAG_KEYHANDLE (optional, multiple occurrences permitted)
1.8.1	UINT16 Length	Length of KeyHandle
1.8.2	UINT8[] KeyHandle	(binary value of) key handle

6.3.2 Command Response

	TLV Structure	Description
1	UINT16 Tag	TAG_UAFV1_SIGN_CMD_RESPONSE
1.1	UINT16 Length	Entire Length of Command Response
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 Value	Status code returned by authenticator
1.3 **(choice 1)**	UINT16 Tag	TAG_USERNAME_AND_KEYHANDLE (optional, multiple occurances) This TLV tag can be used to convey multiple (>=1) {Username, Keyhandle} entries. Each occurance of TAG_USERNAME_AND_KEYHANDLE contains one pair. If this tag is present, TAG_AUTHENTICATOR_ASSERTION must not be present
1.3.1	UINT16 Length	Length of the structure
1.3.2	UINT16 Tag	TAG_USERNAME
1.3.2.1	UINT16 Length	Length of Username
1.3.2.2	UINT8[] Username	Username
1.3.3	UINT16 Tag	TAG_KEYHANDLE
1.3.3.1	UINT16 Length	Length of `KeyHandle`
1.3.3.2	UINT8[] KeyHandle	(binary value of) key handle
1.3 **(choice 2)**	UINT16 Tag	TAG_AUTHENTICATOR_ASSERTION (optional) If this tag is present, TAG_USERNAME_AND_KEYHANDLE must not be present
1.3.1	UINT16 Length	Assertion Length
1.3.2	UINT8[] Assertion	Authentication assertion generated by the authenticator (see section TAG_UAFV1_AUTH_ASSERTION).

6.3.3 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ERR_UNKNOWN`
- `UAF_CMD_STATUS_ACCESS_DENIED`

- UAF_CMD_STATUS_USER_NOT_ENROLLED
- UAF_CMD_STATUS_USER_CANCELLED
- UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT
- UAF_CMD_STATUS_PARAMS_INVALID
- UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY
- UAF_CMD_STATUS_TIMEOUT
- UAF_CMD_STATUS_USER_NOT_RESPONSIVE
- UAF_CMD_STATUS_USER_LOCKOUT

6.3.4 Command Description

> NOTE
>
> First-factor authenticators should implement this command in two stages.
>
> 1. The first stage will be executed only if the authenticator finds out that there are multiple key handles after filtering with the KHAccessToken. In this stage, the authenticator must return a list of usernames along with corresponding key handles
> 2. In the second stage, after the user selects a username, this command will be called with a single key handle and will return a UAF assertion based on this key handle
>
> If a second-factor authenticator is presented with more than one valid key handles, it must exercise only the first one and ignore the rest.
>
> The command is implemented in two stages to ensure that only one assertion can be generated for each command invocation.

Authenticators must take the following steps:

If the command structure is invalid (e.g. cannot be parsed correctly), return `UAF_CMD_STATUS_PARAMS_INVALID`.

1. If this authenticator has a transaction confirmation display, and is able to display the AppID - make sure Command.TAG_APPID is provided, and show it on the display when verifying the user. Return `UAF_CMD_STATUS_PARAMS_INVALID` if `Command.TAG_APPID` is not provided in such case.
   - Update Command.KHAccessToken by mixing it with Command.TAG_APPID. An example of such a mixing function is a cryptographic hash function.
     - Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_APPID)
2. If the user is already enrolled with the authenticator (such as biometric enrollment, PIN setup, etc.) then verify the user. If the verification has already been done in one of the previous commands, make sure that `Command.TAG_USERVERIFY_TOKEN` is a valid token.

   If the user is locked out (e.g. too many failed attempts to get verified) and the authenticator cannot automatically trigger unblocking, return `UAF_CMD_STATUS_USER_LOCKOUT`.

   1. If the user doesn't respond to the request to get verified - return `UAF_CMD_STATUS_USER_NOT_RESPONSIVE`
   2. If verification fails - return `UAF_CMD_STATUS_ACCESS_DENIED`
   3. If the user explicitly cancels the operation - return `UAF_CMD_STATUS_USER_CANCELLED`
3. If the user is not enrolled then return `UAF_CMD_STATUS_USER_NOT_ENROLLED`

   > NOTE
   >
   > This should not occur as the Uauth key must be protected by the authenticator's user verification method. If the authenticator supports alternative user verification methods (e.g. alternative password and finger print verification and the alternative password must be provided before enrolling a finger and *only* the finger print is verified as part of the *Register* or *Sign* operation, then the authenticator should automatically and implicitly ask the user to enroll the modality required in the operation (instead of just returning an error).

4. Unwrap all provided key handles from Command.TAG_KEYHANDLE values using Wrap.sym
   1. If this is a first-factor roaming authenticator:
      - If Command.TAG_KEYHANDLE are provided, then the items in this list are KeyIDs. Use these KeyIDs to locate key handles stored in internal storage
      - If no Command.TAG_KEYHANDLE are provided - unwrap all key handles stored in internal storage

   If no RawKeyHandles are found - return `UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY`.

5. Filter RawKeyHandles with Command.KHAccessToken (RawKeyHandle.KHAccessToken == Command.KHAccessToken)
6. If the number of remaining RawKeyHandles is 0, then fail with `UAF_CMD_STATUS_ACCESS_DENIED`
7. If number of remaining RawKeyHandles is > 1
   1. If this authenticator has a user interface and wants to use it for this purpose: Ask the user which of the usernames

he wants to use for this operation. Select the related RawKeyHandle and jump to step #8.

2. If this is a second-factor authenticator, then choose the first RawKeyHandle only and jump to step #8.
3. Copy {Command.KeyHandle, RawKeyHandle.username} for all remaining RawKeyHandles into TAG_USERNAME_AND_KEYHANDLE tag.
   - If this is a first-factor roaming authenticator, then the returned TAG_USERNAME_AND_KEYHANDLEs must be ordered by the key handle registration date (the latest-registered key handle must come the latest).

     NOTE

     If two or more key handles with the same username are found, a first-factor roaming authenticator may only keep the one that is registered most recently and delete the rest. This avoids having unusable (old) private key in the authenticator which (surprisingly) might become active after deregistering the newly generated one.

4. Copy TAG_USERNAME_AND_KEYHANDLE into TAG_UAFV1_SIGN_CMD_RESPONSE and return

8. If number of remaining RawKeyHandles is 1
   1. If the Uauth key related to the RawKeyHandle cannot be used or disappeared and cannot be restored - return `UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY`.
   2. Create TAG_UAFV1_SIGNED_DATA and set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x01
   3. If `TransactionContent` is not empty
      - If this is a silent authenticator, then return `UAF_CMD_STATUS_ACCESS_DENIED`
      - If the authenticator doesn't support transaction confirmation (it has set `TransactionConfirmationDisplay` to 0 in the response to a `GetInfo` Command), then return `UAF_CMD_STATUS_ACCESS_DENIED`
      - If the authenticator has a built-in transaction confirmation display, then show `Command.TransactionContent` and `Command.TAG_APPID` (optional) on display and wait for the user to confirm it:
        - Return `UAF_CMD_STATUS_USER_NOT_RESPONSIVE` if the user doesn't respond.
        - Return `UAF_CMD_STATUS_USER_CANCELLED` if the user cancels the transaction.
        - Return `UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT` if the provided transaction content cannot be rendered.
      - Compute hash of TransactionContent
      -
        - TAG_UAFV1_SIGNED_DATA.TAG_TRANSACTION_CONTENT_HASH = hash(Command.TransactionContent)
        - Set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x02
   4. If `TransactionContent` is not set, but `TransactionContentHash` is not empty
      - If this is a silent authenticator, then return `UAF_CMD_STATUS_ACCESS_DENIED`
      - If the conditions for receiving TransactionContentHash are not satisfied, i.e. if the authenticator's `TransactionConfirmationDisplay` is NOT set to 0x0003 in the response to a `GetInfo` Command), then return `UAF_CMD_STATUS_PARAMS_INVALID`
      -
        - TAG_UAFV1_SIGNED_DATA.TAG_TRANSACTION_CONTENT_HASH = Command.TransactionContentHash
        - Set TAG_UAFV1_SIGNED_DATA.AuthenticationMode to 0x02
   5. Create TAG_UAFV1_AUTH_ASSERTION
      - Fill in the rest of TAG_UAFV1_SIGNED_DATA fields
      -
        - Increment SignCounter and put into TAG_UAFV1_SIGNED_DATA
        - Copy all the mandatory fields (see section TAG_UAFV1_AUTH_ASSERTION)
        - If TAG_UAFV1_SIGNED_DATA.AuthenticationMode == 0x01 - set TAG_UAFV1_SIGNED_DATA.TAG_TRANSACTION_CONTENT_HASH.Length to 0
      - Sign TAG_UAFV1_SIGNED_DATA with UAuth.priv

      If these steps take longer than expected by the authenticator - return `UAF_CMD_STATUS_TIMEOUT`.
   6. Put the entire TLV structure for TAG_UAFV1_AUTH_ASSERTION as the value of TAG_AUTHENTICATOR_ASSERTION
   7. Copy TAG_AUTHENTICATOR_ASSERTION into TAG_UAFV1_SIGN_CMD_RESPONSE and return

NORMATIVE

Authenticator must not process Sign command without verifying the user first.

Authenticator must not reveal Username without verifying the user first.

Bound authenticators must not process Sign command without validating KHAccessToken first.

UAuth.priv keys must never leave Authenticator's security boundary in plaintext form. UAuth.priv protection boundary is specified in `Metadata.keyProtection` field in Metadata [FIDOMetadataStatement]).

If Authenticator's Metadata indicates that it does support Transaction Confirmation Display - it must display provided transaction content in this display and include the hash of content inside TAG_UAFV1_SIGNED_DATA structure.

Silent Authenticators must not operate in first-factor mode in order to follow the assumptions made in [FIDOSecRef].

If Authenticator doesn't support `SignCounter`, then it must set it to 0 in TAG_UAFV1_SIGNED_DATA. The `SignCounter` must be set to 0 when a factory reset for the Authenticator is performed, in order to follow the assumptions made in [FIDOSecRef].

Some Authenticators might support Transaction Confirmation display functionality not inside the Authenticator but within the boundaries of ASM. Typically these are software based Transaction Confirmation displays. When processing the Sign command with a given transaction such Authenticators should assume that they do have a builtin Transaction Confirmation display and should include the hash of transaction content in the final assertion without displaying anything to the user. Also, such Authenticator's Metadata file must clearly indicate the type of Transaction Confirmation display. Typically the flag of Transaction Confirmation display will be TRANSACTION_CONFIRMATION_DISPLAY_ANY or TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE. See [FIDORegistry] for flags describing Transaction Confirmation Display type.

6.4 Deregister Command

This command deletes a registered UAF credential from Authenticator.

6.4.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_DEREGISTER_CMD
1.1	UINT16 Length	Entire Command Length
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index
1.3	UINT16 Tag	TAG_APPID (optional)
1.3.1	UINT16 Length	Length of AppID
1.3.2	UINT8[] AppID	AppID (max 512 bytes)
1.4	UINT16 Tag	TAG_KEYID
1.4.1	UINT16 Length	Length of KeyID
1.4.2	UINT8[] KeyID	(binary value of) KeyID provided by ASM
1.5	UINT16 Tag	TAG_KEYHANDLE_ACCESS_TOKEN
1.5.1	UINT16 Length	Length of KeyHandle Access Token
1.5.2	UINT8[] KHAccessToken	(binary value of) KeyHandle Access Token provided by ASM (max 32 bytes)

6.4.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_DEREGISTER_CMD_RESPONSE
1.1	UINT16 Length	Entire Length of Command Response
1.2	UINT16 Tag	TAG_STATUS_CODE
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 StatusCode	StatusCode returned by Authenticator

6.4.3 Status Codes

- `UAF_CMD_STATUS_OK`
- `UAF_CMD_STATUS_ERR_UNKNOWN`
- `UAF_CMD_STATUS_ACCESS_DENIED`
- `UAF_CMD_STATUS_CMD_NOT_SUPPORTED`
- `UAF_CMD_STATUS_PARAMS_INVALID`

6.4.4 Command Description

Authenticator must take the following steps:

If the command structure is invalid (e.g. cannot be parsed correctly), return `UAF_CMD_STATUS_PARAMS_INVALID`.

1. If this authenticator has a Transaction Confirmation display and is able to display AppID, then make sure Command.TAG_APPID is provided. Return `UAF_CMD_STATUS_PARAMS_INVALID` if `Command.TAG_APPID` is not provided in such case.
    - Update Command.KHAccessToken by mixing it with Command.TAG_APPID. An example of such mixing function is a cryptographic hash function.
        - Command.KHAccessToken=hash(Command.KHAccessToken | Command.TAG_APPID)
2. If this Authenticator doesn't store key handles internally, then return `UAF_CMD_STATUS_CMD_NOT_SUPPORTED`
3. If the length of `TAG_KEYID` is zero (i.e., 0000 Hex), then
    - if `TAG_APPID` is provided, then
        - for each KeyHandle that maps to `TAG_APPID` do
            1. if RawKeyHandle.KHAccessToken == Command.KHAccessToken, then delete KeyHandle from internal storage, otherwise, note an error occured
        - if an error occured, then return UAF_CMD_STATUS_ACCESS_DENIED
    - if `TAG_APPID` is not provided, then delete all KeyHandles from internal storage where RawKeyHandle.KHAccessToken == Command.KHAccessToken
    - Go to step 5
4. If the length of `TAG_KEYID` is NOT zero, then
    - Find KeyHandle that matches Command.KeyID
    - Ensure that RawKeyHandle.KHAccessToken == Command.KHAccessToken
        - If not, then return `UAF_CMD_STATUS_ACCESS_DENIED`
    - Delete this KeyHandle from internal storage
5. Return `UAF_CMD_STATUS_OK`

> NOTE
>
> The authenticator must unwrap the relevant KeyHandles using Wrap.sym as needed.

> NORMATIVE
>
> Bound authenticators must not process Deregister command without validating KHAccessToken first.
>
> Deregister command should not explicitly reveal whether the provided keyID was registered or not.

> NOTE
> This command *never* returns `UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY` as this could reveal the keyID registration status.

6.5 OpenSettings Command

This command instructs the Authenticator to open its built-in settings UI (e.g. change PIN, enroll new fingerprint, etc).

The Authenticator must return `UAF_CMD_STATUS_CMD_NOT_SUPPORTED` if it doesn't support such functionality.

If the command structure is invalid (e.g. cannot be parsed correctly), the authenticator must return `UAF_CMD_STATUS_PARAMS_INVALID`.

6.5.1 Command Structure

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_OPEN_SETTINGS_CMD
1.1	UINT16 Length	Entire Command Length
1.2	UINT16 Tag	TAG_AUTHENTICATOR_INDEX
1.2.1	UINT16 Length	Length of AuthenticatorIndex (must be 0x0001)
1.2.2	UINT8 AuthenticatorIndex	Authenticator Index

6.5.2 Command Response

TLV Structure		Description
1	UINT16 Tag	TAG_UAFV1_OPEN_SETTINGS_CMD_RESPONSE
1.1	UINT16 Length	Entire Length of Command Response
1.2	UINT16 Tag	TAG_STATUS_CODE

	TLV Structure	Description
1.2.1	UINT16 Length	Status Code Length
1.2.2	UINT16 StatusCode	StatusCode returned by Authenticator

6.5.3 Status Codes

- UAF_CMD_STATUS_OK
- UAF_CMD_STATUS_ERR_UNKNOWN
- UAF_CMD_STATUS_CMD_NOT_SUPPORTED
- UAF_CMD_STATUS_PARAMS_INVALID

7. KeyIDs and key handles

This section is non-normative.

There are 4 types of Authenticators defined in this document and due to their specifics they behave differently while processing commands. One of the main differences between them is how they store and process key handles. This section tries to clarify it by describing the behavior of every type of Authenticator during the processing of relevant command.

7.1 first-factor Bound Authenticator

Register Command	Authenticator doesn't store key handles. Instead KeyHandle is always returned to ASM and stored in ASM database. KeyID is a randomly generated 32 bytes number (or simply the hash of KeyHandle).
Sign Command	When there is no user session (no cookies, a clear machine) the Server doesn't provide any KeyID (since it doesn't know which KeyIDs to provide). In this scenario the ASM selects all key handles and passes them to Authenticator. During step-up authentication (when there is a user session) Server provides relevant KeyIDs. ASM selects key handles that correspond to provided KeyIDs and pass to Authenticator.
Deregister Command	Since Authenticator doesn't store key handles, then there is nothing to delete inside Authenticator. ASM finds the KeyHandle corresponding to provided KeyID and deletes it.

7.2 2ndF Bound Authenticator

Register Command	Authenticator doesn't store key handles. Instead KeyHandle is always returned to ASM and stored in ASM database. KeyID is a randomly generated 32 bytes number (or simply the hash of KeyHandle).
Sign Command	This Authenticator cannot operate without Server providing KeyIDs. Thus it can't be used when there is no user session (no cookies, a clear machine). During step-up authentication (when there is a user session) Server provides relevant KeyIDs. ASM selects key handles that correspond to provided KeyIDs and pass to Authenticator.
Deregister Command	Since Authenticator doesn't store key handles, then there is nothing to delete inside it. ASM finds the KeyHandle corresponding to provided KeyID and deletes it.

7.3 first-factor Roaming Authenticator

Register Command	Authenticator stores key handles inside its internal storage. KeyHandle is never returned back to ASM. KeyID is a randomly generated 32 bytes number (or simply the hash of KeyHandle)
Sign Command	When there is no user session (no cookies, a clear machine) Server doesn't provide any KeyID (since it doesn't know which KeyIDs to provide). In this scenario Authenticator uses all key handles that correspond to the provided AppID. During step-up authentication (when there is a user session) Server provides relevant KeyIDs. Authenticator

	selects key handles that correspond to provided KeyIDs and uses them.
Deregister Command	Authenticator finds the right KeyHandle and deletes it from its storage.

7.4 2ndF Roaming Authenticator

Register Command	Neither Authenticator nor ASM store key handles. Instead KeyHandle is sent to the Server (in place of KeyID) and stored in User's record. From Server's perspective it's a KeyID. In fact KeyID is the KeyHandle.
Sign Command	This Authenticator cannot operate without Server providing KeyIDs. Thus it can't be used when there is no user session (no cookies, a clear machine). During step-up authentication Server provides KeyIDs which are in fact key handles. Authenticator finds the right KeyHandle and uses it.
Deregister Command	Since Authenticator and ASM don't store key handles, then there is nothing to delete on client side.

8. Access Control for Commands

This section is normative.

FIDO Authenticators may implement various mechanisms to guard access to privileged commands.

The following table summarizes the access control requirements for each command.

All UAF Authenticators must satisfy the access control requirements defined below.

Authenticator vendors may offer additional security mechanisms.

Terms used in the table:

- NoAuth - no access control
- UserVerify - explicit user verification
- KHAccessToken - must be known to the caller
- KeyHandleList - must be known to the caller
- KeyID - must be known to the caller

Command	First-factor Bound Authenticator	2ndF Bound Authenticator	First-factor Roaming Authenticator	2ndF Roaming Authenticator
GetInfo	NoAuth	NoAuth	NoAuth	NoAuth
OpenSettings	NoAuth	NoAuth	NoAuth	NoAuth
Register	UserVerify	UserVerify	UserVerify	UserVerify
Sign	UserVerify KHAccessToken KeyHandleList	UserVerify KHAccessToken KeyHandleList	UserVerify KHAccessToken	UserVerify KHAccessToken KeyHandleList
Deregister	KHAccessToken KeyID	KHAccessToken KeyID	KHAccessToken KeyID	KHAccessToken KeyID

Table 1: Access Control for Commands

9. Considerations

This section is non-normative.

9.1 Algorithms and Key Sizes

The proposed algorithms and key sizes are chosen such that compatibility to TPMv2 is possible.

9.2 Indicating the Authenticator Model

Some authenticators (e.g. TPMv2) do not have the ability to include their model identifier (i.e. vendor ID and model name) in attested messages (i.e. the to-be-signed part of the registration assertion). The TPM's endorsement key certificate typically contains that information directly or at least it allows the model to be derived from the endorsement key certificate.

In FIDO, the relying party expects the ability to cryptographically verify the authenticator model (i.e. AAID).

If the authenticator cannot securely include its model (i.e. AAID) in the registration assertion (i.e. in the KRD object), we require the ECDAA-Issuers public key (ipkk) to be dedicated to one single authenticator model (identified by its AAID).

Using this method, the issuer public key is uniquely related to one entry in the Metadata Statement and can be used by the FIDO server to get a cryptographic proof of the Authenticator model.

10. Relationship to other standards

This section is non-normative.

The existing standard specifications most relevant to UAF authenticator are [TPM], [TEE] and [SecureElement].

Hardware modules implementing these standards may be extended to incorporate UAF functionality through their extensibility mechanisms such as by loading secure applications (trustlets, applets, etc) into them. Modules which do not support such extensibility mechanisms cannot be fully leveraged within UAF framework.

10.1 TEE

In order to support UAF inside TEE a special Trustlet (trusted application running inside TEE) may be designed which implements UAF Authenticator functionality specified in this document and also implements some kind of user verification technology (biometric verification, PIN or anything else).

An additional ASM must be created which knows how to work with the Trustlet.

10.2 Secure Elements

In order to support UAF inside Secure Element (SE) a special Applet (trusted application running inside SE) may be designed which implements UAF Authenticator functionality specified in this document and also implements some kind of user verification technology (biometric verification, PIN or similar mechanisms).

An additional ASM must be created which knows how to work the Applet.

10.3 TPM

TPMs typically have a built-in attestation capability however the attestation model supported in TPMs is currently incompatible with UAF's basic attestation model. The future enhancements of UAF may include compatible attestation schemes.

Typically TPMs also have a built-in PIN verification functionality which may be leveraged for UAF. In order to support UAF with an existing TPM module, the vendor should write an ASM which:

- Translates UAF data to TPM data by calling TPM APIs
- Creates assertions using TPMs API
- Reports itself as a valid UAF authenticator to FIDO UAF Client

A special AssertionScheme, designed for TPMs, must be also created (see [FIDOMetadataStatement]) and published by FIDO Alliance. When FIDO Server receives an assertion with this AssertionScheme it will treat the received data as TPM-generated data and will parse/validate it accordingly.

10.4 Unreliable Transports

The command structures described in this document assume a reliable transport and provide no support at the application-layer to detect or correct for issues such as unreliable ordering, duplication, dropping or modification of messages. If the transport layer(s) between the ASM and Authenticator are not reliable, the non-normative private contract between the ASM and Authenticator may need to provide a means to detect and correct such errors.

A. Security Guidelines

This section is non-normative.

Category	Guidelines
AppIDs and KeyIDs	Registered AppIDs and KeyIDs must not be returned by an authenticator in plaintext, without first performing user verification. If an attacker gets physical access to a roaming authenticator, then it should not be easy to read out AppIDs and KeyIDs.
Attestation Private Key	Authenticators must protect the attestation private key as a very sensitive asset. The overall security of the 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, e.g. [SecureElement]. It is assumed by registration assertion schemes, that the authenticator has exclusive control over the data being signed with the attestation key. FIDO Authenticators must ensure that the attestation private key:

Category	Guidelines
	1. is only used to attest authentication keys generated and protected by the authenticator, using the FIDO-defined data structures, KeyRegistrationData. 2. is never accessible outside the security boundary of the authenticator. Attestation must be implemented in a way such that two different relying parties cannot link registrations, authentications or other transactions (see [UAFProtocol]).
Certifications	Vendors should strive to pass common security standard certifications with authenticators, such as [FIPS140-2], [CommonCriteria] and similar. Passing such certifications will positively impact the UAF implementation of the authenticator.
Cryptographic (Crypto) Kernel	The crypto kernel is a module of the authenticator implementing cryptographic functions (key generation, signing, wrapping, etc) necessary for UAF, and having access to UAuth.priv, Attestation Private Key and Wrap.sym. For optimal security, this module should reside within the same security boundary as the UAuth.priv, Att.priv and Wrap.sym keys. If it resides within a different security boundary, then 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 environment [TEE]. 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. 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 a high quality entropy source, for: 1. generating authentication keys 2. generating signatures 3. computing authenticator-generated challenges The authenticator's random number generator (RNG) should be such that it cannot be disabled or controlled in a way that may cause it to generate predictable outputs. If the authenticator doesn't have sufficient entropy for generating strong random numbers, it should fail safely. See the section of this table regarding random numbers
KeyHandle	It is highly recommended to use authenticated encryption while wrapping key handles with Wrap.sym. Algorithms such as AES-GCM and AES-CCM are most suitable for this operation.
Liveness Detection / Presentation Attack Detection	The user verification method should include liveness detection [NSTCBiometrics], i.e. a technique to ensure that the sample submitted is actually from a (live) user. In the case of PIN-based matching, this could be implemented using [TEESecureDisplay] in order to ensure that malware can't emulate PIN entry.
Matcher	By definition, the matcher component is part of the authenticator. This does not impose any restrictions on the authenticator implementation, but implementers need to make sure that there is a proper security boundary binding the matcher and the other parts of the authenticator together. Tampering with the matcher module may have significant security consequences. It is highly recommended for this module to reside within the integrity boundaries of the authenticator, and be capable of detecting tampering. It is highly recommended to run this module inside a trusted execution environment [TEE] or inside a secure element [SecureElement]. Authenticators which have separated matcher and CryptoKernel modules should implement mechanisms which would allow the CryptoKernel to securely receive assertions from the matcher module indicating the 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 an Authenticator receives an invalid UserVerificationToken it should treat this as an attack, and invalidate the cached UserVerificationToken. A UserVerificationToken should have a lifetime not exceeding 10 seconds.

Category	Guidelines
	Authenticators must implement anti-hammering protections for their matchers. Biometrics based authenticators must protect the captured biometrics data (such as fingerprints) as well as the reference data (templates), and make sure that the biometric data never leaves the security boundaries of authenticators. Matchers must only accept verification reference data enrolled by the user, i.e. they must not include any default PINs or default biometric reference data.
Private Keys (UAuth.priv and Attestation Private Key)	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 a very high probability. 3. They have a structure allowing only very few fields containing uncontrolled values, i.e. values which are neither generated nor verified by the authenticator
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. The (pseudo-)random numbers used by authenticators should successfully pass the randomness test specified in [Coron99] and they should follow the guidelines given in [SP800-90b]. Additionally, authenticators may choose to incorporate entropy provided by the FIDO Server via the `ServerChallenge` sent in requests (see [UAFProtocol]). When mixing multiple entropy sources, a suitable mixing function should be used, such as those described in [RFC4086].
RegCounter	The `RegCounter` provides an anti-fraud signal to the relying parties. Using the `RegCounter`, the relying party can detect authenticators which have been excessively registered. If the `RegCounter` is implemented: ensure that 1. it is increased by any registration operation and 2. it cannot be manipulated/modified otherwise (e.g. via API calls, etc.) A registration counter should be implemented as a global counter, i.e. one covering registrations to all AppIDs. This global counter should be increased by 1 upon any registration operation. Note: The RegCounter value should *not* be decreased by `Deregistration` operations.
SignCounter	When an attacker is able to extract a Uauth.priv key from a registered authenticator, this key can be used independently from the original authenticator. This is considered cloning of an authenticator. Good protection measures of the Uauth private keys is one method to prevent cloning authenticators. In some situations the protection measures might not be sufficient. If the Authenticator maintains a signature counter `SignCounter`, then the FIDO Server would have an additional method to detect cloned authenticators. If the `SignCounter` is implemented: ensure that 1. It is increased by any authentication / transaction confirmation operation and 2. it cannot be manipulated/modified otherwise (e.g. API calls, etc.) Signature counters should be implemented that are dedicated for each private key in order to preserve the user's privacy. A per-key `SignCounter` should be increased by 1, whenever the corresponding UAuth.priv key signs an assertion. A per-key `SignCounter` should be deleted whenever the corresponding UAuth key is deleted. If the authenticator is not able to handle many different signature counters, then a global signature counter covering all private keys should be implemented. A global `SignCounter` should be increased by a random positive integer value whenever any of the UAuth.priv keys is used to sign an assertion.

Category	Guidelines
	NOTE There are multiple reasons why the `SignCounter` value could be 0 in a registration response. A `SignCounter` value of 0 in an authentication response indicates that the authenticator doesn't support the `SignCounter` concept.
Transaction Confirmation Display	A transaction confirmation display must ensure that the user is presented with the provided transaction content, e.g. not overlaid by other display elements and clearly recognizable. See [CLICKJACKING] for some examples of threats and potential counter-measures For more guidelines refer to [TEESecureDisplay].

UAuth.priv	An authenticator must protect all UAuth.priv keys as its **most** sensitive assets. The overall security of the authenticator depends **significantly** on the protection level of these keys.
	
	It is highly recommended that this key is generated, stored and operated inside a trusted execution environment.
	
	In situations where physical attacks and side channel attacks are considered within the threat model, it is highly recommended to use a tamper-resistant hardware module.
	
	FIDO Authenticators must ensure that UAuth.priv keys:
	
	1. are specific to the particular account at one relying party (relying party is identified by an AppID)
	2. are generated based on good random numbers with sufficient entropy. The challenge provided by the FIDO Server during registration and authentication operations should be mixed into the entropy pool in order to provide 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.
	    1. authenticating to the application (as identified by the AppID) they have been generated for, or
	    2. confirming transactions to the application (as identified by AppID) they have been generated for, or
	    3. are only being used to create the FIDO defined data structures, i.e. KRD, SignData.

Username	A username must not be returned in plaintext in any condition other than the conditions described for the SIGN command. In all other conditions usernames must be stored within a `KeyHandle`.
Verification Reference Data	The verification reference data, such as fingerprint templates or the reference value of a PIN, are by definition part of the authenticator. This does not impose any particular restrictions on the authenticator implementation, but implementers need to make sure that there is a proper security boundary binding all parts of the authenticator together.
Wrap.sym	If the authenticator has a wrapping key (Wrap.sym), then the authenticator must protect this key as its most sensitive asset. The overall security of the authenticator depends on the protection of this key. Wrap.sym key strength must be equal or higher than the strength of secrets stored in a RawKeyHandle. Refer to [SP800-57] and [SP800-38F] 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. If the authenticator uses Wrap.sym, it must ensure that unwrapping corrupted KeyHandle and unwrapping data which has invalid contents (e.g. KeyHandle from invalid origin) are indistinguishable to the caller.

B. Table of Figures

C. References

C.1 Normative references

[Coron99]
J. Coron and D. Naccache *An accurate evaluation of Maurer's universal test* LNCS 1556, February 1999, URL: http://www.jscoron.fr/publications/universal.pdf
[FIDOEcdaaAlgorithm]
R. Lindemann, J. Camenisch, M. Drijvers, A. Edgington, A. Lehmann, R. Urian,*FIDO ECDAA Algorithm*. FIDO Alliance Implementation Draft. URLs:
HTML: fido-ecdaa-v1.1-id-20170202.html
PDF: fido-ecdaa-v1.1-id-20170202.pdf.
[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf
[FIDOMetadataStatement]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO Metadata Statements v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.pdf
[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.pdf
[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). International Telecommunications Union, November 2008 URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en
[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119
[SP800-90b]
Elaine Barker and John Kelsey, *NIST Special Publication 800-90b: Recommendation for the Entropy Sources Used for Random Bit Generation*. National Institute of Standards and Technology, April 2016, URL: http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf
[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges,*FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf
[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.pdf

C.2 Informative references

[CLICKJACKING]
D. Lin-Shung Huang, C. Jackson, A. Moshchuk, H. Wang, S. Schlechter*Clickjacking: Attacks and Defenses*. USENIX, July 2012, URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf
[CommonCriteria]
CommonCriteria Publications. CCRA Members, Work in progress, accessed March 2014. URL: http://www.commoncriteriaportal.org/cc/
[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.pdf
[FIPS140-2]
FIPS PUB 140-2: Security Requirements for Cryptographic Modules. National Institute of Standards and Technology, May 2001, URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
[NSTCBiometrics]
NSTC Subcommittee on Biometrics, *Biometrics Glossary*. National Science and Technology Council. 14 September 2006, URL: http://biometrics.gov/Documents/Glossary.pdf
[RFC4086]
D. Eastlake 3rd, J. Schiller, S. Crocker *Randomness Requirements for Security (RFC 4086)*, IETF, June 2005, URL: http://www.ietf.org/rfc/rfc4086.txt
[SP800-38F]
M. Dworkin, *NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*. National Institute of Standards and Technology, December 2012, URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf
[SP800-57]
Recommendation for Key Management – Part 1: General (Revision 3) SP800-57. July 2012. U.S. Department of Commerce/National Institute of Standards and Technology. URL: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf
[SecureElement]
GlobalPlatform Card Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp
[TEE]
GlobalPlatform Trusted Execution Environment Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp
[TEESecureDisplay]
GlobalPlatform Trusted User Interface API Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp
[TPM]
TPM Main Specification Trusted Computing Group. Accessed March 2014. URL:

http://www.trustedcomputinggroup.org/resources/tpm_main_specification

IMPLEMENTATION DRAFT

FIDO UAF Registry of Predefined Values

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html
Editor:
Dr. Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal

Abstract

This document defines all the strings and constants reserved by UAF protocols. The values defined in this document are referenced by various UAF specifications.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

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

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

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

This document defines the registry of UAF-specific constants that are used and referenced in various UAF specifications. It is expected that, over time, new constants will be added to this registry. For example new authentication algorithms and new types of authenticator characteristics will require new constants to be defined for use within the specifications.

FIDO-specific constants that are common to multiple protocol families are defined in [FIDORegistry].

3. Authenticator Characteristics

This section is normative.

3.1 Assertion Schemes

Names of assertion schemes are strings with a length of 8 characters.

UAF TLV based assertion scheme “UAFV1TLV”
: This assertion scheme allows the authenticator and the FIDO Server to exchange an asymmetric authentication key generated by the authenticator. The authenticator must generate a key pair (UAuth.pub/UAuth.priv) to be used with algorithm suites listed in [FIDORegistry] section "Authentication Algorithms" (with prefix `ALG_`). This assertion scheme is using a compact Tag Length Value (TLV) encoding for the KRD and SignData messages generated by the authenticators. This is the default assertion scheme for the UAF protocol.

4. Predefined Tags

This section is normative.

The internal structure of UAF authenticator commands is a “Tag-Length-Value” (TLV) sequence. The tag is a 2-byte unique 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 the value is the variable-sized series of bytes which contain data for this item in the sequence.

Although 2 bytes are allotted for the tag, only the first 14 bits (values up to 0x3FFF) should be used to accommodate the limitations of some hardware platforms.

A tag that has the 14th bit (0x2000) set indicates that it is critical and a receiver must abort processing the entire message if it cannot process that tag.

A tag that has the 13th bit (0x1000) set indicates a composite tag that can be parsed by recursive descent.

4.1 Tags used in the protocol

The following tags have been allocated for data types in UAF protocol messages:

TAG_UAFV1_REG_ASSERTION 0x3E01
: The content of this tag is the authenticator response to a Register command.

TAG_UAFV1_AUTH_ASSERTION 0x3E02

The content of this tag is the authenticator response to a Sign command.

TAG_UAFV1_KRD 0x3E03
Indicates Key Registration Data.

TAG_UAFV1_SIGNED_DATA 0x3E04
Indicates data signed by the authenticator using UAuth.priv key.

TAG_ATTESTATION_CERT 0x2E05
Indicates DER encoded attestation certificate.

TAG_SIGNATURE 0x2E06
Indicates a cryptographic signature.

TAG_ATTESTATION_BASIC_FULL 0x3E07
Indicates full basic attestation as defined in [UAFProtocol].

TAG_ATTESTATION_BASIC_SURROGATE 0x3E08
Indicates surrogate basic attestation as defined in [UAFProtocol].

TAG_ATTESTATION_ECDAA 0x3E09
Indicates use of elliptic curve based direct anonymous attestation as defined in [FIDOEcdaaAlgorithm]. Support for this attestation type is optional at this time. It might be required by FIDO Certification.

TAG_KEYID 0x2E09
Represents a generated KeyID.

TAG_FINAL_CHALLENGE_HASH 0x2E0A
Represents a generated final challenge hash as defined in [UAFProtocol].

TAG_AAID 0x2E0B
Represents an Authenticator Attestation ID as defined in [UAFProtocol].

TAG_PUB_KEY 0x2E0C
Represents a generated public key.

TAG_COUNTERS 0x2E0D
Represents the use counters for an authenticator.

TAG_ASSERTION_INFO 0x2E0E
Represents authenticator information necessary for message processing.

TAG_AUTHENTICATOR_NONCE 0x2E0F
Represents a nonce value generated by the authenticator.

TAG_TRANSACTION_CONTENT_HASH 0x2E10
Represents a hash of the transaction content sent to the authenticator.

TAG_EXTENSION 0x3E11, 0x3E12
This is a composite tag indicating that the content is an extension.

TAG_EXTENSION_ID 0x2E13
Represents extension ID. Content of this tag is a UINT8[] encoding of a UTF-8 string.

TAG_EXTENSION_DATA 0x2E14
Represents extension data. Content of this tag is a UINT8[] byte array.

TAG_RAW_USER_VERIFICATION_INDEX 0x0103
This is the raw UVI as it might be used internally by authenticators. This TAG shall not appear in assertions leaving the authenticator boundary as it could be used as global correlation handle.

TAG_USER_VERIFICATION_INDEX 0x0104
The user verification index (UVI) is a value uniquely identifying a user verification data record.

Each UVI value must be specific to the related key (in order to provide unlinkability). It also must contain sufficient entropy that makes guessing impractical. UVI values must not be reused by the Authenticator (for other biometric data or users).

The UVI data can be used by FIDO Servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".

As an example, the UVI could be computed as SHA256(KeyID I SHA256(rawUVI)), where the rawUVI reflects (a) the biometric reference data, (b) the related OS level user ID and (c) an identifier which changes whenever a factory reset is performed for the device, e.g. rawUVI = biometricReferenceData I OSLevelUserID I FactoryResetCounter.

FIDO Servers supporting UVI extensions must support a length of up to 32 bytes for the UVI value.

Example of the TLV encoded UVI extension (contained in an assertion, i.e. TAG_UAFV1_REG_ASSERTION or TAG_UAFV1_AUTH_ASSERTION)

```
...
04 01                    -- TAG_USER_VERIFICATION_INDEX (0x0104)
20                       -- length of UVI
00 43 B8 E3 BE 27 95 8C  -- the UVI value itself
28 D5 74 BF 46 8A 85 CF
46 9A 14 F0 E5 16 69 31
DA 4B CF FF C1 BB 11 32
82
...
```

TAG_RAW_USER_VERIFICATION_STATE 0x0105
This is the raw UVS as it might be used internally by authenticators. This TAG shall not appear in assertions leaving the authenticator boundary as it could be used as global correlation handle.

TAG_USER_VERIFICATION_STATE 0x0106
The user verification state (UVS) is a value uniquely identifying the set of active user verification data records.

Each UVS value must be specific to the related key (in order to provide unlinkability). It also must contain sufficient entropy that makes guessing impractical. UVS values must not be reused by the Authenticator (for other biometric data sets or users).

The UVS data can be used by FIDO Servers to understand whether an authentication was authorized by one of the biometric data records already known at the initial key generation.

As an example, the UVS could be computed as SHA256(KeyID | SHA256(rawUVS)), where the rawUVS reflects (a) the biometric reference data sets, (b) the related OS level user ID and (c) an identifier which changes whenever a factory reset is performed for the device, e.g. rawUVS = biometricReferenceDataSet | OSLevelUserID | FactoryResetCounter.

FIDO Servers supporting UVS extensions must support a length of up to 32 bytes for the UVS value.

Example of the TLV encoded UVS extension (contained in an assertion, i.e. TAG_UAFV1_REG_ASSERTION or TAG_UAFV1_AUTH_ASSERTION)

```
...
06 01                      -- TAG_USER_VERIFICATION_STATE (0x0106)
20                         -- length of UVS
00 18 C3 47 81 73 2B 65    -- the UVS value itself
83 E7 43 31 46 8A 85 CF
93 6C 36 F0 AF 16 69 14
DA 4B 1D 43 FE C7 43 24
45
...
```

TAG_RESERVED_5 0x0201
Reserved for future use. Name of the tag will change, value is fixed.

5. Predefined (untagged) Extensions

This section is normative.

5.1 Android SafetyNet Extension

This extension can be added

- by FIDO Servers to the UAF Request object (request extension) in the `OperationHeader` in order to trigger generation of the related response extension.
- by FIDO Clients to the ASM Request object (request extension) in order to trigger generation of the related response extension.
- by the ASM to the respective `exts` array in the `ASMResponse` object (response extension).
- by the FIDO Client to the respective `exts` array in either the `OperationHeader`, or the `AuthenticatorRegistrationAssertion`, or the `AuthenticatorSignAssertion` of the UAF Response object (response extension).

Extension identifier
fido.uaf.safetynet
Extension fail-if-unknown flag
`false`, i.e. this (request and response) extension can safely be ignored by all entities.
Extension `data` value

When present in a request (request extension)
empty string, i.e. the FIDO Server might add this extension to the UAF Request with an empty `data` value in order to trigger the generation of this extension for the UAF Response.

EXAMPLE 1: SafetyNet Request Extension

```
"exts": [{"id": "fido.uaf.safetynet", "data": "", "fail_if_unknown": false}]
```

When present in a response (response extension)

- If the request extension was successfully processed, the `data` value is set to the JSON Web Signature attestation result as returned by the call to `com.google.android.gms.safetynet.SafetyNetApi.AttestationResult`.
- If the FIDO Client or the ASM support this extension, but the underlying Android platform does not support it (e.g. Google Play Services is not installed), the `data` value is set to the string "p" (i.e. platform issue).

EXAMPLE 2: SafetyNet Response Extension - not supported by platform

```
"exts": [{"id": "fido.uaf.safetynet", "data": "p", "fail_if_unknown": false}]
```

- If the FIDO Client or the ASM support this extension and the underlying Android platform supports it, but the functionality is temporarily unavailable (e.g. Google servers are unreachable), the `data` value is set to the string "a" (i.e. availability issue).

EXAMPLE 3: SafetyNet Response Extension - temporarily unavailable

```
"exts": [{"id": "fido.uaf.safetynet", "data": "a", "fail_if_unknown": false}]
```

> NOTE
>
> If neither the FIDO Client nor the ASM support this extension, it won't be present in the response object.

FIDO Client processing

FIDO Clients running on Android should support processing of this extension.

If the FIDO Client finds this (request) extension with empty `data` value in the UAF Request and it supports processing this extension, then the FIDO Client

1. must call the Android API `SafetyNet.SafetyNetApi.attest(mGoogleApiClient, nonce)` (see SafetyNet online documentation) and add the response (or an error code as described above) as extension to the response object.
2. must not copy the (request) extension to the ASM Request object (deviating from the general rule in [UAFProtocol], section 3.4.6.2 and 3.5.7.2).

If the FIDO Client does not support this extension it must copy this extension from the UAF Request to the ASM Request object (according to the general rule in [UAFProtocol], section 3.4.6.2 and 3.5.7.2).

If the ASM supports this extension it must call the SafetyNet API (see above) and add the response as extension to the ASM Response object. The FIDO Client must copy the extension in the ASM Response to the UAF Response object (according to sections 3.4.6.4. and 3.5.7.4 step 4 in [UAFProtocol]).

When calling the Android API, the nonce parameter must be set to the serialized JSON object with the following structure:

```
{
  "hashAlg": "S256", // the hash algorithm
  "fcHash": "..."    // the finalChallengeHash
}
```

Where

- `hashAlg` identifies the hash algorithm according to [FIDOSignatureFormat], section IANA Considerations.
- `fcHash` is the base64url encoded hash value of FinalChallenge (see section 3.6.3 and 3.7.4 in [UAFASM] for details on how to compute `finalChallengeHash`).
  We use this method to bind this SafetyNet extension to the respective FIDO UAF message.

  Only hash algorithms belonging to the Authentication Algorithms mentioned in [FIDORegistry] shall be used (e.g. SHA256 because it belongs to `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW`).

Authenticator argument
N/A
Authenticator processing
N/A. This extension is related to the Android platform in general and not to the authenticator in particular. As a consequence there is no need for an authenticator to receive the (request) extension nor to process it.
Authenticator data
N/A
Server processing
If the FIDO Server requested the SafetyNet extension,

1. it should verify that a proper response is provided (if client side support can be assumed), and
2. it should verify the SafetyNet AttestationResult (see SafetyNet online documentation).

> NOTE
>
> The package name in AttestationResult might relate to either the FIDO Client or the ASM.

> NOTE
>
> The response extension is not part of the signed assertion generated by the authenticator. If an MITM or MITB attacker would remove the response extension, the FIDO server might not be able to distinguish this from the "SafetyNet extension not supported by FIDO Client/ASM" case.

5.2 Android Key Attestation

This extension can be added

- by FIDO Servers to the UAF Request object (request extension) in the `OperationHeader` in order to trigger generation of the related response extension.

- by FIDO Clients to the ASM Request object (request extension) in order to trigger generation of the related response extension.
- by the ASM to the respective `exts` array in the `ASMResponse` object (response extension).
- by the FIDO Client to the respective `exts` array in either the `OperationHeader`, or the `AuthenticatorRegistrationAssertion`, or the `AuthenticatorSignAssertion` of the UAF Response object (response extension).

Extension identifier
fido.uaf.android.key_attestation

Extension fail-if-unknown flag
`false`, i.e. this (request and response) extension can safely be ignored by all entities.

Extension `data` value

When present in a request (request extension)
empty string, i.e. the FIDO Server might add this extension to the UAF Request with an empty `data` value in order to trigger the generation of this extension for the UAF Response.

EXAMPLE 4: Android KeyAttestation Request Extension

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "", "fail_if_unknown": false}]
```

When present in a response (response extension)

- If the request extension was successfully processed, the `data` value is set to a JSON array containing the base64 encoded entries of the array returned by the call to the KeyStore API function getCertificateChain.

EXAMPLE 5: Retrieve KeyAttestation and add it as extension

```
KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance(
  KeyProperties.KEY_ALGORITHM_EC, "AndroidKeyStore");
kpGenerator.initialize(
  new KeyGenParameterSpec.Builder(keyUUID, KeyProperties.PURPOSE_SIGN)
    .setDigests(KeyProperties.DIGEST_SHA256)
    .setAlgorithmParameterSpec(new ECGenParameterSpec("prime256v1"))
    .setCertificateSubject(
      new X500Principal(String.format("CN=%s, OU=%s",
        keyUUID, aContext.getPackageName())))
    .setCertificateSerialNumber(BigInteger.ONE)
    .setCertificateNotBefore(notBefore.getTime())
    .setCertificateNotAfter(notAfter.getTime())
    .setUserAuthenticationRequired(true)
    .setAttestationChallenge(fcHash) -- bind to Final Challenge
    .build());

kpGenerator.generateKeyPair(); // generate Uauth key pair

Certificate[] certarray=myKeyStore.getCertificateChain(keyUUID);
String certArray[]=new String[certarray.length];
int i=0;
for (Certificate cert : certarray) {
    byte[] buf = cert.getEncoded();
    certArray[i] = new String(Base64.encode(buf, Base64.DEFAULT));
    i++;
}

JSONArray jarray=new JSONArray(certArray);
String key_attestation_data=jarray.toString();
```

- If the FIDO Client or the ASM support this extension, but the underlying Android platform does not support it (e.g. Android version doesn't yet support it), the `data` value is set to the string "p" (i.e. platform issue).

EXAMPLE 6: KeyAttestation Response Extension - not supported by platform

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "p", "fail_if_unknown": false}]
```

- If the FIDO Client or the ASM support this extension and the underlying Android platform supports it, but the functionality is temporarily unavailable (e.g. Google servers are unreachable), the `data` value is set to the string "a".

EXAMPLE 7: KeyAttestation Response Extension - temporarily unavailable

```
"exts": [{"id": "fido.uaf.android.key_attestation", "data": "a", "fail_if_unknown": false}]
```

NOTE

> If neither the FIDO Client nor the ASM support this extension, it won't be present in the response object.

FIDO Client processing

FIDO Clients running on Android must pass this (request) extension with empty `data` value to the ASM.

If the ASM supports this extension it must call the KeyStore API (see above) and add the response as extension to the ASM Response object. The FIDO Client must copy the extension in the ASM Response to the UAF Response object (according to sections 3.4.6.4. and 3.5.7.4 step 4 in [UAFProtocol]).

More details on Android key attestation can be found at:

- https://developer.android.com/preview/api-overview.html#key_attestation
- https://source.android.com/security/keystore/
- https://source.android.com/security/keystore/implementer-ref.html

Authenticator argument
N/A

Authenticator processing
The authenticator generates the attestation response. The call keyStore.getCertificateChain is finally processed by the authenticator.

Authenticator data
N/A

Server processing
If the FIDO Server requested the key attestation extension,

1. it must follow the registration response processing rules (see FIDO UAF Protocol, section 3.4.6.5) before processing this extension
2. it must verify the syntax of the key attestation extension and it must perform RFC5280 compliant chain validation of the entries in the array to one attestationRootCertificate specified in the Metadata Statement.
3. it must determine the leaf certificate from that chain, and it must perform the following checks on this leaf certificate
    1. Verify that KeyDescripion.attestationChallenge == FCHash (see FIDO UAF Protocol, section 3.4.6.5 Step 6.)
    2. Verify that the public key included in the leaf certificate is identical to the public key included in the FIDO UAF Surrogate attestation block
    3. If the related Metadata Statement claims keyProtection KEY_PROTECTION_TEE, then refer to KeyDescription.teeEnforced using "authzList". If the related Metadata Statement claims keyProtection KEY_PROTECTION_SOFTWARE, then refer to KeyDescription.softwareEnforced using "authzList".
    4. Verify that
        1. authzList.origin == KM_TAG_GENERATED
        2. authzList.purpose == KM_PURPOSE_SIGN
        3. authzList.keySize is acceptable, i.e. =2048 (bit) RSA or =256 (bit) ECDSA.
        4. authzList.digest == KM_DIGEST_SHA_2_256.
        5. authzList.authType only contains acceptable user verification methods.
        6. authzList.authTimeout == 0 (or *not* present).
        7. authzList.noAuthRequired is *not* present (unless the Metadata Statement marks this authenticator as silent authenticator, i.e. userVerificaton set to USER_VERIFY_NONE).
        8. authzList.allApplications is *not* present, since FIDO Uauth keys must be bound to the generating app (AppID).

> NOTE
>
> The response extension is not part of the signed assertion generated by the authenticator. If an MITM or MITB attacker would remove the response extension, the FIDO server might not be able to distinguish this from the "KeyAttestation extension not supported by ASM/Authenticator" case.

ExtensionDescriptor `data` value (for Metadata Statement)
In the case of extension id="fido.uaf.android.key_attestation", the data field of the ExtensionDescriptor as included in the Metadata Statement will contain a dictionary containing the following data fields

DOMString attestationRootCertificates[]
Each element of this array represents a PKIX [RFC5280] X.509 certificate that is valid for this authenticator model. Multiple certificates might be used for different batches of the same model. The array does not represent a certificate chain, but only the trust anchor of that chain.

Each array element is a base64-encoded (section 4 of [RFC4648]), DER-encoded [ITU-X690-2008] PKIX certificate value.

> NOTE
>
> A certificate listed here is either a root certificate or an intermediate CA certificate.

> NOTE
>
> The field `data` is specified with type DOMString in [FIDOMetadataStatement] and hence will contain the serialized object as described above.

An example for the `supportedExtensions` field in the Metadata Statement could look as follows (with line breaks to improve readability):

EXAMPLE 8: Example of a supportedExtensions field in Metadata Statement


```
"supportedExtensions": [{
       "id": "fido.uaf.android.key_attestation",
       "data": "{ \"attestationRootCertificates\": [
\"MIICPTCCAeOgAwIBAgIJAOuexvU3Oy2wMAoGCCqGSM49BAMCMHsxIDAeBgNVBAMM
F1NhbXBsZSBBdHRlc3RhdGlvbiBSb290MRYwFAYDVQQKDA1GSURPIEFsbGlhbmNl
MREwDwYDVQQLDAhVQUYgVFdHLDESMBAGA1UEBwwJUGFsbyBBbHRvMQswCQYDVQQI
DAJDQTELMAkGA1UEBhMCVVMwHhcNMTQwNjE4MTMzMzMyWhcNNDExMTAzMTMzMzMy
WjB7MSAwHgYDVQQDDBdTYW1wbGUgQXR0ZXN0YXRpb24gUm9vdDEWMBQGA1UECgwN
RklETyBBbGxpYW5jZTERMA8GA1UECwwIVUFGIFRXRywxEjAQBgNVBAcMCVBhbG8g
QWx0bzELMAkGA1UECAwCQ0ExCzAJBgNVBAYTAlVTMFkwEwYHKoZIzj0CAQYIKoZI
zj0DAQcDQgAEH8hv2D0HXa59/BmpQ7RZehL/FMGzFd1QBg9vAUpOZ3ajnuQ94PR7
aMzH33nUSBr8fHYDrqOBb58pxGqHJRyX/6NQME4wHQYDVR0OBBYEFPoHA3CLhxFb
C0It7zE4w8hk5EJ/MB8GA1UdIwQYMBaAFPoHA3CLhxFbC0It7zE4w8hk5EJ/MAwG
A1UdEwQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIhAJ06QSXt9ihIbEKYKIjsPkri
VdLIgtfsbDSu7ErJfzr4AiBqoYCZf0+zI55aQeAHjIzA9Xm63rruAxBZ9ps9z2XN
lQ==\"] }",
       "fail_if_unknown": false
                      }]
```


6. Other Identifiers specific to FIDO UAF

6.1 FIDO UAF Application Identifier (AID)

This AID [ISOIEC-7816-5] is used to identify FIDO UAF authenticator applications in a Secure Element.

The FIDO UAF AID consists of the following fields:

Field	RID	AC	AX
Value	0xA000000647	0xAF	0x0001

Table 1: FIDO UAF Applet AID

A. References

A.1 Normative references

[FIDOEcdaaAlgorithm]
R. Lindemann, J. Camenisch, M. Drijvers, A. Edgington, A. Lehmann, R. Urian, *FIDO ECDAA Algorithm*. FIDO Alliance Implementation Draft. URLs:
HTML: fido-ecdaa-v1.1-id-20170202.html
PDF: fido-ecdaa-v1.1-id-20170202.pdf.

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.pdf

[ISOIEC-7816-5]
ISO 7816-5: Identification cards - Integrated circuit cards - Part 5: Registration of application providers

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

A.2 Informative references

[FIDOMetadataStatement]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO Metadata Statements v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.html

PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.pdf
[FIDOSignatureFormat]
FIDO 2.0: Signature format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-signature-format-v2.0-ps-20150904.html
[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). International Telecommunications Union, November 2008 URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en
[RFC4648]
S. Josefsson, *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*, IETF, October 2006, URL: http://www.ietf.org/rfc/rfc4648.txt
[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL:http://www.ietf.org/rfc/rfc5280.txt
[UAFASM]
D. Baghdasaryan, J. Kemp, R. Lindemann, B. Hill, R. Sasson,*FIDO UAF Authenticator-Specific Module API*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.pdf
[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges,*FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

IMPLEMENTATION DRAFT

FIDO UAF APDU

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-apdu-v1.1-id-20170202.html
Editor:
Naama Bak, Morpho
Contributors:
Virginie Galindo, Gemalto
Rolf Lindemann, Nok Nok Labs, Inc.
Ullrich Martini, Giesecke & Devrient
Chris Edwards, Intercede
Jeff Hodges, Paypal

Abstract

This specification defines a mapping of FIDO UAF Authenticator commands to Application Protocol Data Units (APDUs) thus facilitating UAF authenticators based on Secure Elements.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

The notation `base64url(byte[8..64])` reads as 8-64 bytes of data encoded in base64url, "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

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

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

All TLV structures defined in this documentmust be encoded in little-endian format.

All APDU defined in this document must be encoded as defined in [ISOIEC-7816-4-2013].

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. Introduction

This section is non-normative.

This specification defines the interface between the FIDO UAF Authenticator Specific Module (ASM) [UAFASM] and authenticators based upon "Secure Element" technology. The applicable secure element form factors are UICC (SIM card), embedded Secure Element (eSE), μSD, NFC card, and USB token. Their common characteristic is they communicate using Application Programming Data Units (APDU) in compliance with [ISOIEC-7816-4-2013].

Implementation of this specification is optional in the UAF framework, however, products claiming to implement the transport of UAF messages over APDUs should implement it.

This specification first describes the various fashions in which Secure Elements can be incorporated into UAF authenticator implementations — known as *SE-based authenticators* or just *SE authenticators* — and which components are responsible for handling user verification as well as cryptographic operations. The specification then describes the overall architecture of an SE-based authenticator stack from the ASM down to the secure element, the role of the "UAF Applet" running in the secure element, and outlines the nominal communication flow between the ASM and the SE. It then defines the mapping of UAF Authenticator commands to APDUs, as well as the FIDO-specific variants of the VERIFY APDU command.

> NOTE
>
> This specification does not define how an SE-based authenticator stack may be implemented, e.g., its integration with TEE or biometric sensors. However, SE-based authenticator vendors should reflect such implementation characteristics in the authenticator metadata such that FIDO Relying Parties wishing to be informed of said characteristics may have access to it.

3. SE-based Authenticator Implementation Use Cases

This section is non-normative.

Secure elements can be leveraged in different scenarios in the UAF technology. It can support user gestures (used to unlock access to FIDO credentials) or it can be involved in the actual cryptographic operations related to FIDO authentication. In this specification, we will be considering the following SE-based authenticator implementation use cases:

1. The Secure Element (SE) *is* the (silent) Authenticator.
2. The SE is part of the Authenticator which is composed of a Trusted Application (TEE) based User Verification component, potentially a TEE based transaction confirmation display and the crypto kernel inside the SE (**Hybrid SE Authenticator**).
3. The authenticator (Hybrid SE Authenticator) consists of
    - the SE implementing the matcher and the crypto kernel
    - and a specific software module (e.g. running on the FIDO User Device) to capture the user verification data (e.g. PIN, Face, Fingerprint).

3.1 Hybrid SE Authenticator

In FIDO UAF, the access to credentials for performing the actual authentication can be protected by a user verification step. This user verification step can be based on a PIN, a biometric or other methods. The authenticator functionality might be implemented in different components, including combinations such as TEE and SE, or fingerprint sensor and SE. In that case the SE implements only a part of the authenticator functionality.

> NOTE
>
> The reason for using such hybrid configuration is that Secure Elements do not have any user interface and hence cannot directly distinguish physical user interaction from programmatic communication (e.g. by malware). The ability to require a physical user interaction that cannot be emulated by malware is essential for protecting against scalable attacks (see [FIDOSecRef]). On the other hand, TEEs (or biometric sensors implemented in separate hardware) which can provide a trusted user interface typically do not offer the same level of key protection as Secure Elements.
>
> Strictly spoken, a Hybrid SE Authenticator (voluntarily) uses the Authenticator Command interface [UAFAuthnrCommands] *inside* the authenticator, e.g. between the crypto kernel and the user verification component.

Examples of hybrid SE authenticators are:

1. User PIN code capture and verification are implemented entirely in a TEE relying on Trusted User Interface and secure storage capabilities of the TEE and, once the PIN code is verified, the FIDO UAF crypto operations are performed in the SE.
2. User fingerprint is captured via a fingerprint sensor, the fingerprint match is performed in the TEE, relying on matching algorithms. Once the fingerprint has been positively checked, the cryptographic operations are executed in the Secure Element.
3. The user verification is implemented as match-on-chip in separate hardware and FIDO UAF cryptographic operations are implemented in the SE.

In all those cases, the hybrid nature of the authenticator will be managed by the software-based host, regardless of its nature (TEE, SW, Biometric sensor..). There are a number of possible interactions between the ASM and the SE actually implementing the verification and the cryptographic operations to consider within those use cases.

1. PIN user verification where the user interaction for the PIN entry is performed externally to the SE. The PIN may then be passed within a VERIFY command to the SE, followed by the actual cryptographic operations (such as the Register and Sign UAF authenticator commands).
2. Biometric user verification where the sample capture and matching is performed externally to the SE (e.g. in TEE or in a match-on-chip FP sensor). This would then only need to send to the SE the actual cryptographic operation needed in this session (such as the Register and Sign UAF authenticator commands).
3. User verification sample (Faceprint, Fingerprint..) capture is performed externally to the SE. The sample is then sent to a match-on-card applet in the SE that behaves as a global PIN to enable access to the cryptographic operation required within this session.

3.1.1 Architecture of the Hybrid SE Authenticator

In order to support an Hybrid SE Authenticator, a dedicated software-based host must be created which knows how the SE applet works. The communication between the SE applet and the host is defined based on [ISOIEC-7816-4-2013]. Whether a PC or mobile device the architecture is still the same, as defined below:

- `Application Layer`: This component is responsible for acquiring the user verification sample and mapping UAF commands to APDU commands.
- `Communication layer`: This is the [ISOIEC-7816-4-2013] APDUs interface, which provides methods to list and select readers, connect to a Secure Element and interact with it.
- `SE Access OS APIs` : OMA, PC/SC, NFC API, CCID…
- `Secure Element` : UICC, micro SD, eSE, Dual Interface card..

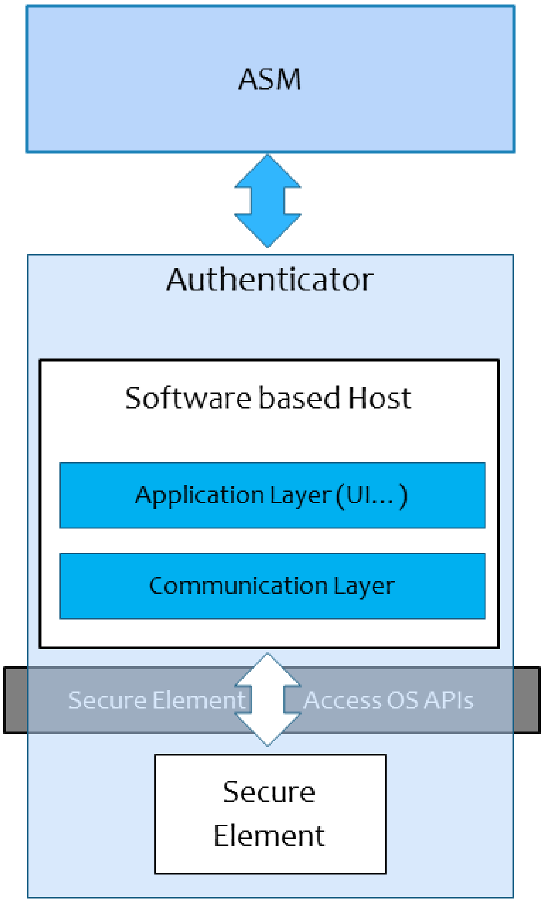


Fig. 1 Architecture of Hybrid SE Authenticator

APDU command-response paire are handled as indicated in [ISOIEC-7816-4-2013].

3.1.2 Communication flow between the ASM and the Hybrid SE Authenticator

The host is the entity communicating with the SE and which knows how the SE and the applet running in the SE can be accessed. The host could be a Trusted Application (TA) which runs inside a TEE or simply an application which runs in the normal world.

The following diagram illustrates how the Host of the Hybrid SE Authenticator may map the UAF commands to APDU commands. In this diagram, the User Verification Module is considered inside the SE applet.

> NOTE
>
> If the User Verification Module is inside the Host, for example in the context of the TEE, the `UserVerificationToken` shall be generated in the Host and not in the SE. As a result step 6 (Figure 2) should be executed in the Host instead of the SE.

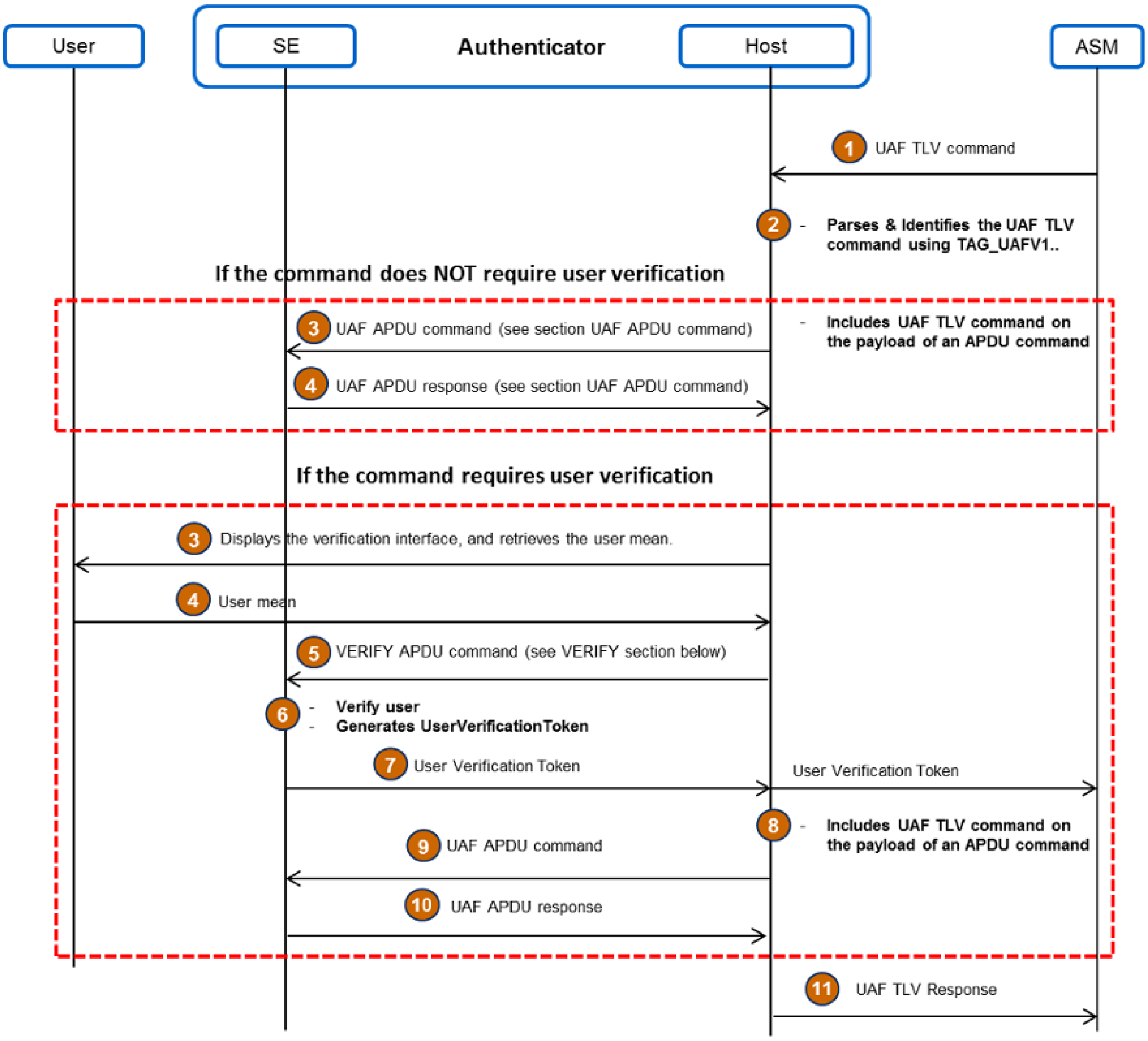


Fig. 2 Communication flow between the ASM and the Hybrid SE Authenticator

4. FIDO UAF Applet and APDU commands

This section is normative.

4.1 UAF Applet in the Authenticator

4.1.1 Application Identifier

The FIDO UAF AID is defined in [UAFRegistry].

4.1.2 User Verification

The User verification is based on the submission of a PIN/password (i.e., knowledge based) or a biometric template (i.e., biometric based).

In this document, the envisaged user verification methods are PIN and biometric based.

4.1.3 Cryptographic operations

The SE applet must be able to perform a set of cryptographic operations, such as key generation and signature computation. The cryptographic operations are defined in [UAFAuthnrCommands]. The SE applet must be able also to create data structures that can be parsed by FIDO Server. The SE applet shall use the cryptographic algorithms indicated in [UAFRegistry].

4.2 APDU Commands for FIDO UAF

4.2.1 Class byte coding

CLA indicates the class of the command.

Commands	CLA
SELECT, VERIFY (ISO Version), GET RESPONSE (ISO Version)	0x00
VERIFY, UAF, GET RESPONSE	0x80

Table 2: Class byte coding

NOTE

If the payload of an APDU command is longer than 255 bytes, command chaining as described in [ISOIEC-7816-4-2013] should be used, even though CLA is proprietary.

4.2.2 APDU command "UAF"

4.2.2.1 Mapping between FIDO UAF authenticator commands and APDU commands

This section describes the mapping between FIDO UAF authenticator commands and APDU commands.

The mapping consists of encapsulating the entire UAF Authenticator Command in the payload of the APDU command, and the UAF Authenticator Command response in the payload of the APDU Response.

The host shall set the INS byte to **"0x36"** for all UAF commands The SE shall read the UAF command number and data from the payload in the data part of the command.

The payload of the APDU command is encoded according to [UAFAuthnrCommands], the first 2 bytes of each command are the UAF command number. Upon command reception, the SE applet must parse the first TLV tag (2 bytes) and figure out which UAF command is being issued. The SE applet shall parse the rest of the FIDO Authenticator Command payload according to [UAFAuthnrCommands].

The mapping of UAF Authenticator Commands to APDU commands is defined in the following table:

CLA	INS	P1	P2	Lc	Data In	Le
Proprietary(See Table 2)	0x36	0x00	0x00	Variable	UAF Authenticator Command structure	None

Table 3: UAF APDU command

The UAF Authenticator Command structures are defined in part `6.2` of [UAFAuthnrCommands].

NOTE

If the `UserVerificationToken` is supported, The ASM must set the `TAG_USERVERIFY_TOKEN` flag in the value of the `UserVerificationToken`, received previously contained in either a `Register` or `Sign` command. Please refer to the **FIG 1** in Use-Case section.

4.2.2.2 Response message and status conditions of an "UAF" APDU command

The status word of an "UAF" APDU response is handled at the Host level; the host must interpret and map the status word based on the table below.

If the status word is equals to **"9000"**, the host shall return back to the ASM the entire data field of the APDU response. It the status word is "61xx", the host shall issue `GET RESPONSE` (see below) until no more data is available, concatenate these response parts and then return the entire response. Otherwise, the host has to build an UAF TLV response with the mapped status codes `TAG_STATUS_CODE`, using the following table.

For example, if the status word returned by the Applet is **"6A88"**, the host shall put `UAF_CMD_STATUS_USER_NOT_ENROLLED` in the status codes of the UAF TLV response.

APDU STATUS CODE	FIDO UAF STATUS CODE	NAME	DESCRIPTION
9000	0x00	UAF_CMD_STATUS_OK	Success.
61xx	0x00	UAF_CMD_STATUS_OK	Success, xx bytes available for GET RESPONSE.
6982	0x02	UAF_CMD_STATUS_ACCESS_DENIED	Access to this operation is denied.
6A88	0x03	UAF_CMD_STATUS_USER_NOT_ENROLLED	User is not enrolled with the authenticator.
N/A	0x04	UAF_CMD_STATUS_CANNOT_RENDER_TRANSACTION_CONTENT	Transaction content cannot be rendered.
N/A	0x05	UAF_CMD_STATUS_USER_CANCELLED	User has cancelled the operation.
6400	0x06	UAF_CMD_STATUS_CMD_NOT_SUPPORTED	Command not supported.
6A81	0x07	UAF_CMD_STATUS_ATTESTATION_NOT_SUPPORTED	Required attestation not supported.
6A80	0x08	UAF_CMD_STATUS_PARAMS_INVALID	The request was rejected due to an incorrect data field.
6983	0x09	UAF_CMD_STATUS_KEY_DISAPPEARED_PERMANENTLY	The UAuth key which is relevant for this command disappeared from the

			authenticator and cannot be restored.
N/A	0x0a	UAF_CMD_STATUS_TIMEOUT	The operation in the authenticator took longer than expected.
N/A	0x0e	UAF_CMD_STATUS_USER_NOT_RESPONSIVE	The user took too long to follow an instruction.
6A84	0x0f	UAF_CMD_STATUS_INSUFFICIENT_RESOURCES	Insufficient resources in the authenticator to perform the requested task.
63C0	0x10	UAF_CMD_STATUS_USER_LOCKOUT	The operation failed because the user is locked out and the authenticator cannot automatically trigger an action to change that.
All other codes	0x01	UAF_CMD_STATUS_ERR_UNKNOWN	An unknown error

Table 4: Mapping between APDU Status Codes and FIDO Status Codes [UAFAuthnrCommands]

The response message of an UAF APDU command is defined in the following table :

Data field	SW1 - SW2
not present	**“6982”** – The request was rejected due to user verification being required. **“6A80”** – The request was rejected due to an incorrect data field. **“6A81”** – Required attestation not supported **“6A88”** – The user is not enrolled with the SE **“6400”** – Execution error, undefined UAF command **“6983”** – Authentication data not usable, Auth key disappeared
UAF Authenticator Command response [UAFAuthnrCommands]	**“61xx”** – Success, xx bytes available for GET RESPONSE. **“9000”** – Success

Table 5: Response message of an "UAF" APDU command

4.2.3 APDU Command "SELECT"

A successful SELECT AID allows the host to know that the applet is active in the SE, and to open a logical channel with this end.

In Android smartphones apps are not allowed to use the basic channel to the SIM because this channel is reserved for the baseband processor and the GSM/UMTS/LTE activities. In this case the app must select the applet in a logical channel.

The host must send a `SELECT APDU` command to the SE applet before any others commands.

As a result, the command for selecting the applet using the FIDO UAF AID is :

CLA	INS	P1	P2	Lc	Data In	Le
0x00	0xA4	0x04	0x0C	0x08	0xA000000647AF0001	No response data is requested if the SELECT command's "Le" field is absent. Otherwise, if the "Le" field is present, vendor-proprietary data is being requested.

Table 6: SELECT AID command

4.2.4 APDU Command "VERIFY"

This command is used to request access rights using a PIN or Biometric sample. The SE applet shall verify the sample data given by the Host against the reference PIN or Biometric held in the SE.

Please refer to [ISOIEC-7816-4-2013] and [ISOIEC-19794] for Personal verification through biometric methods.

If the verification is successful and `UserVerificationToken` is supported by the SE applet, a token shall be generated and sent to the Host. Without having this token, the Host cannot invoke special UAF commands such as Register or Sign.

The support of `UserVerificationToken` can be checked by examining the contents of the `GetInfo` response in the `AuthenticatorType` TAG or the response of `SELECT APDU` command [UAFAuthnrCommands].

Refer to [FIDOGlossary] for more information about `UserVerificationToken`.

4.2.4.1 Command structure

CLA	INS	P1	P2	Lc	Data In	Le

ISO or Proprietary: see [ISOIEC-7816-4-2013]	0x20 (for PIN) or 0x21 (for biometry)	0x00	0x00	Variable	Verification data	None or expected Le for `UserVerificationToken`

Table 7: VERIFY command encoding for PIN verification

4.2.4.2 Response message and status conditions

Data Out	SW1 - SW2
Absent (ISO-Variant) or `UserVerificationToken` (proprietary)	See [ISOIEC-7816-4-2013]

Table 8: Response message and status conditions

NOTE

An SE applet that does not support `UserVerificationToken`, may use the [ISOIEC-7816-4-2013] VERIFY command. In this case, the VERIFY command must be securely bound to `Register` and `Sign` commands, so a secure bound method shall be implemented in the SE applet, such as Secure Messaging.

4.3 Managing Long APDU Commands and Responses

If a Secure Element is able to send a complete response (e.g. extended length APDU, block chaining), `GET RESPONSE` APDU command shall be used, as defined in `ISO Variant` section. Otherwise, the proprietary solution shall be used, as defined in section `Proprietary Variant`.

4.3.1 ISO Variant

The [ISOIEC-7816-4-2013] GET RESPONSE command is used in order to retrieve big data returned by APDU command "UAF".

4.3.2 Proprietary Variant

In order to avoid using Get Response APDU command which is not supported by all devices and terminals, a propriatry method is defined for managing the long data answers at application level.

When using the proprietary variant, the response to the UAF APDU command shall include the Tag **"0x2813"**, that specifies the length of the response.

Response Data Out description

Tag
: **0x2813**

Length
: variable (2 bytes)

Value
: Expected data length (2 bytes)

In the case where the data does not fit into a single Data Out message, the host shall repeat the "UAF" command with P2 = 1 value mentioning this is a repetition of the incoming APDU to get all the data. This process shall be repeated until the entire data are collected by the host.

Here is an example of an APDU Response which contains more than 255 bytes in the payload.

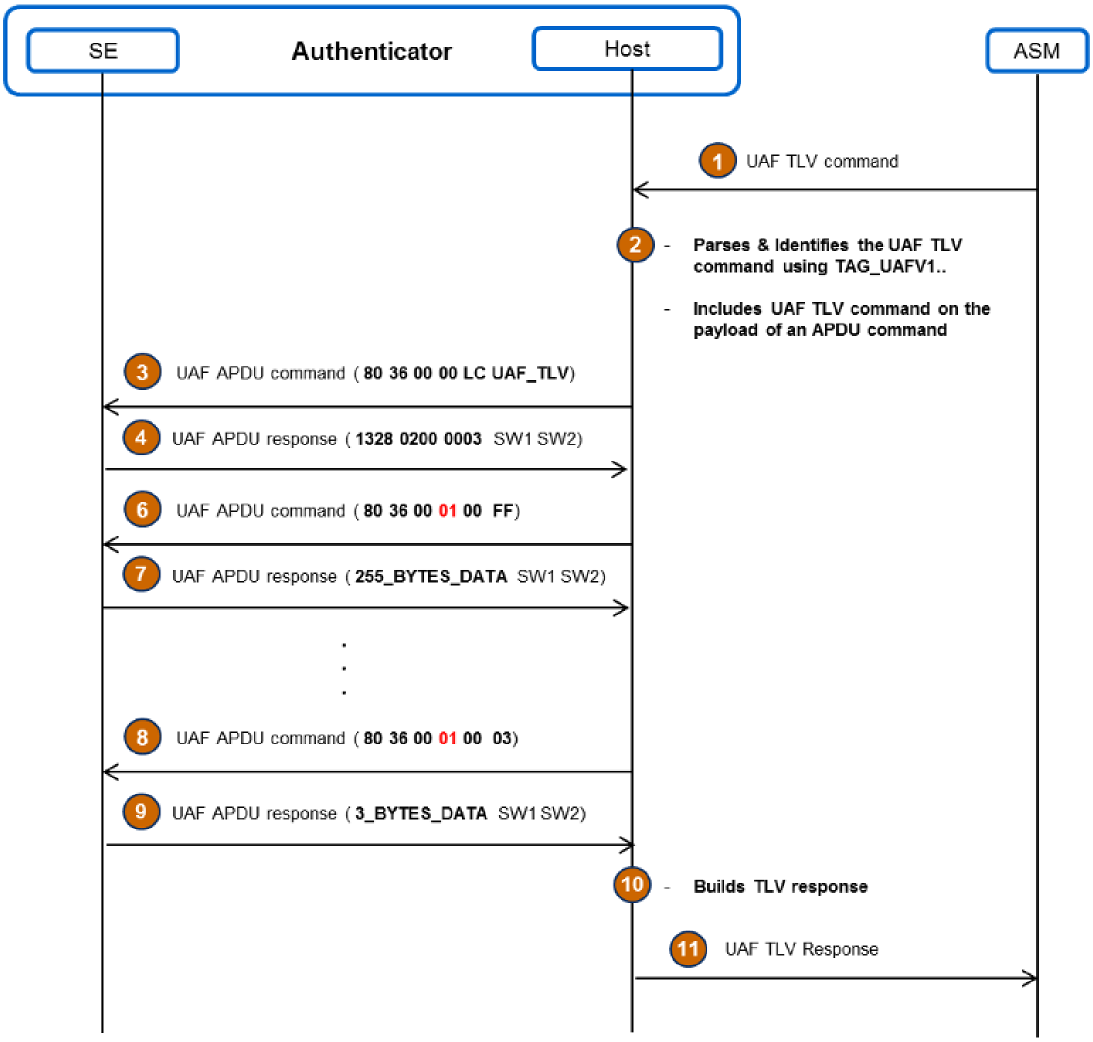


Fig. 3 Long APDU management using the defined proprietary method

> NOTE
>
> The host shall support both versions of Get Response APDU command, and figure out which command must be sent to the Applet by parsing the response of the UAF APDU command. If the UAF APDU command response contains the Tag **"0x2813"**, the host must send a proprietary Get Response APDU command, otherwise the host must send the ISO variant of Get Response APDU command.

5. Security considerations

This section is non-normative.

Guaranteeing trust and security in a fragmented architecture such as the one levering on SE is a challenge that the Host has to address regardless of its nature (TEE or Software based), which results in different challenges from a security and architecture perspective. One could list the following ones:

- use of a trusted user interface to enter a PIN on the device,
- secure transmission of PIN or fingerprint minutiae,
- minutiae extraction format,
- integrity of data transmitted between a Host and a SE.

Hence, we will only consider here, security challenges affecting the interface between the Host and the SE.

A possible way to maintain the integrity and confidentiality when APDUs commands are exchanged is to enable a secure channel between the Host and the SE. While this is left to implementation, there are several technologies allowing to build a secure channel between a SE and a devices, that may be implemented.

- Secure channel between a trusted application in a TEE and an applet in a SE [GlobalPlatform-TEE-SE].
- Secure channel between a device and an applet in a secure element [GlobalPlatform-Card].
- Secure channel between a device and a SE [ETSI-Secure-Channel].

A. References

A.1 Normative references

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

A.2 Informative references

[ETSI-Secure-Channel]
ETSI TS 102 484 Smart Cards; Secure channel between a UICC and an end-point terminal

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.pdf

[GlobalPlatform-Card]
Secure Channel Protocol 03 – GlobalPlatform Card Specification v.2.2 – Amendment D

[GlobalPlatform-TEE-SE]
TEE Secure Element API Specification v1.0 I GPD_SPE_024

[ISOIEC-19794]
ISO 19794: Information technology - Biometric data interchange formats

[ISOIEC-7816-4-2013]
ISO 7816-4: Identification cards – Integrated circuit cards; Part 4 : Organization, security and commands for interchange

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

[UAFASM]
D. Baghdasaryan, J. Kemp, R. Lindemann, B. Hill, R. Sasson, *FIDO UAF Authenticator-Specific Module API*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.pdf

[UAFAuthnrCommands]
D. Baghdasaryan, J. Kemp, R. Lindemann, R. Sasson, B. Hill, *FIDO UAF Authenticator Commands v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.pdf

[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.pdf

IMPLEMENTATION DRAFT

FIDO Metadata Statements

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statement-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-authnr-metadata-v1.0-ps-20141208.html

Editors:
Rolf Lindemann, Nok Nok Labs, Inc.
John Kemp, FIDO Alliance
Contributors:
Brad Hill, PayPal, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.

Abstract

FIDO authenticators may have many different form factors, characteristics and capabilities. This document defines a standard means to describe the relevant pieces of information about an authenticator in order to interoperate with it, or to make risk-based policy decisions about transactions involving a particular authenticator.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

DOM APIs are described using the ECMAScript [ECMA-262] bindings for WebIDL [WebIDL-ED].

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as required.

WebIDL dictionary members must not have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

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

> NOTE
>
> Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

The FIDO family of protocols enable simpler and more secure online authentication utilizing a wide variety of different devices in a competitive marketplace. Much of the complexity behind this variety is hidden from Relying Party applications, but in order to accomplish the goals of FIDO, Relying Parties must have some means of discovering and verifying various characteristics of authenticators. Relying Parties can learn a subset of verifiable information for authenticators certified by the FIDO Alliance with an Authenticator Metadata statement. The URL to access that Metadata statement is provided by the Metadata TOC file accessible through the Metadata Service [FIDOMetadataService].

For definitions of terms, please refer to the FIDO Glossary [FIDOGlossary].

2.1 Scope

This document describes the format of and information contained in *Authenticator Metadata* statements. For a definitive list of possible values for the various types of information, refer to the FIDO Registry of Predefined Values [FIDORegistry].

The description of the processes and methods by which authenticator metadata statements are distributed and the methods how these statements can be verified are described in the Metadata Service Specification [FIDOMetadataService].

2.2 Audience

The intended audience for this document includes:

- FIDO authenticator vendors who wish to produce metadata statements for their products.
- FIDO server implementers who need to consume metadata statements to verify characteristics of authenticators and attestation statements, make proper algorithm choices for protocol messages, create policy statements or tailor various other modes of operation to authenticator-specific characteristics.
- FIDO relying parties who wish to
  - create custom policy statements about which authenticators they will accept
  - risk score authenticators based on their characteristics
  - verify attested authenticator IDs for cross-referencing with third party metadata

2.3 Architecture

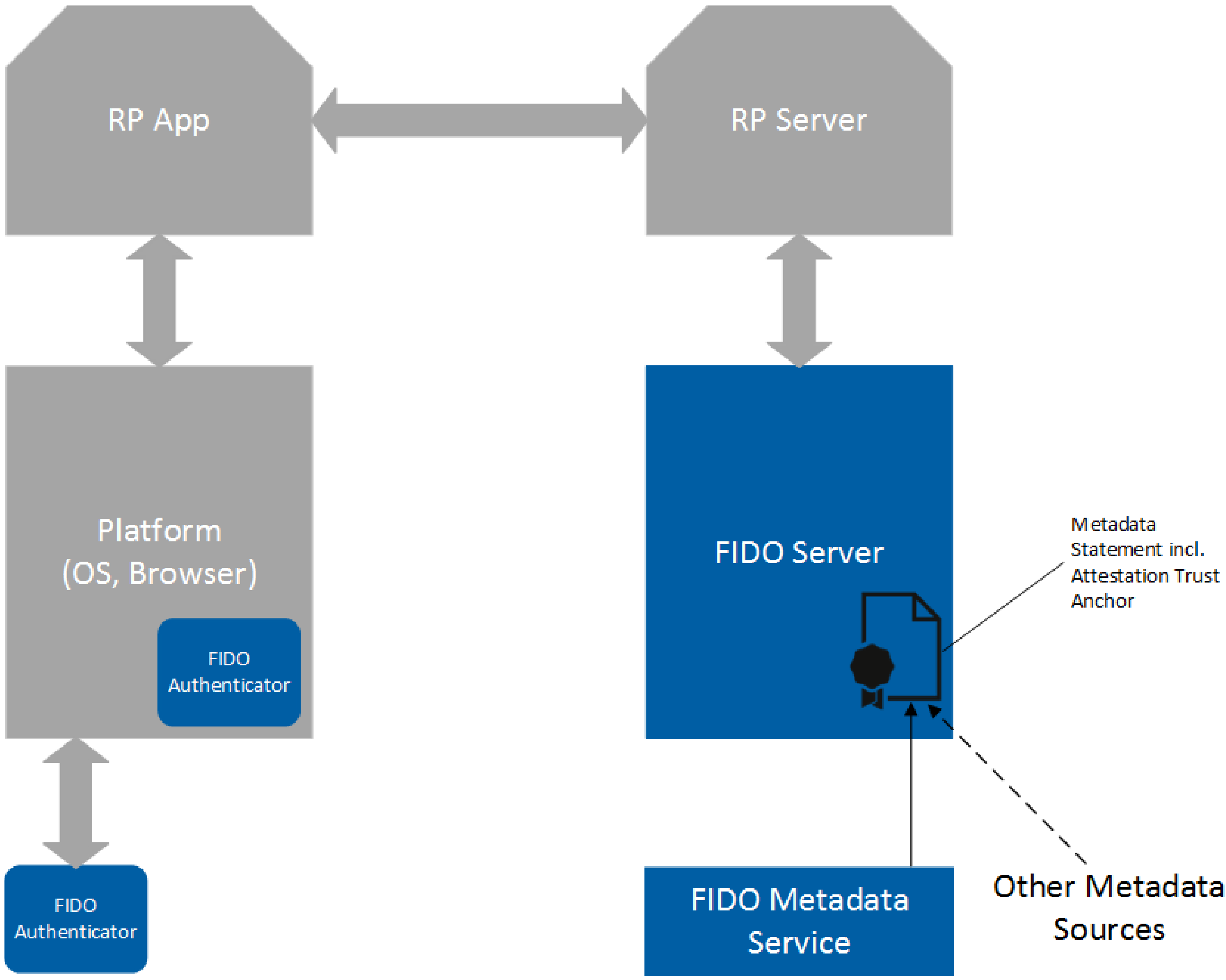


Fig. 1 The FIDO Architecture

Authenticator metadata statements are used directly by the FIDO server at a relying party, but the information contained in the authoritative statement is used in several other places. How a server obtains these metadata statements is described in

[FIDOMetadataService].

The workflow around an authenticator metadata statement is as follows:

1. The authenticator vendor produces a metadata statement describing the characteristics of an authenticator.
2. The metadata statement is submitted to the FIDO Alliance as part of the FIDO certification process. The FIDO Alliance distributes the metadata as described in [FIDOMetadataService].
3. A FIDO relying party configures its registration policy to allow authenticators matching certain characteristics to be registered.
4. The FIDO server sends a registration challenge message. This message can contain such policy statement.
5. Depending on the FIDO protocol being used, either the relying party application or the FIDO UAF Client receives the policy statement as part of the challenge message and processes it. It queries available authenticators for their self-reported characteristics and (with the user's input) selects an authenticator that matches the policy, to be registered.
6. The client processes and sends a registration response message to the server. This message contains a reference to the authenticator model and, optionally, a signature made with the private key corresponding to the public key in the authenticator's attestation certificate.
7. The FIDO Server looks up the metadata statement for the particular authenticator model. If the metadata statement lists an attestation certificate(s), it verifies that an attestation signature is present, and made with the private key corresponding to either (a) one of the certificates listed in this metadata statement or (b) corrsponding to the public key in a certificate that *chains* to one of the issuer certificates listed in the authenticator's metadata statement.
8. The FIDO Server next verifies that the authenticator meets the originally supplied registration policy based on its authoritative metadata statement. This prevents the registration of unexpected authenticator models.
9. *Optionally*, a FIDO Server may, with input from the Relying Party, assign a risk or trust score to the authenticator, based on its metadata, including elements not selected for by the stated policy.
10. *Optionally*, a FIDO Server may cross-reference the attested authenticator model with other metadata databases published by third parties. Such third-party metadata might, for example, inform the FIDO Server if an authenticator has achieved certifications relevant to certain markets or industry verticals, or whether it meets application-specific regulatory requirements.

3. Types

This section is normative.

3.1 CodeAccuracyDescriptor dictionary

The `CodeAccuracyDescriptor` describes the relevant accuracy/complexity aspects of passcode user verification methods.

> NOTE
>
> One example of such a method is the use of 4 digit PIN codes for mobile phone SIM card unlock.
>
> We are using the numeral system `base` (radix) and `minLen`, instead of the number of potential combinations since there is sufficient evidence [iPhonePasscodes] [MoreTopWorstPasswords] that users don't select their code evenly distributed at random. So software might take into account the various probability distributions for different bases. This essentially means that in practice, passcodes are not as secure as they could be if randomly chosen.

WebIDL

```
dictionary CodeAccuracyDescriptor {
    required unsigned short base;
    required unsigned short minLength;
    unsigned short          maxRetries;
    unsigned short          blockSlowdown;
};
```

3.1.1 Dictionary `CodeAccuracyDescriptor` Members

`base` of type required unsigned short
: The numeric system base (radix) of the code, e.g. 10 in the case of decimal digits.

`minLength` of type required unsigned short
: The minimum number of digits of the given base required for that code, e.g. 4 in the case of 4 digits.

`maxRetries` of type unsigned short
: Maximum number of false attempts before the authenticator will block this method (at least for some time). 0 means it will never block.

`blockSlowdown` of type unsigned short
: Enforced minimum number of seconds wait time after blocking (e.g. due to forced reboot or similar). 0 means this user verification method will be blocked, either permanently or until an alternative user verification method method succeeded. All alternative user verification methods must be specified appropriately in the Metadata in `userVerificationDetails`.

3.2 BiometricAccuracyDescriptor dictionary

The `BiometricAccuracyDescriptor` describes relevant accuracy/complexity aspects in the case of a biometric user verification method.

> NOTE
>
> The *False Acceptance Rate* (FAR) and *False Rejection Rate* (FRR) values typically are interdependent via the *Receiver Operator Characteristic* (ROC) curve.
>
> The *False Artefact Acceptance Rate* (FAAR) value reflects the capability of detecting presentation attacks, such as the detection of rubber finger presentation.
>
> The FAR, FRR, and FAAR values given here must reflect the actual configuration of the authenticators (as opposed to being theoretical best case values).

At least one of the values must be set. If the vendor doesn't want to specify such values, then `VerificationMethodDescriptor.baDesc` must be omitted.

> NOTE
>
> Typical fingerprint sensor characteristics can be found in Google Android 6.0 Compatibility Definition and Apple iOS Security Guide.

WebIDL

```
dictionary BiometricAccuracyDescriptor {
    double         FAR;
    double         FRR;
    double         EER;
    double         FAAR;
    unsigned short maxReferenceDataSets;
    unsigned short maxRetries;
    unsigned short blockSlowdown;
};
```

3.2.1 Dictionary `BiometricAccuracyDescriptor` Members

`FAR` of type double

The false acceptance rate [ISO19795-1] for a single reference data set, i.e. the percentage of non-matching data sets that are accepted as valid ones. For example a FAR of `0.002%` would be encoded as `0.00002`.

> NOTE
>
> The resulting FAR when all reference data sets are used is `maxReferenceDataSets * FAR`.
>
> The false acceptance rate is relevant for the security. Lower false acceptance rates mean better security.
>
> Only the live captured subjects are covered by this value - not the presentation of artefacts.

`FRR` of type double

The false rejection rate for a single reference data set, i.e. the percentage of presented valid data sets that lead to a (false) non-acceptance. For example a FRR of `10%` would be encoded as `0.1`.

> NOTE
>
> The false rejection rate is relevant for the convenience. Lower false acceptance rates mean better convenience.

`EER` of type double

The equal error rate for a single reference data set.

`FAAR` of type double

The false artefact acceptance rate [ISO30107-1], i.e. the percentage of artefacts that are incorrectly accepted by the system. For example a FAAR of `0.1%` would be encoded as `0.001`.

> NOTE
>
> The false artefact acceptance rate is relevant for the security of the system. Lower false artefact acceptance rates imply better security.

`maxReferenceDataSets` of type unsigned short

Maximum number of alternative reference data sets, e.g. 3 if the user is allowed to enroll 3 different fingers to a fingerprint based authenticator.

maxRetries of type unsigned short
Maximum number of false attempts before the authenticator will block this method (at least for some time). 0 means it will never block.

blockSlowdown of type unsigned short
Enforced minimum number of seconds wait time after blocking (e.g. due to forced reboot or similar). 0 means that this user verification method will be blocked either permanently or until an alternative user verification method succeeded. All alternative user verification methods must be specified appropriately in the metadata in `userVerificationDetails`.

3.3 PatternAccuracyDescriptor dictionary

The `PatternAccuracyDescriptor` describes relevant accuracy/complexity aspects in the case that a pattern is used as the user verification method.

> NOTE
>
> One example of such a pattern is the 3x3 dot matrix as used in Android [AndroidUnlockPattern] screen unlock. The `minComplexity` would be 1624 in that case, based on the user choosing a 4-digit PIN, the minimum allowed for this mechanism.

WebIDL

```
dictionary PatternAccuracyDescriptor {
    required unsigned long minComplexity;
    unsigned short         maxRetries;
    unsigned short         blockSlowdown;
};
```

3.3.1 Dictionary `PatternAccuracyDescriptor` Members

minComplexity of type required unsigned long
Number of possible patterns (having the minimum length) out of which exactly one would be the right one, i.e. 1/probability in the case of equal distribution.

maxRetries of type unsigned short
Maximum number of false attempts before the authenticator will block authentication using this method (at least temporarily). 0 means it will never block.

blockSlowdown of type unsigned short
Enforced minimum number of seconds wait time after blocking (due to forced reboot or similar mechanism). 0 means this user verification method will be blocked, either permanently or until an alternative user verification method method succeeded. All alternative user verification methods must be specified appropriately in the metadata under `userVerificationDetails`.

3.4 VerificationMethodDescriptor dictionary

A descriptor for a specific *base user verification method* as implemented by the authenticator.

A base user verification method must be chosen from the list of those described in [FIDORegistry]

> NOTE
>
> In reality, several of the methods described above might be combined. For example, a fingerprint based user verification can be combined with an alternative password.

The specification of the related AccuracyDescriptor is optional, but recommended.

WebIDL

```
dictionary VerificationMethodDescriptor {
    required unsigned long       userVerification;
    CodeAccuracyDescriptor       caDesc;
    BiometricAccuracyDescriptor  baDesc;
    PatternAccuracyDescriptor    paDesc;
};
```

3.4.1 Dictionary `VerificationMethodDescriptor` Members

userVerification of type required unsigned long
a *single* `USER_VERIFY` constant (see [FIDORegistry]), **not a bit flag combination**. This value must be non-zero.

caDesc of type *CodeAccuracyDescriptor*
May optionally be used in the case of method`USER_VERIFY_PASSCODE`.

baDesc of type *BiometricAccuracyDescriptor*
May optionally be used in the case of method `USER_VERIFY_FINGERPRINT`, `USER_VERIFY_VOICEPRINT`, `USER_VERIFY_FACEPRINT`, `USER_VERIFY_EYEPRINT`, or `USER_VERIFY_HANDPRINT`.

paDesc of type *PatternAccuracyDescriptor*
May optionally be used in case of method`USER_VERIFY_PATTERN`.

3.5 verificationMethodANDCombinations typedef

WebIDL

```
typedef VerificationMethodDescriptor[] VerificationMethodANDCombinations;
```

`VerificationMethodANDCombinations` must be non-empty. It is a list containing the base user verification methods which must be passed as part of a successful user verification.

This list will contain only a single entry if using a single user verification method is sufficient.

If this list contains multiple entries, then all of the listed user verification methods must be passed as part of the user verification process.

3.6 rgbPaletteEntry dictionary

The `rgbPaletteEntry` is an RGB three-sample tuple palette entry

WebIDL

```
dictionary rgbPaletteEntry {
    required unsigned short r;
    required unsigned short g;
    required unsigned short b;
};
```

3.6.1 Dictionary `rgbPaletteEntry` Members

r of type required unsigned short
Red channel sample value

g of type required unsigned short
Green channel sample value

b of type required unsigned short
Blue channel sample value

3.7 DisplayPNGCharacteristicsDescriptor dictionary

The DisplayPNGCharacteristicsDescriptor describes a PNG image characteristics as defined in the PNG [PNG] spec for IHDR (image header) and PLTE (palette table)

WebIDL

```
dictionary DisplayPNGCharacteristicsDescriptor {
    required unsigned long width;
    required unsigned long height;
    required octet         bitDepth;
    required octet         colorType;
    required octet         compression;
    required octet         filter;
    required octet         interlace;
    rgbPaletteEntry[]      plte;
};
```

3.7.1 Dictionary `DisplayPNGCharacteristicsDescriptor` Members

width of type required unsigned long
image width

height of type required unsigned long
image height

bitDepth of type required octet
Bit depth - bits per sample or per palette index.

colorType of type required octet
Color type defines the PNG image type.

compression of type required octet
: Compression method used to compress the image data.

filter of type required octet
: Filter method is the preprocessing method applied to the image data before compression.

interlace of type required octet
: Interlace method is the transmission order of the image data.

plte of type array of *rgbPaletteEntry*
: 1 to 256 palette entries

3.8 EcdaaTrustAnchor dictionary

In the case of ECDAA attestation, the ECDAA-Issuer's trust anchor must be specified in this field.

WebIDL

```
dictionary EcdaaTrustAnchor {
    required DOMString X;
    required DOMString Y;
    required DOMString c;
    required DOMString sx;
    required DOMString sy;
    required DOMString G1Curve;
};
```

3.8.1 Dictionary `EcdaaTrustAnchor` Members

X of type required DOMString
: base64url encoding of the result of ECPoint2ToB of the ECPoint2 \(X = P_2^x\). See [FIDOEcdaaAlgorithm] for the definition of ECPoint2ToB.

Y of type required DOMString
: base64url encoding of the result of ECPoint2ToB of the ECPoint2 \(Y = P_2^y\). See [FIDOEcdaaAlgorithm] for the definition of ECPoint2ToB.

c of type required DOMString
: base64url encoding of the result of BigNumberToB(\(c\)). See section "Issuer Specific ECDAA Parameters" in [FIDOEcdaaAlgorithm] for an explanation of \(c\). See [FIDOEcdaaAlgorithm] for the definition of BigNumberToB.

sx of type required DOMString
: base64url encoding of the result of BigNumberToB(\(sx\)). See section "Issuer Specific ECDAA Parameters" in [FIDOEcdaaAlgorithm] for an explanation of \(sx\). See [FIDOEcdaaAlgorithm] for the definition of BigNumberToB.

sy of type required DOMString
: base64url encoding of the result of BigNumberToB(\(sy\)). See section "Issuer Specific ECDAA Parameters" in [FIDOEcdaaAlgorithm] for an explanation of \(sy\). See [FIDOEcdaaAlgorithm] for the definition of BigNumberToB.

G1Curve of type required DOMString
: Name of the Barreto-Naehrig elliptic curve for G1. "BN_P256", "BN_P638", "BN_ISOP256", and "BN_ISOP512" are supported. See section "Supported Curves for ECDAA" in [FIDOEcdaaAlgorithm] for details.

> NOTE
>
> Whenever a party uses this trust anchor for the first time, it must first verify that it was correctly generated by verifying \(s, sx, sy\). See [FIDOEcdaaAlgorithm] for details.

3.9 ExtensionDescriptor dictionary

This descriptor contains an extension supported by the authenticator.

WebIDL

```
dictionary ExtensionDescriptor {
    required DOMString id;
    DOMString          data;
    required boolean   fail_if_unknown;
};
```

3.9.1 Dictionary `ExtensionDescriptor` Members

id of type required DOMString

: Identifies the extension.

data of type DOMString
: Contains arbitrary data further describing the extension and/or data needed to correctly process the extension.

This field may be missing or it may be empty.

fail_if_unknown of type required boolean
: Indicates whether unknown extensions must be ignored (`false`) or must lead to an error (`true`) when the extension is to be processed by the FIDO Server, FIDO Client, ASM, or FIDO Authenticator.

- A value of `false` indicates that unknown extensions must be ignored
- A value of `true` indicates that unknown extensions must result in an error.

4. Metadata Keys

This section is normative.

WebIDL

```
dictionary MetadataStatement {
    AAID                                         aaid;
    AAGUID                                       aaguid;
    DOMString[]                                  attestationCertificateKeyIdentifiers;
    required DOMString                           description;
    required unsigned short                      authenticatorVersion;
    DOMString                                    protocolFamily;
    required Version[]                           upv;
    required DOMString                           assertionScheme;
    required unsigned short                      authenticationAlgorithm;
    required unsigned short                      publicKeyAlgAndEncoding;
    required unsigned short[]                    attestationTypes;
    required VerificationMethodANDCombinations[] userVerificationDetails;
    required unsigned short                      keyProtection;
    boolean                                      isKeyRestricted;
    boolean                                      isFreshUserVerificationRequired;
    required unsigned short                      matcherProtection;
    required unsigned long                       attachmentHint;
    required boolean                             isSecondFactorOnly;
    required unsigned short                      tcDisplay;
    DOMString                                    tcDisplayContentType;
    DisplayPNGCharacteristicsDescriptor[]        tcDisplayPNGCharacteristics;
    required DOMString[]                         attestationRootCertificates;
    EcdaaTrustAnchor[]                           ecdaaTrustAnchors;
    DOMString                                    icon;
    ExtensionDescriptor                          supportedExtensions[];
};
```

4.1 Dictionary `MetadataStatement` Members

aaid of type AAID
: The Authenticator Attestation ID. See [UAFProtocol] for the definition of the AAID structure. This field must be set if the authenticator implements FIDO UAF.

> NOTE
>
> FIDO UAF Authenticators support AAID, but they don't support AAGUID.

aaguid of type AAGUID
: The Authenticator Attestation GUID. See [FIDOKeyAttestation] for the definition of the AAGUID structure. This field must be set if the authenticator implements FIDO 2.

> NOTE
>
> FIDO 2 Authenticators support AAGUID, but they don't support AAID.

attestationCertificateKeyIdentifiers of type array of DOMString
: A list of the attestation certificate public key identifiers encoded as hex string. This value must be calculated according to method 1 for computing the keyIdentifier as defined in [RFC5280] section 4.2.1.2. The hex string must not contain any non-hex characters (e.g. spaces). All hex letters must be lower case. This field must be set if neither `aaid` nor `aaguid` are set. Setting this field implies that the attestation certificate(s) are dedicated to a single authenticator model.

All attestationCertificateKeyIdentifier values should be unique within the scope of the Metadata Service.

> NOTE
>
> FIDO U2F Authenticators typically do not support AAID nor AAGUID, but they use attestation certificates

> dedicated to a single authenticator model.

description of type required DOMString
A human-readable short description of the authenticator.

> NOTE
>
> This description should help an administrator configuring authenticator policies. This description might deviate from the description returned by the ASM for that authenticator.
>
> This description should contain the public authenticator trade name and the publicly known vendor name.

authenticatorVersion of type required unsigned short
Earliest (i.e. lowest) trustworthy `authenticatorVersion` meeting the requirements specified in this metadata statement.

Adding new `StatusReport` entries with status `UPDATE_AVAILABLE` to the metadata `TOC` object [FIDOMetadataService] must also change this `authenticatorVersion` if the update fixes severe security issues, e.g. the ones reported by preceding `StatusReport` entries with status code `USER_VERIFICATION_BYPASS`, `ATTESTATION_KEY_COMPROMISE`, `USER_KEY_REMOTE_COMPROMISE`, `USER_KEY_PHYSICAL_COMPROMISE`, `REVOKED`.

It is recommended to assume increased risk if this version is higher (newer) than the firmware version present in an authenticator. For example, if a `StatusReport` entry with status `USER_VERIFICATION_BYPASS` or `USER_KEY_REMOTE_COMPROMISE` precedes the `UPDATE_AVAILABLE` entry, than any firmware version lower (older) than the one specified in the metadata statement is assumed to be vulnerable.

protocolFamily of type DOMString
The FIDO protocol family. The values "uaf", "u2f", and "fido2" are supported. If this field is missing, the assumed protocol family is "uaf". Metadata Statements for U2F authenticators must set the value of protocolFamily to "u2f" and FIDO 2.0 Authenticators implementations must set the value of protocolFamily to "fido2".

upv of type array of required Version
The FIDO unified protocol version(s) (related to the specific protocol family) supported by this authenticator. See [UAFProtocol] for the definition of the `Version` structure.

assertionScheme of type required DOMString
The assertion scheme supported by the authenticator. Must be set to one of the enumerated strings defined in the FIDO UAF Registry of Predefined Values [UAFRegistry] or to "FIDOV2" in the case of the FIDO 2 assertion scheme.

authenticationAlgorithm of type required unsigned short
The authentication algorithm supported by the authenticator. Must be set to one of the `ALG_` constants defined in the FIDO Registry of Predefined Values [FIDORegistry]. This value must be non-zero.

publicKeyAlgAndEncoding of type required unsigned short
The public key format used by the authenticator during registration operations. Must be set to one of the `ALG_KEY` constants defined in the FIDO Registry of Predefined Values [FIDORegistry]. Because this information is not present in APIs related to authenticator discovery or policy, a FIDO server must be prepared to accept and process any and all key representations defined for any public key algorithm it supports. This value must be non-zero.

attestationTypes of type array of required unsigned short
The supported attestation type(s). (e.g. `TAG_ATTESTATION_BASIC_FULL`) See Registry for more information [UAFRegistry].

userVerificationDetails of type array of required VerificationMethodANDCombinations
A list of *alternative* VerificationMethodANDCombinations. Each of these entries is one alternative user verification method. Each of these alternative user verification methods might itself be an "AND" combination of multiple modalities.

All effectively available alternative user verification methods must be properly specified here. A user verification method is considered effectively available if this method can be used to either:

- enroll new verification reference data to one of the user verification methods

  or

- unlock the UAuth key directly after successful user verification

keyProtection of type required unsigned short
A 16-bit number representing the bit fields defined by the `KEY_PROTECTION` constants in the FIDO Registry of Predefined Values [FIDORegistry].

This value must be non-zero.

> NOTE
>
> The keyProtection specified here denotes the effective security of the attestation key and Uauth private

> key and the effective trustworthiness of the attested attributes in the “sign assertion”. Effective security means that key extraction or injecting malicious attested attributes is only possible if the specified protection method is compromised. For example, if keyProtection=TEE is stated, it shall be impossible to extract the attestation key or the Uauth private key or to inject any malicious attested attributes *without breaking the TEE*.

`isKeyRestricted` of type boolean

This entry is set to `true`, if the Uauth private key is restricted by the *authenticator* to only sign valid FIDO signature assertions.

This entry is set to `false`, if the authenticator doesn't restrict the Uauth key to only sign valid FIDO signature assertions. In this case, the calling application could potentially get any hash value signed by the authenticator.

If this field is missing, the assumed value is isKeyRestricted=`true`

.

> NOTE
>
> Note that only in the case of isKeyRestricted=`true`, the FIDO server can trust a signature counter or transaction text to have been correctly processed/controlled by the authenticator.

`isFreshUserVerificationRequired` of type boolean

This entry is set to `true`, if Uauth key usage *always* requires a fresh user verification.

If this field is missing, the assumed value is isFreshUserVerificationRequired=`true`.

This entry is set to `false`, if the Uauth key can be used without requiring a fresh user verification, e.g. without any additional user interaction, if the user was verified a (potentially configurable) caching time ago.

In the case of isFreshUserVerificationRequired=`false`, the FIDO server must verify the registration response and/or authentication response and verify that the (maximum) caching time (sometimes also called "authTimeout") is acceptable.

This entry solely refers to the user verification. In the case of transaction confirmation, the authenticator must always ask the user to authorize the specific transaction.

> NOTE
>
> Note that in the case of isFreshUserVerificationRequired=`false`, the calling App could trigger use of the key without user involvement. In this case it is the responsibility of the App to ask for user consent.

`matcherProtection` of type required unsigned short

A 16-bit number representing the bit fields defined by the `MATCHER_PROTECTION` constants in the FIDO Registry of Predefined Values [FIDORegistry].

This value must be non-zero.

> NOTE
>
> If multiple matchers are implemented, then this value must reflect the *weakest* implementation of all matchers.
>
> The matcherProtection specified here denotes the effective security of the FIDO authenticator’s user verification. This means that a false positive user verification implies breach of the stated method. For example, if matcherProtection=TEE is stated, it shall be impossible to trigger use of the Uauth private key when bypassing the user verification *without breaking the TEE*.

`attachmentHint` of type required unsigned long

A 32-bit number representing the bit fields defined by the `ATTACHMENT_HINT` constants in the FIDO Registry of Predefined Values [FIDORegistry].

> NOTE
>
> The connection state and topology of an authenticator may be transient and cannot be relied on as authoritative by a relying party, but the metadata 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`.

`isSecondFactorOnly` of type required boolean
: Indicates if the authenticator is designed to be used only as a second factor, i.e. requiring some other authentication method as a first factor (e.g. username+password).

`tcDisplay` of type required unsigned short
: A 16-bit number representing a combination of the bit flags defined by the `TRANSACTION_CONFIRMATION_DISPLAY` constants in the FIDO Registry of Predefined Values [FIDORegistry].

This value must be 0, if transaction confirmation is not supported by the authenticator.

> NOTE
>
> The tcDisplay specified here denotes the effective security of the authenticator's transaction confirmation display. This means that only a breach of the stated method allows an attacker to inject transaction text to be included in the signature assertion which hasn't been displayed and confirmed by the user.

`tcDisplayContentType` of type DOMString
: Supported MIME content type [RFC2049] for the transaction confirmation display, such as `text/plain` or `image/png`.

This value must be present if transaction confirmation is supported, i.e. `tcDisplay` is non-zero.

`tcDisplayPNGCharacteristics` of type array of *DisplayPNGCharacteristicsDescriptor*
: A list of *alternative* DisplayPNGCharacteristicsDescriptor. Each of these entries is one alternative of supported image characteristics for displaying a PNG image.

This list must be present if PNG-image based transaction confirmation is supported, i.e. `tcDisplay` is non-zero and `tcDisplayContentType` is `image/png`.

`attestationRootCertificates` of type array of required DOMString
: Each element of this array represents a PKIX [RFC5280] trust root X.509 certificate that is valid for this authenticator model. Multiple certificates might be used for different batches of the same model. The array does not represent a certificate chain, but only the trust anchor of that chain.

Each array element is a base64-encoded (section 4 of [RFC4648]), DER-encoded [ITU-X690-2008] PKIX certificate value. Each element must be dedicated for authenticator attestation.

> NOTE
>
> A certificate listed here is a trust root. It might be the actual certificate presented by the authenticator, or it might be an issuing authority certificate from the vendor that the actual certificate in the authenticator chains to.
>
> In the case of "uaf" protocol family, the attestation certificate itself and the ordered certificate chain are included in the registration assertion (see [UAFAuthnrCommands]).

Either

1. the manufacturer attestation root certificate

   or

2. the root certificate dedicated to a specific authenticator model

must be specified.

In the case (1), the root certificate might cover multiple authenticator models. In this case, it must be possible to uniquely derive the authenticator model from the Attestation Certificate. When using AAID or AAGUID, this can be achieved by either specifying the AAID or AAGUID in the attestation certificate using the extension id-fido-gen-ce-aaid { 1 3 6 1 4 1 45724 1 1 1 } or id-fido-gen-ce-aaguid { 1 3 6 1 4 1 45724 1 1 4 } or - when neither AAID nor AAGUID are defined - by using the `attestationCertificateKeyIdentifier` method.

In the case (2) this is not required as the root certificate only covers a single authenticator model.

When supporting surrogate basic attestation only (see [UAFProtocol], section "Surrogate Basic Attestation"), no attestation root certificate is required/used. So this array must be empty in that case.

`ecdaaTrustAnchors` of type array of *EcdaaTrustAnchor*
: A list of trust anchors used for ECDAA attestation. This entry must be present if and only if attestationType includes TAG_ATTESTATION_ECDAA. The entries in `attestationRootCertificates` have no relevance for ECDAA attestation. Each ecdaaTrustAnchor must be dedicated to a single authenticator model (e.g as identified by its AAID/AAGUID).

`icon` of type DOMString
: A `data:` url [RFC2397] encoded PNG [PNG] icon for the Authenticator.

`supportedExtensions[]` of type *ExtensionDescriptor*
: List of extensions supported by the authenticator.

5. Metadata Statement Format

This section is non-normative.

NORMATIVE

A FIDO Authenticator Metadata Statement is a document containing a JSON encoded dictionary MetadataStatement.

5.1 UAF Example

Example of the metadata statement for an UAF authenticator with:

- authenticatorVersion 2.
- Fingerprint based user verification allowing up to 5 registered fingers, with false acceptance rate of 0.002% and rate limiting attempts for 30 seconds after 5 false trials.
- Authenticator is embedded with the FIDO User device.
- The authentication keys are protected by TEE and are restricted to sign valid FIDO sign assertions only.
- The (fingerprint) matcher is implemented in TEE.
- The Transaction Confirmation Display is implemented in a TEE.
- The Transaction Confirmation Display supports display of "image/png" objects only.
- Display has a width of 320 and a height of 480 pixel. A bit depth of 16 bits per pixel offering True Color (=Color Type 2). The zlib compression method (0). It doesn't support filtering (i.e. filter type of=0) and no interlacing support (interlace method=0).
- The Authentiator can act as first factor or as second factor, i.e. isSecondFactorOnly = false.
- It supports the "UAFV1TLV" assertion scheme.
- It uses the `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW` authentication algorithm.
- It uses the `ALG_KEY_ECC_X962_RAW` public key format (0x100=256 decimal).
- It only implements the `TAG_ATTESTATION_BASIC_FULL` method (0x3E07=15879 decimal).
- It implements UAF protocol version (upv) 1.0 and 1.1.

EXAMPLE 1: MetadataStatement for UAF Authenticator


```
{ "aaid": "1234#5678",
  "description": "FIDO Alliance Sample UAF Authenticator",
  "authenticatorVersion": 2,
  "upv": [{ "major": 1, "minor": 0 }, { "major": 1, "minor": 1 }],
  "assertionScheme": "UAFV1TLV",
  "authenticationAlgorithm": 1,
  "publicKeyAlgAndEncoding": 256,
  "attestationTypes": [15879],
  "userVerificationDetails": [ [ { "userVerification": 2, "baDesc":
            { "FAR": 0.00002, "maxRetries": 5, "blockSlowdown": 30, "maxReferenceDataSets": 5 } } ] ],
  "keyProtection": 6,
  "isKeyRestricted": true,
  "matcherProtection": 2,
  "attachmentHint": 1,
  "isSecondFactorOnly": "false",
  "tcDisplay": 5,
  "tcDisplayContentType": "image/png",
  "tcDisplayPNGCharacteristics": [{"width": 320, "height": 480, "bitDepth": 16,
        "colorType": 2, "compression": 0, "filter": 0, "interlace": 0}],
  "attestationRootCertificates": [
"MIICPTCCAeOgAwIBAgIJAOuexvU3Oy2wMAoGCCqGSM49BAMCMHsxIDAeBgNVBAMM
F1NhbXBsZSBBdHRlc3RhdGlvbiBSb290MRYwFAYDVQQKDA1GSURPIEFsbGlhbmNl
MREwDwYDVQQLDAhVQUYgVFdHLDESMBAGA1UEBwwJUGFsbyBBbHRvMQswCQYDVQQI
DAJDQTELMAkGA1UEBhMCVVMwHhcNMTQwNjE4MTMzMzMyWhcNNDExMTAzMTMzMzMy
WjB7MSAwHgYDVQQDDBdTYW1wbGUgQXR0ZXN0YXRpb24gUm9vdDEWMBQGA1UECgwN
RklETyBBbGxpYW5jZTERMA8GA1UECwwIVUFGIFRXRywxEjAQBgNVBAcMCVBhbG8g
QWx0bzELMAkGA1UECAwCQ0ExCzAJBgNVBAYTAlVTMFkwEwYHKoZIzj0CAQYIKoZI
zj0DAQcDQgAEH8hv2D0HXa59/BmpQ7RZehL/FMGzFd1QBg9vAUpOZ3ajnuQ94PR7
aMzH33nUSBr8fHYDrqOBb58pxGqHJRyX/6NQME4wHQYDVR0OBBYEFPoHA3CLhxFb
C0It7zE4w8hk5EJ/MB8GA1UdIwQYMBaAFPoHA3CLhxFbC0It7zE4w8hk5EJ/MAwG
A1UdEwQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIhAJ06QSXt9ihIbEKYKIjsPkri
VdLIgtfsbDSu7ErJfzr4AiBqoYCZf0+zI55aQeAHjIzA9Xm63rruAxBZ9ps9z2XN
lQ=="],
  "icon": "data:image/png;base64,
iVBORw0KGgoAAAANSUhEUgAAAE8AAAAvCAYAAACiwJfcAAAAAXNSR0IArs4c6QAAAARnQU1BAACx
jwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAAAahSURBVGhD7Zr5bxRlGMf9KzTB8AM/YEhE2W7p
QZcWKKBclSpHATlELARE7kNECCA3FkWK0CKKSCFIsKBcgVCDWGNESdAYidwgggJBiRiMhFc/4wy8
884zu9NdlnGTfZJP2n3nO++88933fveBBx+PqCzJkTUvBbLmpUDWvBTImpcCSZvXLCdX9R05Sk19
bb5atf599fG+/erA541q47aP1LLVa9SIyVNUi8Ii8d5kGTsi30NFv7ai9n7QZPMwbdys2erU2XMq
Udy8+ZcaNmGimE8yXN3RUd3a18nF0fUlovZ+0CTzWpd2Vj+eOm1bEyy6Dx4i5pUMGWveo506q227
dtuWBIuffr6oWpV0FPNLhow1751Nm21LvPH3rVtWjfz66Lfql8tX7FRl9YFSXsmSseb9ceOGbYk7
MNUcGPg8ZsbMe9rfQUaaV/JMX9sqdzDCSvp0kZHmTZg9x7bLHcMnThb16eJ+mVfQq8yaUZQNG64i
XZ+0/kq6uOZFO0QtatdWKfXnRQ99Bj91R5OIFnk54jN0mkUiqlO3XDW+Ml+98mKB6tW7rWpZcPc+
0zg4tLrYlUc86E6eGDjIMubVpcusearfgIYGRk6brhZVr/JcHzooL7550jedLExopWcApi2ZUqhu
7JLvrVsQU81zkzOPeemMRYvVuQsX7PbiDQY5JvZonftK+1VY8H9utx530h0ob+jmRYqj6ouaYvEe
nW/WlYjp8cwbMm682tPwqW1R4tj/2SH13IRJYl4moZvXpiSqDr7dXtQHxa/PK3/+BWsK1dTgHu6V
8tQJ3bwFkwpFrUOQ50s1r3levm8zZcq17+BBaw7K8lEK5qzkYeark9A8p7P3GzDK+nd3DQow+6UC
8SVN82iuv38im7NtaXtV1CVq6Rgw4pksmbdi3bu2De7YfaBBxcqfvqPrUjFQNTQ22lfdUVVT68rT
JKF5DnSmUjgdqg4mSS9pmsfDJR3G6ToH0iW9aV7LWLHYXKllTDt0LTAtkYIaamp1QjVv++uyGUxV
dJ0DNVXSm+b1qRxpl84ddfX1Lp1O/d69tsod0vs5hGre9xu8o+fpLR1cGhNTD6Z57C9KMWXefJdO
Z94bb9oqd1ROnS7qITTzHimMqivbO3g0DdVyk3WQBhBztK35YKNdOnc8O3acS6fDZFgKaXLsEJp5
rdrliBqp89cJcs/m7Tvs0rkjGfN4b0kPoZn3UJuIOrnZ22yP1fmvUx+O5gSqebV1m+zSuYNVhq7T
WbDiLVvljpl6CLXP+2qtvGLIL/1vimISdMBgzSoFZyu6Tqd+jzxgsPaV9BCqee/NjYk6v6lK
9cwiUc/STtf1HDpM3b592y7h3Thx5ozK69HHpYWuAwaqS5cv26q7ceb8+XsP/QUaReP3iFU8zj1knSw
```

```
ZXHMmnCjY0Ogalo7UQfSCM3qQQr2H/XFP7ssXx45Yl91ByeCep4moZoH+1fG3xD4tT7x8kwyj8nw
b9ev26V0B6d+7H4zKvudAH537FjqyzOHdJnHEuzmXq/WjxObvNMbv7nhywsX2aVsWtC8+48aLeap
E7p5wKZi0A2AQRV5nvR4E+uJc+b61kApqInxBgmd/4V5QP/mt18HDC7sRHftmeu5lmhV0rn/ALX2
32bqd4BFnDx7Vi1cWS2uff0IbB47qexxmUj9QutYjupd3tYD6abWBBMrh+apNbOKrNF1+ugCa4ri
XGfwMPPtViavhU3YMOAAnuUb/R07L0yOSeOadE88ApsXFGff30ynhlJgM51CU6vN9EzgnpvHBFUy
iVraePiwJ53DF5ZTZnomENg85kNUd2oJi2Wpr4OmmkfN4x4zHfiVFc8Dv8NzuhNqOidilGvA6DGu
eZwO78AAQn6ciEk6+rw5VcvjvqNDYPOoIUwaKShrxAuXLlkH4aYuGfMYDc10WF5Ta31hPJOfcUhr
U/JlINi6c6elRYdBpo6++Yfjx61lGNfRm4MD5rJ1j3FoGHnjDSBNarYUgMLyMszKpb7tXpoHfPs8
h3Wp1LzNfNk54XxC1wDGUmYzXYefh6z/cKtVm4EBxa9VQGDzYr3LrUMRjHEKkk7zaFKYQA2hGQU1
z+85NFWpXDrkz3vx10GqxQ6BzeNboBk5n8k4nebRh+k1hWfxTF0D1EyWUs5nv+dgQqKaxzuCdE0i
sHl02NQ8ah0mXr12La3m0f9wik9+wLNTMY/86MPo8yi31OfxmT6PWoqG9+DZukYna56mSZt5WWSy
5qVA1rwUyJqXAlnzkiai/gHSD7RkTyihogAAAABJRU5ErkJggg=="
}
```


Example of an *User Verification Methods* entry for an authenticator with:

- Fingerprint based user verification method, with:
  - the ability for the user to enroll up to 5 fingers (reference data sets) with
    - a false acceptance rate of 1 in 50000 (0.002%) per finger. This results in a FAR of 0.01% (0.0001).
    - The fingerprint verification will be blocked after 5 unsuccessful attempts.
- A PIN code with a minimum length of 4 decimal digits has to be set-up as alternative verification method. Entering the PIN will be required to re-activate fingerprint based user verification after it has been blocked.

EXAMPLE 2: User Verification Methods Entry


```
[
  [ { "userVerification": 2, "baDesc": { "FAR": 0.00002, "maxReferenceDataSets": 5,
                            "maxRetries": 5, "blockSlowdown": 0} }],
  [ { "userVerification": 4, "caDesc": { "base": 10, "minLength": 4 } } ]
]
```


5.2 U2F Example

Example of the metadata statement for an U2F authenticator with:

- authenticatorVersion 2.
- Touch based user presence check.
- Authenticator is a USB pluggable hardware token.
- The authentication keys are protected by a secure element.
- The user presence check is implemented in the chip.
- The Authentiator is a pure second factor authenticator.
- It supports the "U2FV1BIN" assertion scheme.
- It uses the `ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW` authentication algorithm.
- It uses the `ALG_KEY_ECC_X962_RAW` public key format (0x100=256 decimal).
- It only implements the `TAG_ATTESTATION_BASIC_FULL` method (0x3E07=15879 decimal).
- It implements U2F protocol version 1.0 only.

EXAMPLE 3: MetadataStatement for U2F Authenticator


```
{ "description": "FIDO Alliance Sample U2F Authenticator",
  "attestationCertificateKeyIdentifiers": ["7c0903708b87115b0b422def3138c3c864e44573"],
  "protocolFamily": "u2f",
  "authenticatorVersion": 2,
  "upv": [{ "major": 1, "minor": 0 }],
  "assertionScheme": "U2FV1BIN",
  "authenticationAlgorithm": 1,
  "publicKeyAlgAndEncoding": 256,
  "attestationTypes": [15879],
  "userVerificationDetails": [ [ { "userVerification": 1} ] ],
  "keyProtection": 10,
  "matcherProtection": 4,
  "attachmentHint": 2,
  "isSecondFactorOnly": "true",
  "tcDisplay": 0,
  "attestationRootCertificates": [
"MIICPTCCAeOgAwIBAgIJAOuexvU3Oy2wMAoGCCqGSM49BAMCMHsxIDAeBgNVBAMM
F1NhbXBsZSBBdHRlc3RhdGlvbiBSb290MRYwFAYDVQQKDA1GSURPIEFsbGlhbmNl
MREwDwYDVQQLDAhVQUYgVFdHLDESMBAGA1UEBwwJUGFsbyBBbHRvMQswCQYDVQQI
DAJDQTELMAkGA1UEBhMCVVMwHhcNMTQwNjE4MTMzMzMyWhcNNDExMTAzMTMzMzMy
WjB7MSAwHgYDVQQDDBdTYW1wbGUgQXR0ZXN0YXRpb24gUm9vdDEWMBQGA1UECgwN
RklETyBBbGxpYW5jZTERMA8GA1UECwwIVUFGIFRXRywxEjAQBgNVBAcMCVBhbG8g
QWx0bzELMAkGA1UECAwCQ0ExCzAJBgNVBAYTAlVTMFkwEwYHKoZIzj0CAQYIKoZI
zj0DAQcDQgAEH8hv2D0HXa59/BmpQ7RZehL/FMGzFd1QBg9vAUpOZ3ajnuQ94PR7
aMzH33nUSBr8fHYDrqOBb58pxGqHJRyX/6NQME4wHQYDVR0OBBYEFPoHA3CLhxFb
C0It7zE4w8hk5EJ/MB8GA1UdIwQYMBaAFPoHA3CLhxFbC0It7zE4w8hk5EJ/MAwG
A1UdEwQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIhAJ06QSXt9ihIbEKYKIjsPkri
VdLIgtfsbDSu7ErJfzr4AiBqoYCZf0+zI55aQeAHjIzA9Xm63rruAxBZ9ps9z2XN
lQ=="],
```


6. Additional Considerations

This section is non-normative.

6.1 Field updates and metadata

Metadata statements are intended to be stable once they have been published. When authenticators are updated in the field, such updates are expected to improve the authenticator security (for example, improve FRR or FAR). The `authenticatorVersion` must be updated if firmware updates fixing severe security issues (e.g. as reported previously) are available.

NOTE

The metadata statement is assumed to relate to all authenticators having the same AAID.

NOTE

The FIDO Server is recommended to assume increased risk if the`authenticatorVersion` specified in the metadata statement is newer (higher) than the one present in the authenticator.

NORMATIVE

Significant changes in authenticator functionality are not anticipated in firmware updates. For example, if an authenticator vendor wants to modify a PIN-based authenticator to use "Speaker Recognition" as a user verification method, the vendor must assign a new AAID to this authenticator.

NORMATIVE

A single authenticator implementation could report itself as two "virtual" authenticators using different AAIDs. Such implementations must properly (i.e. according to the security characteristics claimed in the metadata) protect `UAuth` keys and other sensitive data from the other "virtual" authenticator - just as a normal authenticator would do.

NOTE

Authentication keys (`UAuth.pub`) registered for one AAID cannot be used by authenticators reporting a different AAID - even when running on the same hardware (see section "Authentication Response Processing Rules for FIDO Server" in [UAFProtocol]).

A. References

A.1 Normative references

[ISO19795-1]
ISO/IEC JTC 1/SC 37, *Information Technology - Biometric peformance testing and reporting - Part 1: Principles and framework*, URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=41447

[ISO30107-1]
ISO/IEC JTC 1/SC 37, *Information Technology - Biometrics - Presentation attack detection - Part 1: Framework*, URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=53227

[RFC2049]
N. Freed, N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples (RFC 2049)*, IETF, November 1996, URL:http://www.ietf.org/rfc/rfc2049.txt

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

[RFC2397]
L. Masinter. *The "data" URL scheme*. August 1998. Proposed Standard. URL:https://tools.ietf.org/html/rfc2397

[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[AndroidUnlockPattern]
Android Unlock Pattern Security Analysis. Sinustrom.info web site. URL: http://www.sinustrom.info/2012/05/21/android-unlock-pattern-security-analysis/

[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[FIDOEcdaaAlgorithm]
R. Lindemann, J. Camenisch, M. Drijvers, A. Edgington, A. Lehmann, R. Urian,*FIDO ECDAA Algorithm*. FIDO Alliance Implementation Draft. URLs:
HTML: fido-ecdaa-v1.1-id-20170202.html
PDF: fido-ecdaa-v1.1-id-20170202.pdf.

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:

HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html

[FIDOMetadataService]
R. Lindemann, B. Hill, D. Baghdasaryan, *FIDO Metadata Service v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-service-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-service-v1.1-id-20170202.pdf

[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.pdf

[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). International Telecommunications Union, November 2008 URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en

[MoreTopWorstPasswords]
10000 Top Passwords, Mark Burnett (Accessed July 11, 2014) URL:https://xato.net/passwords/more-top-worst-passwords/

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

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

[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL:http://www.ietf.org/rfc/rfc5280.txt

[UAFAuthnrCommands]
D. Baghdasaryan, J. Kemp, R. Lindemann, R. Sasson, B. Hill,*FIDO UAF Authenticator Commands v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.pdf

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges,*FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

[UAFRegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO UAF Registry of Predefined Values*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-reg-v1.1-id-20170202.pdf

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Working Draft. URL: https://www.w3.org/TR/WebIDL-1/

[iPhonePasscodes]
Most Common iPhone Passcodes, Daniel Amitay (Accessed July 11, 2014) URL: http://danielamitay.com/blog/2011/6/13/most-common-iphone-passcodes

IMPLEMENTATION DRAFT

FIDO Metadata Service

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-service-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-metadata-service-v1.0-ps-20141208.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Brad Hill, PayPal, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.

Abstract

The FIDO Authenticator Metadata Specification defines so-called "Authenticator Metadata" statements. The metadata statements contain the "Trust Anchor" required to validate the attestation object, and they also describe several other important characteristics of the authenticator.

The metadata service described in this document defines a baseline method for relying parties to access the latest metadata statements.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

The notation `base64url(byte[8..64])` reads as 8-64 bytes of data encoded in base64url, "Base 64 Encoding with URL and Filename Safe Alphabet" [RFC4648] *without padding*.

Following [WebIDL-ED], dictionary members are optional unless they are explicitly marked as `required`.

WebIDL dictionary members must not have a value of null.

Unless otherwise specified, if a WebIDL dictionary member is DOMString, it must not be empty.

Unless otherwise specified, if a WebIDL dictionary member is a List, it must not be an empty list.

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

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

NOTE

Note: Certain dictionary members need to be present in order to comply with FIDO requirements. Such members are marked in the WebIDL definitions found in this document, as `required`. The keyword `required` has been introduced by [WebIDL-ED], which is a work-in-progress. If you are using a WebIDL parser which implements [WebIDL], then you may remove the keyword `required` from your WebIDL and use other means to ensure those fields are present.

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

[FIDOMetadataStatement] defines authenticator metadata statements.

These metadata statements contain the trust anchor required to verify the attestation object (more specifically the `KeyRegistrationData` object), and they also describe several other important characteristics of the authenticator, including supported authentication and registration assertion schemes, and key protection flags.

These characteristics can be used when defining policies about which authenticators are acceptable for registration or authentication.

The metadata service described in this document defines a baseline method for relying parties to access the latest metadata statements.

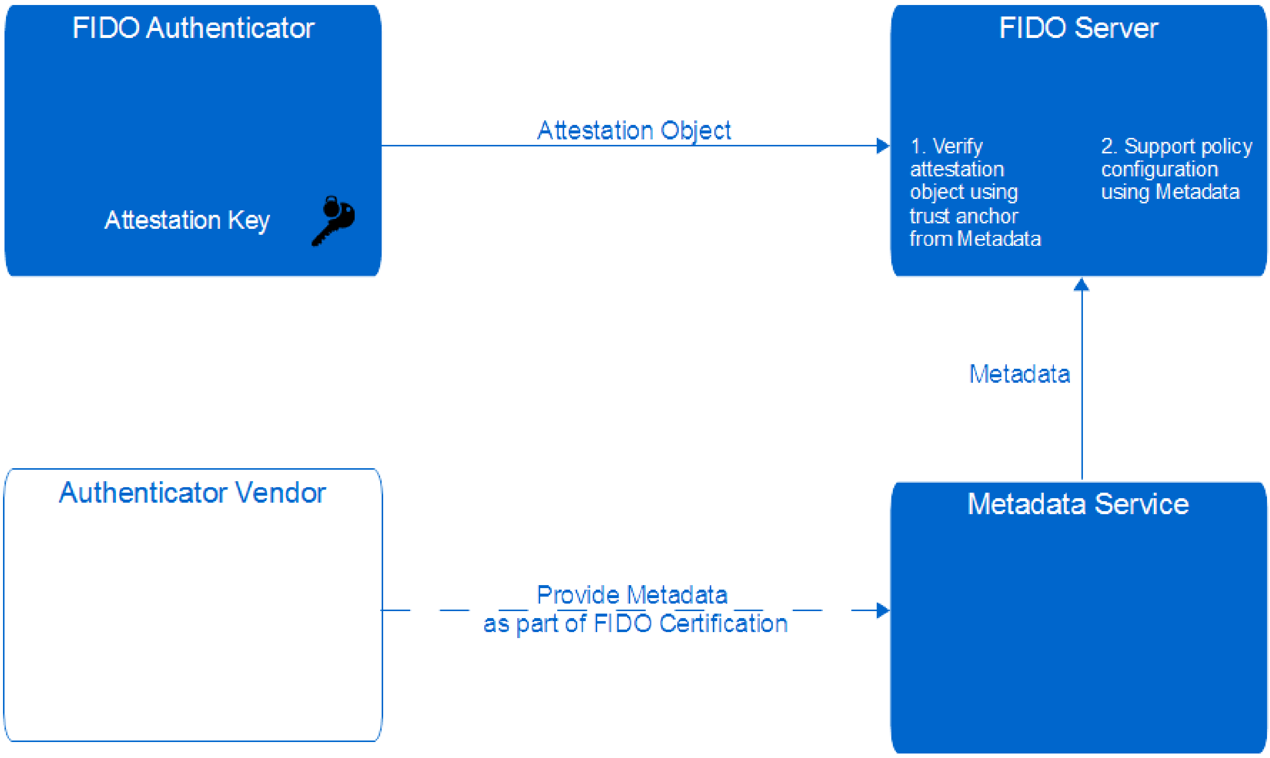


Fig. 1 FIDO Metadata Service Architecture Overview

2.1 Scope

This document describes the FIDO Metadata Service architecture in detail and it defines the structure and interface to access this service. It also defines the flow of the metadata related messages and presents the rationale behind the design choices.

2.2 Detailed Architecture

The metadata "table-of-contents" (TOC) file contains a list of metadata statements related to the authenticators known to the FIDO Alliance (FIDO Authenticators).

The FIDO Server downloads the metadata TOC file from a well-known FIDO URL and caches it locally.

The FIDO Server verifies the integrity and authenticity of this metadata TOC file using the digital signature. It then iterates through the individual entries and loads the metadata statements related to authenticator AAIDs relevant to the relying party.

Individual metadata statements will be downloaded from the URL specified in the entry of the metadata TOC file, and may be cached by the FIDO Server as required.

The integrity of the metadata statements will be verified by the FIDO Server using the hash value included in the related entry of the metadata TOC file.

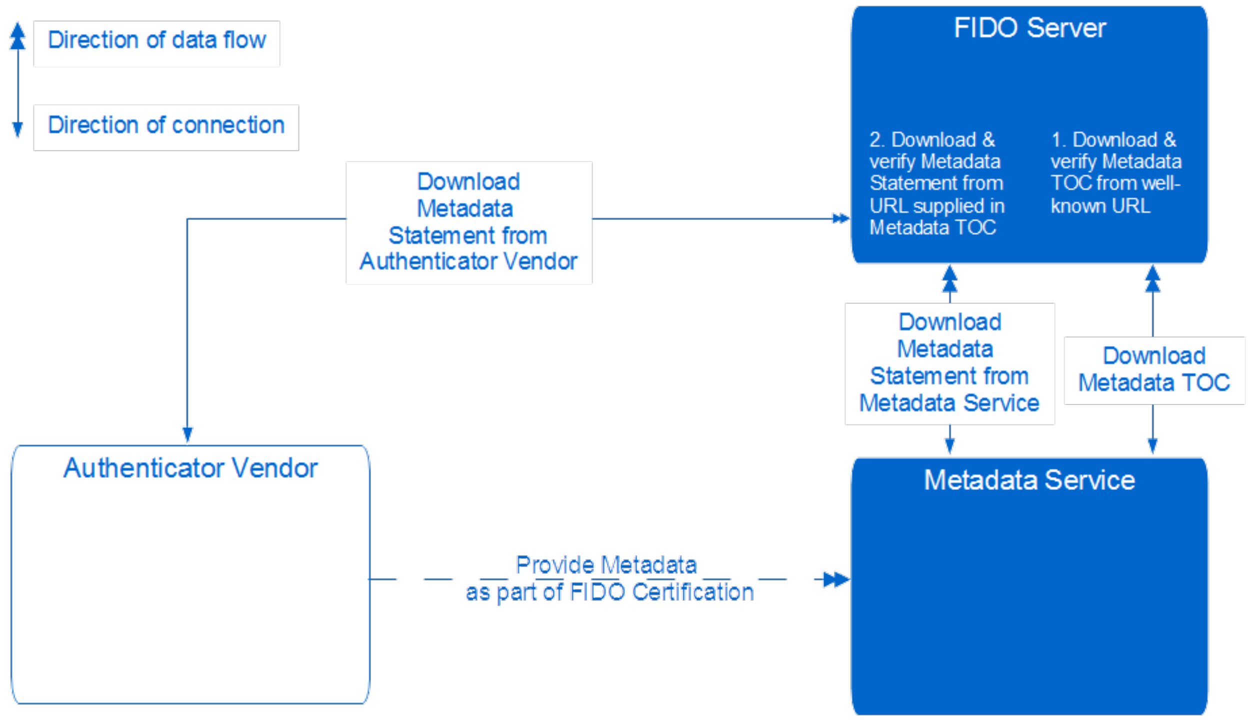


Fig. 2 FIDO Metadata Service Architecture

NOTE

The single arrow indicates the direction of the network connection, the double arrow indicates the direction of the data flow.

NOTE

The metadata TOC file is freely accessible at a well-known URL published by the FIDO Alliance.

NOTE

The relying party decides how frequently the metadata service is accessed to check for metadata TOC updates.

3. Metadata Service Details

This section is normative.

NOTE

The relying party can decide whether it wants to use the metadata service and whether or not it wants to accept certain authenticators for registration or authentication.

The relying party could also obtain metadata directly from authenticator vendors or other trusted sources.

3.1 Metadata TOC Format

NOTE

The metadata service makes the metadata TOC object (see Metadata TOC) accessible to FIDO Servers.

This object is a "table-of-contents" for metadata, as it includes the AAID, the download URL and the hash value of the individual metadata statements. The TOC object contains one signature.

3.1.1 Metadata TOC Payload Entry dictionary

Represents the MetadataTOCPayloadEntry

WebIDL

```
dictionary MetadataTOCPayloadEntry {
    AAID                      aaid;
    AAGUID                    aaguid;
    DOMString[]               attestationCertificateKeyIdentifiers;
    required DOMString        hash;
    required DOMString        url;
    required StatusReport[]   statusReports;
    required DOMString        timeOfLastStatusChange;
    DOMString                 rogueListURL;
    DOMString                 rogueListHash;
};
```

3.1.1.1 *Dictionary* `MetadataTOCPayloadEntry` *Members*

`aaid` of type AAID
: The AAID of the authenticator this metadata TOC payload entry relates to. See [UAFProtocol] for the definition of the AAID structure. This field must be set if the authenticator implements FIDO UAF.

> NOTE
>
> FIDO UAF authenticators support AAID, but they don't support AAGUID.

`aaguid` of type AAGUID
: The Authenticator Attestation GUID. See [FIDOKeyAttestation] for the definition of the AAGUID structure. This field must be set if the authenticator implements FIDO 2.

> NOTE
>
> FIDO 2 authenticators support AAGUID, but they don't support AAID.

`attestationCertificateKeyIdentifiers` of type array of DOMString
: A list of the attestation certificate public key identifiers encoded as hex string. This value must be calculated according to method 1 for computing the keyIdentifier as defined in [RFC5280] section 4.2.1.2. The hex string must not contain any non-hex characters (e.g. spaces). All hex letters must be lower case. This field must be set if neither `aaid` nor `aaguid` are set. Setting this field implies that the attestation certificate(s) are dedicated to a single authenticator model.

> NOTE
>
> FIDO U2F authenticators do not support AAID nor AAGUID, but they use attestation certificates dedicated to a single authenticator model.

`hash` of type required DOMString
: `base64url(string[1..512])`

The hash value computed over the base64url encoding of the UTF-8 representation of the JSON encoded metadata statement available at `url` and as defined in [FIDOMetadataStatement]. The hash algorithm related to the signature algorithm specified in the JWTHeader (see Metadata TOC) must be used.

> NOTE
>
> This method of base64url encoding the UTF-8 representation is also used by JWT [JWT] to avoid encoding ambiguities.

`url` of type required DOMString
: Uniform resource locator (URL) of the encoded metadata statement for this authenticator model (identified by its AAID, AAGUID or attestationCertificateKeyIdentifier). This URL must point to the base64url encoding of the UTF-8 representation of the JSON encoded metadata statement as defined in [FIDOMetadataStatement].

```
encodedMetadataStatement = base64url(utf8(JSONMetadataStatement))
```

> NOTE
>
> This method of the base64url encoding the UTF-8 representation is also used by JWT [JWT] to avoid encoding ambiguities.

`statusReports` of type array of required StatusReport
: An array of status reports applicable to this authenticator.

`timeOfLastStatusChange` of type required DOMString
: ISO-8601 formatted date since when the status report array was set to the current value.

`rogueListURL` of type DOMString
: URL of a list of rogue (i.e. untrusted) individual authenticators.

`rogueListHash` of type DOMString
: `base64url(string[1..512])`

The hash value computed over the Base64url encoding of the UTF-8 representation of the JSON encoded rogueList available at `rogueListURL` (with type rogueListEntry[]). The hash algorithm related to the signature algorithm specified in the JWTHeader (see Metadata TOC) must be used.

This hash value must be present and non-empty whenever `rogueListURL` is present.

> NOTE
>
> This method of base64url-encoding the UTF-8 representation is also used by JWT [JWT] to avoid encoding ambiguities.

EXAMPLE 1: UAF Metadata TOC Payload


```
{ "no": 1234, "nextUpdate": "2014-03-31",
  "entries": [
   { "aaid": "1234#5678",
     "hash": "90da8da6de23248abb34da0d4861f4b30a793e198a8d5baa7f98f260db71acd4",
     "url": "https://fidoalliance.org/metadata/1234%x23abcd",
     "rogueListHash": "b5079cf40fd7ed174c645cc04df1e72b7f1229590585d16df62dd20b9541c6b5",
     "rogueListURL": "https://fidoalliance.org/metadata/1234%x23abcd.rl",
     "statusReports": [
                         { status: "FIDO_CERTIFIED", effectiveDate: "2014-01-04"}
                     ],
     "timeOfLastStatusChange": "2014-01-04"
     },
   { "attestationCertificateKeyIdentifiers": ["7c0903708b87115b0b422def3138c3c864e44573"],
     "hash": "785d16df640fd7b50ed174cb5645cc0f1e72b7f19cf22959052dd20b9541c64d",
     "url": "https://authnr-vendor-a.com/metadata/9876%x234321",
     "statusReports": [
                         { status: "FIDO_CERTIFIED", effectiveDate: "2014-01-07"},
                         { status: "UPDATE_AVAILABLE", effectiveDate: "2014-02-19",
                           url: "https://example.com/update1234" }
                     ],
     "timeOfLastStatusChange": "2014-02-19"
     }
  ]
}
```


> NOTE
>
> The character `#` is a reserved character and not allowed in URLs [RFC3986]. As a consequence it has been replaced by its hex value `%x23`.
>
> The authenticator vendors can decide to let the metadata service publish its metadata statements or to publish metadata statements themselves. Authenticator vendors can restrict access to the metadata statements they publish themselves.

3.1.2 StatusReport dictionary

> NOTE
>
> Contains an `AuthenticatorStatus` and additional data associated with it, if any.
>
> New `StatusReport` entries will be added to report known issues present in firmware updates.

The latest `StatusReport` entry must reflect the "current" status. For example, if the latest entry has status `USER_VERIFICATION_BYPASS`, then it is recommended assuming an increased risk associated with all authenticators of this AAID; if the latest entry has status `UPDATE_AVAILABLE`, then the update is intended to address at least all previous issues *reported* in this StatusReport dictionary.

WebIDL

```
dictionary StatusReport {
    required AuthenticatorStatus status;
    DOMString                    effectiveDate;
    DOMString                    certificate;
    DOMString                    url;
};
```

3.1.2.1 Dictionary `StatusReport` Members

`status` of type required AuthenticatorStatus
Status of the authenticator. Additional fields may be set depending on this value.

`effectiveDate` of type DOMString
ISO-8601 formatted date since when the status code was set, if applicable. If no date is given, the status is assumed to be effective while present.

`certificate` of type DOMString
Base64-encoded [RFC4648] (not base64url!) DER [ITU-X690-2008] PKIX certificate value related to the current status, if applicable.

> NOTE
>
> As an example, this could be an Attestation Root Certificate (see [FIDOMetadataStatement]) related to a set of compromised authenticators (ATTESTATION_KEY_COMPROMISE).

`url` of type DOMString
HTTPS URL where additional information may be found related to the current status, if applicable.

> NOTE
>
> For example a link to a web page describing an available firmware update in the case of status `UPDATE_AVAILABLE`, or a link to a description of an identified issue in the case of status `USER_VERIFICATION_BYPASS`.

3.1.3 AuthenticatorStatus enum

This enumeration describes the status of an authenticator model as identified by its AAID and potentially some additional information (such as a specific attestation key).

WebIDL

```
enum AuthenticatorStatus {
    "NOT_FIDO_CERTIFIED",
    "FIDO_CERTIFIED",
    "USER_VERIFICATION_BYPASS",
    "ATTESTATION_KEY_COMPROMISE",
    "USER_KEY_REMOTE_COMPROMISE",
    "USER_KEY_PHYSICAL_COMPROMISE",
    "UPDATE_AVAILABLE",
    "REVOKED",
    "SELF_ASSERTION_SUBMITTED",
    "FIDO_SECURITY_CERTIFIED_L1",
    "FIDO_SECURITY_CERTIFIED_L2",
    "FIDO_SECURITY_CERTIFIED_L3",
    "FIDO_SECURITY_CERTIFIED_L4"
};
```

Enumeration description	
`NOT_FIDO_CERTIFIED`	This authenticator is not FIDO certified - no functional and no security certification.
`FIDO_CERTIFIED`	This authenticator has passed FIDO functional certification.
`USER_VERIFICATION_BYPASS`	Indicates that malware is able to bypass the user verification. This means that the authenticator could be used without the user's consent and potentially even

	without the user's knowledge.
ATTESTATION_KEY_COMPROMISE	Indicates that an attestation key for this authenticator is known to be compromised. Additional data should be supplied, including the key identifier and the date of compromise, if known.
USER_KEY_REMOTE_COMPROMISE	This authenticator has identified weaknesses that allow registered keys to be compromised and should not be trusted. This would include both, e.g. weak entropy that causes predictable keys to be generated or side channels that allow keys or signatures to be forged, guessed or extracted.
USER_KEY_PHYSICAL_COMPROMISE	This authenticator has known weaknesses in its key protection mechanism(s) that allow user keys to be extracted by an adversary in physical possession of the device.
UPDATE_AVAILABLE	A software or firmware update is available for the device. Additional data should be supplied including a URL where users can obtain an update and the date the update was published. When this code is used, then the field `authenticatorVersion` in the metadata Statement [FIDOMetadataStatement] must be updated, if the update fixes severe security issues, e.g. the ones reported by preceding StatusReport entries with status code `USER_VERIFICATION_BYPASS`, `ATTESTATION_KEY_COMPROMISE`, `USER_KEY_REMOTE_COMPROMISE`, `USER_KEY_PHYSICAL_COMPROMISE`, `REVOKED`. NOTE Relying parties might want to inform users about available firmware updates.
REVOKED	The FIDO Alliance has determined that this authenticator should not be trusted for any reason, for example if it is known to be a fraudulent product or contain a deliberate backdoor.
SELF_ASSERTION_SUBMITTED	The authenticator vendor has completed and submitted the self-certification checklist to the FIDO Alliance. If this completed checklist is publicly available, the URL will be specified in `StatusReport.url`.
FIDO_SECURITY_CERTIFIED_L1	The authenticator has passed a sanctioned third party security validation according to FIDO level 1.
FIDO_SECURITY_CERTIFIED_L2	The authenticator has passed a sanctioned third party security validation according to FIDO level 2.
FIDO_SECURITY_CERTIFIED_L3	The authenticator has passed a sanctioned third party security validation according to FIDO level 3.
FIDO_SECURITY_CERTIFIED_L4	The authenticator has passed a sanctioned third party security validation according to FIDO level 4.

More values might be added in the future. FIDO Servers must silently ignore all unknown AuthenticatorStatus values.

3.1.4 RogueListEntry dictionary

> NOTE
>
> Contains a list of individual authenticators known to be rogue.
>
> New `RogueListEntry` entries will be added to report new individual authenticators known to be rogue.
>
> Old `RogueListEntry` entries will be removed if the individual authenticator is known to not be rogue any longer.

WebIDL

```
dictionary RogueListEntry {
    required DOMString sk;
    required DOMString date;
};
```

3.1.4.1 Dictionary `RogueListEntry` Members

`sk` of type required DOMString
: Base64url encoding of the rogue authenticator's secret key (sk value, see [FIDOEcdaaAlgorithm], section ECDAA Attestation).

> NOTE
>
> In order to revoke an individual authenticator, its secret key (sk) must be known.

`date` of type required DOMString
ISO-8601 formatted date since when this entry is effective.

EXAMPLE 2: RogueListEntry[] example


```
[
   { "sk": "30efa86aa6de25249acb35da0d4861f4b30a793e198a8d5baa7e96f240da51f3",
     "date": "2016-06-07"},
   { "sk": "93de8da6de23248abb34da0d4861f4b30a793e153a8d5bb27f98f260db71acd4",
     "date": "2016-06-09"},
]
```


3.1.5 Metadata TOC Payload dictionary

Represents the MetadataTOCPayload

WebIDL

```
dictionary MetadataTOCPayload {
    required Number                     no;
    required DOMString                  nextUpdate;
    required MetadataTOCPayloadEntry[]  entries;
};
```

3.1.5.1 Dictionary `MetadataTOCPayload` Members

`no` of type required Number
The serial number of this UAF Metadata TOC Payload. Serial numbers must be consecutive and strictly monotonic, i.e. the successor TOC will have a `no` value exactly incremented by one.

`nextUpdate` of type required DOMString
ISO-8601 formatted date when the next update will be provided at latest.

`entries` of type array of required MetadataTOCPayloadEntry
List of zero or more MetadataTOCPayloadEntry objects.

3.1.6 Metadata TOC

The metadata table of contents (TOC) is a JSON Web Token (see [JWT] and [JWS]).

It consists of three elements:

- The base64url encoding, without padding, of the UTF-8 encoded JWT Header (see example below),
- the base64url encoding, without padding, of the UTF-8 encoded UAF Metadata TOC Payload (see example at the beginning of section Metadata TOC Format),
- and the base64url-encoded, also without padding, JWS Signature [JWS] computed over the to-be-signed payload, i.e.

```
tbsPayload = EncodedJWTHeader | "." | EncodedMetadataTOCPayload
```

All three elements of the TOC are concatenated by a period ("."):

```
MetadataTOC = EncodedJWTHeader | "." | EncodedMetadataTOCPayload | "." | EncodedJWSSignature
```

The hash algorithm related to the signing algorithm specified in the JWT Header (e.g. SHA256 in the case of "ES256") must also be used to compute the hash of the metadata statements (see section Metadata TOC Payload Entry Dictionary).

3.1.6.1 Examples

This section is non-normative.

EXAMPLE 3: Encoded Metadata Statement


```
eyAiQUFJRCI6ICIxMjM0IzU2NzgiLA0KICAiQXR0ZXN0YXRpb25Sb290Q2VydGlmaWNhdGUiOiAi
TUlJQ1BUQ0NBZU9nQXdJQkFnSUpBT3VleHZVM095MndNQ9HQ0NxR1NNNDlCQU1DTUhzeElEQWVC
```

```
Z05WQkFNTQ0KRjFOaGJYQnNaU0JCZEhSbGMzUmhkR2x2YmlCU2IyOTBNUll3RkFZRFZRUUtEQTFH
U1VSUElFRnNiR2xoYm1ObA0KTVJFd0R3WURWUVFMREFoVlFVWWdWRmRITERFU01CQUdBMVVFQnd3
SlVHRnNieUJCYkhSdk1Rc3dDUVlEVlFRSQ0KREFKRFFURUxNQWtHQTFVRUJoTUNWVk13SGhjTk1U
UXdOakU0TVRNek16TXlXaGNOTkRFeE1UQXpNVE16TXpNeQ0KV2pCN01TQXdIZ1lEVlFRRERCZFRZ
VzF3YkdVZ1FYUjBaWE4wWVhScGIyNGdVbTl2ZERFV01CUUdBMVVFQ2d3Tg0KUmtsRVR5QkJiR3hw
WVc1alpURVJNQThHQTFVRUN3d0lWVUZHSUZSWFJ5d3hFakFRQmdOVkJBY01DVkJoYkc4Zw0KUVd4
MGJ6RUxNQWtHQTFVRUNBd0NRMEV4Q3pBSkJnTlZCQVlUQWxWVE1Ga3dFd1lIS29aSXpqMENBUVlJ
S29aSQ0KemowREFRY0RRZ0FFSDhodjJEMEhYYTU5L0JtcFE3UlplaEwvRk1HekZkMVFCZzl2QVVw
T1ozYWpudVE5NFBSNw0KYU16SDMzblVTQnI4ZkhZRHJxT0JiNThweEdxSEpSeVgvNk5RTUU0d0hR
WURWUjBPQkJZRUZQb0hBM0NMaHhGYg0KQzBJdDd6RTR3OGhrNUVKL01COEdBMVVkSXdRWU1CYUFG
UG9IQTNDTGh4RmJDMEl0N3pFNHc4aGs1RUovTUF3Rw0KQTFVZEV3UUZNQU1CQWY4d0NnWUlLb1pJ
emowRUF3SURTQUF3UlFJaEFKMDZRU1h0OWloSWJFS1lLSWpzUGtyaQ0KVmRMSWd0ZnNiRFN1N0Vy
SmZ6cjRBaUJxb1lDWmYwK3pJNTVhUWVBSGpJekE5WG02M3JydUF4Qlo5cHM5ejJYTg0KbFE9PSIs
DQogICJEZXNjcmlwdGlvbiI6ICJGSURPIEFsbGlhbmNlIFNhbXBsZSBVQUYgQXV0aGVudGljYXRv
ciIsDQogICJVc2VyVmVyaWZpY2F0aW9uTWV0aG9kcyI6IDIsDQogICJWYWxpZEF0dGFjaG1lbnRU
eXBlcyI6IDEsDQogICJLZXlQcm90ZWN0aW9uIjogNiwNCiAgIk1hdGNoZXJQcm90ZWN0aW9uIjog
MiwNCiAgIlNlY3VyZURpc3BsYXkiOiA0LA0KICAiU2VjdXJlRGlzcGxheUNvbnRlbnRUeXBlcyI6
IFsiaW1hZ2UvcG5nIl0sDQogICJTZWN1cmVEaXNwbGF5UE5HQ2hhcmFjdGVyaXN0aWNzIjogW1sw
LDAsMSw2NCwwLDAsMSwyMjQsMTYsMiwwLDAsMF1dLA0KICAiaXNTZWNvbmRGYWN0b3JPbmx5Ijog
ImZhbHNlIiwNCiAgIkljb24iOiAiZGF0YTppbWFnZS9wbmc7YmFzZTY0LGlWQk9SdzBLR2dvQUFB
QU5TVWhFVWdBQUFFOEFBQUF2Q0FZQUFBQ2l3SmZjQUFBQUFYTlNSMElBcnM0YzZRQUFBQVJuUVUx
QkFBQ3gNCmp3djhZUVVBQUFBSmNFaFpjd0FBRHNNQUFBN0RBY2R2cUdRQUFBYWhTVVJCVkdoRDda
cjVieFJsR01mOUt6VEI4QU0vWUVoRTJXN3ANCjlFaY1dLS0JjbFNwSEFUbEVMQVJFN2tORUNTNGtRTNG
a1dLMENLS1NNDRklzS0JjZ1ZDRFFFdHkVTZEFaaWR3Z2dnSkpUmlNaEZjLzR3eTGNCjg4NHpp0U5k
bG5HVGZaSlAybUJjNuTysrODg5MzNmdmCQngrUHFDEekprVFV2QmJMMXBVRVFEd2QlRJbXBjQ1NadlhM
Q2RYOVIwNVNrMTkNCmJiNWF0ZjU5OWZHKy9lckE1NDFxNDdhhUDFMTFZhOVNJeVZ6OVZWk4SWk4ZDVr
R1RzaTMwTkZ2N2FpOW43UVpQQTXdiZHlzMmVyVTJYTENlClVkeVgrWmNhV0xDaX9ZUUKvUHREQXBwNU8dGM3a3
YTE4bkYwZlVsb3ZaaKzBDVHpXcGQyVmeorZU9tMWJGeXk2RHg0aVVWWVU1HV3ZlbzUwNnEyMjcNCmR0
dVdcSXVmZnI2b1dwVjBGUE5Ma0G93MTc1MU5TtMjFMd2lBIM3JWdFdqZno2Nkx4mcWw4dFG3RlJzOVlG
U1hzbVNzZWI5Y2VPR2JZazcNCk1OVWNHVGc0WnNiTWU5cmZRVWFhVi9KTVg5c3FFekekRDU3ZwMGta
SG1UWmc5eeDdiTEhjTW5UaGIxNmVKK21WZlFxOHlhVVpRTkc2NGkNClhaKzAva3E2dU9aRk8wVUXRh
dGRXS2ZYblJROTlCajkxUjVPSUZuazU0ak4wbWt4VaXFsTzNYRFcrTWwrOThtS0I2dFc3cldmNQ
YysNCjB6ZzR0THJZbFVjODZFNmVHRGpJTXViVnBjdXNlYXJmZ0lZR1JrNmJyaWFpcGci9KY0h6b29M
NzU1MGplZExxFeG9wV2NBcGkyWlVxaHUNCjdKTHZyVnNRVTgxemt6T1BlZW1NUll2VnVRc1g3UGJp
RFFZNUp2Wm9uZnRLKzFWWThIOXV0eDUzMGgwb2Iram1SWXFqNm91YVl2RWUNCm5XL1dsWWpwOGN3
Yk1tNjgydFB3cVcxUjR0ai8yU0gxM0lSSllsNG1vWnZYcGlTcURyN2RYdFFIeGEvUEszLytCV3NL
MWRUZ0h1NlYNCjh0UUozYndGa3dwRnJVT1E1MHMxcjNsZXZtOHpaY3ExNytCQmF3N0s4bEVLNXF6
a1llYXJrOUE4cDdQM0d6REsrbmQzRFFvdys2VUMNCjhTVk44Mml1djM4aW03TnRhWHRWMUNWcTZS
Z3c0cGtzbWJKaTNIZEdzZmFCQnhjWWZ2Vk2VDY0clQNCkpLRjVE
blNtVWpnZHFnNG1TUzlwbXNmREpSM0c2VG9IMGlXOWFWcjk4WDBaZWEtsbFREdDBMVEF0a11JYWFt
cDFRalZ2Kyt1eUdVeFYNCmRKMEROVlhTbStiMXFSeHBsODRkZGZYMUxwMU8vZDY5dHNvZDB2czVo
R3JlOXh1OG8rZnBMUjFjR0hOVEQ2WjU3QzFlTVd2ZWZKZE8NClo5NGJJOW9xZDFST25STTN5RjVR
SGltTXFpdmJPM2cwRGRWeszV1FCaEJ6dEszNVlMTmRPbmM4TzNhY1M2ZkRaRmdLYVhMc0VkcDUN
CnJkcmxpQnFwODljSmNzL203VHZzMHJhkdmTjRiMGtQb1puMVVKdUlPcm5aMjJ5JUDFmbXZVeCtP
NWdTcWViVmVqFtK3pTdVl0OVmhxN1QNCldiRGlMVnZsanBsTGxvcDZDTFhQKzJxdHZHTEE1MLzF2aW1J
U2RNQmd6U29GWnnl1NlRxZCtqenhnc0lBhVjlCQ3FlZS9aSGZ2bGVkNCljljd2lVVyy9TVHRmMUhE
cE0zYjU5Mnk3aDNUaHg1b3pLNjlITHBBZV3VBd2FxUzVjdjI2cTdjZWI4ZWZWWWFTZVAzaUZVOHpq
MWtuU3cNClpYSE1tbkQWTBPZ2FsbzdVUWZTQ00zcVFRcjJIL1hGUDdzc1h4NDVZbDkxQnl1Q2Vw
NG1vWm9IKzFmRzN4RDR0VDd4OGt3eWo4bncNCmI5ZXYyYnNlYmVaUTYzUnFYeE5BZEFHcUFTVVY0
ek9IZEpuUXlYeGtyUTdLdnFMSEVyajNndlRXTjVmZHpXQ1NqZGpzRDQwdmhaMS9pc1dGl
MEEyQVFFSVVjVnVkbI0RSt1SmMrYjFYxa0FwcUlueEJnbWQvNFY1UVAvbXQxOEhEQzdzUkhmdG11dTZs
bWhWMHJuL0FMWDINCjMyYnFkNEJGbkR4N1ppMWNXcUxnJ1ZmYwSWJCNDdxZXh4bnVVQVF1VF1dFlqdXBK
M3RzRDZhYldCQk1yaCthE5iT0t5TkYyK3VnQ2E0cmkNCkl1hHZnUNFdB0VmlhdmhhVM1lNT0FBbnVV
Yi9SMDdMMH1lPU2VPYWRFODhBcHNyYkdmZjMweW5obEpnTTUxQ1U2dk45RXpwnbnB2SEJGVXkNCmlW
cmFlUGl3SjUzREY1WlRabm9tRU5oODRDRmTvQY3ZENCSNVlVkMm9aTmFjVFNuRXNSTGpqaE1
enVoTnFPaWRpbEd2QTZER3UNCmVad083RUFBUW42Y21FazYrcnc1VmN2anZxTkRZRUE9vSVV3YUtT
aHJ4QXVYTGxSDRhWXVHZk1ZRGMxMFdGNVRhMzFoUUEpPZmNWaHINCmxVvSmxJTmk2YzZlbFJZZEJw
bzYrSzllbWang2MWxHTmZSbTRNRDVySjFqM0ZvR0huakRTQk5hcllZ01MeU1zektwWjjd0WHBvSGZQ
czgNCmgzV3AxVHhpOZk5rNTRYeEMxd0RHVW1ZelhhZWZoNnovY0t0Vm00RUJ4NYVl0WUdEEellyM0xy
VU1Sakh1FS2trN3phRktzUUEyaEdRVTENCnorODVORldRYWozb3pdngxMEdxeFE2QnplTmJvQms1l
bjhrNG5lYlJoK2sxaFdmeFRGMEQxRX1XVXM1bnYrZGdRcUtheHp1Q2RFMGkNCnNIbIDAyTlE4YWgw
bVhyMTJMYTNtMGY5d2lrOSt3TE5UTVkvODZNUG84eWkzMU9meG1UQ1NlQXh3FHOStEWnVrWW5hNTZt
U1p0NVdXU3kNCjVxVkxxcndVeUpxWEFsnbnpraWFwL2dIU0Q3UmtUeWlob2dBQUFBQkpSSVZFcmtK
Z2dnPT0iLA0KICAiQXNzZXJ0aW9uU2NoZW1lIjogIlVBRlYxVExWEXWIiwNCiAgIkF1dGhlbnRpY2F0
aW9uQWxnb3JpdGhtIjogMSwNCiAgIkF0dGVzdGF0aW9uVHlwZXMiOiBbMTYzOTFdLA0KICAiVVBW
IjogW1sxLDBdXQ0KfQ0K
```


EXAMPLE 4: JWT Header


```
{"typ":"JWT",
 "alg":"ES256"
 "x5t#S256":"7231962210d2933ec993a77b4a7203898ab74cdf974ff02d2de3f1ec7cb9de68"}
```


In order to produce the tbsPayload, we first need the base64url-encoded (without padding) JWT Header:

EXAMPLE 5: Encoded JWT Header


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IkVTMjU2IiwKICJ4NXQjUzI1NiI6IjcyMzE5NjIyMTBkMjkz
M2VjOTkzYTc3YjRhNzIwMzg5OGFiNzRjZGY5NzRmZjAyZDJkZTNmMWVjN2NiOWRlNjgifQ
```


then we have to append a period (".") and the base64url encoding of the `EncodedMetadataTOCPayload` (taken from the example in section Metadata TOC Format):

EXAMPLE 6: tbsPayload


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IkVTMjU2IiwKICJ4NXQjUzI1NiI6IjcyMzE5NjIyMTBkMjkz
M2VjOTkzYTc3YjRhNzIwMzg5OGFiNzRjZGY5NzRmZjAyZDJkZTNmMWVjN2NiOWRlNjgifQ.
eyAibm8iOiAxMjM0LCAibmV4dC11cGRhdGUiOiAiMzEtMDMtMjAxNCIsDQogICJlbnRyaWVzIjog
```

```
Ww0KICAgeyAiYWFpZCI6ICIxMjM0IzU2NzgiLCANCiAgICAgImhhc2giOiAiOTBkYThkYTZkZTIz
MjQ4YWJiMzRkYTBkNDg2MWY0YjMwYTc5M2UxOThhOGQ1YmFhN2Y5OGYyNjBkYjcxYWNkNCIsIA0K
ICAgICAidXJsIjogImh0dHBzOi8vZmlkb2FsbGlhbmNlLm9yZy9tZXRhZGF0YS8xMjM0JXgyM2Fi
Y2QiLCANCiAgICAgInN0YXR1cyI6ICJmaWRvQ2VydGlmaWVkIg0KICAgICAidGltZU9mTGFzdFN0
YXR1c0NoYW5nZSI6ICIiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAiMjAxNC0wMS0wNCIg
fSwNCiAgIHsgImFhaWQiOiAiOTg3NiM0MzIxIiwgDQogICAgICJoYXNoIjogIjc4NWQxNmRmNjQw
ZmQ3YjUwZWQxNzRjYjU2NDVjYzBmMWU3MmI3ZjE5Y2YyMjk1OTA1MmRkMjBiOTU0MWM2NGQiLA0K
ICAgICAidXJsIjogImh0dHBzOi8vYXV0aG5yLXZlbmRvci1hLmNvbS9tZXRhZGF0YS85ODc2JXgy
MzQzMjEiLA0KICAgICAic3RhdHVzIjogImZpZG9DZXJ0aWZpZWQiDQogICAgICJ0aW1lT2ZMYXN0
U3RhdHVzQ2hhbmdlIjogIjIwMTQtMDItMTkiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAi
MjAxNC0wMS0wNyIgfQ0KICBdDQp9DQo
```


and finally we have to append another period (".") followed by the base64url-encoded signature.

EXAMPLE 7: JWT


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IkVTMjU2IiwKICJ4NXQjUzI1NiI6IjcyMzE5NjIyMTBkMjkz
M2VjOTkzYTc3YjRhNzIwMzg5OGFiNzRjZGY5NzRmZjAyZDJkZTNmMWVjN2NiOWRlNjgifQ.
eyAibm8iOiAxMjM0LCAibmV4dC11cGRhdGUiOiAiMzEtMDMtMjAxNCIsDQogICJlbnRyaWVzIjog
Ww0KICAgeyAiYWFpZCI6ICIxMjM0IzU2NzgiLCANCiAgICAgImhhc2giOiAiOTBkYThkYTZkZTIz
MjQ4YWJiMzRkYTBkNDg2MWY0YjMwYTc5M2UxOThhOGQ1YmFhN2Y5OGYyNjBkYjcxYWNkNCIsIA0K
ICAgICAidXJsIjogImh0dHBzOi8vZmlkb2FsbGlhbmNlLm9yZy9tZXRhZGF0YS8xMjM0JXgyM2Fi
Y2QiLCANCiAgICAgInN0YXR1cyI6ICJmaWRvQ2VydGlmaWVkIg0KICAgICAidGltZU9mTGFzdFN0
YXR1c0NoYW5nZSI6ICIiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAiMjAxNC0wMS0wNCIg
fSwNCiAgIHsgImFhaWQiOiAiOTg3NiM0MzIxIiwgDQogICAgICJoYXNoIjogIjc4NWQxNmRmNjQw
ZmQ3YjUwZWQxNzRjYjU2NDVjYzBmMWU3MmI3ZjE5Y2YyMjk1OTA1MmRkMjBiOTU0MWM2NGQiLA0K
ICAgICAidXJsIjogImh0dHBzOi8vYXV0aG5yLXZlbmRvci1hLmNvbS9tZXRhZGF0YS85ODc2JXgy
MzQzMjEiLA0KICAgICAic3RhdHVzIjogImZpZG9DZXJ0aWZpZWQiDQogICAgICJ0aW1lT2ZMYXN0
U3RhdHVzQ2hhbmdlIjogIjIwMTQtMDItMTkiLA0KICAgICAiY2VydGlmaWNhdGlvbkRhdGUiOiAi
MjAxNC0wMS0wNyIgfQ0KICBdDQp9DQo.
AP-qoJ3VPzj7L6lCE1UzHzJYQnszFQ8d2hJz51sPASgyABK5VXOFnAHzBTQRRkgwGqULy6PtTyUV
zKxM0HrvoyZq
```


NOTE

The line breaks are for display purposes only.

The signature in the example above was computed with the following ECDSA key

EXAMPLE 8: ECDSA Key used for signature computation

```
x: d4166ba8843d1731813f46f1af32174b5c2f6013831fb16f12c9c0b18af3a9b4
y: 861bc2f803a2241f4939bd0d8ecd34e468e42f7fdccd424edb1c3ce7c4dd04e
d: 3744c426764f331f153e182d24f133190b6393cea480a8eec1c722fce161fe2d
```

3.1.7 Metadata TOC object processing rules

The FIDO Server must follow these processing rules:

1. The FIDO Server must be able to download the latest metadata TOC object from the well-known URL, when appropriate. The `nextUpdate` field of the Metadata TOC specifies a date when the download should occur at latest.
2. If the `x5u` attribute is present in the JWT Header, then:
    1. The FIDO Server must verify that the URL specified by the `x5u` attribute has the same web-origin as the URL used to download the metadata TOC from. The FIDO Server should ignore the file if the web-origin differs (in order to prevent loading objects from arbitrary sites).
    2. The FIDO Server must download the certificate (chain) from the URL specified by the `x5u` attribute [JWS]. The certificate chain must be verified to properly chain to the metadata TOC signing trust anchor according to [RFC5280]. All certificates in the chain must be checked for revocation according to [RFC5280].
    3. The FIDO Server should ignore the file if the chain cannot be verified or if one of the chain certificates is revoked.
3. If the `x5u` attribute is missing, the chain should be retrieved from the `x5c` attribute. If that attribute is missing as well, Metadata TOC signing trust anchor is considered the TOC signing certificate chain.
4. Verify the signature of the Metadata TOC object using the TOC signing certificate chain (as determined by the steps above). The FIDO Server should ignore the file if the signature is invalid. It should also ignore the file if its number (`no`) is less or equal to the number of the last Metadata TOC object cached locally.
5. Write the verified object to a local cache as required.
6. Iterate through the individual entries (of type `MetadataTOCPayloadEntry`). For each entry:
    1. Ignore the entry if the AAID, AAGUID or attestationCertificateKeyIdentifiers is not relevant to the relying party (e.g. not acceptable by any policy)
    2. Download the metadata statement from the URL specified by the field `url`. Some authenticator vendors might require authentication in order to provide access to the data. Conforming FIDO Servers should

support the HTTP Basic, and HTTP Digest authentication schemes, as defined in [RFC2617].

3. Check whether the status report of the authenticator model has changed compared to the cached entry by looking at the fields `timeOfLastStatusChange` and `statusReport`. Update the status of the cached entry. It is up to the relying party to specify behavior for authenticators with status reports that indicate a lack of certification, or known security issues. However, the status `REVOKED` indicates significant security issues related to such authenticators.

> NOTE
>
> Authenticators with an unacceptable status should be marked accordingly. This information is required for building registration and authentication policies included in the registration request and the authentication request [UAFProtocol].

4. Compute the hash value of the (base64url encoding without padding of the UTF-8 encoded) metadata statement downloaded from the URL and verify the hash value to the hash specified in the field `hash` of the metadata TOC object. Ignore the downloaded metadata statement if the hash value doesn't match.
5. Update the cached metadata statement according to the dowloaded one.

4. Considerations

This section is non-normative.

This section describes the key considerations for designing this metadata service.

Need for Authenticator Metadata When defining policies for acceptable authenticators, it is often better to describe the required authenticator characteristics in a generic way than to list individual authenticator AAIDs. The metadata statements provide such information. Authenticator metadata also provides the trust anchor required to verify attestation objects.

The metadata service provides a standardized method to access such metadata statements.

Integrity and Authenticity Metadata statements include information relevant for the security. Some business verticals might even have the need to document authenticator policies and trust anchors used for verifying attestation objects for auditing purposes.

It is important to have a strong method to verify and proof integrity and authenticity and the freshness of metadata statements. We are using a single digital signature to protect the integrity and authenticity of the Metadata TOC object and we protect the integrity and authenticity of the individual metadata statements by including their cryptographic hash values into the Metadata TOC object. This allows for flexible distribution of the metadata statements and the Metadata TOC object using standard content distribution networks.

Organizational Impact Authenticator vendors can delegate the publication of metadata statements to the metadata service in its entirety. Even if authenticator vendors choose to publish metadata statements themselves, the effort is very limited as the metadata statement can be published like a normal document on a website. The FIDO Alliance has control over the FIDO certification process and receives the metadata as part of that process anyway. With this metadata service, the list of known authenticators needs to be updated, signed and published regularly. A single signature needs to be generated in order to protect the integrity and authenticity of the metadata TOC object.

Performance Impact Metadata TOC objects and metadata statements can be cached by the FIDO Server.

The update policy can be specified by the relying party.

The metadata TOC object includes a date for the next scheduled update. As a result there is *no additional impact* to the FIDO Server during FIDO Authentication or FIDO Registration operations.

Updating the Metadata TOC object and metadata statements can be performed asynchronously. This reduces the availability requirements for the metadata service and the load for the FIDO Server.

The metadata TOC object itself is relatively small as it does not contain the individual metadata statements. So downloading the metadata TOC object does not generate excessive data traffic.

Individual metadata statements are expected to change less frequently than the metadata TOC object. Only the modified metadata statements need be downloaded by the FIDO Server.

Non-public Metadata Statements Some authenticator vendors might want to provide access to metadata statements only to their subscribed customers.

They can publish the metadata statements on access protected URLs. The access URL and the cryptographic hash of the metadata statement is included in the metadata TOC object.

High Security Environments Some high security environments might only trust internal policy authorities. FIDO Servers in such environments could be restricted to use metadata TOC objects from a proprietary trusted source only. The metadata service is the baseline for most relying parties.

Extended Authenticator Information Some relying parties might want additional information about authenticators before accepting them. The policy configuration is under control of the relying party, so it is possible to only accept authenticators for which additional data is available and meets the requirements.

A. References

A.1 Normative references

[FIDOMetadataStatement]
B. Hill, D. Baghdasaryan, J. Kemp, *FIDO Metadata Statements v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-metadata-statements.pdf

[JWS]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Signature (JWS)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7515

[JWT]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Token (JWT)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7519

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

[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL: http://www.ietf.org/rfc/rfc5280.txt

[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[FIDOEcdaaAlgorithm]
R. Lindemann, J. Camenisch, M. Drijvers, A. Edgington, A. Lehmann, R. Urian,*FIDO ECDAA Algorithm*. FIDO Alliance Implementation Draft. URLs:
HTML: fido-ecdaa-v1.1-id-20170202.html
PDF: fido-ecdaa-v1.1-id-20170202.pdf.

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html

[ITU-X690-2008]
X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), (T-REC-X.690-200811). International Telecommunications Union, November 2008 URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en

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

[RFC2617]
J. Franks; P. Hallam-Baker; J. Hostetler; S. Lawrence; P. Leach; A. Luotonen; L. Stewart*HTTP Authentication: Basic and Digest Access Authentication*. June 1999. Draft Standard. URL: https://tools.ietf.org/html/rfc2617

[RFC3986]
T. Berners-Lee; R. Fielding; L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges,*FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. *Web IDL*. 15 December 2016. W3C Working Draft. URL: https://www.w3.org/TR/WebIDL-1/

IMPLEMENTATION DRAFT

FIDO Registry of Predefined Values

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-registry-v1.1-id-20170202.html
Previous version:
https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal

Abstract

This document defines all the strings and constants reserved by FIDO protocols. The values defined in this document are referenced by various FIDO specifications.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

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

Some entries are marked as **"(optional)"** in this spec. The meaning of this is defined in other FIDO specifications referring to this document.

1.1 Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

This document defines the registry of FIDO-specific constants common to multiple FIDO protocol families. It is expected that, over time, new constants will be added to this registry. For example new authentication algorithms and new types of authenticator characteristics will require new constants to be defined for use within the specifications.

3. Authenticator Characteristics

This section is normative.

3.1 User Verification Methods

The `USER_VERIFY` constants are flags in a bitfield represented as a 32 bit long integer. They describe the methods and capabilities of an UAF authenticator for *locally* verifying a user. The operational details of these methods are opaque to the server. 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.

All user verification methods must be performed locally by the authenticator in order to meet FIDO privacy principles.

USER_VERIFY_PRESENCE 0x00000001
: This flag must 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 0x00000002
: This flag must be set if the authenticator uses any type of measurement of a fingerprint for user verification.

USER_VERIFY_PASSCODE 0x00000004
: This flag must be set if the authenticator uses a local-only passcode (i.e. a passcode not known by the server) for user verification.

USER_VERIFY_VOICEPRINT 0x00000008
: This flag must be set if the authenticator uses a voiceprint (also known as speaker recognition) for user verification.

USER_VERIFY_FACEPRINT 0x00000010
: This flag must be set if the authenticator uses any manner of face recognition to verify the user.

USER_VERIFY_LOCATION 0x00000020
: This flag must be set if the authenticator uses any form of location sensor or measurement for user verification.

USER_VERIFY_EYEPRINT 0x00000040
: This flag must be set if the authenticator uses any form of eye biometrics for user verification.

USER_VERIFY_PATTERN 0x00000080
: This flag must be set if the authenticator uses a drawn pattern for user verification.

USER_VERIFY_HANDPRINT 0x00000100
: This flag must be set if the authenticator uses any measurement of a full hand (including palm-print, hand geometry or vein geometry) for user verification.

USER_VERIFY_NONE 0x00000200
: This flag must be set if the authenticator will respond without any user interaction (e.g. Silent Authenticator).

USER_VERIFY_ALL 0x00000400
: If an authenticator sets multiple flags for user verification types, itmay also set this flag to indicate that all verification methods will be enforced (e.g. faceprint AND voiceprint). If flags for multiple user verification methods are set and this flag is not set, verification with only one is necessary (e.g. fingerprint OR passcode).

3.2 Key Protection Types

The `KEY_PROTECTION` constants are flags in a bit field represented as a 16 bit long integer. They describe the method an authenticator uses to protect the private key material for FIDO registrations. Refer to [UAFAuthnrCommands] for more details on the relevance of keys and key protection. 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* of others (i.e. can not be combined) - 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` or `KEY_PROTECTION_HARDWARE`.

> NOTE
>
> These flags must be set according to the *effective* security of the keys, in order to follow the assumptions made in [FIDOSecRef]. For example, if a key is stored in a secure element *but* software running on the FIDO User Device could call a function in the secure element to export the key either in the clear or using an arbitrary wrapping key, then the effective security is `KEY_PROTECTION_SOFTWARE` and not `KEY_PROTECTION_SECURE_ELEMENT`.

`KEY_PROTECTION_SOFTWARE` 0x0001
: This flag must 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` 0x0002
: This flag should be set if the authenticator uses hardware-based key management. Exclusive in authenticator metadata with `KEY_PROTECTION_SOFTWARE`

`KEY_PROTECTION_TEE` 0x0004
: This flag should be set if the authenticator uses the Trusted Execution Environment [TEE] for key management. In authenticator metadata, this flag should be set in conjunction with `KEY_PROTECTION_HARDWARE`. Mutually exclusive in authenticator metadata with `KEY_PROTECTION_SOFTWARE`, `KEY_PROTECTION_SECURE_ELEMENT`

`KEY_PROTECTION_SECURE_ELEMENT` 0x0008
: This flag should be set if the authenticator uses a Secure Element [SecureElement] for key management. In authenticator metadata, this flag should be set in conjunction with `KEY_PROTECTION_HARDWARE`. Mutually exclusive in authenticator metadata with `KEY_PROTECTION_TEE`, `KEY_PROTECTION_SOFTWARE`

`KEY_PROTECTION_REMOTE_HANDLE` 0x0010
: This flag must be set if the authenticator does not store (wrapped) UAuth 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 wrapping key and operations are protected. Servers may unset this flag in authenticator policy if they are not prepared 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. Refer to [UAFProtocol] for more details.

3.3 Matcher Protection Types

The `MATCHER_PROTECTION` constants are flags in a bit field represented as a 16 bit long integer. They describe the method an authenticator uses to protect the matcher that performs user verification. 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. Refer to [UAFAuthnrCommands] for more details on the matcher component.

> NOTE
>
> These flags must be set according to the *effective* security of the matcher, in order to follow the assumptions made in [FIDOSecRef]. For example, if a passcode based matcher is implemented in a secure element, but the passcode is expected to be

provided as unauthenticated parameter, then the effective security is MATCHER_PROTECTION_SOFTWARE and not MATCHER_PROTECTION_ON_CHIP.

MATCHER_PROTECTION_SOFTWARE 0x0001
: This flag must be set if the authenticator's matcher is running in software. Exclusive in authenticator metadata with MATCHER_PROTECTION_TEE, MATCHER_PROTECTION_ON_CHIP

MATCHER_PROTECTION_TEE 0x0002
: This flag should be set if the authenticator's matcher is running inside the Trusted Execution Environment [TEE]. Mutually exclusive in authenticator metadata with MATCHER_PROTECTION_SOFTWARE, MATCHER_PROTECTION_ON_CHIP

MATCHER_PROTECTION_ON_CHIP 0x0004
: This flag should be set if the authenticator's matcher is running on the chip. Mutually exclusive in authenticator metadata with MATCHER_PROTECTION_TEE, MATCHER_PROTECTION_SOFTWARE

3.4 Authenticator Attachment Hints

The ATTACHMENT_HINT constants are flags in a bit field represented as a 32 bit long. They describe the method an authenticator uses to communicate with the FIDO User Device. These constants are reported and queried through the UAF Discovery APIs [UAFAppAPIAndTransport], 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.

> NOTE
>
> These flags are not a mandatory part of authenticator metadata and, when present, only indicate possible states that may be reported during authenticator discovery.

ATTACHMENT_HINT_INTERNAL 0x0001
: This flag may be set to indicate that the authenticator is permanently attached to the FIDO User Device.

  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.

  This flag cannot be combined with any otherATTACHMENT_HINT flags.

ATTACHMENT_HINT_EXTERNAL 0x0002
: This flag may be set to indicate, for a hardware-based authenticator, that it is removable or remote from the FIDO User Device.

  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 UAF Client must filter and exclusively report only the relevant bit during discovery and when performing policy matching.

ATTACHMENT_HINT_WIRED 0x0004
: This flag may be set to indicate that an external authenticator currently has an exclusive wired connection, e.g. through USB, Firewire or similar, to the FIDO User Device.

ATTACHMENT_HINT_WIRELESS 0x0008
: This flag may be set to indicate that an external authenticator communicates with the FIDO User Device through a personal area or otherwise non-routed wireless protocol, such as Bluetooth or NFC.

ATTACHMENT_HINT_NFC 0x0010

This flag may be set to indicate that an external authenticator is able to communicate by NFC to the FIDO User Device. 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 as well.

`ATTACHMENT_HINT_BLUETOOTH 0x0020`
: This flag may be set to indicate that an external authenticator is able to communicate using Bluetooth with the FIDO User Device. 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 0x0040`
: This flag may be set to indicate that the authenticator is connected to the FIDO User Device over a non-exclusive network (e.g. over a TCP/IP LAN or WAN, as opposed to a PAN or point-to-point connection).

`ATTACHMENT_HINT_READY 0x0080`
: This flag may be set to indicate that an external authenticator is in a "ready" state. This flag is set by the ASM at its discretion.

> NOTE
>
> Generally this should indicate that the device is immediately available to perform user verification without additional actions such as connecting the device or creating a new biometric profile enrollment, but the exact meaning may vary for different types of devices. For example, a USB authenticator may only report itself as ready when it is plugged in, or a Bluetooth authenticator when it is paired and connected, but an NFC-based authenticator may always report itself as ready.

`ATTACHMENT_HINT_WIFI_DIRECT 0x0100`
: This flag may be set to indicate that an external authenticator is able to communicate using WiFi Direct with the FIDO User Device. As part of authenticator metadata and when reporting characteristics through discovery, if this flag is set, the `ATTACHMENT_HINT_WIRELESS` flag should also be set.

3.5 Transaction Confirmation Display Types

The `TRANSACTION_CONFIRMATION_DISPLAY` constants are flags in a bit field represented as a 16 bit long integer. They describe the availability and implementation of a transaction confirmation 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. Refer to [UAFAuthnrCommands] for more details on the security aspects of TransactionConfirmation Display.

`TRANSACTION_CONFIRMATION_DISPLAY_ANY 0x0001`
: This flag must be set to indicate that a transaction confirmation display, of any type, is available on this authenticator. Other `TRANSACTION_CONFIRMATION_DISPLAY` flags may also be set if this flag is set. If the authenticator does not support a transaction confirmation display, then the value of `TRANSACTION_CONFIRMATION_DISPLAY` must be set to 0.

`TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE 0x0002`
: This flag must be set to indicate, that a software-based transaction confirmation display operating in a privileged context is available on this authenticator.

A FIDO client that is capable of providing this capability may set this bit (in conjunction with `TRANSACTION_CONFIRMATION_DISPLAY_ANY`) for all authenticators of type `ATTACHMENT_HINT_INTERNAL`, even if the authoritative metadata for the authenticator does not indicate this capability.

> NOTE
>
> Software based transaction confirmation displays might be implemented within

> the boundaries of the ASM rather than by the authenticator itself [UAFASM].

This flag is mutually exclusive with TRANSACTION_CONFIRMATION_DISPLAY_TEE and TRANSACTION_CONFIRMATION_DISPLAY_HARDWARE.

TRANSACTION_CONFIRMATION_DISPLAY_TEE 0x0004
: This flag should be set to indicate that the authenticator implements a transaction confirmation display in a Trusted Execution Environment ([TEE], [TEESecureDisplay]). This flag is mutually exclusive with TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE and TRANSACTION_CONFIRMATION_DISPLAY_HARDWARE.

TRANSACTION_CONFIRMATION_DISPLAY_HARDWARE 0x0008
: This flag should be set to indicate that a transaction confirmation display based on hardware assisted capabilities is available on this authenticator. This flag is mutually exclusive with TRANSACTION_CONFIRMATION_DISPLAY_PRIVILEGED_SOFTWARE and TRANSACTION_CONFIRMATION_DISPLAY_TEE.

TRANSACTION_CONFIRMATION_DISPLAY_REMOTE 0x0010
: This flag should be set to indicate that the transaction confirmation display is provided on a distinct device from the FIDO User Device. This flag can be combined with any other flag.

3.6 Tags used for crypto algorithms and types

These tags indicate the specific authentication algorithms, public key formats and other crypto relevant data.

3.6.1 Authentication Algorithms

The `ALG_SIGN` constants are 16 bit long integers indicating the specific signature algorithm and encoding.

> NOTE
>
> FIDO UAF supports RAW and DER signature encodings in order to allow small footprint authenticator implementations.

ALG_SIGN_SECP256R1_ECDSA_SHA256_RAW 0x0001
: An ECDSA signature on the NIST secp256r1 curve which must have raw R and S buffers, encoded in big-endian order. This is the signature encoding as specified in [ECDSA-ANSI].

I.e. `[R (32 bytes), S (32 bytes)]`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW
- ALG_KEY_ECC_X962_DER

ALG_SIGN_SECP256R1_ECDSA_SHA256_DER 0x0002
: DER [ITU-X690-2008] encoded ECDSA signature [RFC5480] on the NIST secp256r1 curve.

I.e. a DER encoded `SEQUENCE { r INTEGER, s INTEGER }`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW

- ALG_KEY_ECC_X962_DER

ALG_SIGN_RSASSA_PSS_SHA256_RAW 0x0003

RSASSA-PSS [RFC3447] signature must have raw S buffers, encoded in big-endian order [RFC4055] [RFC4056]. The default parameters as specified in [RFC4055] must be assumed, i.e.

- Mask Generation Algorithm MGF1 with SHA256
- Salt Length of 32 bytes, i.e. the length of a SHA256 hash value.
- Trailer Field value of 1, which represents the trailer field with hexadecimal value `0xBC`.

I.e. `[ S (256 bytes) ]`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_RSA_2048_RAW
- ALG_KEY_RSA_2048_DER

ALG_SIGN_RSASSA_PSS_SHA256_DER 0x0004

DER [ITU-X690-2008] encoded OCTET STRING (not BIT STRING!) containing the RSASSA-PSS [RFC3447] signature [RFC4055] [RFC4056]. The default parameters as specified in [RFC4055] must be assumed, i.e.

- Mask Generation Algorithm MGF1 with SHA256
- Salt Length of 32 bytes, i.e. the length of a SHA256 hash value.
- Trailer Field value of 1, which represents the trailer field with hexadecimal value `0xBC`.

I.e. a DER encoded `OCTET STRING` (including its tag and length bytes).

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_RSA_2048_RAW
- ALG_KEY_RSA_2048_DER

ALG_SIGN_SECP256K1_ECDSA_SHA256_RAW 0x0005

An ECDSA signature on the secp256k1 curve which must have raw R and S buffers, encoded in big-endian order.

I.e.`[R (32 bytes), S (32 bytes)]`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW
- ALG_KEY_ECC_X962_DER

ALG_SIGN_SECP256K1_ECDSA_SHA256_DER 0x0006

DER [ITU-X690-2008] encoded ECDSA signature [RFC5480] on the secp256k1 curve.

I.e. a DER encoded `SEQUENCE { r INTEGER, s INTEGER }`

This algorithm is suitable for authenticators using the following key representation formats:

- ALG_KEY_ECC_X962_RAW

- ALG_KEY_ECC_X962_DER

`ALG_SIGN_SM2_SM3_RAW 0x0007` **(optional)**
: Chinese SM2 elliptic curve based signature algorithm combined with SM3 hash algorithm [OSCCA-SM2][OSCCA-SM3]. We use the 256bit curve [OSCCA-SM2-curve-param].

  This algorithm is suitable for authenticators using the following key representation format: ALG_KEY_ECC_X962_RAW.

`ALG_SIGN_RSA_EMSA_PKCS1_SHA256_RAW 0x0008`
: This is the EMSA-PKCS1-v1_5 signature as defined in [RFC3447]. This means that the encoded message EM will be the input to the cryptographic signing algorithm RSASP1 as defined in [RFC3447]. The result s of RSASP1 is then encoded using function I2OSP to produce the raw signature octets.

  - `EM = 0x00 | 0x01 | PS | 0x00 | T`
  - with the padding string PS with length=emLen - tLen - 3 octets having the value 0xff for each octet, e.g. `(0x) ff ff ff ff ff ff ff ff`
  - with the DER [ITU-X690-2008] encoded DigestInfo value T: `(0x)30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20 | H`, where H denotes the bytes of the SHA256 hash value.

  This algorithm is suitable for authenticators using the following key representation formats:

  - ALG_KEY_RSA_2048_RAW
  - ALG_KEY_RSA_2048_DER

  NOTE

  Implementers should verify that their implementation of the PKCS#1 V1.5 signature follows the recommendations in [RFC3218] to protect against adaptive chosen-ciphertext attacks such as Bleichenbacher.

`ALG_SIGN_RSA_EMSA_PKCS1_SHA256_DER 0x0009`
: DER [ITU-X690-2008] encoded OCTET STRING (not BIT STRING!) containing the EMSA-PKCS1-v1_5 signature as defined in [RFC3447]. This means that the encoded message EM will be the input to the cryptographic signing algorithm RSASP1 as defined in [RFC3447]. The result s of RSASP1 is then encoded using function I2OSP to produce the raw signature. The raw signature is DER [ITU-X690-2008] encoded as an OCTET STRING to produce the final signature octets.

  - `EM = 0x00 | 0x01 | PS | 0x00 | T`
  - with the padding string PS with length=emLen - tLen - 3 octets having the value 0xff for each octet, e.g. `(0x) ff ff ff ff ff ff ff ff`
  - with the DER encoded DigestInfo value T:`(0x)30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20 | H`, where H denotes the bytes of the SHA256 hash value.

  This algorithm is suitable for authenticators using the following key representation formats:

  - ALG_KEY_RSA_2048_RAW
  - ALG_KEY_RSA_2048_DER

  NOTE

> Implementers should verify that their implementation of the PKCS#1 V1.5 signature follows the recommendations in [RFC3218] to protect against adaptive chosen-ciphertext attacks such as Bleichenbacher.

3.6.2 Public Key Representation Formats

The `ALG_KEY` constants are 16 bit long integers indicating the specific Public Key algorithm and encoding.

> NOTE
>
> FIDO UAF supports RAW and DER encodings in order to allow small footprint authenticator implementations. By definition, the authenticator must encode the public key as part of the registration assertion.

`ALG_KEY_ECC_X962_RAW 0x0100`

Raw ANSI X9.62 formatted Elliptic Curve public key [SEC1].

I.e. `[0x04, X (32 bytes), Y (32 bytes)]`. Where the byte `0x04` denotes the uncompressed point compression method.

`ALG_KEY_ECC_X962_DER 0x0101`

DER [ITU-X690-2008] encoded ANSI X.9.62 formatted `SubjectPublicKeyInfo` [RFC5480] specifying an elliptic curve public key.

I.e. a DER encoded `SubjectPublicKeyInfo` as defined in [RFC5480].

Authenticator implementations must generate `namedCurve` in the `ECParameters` object which is included in the `AlgorithmIdentifier`. A FIDO UAF Server must accept `namedCurve` in the `ECParameters` object which is included in the `AlgorithmIdentifier`.

`ALG_KEY_RSA_2048_RAW 0x0102`

Raw encoded 2048-bit RSA public key [RFC3447].

That is, `[n (256 bytes), e (N-256 bytes)]`. Where `N` is the total length of the field.

This total length should be taken from the object containing this key, e.g. the TLV encoded field.

`ALG_KEY_RSA_2048_DER 0x0103`

ASN.1 DER [ITU-X690-2008] encoded 2048-bit RSA [RFC3447] public key [RFC4055].

That is a DER encoded `SEQUENCE { n INTEGER, e INTEGER }`.

A. References

A.1 Normative references

[FIDOGlossary]

R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[ITU-X690-2008]

X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules

(DER), (T-REC-X.690-200811). International Telecommunications Union, November 2008 URL: http://www.itu.int/rec/T-REC-X.690-200811-I/en

[OSCCA-SM2]
SM2: Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves: Part 1: General. December 2010. URL: http://www.oscca.gov.cn/UpFile/2010122214822692.pdf

[OSCCA-SM2-curve-param]
SM2: Elliptic Curve Public-Key Cryptography Algorithm: Recommended Curve Parameters. December 2010. URL: http://www.oscca.gov.cn/UpFile/2010122214836668.pdf

[OSCCA-SM3]
SM3 Cryptographic Hash Algorithm. December 2010. URL: http://www.oscca.gov.cn/UpFile/20101222141857786.pdf

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

[RFC3447]
J. Jonsson; B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. February 2003. Informational. URL: https://tools.ietf.org/html/rfc3447

[RFC4055]
J. Schaad; B. Kaliski; R. Housley. *Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. June 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc4055

[RFC4056]
J. Schaad. *Use of the RSASSA-PSS Signature Algorithm in Cryptographic Message Syntax (CMS)*. June 2005. Proposed Standard. URL:https://tools.ietf.org/html/rfc4056

[RFC5480]
S. Turner; D. Brown; K. Yiu; R. Housley; T. Polk. *Elliptic Curve Cryptography Subject Public Key Information*. March 2009. Proposed Standard. URL: https://tools.ietf.org/html/rfc5480

[SEC1]
Standards for Efficient Cryptography Group (SECG), *SEC1: Elliptic Curve Cryptography*, Version 2.0, September 2000.

A.2 Informative references

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

[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.pdf

[RFC3218]
E. Rescorla. *Preventing the Million Message Attack on Cryptographic Message Syntax* January 2002. Informational. URL: https://tools.ietf.org/html/rfc3218

[SecureElement]
GlobalPlatform Card Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp

[TEE]
GlobalPlatform Trusted Execution Environment Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp

[TEESecureDisplay]
GlobalPlatform Trusted User Interface API Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp

[UAFASM]
D. Baghdasaryan, J. Kemp, R. Lindemann, B. Hill, R. Sasson, *FIDO UAF Authenticator-*

Specific Module API. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-asm-api-v1.1-id-20170202.pdf

[UAFAppAPIAndTransport]
B. Hill, D. Baghdasaryan, B. Blanke, *FIDO UAF Application API and Transport Binding Specification*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-client-api-transport-v1.1-id-20170202.pdf

[UAFAuthnrCommands]
D. Baghdasaryan, J. Kemp, R. Lindemann, R. Sasson, B. Hill, *FIDO UAF Authenticator Commands v1.0*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.pdf

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges, *FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

IMPLEMENTATION DRAFT

FIDO AppID and Facet Specification

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-appid-and-facets-v1.1-id-20170202.html
Previous version:
http://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-appid-and-facets-v1.0-ps-20141208.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Brad Hill, PayPal, Inc.
Dirk Balfanz, Google, Inc.
Davit Baghdasaryan, Nok Nok Labs, Inc.

Abstract

The FIDO family of protocols introduce a new security concept, *Application Facets*, to describe the scope of user credentials and how a trusted computing base which supports application isolation may make access control decisions about which keys can be used by which applications and web origins.

This document describes the motivations for and requirements for implementing the Application Facet concept and how it applies to the FIDO protocols.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

This document applies to both the U2F protocol and the UAF protocol. UAF specific terminology used in this document is defined in [FIDOGlossary].

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

1.1 Key Words

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

Modern networked applications typically present several ways that a user can interact with them. This document introduces the concept of an *Application Facet* to describe the identities of a single logical application across various platforms. For example, the application MyBank may have an Android app, an iOS app, and a Web app accessible from a browser. These are all facets of the MyBank application.

The FIDO architecture provides for simpler and stronger authentication than traditional username and password approaches while avoiding many of the shortfalls of alternative authentication schemes. At the core of the FIDO protocols are challenge and response operations performed with a public/private keypair that serves as a user's credential.

To minimize frequently-encountered issues around privacy, entanglements with concepts of "identity", and the necessity for trusted third parties, keys in FIDO are tightly scoped and dynamically provisioned between the user and each Relying Party and only optionally associated with a server-assigned username. This approach contrasts with, for example, traditional PKIX client certificates as used in TLS, which introduce a trusted third party, mix in their implementation details identity assertions with holder-of-key cryptographic proofs, lack audience restrictions, and may even be sent in the cleartext portion of a protocol handshake without the user's notification or consent.

While the FIDO approach is preferable for many reasons, it introduces several challenges.

- What set of Web origins and native applications (facets) make up a single logical application and how can they be reliably identified?
- How can we avoid making the user register a new key for each web browser or application on their device that accesses services controlled by the same target entity?
- How can access to registered keys be shared without violating the security guarantees around application isolation and protection from malicious code that users expect on their devices?
- How can a user roam credentials between multiple devices, each with a user-friendly Trusted Computing Base for FIDO?

This document describes how FIDO addresses these goals (where adequate platform mechanisms exist for enforcement) by allowing an application to declare a credential scope that crosses all the various facets it presents to the user.

2.1 Motivation

FIDO conceptually sets a scope for registered keys to the tuple of (Username, Authenticator, Relying Party). But what constitutes a Relying Party? It is quite common for a user to access the same set of services from a Relying Party, on the same device, in one or more web browsers as well as one or more dedicated apps. As the Relying Party may require the user to perform a costly ceremony in order to prove her identity and register a new FIDO key, it is undesirable that the user should have to repeat this ceremony multiple times on the same device, once for each browser or app.

2.2 Avoiding App-Phishing

FIDO provides for user-friendly verification ceremonies to allow access to registered keys, such as entering a simple PIN code and touching a device, or scanning a finger. It should not matter for security purposes if the user re-uses the same verification inputs across Relying Parties, and in the case of a biometric, she may have no choice.

Modern operating systems that use an "app store" distribution model often make a promise to the user that it is "safe to try" any app. They do this by providing strong isolation between applications, so that they may not read each others' data or mutually interfere, and by requiring explicit user permission to access shared system resources.

If a user were to download a maliciously constructed game that instructs her to activate her FIDO authenticator in order to "save your progress" but actually unlocks her banking credential and takes over her account, FIDO has failed, because the risk of phishing has only been moved from the password to an app download. FIDO must not violate a platform's promise that any app is "safe to try" by keeping good custody of the high-value shared state that a registered key represents.

2.3 Comparison to OAuth and OAuth2

The OAuth and OAuth2 of protocols were designed for a server-to-server security model with the assumption that each application instance can be issued, and keep, an "application secret". This approach is ill-suited to the "app store" security model. Although it is common for services to provision an OAuth-style application secret into their apps in an attempt to allow only authorized/official apps to connect, any such "secret" is in fact shared among everyone with access to the app store and can be trivially recovered thorough basic reverse engineering.

In contrast, FIDO's facet concept is designed for the "app store" model from the start. It relies on client-side platform isolation features to make sure that a key registered by a user with a member of a well-behaved "trusted club" stays within that trusted club, even if the user later installs a malicious app, and does not require any secrets hard-coded into a shared package to do so. The user must, however, still make good decisions about which apps and browsers they are willing to preform a registration ceremony with. App store policing can assist here by removing applications which solicit users to register FIDO keys to for Relying Parties in order to make illegitmate or fraudulent use of them.

2.4 Non-Goals

The *Application Facet* concept does not attempt to strongly identify the calling application to a service across a network. Remote attestation of an application identity is an explicit non-goal.

If an unauthorized app can convince a user to provide all the information to it required to register a new FIDO key, the Relying Party cannot use FIDO protocols or the Facet concept to recognize as unauthorized, or deny such an application from performing FIDO operations, and an application that a user has chosen to trust in such a manner can also share access to a key outside of the mechanisms described in this document.

The facet mechanism provides a way for registered keys to maintain their proper scope when created and accessed from a *Trusted Computing Base* (TCB) that provides isolation of malicious apps. A user can also roam their credentials between multiple devices with user-friendly TCBs and credentials will retain their proper scope if this mechanism is correctly implemented by each. However, no guarantees can be made in environments where the TCB is user-hostile, such as a device with malicious code operating with "root" level permissions. On environments that do not provide application isolation but run all code with the privileges of the user, (e.g. traditional desktop operating systems) an intact TCB, including web browsers, may successfully enforce scoping of credentials for web origins only, but cannot meaningfully enforce application scoping.

3. The AppID and FacetID Assertions

When a user performs a Registration operation [UAFArchOverview] a new private key is created by their authenticator, and the public key is sent to the Relying Party. As part of this process, each key is associated with an `AppID`. The `AppID` is a URL carried as part of the protocol message sent by the server and indicates the target for this credential. By default, the audience of the credential is restricted to the *Same Origin* of the `AppID`. In some circumstances, a Relying Party may desire to apply a larger scope to a key. If that `AppID` URL has the `https` scheme, a FIDO client may be able to dereference and process it as a `TrustedFacetList` that designates a scope or audience restriction that includes multiple facets, such as other web origins within the same DNS zone of control of the AppID's origin, or URLs indicating the identity of other types of trusted facets such as mobile apps.

> NOTE
>
> Users may also register multiple keys on a single authenticator for an `AppID`, such as for cases where they have multiple accounts. Such registrations may have a Relying Party assigned username or local nicknames associated to allow them to be distinguished by the user, or they may not (e.g. for 2nd factor use cases, the user account associated with a key may be communicated out-of-band to what is specified by FIDO protocols). All registrations that share an `AppID`, also share these same audience restriction.

3.1 Processing Rules for AppID and FacetID Assertions

3.1.1 Determining the FacetID of a Calling Application

In the Web case, the FacetID must be the Web Origin [RFC6454] of the web page triggering the FIDO operation, written as a URI with an empty path. Default ports are omitted and any path component is ignored.

An example FacetID is shown below:

```
https://login.mycorp.com/
```

In the Android [ANDROID] case, the FacetID must be a URI derived from the Base64 encoding SHA-1 hash of the APK signing certificate [APK-Signing]:

```
android:apk-key-hash:<base64_encoded_sha1_hash-of-apk-signing-cert>
```

The SHA-1 hash can be computed as follows:

EXAMPLE 1: Computing an APK signing certificate hash

```
# Export the signing certificate in DER format, hash, base64 encode and trim '='

keytool -exportcert \
        -alias <alias-of-entry> \
        -keystore <path-to-apk-signing-keystore> &>2 /dev/null | \
        openssl sha1 -binary | \
        openssl base64 | \
        sed 's/=//g'
```

The Base64 encoding is the the "Base 64 Encoding" from Section 4 in [RFC4648], with padding characters removed.

In the iOS [iOS] case, the FacetID must be the BundleID [BundleID] URI of the application:

```
ios:bundle-id:<ios-bundle-id-of-app>
```

3.1.2 Determining if a Caller's FacetID is Authorized for an AppID

1. If the AppID is not an HTTPS URL, and matches the FacetID of the caller, no additional processing is necessary and the operation may proceed.
2. If the AppID is null or empty, the client must set the AppID to be the FacetID of the caller, and the operation may proceed without additional processing.
3. If the caller's FacetID is an `https://` Origin sharing the same host as the AppID, (e.g. if an application hosted at `https://fido.example.com/myApp` set an AppID of `https://fido.example.com/myAppId`), no additional processing is necessary and the operation may proceed. This algorithm may be continued asynchronously for purposes of caching the Trusted Facet List, if desired.
4. Begin to fetch the Trusted Facet List using the HTTP GET method. The location must be identified with an HTTPS URL.
5. The URL must be dereferenced with an anonymous fetch. That is, the HTTP GET must include no cookies, authentication, Origin or Referer headers, and present no TLS certificates or other forms of credentials.
6. The response must set a MIME Content-Type of "application/fido.trusted-apps+json".
7. The caching related HTTP header fields in the HTTP response (e.g. “Expires”) should be respected when fetching a Trusted Facets List.
8. The server hosting the Trusted Facets List must respond uniformly to all clients. That is, it must not vary the contents of the response body based on any credential material, including ambient authority such as originating IP address, supplied with the request.

9. If the server returns an HTTP redirect (status code 3xx) the server must also send the HTTP header `FIDO-AppID-Redirect-Authorized: true` and the client must verify the presence of such a header before following the redirect. This protects against abuse of open redirectors within the target domain by unauthorized parties. If this check has passed, restart this algorithm from step 4.
10. A Trusted Facet List may contain an unlimited number of entries, but clients may truncate or decline to process large responses.
11. From among the objects in the `trustedFacet` array, select the one with the `version` matching that of the protocol message version.
12. The scheme of URLs in `ids` must identify either an application identity (e.g. using the `apk:`, `ios:` or similar scheme) or an `https:` Web Origin [RFC6454].
13. Entries in `ids` using the `https://` scheme must contain only scheme, host and port components, with an optional trailing /. Any path, query string, username/password, or fragment information must be discarded.
14. All Web Origins listed must have host names under the scope of the same least-specific private label in the DNS, using the following algorithm:
    1. Obtain the list of public DNS suffixes from https://publicsuffix.org/list/effective_tld_names.dat (the client may cache such data), or equivalent functionality as available on the platform.
    2. Extract the host portion of the original AppID URL, before following any redirects.
    3. The least-specific private label is the portion of the host portion of the AppID URL that matches a public suffix plus one additional label to the left.
    4. For each Web Origin in the TrustedFacets list, the calculation of the least-specific private label in the DNS must be a case-insensitive match of that of the AppID URL itself. Entries that do not match must be discarded.
15. If the TrustedFacets list cannot be retrieved and successfully parsed according to these rules, the client must abort processing of the requested FIDO operation.
16. After processing the `trustedFacets` entry of the correct `version` and removing any invalid entries, if the caller's FacetID matches one listed in `ids`, the operation is allowed.

3.1.3 TrustedFacets structure

The JSON resource hosted at the AppID URL consists of a dictionary containing a single member, `trustedFacets` which is an array of `TrustedFacets` dictionaries.

WebIDL

```
dictionary TrustedFacets {
    Version     version;
    DOMString[] ids;
};
```

3.1.3.1 Dictionary `TrustedFacets` Members

`version` of type Version
: The protocol version to which this set of trusted facets applies. See [UAFProtocol] for the definition of the `version` structure.

`ids` of type array of DOMString
: An array of URLs identifying authorized facets for this AppID.

3.1.4 AppID Example 1:

".com" is a public suffix. "https://www.example.com/appID" is provided as an AppID. The body of the resource at this location contains:

EXAMPLE 2

```
{
  "trustedFacets" : [{
    "version": { "major": 1, "minor" : 0 },
    "ids": [
    "https://register.example.com", // VALID, shares "example.com" label
    "https://fido.example.com",     // VALID, shares "example.com" label
    "http://www.example.com",       // DISCARD, scheme is not https:
    "http://www.example-test.com",  // DISCARD, "example-test.com" does not match
    "https://www.example.com:444"   // VALID, port is not significant
    ]
  }]
}
```

For this policy, "https://www.example.com" and "https://register.example.com" would have access to the keys registered for this AppID, and "https://user1.example.com" would not.

3.1.5 AppID Example 2:

"hosting.example.com" is a public suffix, operated under "example.com" and used to provide hosted cloud services for many companies. "https://companyA.hosting.example.com/appID" is provided as an AppID. The body of the resource at this location contains:

EXAMPLE 3

```
{
  "trustedFacets" : [{
    "version": { "major": 1, "minor" : 0 },
    "ids": [
        "https://register.example.com",                // DISCARD, does not share  "companyA.hosting.example.com" label
        "https://fido.companyA.hosting.example.com", // VALID, shares "companyA.hosting.example.com" label
        "https://xyz.companyA.hosting.example.com",  // VALID, shares "companyA.hosting.example.com" label
        "https://companyB.hosting.example.com"       // DISCARD, "companyB.hosting.example.com" does not match
     ]
  }]
}
```

For this policy, "https://fido.companyA.hosting.example.com" would have access to the keys registered for this AppID, and "https://register.example.com" and "https://companyB.hosting.example.com" would not as a public-suffix exists between these DNS names and the AppID's.

3.1.6 Obtaining FacetID of Android Native App

This section is non-normative.

The following code demonstrates how a FIDO Client can obtain and construct the FacetID of a calling Android native application.

EXAMPLE 4: AndroidFacetID

```
private String getFacetID(Context aContext, int callingUid) {

    String packageNames[] = aContext.getPackageManager().getPackagesForUid(callingUid);

    if (packageNames == null) {
        return null;
    }

    try {
        PackageInfo info = aContext.getPackageManager().getPackageInfo(packageNames[0], PackageManager.GET_SIGNATURES);

        byte[] cert = info.signatures[0].toByteArray();
        InputStream input = new ByteArrayInputStream(cert);

        CertificateFactory cf = CertificateFactory.getInstance("X509");
        X509Certificate c = (X509Certificate) cf.generateCertificate(input);

        MessageDigest md = MessageDigest.getInstance("SHA1");

        return "android:apk-key-hash:" +
                  Base64.encodeToString(md.digest(c.getEncoded()), Base64.DEFAULT | Base64.NO_WRAP | Base64.NO_PADDING);
    }
    catch (PackageManager.NameNotFoundException e) {
        e.printStackTrace();
    }
    catch (CertificateException e) {
        e.printStackTrace();
    }
    catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    catch (CertificateEncodingException e) {
        e.printStackTrace();
    }

    return null;
}
```

3.1.7 Additional Security Considerations

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

Trusting a web origin facet implicitly trusts all subdomains under the named entity because web user agents do not provide a security barrier between such origins. So, in AppID Example 1, although not explicitly listed, "https://foobar.register.example.com" would still have effective access to credentials registered for the AppID "https://www.example.com/appID" because it can effectively act as "https://register.example.com".

The component implementing the controls described here must reliably identify callers to securely enforce the mechanisms. Platform inter-process communication mechanisms which allow such identification should be used when available.

It is unlikely that the component implementing the controls described here can verify the integrity and intent of the entries on a `TrustedFacetList`. If a trusted facet can be compromised or enlisted as a *confused deputy* [FIDOGlossary] by a malicious party, it may be possible to trick a user into completing an authentication ceremony under the control of that malicious party.

3.1.7.1 Wildcards in TrustedFacet identifiers

This section is non-normative.

Wildcards are not supported in TrustedFacet identifiers. This follows the advice of RFC6125 [RFC6125], section 7.2.

FacetIDs are URIs that uniquely identify specific security principals that are trusted to interact with a given registered credential. Wildcards introduce undesirable ambiguitiy in the defintion of the principal, as there is no consensus syntax for what wildcards mean, how they are expanded and where they can occur across different applications and protocols in common use. For schemes indicating application identities, it is not clear that wildcarding is appropriate in any fashion. For Web Origins, it broadly increases the scope of the credential to potentially include rogue or buggy hosts.

Taken together, these ambiguities might introduce exploitable differences in identity checking behavior among client implementations and would necessitate overly complex and inefficient identity checking algorithms.

A. References

A.1 Normative references

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

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

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

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

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

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges,*FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html

PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

A.2 Informative references

[ANDROID]
The Android™ Operating System. Google, Inc., the Open Handset Alliance and the Android Open Source Project (Work in progress) URL: http://developer.android.com/
[APK-Signing]
Signing Your Applications.The Android™ Operating System. Google, Inc., the Open Handset Alliance and the Android Open Source Project (Accessed 11-March-2014) URL: http://developer.android.com/tools/publishing/app-signing.html
[BundleID]
"Configuring your Xcode Project for Distribution", section "About Bundle IDs", Apple, Inc. Accessed March 11, 2014. URL: https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/ConfiguringYourApp/ConfiguringYourApp.html
[UAFArchOverview]
S. Machani, R. Philpott, S. Srinivas, J. Kemp, J. Hodges,*FIDO UAF Architectural Overview*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-overview-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-overview-v1.1-id-20170202.pdf
[iOS]
iOS Dev Center Apple, Inc. (Accessed March 11, 2014) URL:https://developer.apple.com/devcenter/ios/index.action

IMPLEMENTATION DRAFT

FIDO ECDAA Algorithm

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ecdaa-algorithm-v1.1-id-20170202.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Jan Camenisch, IBM
Manu Drijvers, IBM
Alec Edgington, Trustonic
Anja Lehmann, IBM
Rainer Urian, Infineon

Abstract

The FIDO Basic Attestation scheme uses attestation "group" keys shared across a set of authenticators with identical characteristics in order to preserve privacy by avoiding the introduction of global correlation handles. If such an attestation key is extracted from one single authenticator, it is possible to create a "fake" authenticator using the same key and hence indistinguishable from the original authenticators by the relying party. Removing trust for registering new authenticators with the related key would affect the entire set of authenticators sharing the same "group" key. Depending on the number of authenticators, this risk might be unacceptable high.

This is especially relevant when the attestation key is primarily protected against malware attacks as opposed to targeted physical attacks.

An alternative approach to "group" keys is the use of individual keys combined with a Privacy-CA [TPMv1-2-Part1]. Translated to FIDO, this approach would require one Privacy-CA interaction for each Uauth key. This means relatively high load and high availability requirements for the Privacy-CA. Additionally the Privacy-CA aggregates sensitive information (i.e. knowing the relying parties the user interacts with). This might make the Privacy-CA an interesting attack target.

Another alternative is the Direct Anonymous Attestation [BriCamChe2004-DAA]. Direct Anonymous Attestation is a cryptographic scheme combining privacy with security. It uses the authenticator specific secret once to communicate with a single DAA Issuer and uses the resulting DAA credential in the DAA-Sign protocol with each relying party. The DAA scheme has been adopted by the Trusted Computing Group for TPM v1.2 [TPMv1-2-Part1].

In this document, we specify the use of an improved DAA scheme based on elliptic curves and bilinear pairings largely compatible with [CheLi2013-ECDAA] called ECDAA. This scheme provides significantly improved performance compared with the original DAA and basic building blocks for its implementation are part of the TPMv2 specification [TPMv2-Part1].

Our improvements over [CheLi2013-ECDAA] mainly consist of security fixes (see [ANZ-2013] and [XYZF-

2014]) when splitting the sign operation into two parts.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

String literals are enclosed in “”, e.g. “ED256”.

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

$X = P^x$ denotes scalar multiplication (with scalar x) of a (elliptic) curve point P.

RAND(x) denotes generation of a random number between 0 and x-1.

RAND(G) denotes generation of a random number belonging to Group G.

Specific terminology used in this document is defined in [FIDOGlossary].

The type `BigNumber` denotes an arbitrary length integer value.

The type `ECPoint` denotes an elliptic curve point with its affine coordinates x and y.

The type `ECPoint2` denotes a point on the sextic twist of a BN elliptic curve over$F(q^2)$. The ECPoint2 has two affine coordinates each having two components of type BigNumber

1.1 Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

2. Overview

This section is non-normative.

FIDO uses the concept of attestation to provide a cryptographic proof of the **authenticator** [FIDOGlossary] model to the relying party. When the authenticator is registered to the relying party (RP), it generates a new authentication key pair and includes the public key in the attestation message (also known as key registration data object, **KRD**). When using the ECDAA algorithm, the KRD object is signed using 3.5 ECDAA-Sign.

For privacy reasons, the authentication key pair is dedicated to one RP (to an application identifier **AppID** [FIDOGlossary] to be more specific). Consequently the attestation method needs to provide the same level of unlinkability. This is the reason why the FIDO ECDAA Algorithm doesn't use a basename (bsn) often found in other direct anonymous attestation algorithms, e.g. [BriCamChe2004-DAA] or [BFGSW-2011].

The authenticator encapsulates all user verification operations and cryptographic functions. An authenticator specific module (**ASM**) [FIDOGlossary] is used to provide a standardized communication interface for authenticators. The authenticator might be implemented in separate hardware or trusted execution environments. The ASM is assumed to run in the normal operating system (e.g. Android, Windows, ...).

2.1 Scope

This document describes the FIDO ECDAA attestation algorithm in detail.

2.2 Architecture Overview

ECDAA attestation defines global system parameters and issuer specific parameters. Both parameter sets need to be installed on the host, in the authenticator and in the FIDO Server. The ECDAA method consists of two steps:

- ECDAA-Join to be performed *before* the first FIDO Registration
  - n = GetNonceFromECDAAIssuer()
  - (Q, c1, s1) = EcdaaJoin1(X, Y, n)
  - (A, B, C, D, s2, c2) = EcdaaIssuerJoin(Q, c1, s1)
  - EcdaaJoin2(A, B, C, D, c2, s2) // store cre=(A, B, C, D)
- and the pair of ECDAA-Sign performed by the authenticator and ECDAA-Verify performed by the FIDO Server as part of the FIDO Registration.
  - Client: Attestation = (signature, KRD) = EcdaaSign(AppID)
  - Server: success=EcdaaVerify(signature, KRD, AppID)

The technical implementation details of the ECDAA-Join step are out-of-scope for FIDO. In this document we normatively specify the general algorithm to the extent required for interoperability and we outline examples of some possible implementations for this step.

The ECDAA-Sign and ECDAA-Verify steps and the encoding of the related ECDAA Signature are normatively specified in this document. The generation and encoding of the KRD object is defined in other FIDO specifications.

The algorithm and terminology are inspired by [BFGSW-2011]. The algorithm was modified in order to fix security weaknesses (e.g. as mentioned by [ANZ-2013] and [XYZF-2014]). Our algorithm proposes an improved task split for the sign operation while still being compatible to TPMv2 (without fixing the TPMv2 weaknesses in such case).

3. FIDO ECDAA Attestation

This section is normative.

3.1 Object Encodings

We need to convert `BigNumber` and `ECPoint` objects to byte strings using the following encoding functions:

3.1.1 Encoding `BigNumber` values as byte strings (BigNumberToB)

We use the I2OSP algorithm as defined in [RFC3447] for converting big numbers to byte arrays. The bytes from the big endian encoded (non-negative) number `n` will be copied right-aligned into the buffer area `b`. The unused bytes will be set to 0. Negative values will not occur due to the construction of the algorithms.

EXAMPLE 1: Converting BigNumber n to byte string b

```
b0 b1 b2 b3 b4 b5 b6 b7
 0  0 n0 n1 n2 n3 n4 n5
```

The algorithm implemented in Java looks like this:

EXAMPLE 2: Algorithm for converting BigNumber to byte strings

```
ByteArray BigNumberToB(
      BigNumber inVal, // IN: number to convert
      int size         // IN: size of the output.
     )
{
  ByteArray buffer = new ByteArray(size);
  int oversize = size - inVal.length;
  if (oversize < 0)
    return null;
  for (int i=overvize; i > 0; i--)
     buffer[i] = 0;
  ByteCopy( inVal.bytes, &buffer[oversize], inVal.length);
 return buffer;
}
```

3.1.2 Encoding `ECPoint` values as byte strings (ECPointToB)

We use the ANSI X9.62 Point-to-Octet-String [ECDSA-ANSI] conversion using the expanded format, i.e. the format where the compression byte (i.e. 0x04 for expanded) is followed by the encoding of the affine x coordinate, followed by the encoding of the affine y coordinate.

EXAMPLE 3: Converting ECPoint P to byte string

```
(x, y) = ECPointGetAffineCoordinates(P)
len = G1.byteLength
byte string = 0x04 | BigIntegerToB(x,len) | BigIntegerToB(y,len)
```

3.1.3 Encoding `ECPoint2` values as byte strings (ECPoint2ToB)

The type `ECPoint2` denotes a point on the sextic twist of a BN elliptic curve over $F(q^2)$, see section 4.1 Supported Curves for ECDAA. Each `ECPoint2` is represented by a pair `(a, b)` of elements of $F(q)$.

The group zero element is always encoded (using the encoding rules as described below) as a an element having all components set to zero (i.e. cx.a=0, cx.b=0, cy.a=0, cy.b=0).

We always assume normalized (non-zero) ECPoint2 values (i.e. cz = 1) before encoding them. Non-zero values are encoded using the expanded format (i.e. 0x04 for expanded) followed by the cx followed by the cy value. This leads to the concatenation of 0x04 followed by the first element (`cx.a`) and second element (`cx.b`) of the pair of cx followed by the first element (`cy.a`) and second element (`cy.b`) of the pair of cy. All individual numbers are padded to the same length (i.e. the maximum byte length of all relevant 4 numbers).

EXAMPLE 4: Converting ECPoint2 P2 to byte string

```
(cx, cy) = ECPointGetAffineCoordinates(P2)
len = G2.byteLength
byte string = 0x04 | BigIntegerToB(cx.a,len) | BigIntegerToB(cx.b,len)
                   | BigIntegerToB(cy.a,len) | BigIntegerToB(cy.b,len)
```

3.2 Global ECDAA System Parameters

1. Groups G_1, G_2 and G_T, of sufficiently large prime order p
2. Two generators P_1 and P_2, such that $G_1 = \langle P_1 \rangle$ and $G_2 = \langle P_2 \rangle$
3. A bilinear pairing $e : G_1 \times G_2 \rightarrow G_T$. We propose the use of "ate" pairing (see [BarNae-2006]). For example source code on this topic, see BNPairings.
4. Hash function H with $H : \{0, 1\}^* \rightarrow Z_p$.
5. (G_1, P_1, p, H) are installed in all authenticators implementing FIDO ECDAA attestation.

Definition of G_1, G_2, G_T, Pairings and hash function H

See section 4.1 Supported Curves for ECDAA.

3.3 Issuer Specific ECDAA Parameters

Issuer Parameters parI

1. Randomly generated issuer private key $isk = (x, y)$ with $[x, y = RAND(p)]$.
2. Issuer public key (X, Y), with $X = P_2^x$ and $Y = P_2^y$.
3. A proof that the issuer key was correctly computed
    1. BigInteger $r_x = RAND(p)$
    2. BigInteger $r_y = RAND(p)$
    3. ECPoint2 $U_x = P_2^{r_x}$
    4. ECPoint2 $U_y = P_2^{r_y}$
    5. BigInteger $c = H(U_x|U_y|P_2|X|Y)$
    6. BigInteger $s_x = r_x + c \cdot x \pmod p$

7. BigInteger $s_y = r_y + c \cdot y \pmod{p}$

4. $ipk = X, Y, c, s_x, s_y$

Whenever a party uses ipk for the first time, it must first verify that it was correctly generated:

$$H(P_2^{s_x} \cdot X^{-c} | P_2^{s_y} \cdot Y^{-c} | P_2 | X | Y) \stackrel{?}{=} c$$

NOTE

$$P_2^{s_x} \cdot X^{-c} = P_2^{r_x + cx} \cdot P_2^{-cx} = P_2^{r_x} = U_x$$

$$P_2^{s_y} \cdot Y^{-c} = P_2^{r_y + cy} \cdot P_2^{-cy} = P_2^{r_y} = U_y$$

The ECDAA-Issuer public key ipk must be dedicated to a single authenticator model.

3.4 ECDAA-Join

NOTE

One ECDAA-Join operation is required once in the lifetime of anauthenticator prior to the first registration of a credential.

In order to use ECDAA, the authenticator must first receive ECDAA credentials from an ECDAA-Issuer. This is done by the ECDAA-Join operation. This operation needs to be performed a single time (before the first credential registration can take place). After the ECDAA-Join, the authenticator will use the ECDAA-Sign operation as part of each FIDO Registration. The ECDAA-Issuer is not involved in this step. ECDAA plays no role in FIDO Authentication / Transaction Confirmation operations.

In order to use ECDAA, (at least) one ECDAA-Issuer is needed. The approach specified in this document easily scales to multiple ECDAA-Issuers, e.g. one per authenticator vendor. FIDO lets the authenticator vendor choose any ECDAA-Issuer (similar to his current freedom for selecting any PKI infrastructure/service provider to issuing attestation certificates required for FIDO Basic Attestation).

- All ECDAA-Join operations (of the related authenticators) are performed with one of the ECDAA-Issuer entities.
- Each ECDAA-Issuer has a set of public parameters, i.e. ECDAA public key material. The related Attestation Trust Anchor is contained in the metadata of each authenticator model identified by its AAGUID.

There are two different implementation options relevant for the authenticator Vendors (the authenticator vendor can freely choose them):

1. In-Factory ECDAA-Join
2. Remote ECDAA-Join and

In the first case, physical proximity is used to locally establish the trust between the ECDAA-Issuer and the authenticator (e.g. using a key provisioning station in a production line). There is no requirement for the ECDAA-Issuer to operate an online web service.

In the second case, some credential is required to remotely establish the trust between the ECDAA-Issuer and the authenticator. As this operation is performed once and only with a single ECDAA-Issuer, privacy is preserved and an authenticator specific credential can and should be used.

Not all ECDAA authenticators might be able to add theirauthenticator model IDs (e.g. AAGUID) to the registration assertion (e.g. TPMs). In all cases, the ECDAA-Issuer will be able to derive the exact the authenticator model from either the credential or the physically proximiate authenticator. So the ECDAA-Issuer root key must be dedicated to a single authenticator model.

3.4.1 ECDAA-Join Algorithm

This section is normative.

NOTE

If this join is not in-factory, the value Q must be authenticated by the authenticator. Upon receiving this value, the issuer must verify that this authenticator did not join before.

1. The authenticator asks the issuer for a nonce.
2. The issuer chooses a nonce BigInteger $n = RAND(p)$ and sends n via the ASM to the authenticator.
3. The authenticator chooses and stores the ECDAA private key BigInteger $sk = RAND(p)$
4. The authenticator computes its ECDAA public key ECPoint $Q = P_1^{sk}$
5. The authenticator proves knowledge of sk as follows
    1. BigInteger $r_1 = RAND(p)$
    2. ECPoint $U_1 = P_1^{r_1}$
    3. BigInteger $c_1 = H(U_1|P_1|Q|n)$
    4. BigInteger $s_1 = r_1 + c_1 \cdot sk$
6. The authenticator sends Q, c_1, s_1 via the ASM to the issuer
7. The issuer verifies that the authenticator is "authentic" and that Q was indeed generated by the authenticator. In the case of an in-factory Join, this might be trivial; in the case of a remote Join this typically requires the use of other cryptographic methods. Since ECDAA-Join is a one-time operation, unlinkability is not a concern for that.
8. The issuer verifies that $Q \in G_1$ and verifies $H(P_1^{s_1} \cdot Q^{-c_1}|P_1|Q|n) \stackrel{?}{=} c_1$ (check proof-of-possession of private key).

    NOTE

    $$P_1^{s_1} \cdot Q^{-c_1} = P_1^{r_1+c_1 sk} \cdot Q^{-c_1} = P_1^{r_1+c_1 sk} \cdot P_1^{-c_1 sk} = P_1^{r_1} = U_1$$

9. The issuer creates credential (A, B, C, D) as follows
    1. BigInteger $l_J = RAND(p)$
    2. ECPoint $A = P_1^{l_J}$
    3. ECPoint $B = A^y$
    4. ECPoint $C = A^x \cdot Q^{xyl_J}$
    5. ECPoint $D = Q^{l_J y}$
10. The issuer proves that it computed this credential correctly:
    1. BigInteger $r_2 = RAND(p)$
    2. ECPoint $U_2 = P_1^{r_2}$
    3. ECPoint $V_2 = Q^{r_2}$
    4. BigInteger $c_2 = H(U_2|V_2|P_1|B|Q|D)$
    5. BigInteger $s_2 = r_2 + c_2 \cdot l_J \cdot y$
11. The issuer sends A, B, C, D, c_2, s_2 to the authenticator.
12. The authenticator checks that $A, B, C, D \in G_1$ and $A \neq 1_{G_1}$
13. The authenticator checks $H(P_1^{s_2} \cdot B^{-c_2}|Q^{s_2} \cdot D^{-c_2}|P_1|B|Q|D) \stackrel{?}{=} c_2$

    NOTE

    $$P_1^{s_2} \cdot B^{-c_2} = P_1^{r_2} \cdot P_1^{c_2 \cdot l_J \cdot y} \cdot B^{-c_2} = U_2 \cdot B^{c_2} \cdot B^{-c_2} = U_2$$

$$Q^{s_2} \cdot D^{-c_2} = Q^{r_2} \cdot Q^{c_2 \cdot l_J \cdot y} \cdot D^{-c_2} = V_2 \cdot D^{c_2} \cdot D^{-c_2} = V_2$$

14. The authenticator checks $e(A, Y) \stackrel{?}{=} e(B, P_2)$

> NOTE
>
> $e(A, Y) = e(P_1^{l_J}, P_2^y); e(B, P_2) = e(A^y, P_2) = e(P_1^{yl_J}, P_2)$

15. and the authenticator checks $e(C, P_2) \stackrel{?}{=} e(A \cdot D, X)$

> NOTE
>
> $e(C, P_2) = e(A^x \cdot Q^{xyl_J}, P_2); e(A \cdot D, X) = e(A \cdot Q^{yl_J}, P_2^x)$

16. The authenticator stores credential A, B, C, D

3.4.2 ECDAA-Join Split between Authenticator and ASM

This section is non-normative.

> NOTE
>
> If this join is not in-factory, the value Q must be authenticated by theauthenticator. Upon receiving this value, the issuer must verify that this authenticator did not join before.

1. The ASM asks the issuer for a nonce.
2. The issuer chooses a nonce BigInteger $n = RAND(p)$ and sends n to the ASM.
3. The ASM forwards n to the authenticator
4. The authenticator chooses and stores the private key BigInteger $sk = RAND(p)$
5. The authenticator computes its ECDAA public key ECPoint $Q = P_1^{sk}$
6. The authenticator proves knowledge of sk as follows
    1. BigInteger $r_1 = RAND(p)$
    2. ECPoint $U_1 = P_1^{r_1}$
    3. BigInteger $c_1 = H(U_1|P_1|Q|n)$
    4. BigInteger $s_1 = r_1 + c_1 \cdot sk$
7. The authenticator sends Q, c_1, s_1 to the ASM, who forwards it to the issuer.
8. The issuer verifies that the authenticator is "authentic" and that Q was indeed generated by the authenticator. In the case of an in-factory Join, this might be trivial; in the case of a remote Join this typically requires the use of other cryptographic methods. Since ECDAA-Join is a one-time operation, unlinkability is not a concern for that.
9. The issuer verifies that $Q \in G_1$ and verifies $H(P_1^{s_1} \cdot Q^{-c_1}|P_1|Q|n) \stackrel{?}{=} c_1$.
10. The issuer creates credential (A, B, C, D) as follows
    1. BigInteger $l_J = RAND(p)$
    2. ECPoint $A = P_1^{l_J}$
    3. ECPoint $B = A^y$
    4. ECPoint $C = A^x \cdot Q^{xyl_J}$

5. ECPoint $D = Q^{l_J y}$

11. The issuer proves that it computed this credential correctly:
    1. BigInteger $r_2 = RAND(p)$
    2. ECPoint $U_2 = P_1^{r_2}$
    3. ECPoint $V_2 = Q^{r_2}$
    4. BigInteger $c_2 = H(U_2|V_2|P_1|B|Q|D)$
    5. BigInteger $s_2 = r_2 + c_2 \cdot l_J \cdot y$
12. The issuer sends A, B, C, D, c_2, s_2 to the ASM. The issuer authenticates B, D, c_2, s_2 such that the authenticator can verify they were created by the issuer.
13. The ASM checks that $A, B, C, D \in G_1$ and $A \neq 1_{G_1}$
14. The ASM checks $H(P_1^{s_2} \cdot B^{-c_2}|Q^{s_2} \cdot D^{-c_2}|P_1|B|Q|D) \stackrel{?}{=} c_2$
15. The ASM checks $e(A, Y) \stackrel{?}{=} e(B, P_2)$
16. and the ASM checks that $e(C, P_2) \stackrel{?}{=} e(A \cdot D, X)$
17. The ASM stores A, B, C, D and sends B, D, c_2, s_2 to the authenticator
18. The authenticator checks $B, D \in G_1$ and $B \neq 1_{G_1}$, and verifies that B, D, c_2, s_2 were sent by the issuer.
19. The authenticator checks $H(P_1^{s_2} \cdot B^{-c_2}|Q^{s_2} \cdot D^{-c_2}|P_1|B|Q|D) \stackrel{?}{=} c_2$
20. The authenticator stores B, D and ignores further join requests.

> NOTE
>
> These values belong to the ECDAA secret key sk. They should persist even in the case of a factory reset.

3.4.3 ECDAA-Join Split between TPM and ASM

This section is non-normative.

> NOTE
>
> The Endorsement key credential (EK-C) and TPM2_ActivateCredentials are used for supporting the remote Join.

This description is based on the principles described in [TPMv2-Part1] section 24 and [Arthur-Challener-2015], page 109 ("Activating a Credential").

1. The ASM asks the ECDAA Issuer for a nonce.
2. The ECDAA Issue chooses a nonce BigInteger $n = RAND(p)$ and sends n to the ASM.
3. The ASM
    1. instructs the TPM to create a restricted key by calling TPM2_Create, giving the public key template `TPMT_PUBLIC` [TPMv2-Part2] (including the public key Q in field `unique`) to the ASM.
    2. retrieves TPM Endorsement Key Certificate (EK-C) from the TPM
    3. calls TPM2_Commit(keyhandle, P1, s2, y2) where keyhandle is the handle of the restricted key generated before (see above), P1 is set to P_1, and s2 and y2 are left empty. This call returns K, L, E, and ctr; where K and L will be empty.
    4. computes BigInteger $c_1 = H(E|P_1|Q|n)$
    5. call TPM2_Sign(c_1, ctr), returning s_1.
    6. sends EK-C, `TPMT_PUBLIC` (including Q in field `unique`), c_1, s_1 to the ECDAA Issuer.

4. The ECDAA Issuer
    1. verifies EK-C and its certificate chain. As a result the ECDAA Issuer knows the TPM model related to EK-C.
    2. verifies that this EK-C was not used in a (successful) Join before
    3. Verifies that the `objectAttributes` in `TPMT_PUBLIC` [TPMv2-Part2] matches the following flags: `fixedTPM` = 1; `fixedParent` = 1; `sensitiveDataOrigin` = 1; `encryptedDuplication` = 0; `restricted` = 1; `decrypt` = 0; `sign` = 1.
    4. examines the public key Q, i.e. it verifies that $Q \in G_1$
    5. checks $H(P_1^{s_1} \cdot Q^{-c_1} | P_1 | Q | n) \stackrel{?}{=} c_1$
    6. generates the ECDAA credential (A, B, C, D) as follows
        1. BigInteger $l_J = RAND(p)$
        2. ECPoint $A = P_1^{l_J}$
        3. ECPoint $B = A^y$
        4. ECPoint $C = A^x \cdot Q^{xyl_J}$
        5. ECPoint $D = Q^{l_J y}$
    7. proves that it computed this credential correctly:
        1. BigInteger $r_2 = RAND(p)$
        2. ECPoint $U_2 = P_1^{r_2}$
        3. ECPoint $V_2 = Q^{r_2}$
        4. BigInteger $c_2 = H(U_2 | V_2 | P_1 | B | Q | D)$
        5. BigInteger $s_2 = r_2 + c_2 \cdot l_J \cdot y$
    8. generates a *secret* (derived from a *seed*) and wraps the credential A, B, C, D using that *secret*.
    9. encrypts the *seed* using the public key included in EK-C.
    10. uses *seed* and *name* in KDFa (see [TPMv2-Part2] section 24.4) to derive HMAC and *symmetric encryption key*. Wrap the *secret* in *symmetric encryption key* and protect it with the *HMAC key*.

        NOTE

        The parameter *name* in KDFa is derived from `TPMT_PUBLIC`, see [TPMv2-Part1], section 16.

    11. sends the credential proof c_2, s_2 and the wrapped object including the credential from previous step to the ASM.
5. The ASM instructs the TPM (by calling TPM2_ActivateCredential) to
    1. decrypt the *seed* using the TPM Endorsement key
    2. compute the *name* (for the ECDAA attestation key)
    3. use the *seed* in KDFa (with *name*) to derive the *HMAC key* and the *symmetric encryption key*.
    4. use the *symmetric encryption key* to unwrap the *secret*.
6. The ASM
    1. unwraps the credential A, B, C, D using the *secret* received from the TPM.
    2. checks that $A, B, C, D \in G_1$ and $A \neq 1_{G_1}$
    3. checks $H(P_1^{s_2} \cdot B^{-c_2} | Q^{s_2} \cdot D^{-c_2} | P_1 | B | Q | D) \stackrel{?}{=} c_2$
    4. checks $e(A, Y) \stackrel{?}{=} e(B, P_2)$ and $e(C, P_2) \stackrel{?}{=} e(A \cdot D, X)$
    5. stores A, B, C, D

3.5 ECDAA-Sign

NOTE

One ECDAA-Sign operation is required for the client-side environment whenever a new credential is being registered at a relying party.

3.5.1 ECDAA-Sign Algorithm

This section is normative.

(signature, KRD) = EcdaaSign(String AppID)

Parameters

- p: System parameter prime order of group G1 (global constant)
- AppID: FIDO AppID (i.e. https-URL of TrustedFacets object)

Algorithm outline

1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here
2. BigNumber $l = \mathrm{RAND}(p)$
3. ECPoint $R = A^l$;
4. ECPoint $S = B^l$;
5. ECPoint $T = C^l$;
6. ECPoint $W = D^l$;
7. BigInteger $r = \mathrm{RAND}(p)$
8. ECPoint $U = S^r$
9. BigInteger $c = H(U|S|W|AppID|H(KRD))$
10. BigInteger $s = r + c \cdot sk \pmod p$
11. signature = (c, s, R, S, T, W)
12. return (signature, KRD)

3.5.2 ECDAA-Sign Split between Authenticator and ASM

This section is non-normative.

NOTE

This split requires both the authenticator and ASM to be honest to achieve anonymity. Only the authenticator must be trusted for unforgeability. The communication between ASM and authenticator must be secure.

Algorithm outline

1. The ASM randomizes the credential
   1. BigNumber $l = \mathrm{RAND}(p)$
   2. ECPoint $R = A^l$;
   3. ECPoint $S = B^l$;
   4. ECPoint $T = C^l$;
   5. ECPoint $W = D^l$;
2. The ASM sends l, $AppID$ to the authenticator
3. The authenticator performs the following tasks
   1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here

2. ECPoint $S' = B^l$
3. ECPoint $W' = D^l$
4. BigInteger $r = RAND(p)$
5. ECPoint $U = S'^r$
6. BigInteger $c = H(U|S'|W'|AppID|H(KRD))$
7. BigInteger $s = r + c \cdot sk \pmod{p}$
8. Send c, s, KRD to the ASM

4. The ASM sets signature = (c, s, R, S, T, W) and outputs (signature,KRD)

3.5.3 ECDAA-Sign Split between TPM and ASM

This section is non-normative.

> NOTE
>
> This algorithm is for the special case of a TPMv2 as authenticator. This case requires both the TPM and ASM to be honest for anonymity and unforgeability (see [XYZF-2014]).

Algorithm outline

1. The ASM randomizes the credential
   1. BigNumber $l = RAND(p)$
   2. ECPoint $R = A^l$;
   3. ECPoint $S = B^l$;
   4. ECPoint $T = C^l$;
   5. ECPoint $W = D^l$;
2. The ASM calls TPM2_Commit() with $P1$ set to S and $s2, y2$ empty buffers. The ASM receives the result values $K, L, E = S^r$ and ctr. K and L are empty since $s2, y2$ are empty buffers.
3. The ASM calls TPM2_Create to generate the new authentication key pair.
4. The ASM calls TPM2_Certify() on the newly created key with ctr from the TPM2_Commit and $E, S, W, AppID$ as qualifying data ($E = S^r$ is returned by step 2). The ASM receives signature c, s and attestation block KRD (i.e. TPMS_ATTEST structure in this case).
5. The ASM sets signature = (c, s, R, S, T, W) and outputs (signature,KRD)

3.6 ECDAA-Verify Operation

This section is normative.

> NOTE
>
> One ECDAA-Verify operation is required for the FIDO Server as part of each FIDO Registration.

boolean EcdaaVerify(signature, AppID, KRD, ModelName)

Parameters

- p: System parameter prime order of group G_1 (global constant)
- P_2: System parameter generator of group G_2 (global constant)
- signature: (c, s, R, S, T, W)
- AppID: FIDO AppID
- KRD: Attestation Data object as defined in other specifications.
- ModelName: the claimed FIDO authenticator model (i.e. either AAID or AAGUID)

Algorithm outline

1. Based on the claimed ModelName, look up X, Y from trusted source
2. Check that $R, S, T, W \in G_1$, $R \neq 1_{G_1}$, and $S \neq 1_{G_1}$.
3. $H(S^s \cdot W^{-c} | S | W | AppID | H(KRD)) \stackrel{?}{=} c$; fail if not equal

> NOTE
>
> $B = A^y = P_1^{ly}$
>
> $D = Q^{l_J y} = P_1^{skl_J y} = B^{sk}$
>
> $S = B^l$ and $W = D^l$
>
> $U = S^r$
>
> $S^s \cdot W^{-c} = S^{r+csk} \cdot W^{-c} = U \cdot S^{csk} \cdot W^{-c}$
> $= U \cdot B^{lcsk} \cdot D^{-lc} = U \cdot B^{lcsk} \cdot B^{-lcsk} = U$

4. $e(R, Y) \stackrel{?}{=} e(S, P_2)$; fail if not equal

> NOTE
>
> $e(R, Y) = e(A^l, P_2^y); e(S, P_2) = e(B^l, P_2) = e(A^{ly}, P_2)$

5. $e(T, P_2) \stackrel{?}{=} e(R \cdot W, X)$; fail if not equal

> NOTE
>
> $e(T, P_2) = e(C^l, P_2) = e(A^{xl} \cdot Q^{xlyl_J}, P_2); e(A^l \cdot D^l, X) = e(A^l \cdot Q^{lyl_J}, P_2^x)$

6. for (all sk' on RogueList) do if $W \stackrel{?}{=} S^{sk'}$ fail;
7. // perform all other processing steps for new credential registration

> NOTE
> In the case of a TPMv2, i.e. KRD is a `TPMS_ATTEST` object. In this case the verifier must check whether the `TPMS_ATTEST` object starts with `TPM_GENERATED` magic number and whether its field `objectAttributes` contains the flag `fixedTPM`=1 (indicating that the key was generated by the TPM).

8. return true;

4. FIDO ECDAA Object Formats and Algorithm Details

This section is normative.

4.1 Supported Curves for ECDAA

Definition of G1

G1 is an elliptic curve group E : $y^2 = x^3 + ax + b$ over $F(q)$ with $a = 0$.

Definition of G2

G2 is the p-torsion subgroup of $E'(F_{q^2})$ where E' is a sextic twist of E. With E' : $y'^2 = x'^3 + b'$.

An element of $F(q^2)$ is represented by a pair (a,b) where a + bX is an element of $F(q)[X]/ < X^2 + 1 >$. We use angle brackets $< Y >$ to signify the ideal generated by the enclosed value.

> NOTE
>
> In the literature the pair (a,b) is sometimes also written as a complex number $a + b * i$.

Definition of GT

GT is an order-p subgroup of $F_{q^{12}}$.

Pairings

We propose the use of Ate pairings as they are efficient (more efficient than Tate pairings) on Barreto-Naehrig curves [DevScoDah2007].

Supported BN curves

We use pairing-friendly Barreto-Naehrig [BarNae-2006] [ISO15946-5] elliptic curves. The curves `TPM_ECC_BN_P256` and `TPM_ECC_BN_P638` curves are defined in [TPMv2-Part4].

BN curves have a Modulus $q = 36 \cdot u^4 + 36 \cdot u^3 + 24 \cdot u^2 + 6 \cdot u + 1$ [ISO15946-5] and a related order of the group $p = 36 \cdot u^4 + 36 \cdot u^3 + 18 \cdot u^2 + 6 \cdot u + 1$ [ISO15946-5].

- `TPM_ECC_BN_P256` is a curve of form E(F(q)), where q is the field modulus [TPMv2-Part4] [BarNae-2006]. This curve is identical to the P256 curve defined in [ISO15946-5] section C.3.5.
  - The values have been generated using u=-7 530 851 732 716 300 289.
  - Modulus q = 115 792 089 237 314 936 872 688 561 244 471 742 058 375 878 355 761 205 198 700 409 522 629 664 518 163
  - Group order p = 115 792 089 237 314 936 872 688 561 244 471 742 058 035 595 988 840 268 584 488 757 999 429 535 617 037
  - p and q have length of 256 bit each.
  - b = 3
  - P_1_256 = (x=1, y=2)
  - b' = (a=3, b=3)
  - P_2_256 = (x,y), with
    - P_2_256.x = (a=114 909 019 869 825 495 805 094 438 766 505 779 201 460 871 441 403 689 227 802 685 522 624 680 861 435, b=35 574 363 727 580 634 541 930 638 464 681 913 209 705 880 605 623 913 174 726 536 241 706 071 648 811)
    - P_2_256.y = (a=65 076 021 719 150 302 283 757 931 701 622 350 436 355 986 716 727 896 397 520 706 509 932 529 649 684, b=113 380 538 053 789 372 416 298 017 450 764 517 685 681 349 483 061 506 360 354 665 554 452 649 749 368)
- `TPM_ECC_BN_P638` [TPMv2-Part4] uses
  - The values have been generated using u=365 375 408 992 443 362 629 982 744 420 548 242 302 862 098 433
  - Modulus q = 641 593 209 463 000 238 284 923 228 689 168 801 117 629 789 043 238 356 871 360 716 989 515 584 497 239 494 051 781 991 794 253 619 096 481 315 470 262 367 432 019 698 642 631 650 152 075 067 922 231 951 354 925 301 839 708 740 457 083 469 793 717 125 223
  - The related order of the group is p = 641 593 209 463 000 238 284 923 228 689 168 801 117 629 789 043 238 356 871 360 716 989 515 584 497 239 494 051 781 991 794 252 818 101 344 337 098 690 003 906 272 221 387 599 391 201 666 378 807 960 583 525 233 832 645 565 592 955 122 034 352 630 792 289
  - p and q have length of 638 bit each.
  - b = 257

- P_1_638 = (x=641 593 209 463 000 238 284 923 228 689 168 801 117 629 789 043 238 356 871 360 716 989 515 584 497 239 494 051 781 991 794 253 619 096 481 315 470 262 367 432 019 698 642 631 650 152 075 067 922 231 951 354 925 301 839 708 740 457 083 469 793 717 125 222, y=16)
- b' = (a=771, b=1542)
- P_2_638 = (x, y), with
  - P_2_638.x = (a=192 492 098 325 059 629 927 844 609 092 536 807 849 769 208 589 403 233 289 748 474 758 010 838 876 457 636 072 173 883 771 602 089 605 233 264 992 910 618 494 201 909 695 576 234 119 413 319 303 931 909 848 663 554 062 144 113 485 982 076 866 968 711 247, b=166 614 418 891 499 184 781 285 132 766 747 495 170 152 701 259 472 324 679 873 541 478 330 301 406 623 174 002 502 345 930 325 474 988 134 317 071 869 554 535 111 092 924 719 466 650 228 182 095 841 246 668 361 451 788 368 418 036 777 197 454 618 413 255)
  - P_2_638.y = (a=622 964 952 935 200 827 531 506 751 874 167 806 262 407 152 244 280 323 674 626 687 789 202 660 794 092 633 841 098 984 322 671 973 226 667 873 503 889 270 602 870 064 426 165 592 237 410 681 318 519 893 784 898 821 343 051 339 820 566 224 981 344 169 470, b=514 285 963 827 225 043 076 463 721 426 569 583 576 029 220 880 138 564 906 219 230 942 887 639 456 599 654 554 743 732 087 558 187 149 207 036 952 474 092 411 405 629 612 957 921 369 286 372 038 525 830 610 755 207 588 843 864 366 759 521 090 861 911 494)

* `ECC_BN_DSD_P256` [DevScoDah2007] section 3 uses
  - The values have been generated using u=6 917 529 027 641 089 837
  - Modulus q = 82434016654300679721217353503190038836571781811386228921167322412819029493183
  - The related order of the group is p = 82434016654300679721217353503190038836284668564296686430114510052556401373769
  - p and q have length of 256 bit each.
  - b = 3
  - P_1_DSD_P256 = (1, 2)
  - b' = (a=3, b=6)
  - P_2_DSD_P256 = (x, y), with
    - P_2_DSD_P256.x = (a=73 481 346 555 305 118 071 940 904 527 347 990 526 214 212 698 180 576 973 201 374 397 013 567 073 039, b=28 955 468 426 222 256 383 171 634 927 293 329 392 145 263 879 318 611 908 127 165 887 947 997 417 463)
    - P_2_DSD_P256.y = (a=3 632 491 054 685 712 358 616 318 558 909 408 435 559 591 759 282 597 787 781 393 534 962 445 630 353, b=60 960 585 579 560 783 681 258 978 162 498 088 639 544 584 959 644 221 094 447 372 720 880 177 666 763)
* `ECC_BN_ISOP512` [ISO15946-5] section C.3.7 uses
  - The values have been generated using u=138 919 694 570 470 098 040 331 481 282 401 523 727
  - Modulus q = 13 407 807 929 942 597 099 574 024 998 205 830 437 246 153 344 875 111 580 494 527 427 714 590 099 881 795 845 981 157 516 604 994 291 639 750 834 285 779 043 186 149 750 164 319 950 153 126 044 364 566 323
  - The related order of the group is p = 13 407 807 929 942 597 099 574 024 998 205 830 437 246 153 344 875 111 580 494 527 427 714 590 099 881 680 053 891 920 200 409 570 720 654 742 146 445 677 939 306 408 461 754 626 647 833 262 056 300 743 149
  - p and q have length of 512 bit each.
  - b = 3
  - P_1_ISO_P512 = (x=1,y=2)
  - b' = (a=3, b=3)
  - P_2_ISO_P512 = (x, y), with
    - P_2_ISO_P512.x = (a=3 094 648 157 539 090 131 026 477 120 117 259 896 222 920 557 994 037 039 545 437 079 729 804 516 315 481 514 566 156 984 245 473 190 248 967 907 724 153 072 490 467 902 779 495 072 074 156 718 085 785 269, b=3 776 690 234 788 102 103 015 760 376 468 067 863 580 475 949 014 286 077 855 600 384 033 870 546 339 773 119 295 555 161 718 985 244 561 452 474 412 673 836 012 873 126 926 524 076 966 265 127 900 471 529)
    - P_2_ISO_P512.y = (a=7 593 872 605 334 070 150 001 723 245 210 278 735 800 573 263 881 411 015 285 406 372 548 542 328 752 430 917 597 485 450 360 707 892 769

159 214 115 916 255 816 324 924 295 339 525 686 777 569 132 644 242, b=9 131 995 053 349 122 285 871 305 684 665 648 028 094 505 015 281 268 488 257 987 110 193 875 868 585 868 792 041 571 666 587 093 146 239 570 057 934 816 183 220 992 460 187 617 700 670 514 736 173 834 408)

> NOTE
>
> Spaces are used inside numbers to improve readability.

Hash Algorithms

Depending on the curve, we use `H(x) = SHA256(x) mod p` or `H(x) = SHA512(x) mod p` as hash algorithm H: $\{0, 1\}^* \rightarrow Z_p$.

The argument of the hash function must always be converted to a byte string using the appropriate encoding function specific in section 3.1 Object Encodings, e.g. according to section 3.1.3 Encoding ECPoint2 values as byte strings (ECPoint2ToB) in the case of `ECPoint2` points.

> NOTE
>
> We don't use IEEE P1363.3 section 6.1.1 IHF1-SHA with security parameter t (e.g. t=128 or 256) as it is more complex and not supported by TPMv2.

4.2 ECDAA Algorithm Names

We define the following JWS-style algorithm names (see [RFC7515]):

ED256
: `TPM_ECC_BN_P256` curve, using SHA256 as hash algorithm H.

ED256-2
: `ECC_BN_DSD_P256` curve, using SHA256 as hash algorithm H.

ED512
: `ECC_BN_ISOP512` curve, using SHA512 as hash algorithm H.

ED638
: `TPM_ECC_BN_P638` curve, using SHA512 as hash algorithm H.

4.3 ecdaaSignature object

The fields c and s both have length N. The fields R, S, T, W have equal length (2*N+1 each).

In the case of BN_P256 curve (with key length N=32 bytes), the fields R, S, T, W have length 2*32+1=65 bytes. The fields c and s have length N=32 each.

The ecdaaSignature object is a binary object generated as the concatenation of the binary fields in the order described below (total length of 324 bytes for 256bit curves):

Value	Length (in Bytes)	Description
UINT8[] ECDAA_Signature_c	N	The c value, c=H(U I S I W IKRD I AppID) as returned by AuthnrEcdaaSign encoded as byte string according to BigNumberToB. Where • $U = S^r$, with $r = RAND(p)$ computed by the signer. • KRD is the the entire to-be-signed object (e.g. TAG_UAFV1_KRD in the case of FIDO UAF). • $S = B^l$, with $l = RAND(p)$ computed by the signer and $B = A^y$ computed in the ECDAA-Join

Value	Length (in Bytes)	Description
UINT8[] ECDAA_Signature_s	N	The s value, s=r + c * sk (mod p), as returned by AuthnrEcdaaSign encoded as byte string according to BigNumberToB. Where • r = RAND(p), computed by the signer at FIDO registration (see 3.5.2 ECDAA-Sign Split between Authenticator and ASM) • p is the group order of G1 • sk: is the authenticator's attestation secret key, see above
UINT8[] ECDAA_Signature_R	2*N+1	$R = A^l$; computed by the ASM or the authenticator at FIDO registration; encoded as byte string according to ECPointToB. Where • l = RAND(p), i.e. random number 0≤l≤p. Computed by the ASM or the authenticator at FIDO registration. • And where $R = A^l$ denotes the scalar multiplication (of scalar l) of a curve point A. • Where A has been provided by the ECDAA-Issuer as part of ECDAA-Join: $A = P_1^{l_J}$, see 3.4.1 ECDAA-Join Algorithm. • Where P_1 and p are system values, injected into the authenticator and l_J is a random number computed by the ECDAA-Issuer on Join.
UINT8[] ECDAA_Signature_S	2*N+1	$S = B^l$; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where B has been provided by the ECDAA-Issuer on Join: $B = A^y$, see 3.4.1 ECDAA-Join Algorithm.
UINT8[] ECDAA_Signature_T	2*N+1	$T = C^l$; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where • $C = A^x \cdot Q^{xyl_J}$, provided by the ECDAA-Issuer on Join • $l_J = RAND(p)$ computed by the ECDAA-Issuer at Join (see 3.4.1 ECDAA-Join Algorithm) • x and y are components of the ECDAA-Issuer private key, iskk=(x,y). • Q is the authenticator public key
UINT8[] ECDAA_Signature_W	2*N+1	$W = D^l$; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where $D = Q^{l_J y}$ is computed by the ECDAA-Issuer at Join (see 3.4.1 ECDAA-Join Algorithm).

5. Considerations

This section is non-normative.

A detailed security analysis of this algorithm can be found in [FIDO-DAA-Security-Proof].

5.1 Algorithms and Key Sizes

The proposed algorithms and key sizes are chosen such that compatibility to TPMv2 is possible.

5.2 Indicating the Authenticator Model

Some authenticators (e.g. TPMv2) do not have the ability to include their model (i.e. vendor ID and model name) in attested messages (i.e. the to-be-signed part of the registration assertion). The TPM's endorsement key certificate typically contains that information directly or at least it allows the model to be derived from the endorsement key certificate.

In FIDO, the relying party expects the ability to cryptographically verify the authenticator model.

We require the ECDAA-Issuers public key (ipk=(X,Y,c,sx,sy)) to be dedicated to one single authenticator model (e.g. as identified by AAID or AAGUID).

5.3 Revocation

If the private ECDAA attestation key sk of an authenticator has been leaked, it can be revoked by adding its value to a RogueList.

The ECDAA-Verifier (i.e. FIDO Server) check for such revocations. See section 3.6 ECDAA-Verify Operation.

The ECDAA-Issuer is expected to check revocation by other means:

1. if ECDAA-Join is done in-factory, it is assumed that produced devices are known to be uncompomised (at time of production).
2. if a remote ECDAA-Join is performed, the (remote) ECDAA-Issuer already must use a different method to remotely authenticate the authenticator (e.g. using some endorsement key). We expect the ECDAA-Issuer to perform a revocation check based on that information. This is even more flexible as it does not require access to the authenticator ECDAA private key sk.

5.4 Pairing Algorithm

The pairing algorithm e needs to be used by the ASM as part of the Join process and by the verifier (i.e. FIDO relying party) as part of the verification (i.e. FIDO registration) process.

The result of such a pairing operation is only compared to the result of another pairing operation computed by the same entity. As a consequence, it doesn't matter whether the ASM and the verifier use the exact same pairings or not (as long as they both use valid pairings).

5.5 Performance

For performance reasons the calculation of Sig2=(R, S, T, W) may be performed by the ASM running on the FIDO user device (as opposed to inside the authenticator). See section 3.5.2 ECDAA-Sign Split between Authenticator and ASM.

The cryptographic computations to be performed inside the authenticator are limited to G1. The ECDAA-Issuer has to perform two G2 point multiplications for computing the public key. The Verifier (i.e. FIDO relying party) has to perform G1 operations and two pairing operations.

5.6 Binary Concatentation

We use a simple byte-wise concatenation function for the different parameters, i.e. H(a,b) = H(a | b).

This approach is as secure as the underlying hash algorithm since the authenticator controls the length of the (fixed-length) values (e.g. U, S, W). The AppID is provided externally and has unverified structure and length. However, it is only followed by a fixed length entry - the (system defined) hash of KRD. As a consequence, no parts of the AppID would ever be confused with the fixed length value.

5.7 IANA Considerations

This specification registers the algorithm names "ED256", "ED512", and "ED638" defined in section 4. FIDO ECDAA Object Formats and Algorithm Details with the IANA JSON Web Algorithms registry as defined in section "Cryptographic Algorithms for Digital Signatures and MACs" in [RFC7518].

Algorithm Name	"ED256"
Algorithm Description	FIDO ECDAA algorithm based on TPM_ECC_BN_P256 [TPMv2-Part4] curve using SHA256 hash algorithm.
Algorithm Usage	

Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

Algorithm Name	"ED512"
Algorithm Description	ECDAA algorithm based on ECC_BN_ISOP512 [ISO15946-5] curve using SHA512 algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

Algorithm Name	"ED638"
Algorithm Description	ECDAA algorithm based on TPM_ECC_BN_P638 [TPMv2-Part4] curve using SHA512 algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

A. References

A.1 Normative references

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

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

[RFC3447]
J. Jonsson; B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. February 2003. Informational. URL:https://tools.ietf.org/html/rfc3447

[TPMv2-Part4]

Trusted Computing Group, *Trusted Platform Module Library, Part 4: Supporting Routines*, URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C6CABBC-1A4B-B294-D0DA8CE1B452CAB4/TPM%20Rev%202.0%20Part%204%20-%20Supporting%20Routines%2001.16-code.pdf

A.2 Informative references

[ANZ-2013]
Tolga Acar, Lan Nguyen and Greg Zaverucha, *A TPM Diffie-Hellman Oracle*, October 18, 2013., Microsoft Research, Redmond, WA. URL: http://eprint.iacr.org/2013/667.pdf

[Arthur-Challener-2015]
Will Arthur and David Challener with Kenneth Goldman, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*, 2014, URL: http://www.apress.com/9781430265832

[BFGSW-2011]
D. Bernhard, G. Fuchsbauer, E. Ghadafi, N. P. Smart and B. Warinschi, *Anonymous attestation with user-controlled linkability*, 2011, URL: http://eprint.iacr.org/2011/658.pdf

[BarNae-2006]
Paulo S. L. M. Barreto and Michael Naehrig, *Pairing-Friendly Elliptic Curves of Prime Order*, 2006, URL: http://research.microsoft.com/pubs/118425/pfcpo.pdf

[BriCamChe2004-DAA]
Ernie Brickell, Intel Corporation; Jan Camenisch, IBM Research; Liqun Chen, HP Laboratories, *Direct Anonymous Attestation*, 2004, URL: http://eprint.iacr.org/2004/205.pdf

[CheLi2013-ECDAA]
Liqun Chen, HP Laboratories and Jiangtao Li, Intel Corporation, *Flexible and Scalable Digital Signatures in TPM 2.0*, 2013. URL: http://dx.doi.org/10.1145/2508859.2516729

[DevScoDah2007]
Augusto Jun Devegili, Michael Scott, and Ricardo Dahab *Implementing Cryptographic Pairings over Barreto-Naehrig Curves*. 2007, URL: https://eprint.iacr.org/2007/390.pdf

[FIDO-DAA-Security-Proof]
Jan Camenisch; Manu Drijvers; Anja Lehmann. *Universally Composable Direct Anonymous Attestation*. 2015. URL: https://eprint.iacr.org/2015/1246

[FIDOEcdaaAlgorithm]
R. Lindemann, J. Camenisch, M. Drijvers, A. Edgington, A. Lehmann, R. Urian, *FIDO ECDAA Algorithm*. FIDO Alliance Implementation Draft. URLs:
HTML: fido-ecdaa-v1.1-id-20170202.html
PDF: fido-ecdaa-v1.1-id-20170202.pdf.

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[ISO15946-5]
ISO/IEC 15946-5, *Information Technology - Security Techniques - Cryptographic techniques based on elliptic curves - Part 5: Elliptic curve generation*, URL: https://webstore.iec.ch/publication/10468

[RFC7515]
M. Jones, J. Bradley, N. Sakimura, *JSON Web Signature (JWS) (RFC7515)*, IETF, May 2015, URL: http://www.ietf.org/rfc/rfc7515.txt

[RFC7518]
M. Jones. *JSON Web Algorithms (JWA)*. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7518

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

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

[TPMv2-Part2]
Trusted Computing Group, *Trusted Platform Module Library, Part 2: Structures*, URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C583202-1A4B-B294-D0469592DB10A6CD/TPM%20Rev%202.0%20Part%202%20-%20Structures%2001.16.pdf

[XYZF-2014]
Li Xi, Kang Yang, Zhenfeng Zhang, and Dengguo Feng, *DAA-Related APIs in TPM 2.0 Revisited*, 2014, in T. Holz and S. Ioannidis (Eds.): TRUST 2014, LNCS 8564, pp. 1–18, 2014.

FIDO Security Reference

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-security-ref-v1.1-id-20170202.html
Previous version:
http://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-security-ref-v1.0-ps-20141208.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal, Inc.

Abstract

This document analyzes the FIDO security. The analysis is performed on the basis of the FIDO Universal Authentication Framework (UAF) specification and FIDO Universal 2nd Factor (U2F) specifications as of the date of this publication.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

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

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. Introduction

This document analyzes the security properties of the FIDO UAF and U2F families of protocols. Although a brief architectural summary is provided below, readers should familiarize themselves with the the FIDO Glossary of Terms [FIDOGlossary] for definitions of terms used throughout. For technical details of various aspects of the architecture, readers should refer to the FIDO Alliance specifications in the Bibliography.

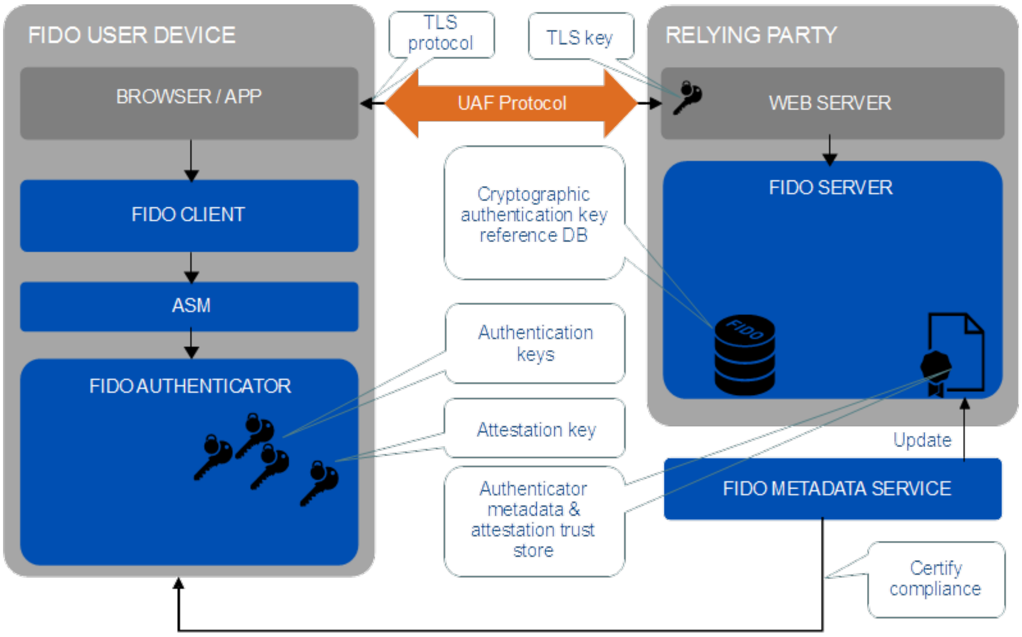


Fig. 1 FIDO Reference Architecture

Conceptually, FIDO involves a conversation between a computing environment controlled by a Relying Party and one controlled by the user to be authenticated. The Relying Party's environment consists conceptually of at least a web server and the server-side portions of a web application, plus a FIDO Server. The FIDO Server has a trust store, containing the (public) trust anchors for the attestation of FIDO Authenticators. The user's environment, referred to as the FIDO user device, consists of one or more FIDO Authenticators, a piece of software called the FIDO Client that is the endpoint for UAF and U2F conversations, and User Agent software. The User Agent software may be a browser hosting a web application delivered by the Relying Party, or it may be a standalone application delivered by the Relying Party. In either case, the FIDO Client, while a conceptually distinct entity, may actually be implemented in whole or part within the boundaries of the User Agent.

2.1 Intended Audience

This document assumes a technical audience that is proficient with security analysis of computing systems and network protocols as well as the specifics of the FIDO architecture and protocol families. It discusses the security goals, security measures, security assumptions and a series of threats to FIDO systems, including the user's computing environment, the Relying Party's computing environment, and the supply chain, including the vendors of FIDO components.

3. Attack Classification

We want to distinguish the following threat classes (all leading to the impersonation of the user):

1. Automated attacks focused on relying parties, which affect the user but cannot be prevented by the user
2. Automated attacks which are performed once and lead to the ability to impersonate the user on an on-going basis without involving him or his device directly.
3. Automated attacks which involve the user or his device for each successful impersonation.
4. Automated attacks to sessions authenticated by the user.
5. Not automatable attacks to the user or his device which are performed once and lead to the ability to impersonate the user on an on-going basis without involving him or his device directly.
6. Not automatable attacks to the user or his device which involve the user or his device for each successful impersonation.

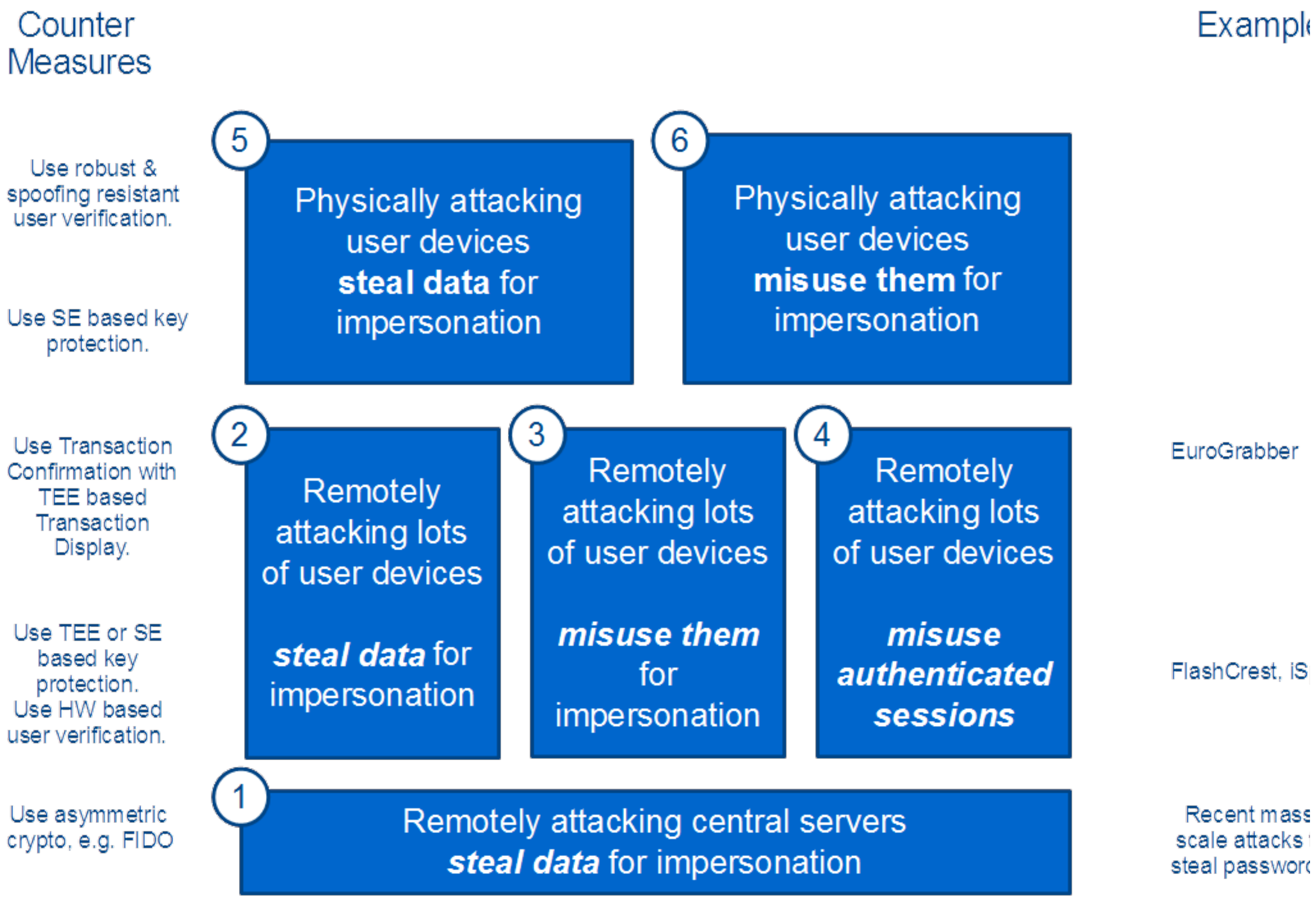


Fig. 2 Attack Classes

The first four attack classes are considered scalable as they are automated (or at least can be automated). The attack classes 5 and 6 are not automatable; they involve some kind of manual/physical interaction of the attacker with the user or his device. We will attribute the threats analyzed in this document with the related attack class (AC1 – AC6).

NOTE

1. FIDO UAF uses asymmetric cryptography to protect against this class of attacks. This gives control back to the user, i.e. when using good random numbers, the user's authenticator can make breaking the key as hard as the underlying factoring (in the case of RSA) or discrete logarithm (in the case of DSA or ECDSA) problem.
2. Once counter-measures for this kind of attack are commonly in place, attackers will likely focus on another attack class.
3. The numbers at the attack classes do not imply a feasibility ranking of the related attacks, e.g. it is not necessarily more difficult to perform (4) than it is to perform (3).
4. Feasibility of attack class (1) cannot be influenced by the user at all. This makes this attack class really bad.
5. The concept of physical security (i.e. "protect your Authenticator from being stolen"), related to attack classes (5) and (6) is much better internalized by users than the concept of logical security, related to attack classes (2), (3) and (4).
6. In order to protect against misuse of authenticated sessions (e.g. MITB attacks), the FIDO Authenticator must support the concept of transaction confirmation and the relying party must use it.
7. For an attacker to succeed, any attack class is sufficient.

4. UAF Security Goals

In this section the specific security goals of UAF are described. The FIDO UAF protocol [UAFProtocol] supports a variety of different FIDO Authenticators. Even though the security of those authenticators varies, the UAF protocol and the FIDO Server should provide a very high level of security - at least on a conceptual level. In reality it might require a FIDO Authenticator with a high security level in order to fully leverage the UAF security strength.

NOTE

In certain environments the overall security of the explicit authentication (as provided by FIDO) is less important, as it might be supplemented with a high degree of implicit authentication or the application doesn't even require a high level of authentication strength.

The FIDO U2F protocol [U2FOverview] supports a more constrained set of Authenticator capabilities. It shares the same security goals as UAF, with the exception of [SG-14] Transaction Non- Repudiation. The UAF protocol has the following security goals:

[SG-1]
Strong User Authentication: Authenticate (i.e. recognize) a user and/or a device to a relying party with high (cryptographic) strength.
[SG-2]
Credential Guessing Resilience: Provide robust protection against eavesdroppers, e.g. *be resilient to physical observation, resilient to targeted impersonation, resilient to throttled and unthrottled guessing.*
[SG-3]
Credential Disclosure Resilience: Be *resilient to phishing attacks* and real-time phishing attack, including resilience to online attacks by adversaries able to actively manipulate network traffic.

[SG-4]
Unlinkablity: Protect the protocol conversation such that any two relying parties cannot link the conversation to one user (i.e. be *unlinkable*).

[SG-5]
Verifier Leak Resilience: Be *resilient to leaks from other relying parties* I.e., nothing that a verifier could possibly leak can help an attacker impersonate the user to another relying party.

[SG-6]
Authenticator Leak Resilience: Be resilient to leaks from other FIDO Authenticators. I.e., nothing that a particular FIDO Authenticator could possibly leak can help an attacker to impersonate any other user to any relying party.

[SG-7]
User Consent: Notify the user before a relationship to a new relying party is being established (*requiring explicit consent*).

[SG-8]
Limited PII: Limit the amount of personal identifiable information (PII) exposed to the relying party to the absolute minimum.

[SG-9]
Attestable Properties: Relying Party must be able to verify FIDO Authenticator model/type (in order to calculate the associated risk).

[SG-10]
DoS Resistance: Be resilient to *Denial of Service Attacks* I.e. prevent attackers from inserting invalid registration information for a legitimate user for the next login phase. Afterward, the legitimate user will not be able to login successfully anymore.

[SG-11]
Forgery Resistance: Be resilient to *Forgery Attacks (Impersonation Attacks)*. I.e. prevent attackers from attempting to modify intercepted communications in order to masquerade as the legitimate user and login to the system.

[SG-12]
Parallel Session Resistance: Be resilient to *Parallel Session Attacks*. Without knowing a user's authentication credential, an attacker can masquerade as the legitimate user by creating a valid authentication message out of some eavesdropped communication between the user and the server.

[SG-13]
Forwarding Resistance: Be resilient to *Forwarding and Replay Attacks*. Having intercepted previous communications, an attacker can impersonate the legal user to authenticate to the system. The attacker can replay or forward the intercepted messages.

[SG-14]
Transaction Non-Repudiation: Provide strong cryptographic non-repudiation for secure transactions.

[SG-15]
Respect for Operating Environment Security Boundaries: Ensure that registrations and key material as a shared system resource is appropriately protected according to the operating environment privilege boundaries in place on the FIDO user device.

> NOTE
>
> For a definition of the phrases printed in *italics*, refer to [QuestToReplacePasswords] and to [PasswordAuthSchemesKeyIssues]

4.1 Assets to be Protected

Independent of any particular implementation, the UAF protocol assumes some assets to be present and to be protected.

1. Cryptographic Authentication Key. Typically keys in FIDO are unique for each tuple of (relying party, user account, authenticator).
2. Cryptographic Authentication Key Reference. This is the cryptographic material stored at the relying party and used to uniquely verify the Cryptographic Authentication Key, typically the public portion of an asymmetric key pair.
3. Authenticator Attestation Key(as stored in each authenticator). This should only be usable to attest a Cryptographic Authentication Key and the type and manufacturing batch of an Authenticator. Attestation keys and certificates are shared by a large number of authenticators in a device class from a given vendor in order to prevent their becoming a linkable identifier across relying parties. Authenticator attestation certificates may be self-signed, or signed by an authority key controlled by the vendor.
4. Authenticator Attestation Authority Key. An authenticator vendor may elect to sign authenticator attestation certificates with a per-vendor certificate authority key.
5. Authenticator Attestation Authority Certificate. Contained in the initial/default trust store as part of the FIDO Server and contained in the active trust store maintained by each relying party.
6. Active Trust Store. Contains all trusted attestation master certificates for a given FIDO server.
7. All data items suitable for uniquely identifying the authenticator across relying parties. An attack on those would break the non-linkability security goal.
8. Private key of Relying Party TLS server certificate.
9. TLS root certificate trust store for the user's browser/app.

5. FIDO Security Measures

> NOTE
>
> Particular implementations of FIDO Clients, Authenticators, Servers and participating applications may not implement all of these security measures (e.g. Secure Display, [SM-10] Transaction Confirmation) and they also might (and should) implement add itional security measures.

> NOTE
>
> The U2F protocol lacks support for [SM-5] Secure Display, [SM-10] Transaction Confirmation, has only server-supplied [SM-8] Protocol Nonces, and [SM-3] Authenticator Class Attestation is implicit as there is only a single class of device.

[SM-1] (U2F + UAF)
Key Protection: Authentication key is protected against misuse. Misuse means any use violating the FIDO specification or the details given in the Metadata Statement. Before a key can be used, it requires the User to unlock it using the user verification method specified in the Authenticator Metadata Statement (Silent Authenticators do not require any user verification method).

[SM-2] (U2F + UAF)
Unique Authentication Keys: Cryptographic authentication key is specific and unique to the tuple of (FIDO Authenticator, User, Relying Party).

[SM-3] (U2F + UAF)
Authenticator Class Attestation: Hardware-based FIDO Authenticators support authenticator attestation using an attestation key using one of the FIDO specified attestation types and algorithms. Each relying party receives regular updates of the trust store (through the FIDO Metadata service).

[SM-4] (UAF)
Authenticator Status Checking: Relying Parties will be notified of compromised authenticators or authenticator attestation keys. The FIDO Server must take this information into account. Authenticator manufacturers have to inform FIDO alliance about compromised authenticators.

[SM-5] (UAF)
User Consent: FIDO Client implements a user interface for getting user's consent on any actions (except authentication with silent authenticator) and displaying RP name (derived from server URL).

[SM-6] (U2F + UAF)
Cryptographically Secure Verifier Database: The relying party stores only the public portion of an asymmetric key pair, or an encrypted key handle, as a cryptographic authentication key reference.

[SM-7] (U2F + UAF)

Secure Channel with Server Authentication: The TLS protocol with server authentication or a transport with equivalent properties is used as transport protocol for UAF. The use of https is enforced by a browser or Relying Party application.

[SM-8] (UAF)
Protocol Nonces: Both server and client supplied nonces are used for UAF registration and authentication. U2F requires server supplied nonces.

[SM-9] (U2F + UAF)
Authenticator Certification: Only Authenticators meeting certification requirements defined by the FIDO Alliance and accurately describing their relevant characteristics will have have their related attestation keys included in the default Trust Store.

[SM-10] (UAF)
Transaction Confirmation (WYSIWYS): Secure Display (WYSIWYS) (optionally) implemented by the FIDO Authenticators is used by FIDO Client for displaying relying party name and transaction data to be confirmed by the user.

[SM-11] (U2F + UAF)
Round Trip Integrity: FIDO server verifies that the transaction data related to the server challenge received in the UAF message from the FIDO client is identical to the transaction data and server challenge delivered as part of the UAF request message.

[SM-12] (U2F + UAF)
Channel Binding: Relying Party servers may verify the continuity of a secure channel with a client application.

[SM-13] (UAF)
Key Handle Access Token: Authenticators not intended to roam between untrusted systems are able to constrain the use of registration keys within the privilege boundaries defined by the operating environment of the user device. (per-user, or perapplication, or per-user + per-application as appropriate)

[SM-14] (U2F + UAF)
AppID Separation: A Relying Party can declare the application identities allowed to access its registered keys, for operating environments on user devices that support this concept.

[SM-15] (U2F + UAF)
Signature Counter: Authenticators send a monotonically increasing signature counter that a Relying Party can check to possibly detect cloned authenticators.

5.1 Relation between Measures and Goals

Security Goal	Supporting Security Measures
[SG-1] Strong User Authentication	[SM-1] Key Protection [SM-12] Channel Binding [SM-14] AppID Separation [SM-15] Signature Counter
[SG-2] Credential Guessing Resilience	[SM-1] Key Protection [SM-6] Cryptographically Secure Verifier Database
[SG-3] Credential Disclosure Resilience	[SM-1] Key Protection [SM-9] Authenticator Certification [SM-15] Signature Counter
[SG-4] Unlinkability	[SM-2] Unique Authentication Keys [SM-3] Authenticator Class Attestation
[SG-5] Verifier Leak Resilience	[SM-2] Unique Authentication Keys [SM-6] Cryptographically Secure Verifier Database
[SG-6] Authenticator Leak Resilience	[SM-9] Authenticator Certification [SM-15] Signature Counter
[SG-7] User Consent	[SM-1] Key Protection [SM-5] User Consent [SM-7] Secure Channel with Server Authentication [SM-10] Transaction Confirmation (WYSIWYS)
[SG-8] Limited PII	[SM-2] Unique Authentication Keys
[SG-9] Attestable Properties	[SM-3] Authenticator Class Attestation [SM-4] Authenticator Status Checking [SM-9] Authenticator Certification
[SG-10] DoS Resistance	[SM-8] Protocol Nonces
	[SM-7] Secure Channel with Server Authentication

Security Goal	Supporting Security Measures
[SG-11] Forgery Resistance	[SM-8] Protocol Nonces [SM-11] Round Trip Integrity [SM-12] Channel Binding
[SG-12] Parallel Session Resistance	[SM-7] Secure Channel with Server Authentication [SM-8] Protocol Nonces [SM-11] Round Trip Integrity [SM-12] Channel Binding
[SG-13] Forwarding Resistance	[SM-7] Secure Channel with Server Authentication [SM-8] Protocol Nonces [SM-11] Round Trip Integrity [SM-12] Channel Binding
[SG-14] Transaction Non-Repudiation	[SM-1] Key Protection [SM-2] Unique Authentication Keys [SM-8] Protocol Nonces [SM-9] Authenticator Certification [SM-10] Transaction Confirmation (WYSIWYS) [SM-11] Round Trip Integrity [SM-12] Channel Binding
[SG-15] Respect for Operating Environment Security Boundaries	[SM-13] Key Handle Access Token [SM-14] AppID Separation

6. UAF Security Assumptions

Today’s computer systems and cryptographic algorithms are not provably secure. In this section we list the security assumptions, i.e. assumptions on security provided by other components. A violation of any of these assumptions will prevent reliable achievement of the Security Goals.

[SA-1]
: The cryptographic algorithms and parameters (key size, mode, output length, etc.) in use are not subject to unknown weaknesses that make them unfit for their purpose in encrypting, digitally signing, and authenticating messages.

[SA-2]
: Operating system privilege separation mechanisms relied up on by the software modules involved in a FIDO operation on the user device perform as advertised. E.g. boundaries between user and kernel mode, between user accounts, and between applications (where applicable) are securely enforced and security principals can be mutually, securely identifiable.

[SA-3]
: Applications on the user device are able to establish secure channels that provide trustworthy server authentication, and confidentiality and integrity for messages (e.g., through TLS).

[SA-4]
: The secure display implementation is protected against spoofing and tampering.

[SA-5]
: The computing environment on the FIDO user device and the and applications involved in a FIDO operation act as trustworthy agents of the user.

[SA-6]
: The inherent value of a cryptographic key resides in the confidence it imparts, and this commodity decays with the passage of time, irrespective of any compromise event. As a result the effective assurance level of authenticators will be reduced over time.

[SA-7]
: The computing resources at the Relying Party involved in processing a FIDO operation act as trustworthy agents of the Relying Party.

6.1 Discussion

With regard to [SA-5] and malicious computation on the FIDO user's device, only very limited guarantees can be made within the scope of these assumptions. Malicious code privileged at the level of the trusted computing base can always violate [SA-2] and [SA- 3]. Malicious code privileged at the level of the user's account in traditional multi-user environments will also likely be able to violate [SA-3].

FIDO can also provide only limited protections when a user chooses to deliberately violate [SA-5], e.g. by roaming a USB authenticator to an untrusted system like a kiosk, or by granting permissions to access all authentication keys to a malicious app in a mobile environment. Transaction Confirmation can be used as a method to protect against compromised FIDO user devices.

In to components such as the FIDO Client, Server, Authenticators and the mix of software and hardware modules they are comprised of, the end-to-end security goals also depend on correct implementation and adherence to FIDO security guidance by other participating components, including web browsers and relying party applications. Some configurations and uses may not be able to meet all security goals. For example, authenticators may lack a secure display, they may be composed only of unattestable software components, they may be deliberately designed to roam between untrusted operating environments, and some operating environments may not provide all necessary security primitives (e.g., secure IPC, application isolation, modern TLS implementations, etc.)

7. Threat Analysis

7.1 Threats to Client Side

7.1.1 Exploiting User’s pattern matching weaknesses

T-1.1.1	Homograph Mis-Registration	Violates
AC3	The user registers a FIDO authentication key with a fraudulent web site instead of the genuine Relying Party. **Consequences:** The fraudulent site may convince the user to disclose a set of non-FIDO credentials sufficient to allow the attacker to register a FIDO Authenticator under its own control, at the genuine Relying Party, on the user's behalf, violating [SG-1] Strong User Authentication. **Mitigations:** Disclosure of non-FIDO credentials is outside of the scope of the FIDO security measures, but Relying Parties should be aware that the initial strength of an authentication key is no better than the identity-proofing applied as part of the registration process.	SG-1

7.1.2 Threats to the User Device, FIDO Client and Relying Party Client Applications

T-1.2.1	FIDO Client Corrpution	Violates
AC3	Attacker gains ability to execute code in the security context of the FIDO Client. **Consequences:** Violation of [SA-5]. **Mitigations:** When the operating environment on the FIDO user device allows, the FIDO Client should operate in a privileged and isolated context under [SA-2] to protect itself from malicious modification by anything outside of the Trusted Computing Base.	SA-5

T-1.2.2	Logical/Physical User Device Attack	Violates
AC3 / AC5	Attacker gains physical access to the FIDO user device but not the FIDO Authenticator. **Consequences:** Possible violation of [SA-5] by installing malicious software or otherwise tampering with the FIDO user device. **Mitigations:** [SM-1] Key Protection prevents the disclosure of authentication keys or other assets during a transient compromise of the FIDO user device. A persistent compromise of the FIDO user device can lead to a violation of [SA-5] unless additional protection measures outside the scope of FIDO are applied to the FIDO user device. (e,g. whole disk encryption and boot-chain integrity)	SA-5

T-1.2.3	User Device Account Access	Violates
AC3 / AC4	Attacker gains access to a user's login credentials on the FIDO user device. **Consequences:** Authenticators might be remotely abused, or weakly-verifying authenticators might be locally abused, violating [SG-1] Strong User Authentication and [SG-13] Transaction Non-Repudiation. Possible violation of [SA-5] by the installation of malicious software. **Mitigations:** Relying Parties can use [SM-9] Authenticator Certification and [SM-3] Authenticator Class Attestation to determine the nature of authenticators and not rely on weak, or weakly-verifying authenticators for high value operations.	SG-1, SG-13; SA-5

T-1.2.4	App Server Verification Error	Violates
AC3	A client application fails to properly validate the remote sever identity, accepts forged or stolen credentials for a remote server, or allows weak or missing cryptographic protections for the secure channel. **Consequences:** An active network adversary can modify the Relying Party's authenticator policy and downgrade the client's choice of authenticator to make it easier to attack. An active network adversary can intercept or view FIDO messages intended for the Relying Party. It may be able to use this ability to violate [SG-12] Parallel Session Resistance, [SG-11] Forgery Resistance or [SG-13] Forwarding Resistance, **Mitigations:** The server can verify [SM-8] Protocol Nonces to detect replayed messages and protect from an adversary that can read but not modify traffic in a secure channel. The server can mandate a channel with strong cryptographic protections to prevent message forgery and can verify a [SM-12] Channel Binding to detect forwarded messages.	SG-11, SG-12, SG-13

T-1.2.5	RP Web App Corruption	Violates
	An attacker is able to obtain malicious execution in the security context of the Relying Party application (e.g. via Cross-Site Scripting) or abuse the secure channel or session identifier after the user has successfully authenticated. **Consequences:** The attacker is able to control the user's session, violating [SG-14] Transaction Non-Repudiation. **Mitigations:** The server can employ [SM-10] Transaction Confirmation to gain additional assurance for high value operations.	SG-14

T-1.2.6	Fingerprinting Authenticators	Violates
	A remote adversary is able to uniquely identify a FIDO user device using the fingerprint of discoverable configuration of its FIDO Authenticators. **Consequences:** The exposed information violates [SG-8] Limited PII, allowing an adversary to violate [SG-7] User Consent by strongly authenticating the user without their knowledge and [SG-4] Unlinkablity by sharing that fingerprint. **Mitigations:** [SM-3] Authenticator Class Attestation ensures that the fingerprint of an Authenticator will not be unique. For web browsing situations where this threat is most prominent, user agents may provide additional user controls around the discoverability of FIDO Authenticators.	SG-4, SG7, SG-8

T-1.2.6	Fingerprinting Authenticators	Violates

T-1.2.7	App to FIDO Client full MITM attack	Violates
AC3	Malicious software on the FIDO user device is able to read, tamper with, or spoof the endpoint of inter-process communication channels between the FIDO Client and browser or Relying Party application. **Consequences:** Adversary is able to subvert [SA-2]. **Mitigations:** On platforms where [SA-2] is not strong the security of the system may depend on preventing malicious applications from arriving on the FIDO user device. Such protections, e.g. app store policing, are outside the scope of FIDO. When using [SM-10] Transaction Confirmation, the user would see the relevant AppID and transaction text and decide whether or not to accept an action.	SA-2

T-1.2.8	Authenticator to App Read-Only MITM attack	Violates
AC3	An adversary is able to obtain an authenticator's signed protocol response message. **Consequences:** The attacker attempts to replay the message to authenticate as the user, violating [SG-1] Strong User Authentication, [SG-13] Forwarding Resistance and [SG-12] Parallel Session Resistance. **Mitigations:** The server can use [SM-8] Protocol Nonces to detect replay of messages and verify [SM-11] Round Trip Integrity to detect modified messages.	SG-1, SG-12, SG-13

T-1.2.9	Malicious App	Violates
AC3	A user installs an application that represents itself as being associated with to one Relying Party application but actually initiates a protocol conversation with a different Relying Party and attempts to abuse previously registered authentication keys at that Relying Party. **Consequences:** Adversary is able to violate [SG-7] User Consent by misrepresenting the target of authentication. Other consequences equivalent to [T-1.2.5] **Mitigations:** If a [SM-5] Transaction Confirmation Display is present, the user may be able to verify the true target of an operation. If the malicious application attempts to communicate directly with an Authenticator that uses [SM-13] KeyHandleAccessToken, it should not be able to access keys registered by other FIDO Clients. If the operating environment on the FIDO user device supports it, the FIDO client may be able to determine the application's identity and verify if it is authorized to target that Relying Party using a [SM-14] AppID Separation.	SG-7

T-1.2.10	Phishing Attack	Violates
	A Phisher convinces the user to enter his PIN used for user verification into an application / web site disclosing the PIN to the Phisher. In the traditional username/password world this enables the attacker to successfully impersonate the user (to the relying party). **Consequences:** None as the phisher additionally would need access to the Authenticator in order to pass user verification [SM-1]. In FIDO, the user verification PIN (if user verification is done via PIN) is not known to the relying party and hence isn't sufficient for user impersonation. If user verification is done using an alternative user verification method, this applies accordingly. **Mitigations:** In FIDO, the Uauth.priv key is used to sign a relying party supplied challenge. without (use) access to that key, no impersonation is possible.	

7.1.3 Creating a Fake Client

T-1.3.1	Malicious FIDO Client	Violates
AC3	Attacker convinces users to install and use a malicious FIDO Client. **Consequences:** Violation of [SA-5] **Mitigations:** Mitigating malicious software installation is outside the scope of FIDO. If an authenticator implements [SM-1] Key Protection, the user may be able to recover full control of their registered authentication keys by removing the malicious software from their user device. When using [SM-10] Transaction Confirmation, the user sees the real AppIDs and transaction text and can decide to accept or reject the action.	SA-5

7.1.4 Threats to FIDO Authenticator

T-1.4.1	Malicious Authenticator	Violates
AC2	Attacker convinces users to use a maliciously implemented authenticator. **Consequences:** The fake authenticator does not implement any appropriate security measures and is able to violate all security goals of FIDO. **Mitigations:** A user may be unable to distinguish a malicious authenticator, but a Relying Party can use [SM-3] Authenticator Class Attestation to identify and only allow registration of reliable authenticators that have passed [SM-9] Authenticator Certification A Relying Party can additionally rely on [SM-4] Authenticator Status Checking to check if an attestation presented by a malicious authenticator has been marked as compromised.	SG-1

T-1.4.2	Uauth.priv Key Compromise	Violates
AC2	Attacker succeeds in extracting a user's cryptographic authentication key for use in a different context. **Consequences:** The attacker could impersonate the user with a cloned authenticator that does not do trustworthy user verification, violating [SG-1]. **Mitigations:** [SM-1] Key Protection measures are intended to prevent this. Relying Parties can check [SM-9] Authenticator Certification attributes to determine the type of key protection in use by a given authenticator class. Relying Parties can additionally verify the [SM-15] Signature Counter and detect that an authenticator has been cloned if it ever fails to advance relative to the prior operation.	SG-1

T-1.4.3	User Verification By-Pass	Violates
AC3	Attacker could use the cryptographic authentication key (inside the authenticator) either with or without being noticed by the legitimate user. **Consequences:** Attacker could impersonate user, violating [SG-1]. **Mitigations:** A user can only register and a Relying Party only allow authenticators that perform [SM-1] Key Protection with an appropriately secure user verification process. Does not apply to Silent Authenticators.	SG-1

T-1.4.4	Physical Authenticator Attack	Violates
AC5 / AC6	Attacker could get physical access to FIDO Authenticator (e.g. by stealing it). **Consequences:** Attacker could launch offline attack in order to use the authentication key. If this offline attack succeeds, the attacker could successfully impersonate the user, violating [SG-1] Strong User Authentication. Attacker can introduce a low entropy situation to recover an ECDSA signature key (or optherwise extract the Uauth.priv key), violating [SG-9] Attestable Properties if the attestation key is targeted or [SG-1] Strong User Authentication if a user key is targeted. **Mitigations:** [SM-1] Key Protection includes requirements to implement strong protections for key material, including resistance to offline attacks and low entropy situations. Relying Parties should use [SM-3] Authenticator Class Attestation to only accept Authenticators implementing a sufficiently strong user verification method.	SG-1

T-1.4.6	Fake Authenticator	Violates
	Attacker is able to extract the authenticator attestation key from an authenticator, e.g. by neutralizing physical countermeasures in a laboratory setting. **Consequences:** Attacker can violate [SG-9] Attestable Properties by creating a malicious hardware or software device that represents itself as a legitimate one. **Mitigations:** Relying Parties can use [SM-4] Authenticator Status Checking to identify known-compromised keys. Identification of such compromise is outside the strict scope of the FIDO protocols.	SG-9

T-1.4.7	Transaction Confirmation Display Overlay Attack	Violates
	Attacker is able to subvert [SM-5] Secure Display functionality (WYSIWYS), perhaps by overlaying the display with false information. **Consequences:** Violation of [SG-14] Transaction Non-Repudiation. **Mitigations:** Implementations must take care to protect [SA-4] in their implementation of a secure display, e.g. by implementing a distinct hardware display or employing appropriate privileges in the operating environment of the user device to protect against spoofing and tampering. [SM-9] Authenticator Certification will provide Relying Parties with metadata about the nature of a transaction confirmation display information that can be used to assess whether it matches the assurance level and risk tolerance of the Relying Party for that particular transaction.	SG-14

T-1.4.8	Signature Algorithm Attack	Violates
AC2	A cryptographic attack is discovered against the public key cryptography system used to sign data by the FIDO authenticator. **Consequences:** Attacker is able to use messages generated by the client to violate [SG-2] Credential Guessing Resistance **Mitigations:** [SM-8] Protocol Nonces, including client-generated entropy, limit the amount of control any adversary has over the internal structure of an authenticator. [SM-1] Key Protection for non-silent authenticators requires user interaction to authorize any operation performed with the authentication key, severely limiting the rate at which an adversary can perform adaptive cryptographic attacks.	SG-2

T-1.4.9	Abuse Functionality	Violates
	It might be possible for an attacker to abuse the Authenticator functionality by sending commands with invalid parameters or invalid commands to the Authenticator.	

T-1.4.9	Abuse Functionality	Violates
	Consequences: This might lead to e.g. user verification by-pass or potential key extraction. **Mitigations:** Proper robustness (e.g. due to testing) of the Authenticator firmware.	SG-1

T-1.4.10	Random Number prediction	Violates
	It might be possible for an attacker to get access to information allowing the prediction of RNG data. **Consequences:** This might lead to key compromise situation (T-1.4.2) when using ECDSA (if the k value is used multiple times or if it is predictable). **Mitigations:** Proper robustness of the Authenticator's RNG and verification of the relevant operating environment parameters (e.g. temperature, ...).	SG-1

T-1.4.11	Firmware Rollback	Violates
	Attacker might be able to install a previous and potentially buggy version of the firmware. **Consequences:** This might lead to successful attacks, e.g. T-1.4.9. **Mitigations:** Proper robustness firmware verification method.	SG-1

T-1.4.12	User Verification Data Injection	Violates
AC3, AC6	Attacker might be able to inject pre-captured user verification data into the Authenticator. For example, if a password is used as user verification method, the attacker could capture the password entered by the user and then send the correct password to the Authenticator (by-passing the expected keyboard/PIN pad). Passwords could be captured ahead of the attack e.g. by convincing the user to enter the password into a malicious app ("phishing") or by spying directly or indirectly the password data. In another example, some malware could play an audio stream which would be recorded by the microphone and used by a Speaker-Recognition based Authenticator. **Consequences:** This might lead to successful user impersonation (if the attacker has access to valid user verification data). **Mitigations:** Use a physically secured user verification input method, e.g. Fingerprint Sensor or Trusted-User-Interface for PIN entry which cannot be by-passed by malware.	SG-1

T-1.4.13	Verification Reference Data Modification	Violates
AC3, AC6	The Attacker gained logical or physical access to the Authenticator and modifies Verification Reference Data (e.g. hashed PIN value, fingerprint templates) stored in the Authenticator and adds reference data known to or reproducible by the attacker. **Consequences:** The attacker would be recognized as the legitimate User and could impersonate the user. **Mitigations:** Proper protection of the the verification reference data in the Authenticator.	SG-1

T-1.4.14	Read access to captured user verification data	Violates
AC3, AC6	The Attacker gained read access to the captured user verification dat (e.g. PIN, fingerprint image, ...). **Consequences:** The attacker gets access to PII and could disclose it violating SG-8. **Mitigations:** Limiting access to the user verification data to the Authenticator exclusively.	SG-8

7.2 Threats to Relying Party

7.2.1 Threats to FIDO Server Data

T-2.1.1	FIDO Server DB Read Attack	Violates
	Attacker could obtains read-access to FIDO Server registration database. **Consequences:**Attacker can access all cryptographic key handles and authenticator characteristics associated with a username. If an authenticator or combination of authenticators is unique, they might use this to try to violate [SG-2] Unlinkability Attacker attempts to perform factorization of public keys by virtue of having access to a large corpus of data, violating [SG-5] Verifier Leak Resiliance and [SG-2] Credential Guessing Resilience **Mitigations:** [SM-2] Unique Authentication Keys help prevent disclosed key material from being useful against any other Relying Party, even if successfully attacked. The use of an [SM-6] Cryptographically Secure Verifier Database helps assure that it is infeasible to attack any leaked verifier keys. [SM-9] Authenticator Certification should help prevent authenticators with poor entropy from entering the market, reducing the likelihood that even a large corpus of key material will be useful in mounting attacks.	SG-2, SG-5

T-2.1.2	FIDO Server DB Modification Attack	Violates
	Attacker gains write-access to the FIDO Server registration database. **Consequences:** Violation of [SA-7] The attacker may inject a key registration under its control, violating [SG-1] Strong User Authentication	SA-7

T-2.1.2	FIDO Server DB Modification Attack	Violates
	Mitigations: Mitigating such attacks is outside the scope of the FIDO specifications. The Relying Party must maintain the integrity of any information it relies up on to identify a user as part of [SA-7].	

T-2.2.1	WebApp Malware	Violates
	Attacker gains ability to execute code in the security context of the Relying Party web application or FIDO Server. **Consequences:** Attacker is able to violate [SG-1], [SG-10], [SG-9] and any other Relying Party controls. **Mitigations:** The consequences of such an incident are limited to the relationship between the user and that particular Relying Party by [SM-1], [SM-2], and [SM-5]. Even within the Relying Party to user relationship, a user can be protected by [SM-10] Transaction Confirmation if the compromise does not include to the user's computing environment	SG-1, SG-9, SG-10

7.3 Threats to the Secure Channel between Client and Relying Party

7.3.1 Exploiting Weaknesses in the Secure Transport of FIDO Messages

FIDO takes as a base assumption that [SA-3] applications on the user device are able to establish secure channels that provide trustworthy server authentication, and confidentiality and integrity for messages. e.g. through TLS. [T-1.2.4] Discusses some consequences of violations of this assumption due to implementation errors in a browser or client application, but other threats exist in different layers.

T-3.1.1	TLS Proxy	Violates
	The FIDO user device is administratively configured to connect through a proxy that terminates TLS connections. The client trusts this device, but the connection between the user and FIDO server is no longer end-to-end secure. **Consequences:** Any such proxies introduce a new party into the protocol. If this party is untrustworthy, consequences may be as for [T-1.2.4] **Mitigations:** Mitigations for [T-1.2.4] apply, except that the proxy is considered trusted by the client, so certain methods of [SM-12] Channel Binding may indicate a compromised channel even in the absence of an attack. Servers should use multiple methods and adjust their risk scoring appropriately. A trustworthy client that reports a server certificate that is unknown to the server and does not chain to a public root may indicate a client behind such a proxy. A client reporting a server certificate that is unknown to the server but validates for the server's identity according to commonly used public trust roots is more likely to indicate [T-3.1.2]	SG-11, SG-12, SG-13

T-3.1.2	Fraudulent TLS Server Certificate	Violates
	An attacker is able to obtain control of a certificate credential for a Relying Party, perhaps from a compromised Certification Authority or poor protection practices by the Relying Party. **Consequences:**As for [T-1.2.4]. **Mitigations:**As for [T-1.2.4].	

T-3.1.3	Protocol level real-time MITM attack	Violates
	An adversary can intercept and manipulate network packages sent from the relying party to the client. The adversary uses this capability to (a) terminate the underlying TLS session from the client at the adversary and to (b) simultaneously use another TLS session from the adversary to the relying party. In the traditional username/password world, this allows the adversary to intercept the username and the password and then successfully impersonate the user at the relying party. **Consequences:** None if FIDO channelBinding [SM-12] or transaction confirmation [SM-10] are used. **Mitigations:** In the case of channelBinding [SM-12], the FIDO server will detect the MITM in the TLS channel by comparing the channel binding information provided by the client and the channel binding information retrieved locally by the server. In the case of transaction confirmation [SM-10], the user verifies and approves a particular transaction. The adversary could modify the transaction before approval. This would lead to rejection by the user. Alternatively, the adversary could modify the transaction after approval. This will break the signature in the transaction confirmation response. The FIDO Server will not accept it as a consequence.	

7.4 Threats to the Infrastructure

7.4.1 Threats to FIDO Authenticator Manufacturers

T-4.1.1	Manufacturer Level Attestation Key Compromise	Violates
	Attacker obtains control of an attestation key or attestation key issuing key. **Consequences:** Same as [T-1.4.6]: Attacker can violate [SG-9] Attestable Properties by creating a malicious hardware or software device that represents itself as a legitimate one. **Mitigations:** Same as [T-1.4.6]: Relying Parties can use [SM-4] Authenticator Status Checking to identify known-compromised keys. Identification of such compromise is outside the strict scope of the FIDO protocols.	SG-9

T-4.1.2	Malicious Authenticator HW	Violates
	FIDO Authenticator manufacturer relies on hardware or software components that generate weak cryptographic authentication key material or contain backdoors. **Consequences:** Effective violation of [SA-1] in the context of such an Authenticator. **Mitigations:** The process of [SM-9] Authenticator Certification may reveal a subset of such threats, but it is not possible that all such can be revealed with black box testing and white box examination may be is economically infeasible. Users and Relying Parties with special concerns about this class of threat must exercise their own necessary caution about the trustworthiness and	SA-1

T-4.1.2	Malicious Authenticator HW	Violates
	verifiability of their vendors and supply chain.	

7.4.2 Threats to FIDO Server Vendors

T-4.2.1	Vendor Level Trust Anchor Injection Attack	Violates
	Attacker adds malicious trust anchors to the trust list shipped by a FIDO Server vendor. **Consequences:** Attacker can deploy fake Authenticators which Relying Parties cannot detect as such, which do not implement any appropriate security measures, and is able to violate all security goals of FIDO. **Mitigations:** This type of supply chain threat is outside the strict scope of the FIDO protocols and violates [SA-7]. Relying Parties can their trust list against definitive data published by the FIDO Alliance.	SA-7

7.4.3 Threats to FIDO Metadata Service Operators

T-4.3.1	Metadata Service Signing Key Compromise	Violates
	The attacker gets access to the private Metadata signing key. **Consequences:** The attacker could sign invalid Metadata. The attacker could • make trustworthy authenticators look less trustworthy (e.g. by increasing FAR). • make weak authenticators look strong (e.g. by changing the key protection method to a more secure one) • inject malicious attestation trust anchors, e.g. root certificates which cross-signed the original attestation trust anchor and the cross signed original attestation root certificate. This malicious trust anchors could be used to sign attestation certificates for fraudulent authenticators, e.g. authenticators using the AAID of trustworthy authenticators but not protecting their keys as stated in the metadata. **Mitigations:** The Metadata Service operator should protect the Metadata signing key appropriately, e.g. using a hardware protected key storage. Relying parties could use out-of-band methods to cross-check Metadata Statements with the respective vendors and cross-check the revocation state of the Metadata signing key with the provider of the Metadata Service.	SG-9

T-4.3.2	Metadata Service Data Injection	Violates
	The attacker injects malicious Authenticator data into the Metadata source. **Consequences:** The attacker could make the Metadata Service operator sign invalid Metadata. The attacker could • make trustworthy authenticators look less trustworthy (e.g. by increasing FAR). • make weak authenticators look strong (e.g. by changing the key protection method to a more secure one) • inject malicious attestation trust anchors, e.g. root certificates which cross-signed the original attestation trust anchor and the cross signed original attestation root certificate. This malicious trust anchors could be used to sign attestation certificates for fraudulent authenticators, e.g. authenticators using the AAID of trustworthy authenticators but not protecting their keys as stated in the metadata. **Mitigations:** The Metadata Service operator could carefully review the delta between the old and the new Metadata. Authenticator vendors could verify the published Metadata related to their Authenticators.	SG-9

7.5 Threats Specific to UAF with a second factor / U2F

T-5.1.1	Error Status Side Channel	Violates
	Relying parties issues an authentication challenge to an authenticator and can infer from error status if it is already enrolled. **Consequences:** U2F authenticators not requiring user interaction may be used to track users without their consent by issuing a pre-authentication challenge to a U2F token, revealing the identity of an otherwise anonymous user. Users would be identifiable by relying parties without their knowledge, violating [SG-7] **Mitigations:** The U2F specification recommends that browsers prompt users whether to allow this operation using mechanisms similar to those defined for other privacy sensitive operations like Geolocation.	SG-7

T-5.1.2	Malicious RP	Violates
	Malicious relying party mounts a cryptographic attack on a key handle it is storing. **Consequences:** U2F does not have a protocol-level notion of [SG-14] Transaction Non-Repudiation but If the Relying Party is able to recover the contents of the key handle it might forge logs of protocol exchanges to associate the user with actions he or she did not perform. If the Relying Party is able to recover the key used to wrap a key handle, that key is likely shared, and might be used to decrypt key handles stored with other Relying Parties and violate [SG-1] Strong User Authentication. **Mitigations:** None. U2F depends on [SA-1] to hold for key wrapping operations.	

T-5.1.3	Physical U2F Authenticator Attack	Violates
	Attacker gains physical access to U2F Authenticator (e.g., by stealing it). **Consequences:** Same as for T-1.4.4 A U2F authenticator has weak local user verification. If the attacker can guess the username and password/PIN, they can	

T-5.1.3	Physical U2F Authenticator Attack	Violates
	impersonate the user, violating [SG-1] Strong User Authentication **Mitigations:** Relying Parties can use strong additional factors. Relying Parties should provide users a means to revoke keys associated with a lost device.	SG-1

8. Acknowledgements

We thank iSECpartners for their review of, and contributions to, this document.

A. References

A.1 Informative references

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.pdf

[PasswordAuthSchemesKeyIssues]
Chwei-Shyong Tsai, Cheng-Chi Lee, and Min-Shiang Hwang, *Password Authentication Schemes: Current Status and Key Issues* International Journal of Network Security, Vol.3, No.2, PP.101–115, Sept. 2006, URL: http://ijns.femto.com.tw/contents/ijns-v3-n2/ijns-2006-v3-n2-p101-115.pdf

[QuestToReplacePasswords]
Joseph Bonneau, Cormac Herley, Paul C. van Oorschot and Frank Stajano, The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes, Microsoft Research, Carleton University and University of Cambridge, March 2012, URL: http://research.microsoft.com/pubs/161585/QuestToReplacePasswords.pdf

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

[U2FOverview]
S. Srinivas, D. Balfanz, E. Tiffany, *FIDO U2F Overview v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: http://fidoalliance.org/specs/fido-u2f-overview-v1.0-rd-20140209.pdf

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges, *FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.html
PDF: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-protocol-v1.1-id-20170202.pdf

IMPLEMENTATION DRAFT

FIDO Technical Glossary

FIDO Alliance Implementation Draft 02 February 2017

This version:
https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-glossary-v1.1-id-20170202.html
Previous version:
http://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-glossary-v1.0-ps-20141208.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Davit Baghdasaryan, Nok Nok Labs, Inc.
Brad Hill, PayPal
Jeff Hodges, PayPal

Abstract

This document defines all the strings and constants reserved by UAF protocols. The values defined in this document are referenced by various UAF specifications.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

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

Table of Contents

1. Notation

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

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

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

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. 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.

3. Definitions

AAID

Authenticator Attestation ID. See Attestation ID.

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.

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 RFC6454 origin [RFC6454].
- 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>*

AppID

The AppID is an identifier for a set of different Facets of a relying party's application. The AppID is a URL pointing to the TrustedFacets, i.e. list of FacetIDs related to this AppID.

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.

Authenticator Attestation ID / AAID

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

Authentication is the process in which user employs their FIDO Authenticator to prove possession of a registered key to a relying party.

Authentication Algorithm

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

Authentication Scheme

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

Authenticator, Authnr

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.

It is assumed that these authenticators have an internal matcher. The matcher is able to verify an already enrolled user. If there is more than one user enrolled – the matcher is also able to identify the right user.

Examples of such authenticator is a biometric sensor or a PIN based verification. Authenticators which only verify presence, such as a physical button, or perform no verification at all, cannot act as a first-factor authenticator.

Authenticator, 2ndF / Second Factor

A FIDO Authenticator which acts only as a second factor. Second-factor authenticators 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. It is assumed that these authenticators may or may not have an internal matcher.

Authenticator Attestation

The process of communicating a cryptographic assertion to a relying party that a key presented during authenticator 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.

AV

ASM Version

Bound Authenticator

A FIDO Authenticator or combination of authenticator and ASM, which uses an access control mechanism to restrict the use of registered keys to trusted FIDO Clients and/or trusted FIDO User Devices. Compare to a *Roaming Authenticator*.

Certificate

An X.509v3 certificate defined by the profile specified in [RFC5280] and its successors.

Channel Binding

See: [RFC5056], [RFC5929] and [ChannelID]. 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.

Client

This term is used “in context”, and may refer to a FIDO UAF Client or some other type of client, e.g. a TLS client. See FIDO Client.

Confused Deputy Problem

A confused deputy is a computer program that is innocently fooled by some other party into misusing its authority. It is a specific type of privilege escalation.

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. [AnonTerminology]

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)

Denotes the Encryption of data D with key K

ECDSA

Elliptic Curve Digital Signature Algorithm, as defined by ANSI X9.62 [ECDSA-ANSI].

Enrollment

The process of making a user known to an authenticator. This might be a biometric enrollment as defined in [NSTCBiometrics] 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.

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 related metadata.

A FIDO Authenticator is responsible for user verification, and maintaining the cryptographic material required for the 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 a roaming or bound authenticator, and whether it is a first-factor, or second-factor authenticator.

It is assumed by registration assertion schemes that the authenticator has exclusive control over the data being signed by the attestation key.

Some authentication assertion schemes (e.g. TAG_UAFV1_AUTH_ASSERTION) assume the authenticator to have exclusive control over the data being signed by the `Uauth key`.

FIDO Client

This is the software entity processing the UAF or U2F protocol messages on the FIDO User Device. FIDO Clients may take one of 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 reference data for the user and FIDO transaction history.

FIDO Server

Server software typically deployed in the relying party’s infrastructure that meets UAF protocol server requirements.

FIDO UAF Client

See FIDO Client.

FIDO User Device

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

Key Identifier (KeyID)

The KeyID is an opaque identifier for a key registered by an authenticator with a FIDO Server, for first-factor authenticators. It is used in concert with an AAID to identify a particular authenticator that holds the necessary key. Thus key identifiers must be unique within the scope of an AAID.

One possible implementation is that the 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*. Second-factor authenticators must retrieve their key handles from the relying party to function. First-factor authenticators manage the storage of their own key handles, either internally (for roaming authenticators) or via the associated ASM (for bound authenticators).

Key Registration

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

KeyRegistrationData (KRD)

A `KeyRegistrationData` object is created and returned by an authenticator as the result of the authenticator's `Register` command. The KRD object contains items such as the authenticator's AAID, the newly generated UAuth.pub key, as well as other authenticator-specific information such as algorithms used by the authenticator for performing cryptographic operations, and counter values. The KRD object is signed using the authenticator's attestation private key.

KHAccessToken

A secret value that acts as a guard for authenticator commands. KHAccessTokens are generated and provided by an ASM.

Matcher

A component of a FIDO Authenticator which is able to perform (local) user verification, e.g. biometric comparison [ISOBiometrics], PIN verification, etc.

Matcher Protections

The security mechanisms that an authenticator may use to protect the matcher component.

Persona

All relevant data stored in an authenticator (e.g. cryptographic keys) are related to a single "persona" (e.g. “business” or “personal” persona). Some administrative interface (not standardized by FIDO) provided by the authenticator may allow maintenance and switching of personas.

The user can switch to the “Personal” Persona and register new accounts. After switching back to the “Business” Persona, these accounts will not be recognized by the authenticator (until the User switches back to “Personal” Persona again).

This mechanism may be used to provide an additional measure of privacy to the user, where the user wishes to use the same authenticator in multiple contexts, without allowing correlation via the authenticator across those contexts.

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.

Reference Data

A (biometric) reference data (also called template) is a digital reference of distinct characteristics that have been extracted from a biometric sample. Biometric reference data is used during the biometric user verification process [ISOBiometrics]. Non-biometric reference data is used in conjunction with PIN-based user verification.

Registration

A FIDO protocol operation 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.

Relying Party

A web site or other entity that uses a FIDO protocol to directly authenticate users (i.e., performs peer-entity authentication). Note that if FIDO is composed with federated identity management protocols (e.g., SAML, OpenID Connect, etc.), the identity provider will also be playing the role of a FIDO Relying Party.

Roaming Authenticator

A FIDO Authenticator configured to move between different FIDO Clients and FIDO User Devices lacking an established trust relationship by:

1. Using only its own internal storage for registrations
2. Allowing registered keys to be employed without access control mechanisms at the API layer. (Roaming authenticators still may perform user verification.)

Compare to Bound Authenticator.

S(K, D)

Signing of data D with key K

Server Challenge

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

Sign Counter

A monotonically increasing counter maintained by the Authenticator. It is increased on every use of the UAuth.priv key. This value can be used by the FIDO Server to detect cloned authenticators.

SignedData

A `SignedData` object is created and returned by an authenticator as the result of the authenticator's `Sign` command. The to-be-signed data input to the signature operation is represented in the returned SignedData object as intact values or as hashed values. The SignedData object also contains general information about the authenticator and its mode, a nonce, information about authenticator-specific cryptographic algorithms, and a use counter. The `SignedData` object is signed using a relying party-specific UAuth.priv key.

Silent Authenticator

FIDO Authenticator that does not prompt the user or perform any user verification.

Step-up Authentication

An authentication which is performed on top of an already authenticated session.

Example: The user authenticates the session initially using a username and password, and the web site later requests a FIDO authentication on top of this authenticated session.

One reason for requesting step-up authenication could be a request for a high value resource.

FIDO U2F is always used as a step-up authentication. FIDO UAF could be used as step-up authentication, but it could also be used as an initial authentication mechanism.

Note: In general, there is no implication that the step-up authentication method itself is "stronger" than the initial authentication. Since the step-up authentication is performed on top of an existing authentication, the resulting combined authentication strength will increase most likely, but it will never decrease.

Template

See reference data.

TLS

Transport Layer Security

Token

In FIDO U2F, the term Token is often used to mean what is called an authenticator in UAF. Also, note that other uses of “token”, e.g. KHAccessToken, User Verification Token, etc., are separately distinct. If they are not explicitly defined, their meaning needs to be determined from context.

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 the information, and provide proof-of-possession of previously registered key material and an attestation of the confirmation back to the relying party.

Transaction Confirmation 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. It could be implemented using the GlobalPlatform specified TrustedUI [TEESecureDisplay].

TrustedFacets

The data structure holding a list of trusted FacetIDs. The AppID is used to retrieve this data structure.

TTEXT

Transaction Text, i.e. text to be confirmed in the case of transaction confirmation.

Type-length-value/tag-length-value (TLV)

A mechanism for encoding data such that the type, length and value of the data are given. Typically, the type and length data fields are of a fixed size. This format offers some advantages over other data encoding mechanisms, that make it suitable for some of the FIDO UAF protocols.

Universal Second Factor (U2F)

The FIDO protocol and family of authenticators which enable a cloud service to offer its users the options of using an easy–to–use, strongly–secure open standards–based second-factor device for authentication. The protocol relies on the server to know the (expected) user before triggering the authentication.

Universal Authentication Framework (UAF)

. The FIDO Protocol and family of authenticators which enable a service to offer its users flexible and interoperable authentication. This protocol allows triggering the authentication before the server knows the user.

UAF Client

See FIDO Client.

UAuth.pub / UAuth.priv / UAuth.key

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.key is the more generic notation to refer to UAuth.priv.

UINT8

An 8 bit (1 byte) unsigned integer.

UINT16

A 16 bit (2 bytes) unsigned integer.

UINT32

A 32 bit (4 bytes) unsigned integer.

UPV

UAF Protocol Version

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.

User Verification

The process by which a FIDO Authenticator locally authorizes use of key material, for example through a touch, pin code, fingerprint match or other biometric.

User Verification Token

The user verification token is generated by Authenticator and handed to the ASM after successful user verification. Without having this token, the ASM cannot invoke special commands such as `Register` or `Sign`.

The lifecycle of the user verification token is managed by the authenticator. The concrete techniques for generating such a token and managing its lifecycle are vendor-specific and non-normative.

Username

A human-readable string identifying a user's account at a relying party.

Verification Factor

The specific means by which local user verification is accomplished. e.g. fingerprint, voiceprint, or PIN.

This is also known as modality.

Web Application, Client-Side

The portion of a relying party application built on the "Open Web Platform" which executes in the context of 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 on the web server, 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.

A. References

A.1 Normative references

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

A.2 Informative references

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

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

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

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

[NSTCBiometrics]
NSTC Subcommittee on Biometrics, *Biometrics Glossary*. National Science and Technology Council. 14 September 2006, URL: http://biometrics.gov/Documents/Glossary.pdf

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

[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL: http://www.ietf.org/rfc/rfc5280.txt

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

[RFC6454]

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

[TEESecureDisplay]

GlobalPlatform Trusted User Interface API Specifications GlobalPlatform. Accessed March 2014. URL: https://www.globalplatform.org/specifications.asp