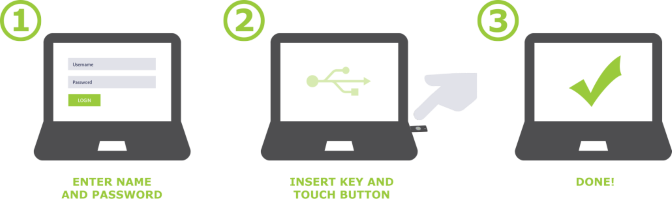
U2F Client Integration

# FIDO U2F Overview

U2F is an open authentication standard that enables keychain devices, mobile phones and other devices to securely access any number of web-based services — instantly and with no drivers or client software needed. U2F was created by Google and Yubico, with contribution from NXP, and is today hosted by the open-authentication industry consortium [FIDO Alliance](https://fidoalliance.org/). The technical specifications were launched in late 2014, including native support in Google Accounts and Chrome, and have since resulted in a thriving ecosystem of hardware, software and service providers.



*U2F is used with USB devices, including YubiKeys, as one of many authentication methods*

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.

U2F Tokens provide cryptographic assertions that can be verified by server/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 are not 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.

## Site-Specific Public/Private Key Pairs

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 clouding sites to strongly verify and correlate a particular user's identity.

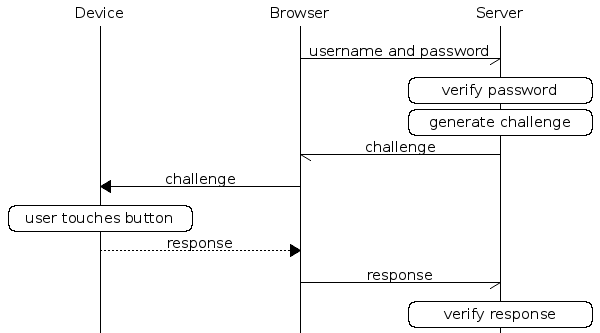
## U2F Registration and Authentication Operations

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 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. The U2F device needs to see a 'test of user presence' before it will sign -- i.e, the user has to press a button on the device for example. 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).

This diagram explains the basic process flow of U2F:



## U2F Technical Overview

U2F is a challenge-response protocol extended with *phishing and MitM protection, application-specific keys*, *device cloning detection* and *device attestation*. There are two flows: *registration* and *authentication*.

1. Challenge-response

Let’s start out with a simple [challenge-response](http://en.wikipedia.org/wiki/Challenge%E2%80%93response_authentication) authentication flow, based on [public-key cryptography](http://en.wikipedia.org/wiki/Public-key_cryptography). The U2F device has a private key *Kpriv* and the RP is given the corresponding public key K*pub*. The key pair is generated in the device’s [tamper-resistant](http://en.wikipedia.org/wiki/Tamper_resistance#Chips) execution environment, from where *Kpriv* cannot leave.

Relaying Party

challenge

challenge

signature (challenge)

s

s

Sign with Kpriv

lookup Kpub

check signature using Kpub

U2F Token

Client

2. Phishing and MitM protection

The concept is that the client compiles what it knows about the current HTTP connection (URI and TLS Channel ID). This information is then signed by the U2F device and sent to the RP, which verifies that the information is correct.

Additions to the authentication flow:

* Origin (URI) — prevents phishing.
* [TLS Channel ID](http://en.wikipedia.org/wiki/Transport_Layer_Security_Channel_ID) (optional) — prevents MitM.

Client

Relaying Party

challenge

challenge, origin, channel id, etc

signature (c)

s

c, s

Sign with Kpriv

lookup Kpub

check signature using Kpub

U2F Token

c

3. Application-specific keys

Application-specific keys prevents relying parties from tracking devices between different user accounts. This means that Example.com cannot know whether *User1* and *User2* shares the same device.

The U2F device generates a new key pair and key handle for each registration. The handle is stored by the RP and sent back to the device upon authentication. This way, the device knows which key to authenticate with (e.g. *User1*'s key or *User2*'s key).

Additions to the authentication flow:

* [Key generation](https://developers.yubico.com/U2F/Protocol_details/Key_generation.html) on the device
* Key handle, stored by the server together with *kpub*.
* [App ID](https://developers.yubico.com/U2F/App_ID.html), used for scoping a key handle.

Relaying Party

Handle, app id, challenge

h, a, challenge, origin, channel id, etc

signature (a, c)

s

c, s

lookup Kpriv associated with h

lookup Kpub associated with h

Check Signature using Kpub

Verify c

U2F Token

Client

Check app id

a

h

c

4. Device cloning detection

As already mentioned, Yubico’s U2F devices are tamper-resistant and *kpriv* cannot be read externally. However, to provide cloning detection to U2F devices without tamper-resistant secure elements (e.g software implementations), Yubikey adds an authentication counter. The concept is simple: The device increments the counter when authenticating, and the RP verifies that the counter is higher than last time.

Additions to the authentication flow:

* A counter, sent from the device to the RP.

Relaying Party

handle, app id, challenge

h, a, challenge, origin, channel id, etc

counter, signature (a, c, counter)

s

counter, c, s

lookup Kpriv associated with h

counter++

lookup Kpub associated with h

Check Signature using Kpub

Verify c

U2F Token

Client

check app id

h

a

c

5. Device attestation

Attestation gives relying parties *the possibility* to verify token properties, such as token model. It is implemented via an attestation certificate, signed by the device vendor, that the device sends to the RP upon registration. Attestation does not affect the authentication flow.

Additions to the registration flow:

* [Attestation certificate](https://developers.yubico.com/U2F/Libraries/Advanced_topics.html)

Client

Relaying Party

app id, challenge

a, challenge, origin, channel id, etc

Kpub,h, attestation cert, signature (a, c, Kpub, h)

s

c, Kpub, h, attestation cert, s

Generate Kpub Kpriv handle h

Associate Kpub with handle h for users

U2F Token

check app id

a

c

## U2F Protocol Specifications

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).

# U2F Server Application Messages

FIDO U2F specification only defines Application messages between Server and Client, but does not specify what transport layer to be used to transmit the application messages. In Google’s case, Relay Party web pages communicate with the FIDO client over an instance of the HTML5 MessagePort interface on top of the existing https connection. Client implementations may choose how this instance is made available to web pages. The U2F application messages are presented in JSON format and the RP's web pages communicate with the U2F tokens on the client through a Javascript API.

## U2F Server Request Message

Messages sent from U2F server to the FIDO client SHOULD be Request dictionaries:

dictionary **Request** {

DOMString type;

**SignRequest**[] signRequests;

**RegisterRequest**[]? registerRequests;

int? timeoutSeconds;

optional int? requestId;

};

**type** of type DOMString

The type of request, either "u2f\_register\_request" or "u2f\_sign\_request".

**signRequests** of type array of *SignRequest*

A list of SignRequest dictionaries, one for each token already registered with this RP.

**registerRequests** of type array of *RegisterRequest*, nullable

A list of RegisterRequest dictionaries, one for each protocol version that the RP is willing to register.

**timeoutSeconds** of type int, nullable

A timeout for the FIDO Client's processing, in seconds.

**requestId** of type optional int, nullable

An integer identifying this request from concurrent requests.

SignRequest and RegisterRequest are defined below. 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.

### Dictionary RegisterRequest

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,
* store all private information associated with the registration (expiration times, user ids, etc.)

dictionary **RegisterRequest** {

DOMString version;

DOMString challenge;

DOMString appId;

};

**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 websafebase64encoded challenge.

**appId** of type DOMString

The application id that the RP asserts. The new key pair that the U2F token generates will be associated with this application id.

Additionally, the RP SHOULD prepare a SignRequest for each U2F token that is already registered for the current user. See the following section for the specification of sign requests. The FIDO client will create the raw registration and sign request messages from this data, and attempt to perform a registration operation with a U2F token. The sign request messages will have the (internal) 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 a U2F token that is not already registered (if such a token is present).

### Dictionary SignRequest

To obtain an identity assertion from a locally attached U2F token, the RP must prepare a SignRequest object for each U2F token that the user has currently registered with the RP.

dictionary **SignRequest** {

DOMString version;

DOMString challenge;

DOMString keyHandle;

DOMString appId;

};

**version** of type DOMString

Version of the protocol that the to-be-registered U2F token must speak. E.g. "U2F\_V2"

**challenge** of type DOMString

The websafebase64encoded challenge.

**keyHandle** of type DOMString

The registered keyHandle to use for signing, as returned by the U2F token during registration.

**appId** of type DOMString

The application id that the RP would like to assert.

### Request Message Examples between U2F Server and Client

U2F specifications do not dictate the transport layer between U2F server and client. It only defines the message contents on the application in JASON string format.

#### Register Request from Server to Client

Register Request with no key handle

{ 'type':'u2f\_register\_request', 'signRequests':'[ ]', 'registerRequests':'[ { 'version':'U2F\_V2', 'challenge':'pj+EhsWuSQJxx7przmb8HM+ZkeNcG3He','appId':'www.boa.com/members/onlinebanking' } ]', 'timeoutSeconds':'30', 'requestId':'1' }

Register Request with one key handle

{ 'type':'u2f\_register\_request', 'signRequests':'[ { 'version':'U2F\_V2', 'challenge':'2iRdhLfawKZC5ydJsA+FZVpAyqJMT/nW', 'keyHandle':'rLxO094dRZH/yPc2Dq8I1NeNrRXeTgfYo9zYfgXxZRvH39mSN6Zlfz95/hZqaQLAh2eEYHsL/FE/8M/YAV5VLw==', 'appId':'www.boa.com/members/onlinebanking' } ]', 'registerRequests':'[ { 'version':'U2F\_V2', 'challenge':'gaQuA429SJy2e9JctPUyEAmQrEw3y7hi', 'appId':'www.boa.com/members/onlinebanking' } ]', 'timeoutSeconds':'30', 'requestId':'3' }

#### Sign Request from Server to Client

{ 'type':'u2f\_sign\_request', 'signRequests':'[ { 'version':'U2F\_V2', 'challenge':'gaQuA429SJy2e9JctPUyEAmQrEw3y7hi', 'keyHandle':'xnIPvmkPFKlzjj5kX2nmoZzahWsX7AVsXuAoZVRzgfbDQCDvIOxdywquvKvXa2EVuGCSXuWJwiVub6+iqgi0Dw==', 'appId':'www.boa.com/members/onlinebanking' } ]', 'timeoutSeconds':'30', 'requestId':'3' }

## U2F Server Response Message

Responses from the FIDO client to the U2F Server SHOULD be Response dictionaries:

dictionary **Response** {

DOMString type;

(**Error** or **RegisterResponse** or **SignResponse**) responseData;

int? requestId;

};

**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 4. U2F operations

**requestId** of type int, nullable

The requestId value of the corresponding request, if present. Otherwise omitted.

### Dictionary Error

Errors are indicated by an Error dictionary sent as the response data. An error dictionary can be identified by checking for its nonzero 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.

dictionary **Error** {

**ErrorCode** errorCode;

DOMString? errorMessage;

};

**errorCode** of type *ErrorCode*

An error code from the ErrorCode enumeration.

interface **ErrorCode** {

const int OK = 0;

const int OTHER\_ERROR = 1;

const int BAD\_REQUEST = 2;

const int CONFIGURATION\_UNSUPPORTED = 3;

const int DEVICE\_INELIGIBLE = 4;

const int TIMEOUT = 5;

};

**errorMessage** of type DOMString, nullable

A description of the error.

### Dictionary RegisterResponse

Note that as part of creating the registration request message, the FIDO client will create a Client Data object. 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.

dictionary **RegisterResponse** {

DOMString registrationData;

DOMString clientData;

};

**registrationData** of type DOMString

The raw registration response websafebase64

**clientData** of type DOMString

The client data created by the FIDO client, websafebase64 encoded.

#### Server Actions when Receiving RegisterResponse

U2F Server should verify all parameters and validate the signature using Token certificate, and then store the Key Handle and public key for Authentication (Sign) Request.

1. Decode both **registrationData** and **clientData** base 64 strings
2. Retrieve and compare elements inside **clientData**
   1. **Challenge**
   2. **Origin**
   3. **Others…**
3. Parse to retrieve information inside **registrationData**
   1. **User Public Key**
   2. **Ken Handle and its length**
   3. **Token Attestation Certificate**
   4. **ECDSA signature**
4. Extract token attestation public key from token attestation certificate
5. Format signature strings with application SHA256 string, challenge SHA256 string, key handle and length, and user public key, and then create signature digest from signature string prepared
6. Verify signature digest created against ECDSA signature retrieved from **registrationData** using token attestation public key
7. Store Key Handle and User Public Key for this Token

### Dictionary SignResponse

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 and send it to the U2F token.

Eventually the FIDO client must respond. In the case of an error, an Error dictionary is returned. In case of success, a SignResponse is returned.

dictionary **SignResponse** {

DOMString keyHandle;

DOMString signatureData;

DOMString clientData;

};

**keyHandle** of type DOMString

The keyHandle of the SignRequest that was processed.

**signatureData** of type DOMString

The raw response from U2F device, websafebase64 encoded.

**clientData** of type DOMString

The client data created by the FIDO client, websafebase64 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.

#### Server Actions when Receiving SignResponse

Upon receiving the SignResponse from the Client, U2F Server should verify all parameters and validate the signature using public key previously stored from RegisterReponse.

1. Retrieve **keyHandle** and decode both **signatureData** and **clientData** base 64 strings
2. Retrieve and compare elements inside **clientData**
   1. **Challenge**
   2. **Origin**
   3. **Others…**
3. Parse to retrieve information inside **signatureData**
   1. **Presence**
   2. **Counter**
   3. **ECDSA signature**
4. Format signature strings with application SHA256 string, challenge SHA256 string, presence, and counter, and then create signature digest from signature string prepared
5. Verify signature digest created against ECDSA signature retrieved from **signatureData** using User Public Key previously stored from RegistrationResponse

### Response Message Examples between U2F Server and Client

U2F specifications do not dictate the transport layer between U2F server and client. It only defines the message contents on the application in JASON string format.

#### Register Response from Client to Server

{ 'type':'u2f\_register\_response', 'responseData':'{ 'registrationData':'', 'clientData':'eyAndHlwZSc6J25hdmlnYXRvci5pZC5maW5pc2hFbnJvbGxtZW50JywgJ29yaWdpbic6J2h0dHBzOi8vYXV0aGVudGlmeS9yZWdpc3RlcicsICdjaGFsbGVuZ2UnOidwaitFaHNXdVNRSnh4N3Byem1iOEhNK1prZU5jRzNIZScgfQ==' }', 'requestId':'1' }

#### Sign Response from Client to Server

{ 'type':'u2f\_sign\_response', 'responseData':'{ 'keyHandle':'xnIPvmkPFKlzjj5kX2nmoZzahWsX7AVsXuAoZVRzgfbDQCDvIOxdywquvKvXa2EVuGCSXuWJwiVub6+iqgi0Dw==', 'signatureData' (76):'AQAAAH8wRQIhAKVxoeXVATpxtcxSRmZF2TtHnfdfATekQwIP0DfmbMzjAiBrZzk5YpBOQzRDyWufu3eSu51jQ+sCA5OZ9lVqWib+VQ==', 'clientData':'eyAndHlwZSc6J25hdmlnYXRvci5pZC5nZXRBc3NlcnRpb24nLCAnb3JpZ2luJzonaHR0cHM6Ly9hdXRoZW50aWZ5L3NpZ24nLCAnY2hhbGxlbmdlJzonZ2FRdUE0MjlTSnkyZTlKY3RQVXlFQW1RckV3M3k3aGknIH0=' }', 'requestId':'3' }

# U2F Client Application (Raw) Messages

The U2F protocol supports two operations, registration and authentication. The registration operation introduces the relying party to a freshly minted key pair that is under control of the U2F authenticator. The authentication operation proves possession of a previously registered key pair to the relying party.

The registration and authentication operations consist of a request and response pair of messages.

## Request Message

The registration and authentication request messages will be warped inside APDU Request defined in specification ISO 7816-4 and sent to the U2F token.

### APDU Request

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Description** |
| CLA | 1 | Class of instruction |
| INS | 1 | Instruction code |
| P1 | 1 | Instruction parameter 1 |
| P2 | 1 | Instruction parameter 2 |
| Lc1 | 1 | Length counter 1 If not 0, data length = Lc1, there will be no Lc2 & Lc3 If 0, data length = Lc2 << 8 + Lc3 |
| Lc2 | 1 | Length counter 2 |
| Lc3 | 1 | Length counter 3 |
| Request Data | Lc | Data of Lc bytes |
| Le | 2 | 0x0000 – legacy coding from Google U2F client |

### Registration Request – U2F\_REGISTER

Registration Request raw data APDU format defined below.

|  |  |
| --- | --- |
| **Field** | **Values** |
| CLA | 0x00 |
| INS | 0x01 (U2F\_REGISTER) |
| P1 | 0x03 (Request User Presence) |
| P2 | 0x00 |
| Lc1 | 0x00 (use extended length) |
| Lc2 | 0x00 |
| Lc3 | 0x40 (64 bytes) |
| DATA (Registration Request) | Defined in ‘Request Data – Registration’ |
| Le | 0x0000 |

#### Request Data – Registration

|  |  |
| --- | --- |
| **Challenge Parameter** | **Application Parameter** |
| 32 bytes | 32 bytes |

The ***challenge parameter*** [32 bytes] is the SHA256 hash of the Client Data, a stringified JSON data structure that the FIDO Client prepares. Among other things, the Client Data contains the challenge and maybe the origin from the relying party/server.

Example of the Client Data in JASON data structure:

{ 'type':'navigator.id.finishEnrollment, 'origin':' https://authentify/register', 'challenge':'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYz123456789012' }

The ***application parameter*** [32 bytes] is the SHA256 hash of the application identity of the application requesting the registration as provided by the relying party/server.

### Raw data example for Register Request in HEX

**00010300000040A1A50C40131A1E3F905826942130173645CA1183FD3207C75179C24D59920421332BF27F1C3946F2FDFC3979AB022204AF3C0D1A7016CD9A927FBF262339B4C30000**

**00** - CLA

**01** – INS (U2F\_REGISTER)

**03** – P1 (Request User Presence)

**00** – P2

**000040** – Length (extended format, value 64 bytes)

**A1A50C40131A1E3F905826942130173645CA1183FD3207C75179C24D59920421** – challenge parameter – 32 bytes

**332BF27F1C3946F2FDFC3979AB022204AF3C0D1A7016CD9A927FBF262339B4C3** – application parameter – 32 bytes

**0000** – expected length – 2 bytes

### Authentication Request - U2F\_AUTHENTICATE

Registration Request raw data APDU format defined below.

|  |  |
| --- | --- |
| **Field** | **Values** |
| CLA | 0x00 |
| INS | 0x02 (U2F\_AUTHENTICATE) |
| P1 | 0x03 (Request User Presence) |
| P2 | 0x00 |
| Lc1 | 0x00 (use extended length) |
| Lc2 | 0xmm |
| Lc3 | 0xnn (64 + 1 + L) bytes |
| DATA (Registration Request) | Defined in ‘Request Data – Authentication’ |
| Le | 0x0000 |

#### Request Data - Authentication

|  |  |  |  |
| --- | --- | --- | --- |
| **Challenge Parameter** | **Application Parameter** | **Key Handle Length** | **Key Handle** |
| 32 bytes | 32 bytes | 1 byte | L bytes |

The ***challenge parameter*** [32 bytes] is the SHA256 hash of the Client Data, a stringified JSON data structure that the FIDO Client prepares. Among other things, the Client Data contains the challenge and maybe the origin from the relying party/server.

Example of the Client Data in JASON data structure:

{ 'type':'navigator.id.getAssertion', 'origin':'https://authentify/sign', 'challenge':'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYz123456789012' }

The ***application parameter*** [32 bytes] is the SHA256 hash of the application identity of the application requesting the authentication as provided by the relying party/server.

A ***key handle length*** byte [1 byte], which specifies the length of the key handle.

A ***key handle*** [length specified in previous field] is provided by the relying party, and was obtained by the relying party during registration from the U2F token.

### Raw data example for Sign Request in HEX

**0002030000008195565CD17291F70D6A1FB99CB1963AE56E4CA80E2239524ADCEC5861EFA99DFB332BF27F1C3946F2FDFC3979AB022204AF3C0D1A7016CD9A927FBF262339B4C3408B43AA955FC69022C061E51C0EFD50132EC2B93958DAF07D1B6628C970672E193586E1FC0FD462ABFAA0D7B9E2885B96C2A63B64F1A4E82AEC7F2D6135743F050000**

**00** - CLA

**02** – INS (U2F\_ AUTHENTICATE)

**03** – P1 (Request User Presence)

**00** – P2

**000081** – Length (extended format, value 129 bytes)

**95565CD17291F70D6A1FB99CB1963AE56E4CA80E2239524ADCEC5861EFA99DFB** – challenge parameter – 32 bytes

**332BF27F1C3946F2FDFC3979AB022204AF3C0D1A7016CD9A927FBF262339B4C3** – application parameter – 32 bytes

**40** – Length of Key Handle (value 64) – 1 byte

**8B43AA955FC69022C061E51C0EFD50132EC2B93958DAF07D1B6628C970672E193586E1FC0FD462ABFAA0D7B9E2885B96C2A63B64F1A4E82AEC7F2D6135743F05** – Key Handle - 64 bytes

**0000** – expected length – 2 bytes

## Response Message

The registration and authentication response messages will be warped inside APDU response defined in ISO 7816-4 and returned from the U2F token.

### APDU Response

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Description** |
| Response Data  (Registration & Authentication) | Lr - 2 | Data of received response length (Lr) - 2 bytes  Response data defined in next sections |
| SW1 | 1 | Process result |
| SW2 | 1 | additional result detail |

### Response Status Codes (SW1 & SW2)

Denoted as SW1-SW2, the status bytes tell the processing state in the token which are defined in the part 4 of ISO/IEC 7816. A good response will have SW1&2 value 0x9000. While waiting for the user presence (touch the button), error code ‘6985’ is returned.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SW1-SW2 | | | | | | | | | | |
|  |  | |  | |  |  |  |  |  |  |
|  | Process Completed | | | |  |  |  | Process Aborted | |  |
|  |  | |  | |  |  |  |  |  |  |
|  |  | |  | |  |  |  |  |  |  |
| Normal Processing '9000' and '61XX' | | | Warning Processing '62XX' and '63XX' | | |  | Execution Error '64XX' to '66XX' | | Checking Error '67XX' to '6FXX' | |
|  | | SW1-SW2 | | Meaning | | | | | | | |
| Normal processing | | 9000 | | No further qualification | | | | | | | |
| 61XX | | SW2 tells the number of response bytes still available | | | | | | | |
| Warning processing | | 62XX | | State of non-volatile memory is unchanged (further qualification in SW2) | | | | | | | |
| 63XX | | State of non-volatile memory is changed (further qualification in SW2) | | | | | | | |
| Execution error | | 64XX | | State of non-volatile memory is unchanged (further qualification in SW2) | | | | | | | |
| 65XX | | State of non-volatile memory is changed (further qualification in SW2) | | | | | | | |
| 66XX | | Security-related issues | | | | | | | |
| Checking error | | 6700 | | Wrong length | | | | | | | |
| 68XX | | Functions in CLA not supported (further qualification in SW2) | | | | | | | |
| 69XX | | Command not allowed (further qualification in SW2) | | | | | | | |
| 6AXX | | Wrong parameters P1-P2 (further qualification in SW2) | | | | | | | |
| 6BXX | | Wrong parameters P1-P2 | | | | | | | |
| 6CXX | | Wrong length Le; SW2 tells the exact length | | | | | | | |
| 6D00 | | Instruction code not supported or invalid | | | | | | | |
| 6E00 | | Class not supported | | | | | | | |
| 6F00 | | No precise diagnosis | | | | | | | |

### Response Data - Registration

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Reserved Byte** | **User Public Key** | **Key Handle Length** | **Key Handle** | **Attestation Certificate** | **Signature** |
| 1 byte | 65 bytes | 1 byte | L bytes | X.509 bytes | X bytes |

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 P256.

NIST elliptic curve.

A ***key handle length*** byte [1 byte], which specifies the length of the key handle (see below).

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***. This is an ECDSA signature (on P256) over the following byte string and is to be verified by the relying party using the X.509 attestation certificate.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reserved Byte** | **Application Parameter** | **Challenge Parameter** | **Key Handle** | **User Public Key** |
| 1 byte | 32 bytes | 32 bytes | L | 65 bytes |

A ***reserved byte*** for future use [1 byte] with the value 0x00. This will evolve into a byte that will allow RPs to track known good applet version of U2F tokens from specific vendors.

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].

The above ***user public key*** [65 bytes].

### Raw data example for Register Response in HEX



**05** – Reserved byte (value 0x05) – 1 byte

**04EEB32988F1C46467BB4FEE4C27036DBBFE4C82AA2F3E2AA1A79F981995A29B81DB219B670DEDBF8C87591B54C4FC191BE2F7E7490AD4C95457FCC82F9FDE562C** – User public key – 65 bytes

**40** – Length of Key Handle (value 64 bytes) - 1 byte

**A6218B3B99AF17046592270FE55414C034B55FF5192D28277F48780EFA6052F6D520E25B0B507A38DA6B303F13726E98AB1B50FB801270A21735C8EEB3EA1DC2** – Key Handle – 64 bytes

– Attestation certificate in X.509 format - 544 bytes (0x**021C+4**)

**304402200A0C88C7C7BA5DAF4A999275094200DEC1DE831D13ACA8EADCD4E6C17BB570FE022010A84396D09E3A4B834DD696DFC83B7F403A77FC84D1370F6A678881253B8F13** – Signature

### Response Data - Authentication

|  |  |  |
| --- | --- | --- |
| **User Presence** | **Counter** | **Signature** |
| 1 byte | 4 bytes | X bytes |

A ***user presence*** byte [1 byte]. Should set to 1, which means that user presence was verified.

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.

A ***signature***. This is an ECDSA signature (on P256) over the following byte string:

|  |  |  |  |
| --- | --- | --- | --- |
| **Application Parameter** | **User Presence** | **Counter** | **Challenge Parameter** |
| 32 bytes | 1 byte | 4 bytes | 32 bytes |

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 to be verified by the relying party using the ***user public key*** obtained during registration.

### Raw data example for Sign Response in HEX

**0100000080304402204FC6B50EE010D5A3797DF428609450BDB3AE2D77F18B1AC352C9DA1008D9FA45022059322C052A6210EF517FDDEBE465639693E4B887FD7F6E133F54618F9DA27280**

**01** – User presence

**00000080** – Counter

**304402204FC6B50EE010D5A3797DF428609450BDB3AE2D77F18B1AC352C9DA1008D9FA45022059322C052A6210EF517FDDEBE465639693E4B887FD7F6E133F54618F9DA27280** – Signature

# FIDO U2F Client Software Programming Model

The FIDO U2F Client Software Programming Model illustrated below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Registration & Authentication Request/Response | Application | U2F Client Messages in Raw Data with APDU Format | | | |  |
| Session | N/A | N/A | N/A | Channel ID |  |
| Data Transmission | Transport - Fragmentation | N/A (\*) | Using Transport U2F Framing | N/A (\*) | Using HID Framing |  |
| Transport - Framing | BT/BLE U2F Framing | | N/A | HID Report | (\*) Always use extended length |
| Transport - Encapsulation | RFCOMM | GATT U2F Service | isoDep | HID Packet |  |
| Device Discovery Scan & Realtime Device Change | Link/Network - Encryption | Security Mode 4 / LTK | Security Mode 1 / Level 2 / LTK | N/A | N/A |  |
| Link/Network - Establishment | Paring/SDP | Paring/Discover FIDO U2F Service | FIDO U2F AppId | Usage Page/ID |  |
|  |  | BT | BLE | NFC | USB |  |

The U2F Client Application Messages in Raw Data with APDU Format is described in Chapter 3. Currently there is only YubiKey USB U2F Token available in the market, and officially there are only low layer specifications ready for the USB token. Draft documents just made available lately for the NFC and BT/BLE devices and there are SIM and µ-SD devices still under discussion. The Client interface software programming model for each different physical layer device will be discussed in next chapter.

## USB Tokens

Probably make sense to implement U2F Client application only on Window PC for U2F USB Token.

### Device Discovery

#### Link / Network - Establishment

On the Window PC, enumerate USB HID Composite devices currently registered in the system. For each USB HID Composite device found, retrieve the HID ‘Usage Page’ and ‘Usage’ information and check if it is a U2F device.

The FIDO U2F HID token consists of the following fields:

|  |  |
| --- | --- |
| **Field** | **Value** |
| Usage Page | 0xFID0 |
| Page | 0x01 |

In addition, equip the trigger to catch the Window ‘Device Change’ event to monitor if there is a USB U2F token inserted into or removed from the system.

#### Link/Network - Encryption

There will be no data encryption for the USB communication.

### Data Transmission

For U2F USB token, application data will be transmitted through the Human Interface Device (HID) transport layer. The basic communication mechanism for HID class devices is a HID Report that is used to transfer HID data. HID Report Descriptor defines the format of each report and Input and Output report size can retrieved from the HID Report Descriptor.

#### Transport – Framing

HID packets are transmitted via Report frames. Although it is not specifically defined in the U2F HID Protocol documents; in YubiKey U2F Token implementation, one extra Report ID byte was added on top of the HID packet frame and currently it is always set to 0x00. It is recommended in the U2F HID Protocol Specification to implement U2F token with a full-speed USB device and 64-bytes endpoints. So the Input and Output report frame size for Yubikey is 64+1=65 bytes.

#### Transport –Encapsulation

Packets are one of two encapsulation types, initialization packets and continuation packets.

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 length (\*) |

(\*) s = ( Input/Output length in Report Descriptor - Encapsulation size )

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 clear) |
| 5 | (s - 5) | DATA | Payload data length (\*) |

#### Transport – Fragmentation

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, one or more continuation packets have to be sent in ascending order to complete the message transfer. With this approach, a message with a payload less or equal to (s - 7) will 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.

#### Session - HID Channel ID

U2FHID\_INIT command synchronizes a channel and requests the device to allocate a unique 32bit 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 application shall use the broadcast channel U2FHID\_BROADCAST\_CID.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Length** | **Mnemonic** | **Values** |
| 0 | 4 | CID | CID\_BROADCAST (0xFFFFFFFF) |
| 4 | 1 | CMD | U2FHID \_INIT (0x86) |
| 5 | 1 | BCNTH | 0x00 |
| 6 | 1 | BCNTL | 0x08 |
| 7 | (s - 7) | DATA | 8 byte Nonce |

The device then responds the newly allocated channel in the response, using the broadcast channel.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Length** | **Mnemonic** | **Values** |
| 0 | 4 | CID | CID\_BROADCAST (0xFFFFFFFF) |
| 4 | 1 | CMD | U2FHID \_INIT (0x86) |
| 5 | 1 | BCNTH | 0x00 |
| 6 | 1 | BCNTL | 0x11 (17 bytes) |
| 7 | 17 | DATA | 17 byte response defined below |

17 byte response data.

|  |  |  |
| --- | --- | --- |
| **Offset** | **Length** | **Description** |
| DATA+0 | 8 | 8 byte Nonce |
| DATA+8 | 4 | New 4 byte Channel ID |
| DATA+12 | 1 | U2FHID protocol version identifier |
| DATA+13 | 1 | Major device version number |
| DATA+14 | 1 | Minor device version number |
| DATA+15 | 1 | Build device version number |
| DATA+16 | 1 | Capabilities flags |

### Application Messages

After a session Channel ID is allocated (described above), the Command ID used to send regular request and response control data is U2FHID \_MSG (0x83).

Formation of the application Registration/Authentication Request and Response messages is described in “U2F Client Application (Raw) Messages” chapter.

## NFC Tokens

Probably make sense to implement U2F Client application on Android smartphone or iPhone for U2F NFC Token, but not Windows PC.

### Device Discovery

#### Link / Network - Establishment

On the NFC equipped devices like smart phones (Android and iPhone) or Window PC, setup the NFC application with read and write permission using IsoDep tag and wait for NFC equipped token discovery. Once found, Client sends an applet selection command in APDU format to the token. If token replies successfully with a valid U2F version string, then it is a U2F token.

The FIDO U2F AID (Application ID) 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. The current version string is "U2F\_V2", hence a successful response to the applet selection command would consist of the following bytes:

**0x5532465F56329000**

|  |  |  |
| --- | --- | --- |
| **Field** | **Value** | **Description** |
| DATA (Response) | 0x5532465F5632 | U2F\_V2 |
| SW1+SW2 | 0x9000 | Successful Response |

#### Link/Network - Encryption

There will be no data encryption for the NFC communication.

### Data Transmission

Unlike the USB HID protocol, for example; the NFC protocol SHALL NOT use any additional framing. Instead, messages sent to an NFC authenticator SHALL follow the U2F raw message format as defined.

#### Transport –Framing

There will be no Transport framing for NFC token.

#### Transport – Encapsulation

There will be no Transport encapsulation for NFC token.

#### Transport – Fragmentation

There will be no Fragmentation in Transport layer. If response fragmentation is needed in case 1 length byte is used in the request, then fragmentation needs to be done using APDU Chaining mechanism in the application layer. If the request was of extended length (*i.e.,* had 3 length bytes), the authenticator MUST respond using the extended length APDU format (*i.e.* no APDU Chaining is needed).

### Session

There will be no Session layer for NFC token.

### Application Messages

Formation of the application Registration/Authentication Request and Response messages is described in “U2F Client Application (Raw) Messages” chapter.

### Android Sample Codes

Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA\_TAG);

IsoDep isoDep = IsoDep.get(tag);

// Connect to the remote NFC device

isoDep.connect();

isoDep.setTimeout(2000);

byte[] responseResult = isoDep.transceive(U2FAID);

int responseLen = responseResult.length;

if (responseResult[responseLen-2]==(byte)0x90

&& responseResult[responseLen-1]==(byte) 0x00) {

// Send request

byte[] registrationReq = buildRegistrationRequest();

responseResult = isoDep.transceive(registrationReq);

responseLen = responseResult.length;

decodeRegisterResponse(responseResult, responseLen);

}

## BT / BLE Tokens

BT and BLE are long-range wireless protocols and thus have several implications for privacy, security, and overall user-experience. Because they are wireless, BT and BLE may be subject to monitoring, injection, and other network-level attacks.

Probably make sense to implement U2F Client application on Android smartphone, iPhone, and Window PC for U2F BT/BLE Token.

### BT/BLE Protocol Layers

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Application | Headset Hand-Free | AT Command | vCalendar |  | PC Dialer | PC Browser |  | … |  |
| Application Profiles | Advanced Audio Distribution Profile (A2DP) | Serial Port Profile (SPP) | Object Exchange Profile (OBEX) |  | Telephony APIs (TAPI) | Personal Network Profile (PAN) | Generic Access Profile (GAP) | … |  |
| Application Protocols | Audio | Radio Frequency Communication (RFCOMM) | | Service Discovery Protocol (SDP) | Telephony Control Protocol | BT Network Encapsulation Protocol (BNEP) | Other Link Control Protocol (LLC) | | Control |
| Transport | Logical Link Control and Adaptation Protocol (L2CAP) | | | | | | |
| Link | Link Manager Protocol (LMP) | | | | | | |
| Physical | Basic Rate/Enhanced Data Rate (BR/EDR) | | | | | | | | |

**Bluetooth Classic Protocol Layers**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Application | U2F Client | Blood Pressure Monitor | Body Composition Monitor | … |  |  |
| Application Profiles | Generic Attribute Profile (GATT) | Health Device Profile (HDP) over GATT | | | All Profiles if Encryption is needed | Generic Access Profile (GAP) |
| Application Protocols | Attribute Protocol (ATT) | | | | Secured Manager Protocol (SMP) | Other Link Control Protocol (LLC) |
| Transport | Logical Link Control and Adaptation Protocol (L2CAP) | | | | | |
| Link | Link Layer (LL) | | | | | |
| Physical | Bluetooth Low Energy Physical Layer (BLE PHY) | | | | | |

**Bluetooth Low Energy Protocol Layers**

### Device Discovery – Bluetooth Classic (BT)

Pairing is the only mechanism defined in this protocol to ensure that FIDO Clients are interacting with the expected BT/BLE Authenticator.

For Bluetooth Classic (BT) devices, the Inquiry (Discovering) and Page (Connecting) procedures are used to complete the BT paring. On a special physical channel, a Bluetooth inquiring device that tries to find other nearby devices will actively send inquiry requests. Bluetooth discoverable devices that are available to be found listen for these inquiry requests will send responses back. An Extended Inquiry Response can be used to provide more information and, in addition, the SDP service search procedures can be used to ensure the discoverable device is indeed the connection target. The Paging procedure for forming connections is asymmetrical and requires that one Bluetooth device carries out the page (connection) procedure while the other Bluetooth device is connectable (page scanning). The procedure is targeted, so that the page procedure is only responded to by one specified Bluetooth device.

#### Link / Network Establishment

***Paring:***

1. Following the BT discovering procedures, if the Client and Authenticator are not yet bonded, the Authenticator becomes discoverable (enters Discoverable Mode). It SHOULD include a distinctive device name in the Extended Inquiry Response (EIR) packet during the Inquiry procedure. An Authenticator SHALL only allow connections from new Clients while in this mode.
2. Once confirmed it is a U2F Authenticator, the client will connect to Authenticator. If not already paired before, Client and Authenticator perform BT bonding to create a link key and connect. Authenticator SHALL only allow connections from previously bonded Clients without user intervention.
3. Client performs service discovery procedure on the Authenticator.

***Service Discovery Protocol:***

The U2F Authenticator SHALL contain a Service Discovery Protocol (SDP) record with the following data:

uint8 fido\_client\_spp\_sdprecord [] =

{

0x09, 0x00, 0x01, */\* ServiceClassIDList(0x0001) \*/*

0x35, 0x11, */\* DataElSeq 17 bytes \*/*

0xT, 0xB, 0xD, 0xT, 0xB, 0xD, 0xT, 0xB, 0xD, 0xT, 0xB, 0xT, 0xB, 0xD, 0xT, 0xB, 0xD, */\*UUID TBD\*/*

0x09, 0x00, 0x04, */\* ProtocolDescriptorList(0x0004) \*/*

0x35, 0x0c, */\* DataElSeq 12 bytes \*/*

0x35, 0x03, */\* DataElSeq 3 bytes \*/*

0x19, 0x01, 0x00, */\* UUID L2CAP(0x0100) \*/*

0x35, 0x05, */\* DataElSeq 5 bytes \*/*

0x19, 0x00, 0x03, */\* UUID RFCOMM(0x0003) \*/*

0x08, 0x00,

*/\* uint8 0x00 - Change 0x00 to actual RFCOMM Channel Number \*/*

0x09, 0x00, 0x06, */\* LanguageBaseAttributeIDList(0x0006) \*/*

0x35, 0x09, */\* DataElSeq 9 bytes \*/*

0x09, 0x65, 0x6e, */\* uint16 0x656e \*/*

0x09, 0x00, 0x6a, */\* uint16 0x006a \*/*

0x09, 0x01, 0x00, */\* uint16 0x0100 \*/*

0x09, 0x01, 0x00,

*/\* ServiceName(0x0100) = "U2FAUTHDEVICE" \*/*

0x25, 0x0D, */\* String length 13 \*/*

'U','2','F','A','U','T','H','D','E','V','I','C','E'

};

#### Link/Network – Encryption

For link security concerns, 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. ***For BT connections***, during phase 4 of the Secure Simple Pairing, the Authenticator SHOULD generate Mode 4 (or better) link key. Encryption SHALL be enabled using a key size of 16 bytes before any U2F messages are sent.

### Data Transmission – Bluetooth Classic (BT)

If one or both of the Authenticator and Client only supports Classic Bluetooth, they SHALL communicate over RFCOMM (on top of L2CAP) which is a serial port emulation protocol.

#### Transport – Framing

For BT through RFCOMM, all requests and their responses are conceptually written as a single frame.

#### Transport – Encapsulation

The format of the request and response frame has encapsulation described below.

Request frame format:

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Length** | **Mnemonic** | **Description** |
| 0 | 1 | CMD | Command identifier (bit 7 always set) |
| 1 | 1 | HLEN | High part of payload length |
| 2 | 1 | LLEN | Low part of payload length |
| 3 | s | DATA | Data with s length |

Response frame format:

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Length** | **Mnemonic** | **Description** |
| 0 | 1 | STAT | Response status |
| 1 | 1 | HLEN | High part of payload length |
| 2 | 1 | LLEN | Low part of payload length |
| 3 | s | DATA | Data with s length |

When the ‘STAT’ byte in the response is the same as the request command byte, 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.

Command and Status Constants: Error Constants:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constant** | **Value** |  | **Constant** | **Value** |
| PING | 0x81 |  | ERR\_INVALID\_CMD | 0x01 |
| MSG | 0x83 |  | ERR\_INVALID\_LEN | 0x02 |
| ERROR | 0xBF |  | ERR\_INVALID\_SEQ | 0x03 |
|  |  |  | ERR\_MSG\_TIMEOUT | 0x04 |
|  |  |  | ERR\_OTHER | 0x7F |

#### Transport – Fragmentation

For RFCOMM, No fragmentation is supported as communication over RFCOMM transport should be able to handle all messages without fragmentation.

### Device Discovery – Bluetooth Low Energy (BLE)

For Bluetooth Low Energy (BLE) devices, the Discovering (Advertising and Scanning) procedures and Connecting Procedure are used to complete the BLE paring. An advertiser uses the advertising procedure to perform unidirectional broadcasts to devices in the area using the advertising physical channel. The unidirectional broadcast occurs without a connection between the advertising device and the listening devices. On the other hand, a scanning device uses the scanning procedure to listen for unidirectional broadcasts of user data from advertising devices on the advertising physical channel. A scanning device can request additional user data from an advertising device by making a scan request over the advertising physical channel. The advertising device responds to these requests with additional user data sent to the scanning device over the same advertising physical channel. After receiving connectable advertising events from the targeted advertising device, the scanning device can initiate a connection by sending the connection request to the targeted advertising device over the advertising broadcast physical channel.

#### Link / Network Establishment – Bluetooth Low Energy (BLE)

***Paring:***

1. Authenticator advertises (optionally including the FIDO U2F service UUID in the advertisement).
2. Client performs a scan and service discovery looking for the FIDO U2F service.
3. Client performs characteristic discovery on the Authenticator.
4. If not already paired, the Client and Authenticator SHALL perform BLE pairing and create a LTK. Authenticator SHALL only allow connections from previously bonded Clients without user intervention.

#### Link/Network - Encryption

For link security concerns, 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. ***For BLE connections***, the Authenticator SHALL enforce Security Mode 1, Level 2 (unauthenticated pairing with encryption) during phase 2 of the Secure Simple Pairing before any U2F messages are exchanged.

### Data Transmission – Bluetooth Low Energy (BLE)

If both Authenticator and Client are single mode BLE devices, they SHALL communicate using GATT over L2CAP on the LE physical connection. If both Authenticator and Client are dual mode devices (BT & BLE), they SHALL communicate using GATT over L2CAP on the BR/EDR connection.

#### 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 graphic illustrates the mandatory services and characteristics that SHALL be offered by a FIDO Authenticator as part of its GATT server:

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

**U2F Status**

**U2F Control Point**

**U2F Authenticator GATT Server**

U2F Service

Min. Security:

Mode 1, Level 2

***Write***

***Notify***

***Read***

***Read***

Device Information Service

***Read***

***Read***

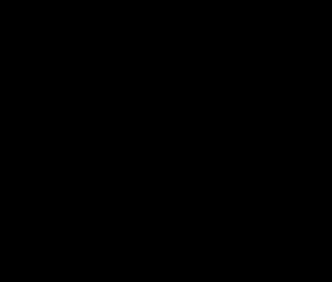
***Read***

**U2F Control Point Length**

**U2F Service Revision**

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

**Manufacturer Name String**



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

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

**Model Number String**

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 notmentioned explicitly are optional.

**Firmware Revision String**

##### U2F Service

An Authenticator SHALL implement the U2F Service described below. The UUID for the FIDO U2F GATT service is TBD (it is being standardized by the Bluetooth SIG). The service contains the following characteristics:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Characteristic Name** | **Mnempnic** | **Property** | **Length** | **UUID** |
| U2F Control Point | u2fControlPoint | Write | Defined by Vendor  (20-512 bytes) | TBD |
| U2F Status | u2fStatus | Notify | Defined by Vendor  (20-512 bytes) | TBD |
| U2F Control Point Length | u2fControlPointLength | Read | 2 bytes | TBD |
| U2F Service Revision | u2fServiceRevision | Read | 8 bytes | 0x2A28 |

u2fControlPoint is a write-only command buffer.

u2fStatus is a notify-only response attribute.

u2fControlPointLength defines the size of 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 this version of the specification, the value u2fServiceRevision SHALL be 1.0 or in raw bytes: 0x312e30.

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

##### Device Information Service

An Authenticator SHALL implement the Device Information Service [BTDIS] 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.

##### U2F Request and Response

Clients SHOULD make requests by connecting to the Authenticator and performing a write into the u2fControlPoint characteristic. 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, etc.) Upon receiving a valid request (containing a known key handle), the Authenticators MAY alert the user and MAY use a short delay before sending a response. In this case, before the user has completed the test of user presence, the Authenticator SHALL respond with

SW\_CONDITIONS\_NOT\_SATISFIED. Upon receiving an SW\_CONDITIONS\_NOT\_SATISFIED message, the Client SHALL assume the Authenticator is still processing the command; the Client SHALL not resend the command. Until a timeout occurs, the Client SHALL NOT move on to other devices when it receives a SW\_CONDITIONS\_NOT\_SATISFIED, as it knows this is a device that can satisfy its request.

#### Transport – Framing

For BLE token, characteristics/attributes in the GATT service are used for packet framing.

1. Client reads the u2fControlPointLength characteristic.
2. Client registers for notifications on the u2fStatus characteristic.
3. Client writes a request (*e.g.*, an enroll request) into the u2fControlPoint characteristic.
4. Authenticator evaluates the request and responds by sending notifications over u2fStatus characteristic.

#### Transport – Encapsulation

The format of the request and response frame has the similar encapsulation as Bluetooth Classic. However, due to size constraints for the u2fControlPoint characteristic (request) and u2fStatus characteristic (response), initialization and continuation fragment packets are defined below. 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. The first maxLen - 3 bytes of data then follow.

Initialization fragment format:

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Length** | **Mnemonic** | **Description** |
| 0 | 1 | CMD/STAT | Command identifier (bit 7 always set) |
| 1 | 1 | HLEN | High part of data length |
| 2 | 1 | LLEN | Low part of data length |
| 3 | maxLen - 3 | DATA | Data |

Where maxLen is the maximum packet size supported be the characteristic or notification.

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. The first maxLen - 3 bytes of data then follow.

Continuation fragment format:

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Length** | **Mnemonic** | **Description** |
| 0 | 1 | SEQ | Response status |
| 1 | maxLen - 1 | DATA | Data |

Both Request (CMD) and Response (STAT) messages will be encapsulated with Initialization and Continuation fragment frames. When the ‘STAT’ byte in the response is the same as the request command byte, 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.

Command and Status Constants: Error Constants:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constant** | **Value** |  | **Constant** | **Value** |
| PING | 0x81 |  | ERR\_INVALID\_CMD | 0x01 |
| MSG | 0x83 |  | ERR\_INVALID\_LEN | 0x02 |
| ERROR | 0xBF |  | ERR\_INVALID\_SEQ | 0x03 |
|  |  |  | ERR\_MSG\_TIMEOUT | 0x04 |
|  |  |  | ERR\_OTHER | 0x7F |

#### Transport – Fragmentation

Fragmentation is performed using initialization and continue fragment packets described above. A single request/response sent over BLE MAY be split over multiple writes and notifications, due to the inherent limitations of BLE 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.

### Session

There will be no Session layer for BT/BLE token.

### Application Messages

Formation of the application Registration/Authentication Request and Response messages is described in “U2F Client Application (Raw) Messages” chapter.

U2F application message data format is defined in previous chapter. For BT/BLE tokens, note that as all communication SHALL be done using extended length APDU format; therefore, there is no fragmentation needed in application layer using APDU Chaining.