Abstract

A FIDO 2.0 signature proves possession of a private key of a FIDO 2.0 credential and asserts contextual information about the client and authenticator that generated it. This document describes the data structures representing these assertions, how they are serialized to byte streams for signing with an authenticator, and the representation of the resulting signature and its associated data.
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 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. Conformance
   1.1 Dependencies
2. Overview
3. Client data
   3.1 Dictionary ClientData Members
4. Authenticator data
5. Generating a signature
6. Client encoding of assertions
   6.1 Attributes
7. FIDO Extensions
   7.1 Extension identifiers
   7.2 Defining extensions
       7.2.1 Extending request parameters
       7.2.2 Extending client processing
       7.2.3 Extending authenticator processing with signature extensions
       7.2.4 Example extension
   7.3 Standard extensions
       7.3.1 Transaction authorization
       7.3.2 Authenticator Selection Extension
8. IANA Considerations
   A. References
       A.1 Normative references
       A.2 Informative references

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].

The term Base64url Encoding refers to the base64 encoding using the URL- and filename-safe character set defined in Section 5 of [RFC4648], with all trailing '=' characters omitted (as permitted by Section 3.2) and without the inclusion of any line...
1.1 Dependencies

This specification relies on several other underlying specifications.

**HTML5**
The concept of **origin** and the **Window** interface are defined in [HTML5].

**Web IDL**
Many of the interface definitions and all of the IDL in this specification depend on [WebIDL-ED]. This updated version of the Web IDL standard adds support for **Promises**, which are now the preferred mechanism for asynchronous interaction in all new web APIs.

**DOM**
**DOMException** and the DOMException values used in this specification are defined in [DOM4].

**FIDO External Authenticator Protocol**
This specification references methods for the client to communicate with FIDO 2.0 authenticators. These methods are specified in [FIDOEAP].

**Web Cryptography API**
The **AlgorithmIdentifier** type and the method for normalizing an algorithm are defined in [WebCrypto].

2. Overview

FIDO 2.0 signatures are bound to various contextual data. These data are observed, and added at different levels of the stack as a signature request passes from the server to the authenticator. In verifying a signature, the server checks these bindings against expected values.

The components of a system using FIDO 2.0 can be divided into three layers:

1. The relying party (RP), which uses the FIDO 2.0 services. The relying party may, for example, be a web-application running in a browser, or a native application that runs directly on the OS platform.
2. The client platform, which consists of the user’s OS and device used to host the RP’s client-side app. For web-applications, the browser also belongs to this layer.
3. The authenticator itself, which provides key management and cryptographic signatures.

When the RP client-side application is a web-application, the interface between 1 and 2 is the FIDO 2.0 Web API [FIDOWebApi], but is platform specific for native applications. In cases where the authenticator is not tightly integrated with the platform, the interface between 2 and 3 is the FIDO External Authenticator Protocol [FIDOEAP]. This document defines the common signature format shared by all layers. This includes how the different contextual bindings are encoded, signed over, and delivered to the RP.

The goals of this design can be summarized as follows.

- The scheme for generating signatures should accommodate cases where the link between the client platform and authenticator is very limited, in bandwidth and/or latency. Examples include Bluetooth Low Energy and Near-Field Communication.
- The data processed by the authenticator should be small and easy to interpret in low-level code. In particular, authenticators should not have to parse high-level encodings such as JSON.
- Both the client platform and the authenticator should have the flexibility to add contextual bindings as needed.
The design aims to reuse as much as possible of existing encoding formats in order to aid adoption and implementation.

The contextual bindings are divided in two: Those added by the RP or the client platform, referred to as \textbf{client data}; and those added by the authenticator, referred to as the \textbf{authenticator data}. The client data must be signed over, but an authenticator is otherwise not interested in its contents. To save bandwidth and processing requirements on the authenticator, the client platform hashes the client data and sends only the result to the authenticator. The authenticator signs over the combination of this hash, and its own authenticator data.

3. Client data

The client data represents the contextual bindings of both the RP and the client platform. It is a key-value mapping with string-valued keys. Values may be any type that has a valid encoding in JSON. It \textit{must} contain at least the following key-value pairs.

\begin{verbatim}
WebIDL
dictionary ClientData {
  DOMString challenge;
  DOMString facet;
  JsonWebKey tokenBinding;
  optional object extensions;
  DOMString hashAlg;
};
\end{verbatim}

3.1 Dictionary ClientData Members

- **challenge** of type \texttt{DOMString}
  A base64url-encoded challenge provided by the RP.

- **facet** of type \texttt{DOMString}
  A string value describing the RP identifier facet \texttt{[FIDOPlatformApiReqs]}. When the RP client-side app is a website, this is its fully qualified web origin, using the syntax defined by \texttt{[RFC6454]}. When the client-side app is a native application, this string is a platform specific identifier.

- **tokenBinding** of type \texttt{JsonWebKey}
  A JsonWebKey object \texttt{[JWK]} describing the public key that this client uses for the Token Binding protocol when communicating with the Relying Party. This can be omitted if no Token Binding has been negotiated between the client and the Relying Party.

- **extensions** of type \texttt{optional object}
  An object with extension-provided authenticator data. Signature extensions are detailed in Section \texttt{7. FIDO Extensions}.

- **hashAlg** of type \texttt{DOMString}
  The hash algorithm used to compute clientDataHash (see \texttt{5. Generating a signature}). Use "S256" for SHA-256, "S384" for SHA384, "S512" for SHA512, and "SM3" for SM3 (see \texttt{8. IANA Considerations}).

The client data \textit{may} contain additional properties.

Before making a request to an authenticator, the client platform layer \textit{shall} perform the following steps.

1. Let clientDataJSON be the UTF-8 encoded JSON serialization \texttt{[RFC7159]} of clientData.
2. Let \( \text{clientDataHash} \) be the hash (computed using \( \text{hashAlg} \)) of \( \text{clientDataJSON} \), as an array.

The \( \text{clientDataHash} \) value is incorporated into a signature by a FIDO authenticator (see 5. Generating a signature). It is delivered to integrated authenticators in platform specific manners, and to external authenticators as a part of a signature request as specified by the External Authenticator Protocol [FIDO EAP]. The client platform should also preserve the exact \( \text{encodedClientData} \) string used to create it, for embedding in a signature object sent back to the RP (see 5. Generating a signature). This is necessary since multiple JSON encodings of the same client data are possible.

The hash algorithm \( \text{hashAlg} \) used to compute \( \text{clientDataHash} \) is included in the \( \text{clientData} \) object. This way it is available to the RP and it is also hashed over when computing \( \text{clientDataHash} \) and hence anchored in the signature itself.

### 4. Authenticator data

The authenticator data encodes contextual bindings made by the authenticator itself. The authenticator data has a compact but extensible encoding. This is desired since authenticators can be devices with limited capabilities and low power requirements, with much simpler software stacks than the client platform components.

The encoding of authenticator data is a byte array \( \text{authenticatorData} \) of 5 bytes or more, as follows.

<table>
<thead>
<tr>
<th>Byte Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flags (bit 0 is the least significant bit):</td>
</tr>
<tr>
<td></td>
<td>- Bit 0: Test of User Presence (( \text{TUP} )) result.</td>
</tr>
<tr>
<td></td>
<td>- Bits 1-6: Reserved for future use (( \text{RFU} )).</td>
</tr>
<tr>
<td></td>
<td>- Bit 7: Extension data included (( \text{ED} )). Indicates if the authenticator data has extensions.</td>
</tr>
<tr>
<td>1-4</td>
<td>Signature counter (( \text{signCount} )), 32-bit unsigned big-endian integer.</td>
</tr>
<tr>
<td>5-</td>
<td>Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See 7. FIDO Extensions for details.</td>
</tr>
</tbody>
</table>

The \( \text{TUP} \) flag shall be set if and only if the authenticator detected a user through an authenticator specific gesture. The \( \text{RFU} \) bits in the flags byte shall be set to zero.

If the authenticator does not include any extension data, it must set the \( \text{ED} \) flag in the first byte to zero, and to one if extension data is included.

**Fig. 1 authenticatorData layout**, shows a visual representation of the authenticator data structure.

![authenticatorData layout](image)
5. Generating a signature

A raw cryptographic signature must assert the integrity of both the client data and the authenticator data. Thus, an authenticator shall compute a signature over the concatenation of the authenticatorData and the clientDataHash.

The authenticator must return both the authenticatorData and the raw signature back to the client.

6. Client encoding of assertions

The client platform uses an authenticator assertion to construct the final FIDO assertion object returned to the RP as follows.

```webidl
interface FIDOAssertion {
  attribute Credential credential;
  attribute DOMString clientData;
  attribute DOMString authenticatorData;
  attribute DOMString signature;
};
```

6.1 Attributes

credential of type Credential,
An object representing which credential was used to generate an assertion.
clientData of type DOMString,
A base64url encoding of clientDataJSON. (See 3. Client data)

authenticatorData of type DOMString,
A base64url encoding of authenticatorData. (See 4. Authenticator data)

signature of type DOMString,
A base64url encoding of the raw signature returned from the authenticator.
(See 5. Generating a signature)

This assertion is delivered to the RP in either a platform specific manner, or in the case of web applications, according to the FIDO Web API [FIDOWebApi]. It contains all the information that the RP's FIDO server requires to reconstruct the signature base string, as well as to decode and validate the bindings of both the client- and authenticator data.

7. FIDO Extensions

The mechanism for generating FIDO 2.0 credentials, as well as requesting and generating FIDO 2.0 assertions, as defined in [FIDOWebApi] and in this document, can be extended to suit particular use cases. Each case is addressed by defining a registration extension and/or a signature extension. Extensions can define additions to the following steps and data:

- makeCredential request parameters (see [FIDOWebApi])) for registration extension.
- getAssertion request parameters (see [FIDOWebApi]) for signature extensions.
- Client processing, and the clientData structure, for registration extensions and signature extensions.
- Authenticator processing, and the authenticatorData structure, for signature extensions.

When requesting an assertion for a FIDO 2.0 credential, an RP can list a set of extensions to be used, if they are supported by the client and/or the authenticator. It sends the request parameters for each extension in the getAssertion call (for signature extensions) or makeCredential call (for registration extensions) to the client platform. The client platform performs additional processing for each extension that it supports, and augments clientData as required by the extension. For extensions that the client platform does not support, it passes the request parameters on to the authenticator when possible (criteria defined below). This allows one to define extensions that affect the authenticator only.

Similarly, the authenticator performs additional processing for the extensions that it supports, and augments authenticatorData as specified by the extension.

Extensions that are not supported are ignored.

7.1 Extension identifiers

Extensions are identified by a string, chosen by the extension author. Extension identifiers should aim to be globally unique, e.g. by using reverse domain-name of the defining entity such as com.example.fido.myextension.

Extensions that may exist in multiple versions should take care to include a version in their identifier. In effect, different versions are thus treated as different extensions.

Standard extensions defined by FIDO in this document use a fixed prefix of fido. for the extension identifiers. This prefix should not be used for 3rd party extensions.

7.2 Defining extensions
A definition of an extension must specify, at minimum, an extension identifier and an extension client argument sent via the `getAssertion` or `makeCredential` call (see below). Additionally, extensions may specify additional values in `clientData`, `authenticatorData` (in the case of signature extensions), or both.

**NOTE**

An extension that does not define additions to `clientData` nor `authenticatorData` is possible, but should be avoided. In such cases, the relying party would have no indication if the extension was supported or processed by the client and/or authenticator.

### 7.2.1 Extending request parameters

An extension defines two request arguments. The **client argument** is passed from the RP to the client in the `getAssertion` or `makeCredential` call, while the **authenticator argument** is passed from the client to the authenticator during the processing of these calls, either natively or through the external authenticator protocol [FIDO\[EAP\]].

Extension definitions must specify the valid values for their client argument. Clients are free to ignore extensions with an invalid client argument. Specifying an authenticator argument is optional, since some extensions may only affect client processing.

An RP simultaneously requests the use of an extension and sets its client argument by including an entry in the `extensions` dictionary parameter to the `getAssertion` or `makeCredential` call. The entry key must be the extension identifier, and the value must be the client argument.

**EXAMPLE 1**

```javascript
var assertionPromise = credentials.getAssertion(..., /* extensions */ {
  "com.example.fido.foobar": 42
});
```

Extensions that affect the behavior of the client platform can define their argument to be any set of values that can be encoded in JSON. Such an extension will in general (but not always) specify additional values to the `clientData` structure (see below). It may also specify an authenticator argument that platforms implementing the extension are expected to send to the authenticator. The authenticator argument should be a byte string.

**NOTE**

Extensions should aim to define authenticator arguments that are as small as possible. Some authenticators communicate over low-bandwidth links such as Bluetooth Low-Energy or NFC.

**NOTE**

Extensions that do not need to pass any particular argument value, must still define a client argument. It is recommended that the argument be defined as the constant value `true` in this case.

For extensions that specify additional authenticator processing only, it is desirable that
the platform need not know the extension. To support this, platforms should pass the client argument of unknown extension as the authenticator argument unchanged, under the same extension identifier. The authenticator argument should be the CBOR encoding of the client argument, as specified in Section 4.2 of [RFC7049]. Clients should silently drop unknown extensions whose client argument cannot be encoded as a CBOR structure.

### 7.2.2 Extending client processing

Extensions may define additional processing requirements on the client platform during the creation of credentials or the generation of an assertion. In order for the RP to verify the processing took place, or if the processing has a result value that the RP needs to be aware of, the extension should specify a client data value to be included in the clientData structure.

The value may be any value that can be encoded as a JSON value. If any extension processed by a client defines such a value, the client should include a dictionary in clientData with the key extensions. For each such extension, the client should add an entry to this dictionary with the extension identifier as the key, and the extension’s client data value.

### 7.2.3 Extending authenticator processing with signature extensions

Signature extensions that define additional authenticator processing should similarly define an authenticator data value. The value may be any data that can be encoded as a CBOR value. An authenticator that processes a signature extension that defines such a value must include it in the authenticatorData.

As specified in 4. Authenticator data, the authenticator data value of each processed extension is included in the extended data part of the authenticatorData. This part is a CBOR map, with extension identifiers as keys, and the authenticator data value of each extension as the value.

### 7.2.4 Example extension

This section is non-normative.

To illustrate the requirements above, consider a hypothetical extension Geo. This extension, if supported, lets both clients and authenticators embed their geolocation in signatures.

The extension identifier is chosen as com.example.fido.geo. The client argument is the constant value true, since the extension does not require the RP to pass any particular information to the client, other than that it requests the use of the extension. The RP sets this value in its request for an assertion:

```javascript
var assertionPromise =
  credentials.getAssertion("SGFuIFNvbg8gc2hvdCBmaXJzdC4",
  {}, /* Empty filter */
  { 'com.example.fido.geo': true });
```

The extension defines the additional client data to be the client’s location, if known, as a GeoJSON [GeoJSON] point. The client constructs the following client data:

```json
{
  ...
  'extensions': {
    'com.example.fido.geo': {
      'type': 'Point',
      'coordinates': [65.059962, -13.993041]
    }
  }
}
```
The extension also requires the client to set the authenticator parameter to the fixed value 1.

Finally, the extension requires the authenticator to specify its geolocation in the authenticator data, if known. The extension e.g. specifies that the location shall be encoded as a two-element array of floating point numbers, encoded with CBOR. An authenticator does this by including it in the authenticatorData. As an example, authenticator data may be as follows (notation taken from RFC7049):

```
81 (hex)                      -- Flags, ED and TUP both set.
20 05 58 1F                   -- Signature counter
A1                            -- CBOR map of one element
   68                          -- Key 1: CBOR text string of 8 bytes
   66 69 64 6F 2E 67 65 6F   -- "fido.geo" UTF-8 string
82                            -- Value 1: CBOR array of two elements
   FA 42 82 1E B3            -- Element 1: Latitude as CBOR encoded float
   FA C1 5F E3 7F           -- Element 2: Longitude as CBOR encoded float
```

7.3 Standard extensions

This section defines standard extensions defined by the FIDO Alliance.

7.3.1 Transaction authorization

This signature extension allows for a simple form of transaction authorization. A relying party can specify a prompt string, intended for display on a trusted device on the authenticator.

**Extension identifier**

`fido.txauth.simple`

**Client argument**

A single UTF-8 encoded string `prompt`

**Client processing**

None, except default forwarding of client argument to authenticator argument.

**Authenticator argument**

The client argument encoded as a CBOR text string (major type 3).

**Authenticator processing**

The authenticator **must** display the prompt to the user before performing the user verification / test of user presence. The authenticator may insert line breaks if needed.

**Authenticator data**

A single UTF-8 string, representing the prompt as displayed (including any eventual line breaks).

The generic version of this extension allows images to be used as prints as well. This is allows authenticators without a font rendering engine to be used and also supports a richer visual appearance.

**Extension identifier**

`fido.txauth.generic`

**Client argument**

A CBOR map with one pair of data items (CBOR tagged as 0xa1). The pair of data items consists of

1. one UTF-8 encoded string `contentType`, containing the MIME-Type of the content, e.g. "image/png"
2. and the `content` itself, encoded as CBOR byte array.

**Client processing**

None, except default forwarding of client argument to authenticator argument.
Authenticator argument
The client argument encoded as a CBOR map.

Authenticator processing
The authenticator must display the content to the user before performing the user verification / test of user presence. The authenticator may add other information below the content. No changes are allowed to the content itself, i.e. inside content boundary box.

Authenticator data
The hash value of the content which was displayed. The authenticator must use the same hash algorithm as it uses for the signature itself.

7.3.2 Authenticator Selection Extension

This registration extension allows a Relying Party to guide the selection of the authenticator that will be leveraged when creating the credential. It is intended primarily for Relying Parties that wish to tightly control the experience around credential creation.

Extension identifier
fido.authn-sel (only used during makeCredential)

Client argument
A sequence of AAGUIDs:

```webidl
typedef sequence < AAGUID > AuthenticatorSelectionList;
```

Each AAGUID corresponds to an authenticator attestation that is acceptable to the RP for this credential creation. The list is ordered by decreasing preference. An AAGUID is defined as a DOMString, and is the globally unique identifier of the authenticator attestation being sought.

```webidl
typedef DOMString AAGUID;
```

Client processing
If the client supports the Authenticator Selection Extension, it must use the first available authenticator whose AAGUID is present in the AuthenticatorSelectionList. If none of the available authenticators match a provided AAGUID, the client must select an authenticator from among the available authenticators to generate the credential.

Authenticator argument
There is no authenticator argument.

Authenticator processing
None.

8. IANA Considerations

This specification registers the algorithm names "S256", "S384", "S512", and "SM3" with the IANA JSON Web Algorithms registry as defined in section "Cryptographic Algorithms for Digital Signatures and MACs" in [JWA].

These names follow the naming strategy in draft-ietf-oauth-spop-15.

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;S256&quot;</td>
<td>The SHA256 hash algorithm.</td>
</tr>
<tr>
<td>Algorithm Name</td>
<td>&quot;S384&quot;</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Algorithm Description</td>
<td>The SHA384 hash algorithm.</td>
</tr>
<tr>
<td>Algorithm Usage Location(s)</td>
<td>&quot;alg&quot;, i.e. used with JWS.</td>
</tr>
<tr>
<td>JOSE Implementation Requirements</td>
<td>Optional</td>
</tr>
<tr>
<td>Change Controller</td>
<td>FIDO Alliance, Contact Us</td>
</tr>
<tr>
<td>Specification Documents</td>
<td>[FIPS180-4]</td>
</tr>
<tr>
<td>Algorithm Analysis Document(s)</td>
<td>[SP800-107r1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>&quot;S512&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Description</td>
<td>The SHA512 hash algorithm.</td>
</tr>
<tr>
<td>Algorithm Usage Location(s)</td>
<td>&quot;alg&quot;, i.e. used with JWS.</td>
</tr>
<tr>
<td>JOSE Implementation Requirements</td>
<td>Optional+</td>
</tr>
<tr>
<td>Change Controller</td>
<td>FIDO Alliance, Contact Us</td>
</tr>
<tr>
<td>Specification Documents</td>
<td>[FIPS180-4]</td>
</tr>
<tr>
<td>Algorithm Analysis Document(s)</td>
<td>[SP800-107r1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>&quot;SM3&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Description</td>
<td>The SM3 hash algorithm.</td>
</tr>
<tr>
<td>Algorithm Usage Location(s)</td>
<td>&quot;alg&quot;, i.e. used with JWS.</td>
</tr>
<tr>
<td>JOSE Implementation Requirements</td>
<td>Optional</td>
</tr>
<tr>
<td>Change Controller</td>
<td>FIDO Alliance, Contact Us</td>
</tr>
<tr>
<td>Specification Documents</td>
<td>[OSCCA-SM3]</td>
</tr>
<tr>
<td>Algorithm Analysis Document(s)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

A. References

A.1 Normative references

[DOM4]
Informative references

[FIDOPlatformApiReqs]
FIDO 2.0: Requirements for Native Platforms. URL: fido-platform-api-reqs.html

[FIDOWebApi]
FIDO 2.0: Web API for accessing FIDO 2.0 credentials. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-web-api.html

[GeoJSON]
The GeoJSON Format Specification. URL: http://geojson.org/geojson-spec.html

[JWA]

[JWK]

[JWS]

[RFC4648]

[RFC6454]

[RFC7049]

[RFC7159]

[SP800-107r1]