Table of Contents

Universal 2nd Factor (U2F) Overview

FIDO Alliance Proposed Standard 11 April 2017

This version:
: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.1-v1.2-ps-20170411.html

Editors:
: Sampath Srinivas, Google, Inc.
: Dirk Balfanz, Google, Inc.
: Eric Tiffany, FIDO Alliance
: Alexei Czeskis, Google, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The FIDO U2F protocol enables relying parties to offer a strong cryptographic 2nd factor option for end user security. The relying party's dependence on passwords is reduced. The password can even be simplified to a 4-digit PIN. End users carry a single U2F device which works with any relying party supporting the protocol. The user gets the convenience of a single 'keychain' device and convenient security. This document is an overview of the U2F protocol and is a recommended first-read before reading detailed protocol documents.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third

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

Table of Contents

1. What Is This Document?

This document provides an overview of the FIDO Universal 2nd Factor (U2F). It is intended to be read before the reader reads the detailed protocol documents listed below. It is intended to give the reader context for reading the detailed documents. This document is intended as an interpretive aid - it is not normative.

After reading this overview, it is recommended that the reader go through the detailed protocol documents listed below in the order they are listed. That order starts the reader at the top layer which is the U2F API and progresses down to lower layers such as the transport framing to the U2F device.

1. **FIDO U2F JavaScript API**

2. **FIDO U2F Raw Message Formats**

3. **FIDO U2F USB Framing of APDUs**

4. **FIDO U2F Application Isolation through Facet Identification**

5. **FIDO U2F Implementation Considerations**

6. **FIDO Security Reference**

   A glossary of terms used in the FIDO specifications is also available:

7. **FIDO Glossary**

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

2. Background

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 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.

This document that you are reading gives an overview of the U2F protocol.

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.

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

3. Goal: Strong Authentication and Privacy for the Web

The U2F eco-system is designed to provide strong authentication for users on the web while preserving the user's privacy. The user carries a 'U2F device' as a second factor.

When the user registers the U2F device at an account at a particular origin (such as http://www.company.com) the device creates a new key pair usable only at that origin and gives the origin the public key to associate with the account. When the user authenticates (i.e., logs in) to the origin, in addition to username and password, the origin (in this case, http://www.company.com) can check whether the user has the U2F device by verifying a signature created by the device.

The user is able to use the same device across multiple sites on the web - it thus serves as the user's physical web keychain with multiple (virtual) keys to various sites provisioned from one physical device. Using the open U2F standard, any origin will be able to use any browser (or OS) which has U2F support to talk to any U2F compliant device presented by the user to enable strong authentication.

The U2F device registration and authentication operations are exposed through JavaScript APIs built into the browser and, in following phases, native APIs in mobile OSes.

The U2F device can be embodied in various form factors, such as stand alone USB devices, stand alone Near Field Communication (NFC) device, stand alone Bluetooth LE devices, built-on-board the user's client machine/mobile device as pure software or utilizing secured crypto capabilities. It is strongly preferable to have hardware backed security, but it is not a requirement. However, as we shall see the protocol provides an attestation mechanism which allows the accepting online service or website to identify the class of device and either accept it or not depending on the particular site's policy.

The specs for U2F are in two layers. The upper layer specifies the cryptographic core of the protocol. The lower layer specifies how the user's client will communicate U2F cryptographic requests to the U2F device over a particular transport protocol (e.g., USB, NFC, Bluetooth LE, built-in on a particular OS, etc.).

The current spec set from the U2F group specifies the upper layer (which is unchanged regardless of transport) and the lower layer for the USB transport. Later phases of the protocol spec will specify transports for U2F over NFC, Bluetooth and when built-in (i.e., where the U2F capability is built into the device and accessed locally via the OS).

As one of the founders of the U2F working group in FIDO, Google is working to build U2F support into the Chrome browser and will offer U2F as a 2nd factor option on Google accounts to help the start-up of the open ecosystem.

A critical factor for success will be that a U2F device 'just works' with any modern client device owned by the user without needing additional driver or middleware setup. In this spirit, the USB U2F device is designed to work out of box with existing consumer operating systems with no driver installs or software changes. A U2F device-aware browser is able to discover and communicate with U2F devices using standard built-in OS APIs. To this end, in the first USB based deliverable, we are leveraging the built-in driverless libUSB device support in all modern OSes.

4. Site-Specific Public/Private Key Pairs

The U2F device and protocol need to guarantee user privacy and security. At the core of the protocol, the U2F device has a capability (ideally, embodied in a secure element) which mints an origin-specific public/private key pair. The U2F device gives the public key and a Key Handle to the origin online service or website during the user registration step.

Later, when the user performs an authentication, the origin online service or website sends the Key Handle back to the U2F device via the browser. The U2F device uses the Key Handle to identify the user's private key, and creates a signature which is sent back to the origin to verify the presence of the U2F device. Thus, the Key Handle is simply an identifier of a particular key on the U2F device.

The key pair created by the U2F device during registration is origin specific. During registration, the browser sends the U2F device a hash of the origin (combination of protocol, hostname and port). The U2F device returns a public key and a Key Handle. Very importantly, the U2F device encodes the requesting origin into the Key Handle.

Later, when the user attempts to authenticate, the server sends the user's Key Handle back to the browser. The browser sends this Key Handle and the hash of the origin which is requesting the authentication. The U2F device ensures that it had issued this Key Handle to that particular origin hash before performing any signing operation. If there is a mismatch no signature is returned.

This origin check ensures that the public keys and Key Handles issued by a U2F device to a particular online service or website cannot be exercised by a different online service or website (i.e., a site with a different name on a valid SSL certificate). This is a critical privacy property - assuming the browser is working as it should, a site can verify identity strongly with a user's U2F device only with a key which has been issued to that particular site by that particular U2F device. If this origin check was not present, a public key and Key Handle issued by a U2F device could be used as a 'supercookie' which allows multiple colluding sites to strongly verify and correlate a particular user's identity.

5. Alerting the User: U2F Device 'activation' & Browser Infobars

The U2F device device has a physical 'test of user presence'. The user touches a button (or sensor of some kind) to 'activate' the U2F device and this feeds into the device's operation as follows:

- Registration: The U2F device responds to a request to generate a key pair only if it has been 'activated'. Separately, the browser implementation might ensure that the javascript 'ask the U2F device to issue a key pair' call always results in the user seeing an infobar dialog which asks if he/she indeed wants to allow the current site to register the U2F device.

- Authentication: During authentication, the browser sends some data down to the U2F device that it needs to sign (more about this later). The U2F device can be requested to require a 'test of user presence' before it will sign - e.g., the user has to press a button on the device. This ensures that a signature happens only with the user's permission. It also ensures that that malware cannot exercise the signature when the user is not present.

When the user attempts to authenticate for the first time to a particular origin (i.e. the javascript call for 'Get me a signature from the U2F device' is exercised), the browser may put up an infobar which asks if the user would like to allow the site to talk to the U2F device. In this case, the browser should also present a 'Remember this' option with the infobar so that the browser can remember the permission and not ask every time. This setting can be reset (as with other browser settings).

In summary, the user will have to touch a button to register, and may also be warned by the browser. The relying party can put up screens which will walk the user through these steps. Registration is a very high value operation - it gives an origin a capability to very strongly verify a user and it needs to be taken very seriously. During authentication (or more generally, whenever the online service or website needs to strongly verify the user by requesting a signature), the user needs to activate the device to demonstrate user presence before the signature can happen. From version 1.2 onwards, it is possible to ask a device for a signature without user presence. The signature clearly indicates it was made without user presence and will have to treated by the online service or website accordingly.

6. Man-In-The-Middle Protections During Authentication

If a man-in-the-middle (MITM) tries to intermediate between the user and the origin during the authentication process, the U2F device protocol can detect it in most situations.

Say a user has correctly registered a U2F device with an origin and later, a MITM on a different origin tries to intermediate the authentication. In this case, the user's U2F device won't even respond, since the MITM's (different) origin name will not match the Key Handle that the MITM is relaying from the actual origin. U2F can also be leveraged to detect more sophisticated MITM situations as we shall see below.

As one of the return values of the U2F 'sign' call, the browser returns an object which contains information about what the browser sees about the origin (we will call this the 'client data' object). This 'client data' includes:

a. the random challenge sent by the origin,
b. the origin host name seen by the browser for the web page making the javascript call, and
c. [optionally] if the ChannelID extension to TLS is used, the connection's channelID public key.

The browser sends a hash of this 'client data' to the U2F device. In addition to the hash of the 'client data', as discussed earlier, the browser sends the hash of the origin and the Key Handle as additional inputs to the U2F device.

When the U2F device receives the client data hash, the origin hash and the Key Handle it proceeds as follows: If it had indeed issued that Key Handle for that origin the U2F device proceeds to issue a signature across the hashed 'client data' which were sent to it. This signature is returned back as another return value of the U2F 'sign' call.

The site's web page which made the U2F 'sign' call sends the return values, both the 'client data' and the signature, back to the origin site (or equivalently, relying party). On receiving the 'client data' and the signature, the relying party's first step, of course, is to verify that the signature matches the data as verified by the user's origin-specific public key. Assuming this matches, the relying party can examine the 'client data' further to see if any MITM is present as follows:

- If 'client data' shows that an incorrect origin name was seen by the user

    - an MITM is present
    - (albeit a sophisticated MITM which had also intermediated the registration and thus got the Key Handle issued by the U2F device to match the MITM's own origin name, and the MITM is now trying to intermediate an authentication. As noted earlier, an MITM intermediating only at authentication time and not at registration would fail since the U2F device would refuse to sign due to origin mismatch with the Key Handle relayed from the original origin by the MITM).
- else if 'client data' shows a ChannelID OR origin used a ChannelID for the SSL connection:
    - If ChannelID in 'client data' does not match the ChannelID the origin used, an MITM is present
    - (albeit a very sophisticated MITM which possesses an actual valid SSL cert for the origin and is thus indistinguishable from an 'origin name' perspective)

It is still possible to MITM a user's authentication to a site if the MITM is

a. able to get a server cert for the actual origin name issued by a valid CA, and
b. ChannelIDs are NOT supported by the browser.

But this is quite a high bar.

An MITM case which the U2F device does NOT protect against is as follows: Consider an online service or website which accepts plain password but allows users to self-register and step up to U2F 2nd factor. An MITM with a different origin which is present between the user and the actual site from the time of registration can register the U2F device on to itself and not pass this registration to the actual origin, which would still see the user as just needing a password. Later, for authentications, the MITM can accept the U2F device and just do an authentication with password to the actual origin.

Assuming the user does not notice the wrong (different) origin in the URL, the user would think they are logging in to the actual origin with strong authentication and are thus very secure but in reality, they are actually being MITMed.

7. Allowing for Inexpensive U2F Devices

A key goal of this program is to enable extremely inexpensive yet secure devices. To enable new secure element chips to be as inexpensive as possible it is important to allow them to have minimal or no onboard memory.

A U2F device allows for this. The Key Handle issued by the U2F device does not have to be an index to the private key stored on board the U2F device secure element chip. Instead, the Key Handle can 'store' (i.e., contain) the private key for the origin and the hash of the origin encrypted with a 'wrapping' key known only to the U2F device secure element. When the Key Handle goes back to the secure element it 'unwraps' it to 'retrieve' the private key and the origin that it was generated for.

As another alternative, the U2F device could store this 'wrapped' information in a table in off-chip memory outside the secure element (which is presumably cheaper). This memory is still on board the U2F device. In this case, the Key Handle sent to the origin would be an index into this table in off-chip memory. As another possibility in the design spectrum, the Key Handle might only encode the origin and an index number, while the private key might still be kept on board - this would, of course, imply the number of keys is limited by the amount of memory.

8. Verifying That a U2F Device Is 'genuine'

The U2F device protocol is open. However, for effective security, a U2F device has to be built to certain standards - for example, if the Key Handle contains private keys encrypted with some manufacturer specific method, this has to be certified as well implemented, ideally by some 'certification body' such as FIDO. In addition, the actual cryptographic engine (secure element) should ideally have some strong security properties.

With these considerations in mind, a relying party needs to able to identify the type of device it is speaking to in a strong way so that it can check against a database to see if that device type has the certification characteristics that particular relying party cares about. So, for example, a financial services site may choose to only accept hardware-backed U2F devices, while some other site may allow U2F devices implemented in software.

Every U2F device has a shared 'Attestation' key pair which is present on it - this key is shared across a large number of U2F device units made by the same vendor (this is to prevent individual identifiability of the U2F device). Every public key output by the U2F device during the registration step is signed with the attestation private key.

The intention is that the public keys of all the 'Attestation' key pairs used by each vendor will be available in the public domain - this could be implemented by certificates chaining to a root public key or literally as a list. We will work within FIDO to decide the details on how certified vendors can publish their attestation public keys.

When such an infrastructure is available, a particular relying party - say, a bank - might choose to accept only U2F devices from certain vendors which have the appropriate published certifications. To enforce this policy, it can verify that the public key from a U2F device presented by the user is from a vendor it trusts.

In practice, for high quality U2F devices we expect that the attestation key would be burnt into the on-board secure element - the actual key to be burnt in would be provided by the vendor to the secure element manufacturer for every batch of chips, say about 100,000 units.

Note that the attestation key's presence only guarantees who the vendor is for a well built U2F device - it is one part of the story, albeit a very crucial part. As to whether the U2F device is indeed secure, that guarantee comes from certifications where third parties inspect the implementation by the vendor. In summary, attestation is a strong identifier of the certifications.

In this context, it's worth noting that a U2F device which stores keys on board rather than exporting them in the Key Handle are, in principle, most secure, since it is not vulnerable to any potential vendor specific vulnerabilities in the design of the encryption of the data in the Key Handle. However, a good design with an encrypted Key Handle will be well above the bar in security while also being cheaper.

At this time, the encryption used to embed private keys in the Key Handle are technically not part of the specified protocol. However, strong best practice guidelines are specified in the sample client side javacard applet available in U2F working group materials. It may be appropriate to include a review of particular implementations as part of a U2F certification within FIDO.

Note that it is still possible for a vendor to build a U2F compliant device which is not certified and whose attestation keys are not published in a 'certification database'. A relying party could still choose to accept such devices - but it will do so with the full knowledge that that particular device type is not in the certification database.

8.1 Counters as a Signal for Detecting Cloned U2F Devices

The vendor attestation is one method by which an origin can assess a U2F device. In practice, we do not want to prevent other protocol compliant vendors, perhaps even those without any formal secure element, perhaps even completely software implementations. The problem with these non-secure-element based devices, of course, is that they could potentially be compromised and cloned.

The U2F device protocol incorporates a usage counter to allow the origin to detect problems in some circumstances. The U2F device remembers a count of the number of signature operations it has performed - either per key pair (if it has sufficient memory) or globally (if it has a memory constraint, this leaks some privacy across keys) or even something in between (e.g., buckets of keys sharing a counter, with a bit less privacy leakage). The U2F device sends the actual counter value back to the browser which relays it to the origin after every signing operation. The U2F device also concatenates the counter value on to the hash of the client data before signing so that the origin can strongly verify that the counter value was not tampered with (by the browser).

The server can compare the counter value that the U2F device sent it and compare it against the counter value it saw in earlier interactions with the same U2F device. If the counter value has moved backward, it signals that there is more than one U2F device with the same key pair for the origin (i.e., a clone of the U2F device has been created at some point).

The counter is a strong signal of cloning but cannot detect cloning in every case - for example, if the clone is only one which is used after the cloning operation and the original is never used, this case cannot be detected.

9. Client Malware Interactions with U2F Devices

As long as U2F devices can be accessed directly from user space on the client OS, it is possible for malware to create a keypair using a fake origin and exercise the U2F device. The U2F device will not be able to distinguish 'good' client software from 'bad' client software. On a similar note, it is possible for malware to relay requests from Client machine #1 to a U2F device attached to client machine #2 if the malware is running on both machines. This is conceptually no different from a shared communication channel between the Client machine (in this case #1) and the U2F device (which happens to be on machine #2). It is not in scope to protect against this situation.

Protection against malware becomes more possible if the U2F client is built into the OS system layer as opposed to running in user space. The OS can obtain exclusive access to U2F devices and enforce methods to ensure origin matches.

10. U2F Device User Experience

As described earlier access to the U2F device is manifested in two javascript functions available in the browser - one for creating a key pair and one for generating a signature. These are used by an origin online service or website to create a user flow.

10.1 Registration: Creating a Key Pair

The to-be-registered user is verified by the origin site (with username and password or whatever other means). The registration page rendered by the origin in the browser calls the javascript function for creating a key pair. When the javascript function is called, the user may see a browser infobar warning which he/she has to approve. After user approval, the key pair generation request is sent to every U2F device attached to the computer.

The first U2F device attached to the computer which has a positive 'test of user presence' (i.e., the first attached U2F device on which the user presses the button) responds to this request. The browser packages the response from the U2F device (key handle, public key, etc.) and returns it to the web page as return results of the javascript function call. The registration web page sends these to the origin site and the origin sites stores this information indexed by the user's account to complete the registration process.

10.2 Authentication: Generating a Signature

The user starts the authentication process typically with username and password (or with just the username, if the site only wants a U2F device verification). The origin site renders an intermediate authentication page into which it sends the user's Key Handle and a nonce. It then calls the javascript function to create a signature. The parameters for the function call are the Key Handle and the nonce.

When the signature function is called, the browser may show an infobar asking for the user's approval (the user may choose to ask the browser to skip this in future). After the user's approval, the browser talks to all the U2F devices attached to the computer as described earlier and assembles their responses.

The javascript function call returns the 'client data' object and the first signature response from a U2F device that replied. The intermediate authentication web page sends the 'client data' and the U2F device responses on to the relying party, which determines if any of the signatures matches what it expects.

Note that depending on the U2F implementation multiple devices could reply for a particular Key Handle. For example, consider the case where the Key Handle is implemented purely as an index into memory on board the U2F device (and thus was just, say, a small integer). The user may have registered multiple U2F devices to a particular account on a particular origin and some of those devices could have used the same index integer as Key Handle for that particular account on that particular origin.

Note that though the user does not necessarily have to see the intermediate page described above. If the correct U2F device is present, then the signatures can be obtained and sent back to the origin and the authentication is completed. The user needs to see intermediate screens only for error conditions ('Please insert your U2F device', 'We require you to activate your U2F device', etc.).

11. U2F Device Usage Scenarios

Though the description so far has been in context of a particular user using a single device across multiple accounts, the usage scenarios enabled are broader.

11.1 Sharing a U2F Device Among Multiple Users

Note that a U2F device has no concept of a user - it only knows about issuing keys to origins. So a person and their spouse could share a U2F device and use it for their individual accounts on the same origin. Indeed, as far as the U2F device is concerned the case of two users having accounts on the same origin is indistinguishable from the case of the same user having two accounts on that origin.

Needless to say, the general case where multiple persons share a single U2F device and each person has accounts on whatever origins they choose is similarly supported in U2F.

11.2 Registering Multiple U2F Devices to the Same Account

U2F does not limit the user to have a single device registered on a particular account on a particular site. So for example, a user might have a U2F device mounted permanently on two different computers, where each U2F device is registered to the same account on a particular origin - thus allowing both computers to login securely to that particular origin.

If a user has registered multiple U2F devices to a particular account, then during authentication all the Key Handles are sent by the origin to the intermediate page. The intermediate page call the signature javascript function with the array of Key Handles and sends the aggregated response back to the origin. Each attached activated U2F device signs for those Key Handles in the array that it recognizes. The user authentication experience is unchanged.

As an optimization, note that when a origin detects a particular Key Handle is used successfully to authenticate from a particular browser, it can remember that Key Handle for future reference by setting a cookie on that browser and trying that Key Handle first before attempting other Key Handles.

12. U2F Privacy Considerations: A Recap

As the reader would have noticed, user privacy is a fundamental design consideration for the U2F protocol. The various privacy related design points are reiterated here:

1. A U2F device does not have a global identifier visible across online services or websites.
2. A U2F device does not have a global identifier within a particular online service or website
   - Example 1: If a person loses their U2F device, the finder cannot 'point it at a website' to see if some accounts get listed. The device simply does not know.
   - Example 2: If person A and B share a U2F device and they have each registered their accounts on site X with this device, there isn't any way for the site X to guess that the two accounts share a device based on the U2F protocol alone.
3. A key issued to a particular online service or website can only be exercised by that online service or website.
   - Since a key is essentially a strong identifier this means U2F does not give any signal which allows online services or websites to strongly cross-identify shared users.
4. A user has to activate the U2F device (i.e., 'press the button') before it will issue a key pair (for registration) or sign a challenge.
5. The browser may notify the user before they form a U2F relationship with an online service or website
   - An infobar could appear whenever the 'issue a key' javascript call is made.
   - An infobar (with a once-only option) could appear when the 'sign with this key' javascript call is made for a particular origin

The infobar approach puts a decision burden on the users - this is a downside and the infobar UX design has to be done with care.

13. Other Privacy Related Issues

13.1 An Origin Can Discover that Two Accounts Share a U2F Device

The origin specific key issuance still leaves one possible privacy leak - which is the case where a person with a single U2F device uses it to generate keys to two separate accounts with the same origin. Say the two different accounts are associated with usernames u_1 and u_2 in the site's name space. Now when u_1 is attempting to authenticate, the origin can send down KeyHandle_2 to the U2F device. If it returns a valid signature, it can infer that u_1 and u_2 belong to the same person or two persons who share the same computer who happen to have their U2F devices plugged in simultaneously. This is true even if the users have taken precautions to hide their client identity from the origin server (using an anonymizing proxy, incognito mode, etc.).

It is possible to enhance the U2F device specification to catch this case but it complicates the user experience and we chose not to do so. Users who are concerned about this line of attack would need to use different U2F devices for different accounts on the same site and plug in only the relevant U2F device and no other when initiating a session for a particular account.

13.2 Revoking a Key From an Origin

Say a user registers their U2F device on an online service or website which has unsavory practices without the user realizing that the online service or website is unsavory. Later the user wants to cut off association with that site. It should ideally be possible for the user to 'delink' the key such that the U2F device starts behaving as if it no longer owns the key. Thus the site cannot strongly verify the user even if it can do social engineering to make the user click past warnings.

It is possible for a vendor to design a U2F device which can be 'reset' - in that it stops honoring any key it has issued before the reset. This might mean the earlier Key Handles need to have a generation count and a reset makes the U2F device reject all keys older than the current generation count. Alternatively, if the U2F device uses a key wrapping mechanism, a 'reset' could throw away the old wrapping key and replace it. This renders all earlier keys issued by the device useless, since the device can no longer make any sense of them.

However, if the secure element is stateless and has no hard reset ability, all this 'revocation' logic has to be implemented as blacklists in firmware outside the secure element (for e.g., code on the USB intermediator). In such a case it is possible for a dedicated attacker (e.g., a spy service) to extract the secure element and verify if it indeed does work against keys it has issued in the past. One revocation safeguard available to the user is physical destruction of the U2F device - this could be useful in sensitive high value situations (e.g., a political dissident).

14. Non-USB Transports

As discussed earlier, other transports are available. The specifications currently support USB, Bluetooth LE and NFC.

15. Expanding U2F to Non-browser Apps

The discussion above has been focused on the browser as the client side vehicle, with a JavaScript API to talk to U2F devices. However, it is perfectly sensible to have app on a mobile OS such as Android talk to U2F devices over a system **API.**

When building a native system API, we still need a notion of 'origin'. For example, if foo.com's app mints a key on a particular U2F device, then bar.com's app should not be able to exercise that key. Even more importantly, if the user uses the foo.com web app on a computer and foo.com's app on a mobile device, the user needs to be able to use the same U2F device with both. This means that there has to be mechanism where the origin sent down to the U2F device by the browser for the foo.com web page matches the origin sent down to the U2F device by the mobile OS for the foo.com app.

This is achieved by specifying a level of indirection using the notion of an 'application id', which is a generalization of the origin concept. The 'application id' is a publicly fetchable https URL where a particular origin (such as foo.com) lists its various 'facets' - for example, it may list the hostname 'www.foo.com' and the identifier for the signatures of foo.com's android app. The application id https URL is assumed to be under the control of the origin - in other words, only it can change the list of 'facets'.

The origin website or online service sends its 'application id' down as a parameter to the U2F API on the web page. The browser fetches the content of the 'application id' URL and ensures that the actual origin it sees for the web page calling the U2F API is indeed listed in the 'facets' in the 'application id' URL. For example, if a page served off www.foo.com makes a U2F API call, then this host name needs to be listed as a facet in the 'application id which is passed down. Similarly when a particular mobile app passes a 'application id' to a U2F API on a mobile OS, the OS checks if the code signing signature of that particular app is listed as a facet in the 'application id'. After these check if the 'facet' is indeed in the 'application id' as expected, the hash of the 'application id' is sent down to the U2F device, rather than the hash of the 'origin'. This ensures that foo.com's web page and foo.com's mobile app both are seen as the same site by the U2F device. As mentioned earlier, the 'application id' is a generalized notion of an origin.

FIDO U2F Raw Message Formats

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.1-v1.2-ps-20170411.html
Previous version:
https://fidoalliance.org/specs/fido-u2f-raw-message-formats-v1.1-RD-20140209.html
Editors:
Dirk Balfanz, Google, Inc.
Jakob Ehrensvard, Yubico, Inc
Juan Lang, Google, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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

Table of Contents

1. Notation

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

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

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

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

U2F specific terminology used in this document is defined in [FIDOGlossary].

Symbolic constants such as **U2F_REGISTER** which are referred to when defining messages in this documents have their values defined in (See [U2FHeader] in bibliography].

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

Note: Reading the 'FIDO U2F Overview' (see [U2FOverview] in bibliography) is recommended as a background for this document.

U2F Tokens provide cryptographic assertions that can be verified by relying parties. Typically, the relying party is a web server, and the cryptographic assertions are used as second-factors (in addition to passwords) during user authentication.

U2F Tokens are typically small special-purpose devices that aren't directly connected to the Internet (and hence, able to talk directly to the relying party). Therefore, they rely on a FIDO Client to relay messages between the token and the relying party. Typically, the FIDO Client is a web browser.

The U2F protocol supports two operations, registration and authentication. The registration operation introduces the relying party to a freshly-minted keypair that is under control of the U2F token. The authentication operation proves possession of a previously-registered keypair to the relying party. Both the registration and authentication operation consist of three phases:

1. Setup: In this phase, the FIDO Client contacts the relying party and obtains a challenge. Using the challenge (and possibly other data obtained from the relying party and/or prepared by the FIDO Client itself), the FIDO Client prepares a request message for the U2F Token.

2. Processing: In this phase, the FIDO Client sends the request message to the token, and the token performs some cryptographic operations on the message, creating a response message. This response message is sent to the FIDO Client.

3. Verification: In this phase, the FIDO Client transmits the token's response message, along with other data necessary for the relying party to verify the token response, to the relying party. The relying party then processes the token response and verifies its correctness. A correct registration response will cause the relying party to register a new public key for a user, while a correct authentication response will cause the relying party to accept that the client is in possession of the corresponding private key.

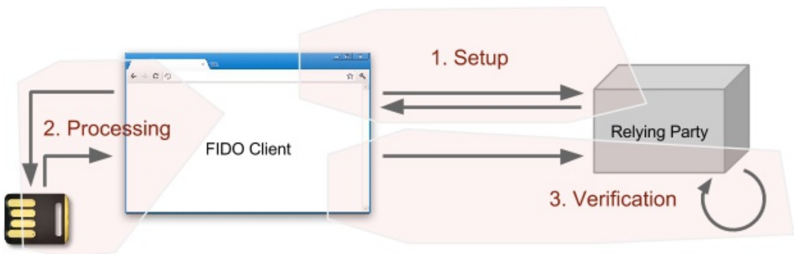


Fig. 1 Three phases of Registration and Authentication

Above is a picture illustrating the three phases.

At the heart of the U2F protocol are the request messages sent to the U2F token, and the response messages received from the U2F token.1

Note that the request message is usually obtained by the FIDO client from the relying party during the setup phase, and therefore reaches the FIDO client as part of an HTTP response. Similarly, the response message that is processed by the relying party during the verification phase is sent by the FIDO Client to the relying party in an HTTP request. Beware the possibility of confusion when talking about requests and responses!

Request messages are created by the relying party and consumed by the U2F token. Response messages are created by the U2F token and consumed by the relying party.

As the messages flow from relying party (through the FIDO Client) to the U2F token and back, they undergo various transformations and encodings. Some of these transformations and encodings are up to the individual implementations and are not standardized as part of FIDO U2F. For example, FIDO U2F does not prescribe how request and response messages are encoded between the FIDO Client and the relying party.

However, to ensure that U2F tokens from different vendors can work across U2F-compliant web sites certain encodings are standardized:

1. FIDO U2F standardizes a Javascript API that prescribes how a web application can pass request messages into the FIDO Client (in the case where the web browser is the FIDO Client), and what the encoding of the response messages is.

2. FIDO U2F standardizes how request and response messages are to be encoded when sent over from the client over the USB, NFC, Bluetooth and Bluetooth Low-Energy transports to U2F tokens. In addition to specifying the encoding, the transport level specification also specifies the

format for control messages to the tokens and the format for the error responses from the tokens.

In this document we describe the *"raw"*, or canonical, format of the messages, i.e., without regard to the various encodings that are prescribed in U2F transport standards or that implementors might choose when sending messages around. The raw format of the messages is important to know for two reasons:

1. The encoding of messages and parameters described elsewhere may refer to the raw messages described in this document. For example, a Javascript API might refer to a parameter of a function as the Base64-encoding of a raw registration response message. It is this document that describes what the raw registration response message looks like.

2. Cryptographic signatures are calculated over raw data. For example, the standard might prescribe that a certain cryptographic signature is taken over bytes 5 through 60 of a certain raw message. The implementor therefore has to know how what the raw message looks like.

3. U2F message framing

The U2F protocol is based on a request-response mechanism, where a requester sends a request message to a U2F device, which always results in a response message being sent back from the U2F device to the requester.

The request message has to be "framed" to send to the lower layer. Taking the signature request as an example, the "framing" is a way for the FIDO client to tell the lower transport layer that it is sending a signature request and then send the raw message contents. The framing also specifies how the transport will carry back the response raw message and any meta-information such as an error code if the command failed.

Framing is defined based on the ISO7816-4:2005 APDU format.

3.1 Request Message Framing

The raw request message is framed as a command APDU. At a high level, APDUs are framed in the following way:

CLA INS P1 P2 [L_c <request-data>] [L_e]

Where:

- **CLA**: Reserved to be used by the underlying transport protocol (if applicable). The host application shall set this byte to zero.

- **INS**: U2F command code, defined in the following sections.

- **P1, P2**: Parameter 1 and 2, defined by each command.

- **L_c**: The length of the **request-data**. If there are no request data bytes, **L_c** is omitted.

- **L_e**: The maximum expected length of the response data. If no response data are expected, **L_e** may be omitted.

The precise format of the APDU depends on the encoding choice. There are two different encodings allowed for an APDU: **short** and **extended length**. The differences are in the way the length of the request data, **L_c**, and the maximum length of the expected response, **L_e**, are encoded.

The choice of encoding varies depending on the needs of the individual transport. Refer to the transport-specific encoding documents for which encodings are allowed with each transport.

3.1.1 Command and parameter values

Command	INS	P1	P2
U2F_REGISTER	0x01	0x00	0x00
U2F_AUTHENTICATE	0x02	0x03|0x07|0x08	0x00
U2F_VERSION	0x03	0x00	0x00
VENDOR SPECIFIC	0x40-0xbf	NA	NA

3.1.2 Short Encoding

In **short encoding**, the maximum length of request-data is 255 bytes. **L_c** is encoded in the following way:

Let **N_c** = | <**request-data**> |. If **N_c** is 0, **L_c** is omitted. Otherwise, **L_c** is encoded as a single byte containing the value of **N_c**.

If the instruction is not expected to yield any response bytes, **L_e** may be omitted. Otherwise, in **short encoding**, **L_e** is encoded in the following way:

Let **N_e** = the maximum length of the response data. In **short encoding**, the maximum value of **N_e** is 256 bytes.

For values of **N_e** between 1 and 255, **L_e** contains the value of **N_e**. When **N_e** = 256, **L_e** contains the value 0.

3.1.3 Extended Length Encoding

In **extended length encoding**, the maximum length of request-data is 65 535 bytes. **L_c** is encoded in the following way:

Let **N_c** = | <**request-data**> |. If **N_c** is 0, **L_c** is omitted. Otherwise, **L_c** is encoded as:

0 MSB(N_c) LSB(N_c)

Where **MSB(N_c)** is the most significant byte of **N_c**, and **LSB(N_c)** is the least significant byte of **N_c**.

In other words, the request-data are preceded by three length bytes, a byte with value 0 followed by the length of request-data, in big-endian order.

If the instruction is not expected to yield any response bytes, **L_e** may be omitted. Otherwise, in **extended length encoding**, **L_e** is encoded in the following way:

Let $\mathbf{N_e}$ = the maximum length of the response data. In **extended length encoding**, the maximum value of $\mathbf{N_e}$ is 65 536 bytes.

For values of $\mathbf{N_e}$ between 1 and 65 535, inclusive, let $\mathbf{L_{e1}} = \mathbf{MSB(N_e)}$, and $\mathbf{L_{e2}} = \mathbf{LSB(N_e)}$, where $\mathbf{MSB(N_e)}$ is the most significant byte of $\mathbf{N_e}$, and $\mathbf{LSB(N_e)}$ is the least significant byte of $\mathbf{N_e}$.

When $\mathbf{N_e}$ = 65 536, let $\mathbf{L_{e1}} = \mathbf{0}$ and $\mathbf{L_{e2}} = \mathbf{0}$.

When $\mathbf{L_c}$ is present, i.e. if $\mathbf{N_c} > 0$, $\mathbf{L_e}$ is encoded as:

$\mathbf{L_{e1}\ L_{e2}}$

When $\mathbf{L_c}$ is absent, i.e. if $\mathbf{N_c} = 0$, $\mathbf{L_e}$ is encoded as:

$\mathbf{0\ L_{e1}\ L_{e2}}$

In other words, $\mathbf{L_e}$ has a single-byte prefix of 0 when $\mathbf{L_c}$ is absent.

3.2 Response Message Framing

The raw response data is framed as a response APDU:

<response-data> $\mathbf{SW_1\ SW_2}$

Where $\mathbf{SW_1}$ and $\mathbf{SW_2}$ are the status word bytes 1 and 2, respectively, forming a 16-bit status word, defined below. SW_1 is the most-significant byte, and SW_2 is the least-significant byte.

3.3 Status Codes

The following ISO7816-4 defined status words have a special meaning in U2F:

- **SW_NO_ERROR (0x9000)**: The command completed successfully without error.
- **SW_CONDITIONS_NOT_SATISFIED (0x6985)**: The request was rejected due to test-of-user-presence being required.
- **SW_WRONG_DATA (0x6A80)**: The request was rejected due to an invalid key handle.
- **SW_WRONG_LENGTH (0x6700)**: The length of the request was invalid.
- **SW_CLA_NOT_SUPPORTED (0x6E00)**: The Class byte of the request is not supported.
- **SW_INS_NOT_SUPPORTED (0x6D00)**: The Instruction of the request is not supported.

Each implementation may define any other vendor-specific status codes, providing additional information about an error condition. Only the error codes listed above will be handled by U2F FIDO Client, where others will be seen as general errors and logging of these is optional.

4. Registration Messages

4.1 Registration Request Message - U2F_REGISTER

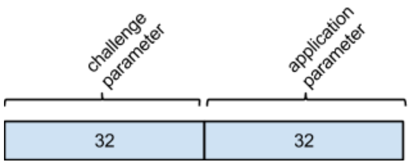


Fig. 2 Registration Request Message

This message is used to initiate a U2F token registration. The FIDO Client first contacts the relying party to obtain a challenge, and then constructs the registration request message. The registration request message has two parts:

- The **challenge parameter** [32 bytes]. The challenge parameter is the SHA-256 hash of the Client Data, a stringified JSON data structure that the FIDO Client prepares. Among other things, the Client Data contains the challenge from the relying party (hence the name of the parameter). See below for a detailed explanation of Client Data.
- The **application parameter** [32 bytes]. The application parameter is the SHA-256 hash of the UTF-8 encoding of the application identity of the application requesting the registration. (See [FIDOAppIDAndFacets] in bibliography for details.)

4.2 Registration Response Message: Error: Test-of-User-Presence Required

This is an error message that is output by the U2F token if no test-of-user-presence could be obtained by the U2F token. The error message details are specified in the framing for the underlying transport (see Section "U2F Message Framing" above).

4.3 Registration Response Message: Success

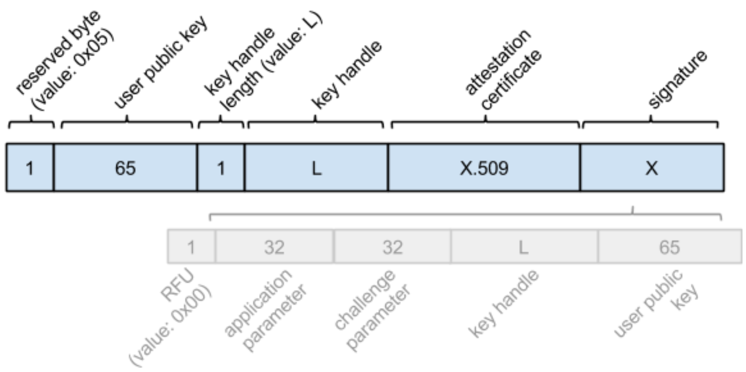


Fig. 3 Registration Response Message

This message is output by the U2F token once it created a new keypair in response to the registration request message. Note that U2F tokens should verify user presence before returning a registration response success message (otherwise they should return a test-of-user-presence-required message - see above). Its raw representation is the concatenation of the following:

- A **reserved byte** [1 byte], which for legacy reasons has the value 0x05.
- A **user public key** [65 bytes]. This is the (uncompressed) x,y-representation of a curve point on the P-256 NIST elliptic curve.
- A **key handle length byte** [1 byte], which specifies the length of the key handle (see below). The value is unsigned (range 0-255).
- A **key handle** [length specified in previous field]. This a handle that allows the U2F token to identify the generated key pair. U2F tokens may wrap the generated private key and the application id it was generated for, and output that as the key handle.
- An **attestation certificate** [variable length]. This is a certificate in X.509 DER format. Parsing of the X.509 certificate unambiguously establishes its ending. The remaining bytes in the message are
- a **signature** [variable length, 71-73 bytes]. This is a ECDSA signature (on P-256) over the following byte string:
    - A *byte reserved for future use* [1 byte] with the value 0x00.
    - The *application parameter* [32 bytes] from the registration request message.
    - The *challenge parameter* [32 bytes] from the registration request message.
    - The above *key handle* [variable length]. (Note that the key handle length is not included in the signature base string.

      This doesn't cause confusion in the signature base string, since all other parameters in the signature base string are fixed-length.)
    - The above *user public key* [65 bytes].

  The signature is encoded in ANSI X9.62 format (see [ECDSA-ANSI] in bibliography).

The signature is to be verified by the relying party using the public key certified in the attestation certificate. The relying party should also verify that the attestation certificate was issued by a trusted certification authority. The exact process of setting up trusted certification authorities is to be defined by the FIDO Alliance and is outside the scope of this document.

Once the relying party verifies the signature, it should store the public key and key handle so that they can be used in future authentication operations.

5. Authentication Messages

5.1 Authentication Request Message - U2F_AUTHENTICATE

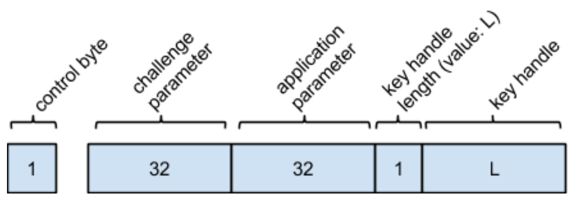


Fig. 4 Authentication Request Message

This message is used to initiate a U2F token authentication. The FIDO Client first contacts the relying party to obtain a challenge, and then constructs the authentication request message. The authentication request message has five parts:

- Control byte (P1). The control byte is determined by the FIDO Client - the relying party cannot specify its value. The FIDO Client will set the control byte to one of the following values:

    - **0x07 ("check-only")**: if the control byte is set to 0x07 by the FIDO Client, the U2F token is supposed to simply check whether the provided key handle was originally created by this token, and whether it was created for the provided application parameter. If so, the U2F token must respond with an authentication response message:error:test-of-user-presence-required (note that despite the name this signals a success condition). If the key handle was not created by this U2F token, or if it was created for a different application parameter, the token must respond with an authentication response message:error:bad-key-handle.

    - **0x03 ("enforce-user-presence-and-sign")**: If the FIDO client sets the control byte to 0x03, then the U2F token is supposed to perform a real signature and respond with either an authentication response message:success or an appropriate error response (see below). The signature should only be provided if user presence could be validated.

    - **0x08 ("dont-enforce-user-presence-and-sign")**: If the FIDO client sets the control byte to 0x08, then the U2F token is supposed to perform a real signature and respond with either an authentication response message:success or an appropriate error response (see below). The signature may be provided without validating user presence.

  Other control byte values are reserved for future use.

  During registration, the FIDO Client may send authentication request messages to the U2F token to figure out whether the U2F token has already been registered. In this case, the FIDO client will use the check-only value for the control byte. In all other cases (i.e., during authentication), the FIDO Client must use the enforce-user-presence-and-sign or don't-enforce-user-presence-and-sign values.

- The **challenge parameter** [32 bytes]. The challenge parameter is the SHA-256 hash of the Client Data, a stringified JSON data structure that the FIDO Client prepares. Among other things, the Client Data contains the challenge from the relying party (hence the name of the parameter). See below for a detailed explanation of Client Data.

- The **application parameter** [32 bytes]. The application parameter is the SHA-256 hash of the UTF-8 encoding of the application identity of the application requesting the authentication as provided by the relying party.

- A **key handle length byte** [1 byte], which specifies the length of the key handle (see below). The value is unsigned (range 0-255).

- A **key handle** [length specified in previous field]. The key handle. This is provided by the relying party, and was obtained by the relying party during registration.

5.2 Authentication Response Message: Error: Test-of-User-Presence Required

This is an error message that is output by the U2F token if no test-of-user-presence could be obtained by the U2F token. The error message details are specified in the framing for the underlying transport (see Section "U2F Message Framing" above).

5.3 Authentication Response Message: Error: Bad Key Handle

This is an error message that is output by the U2F token if the provided key handle was not originally created by this token, or if the provided key handle was created by this token, but for a different application parameter. The error message details are specified in the framing for the underlying transport (see Section "U2F Message Framing" above).

5.4 Authentication Response Message: Success

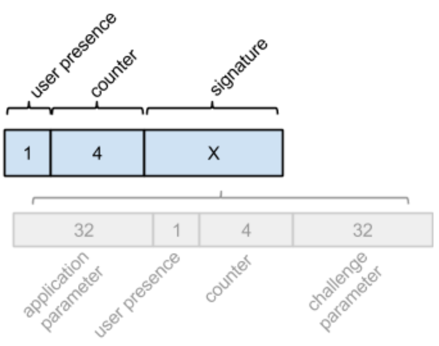


Fig. 5 Authentication Response Message: Success

This message is output by the U2F token after processing/signing the authentication request message described above. Its raw representation is the concatenation of the following:

- A **user presence byte** [1 byte]. Bit 0 indicates whether user presence was verified. If Bit 0 is is to 1, then user presence was verified. If Bit 0 is set to 0, then user presence was not verified. The values of Bit 1 through 7 shall be 0; different values are reserved for future use.

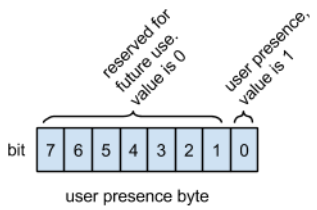


Fig. 6 User Presence Byte Layout

- A **counter** [4 bytes]. This is the big-endian representation of a counter value that the U2F token increments every time it performs an authentication operation. (See Implementation Considerations [U2FImplCons] for more detail.)

- a **signature**. This is a ECDSA signature (on P-256) over the following byte string:

    - The *application parameter* [32 bytes] from the authentication request message.

    - The above *user presence byte* [1 byte].

    - The above *counter* [4 bytes].

    - The *challenge parameter* [32 bytes] from the authentication request message.

  The signature is encoded in ANSI X9.62 format (see [ECDSA-ANSI] in bibliography).

  The signature is to be verified by the relying party using the public key obtained during registration.

6. Other Messages

6.1 GetVersion Request and Response - U2F_VERSION

The FIDO Client can query the U2F token about the U2F protocol version that it implements. The protocol version described in this document is U2F_V2.

The response message's raw representation is the ASCII representation of the string 'U2F_V2' (without quotes, and without any NUL terminator).

The command takes no flags, i.e. P1 and P2 are 0, and takes no data as input. As a result, the complete layout of this command in **short** encoding is, in hexadecimal form:

CLA	INS	P1	P2	Le
00	03	00	00	00

The layout of this command in **extended length** encoding is, in hexadecimal form:

CLA	INS	P1	P2	Le
00	03	00	00	00 00 00

6.2 Extensions and vendor-specific messages

Command codes in the range between **U2F_VENDOR_FIRST** and **U2F_VENDOR_LAST** may be used for vendor-specific implementations. For example, the vendor may choose to put in some testing commands. Note that the FIDO client will never generate these commands. All other command codes are RFU and may not be used.

7. Client Data

Term	Definition
websafe-base64 encoding	This is the "Base 64 Encoding with URL and Filename Safe Alphabet" from Section 5 in [RFC4648] without padding.
stringified javascript object	This is the JSON object (i.e., a string starting with "{" and ending with "}") whose keys are the property names of the javascript object, and whose values are the corresponding property values. Only "data objects" can be stringified, i.e., only objects whose property names and values are supported in **JSON.**

The registration and authentication request messages contain a challenge parameter, which is defined as the SHA-256 hash of a (UTF8 representation of a) stringified JSON data structure that the FIDO client has to prepare. The FIDO Client must send the Client Data (rather than its hash - the challenge parameter) to the relying party during the verification phase, where the relying party can re-generate the challenge parameter (by hashing the client data), which is necessary in order to verify the signature both on the registration response message and authentication response message.

In the case where the FIDO Client is a web browser, the client data is defined as follows (in WebIDL):

WebIDL

```
dictionary ClientData {
    DOMString               typ;
    DOMString               challenge;
    DOMString               origin;
    (DOMString or JwkKey) cid_pubkey;
```

```
};
```

7.1 Dictionary `ClientData` Members

`typ` of type DOMString
: the constant 'navigator.id.getAssertion' for authentication, and 'navigator.id.finishEnrollment' for registration

`challenge` of type DOMString
: the websafe-base64-encoded challenge provided by the relying party

`origin` of type DOMString
: the facet id of the caller, i.e., the web origin of the relying party.
(Note: this might be more accurately called 'facet_id', but for compatibility with existing implementations within Chrome we keep the legacy name.)

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

Otherwise (i.e., both browser and origin server at the above origin support Channel ID), it is present and of type JwkKey.

The JwkKey is a dictionary representing the public key used by a browser for the Channel ID TLS extension. The current version of the Channel ID draft prescribes the algorithm ([ECDSA-ANSI] in bibliography) and curve used, so the dictionary will have the following parameters

WebIDL

```
dictionary JwkKey {
    DOMString kty;
    DOMString crv;
    DOMString x;
    DOMString y;
};
```

7.2 Dictionary `JwkKey` Members

`kty` of type DOMString
: signature algorithm used for Channel ID, i.e., the constant 'EC'

`crv` of type DOMString
: Elliptic curve on which this public key is defined, i.e., the constant 'P-256'

`x` of type DOMString
: websafe-base64-encoding of the x coordinate of the public key (big-endian, 32-byte value)

`y` of type DOMString
: websafe-base64-encoding of the y coordinate of the public key (big-endian, 32-byte value)

8. Examples

8.1 Registration Example

Assume we have a U2F token with the following private attestation key:


```
f3fccc0d00d8031954f90864d43c247f4bf5f0665c6b50cc17749a27d1cf7664
```


the corresponding public key:


```
048d617e65c9508e64bcc5673ac82a6799da3c1446682c258c463fffdf58dfd2fa3e6c378b53d795c4a4dffb4199edd7862f23abaf0203b4b8911ba0569994e101
```


and the following attestation cert:

```
[
[
Version: V3
Subject: CN=PilotGnubby-0.4.1-47901280001155957352 Signature Algorithm: SHA256withECDSA, OID = 1.2.840.10045.4.3.2

Key: EC Public Key
X:
8d617e65c9508e64bcc5673ac82a6799da3c1446682c258c463fffdf58dfd2fa Y:
3e6c378b53d795c4a4dffb4199edd7862f23abaf0203b4b8911ba0569994e101

Validity: [From: Tue Aug 14 11:29:32 PDT 2012, To: Wed Aug 14 11:29:32 PDT 2013]

Issuer: CN=Gnubby Pilot
SerialNumber: [ 47901280 00115595 7352] ]

Algorithm: [SHA256withECDSA]
Signature:
0000: 30 44 02 20 60 CD B6 06 1E 9C 22 26 2D 1A AC 1D 0D. `....."&-...
0010: 96 D8 C7 08 29 B2 36 65 31 DD A2 68 83 2C B8 36 ....).6e1..h.,.6
0020: BC D3 0D FA 02 20 63 1B 14 59 F0 9E 63 30 05 57 ..... c..Y..c0.W
0030: 22 C8 D8 9B 7F 48 88 3B 90 89 B8 8D 60 D1 D9 79 "....H.;....`..y
0040: 59 02 B3 04 10 DF Y.....
]
```

The attestation cert in hex form:


```
3082013c3081e4a003020102020a47901280001155957352300a06082a8648ce3d0403023017311530130603550403130c476e756262792050696c6f74301e170d3132303831
```


Now let's assume that we use the following client data


```
{"typ":"navigator.id.finishEnrollment","challenge":"vqrS6WXDe1JUs5_c3i4-LkKIHRr-3XVb3azuA5TifHo","cid_pubkey":{"kty":"EC","crv":"P-256","x":"HzQwlfXX7Q4S5MtCCnZUNBw3RMzPO9tOyWjBqRl4tJ8","y":"XVguGFLIZx1fXg3wNqfdbn75hi4-_7-BxhMljw42Ht4"},"origin":"http://example.com"}
```


with hash:


```
4142d21c00d94ffb9d504ada8f99b721f4b191ae4e37ca0140f696b6983cfacb
```


and application id:

```
http://example.com
```

with hash:

```
f0e6a6a97042a4f1f1c87f5f7d44315b2d852c2df5c7991cc66241bf7072d1c4
```

to construct a registration request message.

Let's say the U2F token generates the following key pair:

Private key:

```
9a9684b127c5e3a706d618c86401c7cf6fd827fd0bc18d24b0eb842e36d16df1
```

Public key:

```
04b174bc49c7ca254b70d2e5c207cee9cf174820ebd77ea3c65508c26da51b657c1cc6b952f8621697936482da0a6d3d3826a59095daf6cd7c03e2e60385d2f6d9
```

Associated key handle:

```
2a552dfdb7477ed65fd84133f86196010b2215b57da75d315b7b9e8fe2e3925a6019551bab61d16591659cbaf00b4950f7abfe6660e2e006f76868b772d70c25
```

The signature base string for the registration response message is therefore:

```
00f0e6a6a97042a4f1f1c87f5f7d44315b2d852c2df5c7991cc66241bf7072d1c44142d21c00d94ffb9d504ada8f99b721f4b191ae4e37ca0140f696b6983cfacb2a552dfdb7
```

A possible signature over the base string with the above private attestation key is:

```
304502201471899bcc3987e62e8202c9b39c33c19033f7340352dba80fcab017db9230e402210082677d673d891933ade6f617e5dbde2e247e70423fd5ad7804a6d3d3961ef
```

Which means the whole registration response message is:

```
0504b174bc49c7ca254b70d2e5c207cee9cf174820ebd77ea3c65508c26da51b657c1cc6b952f8621697936482da0a6d3d3826a59095daf6cd7c03e2e60385d2f6d9402a552
```

from which (together with challenge and application parameters) the signature base string and signature can be extracted, and verified with the public key from the attestation cert.

8.2 Authentication Example

Let's assume we have a U2F device with private key:

```
ffa1e110dde5a2f8d93c4df71e2d4337b7bf5ddb60c75dc2b6b81433b54dd3c0
```

and corresponding public key:

```
04d368f1b665bade3c33a20f1e429c7750d5033660c019119d29aa4ba7abc04aa7c80a46bbe11ca8cb5674d74f31f8a903f6bad105fb6ab74aefef4db8b0025e1d
```

Example application id:

```
https://gstatic.com/securitykey/a/example.com
```

Example client data:

{"typ":"navigator.id.getAssertion","challenge":"opsXqUifDriAAmWclinfbS0e-USY0CgyJHe_Otd7z8o","cid_pubkey":{"kty":"EC","crv":"P-256","x":"HzQwlfXX7Q4S5MtCCnZUNBw3RMzPO9tOyWjBqRl4tJ8","y":"XVguGFLIZx1fXg3wNqfdbn75hi4-_7-BxhMljw42Ht4"},"origin":"http://example.com"}

Hash of the above client data (challenge parameter):

```
ccd6ee2e47baef244d49a222db496bad0ef5b6f93aa7cc4d30c4821b3b9dbc57
```

Hash of the above application id (application parameter):

```
4b0be934baebb5d12d26011b69227fa5e86df94e7d94aa2949a89f2d493992ca
```

Assuming counter = 1 and user_presence = 1, signature base string is:

```
4b0be934baebb5d12d26011b69227fa5e86df94e7d94aa2949a89f2d493992ca0100000001ccd6ee2e47baef244d49a222db496bad0ef5b6f93aa7cc4d30c4821b3b9dbc57
```

A possible signature with above private key is:

```
304402204b5f0cd17534cedd8c34ee09570ef542a353df4436030ce43d406de870b847780220267bb998fac9b7266eb60e7cb0b5eabdfd5ba9614f53c7b22272ec10047a923
```

Authentication Response Message:

```
0100000001304402204b5f0cd17534cedd8c34ee09570ef542a353df4436030ce43d406de870b847780220267bb998fac9b7266eb60e7cb0b5eabdfd5ba9614f53c7b22272e
```

The above signature and signature base string can be reconstructed from the authentication response message and the challenge and application parameters, and can be verified with the above public key.

9. Implementation Considerations

Earlier revisions of the FIDO U2F specifications defined the U2F_VERSION command with the following byte layout:

CL	IN	P1	P2	L0	L1	L2	Le
00	03	00	00	00	00	00	00 00

This is not compatible with ISO 7816-4. (Compatible encodings are defined earlier in this document.)

For maximum compatibility with U2F Authenticators that followed the earlier specification for the U2F_VERSION command, U2F Clients may choose to support this older encoding over the HID protocol, the only protocol defined which used this encoding.

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

[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[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-u2f-v1.2-ps-20170411/fido-appid-and-facets-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-appid-and-facets-v1.2-ps-20170411.pdf

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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

[U2FHeader]
J. Ehrensvard, *FIDO U2F HID Header Files v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://github.com/fido-alliance/u2f-specs/blob/master/inc/u2f.h

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

A.2 Informative references

[U2FImplCons]
D. Balfanz, *FIDO U2F Implementation Considerations v1.0*. FIDO Alliance Review Draft (Work in progress.) URL:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-implementation-considerations-v1.2-ps-20170411.pdf

[U2FOverview]
S. Srinivas, D. Balfanz, E. Tiffany, *FIDO U2F Overview v1.0*. FIDO Alliance Review Draft (Work in progress.) URL:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.pdf

FIDO U2F Authenticator Transports Extension

FIDO Alliance Proposed Standard 11 April 2017

This version:
: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-authenticator-transports-v1.1-v1.2-ps-20170411.html

Editors:
: Juan Lang, Google, Inc.
: Robin Bertels, STMicroelectronics
: Alexei Czeskis, Google, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

FIDO-compliant relying parties may wish to offer tailored user interfaces based on the transports a FIDO U2F authenticator supports. This standard describes one way relying parties may learn which transports an authenticator supports, by allowing authenticator vendors to embed hardware features as an optional extension in the authenticator's attestation certificate.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

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

Table of Contents

1. Document Information

1.1 Notation

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

1.1.1 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. Attestation certificates

Attestation certificates are X.509 certificates. Transports supported by an authenticator can be embedded as an extension in the authenticator's attestation certificate. As certificate extensions are only available since [X509V3], the attestation certificate's version must be v3.

As such, this specification is a profile of [RFC5280] which is itself a profile of the ISO/IEC/ITU-T [X509V3] specifications for public key certificates. All syntax and semantics are inherited from those specifications unless explicitly documented otherwise. In this document, all fields are defined in ASN.1 and must be DER-encoded ([X690]).

3. FIDO U2F extensions

3.1 FIDO U2F OID arc

The FIDO OID arc and its FIDO U2F OID subarc are defined as:

```
-- FIDO Alliance’s OID
id-fido OBJECT IDENTIFIER ::= 1.3.6.1.4.1.45724

-- FIDO U2F protocol OID
id-fido-u2f OBJECT IDENTIFIER ::= { id-fido 2 }
```

3.2 FIDO U2F certificate extensions

The FIDO U2F certificate extensions arc is defined as:

```
-- FIDO U2F certificate extensions arc
id-fido-u2f-ce OBJECT IDENTIFIER ::= { id-fido-u2f 1 }
```

3.2.1 FIDO U2F certificate transports extension

This extension is identified by `id-fido-u2f-ce-transports` and specifies the transports supported by the authenticator. It's a non-critical extension and therefore FIDO clients and relying parties may ignore it, if present.

The FIDO U2F certificate transports extension is defined as:

```
-- FIDO U2F certificate extensions
id-fido-u2f-ce-transports OBJECT IDENTIFIER ::= { id-fido-u2f-ce 1 }

fidoU2FTransports EXTENSION ::= {
  WITH SYNTAX FIDOU2FTransports
  ID id-fido-u2f-ce-transports
}

FIDOU2FTransports ::= BIT STRING {
  bluetoothRadio(0), -- Bluetooth Classic
  bluetoothLowEnergyRadio(1),
  uSB(2),
  nFC(3),
  uSBInternal(4)
}
```

3.3 Examples

3.3.1 BT classic authenticator

EXAMPLE 1

```
SEQUENCE                                | 30 13
  OBJECT IDENTIFIER                     |   06 0B
    value: id-fido-u2f-ce-transports    |     2B 06 01 04 01 82 E5 1C 02 01 01
  OCTET STRING                          |   04 04
    BIT STRING                          |     03 02
      unused bits: 7                    |       07
      value: 0x80                       |       80
```

3.3.2 USB + NFC authenticator

EXAMPLE 2

```
SEQUENCE                                | 30 13
  OBJECT IDENTIFIER                     |   06 0B
    value: id-fido-u2f-ce-transports    |     2B 06 01 04 01 82 E5 1C 02 01 01
```

```
    OCTET STRING                          04 04
      BIT STRING                            03 02
        unused bits: 4                        04
        value: 0x30                           30
```

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

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

[X509V3]
ITU-T Recommendation X.509 version 3 (1997). "Information Technology - Open Systems Interconnection - The Directory Authentication Framework" ISO/IEC 9594-8:1997.. URL: undefined

[X690]
Recommendation X.690 — Information Technology — ASN.1 Encoding Rules — Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). URL: https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

FIDO U2F JavaScript API

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-javascript-api-v1.1-v1.2-ps-20170411.html
Previous version:
https://fidoalliance.org/specs/fido-u2f-javascript-api-v1.1-RD-20140209.html
Editors:
Dirk Balfanz, Google, Inc.
Arnar Birgisson, Google, Inc.
Juan Lang, Google, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Background

This section is non-normative.

CHANGES: This version *version 1.1* of the FIDO U2F JavaScript API specification supersedes version JavaScript API 1.0. The major difference between these two versions is the way that requests to be signed are formatted between the RP and the client: In version 1.0, a separate *appId* and *challenge* were sent for every *keyHandle*, whereas in version 1.1, an optimization is made that requires only a single *appId* and *challenge* for multiple *keyHandles*.

LOW-LEVEL API: Although this specification refers to two separate API levels, we want to discourage a Relying Party (RP) from implementing directly against the Low-level MessagePort API as this may be deprecated in future versions of this specification. RPs should rather implement against the High-level JavaScript API and use a library that abstracts the lower-level MessagePort API if required.

Abstract

The U2F JavaScript API consists of two calls - one to register a U2F token with a relying party (i.e., cause the U2F token to generate a new key pair, and to introduce the new public key to the relying party), and one to sign an identity assertion (i.e., exercise a previously-registered key pair).

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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

Table of Contents

1. Notation

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

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

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

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

U2F 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].

Below we explain some of the terms used in this document:

Term	Definition
websafe-base64 encoding	This is the "Base 64 Encoding with URL and Filename Safe Alphabet" from Section 5 in [RFC4648] without padding.
stringified javascript object	This is the JSON object (i.e., a string starting with "{" and ending with "}") whose keys are the property names of the javascript object, and whose values are the corresponding property values. Only "data objects" can be stringified, i.e., only objects whose property names and values are supported in **JSON.**

2. Introduction

Note: Reading the 'FIDO U2F Overview' (see [U2FOverview] in bibliography) is recommended as a background for this document.

A Relying Party (RP) consumes identity assertions from U2F tokens. The RP's web pages communicate with the U2F tokens on the client through a JavaScript API. The RP also needs to perform some verification steps on the server side (see below). How the data obtained by the RP's JavaScript is transferred to the RP's server is out of scope of this document. We instead describe the JavaScript API used by the RP.

3. API Levels

The U2F API may be exposed to web pages on two levels. On the required lower level, RPs interact with the FIDO client through a MessagePort [WEBMESSAGING] object. The low-level MessagePort API defines the message formats for messages sent and received on the port, for the two operations supported by the API. This specification does not describe how such a port is made available to RP web pages, as this is (for now) implementation and browser dependent.

For convenience, the FIDO client may also expose a high-level JavaScript API built on top of the MessagePort API. This API consists of functions corresponding to the different requests that can be made to the FIDO client. These functions respond to the RP asynchronously by invoking a callback.

Why two API levels? The messaging API requires only that pages obtain a MessagePort instance to the FIDO client, i.e. no code needs to be injected to JavaScript context of the RP's pages. This allows RPs to keep full control over the JS running in their pages. The JS API is offered as a convenient abstraction of the messaging API, and is useful for RP developers to quickly integrate U2F into their websites.

3.1 Low-level MessagePort API

RP web pages communicate with the FIDO client over an instance of the HTML5 MessagePort interface. Client implementations may choose how this instance is made available to web pages.

Messages sent to the FIDO client should be `U2fRequest` dictionaries:

WebIDL

```
dictionary U2fRequest {
    DOMString       type;
    DOMString?      appId;
    unsigned long?  timeoutSeconds;
    unsigned long?  requestId;
};
```

3.1.1 Dictionary `U2fRequest` Members

`type` of type DOMString
: The type of request, either "u2f_register_request" or "u2f_sign_request".

`appId` of type DOMString, nullable

An application identifer for the request. If none is given, the origin of the calling web page is used.

timeoutSeconds of type unsigned long, nullable
A timeout for the FIDO Client's processing, in seconds.

requestId of type unsigned long, nullable
An integer identifying this request from concurrent requests.

Subtypes of `U2fRequest` for register and sign requests are defined below in their respective sections. If `timeoutSeconds` is omitted, timeout behavior is unspecified. If `requestId` is present, the FIDO client must include its value the corresponding `Response` dictionary under the same key.

Responses from the FIDO client to the RP webpage should be `U2fResponse` dictionaries:

WebIDL

```
dictionary U2fResponse {
    DOMString                                   type;
    (Error or RegisterResponse or SignResponse) responseData;
    unsigned long?                              requestId;
};
```

3.1.2 Dictionary `U2fResponse` Members

type of type DOMString
The response type, either "u2f_register_response" or "u2f_sign_response"

responseData of type (Error or RegisterResponse or SignResponse)
The response data, see 5. U2F operations

requestId of type unsigned long, nullable
The `requestId` value of the corresponding request, if present. Otherwise omitted.

Errors are indicated by an `Error` dictionary sent as the response data. An error dictionary can be identified by checking for its non-zero integer `errorCode` key. `RegisterResponse` and `SignResponse` do not define this key. An error object may optionally contain a string `errorMessage` with further description of the error.

WebIDL

```
dictionary Error {
    ErrorCode  errorCode;
    DOMString? errorMessage;
};
```

3.1.3 Dictionary `Error` Members

errorCode of type *ErrorCode*
An error code from the `ErrorCode` enumeration.

errorMessage of type DOMString, nullable
A description of the error.

3.2 High-level JavaScript API

A FIDO client may provide a JavaScript convenience API that abstracts the lower-level MessagePort API. Implementations may choose how to make such an API available to RP web pages. If such an API is provided, it should provide a namespace object `u2f` of the following interface.

WebIDL

```
interface u2f {
    void register (DOMString appId, sequence<RegisterRequest> registerRequests, sequence<RegisteredKey> registeredKeys, function(Regist
    void sign (DOMString appId, DOMString challenge, sequence<RegisteredKey> registeredKeys, function(SignResponse or Error) callback, 
};
```

3.2.1 Methods

register

Parameter	Type	Nullable	Optional	Description
appId	`DOMString`	✘	✘	An application id for the request.
registerRequests	`sequence<RegisterRequest>`	✘	✘	Register requests, one for each U2F protocol version accepted by RP
registeredKeys	`sequence<RegisteredKey>`	✘	✘	Identifiers for already registered tokens
callback	`function(RegisterResponse or Error)`	✘	✘	Response handler
opt_timeoutSeconds	`unsigned long`	✔	✔	Timeout in seconds, for the FIDO client's handling of the request.

Return type: `void`

sign

Parameter	Type	Nullable	Optional	Description
appId	`DOMString`	✘	✘	An application id for the request.
challenge	`DOMString`	✘	✘	The websafe-base64-encoded challenge.
registeredKeys	`sequence<RegisteredKey>`	✘	✘	Sign requests, one for each registered token
callback	`function(SignResponse or Error)`	✘	✘	Response handler
opt_timeoutSeconds	`unsigned long`	✔	✔	Timeout in seconds, for the FIDO client's handling of the request.

Return type: `void`

The JavaScript API must invoke the provided callbacks with either response objects, or an error object. An error can be detected by testing for a non-zero `errorCode` key.

EXAMPLE 1

```
u2f.sign(reqs, function(response) {
                 if (response.errorCode) {
                   // response is an Error
                   ...
                 } else {
                   // response is a SignResponse
                   ...
                 }
               });
```

4. U2F transports

A U2F token may support one or more of the low-level transport mechanisms. In order to improve user experience, the RP may indicate to the client which transports a particular key handle uses. It does so through the use of the `Transport` enumeration:

WebIDL

```
enum Transport {
    "bt",
    "ble",
    "nfc",
    "usb",
    "usb-internal"
};
```

Enumeration description	
`bt`	Bluetooth Classic (Bluetooth BR/EDR)
`ble`	Bluetooth Low Energy (Bluetooth Smart)
`nfc`	Near-Field Communications
`usb`	USB HID
`usb-internal`	Non-removable USB HID

For convenience, all the transports supported by a token may be referred to by:

WebIDL

```
typedef sequence<Transport> Transports;
```

Throughout this specification, the identifier Transports is used to refer to the sequence<`Transport`> type.

5. U2F operations

Regardless of the API level used, the U2F client must support the two operations of registering a token, and generating a signed assertion. This section describes the interface to each operation, their corresponding request and response dictionaries and possible error codes.

5.1 Registration

To register a U2F token for a user account at the RP, the RP must:

- decide which U2F protocol version(s) of device it wants to register,
- pick an appropriate application id for the registration request,
- generate a random challenge, and
- store all private information associated with the registration (expiration times, user ids, etc.)

The RP may choose an application id for the registration request. If none is chosen, the RP's web origin is used as the application id. The new key pair that the U2F token generates will be associated with this application id. (For application id details see [FIDOAppIDAndFacets] in bibliography).

For each version it is willing to register, it then prepares a `RegisterRequest` dictionary as follows:

WebIDL

```
dictionary RegisterRequest {
    DOMString version;
    DOMString challenge;
};
```

5.1.1 Dictionary `RegisterRequest` Members

`version` of type DOMString
: The version of the protocol that the to-be-registered token must speak. E.g. "U2F_V2".

`challenge` of type DOMString
: The websafe-base64-encoded challenge.

Additionally, the RP should prepare a `RegisteredKey` for each U2F token that is already registered for the current user as follows:

WebIDL

```
dictionary RegisteredKey {
    DOMString   version;
    DOMString   keyHandle;
    Transports? transports;
    DOMString?  appId;
};
```

5.1.2 Dictionary `RegisteredKey` Members

`version` of type DOMString
: Version of the protocol that the to-be-registered U2F token must speak. E.g. "U2F_V2"

keyHandle of type DOMString
: The registered keyHandle to use for signing, as a websafe-base64 encoding of the key handle bytes returned by the U2F token during registration.

transports of type *Transports*, nullable
: The transport(s) this token supports, if known by the RP.

appId of type DOMString, nullable
: The application id that the RP would like to assert for this key handle, if it's distinct from the application id for the overall request. (Ordinarily this will be omitted.)

The RP delivers a registration request to the FIDO client either via the low-level MessagePort API, or by invoking the high-level JavaScript API. Using the low-level MessagePort API, the RP would construct a message of the `U2fRegisterRequest` type:

WebIDL

```
dictionary U2fRegisterRequest : U2fRequest {
    DOMString                  type = 'u2f_register_request';
    sequence<RegisterRequest>  registerRequests;
    sequence<RegisteredKey>    registeredKeys;
};
```

5.1.3 Dictionary `U2fRegisterRequest` Members

type of type DOMString, defaulting to `'u2f_register_request'`
: sequence<RegisterRequest> registerRequests

registerRequests of type sequence<*RegisterRequest*>

registeredKeys of type sequence<*RegisteredKey*>
: An array of RegisteredKeys representing the U2F tokens registered to this user.

EXAMPLE 2

```
// Low-level API
var port = <obtain U2F MessagePort in a browser specific manner>;
port.addEventListener('message', responseHandler);
port.postMessage({
 'type': 'u2f_register_request',
 'appId': <Application id>,
 'registerRequests': [<RegisterRequest instance>, ...],
 'registeredKeys': [<RegisteredKey for known token 1>, ...],
 'timeoutSeconds': 30,
 'requestId': <unique integer>  // optional
});
```

Using the high-level API, the values are passed as parameters:

EXAMPLE 3

```
// High-level API
u2f.register(<Application id>,
             [<RegisterRequest instance>, ...],
             [<RegisteredKey for known token 1>, ...],
             registerResponseHandler);
```

The FIDO client should treat the order of RegisterRequest dictionaries in the first parameter as a prioritized list. That is, if multiple tokens are present that support more than one version provided by the RP, the version that appears first should be selected. Note that this means multiple RegisterRequests with the same version are redundant, since the first one will always be selected.

Note also that the responseHandler in the low-level API receives a `Response` object, while the `registerResponseHandler` in the high-level API receives the `Error` or `RegisterResponse` objects directly.

The FIDO client will create the raw registration messages from this data (see [U2FRawMsgs] in bibliography), and attempt to perform a registration operation with a U2F token. The registration request message is then used to register a U2F token that is not already registered (if such a token is present).

Note that as part of creating the registration request message, the FIDO client will create a Client Data object (see [U2FRawMsgs]). This Client Data object will be returned to the caller as part of the registration response (see below).

If the registration is successful, the FIDO client returns (via the message port, or the JS API callback) a `RegisterResponse` dictionary as follows.

WebIDL

```
dictionary RegisterResponse {
    DOMString version;
    DOMString registrationData;
    DOMString clientData;
};
```

5.1.4 Dictionary `RegisterResponse` Members

version of type DOMString
: The version of the protocol that the registered token speaks. E.g. "U2F_V2".

registrationData of type DOMString
: The raw registration response websafe-base64

clientData of type DOMString
: The client data created by the FIDO client, websafe-base64 encoded.

For the contents of these fields, refer to [U2FRawMsgs] (see bibliography).

5.2 Generating signed identity assertions

To obtain an identity assertion from a locally-attached U2F token, the RP must

- generate a random challenge, and

- prepare a `RegisteredKey` object for each U2F token that the user has currently registered with the RP.

The RP delivers a sign request to the FIDO client either via the low-level MessagePort API, or by invoking the high-level JavaScript API. Using the low-level MessagePort API, the RP would construct a message of the `U2fSignRequest` type:

WebIDL

```
dictionary U2fSignRequest : U2fRequest {
    DOMString                 type = 'u2f_sign_request';
    DOMString                 challenge;
    sequence<RegisteredKey>  registeredKeys;
};
```

5.2.1 Dictionary `U2fSignRequest` Members

`type` of type DOMString, defaulting to `'u2f_sign_request'`
: DOMString challenge

`challenge` of type DOMString
: The websafe-base64-encoded challenge.

`registeredKeys` of type sequence<*RegisteredKey*>
: An array of RegisteredKeys representing the U2F tokens registered to this user.

EXAMPLE 4

```
// Low-level API
var port = <obtain U2F MessagePort in a browser specific manner>;
port.addEventListener('message', responseHandler);
port.postMessage({
 'type': 'u2f_sign_request',
 'appId': <Application id>,
 'challenge': <random challenge>,
 'registeredKeys': [<RegisteredKey for known token 1>, ...],
 'timeoutSeconds': 30,
 'requestId': <unique integer>  // optional
});
```

In response to a sign request, the FIDO client should perform the following steps:

- Verify the application identity of the caller.
- Using the provided challenge, create a client data object.
- Using the client data, the application id, and the key handle, create a raw authentication request message (see [U2FRawMsgs] in bibliography) and send it to the U2F token.

When the RP provides the `transports` value for any `RegisteredKey`, the client may treat that value has a hint about which transports to prefer for the key handle. The client may also use the transports as a hint about user interface, if the client presents any. Irrespective of whether the RP sets any `transports` value for any `RegisteredKey`, the client should send each key handle over all transports supported by the client.

Eventually the FIDO client must respond (via the MessageChannel or the provided callback). In the case of an error, an `Error` dictionary is returned. In case of success, a `SignResponse` is returned.

WebIDL

```
dictionary SignResponse {
    DOMString keyHandle;
    DOMString signatureData;
    DOMString clientData;
};
```

5.2.2 Dictionary `SignResponse` Members

`keyHandle` of type DOMString
: The keyHandle of the RegisteredKey that was processed.

`signatureData` of type DOMString
: The raw response from U2F device, websafe-base64 encoded.

`clientData` of type DOMString
: The client data created by the FIDO client, websafe-base64 encoded.

If there are multiple U2F tokens that responded to the authentication request, the FIDO client will pick one of the responses and pass it to the caller.

5.3 Error codes

When an `Error` object is returned, its `errorCode` field is set to a non-negative integer indicating the general error that occurred, from the following enumeration.

WebIDL

```
interface ErrorCode {
    const short OK = 0;
    const short OTHER_ERROR = 1;
    const short BAD_REQUEST = 2;
    const short CONFIGURATION_UNSUPPORTED = 3;
    const short DEVICE_INELIGIBLE = 4;
    const short TIMEOUT = 5;
};
```

5.3.1 Constants

`OK` of type short
: Success. Not used in errors but reserved

`OTHER_ERROR` of type short
: An error otherwise not enumerated here

`BAD_REQUEST` of type short
: The request cannot be processed

`CONFIGURATION_UNSUPPORTED` of type short
: Client configuration is not supported

`DEVICE_INELIGIBLE` of type short
: The presented device is not eligible for this request. For a registration request this may mean that the token is already registered, and for a sign request it may mean the token does not know the presented key handle.

`TIMEOUT` of type short
: Timeout reached before request could be satisfied

5.4 Backward compatibility with U2F 1.0 API

For backward compatibility with the U2F 1.0 API, the RP may prepare a `SignRequest` in lieu of a `RegisteredKey` for each U2F token that is already registered for the current user. See JavaScript API 1.0 for the specification of `SignRequest`.

Similarly, U2F clients may implement backward compatibility with version 1.0 by accepting a `signRequests` key in lieu of `registeredKeys`.

A. References

A.1 Normative references

[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[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-u2f-v1.2-ps-20170411/fido-appid-and-facets-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-appid-and-facets-v1.2-ps-20170411.pdf

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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

[U2FRawMsgs]
D. Balfanz, *FIDO U2F Raw Message Formats v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.pdf

[WEBMESSAGING]
Ian Hickson. *HTML5 Web Messaging*. 19 May 2015. W3C Recommendation. URL: https://www.w3.org/TR/webmessaging/

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

A.2 Informative references

[U2FOverview]
S. Srinivas, D. Balfanz, E. Tiffany, *FIDO U2F Overview v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.pdf

FIDO U2F HID Protocol Specification

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.1-v1.2-ps-20170411.html
Previous version:
https://fidoalliance.org/specs/fido-u2f-hid-protocol-v1.1-Member Submission-20140721.html
Editors:
Jakob Ehrensvärd, Yubico
John Kemp, FIDO Alliance

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

U2FHID protocol description and implementation specification

The purpose of this documentation is to provide a complete specification how to implement the U2FHID protocol, where FIDO U2F messages are framed for USB transport, using the HID protocol. General FIDO and U2F- concepts, semantics, meaning is beyond the scope of this document and for information on these topics, please refer to the appropriate related documentation.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

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

Table of Contents

1. Document Information

1.1 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].

Symbolic constants such as **U2FHID_MSG** which are referred to when defining messages in this documents have their values defined in [U2FHIDHeader] in the bibliography.

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

1.1.1 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].

1.2 Definitions

Term	Definition
U2F	Universal Second Factor
USB	Universal Serial Bus
HID	Human Interface Device. A specification of typical USB devices used for human interaction, such as keyboards, mice, joysticks etc.
U2FHID	U2F transport over HID as defined by this document

2. U2FHID protocol implementation

This description does not describe the actual raw U2F messages, semantics and functionality but rather how such messages are framed for HID transport. The raw U2F messages are defined in [U2FRawMsgs]. For the U2FHID protocol, all raw U2F messages are encoded using **extended length** APDU encoding.

2.1 U2FHID implementation rationale

The U2FHID protocol is designed with the following design objectives in mind

- Driver-less installation on all major host platforms
- Multi-application support with concurrent application access without the need for serialization and centralized dispatching.
- Fixed latency response and low protocol overhead
- Scalable method for U2FHID device discovery

Since HID data is sent as interrupt packets and multiple applications may access the HID stack at once, a non-trivial level of complexity has to be added to handle this.

2.2 Protocol structure and data framing

The U2F protocol is designed to be concurrent and state-less in such a way that each performed function is not dependent on previous actions. However, there has to be some form of "atomicity" that varies between the characteristics of the underlying transport protocol, which for the U2FHID protocol introduces the following terminology:

- Transaction
- Message
- Packet

A **transaction** is the highest level of aggregated functionality, which in turn consists of a request, followed by a response message. Once a request has been initiated, the transaction has to be entirely completed before a second transaction can take place and a response is never sent without a previous request.

Request- and response **messages** are in turn divided into individual fragments, known as **packets**. The packet is the smallest form of protocol data unit, which in the case of U2FHID are mapped into HID reports.

2.3 Concurrency and channels

Additional logic and overhead is required to allow a U2FHID device to deal with multiple "clients", i.e. multiple applications accessing the single resource through the HID stack. Each client communicates with a U2FHID device through a logical **channel**, where each application uses a unique 32-bit **channel identifier** for routing- and arbitration purposes.

A channel identifier is allocated by the U2F device to ensure its system-wide uniqueness. The actual algorithm for generation of channel identifiers is vendor specific and not defined by this specification.

Channel ID 0 is reserved and `0xffffffff` is reserved for broadcast commands, i.e. at the time of channel allocation.

2.4 Message- and packet structure

Packets are one of two types, **initialization packets** and **continuation packets**. As the name suggests, the first packet sent in a message is an initialization packet, which also becomes the start of a transaction. If the entire message does not fit into one packet (including the U2FHID protocol overhead), one or more continuation packets have to be sent in strict ascending order to complete the message transfer.

A message sent from a host to a device is known as a**request** and a message sent from a device back to the host is known as a **response**. A request always triggers a response and response messages are never sent ad-hoc, i.e. without a prior request message.

The request and response messages have an identical structure. A transaction is started with the initialization packet of the request message and ends with the last packet of the response message.

Packets are always fixed size (defined by the endpoint- and HID report descriptors) and although all bytes may not be needed in a particular packet, the full size always has to be sent. Unused bytes should be set to zero.

An initialization packet is defined as

Offset	Length	Mnemonic	Description
0	4	CID	Channel identifier
4	1	CMD	Command identifier (bit 7 always set)
5	1	BCNTH	High part of payload length
6	1	BCNTL	Low part of payload length

7	(s - 7)	DATA	Payload data (s is equal to the fixed packet size)

The command byte has always the highest bit set to distinguish it from a continuation packet, which is described below.

A continuation packet is defined as

Offset	Length	Mnemonic	Description
0	4	CID	Channel identifier
4	1	SEQ	Packet sequence 0x00..0x7f (bit 7 always cleared)
5	(s - 5)	DATA	Payload data (s is equal to the fixed packet size)

With this approach, a message with a payload less or equal to (s - 7) may be sent as one packet. A larger message is then divided into one or more continuation packets, starting with sequence number 0, which then increments by one to a maximum of 127.

With a packet size of 64 bytes (max for full-speed devices), this means that the maximum message payload length is 64 - 7 + 128 * (64 - 5) = 7609 bytes.

2.5 Arbitration

In order to handle multiple channels and clients concurrency, the U2FHID protocol has to maintain certain internal states, block conflicting requests and maintain protocol integrity. The protocol relies on each client application (channel) behaves politely, i.e. does not actively act to destroy for other channels. With this said, a malign- or malfunctioning application can cause issues for other channels. Expected errors and potentially stalling applications should however be handled properly.

2.5.1 Transaction atomicity, idle- and busy states.

A transaction always consists of three stages:

1. A message is sent from the host to the device
2. The device processes the message
3. A response is sent back from the device to the host

The protocol is built on the assumption that a plurality of concurrent applications may try ad-hoc to perform transactions at any time, with each transaction being atomic, i.e. it cannot be interrupted by another application once started.

The application channel that manages to get through the first initialization packet when the device is in idle state will keep the device locked for other channels until the last packet of the response message has been received. The device then returns to idle state, ready to perform another transaction for the same or a different channel. Between two transactions, no state is maintained in the device and a host application must assume that any other process may execute other transactions at any time.

If an application tries to access the device from a different channel while the device is busy with a transaction, that request will immediately fail with a busy-error message sent to the requesting channel.

2.5.2 Transaction timeout

A transaction has to be completed within a specified period of time to prevent a stalling application to cause the device to be completely locked out for access by other applications.

If for example an application sends an initialization packet that signals that continuation packets will follow and that application crashes, the device will back out that pending channel request and return to an idle state.

2.5.3 Transaction abort and re-synchronization

If an application for any reason "gets lost", gets an unexpected response or error, it may at any time issue an abort-and-resynchronize command. If the device detects a SYNC command during a transaction that has the same channel id as the active transaction, the transaction is aborted (if possible) and all buffered data flushed (if any). The device then returns to idle state to become ready for a new transaction.

2.5.4 Packet sequencing

The device keeps track of packets arriving in correct and ascending order and that no expected packets are missing. The device will continue to assemble a message until all parts of it has been received or that the transaction times out. Spurious continuation packets appearing without a prior initialization packet will be ignored.

2.6 Channel locking

In order to deal with aggregated transactions that may not be interrupted, such as vendor specific tunneling of APDUs, a channel lock command may be implemented. By sending a channel lock command, the device prevents other channels from communicating with the device until the channel lock has timed out or been explicitly unlocked by the application.

This feature is optional and has not to be considered by general U2F HID applications.

2.7 Protocol version and compatibility

The U2FHID protocol is designed to be extensible, yet maintaining backwards compatibility to the extent it is applicable. This means that a U2FHID host shall support any version of a device with the command set available in that particular version.

3. HID device implementation

This description assumes knowledge of the USB- and HID specifications and is intended to provide the basics for implementing a U2FHID device. There are several ways to implement USB devices and reviewing these different methods is beyond the scope of this document. This specification targets the interface part, where a device is regarded as either a single- or multiple interface (composite) device.

The description further assumes (but is not limited to) a full-speed USB device (12 Mbit/s). Although not excluded per se, USB low-speed devices are not practical to use given the 8-byte report size limitation together with the protocol overhead.

3.1 Interface- and endpoint descriptors

The device implements two endpoints (except the control endpoint 0), one for IN- and one for OUT transfers. The packet size is vendor defined, but the reference implementation assumes a full-speed device with two 64-byte endpoints.

Interface Descriptor

Mnemonic	Value	Description
bNumEndpoints	2	One IN- and one OUT endpoint
bInterfaceClass	0x03	HID

bInterfaceSubClass	0x00	No interface subclass
bInterfaceProtocol	0x00	No interface protocol

Endpoint 1 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdresss	0x01	1, OUT
bMaxPacketSize	64	64 bytes packets
bInterval	5	Poll every 5 millisecond

Endpoint 2 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdresss	0x81	1, IN
bMaxPacketSize	64	64 bytes packets
bInterval	5	Poll every 5 millisecond

The actual endpoint order, intervals, endpoint numbers and endpoint packet size may be defined freely by the vendor and the host application is responsible for querying these values and handle these accordingly. For the sake of clarity, the values listed above are used in the following examples.

3.2 HID report descriptor and device discovery

A HID report descriptor is required for all HID devices, even though the reports and their interpretation (scope, range, etc.) makes very little sense from an operating system perspective. The U2FHID just provides two "raw" reports, which basically map directly to the IN and OUT endpoints. However, the HID report descriptor has an important purpose in U2FHID, as it is used for device discovery.

For the sake of clarity, a bit of high-level C-style abstraction is provided

EXAMPLE 1

```
// HID report descriptor

const uint8_t HID_ReportDescriptor[] = {
  HID_UsagePage ( FIDO_USAGE_PAGE ),
  HID_Usage ( FIDO_USAGE_U2FHID ),
  HID_Collection ( HID_Application ),
  HID_Usage ( FIDO_USAGE_DATA_IN ),
  HID_LogicalMin ( 0 ),
  HID_LogicalMaxS ( 0xff ),
  HID_ReportSize ( 8 ),
  HID_ReportCount ( HID_INPUT_REPORT_BYTES ),
  HID_Input ( HID_Data | HID_Absolute | HID_Variable ),
  HID_Usage ( FIDO_USAGE_DATA_OUT ),
  HID_LogicalMin ( 0 ),
  HID_LogicalMaxS ( 0xff ),
  HID_ReportSize ( 8 ),
  HID_ReportCount ( HID_OUTPUT_REPORT_BYTES ),
  HID_Output ( HID_Data | HID_Absolute | HID_Variable ),
HID_EndCollection
```

```
    };
```

A unique **Usage Page** is defined for the FIDO alliance and under this realm, a U2FHID **Usage** is defined as well. During U2FHID device discovery, all HID devices present in the system are examined and devices that match this usage pages and usage are then considered to be U2FHID devices.

The length values specified by the `HID_INPUT_REPORT_BYTES` and the `HID_OUTPUT_REPORT_BYTES` should typically match the respective endpoint sizes defined in the endpoint descriptors.

4. U2FHID commands

The U2FHID protocol implements the following commands.

4.1 Mandatory commands

The following list describes the minimum set of commands required by an U2FHID device. Optional- and vendor-specific commands may be implemented as described in respective sections of this document.

4.1.1 U2FHID_MSG

This command sends an encapsulated U2F message to the device. The semantics of the data message is defined in the U2F protocol specification.

Request

CMD	U2FHID_MSG
BCNT	4..n
DATA	n bytes

Response at success

CMD	U2FHID_MSG
BCNT	2..n
DATA	N bytes

4.1.2 U2FHID_INIT

This command synchronizes a channel and optionally requests the device to allocate a unique 32-bit channel identifier (CID) that can be used by the requesting application during its lifetime. The requesting application generates a nonce that is used to match the response. When the response is received, the application compares the sent nonce with the received one. After a positive match, the application stores the received channel id and uses that for subsequent transactions.

To allocate a new channel, the requesting application shall use the broadcast channel U2FHID_BROADCAST_CID. The device then responds the newly allocated channel in the response, using the broadcast channel.

Request

CMD	U2FHID _INIT

BCNT	8
DATA	8 byte nonce

Response at success

CMD	U2FHID _INIT
BCNT	17 (see note below)
DATA	8 byte nonce
DATA+8	4 byte channel ID
DATA+12	U2FHID protocol version identifier
DATA+13	Major device version number
DATA+14	Minor device version number
DATA+15	Build device version number
DATA+16	Capabilities flags

The protocol version identifies the protocol version implemented by the device. An U2FHID host shall accept a response size that is longer than the anticipated size to allow for future extensions of the protocol, yet maintaining backwards compatibility. Future versions will maintain the response structure to this current version, but additional fields may be added.

The meaning and interpretation of the version number is vendor defined.

The following device capabilities flags are defined. Unused values are reserved for future use and must be set to zero by device vendors.

CAPABILITY_WINK	Implements the WINK function

4.1.3 U2FHID_PING

Sends a transaction to the device, which immediately echoes the same data back. This command is defined to be an uniform function for debugging-, latency- and performance measurements.

Request

CMD	U2FHID_PING
BCNT	0..n
DATA	n bytes

Response at success

CMD	U2FHID_PING
BCNT	n
DATA	N bytes

4.1.4 U2FHID_ERROR

This command code is used in response messages only.

CMD	U2FHID_ERROR
BCNT	1
DATA	Error code

The following error codes are defined

ERR_INVALID_CMD	The command in the request is invalid
ERR_INVALID_PAR	The parameter(s) in the request is invalid
ERR_INVALID_LEN	The length field (BCNT) is invalid for the request
ERR_INVALID_SEQ	The sequence does not match expected value
ERR_MSG_TIMEOUT	The message has timed out
ERR_CHANNEL_BUSY	The device is busy for the requesting channel

4.2 Optional commands

The following commands are defined by this specification but are optional and does not have to be implemented.

4.2.1 U2FHID_WINK

The wink command performs a vendor-defined action that provides some visual- or audible identification a particular U2F device. A typical implementation will do a short burst of flashes with a LED or something similar. This is useful when more than one device is attached to a computer and there is confusion which device is paired with which connection.

Request

CMD	U2FHID_WINK
BCNT	0
DATA	N/A

Response at success

CMD	U2FHID_WINK
BCNT	0
DATA	N/A

4.2.2 U2FHID_LOCK

The lock command places an exclusive lock for one channel to communicate with the device. As long as the lock is active, any other channel trying to send a message will fail. In order to prevent a stalling- or crashing application to lock the device indefinitely, a lock time up to 10 seconds may be set. An application requiring a longer lock has to send repeating lock commands to maintain the lock.

Request

CMD	U2FHID_LOCK
BCNT	1
DATA	Lock time in seconds 0..10. A value of 0 immediately releases the lock

Response at success

CMD	U2FHID_LOCK
BCNT	0
DATA	N/A

4.3 Vendor specific commands

A U2FHID may implement additional vendor specific commands that are not defined in this specification, yet being U2FHID compliant. Such commands, if implemented must have a command in the range between U2FHID_VENDOR_FIRST and U2FHID_VENDOR_LAST.

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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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

[U2FHIDHeader]
J. Ehrensvard, *FIDO U2F HID Header Files v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://github.com/fido-alliance/u2f-specs/blob/master/inc/u2f_hid.h

[U2FRawMsgs]
D. Balfanz, *FIDO U2F Raw Message Formats v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.pdf

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

FIDO NFC Protocol Specification v1.0

FIDO Alliance Proposed Standard 11 April 2017

This version:
: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-nfc-protocol-v1.1-v1.2-ps-20170411.html

Editors:
: Alexei Czeskis, Google, Inc.
: Juan Lang, Google, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The FIDO U2F framework was designed to be able to support multiple authenticator form factors. This document describes the communication protocol with authenticators over Near Field Communication (NFC).

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

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

Table of Contents

1. Notation

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

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

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

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

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. Protocol

The general protocol between a FIDO client and authenticator over NFC is as follows:

1. Client sends an applet selection command
2. Authenticator replies with success
3. Client sends a command for an operation (register / authenticate)
4. Authenticator replies with response data or error

The Authenticator must reply to all commands within 800ms.

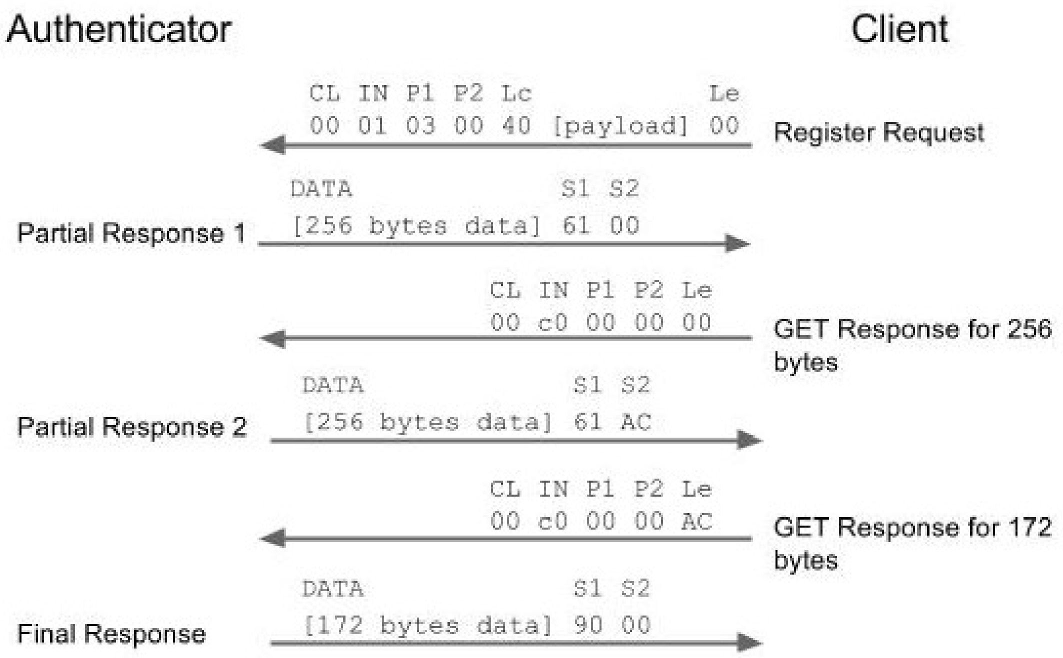
3. Framing

The NFC protocol shall not use any additional framing (unlike the USB HID protocol, for example). Instead, messages sent to an NFC authenticator shall follow the U2F raw message format as defined in [U2FRawMsgs] in the bibliography. In the NFC protocol, either **short** or **extended length** APDU encoding is allowed.

4. APDU Length

Some responses may not fit into a short APDU response. For this reason, U2F authenticators must respond in the following way:

- If the request was encoded using**extended length** APDU encoding, the authenticator must respond using the extended length APDU response format.
- If the request was encoded using**short** APDU encoding, the authenticator must respond using ISO 7816-4 APDU chaining (see Section A.4). See below for an example:



5. Applet selection

A FIDO client shall always send an applet selection command to begin interaction with a FIDO authenticator via NFC. The structure of the applection command shall follow the same APDU structure as in the raw message format mentioned above.

The FIDO U2F AID consists of the following fields:

Field	Value
`RID`	0xA000000647
`AC`	0x2F
`AX`	0x0001

As a result, the command for selecting the applet using the FIDO U2F AID is:

Field	Value
`CLA`	0x00
`INS`	0xA4
`P1`	0x04

P2	0x00
LEN	0x08
DATA	0xA0000006472F0001

In response to the applet selection command, the FIDO authenticator shall reply with its version string in the successful response. In this writing, the version string is "U2F_V2", hence a successful response to the applet selection command would consist of the following bytes:

`0x5532465F56329000`

Note, future versions may respond with other version string values.

6. Implementation Considerations

Correct and reliable functioning of the NFC U2F authenticator requires a reliable contactless communication between the NFC U2F authenticator and the contactless reader device. However, there are currently several relevant specifications describing the contactless proximity interface often summarized under the term “NFC”.

In order to guarantee interoperability, the contactless interface of the NFC U2F authenticators and the various implementations of contactless readers should follow one of the following standards:

a. NFC U2F authenticators should be designed according to ISO/IEC 14443 or ISO/IEC 18092. These standards are commonly used for FIDO authenticators, eID, passports, public transport fare media etc. It is highly recommended to test and certify the conformance of the authenticator to ISO/IEC 14443 or ISO/IEC 18092 by an independent party.
b. For mobile use of FIDO authentication, the reader functionality of NFC-enabled mobile devices will typically be used for NFC U2F authenticators. Mobile devices should be designed according to NFC Forum Analog specification v2.0 or later. NFC Forum also offers testing and certification.

The testing and certification for the above listed specifications will ensure interoperability of NFC U2F authenticators and NFC mobile devices. Generally, all reader devices that may be used with unspecific types of NFC U2F authenticators (see a.) should be conformant to NFC Forum analog specification.

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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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
[U2FRawMsgs]
D. Balfanz, *FIDO U2F Raw Message Formats v1.0*. FIDO Alliance Review Draft (Work

in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.pdf

[WebIDL]

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

FIDO Bluetooth® Specification v1.0

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-bt-protocol-v1.0-v1.2-ps-20170411.html
Previous version:
https://fidoalliance.org/specs/fido-u2f-bt-protocol-v1.0-Member Submission-20140721.html
Editors:
Alexei Czeskis, Google, Inc.
Juan Lang, Google, Inc.
Contributors:
Scott Walsh, Plantronics, Inc.
Deniz Akkaya, Yubico, Inc.
Jakub Pawlowski, Google, Inc.
Hannes Tschofenig, ARM Ltd.
Johan Verrept, VASCO Datasecurity International, Inc

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The FIDO U2F framework was designed to be able to support multiple Authenticator form factors. This document describes the communication protocol with Authenticators over Bluetooth low energy technology.

There are multiple form factors possible for Authenticators. Some might be low cost, low power devices, and others might be implemented as an additional feature of a more powerful device, such as a smartphone. The design proposed here is meant to support multiple form factors, including but not necessarily limited to these two examples.

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](https://www.fidoalliance.org/specifications/) at https://www.fidoalliance.org/specifications/.

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

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

Table of Contents

1. Notation

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

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

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

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

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. Conformance

Authenticator and Client devices using Bluetooth low energy technology shall conform to Bluetooth Core Specification 4.0 or later [BluetoothCORE]

Bluetooth SIG specified UUID values shall be found on the Assigned Numbers website [BluetoothASSNUM]

3. Pairing

Bluetooth low energy technology is a long-range wireless protocol and thus has several implications for privacy, security, and overall user-experience. Because it is wireless, Bluetooth low energy technology may be subject to monitoring, injection, and other network-level attacks.

For these reasons, Clients and Authenticators must create and use a long-term link key (LTK) and shall encrypt all communications. Authenticator must never use short term keys.

Because Bluetooth low energy technology has poor ranging (*i.e.,* there is no good indication of proximity), it may not be clear to a FIDO Client with which Bluetooth low energy technology Authenticator it should communicate. Pairing is the only mechanism defined in this protocol to ensure that FIDO Clients are interacting with the expected Bluetooth low energy technology Authenticator. As a result, Authenticator manufacturers should instruct users to avoid performing Bluetooth pairing in a public space such as a cafe, shop or train station.

One disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This issue is discussed further in Implementation Considerations.

4. Link Security

For Bluetooth low energy technology connections, the Authenticator shall enforce `Security Mode 1, Level 2` (unauthenticated pairing with encryption) or `Security Mode 1, Level 3` (authenticated pairing with encryption) before any U2F messages are exchanged.

5. Framing

Conceptually, framing defines an encapsulation of U2F raw messages responsible for correct transmission of a single request and its response by the transport layer.

All requests and their responses are conceptually written as a single frame. The format of the requests and responses is given first as complete frames. Fragmentation is discussed next for each type of transport layer.

5.1 Request from Client to Authenticator

Request frames must have the following format

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

Supported commands are `PING` and `MSG`. The constant values for them are described below.

The data format for the `MSG` command is defined in [U2FRawMsgs]. For the U2F over Bluetooth protocol, U2F raw messages are encoded using **extended length** APDU encoding.

5.2 Response from Authenticator to Client

Response frames must have the following format, which share a similar format to the request frames:

Offset	Length	Mnemonic	Description
0	1	`STAT`	Response status
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

When the status byte in the response is the same as the command byte in the request, the response is a successful response. The value `ERROR` indicates an error, and the response data contains an error code as a variable-length, big-endian integer. The constant value for `ERROR` is described below.

Note that the errors sent in this response are errors at the encapsulation layer, e.g., indicating an incorrectly formatted request, or possibly an error communicating with the Authenticator's U2F message processing layer. Errors reported by the U2F message processing layer itself are considered a success from the encapsulation layer's point of view, and are reported as a complete `MSG` response.

Data format is defined in [U2FRawMsgs]. Note that as per [U2FRawMsgs] (and unlike the NFC transport specification), all communication shall be done using extended length APDU format.

5.3 Command, Status, and Error constants

The COMMAND constants and values are:

Command Constant	Value
PING	0x81
KEEPALIVE	0x82
MSG	0x83
ERROR	0xbf

The KEEPALIVE command contains a single byte with the following possible values:

Status Constant	Value
PROCESSING	0x01
TUP_NEEDED	0x02
RFU	0x00, 0x03-0xFF

A resulting Keep alive message, including framing, becomes:

Offset	Length	Value	Description
0	1	KEEPALIVE	Command identifier
1	1	0x00	High part of data length
2	1	0x01	Low part of data length
3	1	0xXX	Status byte (see table above)

The ERROR constants and values are:

Error Constant	Value	Meaning
ERR_INVALID_CMD	0x01	The command in the request is unknown/invalid
ERR_INVALID_PAR	0x02	The parameter(s) of the command is/are invalid or missing
ERR_INVALID_LEN	0x03	The length of the request is invalid
ERR_INVALID_SEQ	0x04	The sequence number is invalid
ERR_REQ_TIMEOUT	0x05	The request timed out
NA	0x06	Value reserved (HID)
NA	0x0a	Value reserved (HID)
NA	0x0b	Value reserved (HID)
ERR_OTHER	0x7f	Other, unspecified error

6. GATT Service Description

This profile defines two roles: FIDO Authenticator and FIDO Client.

- The FIDO Client shall be a GATT Client
- The FIDO Authenticator shall be a GATT Server

The following figure illustrates the mandatory services and characteristics that shall be offered by a FIDO Authenticator as part of its GATT server:

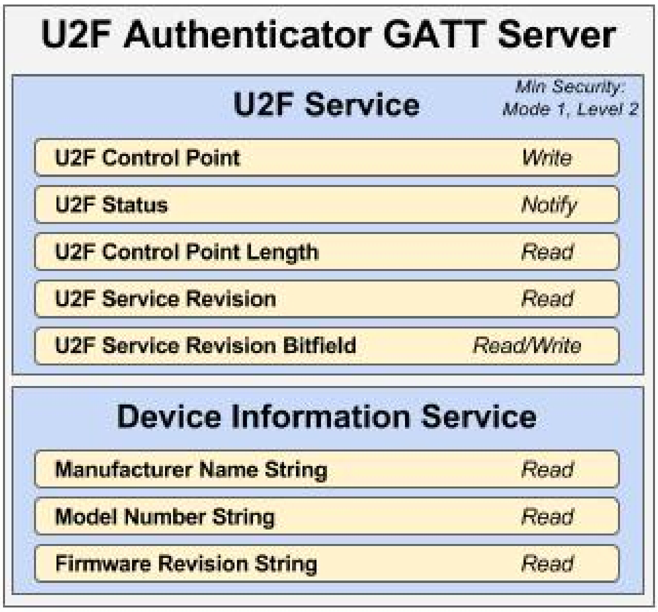


Fig. 1 Mandatory GATT services and characteristics that must be offered by a FIDO Authenticator. Note that the Generic Access Profile service ([BluetoothGAS]) is not present as it is already mandatory for any Bluetooth low energy technology compliant device.

The table below summarizes additional GATT sub-procedure requirements for a FIDO Authenticator (GATT Server) beyond those required by all GATT Servers.

GATT Sub-Procedure	Requirements
Write Characteristic Value	Mandatory
Notifications	Mandatory
Read Characteristic Descriptors	Mandatory
Write Characteristic Descriptors	Mandatory

The table below summarizes additional GATT sub-procedure requirements for a FIDO Client (GATT Client) beyond those required by all GATT Clients.

GATT Sub-Procedure	Requirements
Discover All Primary Services	(*)
Discover Primary Services by Service UUID	(*)
Discover All Characteristics of a Service	(**)
Discover Characteristics by UUID	(**)
Discover All Characteristic Descriptors	Mandatory
Read Characteristic Value	Mandatory

Write Characteristic Value	`Mandatory`
Notification	`Mandatory`
Read Characteristic Descriptors	`Mandatory`
Write Characteristic Descriptors	`Mandatory`

(*): Mandatory to support at least one of these sub-procedures.

(**): Mandatory to support at least one of these sub-procedures.

Other GATT sub-procedures may be used if supported by both client and server.

Specifics of each service are explained below. In the following descriptions: all values are big-endian coded, all strings are in UTF-8 encoding, and any characteristics not mentioned explicitly are optional.

6.1 U2F Service

An Authenticator shall implement the U2F Service described below. The UUID for the FIDO U2F GATT service is `0xFFFD`, it shall be declared as a Primary Service. The service contains the following characteristics:

Characteristic Name	Mnemonic	Property	Length	UUID
U2F Control Point	`u2fControlPoint`	Write	Defined by Vendor (20-512 bytes)	F1D0FFF1-DEAA-ECEE-B42F-C9BA7ED623BB
U2F Status	`u2fStatus`	Notify	N/A	F1D0FFF2-DEAA-ECEE-B42F-C9BA7ED623BB
U2F Control Point Length	`u2fControlPointLength`	Read	2 bytes	F1D0FFF3-DEAA-ECEE-B42F-C9BA7ED623BB
U2F Service Revision	`u2fServiceRevision`	Read	Defined by Vendor (20-512 bytes)	0x2A28
U2F Service Revision Bitfield	`u2fServiceRevisionBitfield`	Read/Write	See below, at least 1 byte	F1D0FFF4-DEAA-ECEE-B42F-C9BA7ED623BB

`u2fControlPoint` is a write-only command buffer.

`u2fStatus` is a notify-only response attribute. The Authenticator will send a series of notifications on this attribute with a maximum length of (ATT_MTU-3) using the response frames defined above. This mechanism is used because this results in a faster transfer speed compared to a notify-read combination.

`u2fControlPointLength` defines the maximum size in bytes of a single write request to `u2fControlPoint`. This value shall be between 20 and 512.

`u2fServiceRevision` defines the revision of the U2F Service. The value is a UTF-8 string. For version 1.0 of the specification, the value `u2fServiceRevision` shall be `1.0` or in raw bytes: `0x312e30`. This field shall be omitted if protocol version 1.0 is not supported.

`u2fServiceRevisionBitfield` defines the revision of the U2F Service. The value is a bit field. Each bit represents the Authenticator's support of a particular protocol version. A bit value of 1 indicates support, while value 0 indicates lack of support. The length of the bitfield is 1 or more bytes. All bytes that are 0 are omitted if all the following bytes are 0 too. The bit field is big endian encoded with the most significant bit representing version 1.1 support, the next most significant bit, representing the next protocol version, etc. If only version 1.0 is supported, this characteristic shall be omitted. If the u2fServiceRevision characteristic is present or more than 1 bit in this u2fServiceRevisionBitfield characteristic is 1, the client shall write the value of the requested protocol version to be used for the lifetime of this connection. If u2fServiceRevision characteristic is not present and only one bit in u2fServiceRevisionBitfield is set, the version that bit represents shall be the default.

Byte (left to right)	Bit	Version
0	7	1.1
0	6	1.2

For example, a device that only supports 1.1 will only have a u2fServiceRevisionBitfield characteristic of length 1 with value 0x80.

The `u2fServiceRevision` Characteristic may include a Characteristic Presentation Format descriptor with format value 0x19, `UTF-8 String`.

6.2 Device Information Service

An Authenticator shall implement the Device Information Service [BluetoothDIS] with the following characteristics:

- Manufacturer Name String
- Model Number String
- Firmware Revision String

All values for the Device Information Service are left to the vendors. However, vendors should not create uniquely identifiable values so that Authenticators do not become a method of tracking users.

6.3 Generic Access Profile service

Every Authenticator shall implement the Generic Access Profile service [BluetoothGAS] with the following characteristics:

- Device Name
- Appearance

7. Protocol Overview

The general overview of the communication protocol follows:

1. Authenticator advertises the FIDO U2F service.
2. Client scans for Authenticator advertising the FIDO U2F service.
3. Client performs characteristic discovery on the Authenticator.

4. If not already paired, the Client and Authenticator shall perform Bluetooth low energy technology pairing and create a LTK. Authenticator shall only allow connections from previously bonded Clients without user intervention.
5. Client reads the `u2fControlPointLength` characteristic.
6. Client registers for notifications on the `u2fStatus` characteristic if not already registered.
7. Client writes a request (e.g., an enroll request) into the `u2fControlPoint` characteristic.
8. Authenticator evaluates the request and responds by sending notifications over `u2fStatus` characteristic.
9. The protocol completes when either:
    - The Client unregisters for notifications on the `u2fStatus` characteristic, or:
    - The connection times out and is closed by the Authenticator.

8. Authenticator Advertising Format

When advertising, the Authenticator shall advertise the FIDO U2F service UUID.

When advertising, the Authenticator may include the TxPower value in the advertisement (see [BluetoothXPLAD]).

When advertising in pairing mode, the Authenticator shall either: (1) set the LE Limited Mode bit to zero and the LE General Discoverable bit to one OR (2) set the LE Limited Mode bit to one and the LE General Discoverable bit to zero. When advertising in non-pairing mode, the Authenticator shall set both the LE Limited Mode bit and the LE General Discoverable Mode bit to zero in the Advertising Data Flags.

The advertisement may also carry a device name which is distinctive and user-identifiable. For example, "ACME Key" would be an appropriate name, while "XJS4" would not be.

The Authenticator shall also implement the Generic Access Profile [BluetoothGAP] and Device Information Service [BluetoothDIS], both of which also provide a user friendly name for the device which could be used by the Client. The Bluetooth DIS shall contain the PnP ID field [BluetoothPNPID].

It is not specified when or how often an Authenticator should advertise, instead that flexibility is left to manufacturers.

9. Requests

Clients should make requests by connecting to the Authenticator and performing a write into the `u2fControlPoint` characteristic.

10. Responses

Authenticators should respond to Clients by sending notifications on the `u2fStatus` characteristic.

Some Authenticators might alert users or prompt them to complete the test of user presence (e.g., via sound, light, vibration) Upon receiving any request, the Authenticators shall send KEEPALIVE commands every `kKeepAliveMillis` milliseconds until completing processing the commands. While the Authenticator is processing the request the KEEPALIVE command will contain status `PROCESSING`. If the Authenticator is waiting to complete the Test of User Presence, the KEEPALIVE command will contains status `TUP_NEEDED`. While waiting to complete the Test of User Presence, the Authenticator may alert the user (e.g., by flashing) in order to prompt the user to complete the test of user presence. As soon the Authenticator has completed processing and confirmed user presence, it shall stop sending KEEPALIVE commands, and send the reply.

Upon receiving a KEEPALIVE command, the Client shall assume the Authenticator is still processing the command; the Client shall not resend the command. The Authenticator shall continue sending KEEPALIVE messages at least every kKeepAliveMillis to indicate that it is still handling the request. Until a client-defined timeout occurs, the Client shall not move on to other devices when it receives a KEEPALIVE with TUP_NEEDED status, as it knows this is a device that can satisfy its request.

11. Framing fragmentation

A single request/response sent over Bluetooth low energy technology may be split over multiple writes and notifications, due to the inherent limitations of Bluetooth low energy technology which is not currently meant for large messages. Frames are fragmented in the following way:

A frame is divided into an *initialization fragment* and one or more *continuation fragments*.

An initialization fragment is defined as:

Offset	Length	Mnemonic	Description
0	1	CMD	Command identifier
1	1	HLEN	High part of data length
2	1	LLEN	Low part of data length
3	0 to (maxLen - 3)	DATA	Data

where maxLen is the maximum packet size supported by the characteristic or notification.

In other words, the start of an initialization fragment is indicated by setting the high bit in the first byte. The subsequent two bytes indicate the total length of the frame, in big-endian order. The first maxLen - 3 bytes of data follow.

Continuation fragments are defined as:

Offset	Length	Mnemonic	Description
0	1	SEQ	Packet sequence 0x00..0x7f (high bit always cleared)
1	0 to (maxLen - 1)	DATA	Data

where maxLen is the maximum packet size supported by the characteristic or notification.

In other words, continuation fragments begin with a sequence number, beginning at 0, implicitly with the high bit cleared. The sequence number must wrap around to 0 after reaching the maximum sequence number of 0x7f.

Example for sending a PING command with 40 bytes of data with a maxLen of 20 bytes:

Frame	Bytes
0	[810028] [17 bytes of data]
1	[00] [19 bytes of data]
2	[01] [4 bytes of data]

Example for sending a ping command with 400 bytes of data with a maxLen of 512 bytes:

Frame	Bytes
0	`[810190] [400 bytes of data]`

12. Notifications

A client needs to register for notifications before it can receive them. Bluetooth Core Specification 4.0 or later [BluetoothCORE] forces a device to remember the notification registration status over different connections [BluetoothCCC]. Unless a client explicitly unregisters for notifications, the registration will be automatically restored after restoring the bond. A client may therefor check the notification status upon connection and only register if notifications aren't already registered. Please note that some clients will disable notifications from a power management point of view (see below) and the notification registration is remembered per bond, not per client. A client must not remember the notification status in its own data storage.

13. Implementation Considerations

13.1 Bluetooth pairing: Client considerations

As noted in the Pairing section, a disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an Authenticator is paired to a FIDO Client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an Authenticator. This poses both security and privacy risks to users.

While Client operating system security is partly out of FIDO's scope, further revisions of this specification may propose mitigations for this issue.

13.2 Bluetooth pairing: Authenticator considerations

The method to put the Authenticator into Pairing Mode should be such that it is not easy for the user to do accidentally **especially** if the pairing method is Just Works. For example, the action could be pressing a physically recessed button or pressing multiple buttons. A visible or audible cue that the Authenticator is in Pairing Mode should be considered. As a counter example, a silent, long press of a single non-recessed button is not advised as some users naturally hold buttons down during regular operation.

Note that at times, Authenticators may legitimately receive communication from an unpaired device. For example, a user attempts to use an Authenticator for the first time with a new Client: he turns it on, but forgets to put the Authenticator into pairing mode. In this situation, after connecting to the Authenticator, the Client will notify the user that he needs to pair his Authenticator. The Authenticator should make it easy for the user to do so, e.g., by not requiring the user to wait for a timeout before being able to enable pairing mode.

Some Client platforms (most notably iOS) do not expose the AD Flag LE Limited and General Discoverable Mode bits to applications. For this reason, Authenticators are also strongly recommended to include the Service Data field [BluetoothSD] in the Scan Response. The Service Data field is 3 or more octets long. This allows the Flags field to be extended while using the minimum number of octets within the data packet. All octets that are 0x00 are not transmitted as long as all other octets after that octet are also 0x00 and it is not the first octet after the service UUID. The first 2 bytes contain the FIDO Service UUID, the following bytes are flag bytes.

To help Clients show the correct UX, Authenticators can use the Service Data field to specify whether or not Authenticators will require a Passkey (PIN) during pairing.

Service Data Bit	Meaning (if set)
7	Device is in pairing mode.

6	Device requires Passkey Entry [BluetoothPESTK].

13.3 Handling command completion

It is important for low-power devices to be able to conserve power by shutting down or switching to a lower-power state when they have satisfied a Client's requests. However, the U2F protocol makes this hard as it typically includes more than one command/response. This is especially true if a user has more than one key handle associated with an account or identity, multiple key handles may need to be tried before getting a successful outcome. Furthermore, Clients that fail to send follow up commands in a timely fashion may cause the Authenticator to drain its battery by staying powered up anticipating more commands.

A further consideration is to ensure that a user is not confused about which command she is confirming by completing the test of user presence. That is, if a user performs the test of user presence, that action should perform exactly one operation.

We combine these considerations into the following series of recommendations:

- Upon initial connection to an Authenticator, and upon receipt of a response from an Authenticator, if a Client has more commands to issue, the Client must transmit the next command or fragment within `kMaxCommandTransmitDelayMillis` milliseconds.
- Upon final response from an Authenticator, if the Client decides it has no more commands to send it should indicate this by disabling notifications on the `u2fStatus` characteristic. When the notifications are disabled the Authenticator may enter a low power state or disconnect and shut down.
- Any time the Client wishes to send a U2F APDU, it must have first enabled notifications on the `u2fStatus` characteristic and wait for the ATT acknowledgment to be sure the Authenticator is ready to process APDU messages.
- Upon successful completion of a command which required a test of user presence, e.g. upon a successful authentication or registration command, the Authenticator can assume the Client is satisfied, and may reset its state or power down.
- Upon sending a command response that did not consume a test of user presence, the Authenticator must assume that the Client may wish to initiate another command, and leave the connection open until the Client closes it or until a timeout of at least `kErrorWaitMillis` elapses. Examples of command responses that do not consume user presence include failed authenticate or register commands, as well as get version responses, whether successful or not. After `kErrorWaitMillis` milliseconds have elapsed without further commands from a Client, an Authenticator may reset its state or power down.

Constant	Value
`kMaxCommandTransmitDelayMillis`	1500 milliseconds
`kErrorWaitMillis`	2000 milliseconds
`kKeepAliveMillis`	500 milliseconds

13.4 Data throughput

Bluetooth low energy technology does not have particularly high throughput, this can cause noticeable latency to the user if request/responses are large. Some ways that implementers can reduce latency are:

- Support the maximum MTU size allowable by hardware (up to the 512 bytes max from the Bluetooth specifications).
- Make the attestation certificate as small as possible, do not include unnecessary extensions.

13.5 Advertising

Though the standard doesn't appear to mandate it (in any way that we've found thus far), advertising and device discovery seems to work better when the Authenticators advertise on all 3 advertising channels and not just one.

13.6 Authenticator Address Type

In order to enhance the user's privacy and specifically to guard against tracking, it is recommended that Authenticators use Resolvable Private Addresses (RPAs) instead of static addresses.

14. Bibliography

[BluetoothASSNUM] Bluetooth Assigned Numbers. URL: https://www.bluetooth.org/en-us/specification/assigned-numbers

[BluetoothCORE] Bluetooth Core Specification 4.0. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothDIS] Device Information Service v1.1. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothGAP] Generic Access Profile. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothGAS] Generic Access Profile service. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12. URL: https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.generic_access.xml

[BluetoothCCC] Client Characteristic Configuration. Bluetooth Core Specification 4.0, Volume 3, Part G, Section 3.3.3.3. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothXPLAD] Bluetooth TX Power AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothSD] Bluetooth Service Data AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothPESTK] Passkey Entry. Bluetooth Core Specification 4.0, Volume 3, Part H, Section 2.3.5.3 URL: https://www.bluetooth.com/specifications/adopted-specifications

[BluetoothPNPID] PnP ID. https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.pnp_id.xml URL: https://www.bluetooth.com/specifications/adopted-specifications

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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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

[U2FRawMsgs]
D. Balfanz, *FIDO U2F Raw Message Formats v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.pdf

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

FIDO AppID and Facet Specification

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-appid-and-facets-v1.2-ps-20170411.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.

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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

Table of Contents

1. Notation

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

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

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

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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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: fido-uaf-protocol-v1.1-id-20170202.html
PDF: 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: fido-uaf-overview-v1.1-id-20170202.html
PDF: 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

FIDO U2F Implementation Considerations

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-implementation-considerations-v1.1-v1.2-ps-20170411.html
Previous version:
https://fidoalliance.org/specs/fido-u2f-implementation-considerations-v1.1-RD-20140209.html
Editor:
Dirk Balfanz, Google, Inc.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This document lists a number of considerations for U2F implementers.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

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

Table of Contents

1. Notation

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

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

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

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

U2F 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. Implementation Considerations

Note: Reading the 'FIDO U2F Overview' (see [U2FOverview] in bibliography) is recommended as a background for this document.

2.1 Timing Considerations

U2F Tokens should respond to authentication and registration request as soon as possible to ensure a responsive user interface. In particular, they should not wait for user presence if the request message does not require it. Usually, this means that U2F tokens should respond within 500ms to requests. (FIDO clients, on the other hand, should be coded more defensively, and should wait for at least 3 seconds before giving up on a U2F token.)

Once user presence is detected, U2F tokens should persist the user presence' state for 10 seconds or until an operation which requires user presence is performed, whichever comes first.

2.2 Generation of Key Handles

U2F tokens might not store private key material, and instead might export a wrapped private key as part of the key handle. If a U2F token chooses to do this, then the following must be taken into consideration:

The U2F token should employ a cipher that offers the best possible security on the given hardware. Sometimes, hardware offers better protections against certain attacks for 'weak' ciphers (e.g., 3DES) than against 'strong' ciphers (e.g., AES). Implementers should carefully weigh the pros and cons of different ciphers on the hardware platform that they're implementing on.

Given a particular U2F token and a relying party, the relying party should not be able to tell the difference between a key handle that was issued for a different token, and a key handle that was issued for a different relying party. (The concern is that a site, evil.com, might want to find out whether a given token has been registered for a site embarrassing.com, and would be able to do so if it had key handles from embarrassing.com if it could tell the difference.) The two error conditions ('wrong key handle' and 'wrong origin (but correct key handle)') should not be distinguishable to the relying party, through careful timings or otherwise.

2.3 Secure Key Generation

U2F tokens should follow best practices when generating private keys (i.e., use a recommended PRNG) and use a good source of entropy (which usually serves as input to the PRNG). If no good source of entropy is available on the token, the token should combine whatever entropy there is with the challenge parameter from the request as input into the PRNG.

2.4 Challenge Parameters

The registration and authentication operations require the relying party to pass a challenge parameter to the Javascript API (as part of the SignData and EnrollData parameters - (see [U2FJSAPI] in bibliography). This parameter is the base64-encoding of a byte array chosen by the relying party.

Relying parties should ensure that the challenge parameter has sufficient entropy. In particular, it is recommended that the challenge parameter contains at least 8 random bytes, following the requirements in [SP800-63-1].

2.5 Revocation of Tokens

Since U2F tokens don't have device identifiers, U2F does not prescribe a way to revoke tokens (through a revocation list or similar mechanism). Instead, it is up to individual relying parties to stop honoring authentication responses that come from certain tokens.

Relying parties should give users a mechanism to report lost or stolen tokens. If the token is lost or stolen, then the relying party should stop honoring authentication responses from the token.

2.6 Token Counters

A U2F token must increase a counter each time it performs an authentication operation.

This counter may be 'global' (i.e., the same counter is incremented regardless of the application parameter in Authentication Request message), or per-application (i.e., one counter for each value of application parameter in the Authentication Request message).

U2F token counters should start at 0.

The counter allows relying parties to detect token cloning in certain situations. Relying parties should implement their own remediation strategies if they suspect token cloning due to non-increasing counter values.

2.7 Key Usage

Keys generated during a U2F registration must not be used for any purpose other than U2F authentications. Implementers of hardware and/or software that serves other purposes beyond U2F need to ensure that U2F keys are not used for such other purposes.

2.8 UI Considerations for FIDO Clients

FIDO Clients should implement a user interface that allows the user to get a clear indication of which relying parties are using the FIDO U2F APIs. Such APIs allow relying parties that are in possession of the user's public key to confirm the identity of the user, even if the user has removed their cookies, is using anonymizing onion routing networks, etc. In the case where the FIDO Client is a web browser, the web browser should indicate to the user which page or web origin is creating or exercising U2F keys for the user. The FIDO client might also give the user the ability to allow or deny the use of the U2F APIs for relying parties.

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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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

[U2FJSAPI]
D. Balfanz, A. Birgisson, J. Lang, *FIDO U2F Javascript API v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-javascript-api-v1.2-ps-20170411.pdf

[U2FOverview]
S. Srinivas, D. Balfanz, E. Tiffany, *FIDO U2F Overview v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.pdf

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

A.2 Informative references

[SP800-63-1]
W. Burr, D. Dodson, E. Newton, R. Perlner, W.T. Polk, S. Gupta and E. Nabbus, *NIST Special Publication 800-63-1: Electronic Authentication Guideline*. National Institute of Standards and Technology, December 2011, URL: http://csrc.nist.gov/publications/nistpubs/800-63-1/SP-800-63-1.pdf

FIDO Metadata Statements

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-metadata-statement-v1.2-ps-20170411.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.

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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

Table of Contents

1. Notation

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

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

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

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

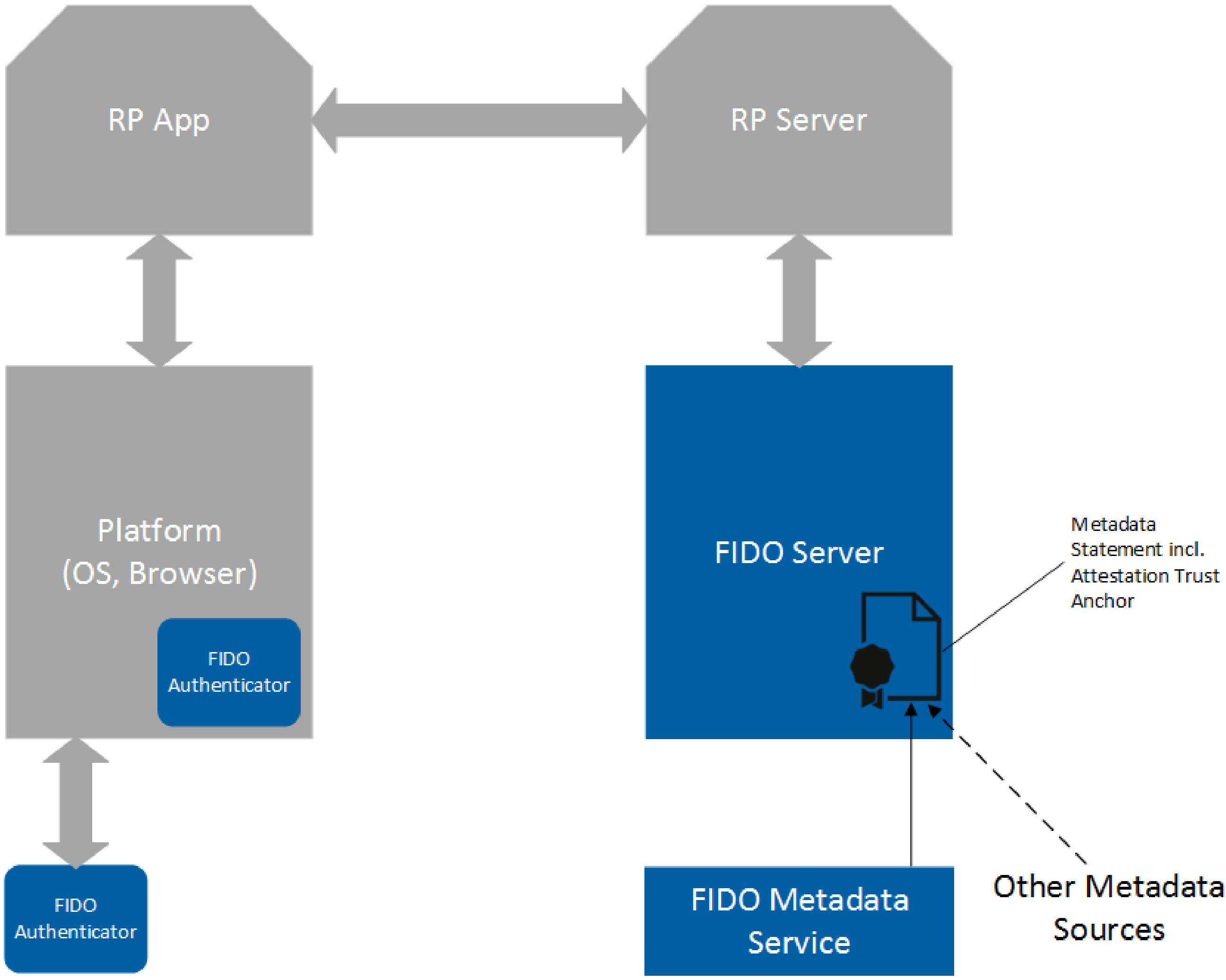


Fig. 1 The FIDO Architecture

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
dtuWBIuffr6oWpV0FPNLhow1751Nm21LvPH3rVtWjfz66Lfql8tX7FRl9YFSXsmSseb9ceOGbk7
MNUcGPg8ZsbMe9rfQUaaV/JMX9sqdzDCSvp0kZHmTZg9x7bLHcMnThbl6eJ+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
WbDiLVvljplLlop6CLXP+2qtvGLIL/1vimISdMBgzSoFZyu6Tqd+jzxgsPaV9BCqee/NjYk6v6lK
9cwiUc/STtf1HDpM3b592y7h3Thx5ozK69HLpYWuAwaqS5cv26q7ceb8efVYaReP3iFU8zj1knSw
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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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-u2f-v1.2-ps-20170411/fido-metadata-service-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-metadata-service-v1.2-ps-20170411.pdf

[FIDORegistry]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Registry of Predefined Values*. FIDO Alliance Implementation Draft. URLs:
HTML: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-registry-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-registry-v1.2-ps-20170411.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: fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: 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: fido-uaf-protocol-v1.1-id-20170202.html
PDF: 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: fido-uaf-reg-v1.1-id-20170202.html
PDF: fido-uaf-reg-v1.1-id-20170202.pdf

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

[iPhonePasscodes]
Most Common iPhone Passcodes, Daniel Amitay (Accessed July 11, 2014) URL: http://danielamitay.com/blog/2011/6/13/most-common-iphone-passcodes

FIDO Metadata Service

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-metadata-service-v1.2-ps-20170411.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.

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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

Table of Contents

1. Notation

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

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

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

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.

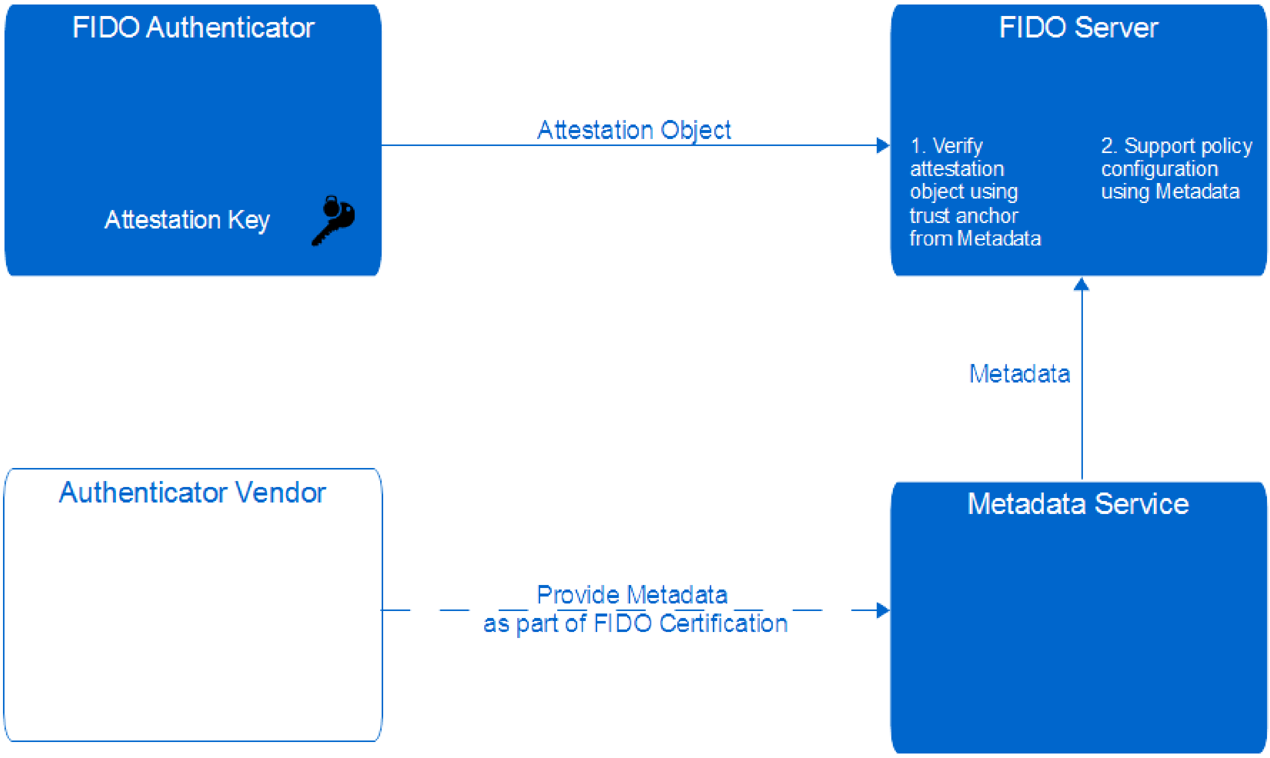


Fig. 1 FIDO Metadata Service Architecture Overview

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.

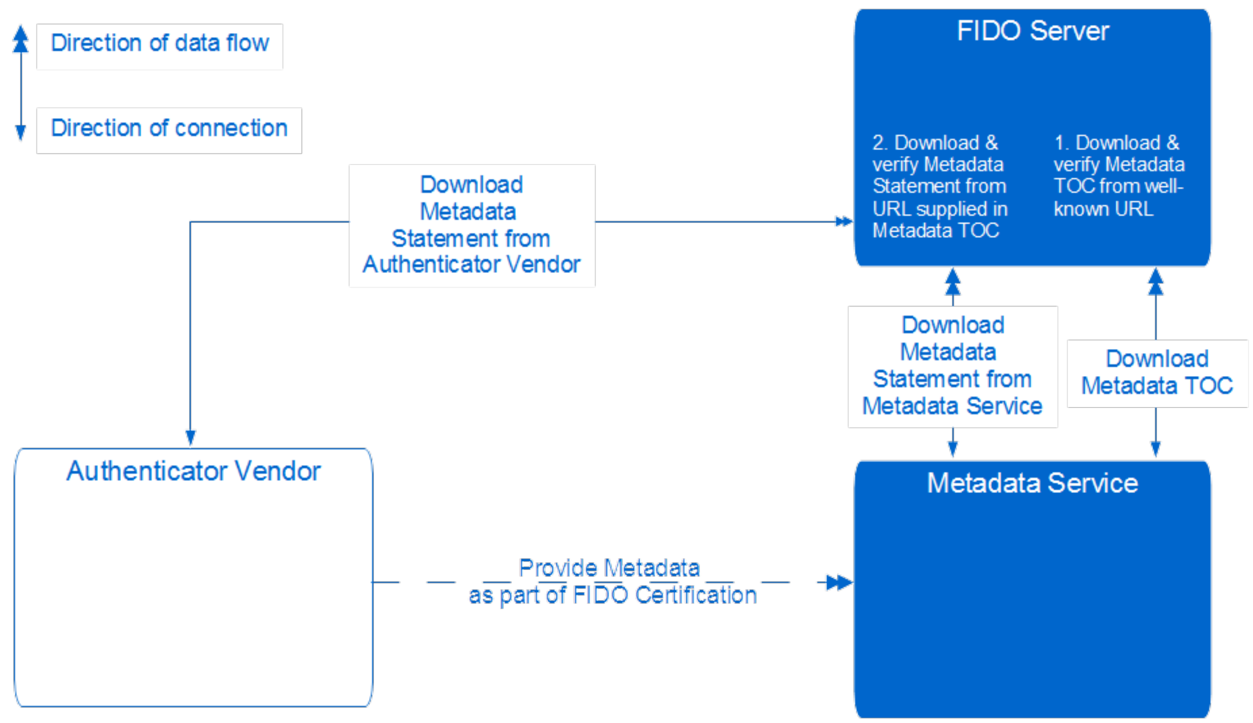


Fig. 2 FIDO Metadata Service Architecture

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
TU1JQ1BUQ0NBZU9nQXdJQkFnSUpBT3VleHZVM095MndNQW9HQ0NxR1NNNDlCQU1DTUhzeElEQWVC
```

```
Z05WQkFNTQ0KRjFOaGJYQnNaU0JCZEhSbGMzUmhkR2x2YmlCU2IyOTBNUll3RkFZRFZRUUtEQTFH
U1VSUElFRnNiR2xoYm1ObA0KTVJFd0R3WURWUVFMREFoVlFVWWdWRmRITERFU01CQUdBMVVFQnd3
SlVHRnNieUJCYkhSdk1Rc3dDUVlEVlFRSQ0KREFKRFFURUxNQWtHQTFVRUJoTUNWVk13SGhjTk1U
UXdOakU0TVRNek16TXlXaGNOTkRFeE1UQXpNVE16TXpNeQ0KV2pCN01TQXdIZ1lEVlFRRERCZFRZ
VzF3YkdVZ1FYUjBaWE4wWVhScGIyNGdVbTl2ZERFV01CUUdBMVVFQ2d3Tg0KUmtsRVR5QkJiR3hw
WVc1alpURVJNQThHQTFVRUN3d0lWVUZHSUZSWFJ5d3hFakFRQmdOVkJBY01DVkJoYkc4Zw0KUVd4
MGJ6RUxNQWtHQTFVRUNBd0NRMEV4Q3pBSkJnTlZCQVlUQWxWVE1Ga3dFd1lIS29aSXpqMENBUVlJ
S29aSQ0KemowREFRY0RRZ0FFSDhodjJEMEhYYTU5L0JtcFE3UlplaEwvRk1HekZkMVFCZzl2QVVw
T1ozYWpudVE5NFBSNw0KYU16SDMzblVTQnI4ZkhZRHJxT0JiNThweEdxSEpSeVgvNk5RTUU0d0hR
WURWUjBPQkJZRUZQb0hBM0NMaHhGYg0KQzBJdDd6RTR3OGhrNUVKL01COEdBMVVkSXdRWU1CYUFG
UG9IQTNDTGh4RmJDMEl0N3pFNHc4aGs1RUovTUF3Rw0KQTFVZEV3UUZNQU1CQWY4d0NnWUlLb1pJ
emowRUF3SURTQUF3UlFJaEFKMDZRU1h0OWloSWJFS1lLSWpzUGtyaQ0KVmRMSWd0ZnNiRFN1N0Vy
SmZ6cjRBaUJxb1lDWmYwK3pJNTVhUWVBSGpJekE5WG02M3JydUF4Qlo5cHM5ejJYTg0KbFE9PSIs
DQogICJEZXNjcmlwdGlvbiI6ICJGSURPIEFsbGlhbmNlIFNhbXBsZSBVQUYgQXV0aGVudGljYXRv
ciIsDQogICJVc2VyVmVyaWZpY2F0aW9uTWV0aG9kcyI6IDIsDQogICJWYWxpZEF0dGFjaG1lbnRU
eXBlcyI6IDEsDQogICJLZXlQcm90ZWN0aW9uIjogNiwNCiAgIk1hdGNoZXJQcm90ZWN0aW9uIjog
MiwNCiAgIlNlY3VyZURpc3BsYXkiOiA0LA0KICAiU2VjdXJlRGlzcGxheUNvbnRlbnRUeXBlcyI6
IFsiaW1hZ2UvcG5nIl0sDQogICJTZWN1cmVEaXNwbGF5UE5HQ2hhcmFjdGVyaXN0aWNzIjogW1sw
LDAsMSw2NCwwLDAsMSwyMjQsMTYsMiwwLDAsMF1dLA0KICAiaXNTZWNvbmRGYWN0b3JPbmx5Ijog
ImZhbHNlIiwNCiAgIkljb24iOiAiZGF0YTppbWFnZS9wbmc7YmFzZTY0LGlWQk9SdzBLR2dvQUFB
QU5TVWhFVWdBQUFFOEFBQUF2Q0FZQUFBQ2l3SmZjQUFBQUFYTlNSMElBcnM0YzZRQUFBQVJuUVUx
QkFBQ3gNCmp3djhZUVVBQUFBSmNFaFpjd0FBRHNNQUFBN0RBY2R2cUdRQUFBYWhTVVJCVkdoRDda
cjVieFJsR01mOUt6N6VEI4QU0vWUVVoRTJXN3ANClFaY1dLS0JjbFNwSEFUbEVMMQVJFN2tORUNQTNG
a1dLMENLS1NNDRklzS0JjZ1ZDZRFdHTkVTZEFaaWR3Z2dnSkJUUmlNaEZjLzR3eTNNNgNCjg4NHp10U5k
bG5HVGZaa2lAybjNuTysrODg5MzNmdmVCQngrUHFVekdprVFV2QmJMbXBXVRFd2QlRJbXBjQ1NadlhM
Q2RYOVIwNVNrMTkNCmJiNWF0ZjU0ZHHKy9lckE1NDFxNDdhUDFMMTFhOVNJeVZaOVVkGSWk4ZDVr
R1lRzaTMwTkZpN2N2UpQTXdiZHlzMmVyVTJYENClVkeTgrWmNhTm5nTm1HaW1PHlYTjNSVWQz
YTE4bkYwZlVsVnsb3ZaKzBDVHpXcGQyVmorZU9tMWJFZXk2RHg0aTVwVU1HV3ZlbzUwnNEyMjcNCmR0
dVdCSXVmZnI2b1dwVjBGUE5MaG93MTc1MU5tMjFMd1BdBIM3JWdFdqZno2NkxmcWw4dFg3RlJsOVlG
U1hzbVNzZWI5Y2VPR2JZazcNCk1OVWNHUGc4WnNiTWU5cmZSVWFhVi9KTVg5c3FkekRU3ZwMGta
SG1UWmc1eDdiTEhjTW5UaGIxNmVKNmVjRWsxVjVPSUZuazc0dWhVXRh
dGRXS2ZYblJROTlCajkxUjVPSUZuazU0ak4wbWtVaXFsTzNYRFcrTWwrOThtS0I2dFc3cldwWmNQ
YysNCjB6ZR0THJZbFVjODZFNmVHRGpJTXViVnBjdXNlYXJmZ0lZR1JrNmJyaFpWci9KY0h6b29M
NzU1MGplZExxFeG9wV2NBcGkyWlVxaHUNCjdKTHZyVnNRVTgxemt6T1BlZW1NUll2VnVRc1g3UGJp
RFFZNUp2Wm9uZnRLKzFWWThIOXV0eDUzMGgwb2Iram1SWXFqNm91YVl2RWUNCm5XL1dsWWpwOGN3
Yk1tNjgydFB3cVcxUjR0ai8yU0gxM0lSSllsNG1vWnZZcGlTcURyN2RYdFFIeGEvUEszLytCV3NL
MWRUZ0h1NlYNCjh0UUozYndGa3dRnJVT1E1MHMxcjNsZXZtCQmF3N0s4bEVLNXF6
a1llYXJrOUE4cDdQM0d6REsrbmQzRFFvdys2VUMNCjhTVk44Mml1djM4aW03TnRhWHRWMUNWcTZS
Z3c0cGtzbWJkaTNidJEZTdZZmFCQnhjcWZ2cVByVWpGUU5UUTIybGZkVVZZWVDY4clQNCkpLRjVE
blNtVWpnZHFnNG1TUzlwbXNmREpSM0c2VG9IMGlXOWFWN0xxXTEhZWEtsbFREdDBMVEF0a1lJYWFt
cDFRalZ2eUdVdVeFYNCmRKMERPVlhTbStiMXFSeHBsODRjZGRkZ2ZZMUxwMDFWcGRZMUdZTUxxd01U8vZDY5dHNvZDB2Y3pVo
R3JlOXhlOG8rZnBMUjFjR2hOVEQ2WjU3QzlLTVdYZVdaKZE8NClo5NGJJOW9xZDFST25N3FJVFR6
SGltTXFpdmJPM2cwRGRWeWszV1FCaEJ6dFNlTVRPbm4ATzNhY2Y1M2ZrMmRhRmRLWVhMYzBWS2NEUN
CnJkcmxpQnFwODljSmNzL203VHZzMHJhdmTjRiMGtHb1puTTFVNVdDUlBjbWFNa5UURGbXXZVeCtP
NWdTcWViVjFtK3pTdVlOVmhxN1QNCldiRlGMVnZsanBsTGxxvcDZDTFhQzJxdhZHZHEFlMLzF2aW1J
U2RNQmd6U29GWnl1NlRxZCtqenhnc1BhVjlsCQ3FlZS9OallrNnY2YnEsNCljlijd2lVYy9TVHRmMUhE
cE0zYjU5Mnk3aDNUaHg1b3pLNjlITHHBZV3VBd2FxUzVjdjI2cTdjZWI4ZWZWWWFSZVAzaUZVOHpq
MWtuU3cNClpYSE1tbkNqWTBPZ2FsbzdVUWZTQ00zcVFRcjJIL1hGUUdzdzc1aDRNDVZbDkxQnl1Q2Vw
NG1vWm9IKzFmRzN4RDR0VDd4OGt3ZWo4bncNCmI5ZXYYYNlYwQjZkKzdINHpLdnVkQUcgMzdGFnF5
ek9IZEpuuSEV1em1Ycyk9XanhPYnZOTwJ2N25oeXdzWDJoVnnXdEM4Kz4Q0YUXlYXANCkU3cDVzS1pp
MEEyQVFSVjVudlI0RS0xSm1rYjYxa0Zwc1VlueEJnbWQvNFlYVUVBdbXxQxOEhEQzdzUkhmdG1ldFVz
bWhWMHJuL0ZNWDINCjMyQnFrNEJGbkR4NlpwMVdNXXVzJ1ZmYwSWJDNDdxZXh4bVZqOVF1ZFlqdFhC
azM3RDdaSFlYRkNrMXlhQ3RjdE5pVDB0eVRrWXhLM1ZuQ2JkJrbkNClhHZndOUTB0VmxhdmhWTTFsTlQwRkJuVlZ
WWNEi9TTUREZE1NSGxQVTJWUFlXUkZPRGhCY0hOWVJrZG1aak13ZVc1b2JFcG5UVFV4UTFVMmRr
NDVSWHBuYm5CMlNFSkdWWGtOQ21sWcmFlUGl3SjUzREVZMVdsUmFibTl0UlU1bk9EVnJUbFZrTW05
S2FUSlhjSEkwVDIxdGEyWk9OSGcwZWtobWFWWkdZemhFZGpoTwplblZvVG5GUGFXUnBiRWQyUVRaRVIzVU5DbVZhZDA4M09FRkJVVzQyWTJsRmF6WXJjbmMxVm1OMmFuWnhUa1JaVUU5dlNWV3NZVXRUYUhKNFFYVllUR3hyU0RSaFdYVkhaazFaUkdNeE1GZEdOVlJoTXpGb1VFcFBabU5WYUhJTkNsVXZTbXhKVG1rMllzZ2xiRkpaWkVKd2J6WXJLMWxtYW5nMk1XeEhUbVpTYlRSTlJEVnlTakZxTTBadlIwaHVha1JUUWs1aGNsbFZaMDFNZVUxemVrdHdZamQwV0hCdlNHWlFjemdOQ21nelYzQXhUSHBPWms1ck5UUlllRU14ZDBSSFZXMWZlbGhaZFdab05ub3ZZMHQwVm00MFJVSjRZVGxXVVVkRWxseU0weVVVMVNha2hGUzJ0ck4zcGhSa3RaVVVFeWFFZFJWVEVOQ25vck9EVk9SbGR3V0VSeWEzb3pkbmd4TUVkeGVGRTJRbnBsVG1KdlFtczFiamhyTkc1bFlsSm9LMnN4YUZkbWVGUkdNRVF4UlhsWFZYTTFibllyWkdkUmNVdGhlSHAxUTJSRk1Ha05DbnNJYkRBeVRsRTRZV2d3YlZoeU1USk1ZVE50TUdZNWQybHJPU3QzVEU1VVRWa3ZPRFpOVUc4NGVXa3pNVTltZUcxVU5sQlhiM0ZIT1N0RVduVnJXVzVoTlRadFUxcDBOVmRYVTNrTkNqVnhWa0V4Y25kVmVVcHhXRUZzYm5wcmFXRnBMMmRJVTBRM1VtdFVlV2xvYjJkQlFVRkJRa3BTVlRWRmNtdEtaMmRuUFQwaUxBMEtJQ0FpUVhOelpYSjBhVzl1U2NoZW1sSWpvZ0lsVkJSbFl4VkV4V0l
```


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-u2f-v1.2-ps-20170411/fido-metadata-statements.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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: fido-uaf-protocol-v1.1-id-20170202.html
PDF: fido-uaf-protocol-v1.1-id-20170202.pdf

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

FIDO Registry of Predefined Values

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-registry-v1.2-ps-20170411.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

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO

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

Table of Contents

1. Notation

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

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

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

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 other `ATTACHMENT_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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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-u2f-v1.2-ps-20170411/fido-security-ref-v1.2-ps-20170411.html

PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-security-ref-v1.2-ps-20170411.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: fido-uaf-asm-api-v1.1-id-20170202.html
PDF: 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: fido-uaf-client-api-transport-v1.1-id-20170202.html
PDF: 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: fido-uaf-authnr-cmds-v1.1-id-20170202.html
PDF: 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: fido-uaf-protocol-v1.1-id-20170202.html
PDF: fido-uaf-protocol-v1.1-id-20170202.pdf

FIDO ECDAA Algorithm

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-ecdaa-algorithm-v1.2-ps-20170411.html
Editor:
Rolf Lindemann, Nok Nok Labs, Inc.
Contributors:
Jan Camenisch, IBM
Manu Drijvers, IBM
Alec Edgington, Trustonic
Anja Lehmann, IBM
Rainer Urian, Infineon

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

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

Table of Contents

1. Notation

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

String literals are enclosed in “”, e.g. “ED256”.

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

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 . 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 , see section 4.1 Supported Curves for ECDAA. Each `ECPoint2` is represented by a pair `(a, b)` of elements of .

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 , and , of sufficiently large prime order
2. Two generators and , such that and
3. A bilinear pairing . We propose the use of "ate" pairing (see [BarNae-2006]). For example source code on this topic, see BNPairings.
4. Hash function with .
5. are installed in all authenticators implementing FIDO ECDAA attestation.

Definition of , Pairings and hash function

See section 4.1 Supported Curves for ECDAA.

3.3 Issuer Specific ECDAA Parameters

Issuer Parameters parI

1. Randomly generated issuer private key with .
2. Issuer public key , with and .
3. A proof that the issuer key was correctly computed
   1. BigInteger
   2. BigInteger

3. ECPoint2
4. ECPoint2
5. BigInteger
6. BigInteger
7. BigInteger

4.

Whenever a party uses ipk for the first time, it must first verify that it was correctly generated:

NOTE

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 an authenticator 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 their authenticator 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 and sends via the ASM to the authenticator.
3. The authenticator chooses and stores the ECDAA private key BigInteger
4. The authenticator computes its ECDAA public key ECPoint
5. The authenticator proves knowledge of as follows
    1. BigInteger
    2. ECPoint
    3. BigInteger
    4. BigInteger
6. The authenticator sends via the ASM to the issuer
7. The issuer verifies that the authenticator is "authentic" and that 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 and verifies (check proof-of-possession of private key).

    > NOTE

9. The issuer creates credential as follows
    1. BigInteger
    2. ECPoint
    3. ECPoint
    4. ECPoint
    5. ECPoint
10. The issuer proves that it computed this credential correctly:
    1. BigInteger
    2. ECPoint

3. ECPoint

4. BigInteger

5. BigInteger

11. The issuer sends to the authenticator.

12. The authenticator checks that and

13. The authenticator checks

> NOTE

14. The authenticator checks

> NOTE
>
> ;

15. and the authenticator checks

> NOTE
>
> ;

16. The authenticator stores credential

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 the authenticator. 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 and sends to the ASM.
3. The ASM forwards to the authenticator
4. The authenticator chooses and stores the private key BigInteger
5. The authenticator computes its ECDAA public key ECPoint
6. The authenticator proves knowledge of as follows
    1. BigInteger

2. ECPoint
3. BigInteger
4. BigInteger

7. The authenticator sends to the ASM, who forwards it to the issuer.
8. The issuer verifies that the authenticator is "authentic" and that 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 and verifies .
10. The issuer creates credential as follows
    1. BigInteger
    2. ECPoint
    3. ECPoint
    4. ECPoint
    5. ECPoint
11. The issuer proves that it computed this credential correctly:
    1. BigInteger
    2. ECPoint
    3. ECPoint
    4. BigInteger
    5. BigInteger
12. The issuer sends to the ASM. The issuer authenticates such that the authenticator can verify they were created by the issuer.
13. The ASM checks that and
14. The ASM checks
15. The ASM checks
16. and the ASM checks that
17. The ASM stores and sends to the authenticator
18. The authenticator checks and , and verifies that were sent by the issuer.
19. The authenticator checks
20. The authenticator stores and ignores further join requests.

> NOTE
>
> These values belong to the ECDAA secret key . 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 and sends 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 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 , 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
   5. call TPM2_Sign(, ctr), returning .
   6. sends EK-C, `TPMT_PUBLIC` (including in field `unique`), 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
   5. checks
   6. generates the ECDAA credential as follows
      1. BigInteger
      2. ECPoint
      3. ECPoint
      4. ECPoint
      5. ECPoint
   7. proves that it computed this credential correctly:
      1. BigInteger
      2. ECPoint
      3. ECPoint
      4. BigInteger
      5. BigInteger
   8. generates a *secret* (derived from a *seed*) and wraps the credential 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 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 using the *secret* received from the TPM.
   2. checks that and
   3. checks
   4. checks and
   5. stores

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
3. ECPoint ;
4. ECPoint ;
5. ECPoint ;
6. ECPoint ;
7. BigInteger

8. ECPoint
9. BigInteger
10. BigInteger
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
   2. ECPoint ;
   3. ECPoint ;
   4. ECPoint ;
   5. ECPoint ;
2. The ASM sends to the authenticator
3. The authenticator performs the following tasks
   1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here
   2. ECPoint
   3. ECPoint
   4. BigInteger
   5. ECPoint
   6. BigInteger
   7. BigInteger
   8. Send 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
2. ECPoint ;
3. ECPoint ;
4. ECPoint ;
5. ECPoint ;

2. The ASM calls TPM2_Commit() with set to and empty buffers. The ASM receives the result values and ctr. and are empty since 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 as qualifying data (is returned by step 2). The ASM receives signature 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 (global constant)
- : System parameter generator of group (global constant)
- signature:
- 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 from trusted source
2. Check that , , and .
3. ; fail if not equal

> NOTE
>
> and

4. ; fail if not equal

NOTE

5. ; fail if not equal

NOTE

6. for (all sk' on RogueList) do if 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 : over with .

Definition of G2

G2 is the p-torsion subgroup of where E' is a sextic twist of E. With E' : .

An element of is represented by a pair (a,b) where a + bX is an element of .
We use angle brackets 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 .

Definition of GT

GT is an order-p subgroup of .

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 [ISO15946-5] and a related order of the group [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.
  - = 3
  - _256 = (x=1, y=2)
  - = (a=3, b=3)
  - _256 = (x,y), with
    - _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)
    - _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.
  - = 257
  - _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)
  - = (a=771, b=1542)
  - _638 = (x, y), with
    - _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)

    - _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.
  - = 3
  - _DSD_P256 = (1, 2)
  - = (a=3, b=6)
  - _DSD_P256 = (x, y), with
    - _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)
    - _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.
  - = 3
  - _ISO_P512 = (x=1,y=2)
  - = (a=3, b=3)
  - _ISO_P512 = (x, y), with
    - _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)
    - _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: .

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 • , with computed by the signer. • KRD is the the entire to-be-signed object (e.g. TAG_UAFV1_KRD in the case of FIDO UAF). • , with computed by the signer and computed in the ECDAA-Join
		The s value, s=r + c * sk (mod p), as returned by AuthnrEcdaaSign encoded as byte string according to BigNumberToB. Where

Value	Length (in Bytes)	Description
UINT8[] ECDAA_Signature_s	N	• 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	; 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 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: , see 3.4.1 ECDAA-Join Algorithm. • Where and p are system values, injected into the authenticator and is a random number computed by the ECDAA-Issuer on Join.
UINT8[] ECDAA_Signature_S	2*N+1	; 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: , see 3.4.1 ECDAA-Join Algorithm.
UINT8[] ECDAA_Signature_T	2*N+1	; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where • , provided by the ECDAA-Issuer on Join • 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	; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where 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 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 .

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=( ) 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"
	FIDO ECDAA algorithm based on TPM_ECC_BN_P256 [TPMv2-Part4] curve

Algorithm Description	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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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 Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-security-ref-v1.2-ps-20170411.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.

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED "AS IS" AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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

Table of Contents

1. Notation

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

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

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

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

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.

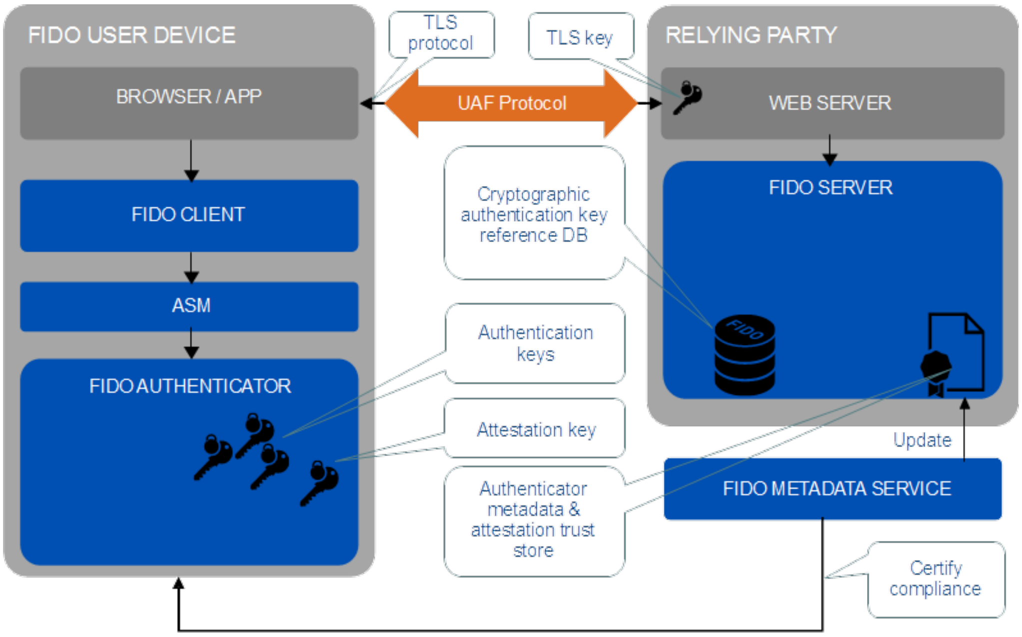


Fig. 1 FIDO Reference Architecture

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.

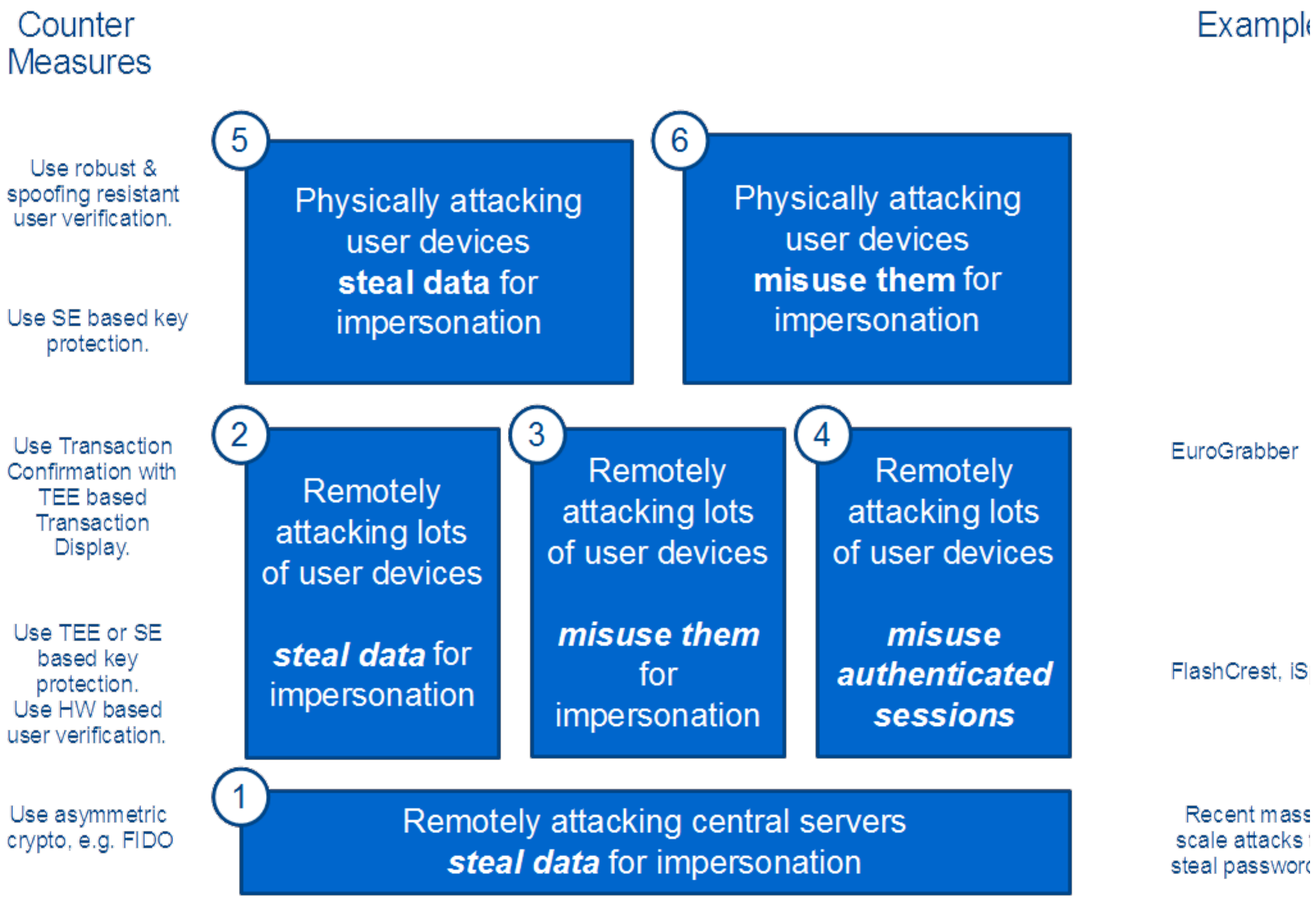


Fig. 2 Attack Classes

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.

NOTE

At this time we are not explicitly covering physical attacks on the authenticator, which might lead to reduced security if the legitimate user uses the authenticator *after* the attacker having physical access to it.

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

Security Goal	Supporting Security Measures
[SG-10] DoS Resistance	[SM-8] Protocol Nonces
[SG-11] Forgery Resistance	[SM-7] Secure Channel with Server Authentication [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	SG-4,

T-1.2.6	Fingerprinting Authenticators	Violates
	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.	SG7, SG-8

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	SG-1

T-1.4.1	Malicious Authenticator	Violates
	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.	

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. **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

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 **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].	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.	

T-4.1.2	Malicious Authenticator HW	Violates
	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 verifiability of their vendors and supply chain.	SA-1

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

	Physical U2F Authenticator Attack	Violates
T-5.1.3	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 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-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.html
PDF: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.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: fido-uaf-protocol-v1.1-id-20170202.html
PDF: fido-uaf-protocol-v1.1-id-20170202.pdf

FIDO Technical Glossary

FIDO Alliance Proposed Standard 11 April 2017

This version:
https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-glossary-v1.2-ps-20170411.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

The English version of this specification is the only normative version. Non-normative translations may also be available.

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 Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third

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

Table of Contents

1. Notation

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

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

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

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 theFIDO 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 Presence Check

The User Presence check in the authenticator verifies that some user is present at the authenticator and agrees with a generic authentication operation.

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