FIDO U2F Javascript API

Specification Set: fido-u2f-v1.0-rd-20140209 REVIEW DRAFT

Editors:

Dirk Balfanz (balfanz@google.com)

Contributors:

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:

This Specification has been prepared by FIDO Alliance, Inc. **This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change**. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

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

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

Table of Contents

1 Notation

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 RFC 4648 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.

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 Introduction

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

A *Relying Party* (RP) consumes *identity assertions* from U2F tokens. The RP uses Javascript calls to communicate with the U2F tokens on the client. 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 (using WebIDL) used by the RP.

3 The CryptoTokenHandler

The CryptoTokenHandler is used both for registrations and identity assertions.

```
callback SuccessCallback =
    void ((SignResponse or RegistrationResponse) response);
callback ErrorCallback =
    void (CryptoTokenCodeTypes errorCode);

[Constructor(SuccessCallback successCallback, ErrorCallback errorCallback)]
interface CryptoTokenHandler {
  void handleSignRequest(SignData[] challenges);
  void handleRegistrationRequest(
      RegistrationData registrationData, SignData[] challenges);
}
```

4 Registration

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

- decide which version of device it wants to register (if it supports multiple versions of the protocol, it should perform separate registration operations).
- pick an appropriate application id for the registration request, and
- store all private information associated with the registration (expiration times, user ids, etc.) opaquely in a “sessionID” parameter.

It can then prepare an RegistrationData dictionary with these parameters:

```
dictionary RegistrationData {
  // Version of the protocol that the to-be-registered U2F token must speak.
  // For the version of the protocol described herein, must be ‘U2F_V2’
  DOMString version;

  // The websafe-base64-encoded challenge.
  DOMString challenge;

  // The application id that the RP would like to assert. The new key pair
  // that the U2F device generates will be associated with this application
  // id.
  DOMString app_id;

  // A session id created by the RP. The RP can opaquely store things
  // like expiration times for the registration session,
  // protocol version used, private key material that certain
  // protocol versions require, etc.
  // The response from the API will include the sessionId. This allows the
  // RP to fire off multiple registration requests, and associate
  // the response with the correct request. (Note: this might be more
  // accurately called ‘relying_party_state’, but for compatibility with
  // existing implementations within Chrome we keep the legacy name.)
  DOMString sessionId;
}
```

Additionally, it should prepare SignData objects for each U2F token that the user has already registered with the RP (see below) and then call handleRegistrationRequest on a CryptoTokenHandler object:

```
/**
 * Looks for a locally-attached non-registered U2F device, and asks it to
 * generate a new key pair (and have it attested by an attestation
 * certificate).
 *
```

```
 * @param registrationData the data supplied by the RP, such as
 *   the application id to which the new key pair will be bound,
 *   alongside with the challenge and sessionId.
 * @param challenges identity assertion challenges for U2F devices that the
 *   user has already registered. This allows the user-agent to
 *   identify those locally-attached U2F devices that are already
 *   registered, and not ask them to register again.
 * @param timeout A timeout (in seconds). The browser SHOULD respond
 *   within this timeout and clean up all allocated space when one or the
 *   other happens: (1) Either a success or failure condition occurred,
 *   (2) the timeout elapsed. This parameter is optional and - if
 *   omitted - defaults to 30
 */
void handleRegistrationRequest(RegistrationData registrationData,
   SignData[] challenges, int? timeout);
```

The web browser will create a registration request message from the registrationData, and authentication request messages from the challenges (see the U2F Raw Message Formats document [U2FRawMsgs]), and attempt to perform a registration operation with a U2F token. The authentication request messages will have the checkOnly boolean of the control state set to true, and are used to identity such U2F tokens that are already registered with the relying party. The registration request message is then used to register such U2F tokens that are *not* already registered.

The web browser SHOULD check the supported version of available U2F tokens (using the GetVersion messages - see U2F Raw Message Formats document [U2FRawMsgs]) to ensure that the registration request message will only be sent to U2F tokens that understand the version of the protocol described herein.

Note that as part of creating the registration request message, the web browser will have to create a Client Data object (see the U2F Raw Message Formats document [U2FRawMsgs]). This Client Data object will be returned to the caller as part of the callback (see below).

The CryptoTokenHandler object will call either the successCallback or the errorCallback. In the case of the errorCallback, a CryptoTokenCodeType error code is passed to the callback:

```
interface CryptoTokenCodeTypes {
/**
 * All available U2F tokens are already registered.
 */
const short ALREADY_REGISTERED = 2;

/**
 * None of the available U2F devices are registered.
 */
const short NONE_REGISTERED_FOUND = 3;

/**
```



```
 * One or more devices are lacking test-of-user-presence (TUP)
 * (e.g., missing touch).
 */
const short WAIT_TUP = 4;

/**
 * No U2F devices found.
 */
const short NONE_FOUND = 5;

/**
 * Time out waiting for touch.
 */
const short TOUCH_TIMEOUT = 6;

/**
 * Unknown error during registration.
 */
const short UNKNOWN_ERROR = 7;

/**
 * FIDO Client not available.
 */
const short CLIENT_NOT_FOUND = 8;

/**
 * Empty SignData was passed to the handleSignRequest method.
 */
const short EMPTY_SIGN_DATA = 9;

/**
 * Bad request.
 */
const short BAD_REQUEST = 12;

/**
 * All U2F tokens are too busy to handle your request.
 */
const short BUSY = 13;

/**
 * There is a bad app_id in the request.
 */
const short BAD_APP_ID = 14;

}
```

Note that the errorCallback could be called multiple times, e.g. with the WAIT_TOUCH code, while we wait for the user to tap the U2F token. In the case of the successCallback, a RegistrationResponse is passed to the successCallback:

```
dictionary RegistrationResponse {
  // websafe-base64(raw registration response message)
  DOMString registrationData;

  // websafe-base64(UTF8(stringified(client data)))
  DOMString bd;

  // session id originally passed to handleRegistrationRequest
  DOMString sessionId;
}
```

The browser will call the successCallback only once. If there are multiple U2F tokens that responded to the registration request, the browser will pick one of the responses and pass it to the caller.

The RP must validate the registration response message, which is passed to the caller in websafe-base64-encoded form as the registrationData field. Presumably, the relying party's client-side Javscript code will transmit the message to the server (along with the Client Data and session id), where it will be verified. See the U2F Raw Message Formats document [U2FRawMsgs] for a description of the registration response message, and how to validate the signature.

The transmission of the registration response message from client to server should happen over an authenticated HTTP session that is associated with a certain user account at the relying party. The relying party thus can associate the above public key and key handle with that user.

5 Identity Assertions

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

- prepare a SignData object for each U2F token that the user has currently registered with the RP:

```
dictionary SignData {
  // Version of the protocol that the to-be-registered U2F token must speak.
  // For the version of the protocol described herein, must be 'U2F_V2'
  DOMString version;

  // The websafe-base64-encoded challenge.
  DOMString challenge;

  // The application id that the RP would like to assert. The U2F token will
  // enforce that the key handle provided above is associated with this
  // application id. The browser enforces that the calling origin belongs to
  // the application identified by the application id.
  DOMString app_id;

  // websafe-base64 encoding of the key handle obtained from
  // the U2F token during registration.
  DOMString keyHandle;

  // A session id created by the RP. The RP can opaquely store things
  // like expiration times for the sign-in session, protocol version used,
  // public key expected to sign the identity assertion, etc.
  // The response from the API will include the sessionId. This allows the
  // RP to fire off multiple signing requests, and associate the responses
  // with the correct request.
  DOMString sessionId;
}
```

The RP then calls handleSignRequest on a CryptoTokenHandler object:

```
/**
 * Looks for available registered U2F devices, and attempts to obtain
 * a signature for at least one of the provided challenges. The U2F device
 * will not sign the provided challenge directly. Instead, it will sign a
 * ClientData object (see below), which will contain (among other things)
 * the challenge passed in as part of the SignData object.
 *
 * @param challenges identity assertion challenges for U2F devices that the
 *   user has already registered.
 * @param timeout A timeout (in seconds). The browser SHOULD respond
 *   within this timeout and clean up all allocated space when one or the
 *   other happens: (1) Either a success or failure condition occurred,
 *   (2) the timeout elapsed. This parameter is optional and - if
 *   omitted - defaults to 30
 */
```

```
void handleSignRequest(SignData[] challenges, int? timeout);
```

The web browser now performs the following steps: First, it verifies the application identity of the caller (see the document “U2F Application Isolation through Facet Identification”). Using the provided challenge, it creates a client data object. Using the client data, the application id, and the key handle, it creates a raw authentication request message (see the U2F Raw Message Formats document [U2FRawMsgs]) and sends it to the U2F token.

Eventually the CryptoTokenHandler object will call either the successCallback or the errorCallback. In the case of the errorCallback, a CryptoTokenCodeType error code is passed to the errorCallback (see above). Note that the errorCallback could be called multiple times, e.g. with the WAIT_TOUCH code, while we wait for the user to tap the U2F token. The successCallback is called at most once. If there are multiple U2F tokens that responded to the authentication request, the browser will pick one of the responses and pass it to the caller.

In the case of the successCallback, a SignResponse is passed to the successCallback:

```
dictionary SignResponse {
  // websafe-base64(client data)
  DOMString bd;

  // websafe-base64(raw response from U2F device)
  DOMString sign;

  // challenge originally passed to handleSignRequest
  DOMString challenge;

  // session id originally passed to handleSignRequest
  DOMString sessionId;

  // application id originally passed to handleSignRequest
  DOMString app_id;
}
```

We explain the first two parameter in the response below: The ‘bd’ parameter is a websafe-base64-encoding of the UTF-8 encoding of a (serialized) JSON Object representation of the following type:

```
dictionary ClientData {

  // the constant ‘navigator.id.getAssertion’ for authentication, and
  // ‘navigator.id.finishEnrollment’ for registration
  DOMString typ;

  // The base64-encoding of the challenge passed to handleSignRequest and
  // handleRegistrationRequests
  DOMString challenge;
```

```
  // the web origin of the caller to handleSignRequest. Note that
  // the browser won’t allow the call to handleSignRequest to succeed
  // unless this origin is a facet of the passed-in application id.
  DOMString origin;

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

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

  // signature algorithm used for Channel ID, i.e., the constant ‘EC’
  DOMString kty;

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

  // websafe-base64-encoding of the x coordinate of the public
  // key (big-endian, 32-byte value)
  DOMString x;

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

The RP must validate the **`sign`** parameter from the SignResponse (presumably server-side). The (base64-decoded) sign parameter is the raw authentication response message as explained in the U2F Raw Message Formats document [U2FRawMsgs]. Apart from verifying the signature (as explained there),

- The RP should verify that the counter value is increasing.
- The RP should validate the ClientData (i.e., verify that the Channel ID, origin, challenge, and typ parameters equal expected values).
- The RP should validate the application id used during the signing as one that it is using. Most servers will use a constant application id, but a hosting provider might use several applications id.

Bibliography

FIDO Alliance Documents:

[FIDOGlossary] Rolf Lindemann, Davit Baghdasaryan, Brad Hill, John Kemp. FIDO Technical Glossary. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-glossary-v1.0-rd-20140209.pdf

[**U2FOverview**] Sampath Srinivas, Dirk Balfanz, Eric Tiffany. FIDO Universal 2nd Factor (U2F) Overview. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-u2f-overview-v1.0-rd-20140209.pdf

[**U2FRawMsgs**] Dirk Balfanz. FIDO U2F Raw Message Formats. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-u2f-raw-message-formats-v1.0-rd-20140209.pdf

Other References:

[RFC2119] Key words for use in RFCs to Indicate Requirement Levels (RFC2119), S. Bradner, March 1997