UAF Application API and Transport Binding Specification

Editors:

Brad Hill, PayPal Inc.

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

Contributors:

Davit Baghdasaryan, Nok Nok Labs, Inc.

Abstract:

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

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

35

1 Notation

Type names, attribute names and element names are written in *italics*.

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

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

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

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

1.1 Key Words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

1.2 Revision History

Revision	Date	Author	Summary
0.2	12/31/13	Brad Hill	First ODT version

2 Overview

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

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

2.1 Audience

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

2.2 Scope

This document describes:

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

The following are out of scope for this document:

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

The goal of describing standard APIs and an interoperability profile for the transport of UAF messages here is to provide an example of how to develop a FIDO-enabled application and to promote the ease of integrating interoperable layers from different vendors

to build a complete FIDO UAF solution. For any given application instance, these particular patterns may not be ideal and are not mandatory. Applications may use alternate transports, bundle UAF Protocol messages with other network data, or discover and utilize alternative APIs as they see fit.

2.3 Architecture

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

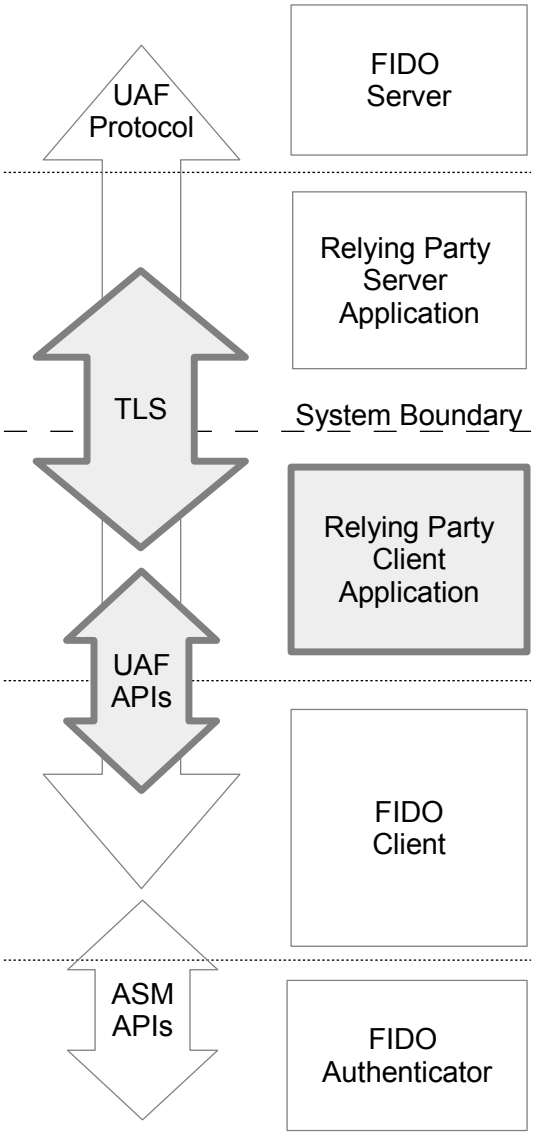

Illustration 1: UAF Architecture

89 This document describes the shaded components in Illustration 1.

90 2.4 Protocol Conversation

91 The core UAF protocol consists of five conceptual phases.

- **Discovery** allows the relying party server to determine the availability of FIDO capabilities at the client, including metadata about the available authenticators.
- **Registration** allows the client to generate and associate new key material with an account at the relying party server, subject to policy set by the server and acceptable attestation that the authenticator and registration matches that policy.
- **Authentication** allows a user to provide an account identifier, proof-of-possession of previously registered key material associated with that identifier, and potentially other attested data, to the relying party server.
- **Transaction Confirmation** allows a server to request that a FIDO client and authenticator with the appropriate capabilities display some information to the user, request that the user authenticate locally to their FIDO authenticator to confirm it, and provide proof-of-possession of previously registered key material and an attestation of the confirmation back to the relying party server.
- **Deregistration** allows a relying party server to tell an authenticator to forget selected locally managed key material associated with that relying party in case such keys are no longer considered valid by the relying party.

Discovery does not involve a protocol exchange with the FIDO Server, although the information available through the Discovery APIs might be communicated back to the server in an application-specific manner, for example, as part of obtaining a UAF Protocol Request Message, in order to receive as part of that message an authenticator policy tailored to the specific capabilities of the FIDO User Device.

Although this UAF Protocol abstractly defines the FIDO Server as the initiator of requests, for client applications utilizing UAF as described in this document will always transport UAF Protocol messages over a client-initiated request/response protocol like HTTP.

The protocol flow from the point of view of the relying party client application for *Registration, Authentication,* and *Transaction Confirmation* is as follows:

1. The client application either explicitly contacts the server to obtain a UAF Protocol Request Message, or this message is delivered in-line with other client application content.
2. The client application invokes the appropriate API to pass the UAF Protocol Request Message asynchronously to the FIDO Client, and receives a set of callbacks.
3. The FIDO Client performs any necessary interactions with the user and authenticator(s) to complete the request and uses a callback to either notify the client application of an error or return a UAF Response Message.
4. The client application delivers the UAF Response Message back to the server over a protocol such as HTTP.
5. The server (optionally) returns an indication of the results of the operation and additional data such as authorization tokens or a redirect.

6. The client application (optionally) uses the appropriate API to inform the FIDO Client of the results of the operation. This allows the FIDO Client to perform "housekeeping" tasks for a better user experience, e.g. by not attempting to use again later a key that the server refused to register.
7. The client application (optionally) processes additional data returned to it in an application-specific manner, e.g. processing new authorization tokens, redirecting the user to a new resource or interpreting an error code to determine if and how it should retry a failed operation.

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

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

3 Common Definitions

These elements are shared by several APIs and layers.

3.1 UAF Status Codes

This table lists UAF protocol status codes. These indicate the result of the UAF operation at the FIDO Server. They do not represent the HTTP [RFC2616] or other transport layers. These codes are intended for consumption by both the client-side web app and FIDO Client to inform application-specific error reporting, retry and housekeeping behavior.

Table 1

Code	Meaning
1200	OK. Operation completed
1202	Accepted. Message accepted, but not completed at this time. The RP may need time to process the attestation, run risk scoring, etc. The server SHOULD NOT send an authenticationToken with a 1202 response
1400	Bad Request. The server did not understand the message
1401	Unauthorized. The userid must be authenticated to perform this operation, or this KeyID is not associated with this UserID.
1403	Forbidden. The userid is not allowed to perform this operation. Client SHOULD NOT retry
1404	Not Found.
1408	Request Timeout.
1480	Unknown AAID. The server was unable to locate authoritative metadata for the AAID [UAFAuthnrMetadata].
1481	Unknown KeyID. The server was unable to locate a registration for the given UserID and KeyID combination.
1490	Channel Binding Refused. The server refused to service the request due to a missing or mismatched channel binding(s).
1491	Request Invalid. The server refused to service the request because the request message nonce was unknown, expired or the server has previously serviced a message with the same nonce and user ID.
1492	Unacceptable Authenticator. The authenticator is not acceptable according to the server's policy, for example because the capability reg-

	istry used by the server reported different capabilities than client-side discovery.
1493	Revoked Authenticator. The authenticator is considered revoked by the server.
1494	Unacceptable Key. The key used is unacceptable. Perhaps it is on a list of known weak keys or uses insecure parameter choices.
1495	Unacceptable Algorithm. The server believes the authenticator to be capable of using a stronger mutually-agreeable algorithm than was presented in the request.
1496	Unacceptable Attestation. The attestation(s) provided were not accepted by the server.
1497	Unacceptable Client Capabilities. The server was unable or unwilling to use required capabilities provided supplementally to the authenticator by the client software.
1498	Unacceptable Content. There was a problem with the contents of the message and the server was unwilling or unable to process it.
1500	Internal Server Error

158

4 DOM API

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

4.1 Feature Detection

FIDO's UAF DOM APIs are rooted in a new *uaf* object, a property of a new *fido* object, which is itself a property of the *window.navigator* object; the existence and properties of which can be used for feature detection.

Example 1: Feature Detection of UAF APIs

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

4.2 UAFMessage Dictionary

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

```
Dictionary UAFMessage {
        DOMString uafProtocolMessage;
        Object    additionalData;
}
```

uafProtcolMessage of type *DOMString*

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

additionalData of type *Object*

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

4.3 UAFResponseCallback

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

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

4.3.1 Arguments

uafResponse of type *UAFMessage*

The UAFMessage and any additional data representing the FIDO Client's response to the server's request message.

4.4 ErrorCallback

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

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

  interface ErrorCode {
    const short NO_ERROR                = 0x0;
    const short WAIT_USER_ACTION        = 0x1;
    const short INSECURE_TRANSPORT      = 0x2;
    const short USER_CANCELLED          = 0x3;
    const short UNSUPPORTED_VERSION     = 0x4;
    const short NO_SUITABLE_AUTHENTICATOR = 0x5;
    const short PROTOCOL_ERROR          = 0x6;
    const short UNTRUSTED_FACET_ID      = 0x7;
    const short UNKNOWN                 = 0xFF;
  }
```

4.4.1 ErrorCode Values

NO_ERROR

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

WAIT_USER_ACTION

Waiting on user action to proceed. (e.g. selecting an authenticator in the FIDO client user interface, performing user verification, or completing an enrollment step with an authenticator)

USER_CANCELLED

The user declined any necessary part of the interaction to complete the registration.

UNSUPPORTED_VERSION

The *UAFMessage* does not specify a protocol version supported by this FIDO Client.

NO_SUITABLE_AUTHENTICATOR

No authenticator matching the AuthenticatorPolicy internal to the UAFProtocolMessage is available to service the request, or the user declined to consent to the use of a suitable authenticator.

INSECURE_TRANSPORT

window.location.protocol is not https or the DOM contains insecure mixed content.

PROTOCOL_ERROR

A violation of the UAF protocol occurred. The message may have timed out, the origin associated with the message may not match the origin of the calling DOM context, or the protocol message may be malformed or tampered with.

UNTRUSTED_FACET_ID

The FIDO Client declined to process the operation because the caller's calculated Facet ID was not found in the trusted list for the Application ID in the UAF Request message.

UNKNOWN

An error condition not described by other codes.

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

4.5 notifyUAFResult Operation

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

If and how a UAF status code is delivered by the server is application and transport specific. A non-normative example can be found below in the HTTPS Transport Interoperability Profile. [6.3]

```
void notifyUAFResult(int responseCode, DOMString uafResponse);
```

4.5.1 Arguments

responseCode of type int

The *uafResult* field of a *ServerResponse*.

uafResponse of type *DOMString*

The UAF response message to which this *responseCode* applies.

4.6 Version Interface

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

```
Interface Version {
        readonly attribute int majorVersion;
        readonly attribute int minorVersion;
}
```

4.7 Authenticator Interface

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

```
interface Authenticator {

    readonly attribute DOMString AAID;
    readonly attribute DOMString description;
    readonly attribute DOMString logo;

    readonly attribute Version[] supportedUAFVersions;

    readonly attribute long     userVerification;
    readonly attribute long     keyProtection;
    readonly attribute long     attachmentHint;
    readonly attribute long     secureDisplay;

    readonly attribute int      authenticationAlgorithm;
    readonly attribute DOMString assertionScheme;

```

```
    // for future use
    readonly attribute long additionalInfo;


    // See FIDO UAF Registry of Predefined Values for constant definitions

}
```

4.7.1 Constants

A number of constants are defined for use with the bit flag fields *userVerification*, *keyProtection*, *attachmentHint*, and *secureDisplay*. To avoid duplication and inconsistencies, please refer to the authoritative definitions found in the FIDO UAF Registry of Predefined Values [FIDORegistry].

4.7.2 Attributes

AAID of type *DOMString*, *readonly*

The Authenticator Attestation ID, which identifies the type and batch of the authenticator.

No exceptions.

description of type *DOMString*, *readonly*

A user-friendly description string for the authenticator.

No exceptions.

logo of type *DOMString*, *readonly*

A PNG [PNG] logo for the authenticator, encoded as a data: url [RFC2397].

No exceptions.

userVerification of type *long*, *readonly*

A set of bit flags indicating the user verification methods(s) supported by the authenticator. The values are defined by the USER_VERIFY_ constants.

No exceptions.

keyProtection of type *long*, *readonly*

A set of bit flags indicating the key protection used by the authenticator. The values are defined by the KEY_PROTECTION constants.

No exceptions.

attachmentHint of type *long*, *readonly*

A set of bit flags indicating how the authenticator is currently connected to the system hosting the FIDO Client software. The values are defined by the ATTACHMENT_HINT constants.

Because the connection state and topology of an authenticator may be transient, these values are only hints that can be used by server-supplied policy to guide the user experience, e.g. to prefer a device that is connected and ready for authenticating or confirming a low-value transaction, rather than one that is more secure but requires more user effort. These values are not reflected in authenticator metadata [UAFAuthnrMetadata] and cannot be relied on by the relying party, although some models of authenticator may provide attested measurements with similar semantics as part of UAF protocol messages.

No exceptions.

secureDisplay of type *long*, *readonly*

A set of bit flags indicating the availability and type of secure display. The values are defined by the SECURE_DISPLAY_ constants.

No exceptions.

supportedUAFVersions of type Version*[]*, *readonly*

Indicates the UAF Protocol Versions supported by the authenticator.

No exceptions.

authenticationAlgorithm of type *int*, *readonly*

Supported Authentication Algorithm. Value MUST be related to constants with prefix UAF_ALG_SIGN in the UAF Registry of Predefined Values. [FIDORegistry]

No exceptions.

assertionScheme of type *DOMString*, *readonly*

The encoding scheme the authenticator uses for attested data and signatures.

Scheme identifiers are defined in the UAF Registry of Predefined Values. [FIDORegistry]

No exceptions.

additionalInfo of type *long*, *readonly*

RESERVED FOR FUTURE USE

No exceptions.

4.8 Discovery Interface

To discover if the user's client software and devices support UAF and if Authenticator capabilities are available that it may be willing to accept for authentication, Relying Party code in the browser can use the following interface.

```
interface Discovery {
   readonly attribute Version[] supportedUAFVersions;
   readonly attribute DOMString clientVendor;
   readonly attribute Version clientVersion;
   readonly attribute Authenticator[] availableAuthenticators;
   void checkPolicy(DOMString message, ErrorCallback cb);
}
```

4.8.1 Attributes

supportedUAFVersions of type Version*[]*, *readonly*

A list of the FIDO UAF protocol versions supported by the client, most-preferred first.

No exceptions.

clientVendor of type *DOMString*, *readonly*

The vendor of the FIDO UAF Client.

No exceptions.

clientVersion of type *Version readonly*

The version of the FIDO UAF Client, ordered by version number significance. This is a vendor-specific version for the client software, not a UAF version.

No exceptions.

availableAuthenticators of type *Authenticator[]*, *readonly*

An array containing Authenticator dictionaries describing the available UAF authenticators. The order is not significant.

No exceptions.

4.8.2 Operations

checkPolicy(DOMString message, ErrorCallback cb) of return type *void*

Ask the FIDO plugin if it would be able to process the supplied UAF Request message, without prompting the user. Unlike other operations using an ErrorCallback, this operation MUST always trigger the callback and return NO_ERROR if it believes that the message can be processed and a suitable authenticator matching the embedded policy is available, or the appropriate ErrorCode value

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

4.8.2.1 Arguments

message of type DOMString

A UAF Request Message containing the policy and operation to be tested.

4.8.3 Privacy Considerations

This section is non-normative.

Differences in the FIDO capabilities on a user device may (among many other characteristics) allow a server to "fingerprint" a remote client and attempt to persistently identify it, even in the absence of any explicit session state maintenance mechanism. Although it may contribute some amount of signal to servers attempting to fingerprint clients, the attributes exposed by the Discovery API are designed to have a large anonymity set size and should present little or no qualitatively new privacy risk. Nonetheless, an unusual configuration of FIDO Authenticators may be sufficient to uniquely identify a user. It is recommended that user agents expose the Discovery API to all applications without requiring explicit user consent by default, but user agents or FIDO Client implementers should provide users with the means to opt-out of discovery if they wish to do so for privacy reasons.

4.9 FIDOClient Interface

The FIDOClient interface allows the application to send UAF request messages to the FIDO Client asynchronously and receive UAF response messages back.

```
interface FIDOClient {
   void processUAFOperation(UAFMessage          message,
                            UAFResponseCallback completionCallback,
                            ErrorCallback       errorCallback);
}
```

4.9.1 Operations

processUAFMessage(UAFMessage message, UAFResponseCallback completionCallback, ErrorCallback errorCallback) of return type *void*

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

No exceptions.

4.9.1.1 Arguments

message of type *UAFMessage*

The *UAFMessage* to be used by the FIDO client software.

completionCallback of type *UAFResponseCallback*

The callback that receives the client response *UAFMessage* from the FIDO Client, to be delivered to the Relying Party server.

errorCallback of type *ErrorCallback*

A callback function to receive error and progress events from the FIDO Client.

4.10 Security Considerations for the DOM API

4.10.1 Insecure Mixed Content

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

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

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

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

1. UAF messages SHOULD NOT be transported using methods that opt-out of the Same Origin Policy [SOP], for example, using <script src="*url*"> to non-Same-Origin URLs or by setting Access-Control-Allow-Origin headers at the server.

2. When transporting UAF messages using XMLHttpRequest [XHR] the client SHOULD NOT follow redirects that are not to URLs within the same origin.
3. UAF messages SHOULD NOT be exposed in HTTP responses where the entire response body parses as valid ECMAScript. Resources exposed in this manner may be subject to unauthorized interactions by hostile applications hosted at untrusted origins through cross-origin embedding using <script src="*url*">.
4. Web applications SHOULD NOT share UAF messages across origins through channels like postMessage() [WEBMESSAGING].

4.10.3 Implementation Notes for Browser/Plugin Authors

This section is non-normative.

Web applications utilizing UAF depend on services from the web browser as a trusted platform. The APIs for web applications do not provide a means to assert an origin as an application identity for the purposes of FIDO operations as this will be provided to the FIDO Client by the browser based on its privileged understanding of the actual origin context. The browser MUST enforce that the web origin communicated to the FIDO Client as the application identity is accurate and that resource instances with insecure mixed-content cannot utilize the FIDO ECMAScript APIs.

5 Android API

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

As with web applications, a variety of integration patterns are possible on the Android platform. The API described here allows an app to discover a shared FIDO Client on the user device, implemented as a Bound Service, and communicate with it using an ADIL interface.

The interfaces are defined in the *org.fidoalliance.uaf.client* package.

5.1 IUAFClient.aidl

IUAFClient represents the primary interface for interacting with an Android device's shared FIDO Client installation. It provides methods to get a reference to the Discovery interface, to process UAF messages, and to notify the FIDO Client of the result of UAF Protocol Response messages delivered to the remote server. The operations are as described in the DOM API, with the exception of the *checkPolicy, origin*, and *channel-Bindings* parameters.

```
package org.fidoalliance.uaf.client;

import org.fidoalliance.uaf.client.IUAFErrorCallback;
import org.fidoalliance.uaf.client.IUAFResponseCallback;
import org.fidoalliance.uaf.client.Discovery;
import org.fidoalliance.uaf.client.UAFMessage;

import java.util.Map;

interface IUAFClient
{
        Discovery getDiscovery();

        void notifyUAFResult(in int    responseCode,
                             in String  uafResponse);

        void processUAFMessage(in UAFMessage                msg,
                               in String                    origin,
                               in Map                       channelBindings,
                        in boolean              checkPolicy,
                        in IUAFResponseCallback     cb,
                               in IUAFErrorCallback         errorCb);
}
```

Setting the *checkPolicy* flag to **true** on a call emulates the behavior of the *checkPolicy* operation in the DOM API.

5.1.1 channelBindings

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

The *channelBindings* data structure is *Map<String,String>* with the keys as defined for the *TLSData* structure in the UAF Protocol Specification. [UAFProtocol]

The use of channel bindings for TLS helps assure the server that the channel over which UAF protocol messages are transported is the same channel the legitimate client is using and that messages have not been forwarded through a malicious party. UAF defines support for the *tls-unique* and *tls-server-endpoint* bindings from [RFC5929], as well as server certificate and ChannelID [CHANNELID] bindings. The client SHOULD supply all channel binding information available to it. Failure to supply appropriate channel binding information MAY cause a Relying Party server to reject a UAF transaction.

5.1.2 origin

Android apps using *IUAFClient* to request services from the FIDO Client can do so in one of two ways. They can do so under their own identity, by setting *origin* to **null**, or they can act as the user's agent on behalf of multiple relying party applications, by setting *origin* to the RFC6454 [RFC6454] serialization of the remote server's Origin.

An application that is operating on behalf of a single entity SHOULD always set *origin* to **null**. This will cause the FIDO Client to determine the caller's identity as *android:apk-key-hash:<hash-of-public-key>*. The FIDO Client will then compare this with the list of authorized application facets and proceed if it is listed as trusted.

For example, if the application at "www.example.com" is exposed as both a browser-based web application and through an Android app, the Android app should not set origin, rather "www.example.com" should list the Android app's facet identity as trusted.

An application may access registrations made by web browsers and other applications on the system for the same target relying party application. Continuing the example. If the Android app is listed as a trusted facet, it may use a registration for "www.example.com" that was originally made in a web browser on the same system with the same FIDO client.

If an App accesses multiple logical applications that are still controlled by either a single or a constrained set of entities (e.g. "app1.example.com" and "app2.example.com") it SHOULD still set *origin* to **null** and use a federation pattern to accomplish single-sign on.

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

If the application is explicitly intended to operate as the user's agent in the context of an arbitrary number of remote applications (as in a web browser) it may set *origin* to the RFC6454 [RFC6454] Unicode serialization of the remote application's Origin. The application MUST satisfy the necessary conditions described in the Transport Security Requirements [6.1] for authenticating the remote server before setting *origin*.

Use of the *origin* parameter requires the application to declare the *org.fidoalliance.uaf.permisisons.ACT_AS_WEB_BROWSER* permission, and the FIDO Client MUST verify that the calling application has this permission before processing the operation.

5.1.2.1 org.fidialliance.uaf.permissions.ACT_AS_WEB_BROWSER

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

5.2 IUAFErrorCallback.aidl

This interface defines the callback for the FIDO Client to return error information to the application. It may be called multiple times, for example, with the *WAIT_USER_ACTION* status code.

```
package org.fidoalliance.uaf.client;

interface IUAFErrorCallback
{
        void response(long code);
}
```

5.3 IUAFResponseCallback.aidl

This interface defines the callback for returning a UAF Protocol response message in the event that a *processUAFMessage* call is completed successfully. If the *checkPolicy* flag was set to *false* this callback will never be called; only the *IUAFErrorCallback* will be called.

```
package org.fidoalliance.uaf.client;

import org.fidoalliance.uaf.client.UAFMessage;

interface IUAFResponseCallback
{
        void response(in UAFMessage uafResponse);
```

```
}
```

5.4 UAFMesage.aidl

AIDL wrapper for *UAFMessage.java*.

```
package org.fidoalliance.uaf.client;

parcelable UAFMessage;
```

5.5 UAFMessage.java

This structure represents the data type for communications between the client application in both directions, to the FIDO Client, and the FIDO Server. It wraps the actual UAF Protocol message while providing *additionalData* to hold application-specific information or protocol extensions. Modification of *uafProtocolMessage* by the client application may invalidate the message. A client application MAY examine the contents of a message, for example, to determine if a message is still fresh. Details of the structure of the message can be found in the UAF Protocol Specification.

The *additionalData* parameter is a *String* and may be used to exchange additional data between the Relying Party application and the FIDO Client. Use of this field is OPTIONAL and vendor-specific but a JSON [ECMA-404] structure is RECOMMENDED.

```
package org.fidoalliance.uaf.client;

import android.os.Parcel;
import android.os.Parcelable;

public class UAFMessage implements Parcelable {

    public String   uafProtocolMessage;
    public String   additionalData;

    public static final Parcelable.Creator<UAFMessage> CREATOR
            = new Parcelable.Creator<UAFMessage>() {
        public UAFMessage createFromParcel(Parcel in) {
            return new UAFMessage(in);
        }

        public UAFMessage[] newArray(int size) {
            return new UAFMessage[size];
        }
    };

    public UAFMessage() {
    }
```

```
    private UAFMessage(Parcel in) {
        uafProtocolMessage     = in.readString();
        additionalData   = in.readString();
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(uafProtocolMessage);
        dest.writeString(additionalData);
    }
}
```

5.6 Version.aidl

AIDL wrapper for *Version.java.*

```
package org.fidoalliance.uaf.client;

parcelable Version;
```

5.7 Version.java

A class describing a major and minor version number.

```
package org.fidoalliance.uaf.client;

import java.util.ArrayList;
import java.util.List;

import android.os.Parcel;
import android.os.Parcelable;

public class Version implements Parcelable {

    public int majorVersion;
    public int minorVersion;

    public static final Parcelable.Creator<Version> CREATOR
        = new Parcelable.Creator<Version>() {
        public Discovery createFromParcel(Parcel in) {
            return new Version(in);
        }

        public Version[] newArray(int size) {
            return new Version[size];
```

```
        }
    };

    public Version() {
    }

    private Version(Parcel in) {
        majorVersion = in.readInt();
        minorVersion = in.readInt();
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(majorVersion);
        dest.writeInt(minorVersion);
    }
}
```

5.8 Discovery.aidl

AIDL wrapper for *Discovery.java*.

```
package org.fidoalliance.uaf.client;

parcelable Discovery;
```

5.9 Discovery.java

This class describes the Discovery interface that allows the application to query the available FIDO Client and UAF authenticators available on the FIDO user device. For field and method definitions, see the analogous attribute and operation descriptions for the DOM API. [4.8]

```
package org.fidoalliance.uaf.client;

import java.util.ArrayList;
import java.util.List;

import android.os.Parcel;
import android.os.Parcelable;

public class Discovery implements Parcelable {

```

```
    public List<Version> supportedUAFVersions =
                            new ArrayList<Version>();
    public String    clientVendor;
    public Version   clientVersion;
    public List<Authenticator> availableAuthenticators =
                            new ArrayList<Authenticator>();

    public static final Parcelable.Creator<Discovery> CREATOR
            = new Parcelable.Creator<Discovery>() {
        public Discovery createFromParcel(Parcel in) {
            return new Discovery(in);
        }

        public Discovery[] newArray(int size) {
            return new Discovery[size];
        }
    };

    public Discovery() {
    }

    private Discovery(Parcel in) {
        in.readTypedList(supportedUAFVersions,
                                    Version.CREATOR);
        clientVendor = in.readString();
        clientVersion = in.readParcellable(null);
        in.readTypedList(availableAuthenticators,
                                    Authenticator.CREATOR);
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeTypedList(version);
        dest.writeString(clientVendor);
        dest.writeParcelable(clientVersion, 0);
        dest.writeTypedList(availableAuthenticators);
    }
}
```

5.10 Authenticator.aidl

AIDL wrapper for *Authenticator.java*.

```
package org.fidoalliance.uaf.client;
```

776 parcelable Authenticator;

777 5.11 Authenticator.java

778 This class represents the set of metadata for an authenticator instance obtained through
779 the *Discovery* interface. For field definitions refer to analogous attributes in the DOM
780 API. [4.7.2]

781

```
782 package org.fidoalliance.uaf.client;
783
784 import java.util.List;
785 import java.util.ArrayList;
786
787 import android.os.Parcel;
788 import android.os.Parcelable;
789
790 public class Authenticator implements Parcelable {
791
792     public String AAID;
793     public String description;
794     public String logo;
795     public long   userVerification;
796     public long   keyProtection;
797     public long   attachmentHint;
798     public long   secureDisplay;
799     public String assertionScheme;
800     public long   additionalInfo;
801     public int    authenticationAlgorithm;
802     public List<Version> supportedUAFVersions;
803
804     public static final Parcelable.Creator<Authenticator> CREATOR
805             = new Parcelable.Creator<Authenticator>() {
806         public Authenticator createFromParcel(Parcel in) {
807             return new Authenticator(in);
808         }
809
810         public Authenticator[] newArray(int size) {
811             return new Authenticator[size];
812         }
813     };
814
815     private Authenticator(Parcel in) {
816         AAID                = in.readString();
817         description         = in.readString();
818         logo                = in.readString();
819         userVerification    = in.readLong();
820         keyProtection       = in.readLong();
821         attachmentHint      = in.readLong();
822         secureDisplay       = in.readLong();
823         assertionScheme     = in.readString();
824         additionalInfo      = in.readLong();
```

```
        authenticationAlgorithm         = in.readInt();
            in.readTypedList(supportedUAFVersions,
                                    Version.CREATOR);
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(AAID);
        dest.writeString(description);
        dest.writeString(logo);
        dest.writeLong(userVerification);
        dest.writeLong(keyProtection);
        dest.writeLong(attachmentHint);
        dest.writeLong(secureDisplay);
        dest.writeString(assertionScheme);
        dest.writeLong(additionalInfo);
        dest.writeInt(authenticationAlgorithm);
        dest.writeTypedArray(supportedUAFVersions);
    }
}
```

A number of constants are defined for use with the bit flag fields *userVerification*, *keyProtection*, *attachmentHint*, and *secureDisplay*. To avoid duplication and inconsistencies, please refer to the authoritative definitions found in the FIDO UAF Registry of Predefined Values [FIDORegistry].

5.11.1 Security Considerations

Android applications may choose to implement the user-interactive portion of FIDO in at least two ways: by authoring an *Activity* using Android-native user interface components, or with an HTML-based experience by loading a *WebView* and injecting the UAF DOM APIs with *addJavaScriptInterface()*. An application that chooses to inject the UAF interface into a *WebView* MUST follow all appropriate security considerations that apply to the DOM APIs and user agent implementers. In particular, the content of a *WebView* into which an API will be injected MUST be loaded only from trusted local content or over a secure channel as specified in [6.1] and MUST NOT contain insecure mixed-content.

6 Transport Binding Profile

This section describes general normative security requirements for how a client application transports FIDO UAF Protocol messages, gives specific requirements for Transport Layer Security [TLS], and describes an interoperability profile for using HTTP over TLS [RFC2818] with the UAF Protocol.

6.1 Transport Security Requirements

The UAF Protocol contains no inherent means of identifying a Relying Party server or for end-to-end protection of UAF Protocol messages. To perform a secure UAF Protocol exchange, the following abstract requirements apply:

1. The client application MUST securely authenticate the server endpoint as authorized, from that client's viewpoint, to represent the Origin [RFC6454] (scheme:host:port tuple) reported to the FIDO Client by the client application. Most typically this will be done by using TLS and verifying the server's certificate is valid, asserts the correct DNS name, and chains up to a root trusted by the client platform. Clients MAY also utilize other means to authenticate a server, such as a pre-provisioned certificate or key that is distributed with an application, or alternate network authentication protocols such as Kerberos [RFC4120].
2. The transport mechanism for UAF Protocol messages MUST provide confidentiality for the message, to prevent disclosure of their contents to unauthorized third parties. These protections should be cryptographically bound to proof of the server's identity in (1).
3. The transport mechanism for UAF Protocol messages MUST protect the integrity of the message from tampering by unauthorized third parties. These protections should be cryptographically bound to proof of the server's identity in (1).

6.2 TLS Security Requirements

If using HTTP over TLS to transport an UAF Protocol exchange, the following specific requirements apply:

1. If there are any TLS errors, whether "warning" or "fatal" or any other error level with the TLS connection, the HTTP client MUST terminate the connection without prompting the user. For example, this includes any errors found in certificate validity checking that HTTP clients employ, such as via TLS server identity checking [RFC6125], Certificate Revocation Lists (CRLs) [RFC5280], or via the Online Certificate Status Protocol (OCSP) [RFC2560].

2. Whenever comparisons are made between the presented TLS server identity (as presented during the TLS handshake, typically within the server certificate) and the intended source TLS server identity (e.g., as entered by a user, or embedded in a link), [RFC6125] server identity checking MUST be employed. The client MUST terminate the connection without prompting the user on any error condition.
3. The TLS server certificate MUST either be provisioned explicitly out-of-band (e.g. packaged with an app as a "pinned certificate") or be trusted by chaining to a root included in the certificate store of the operating system or a major browser by virtue of being currently in compliance with their root store program requirements. The client MUST terminate the connection without user recourse if there are any error conditions when building the chain of trust.
4. The "anon" and "null" crypto suites are not allowed and insecure cryptographic algorithms in TLS (e.g. MD4, RC4, SHA1) SHOULD be avoided (see NIST SP800-131A [SP800-131A]).
5. The client and server SHOULD use the latest practicable TLS version.
6. The client SHOULD supply and the server SHOULD verify whatever practicable channel binding information is available, including a ChannelID [CHANNELID] public key, the *tls-unique* and *tls-server-endpoint* bindings [RFC5929], and TLS server certificate binding [UAFProtocol]. This information provides protection against certain classes of network attackers and the forwarding of protocol messages, and a server MAY reject a message that lacks or has channel binding data that does not verify correctly.

6.3 HTTPS Transport Interoperability Profile

This section is non-normative.

Complex and highly-optimized applications utilizing UAF will often transport UAF protocol messages in-line with other application protocol messages. The profile defined here for transporting UAF protocol messages over HTTPS is intended to:

- Provide an interoperability profile to enable easier composition of client-side application libraries and server-side implementations for FIDO UAF-enabled products from different vendors.
- Provide detailed illustration of specific necessary security properties for the transport layer and HTTP interfaces, especially as they may interact with a browser-hosted application.

This profile is also utilized in the examples that constitute the Appendices of this document. This profile is OPTIONAL to implement. RFC 2119 key words are used in this

section to indicate necessary security and other properties for implementations that intend to use this profile to interoperate.

It is important to note that certain UAF operations, in particular Transaction Confirmation, will always require application-specific implementation. This interoperability profile only provides a skeleton framework suitable for replacing username/password authentication.

6.3.1 Obtaining a UAF Request message

A UAF-enabled web application might typically deliver request messages as part of a response body containing other application content, e.g in a script block as such:

```
...
<script type=”application/json”>
{
   “initialRequest”: {
       // initial request message here
   },
   “lifetimeMillis”: 60000; // hint: this initial request is valid for 60
seconds
}
</script>
...
```

However, request messages have a limited lifetime, and an installed application cannot be delivered with a request, so client applications generally need the ability to retrieve a fresh request.

When requesting a request message over HTTPS with XMLHttpRequest [XHR] or another HTTP API:

1. The URI of the server endpoint and how it is communicated to the client is application-specific.
2. The client MUST set the HTTP method to POST. [RFC2616]
3. The client MUST set the HTTP “Content-Type” header to “application/fido+uaf; charset=utf8”. [RFC2616]
4. The client SHOULD include “application/fido+uaf” as a media type in the HTTP “Accept” header. [RFC2616]
5. The client MAY need to supply additional headers, such as a Cookie [RFC6265], to demonstrate, in an application-specific manner, their authorization to perform a request.
6. The entire POST body MUST consist entirely of a JSON [ECMA-404] structure described by the *GetUAFRequest* interface.

The server's response SHOULD set the “Content-Type” to “application/fido+uaf; charset=utf8” and the body of the response MUST consist entirely of a JSON structure described by the *ReturnUAFRequest* interface.

6.3.2 Operation Enum

```
enum Operation {
        "Reg",   // Registration
        "Auth",  // Authentication or Transaction Confirmation
        "Dereg", // Deregistration
}
```

6.3.3 GetUAFRequest Interface

```
interface GetUAFRequest {
        Operation    op;
        DOMString    previousRequest;
        DOMString    context;
}
```

op of type *Operation*

The type of the UAF Request Message desired. Allowable string values are defined by the *Operation* enum. This field is OPTIONAL but MUST be set if the operation is not known to the server through other context. (e.g. an operation-specific URL endpoint)

previousRequest of type *DOMString*

If the application is requesting a new UAF request message because a previous one has expired, it may *OPTIONALLY* include the previous one to assist the server in locating any state that should be re-associated with a new request message, should one be issued.

context of type *DOMString*

Any additional contextual information that may be useful or necessary for the server to generate the correct request message. This key is *OPTIONAL* and the format and nature of this data is application-specific.

6.3.4 ReturnUAFRequest Interface

```
interface ReturnUAFRequest {
        int          statusCode;
        DOMString    uafRequest;
```

```
        Operation     op;
        long          lifetimeMillis;
}
```

statusCode of type *int*

The UAF Status Code for the operation.

uafRequest of type *DOMString*

The new UAF Request Message, *OPTIONAL,* if the server decided to issue one.

op of type *Operation*

An *OPTIONAL* hint to the client of the operation type of the message, useful if the server might return a different type than was requested. For example, a server might return a deregister message if an authentication request referred to a key it no longer considers valid. Allowable string values are defined by the *Operation* enum.

lifetimeMillis of type *long*

If the server returned a *uafRequest*, this is an *OPTIONAL* hint informing the client application of the lifetime of the message in milliseconds.

6.3.5 Delivering a UAF Response

Although it is not the only pattern possible, an asynchronous HTTP request is a useful way of delivering a UAF Response to the remote server for either web applications or standalone apps.

When delivering a response message over HTTPS with XMLHttpRequest [XHR] or another API:

1. The URI of the server endpoint and how it is communicated to the client is application-specific.
2. The client MUST set the HTTP method to POST. [RFC2616]
3. The client MUST set the HTTP “Content-Type” header to “application/fido+uaf; charset=utf8”. [RFC2616]
4. The client SHOULD include “application/fido+uaf” as a media type in the HTTP “Accept” header. [RFC2616]
5. The client MAY need to supply additional headers, such as a Cookie [RFC6265], to demonstrate, in an application-specific manner, their authorization to perform an operation.

6. The entire POST body MUST consist entirely of a the JSON [ECMA-404] structure described by the *UAFMessage.uafProtocolMessage* with any *additionalData* array removed.
7. The server's response SHOULD set the “Content-Type” to “application/fido+uaf; charset=utf8” and the body of the response MUST consist entirely of a JSON structure described by the *ServerResponse* interface.

6.3.6 ServerResponse Interface

The *ServerResponse* interface represents the completion status and additional application-specific additional data that results from successful processing of a *Register*, *Authenticate*, or *Transaction Confirmation* operation. This message is not formally part of the UAF Protocol, but the *uafResult* should be posted to the FIDO Client for housekeeping using through the *notifyUAFResult()* operation.

```
interface ServerResponse {

   readonly int       statusCode;
   readonly DOMString  description;
   readonly Token[]    additionalTokens;
   readonly DOMString  location;
   readonly DOMString  postData;
   readonly DOMString  newUAFRequest;

   interface Token {
     enum TokenType {
       "HTTP_COOKIE",
       "OAUTH",
       "OAUTH2",
       "SAML1_1",
       "SAML2",
       "JWT",
       "OPENID_CONNECT"
     };
     readonly TokenType type;
     readonly DOMString value;
   }
}
```

6.3.6.1 Attributes

statusCode of type *int*, *readonly*

The FIDO UAF response status code. Note that this status code describes the result of processing the tunneled UAF operation, not the status code for the outer HTTP transport.

No exceptions.

description of type *DOMString*, *readonly*

A detailed message describing the status code or providing additional information to the user.

No exceptions.

additionalTokens of type *ServerResponse*.*Token[]*, *readonly*

This key contains new authentication or authorization token(s) for the client that are not natively handled by the HTTP transport. Tokens SHOULD be processed prior to processing of *location*.

Note: The FIDO Server is not responsible for creating these tokens, they exist to provide a means for the relying party application to update the authentication/authorization state of the client in response to a successful FIDO operation. For example, these fields could be used to allow FIDO to serve as the initial authentication leg of a federation protocol, but the scope and details of any such federation are outside of the scope of FIDO.

No exceptions.

location of type *DOMString*, *readonly*

If present, indicates to the client web application that it should navigate the Document context to the URI contained on this field after processing any tokens.

No exceptions.

postData of type *DOMString*, *readonly*

If present in combination with location, indicates that the client should POST the contents to the *location* after processing any *tokens*.

No exceptions.

newUAFRequest of type *DOMString*, *readonly*

The server may use this to return a new UAF protocol message. This might be used to supply a fresh request to retry an operation in response to a transient failure, to request additional confirmation for a transaction or it a deregistration message in response to a permanent failure.

No exceptions.

6.3.6.2 ServerResponse.AdditionalToken Attributes

type of type *TokenType*, *readonly*

The type of the additional authentication / authorization token.

No exceptions.

value of type *DOMString*, *readonly*

The value of the additional authentication / authorization token.

No exceptions

6.3.6.3 ServerResponse.AdditionalToken enum TokenType

HTTP_COOKIE

If the user agent is a standard web browser or other HTTP native client with a cookie store, this *TokenType* SHOULD NOT be used. Cookies should be set directly with the Set-Cookie HTTP header for processing by the user agent. For non-HTTP or non-browser contexts this indicates a token intended to be set as an HTTP cookie. [RFC6265] For example, a native VPN client that authenticates with UAF might use this TokenType to automatically add a cookie to the browser cookie jar.

OAUTH

Indicates that the token is of type OAUTH. [RFC5849].

OAUTH2

Indicates that the token is of type OAUTH2. [RFC6749].

SAML1_1

Indicates that the token is of type SAML 1.1. [SAML1_1].

SAML2

Indicates that the token is of type SAML 2.0. [SAML2]

JWT

Indicates that the token is of type JSON Web Token (JWT). [JWT]

OPENID_CONNECT

Indicates that the token is an OpenID Connect “id_token”. [OPENIDCONNECT]

6.3.7 Security Considerations

It is important that the client set, and the server require, the method be POST and the “Content-Type” header be the correct value. Because the response body is valid ECMAScript, to protect against unauthorized cross-origin access, a server MUST NOT respond to the type of request that can be generated by a script tag, e.g. *<script src=”https://example.com/fido/uaf/getRequest”>*. The request a user agent generates with this kind of embedding cannot set custom headers.

Likewise, by requiring a custom “Content-Type” header, cross-origin requests cannot be made with an XMLHttpRequest without triggering a CORS preflight access check. [CORS]

As UAF messages are only valid when used same-origin, servers SHOULD NOT supply an “Access-Control-Allow-Origin” [CORS] header with responses that would allow them to be read by non-same-origin content.

To protect from some classes of cross-origin, browser-based, distributed denial-of-service attacks, request endpoints SHOULD ignore, without performing additional processing, all requests with an “Access-Control-Request-Method” [CORS] HTTP header or an incorrect “Content-Type” HTTP header.

If a server chooses to respond to requests made with the GET method and without the custom “Content-Type” header, it SHOULD apply a prefix string such as *“while(1);” or “&&&BEGIN_UAF_RESPONSE&&&”* to the body of all replies and so prevent their being read through cross-origin <*script*> tag embedding. Legitimate same-origin callers will need to (and alone be able to) strip this prefix string before parsing the JSON content.

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*

[UAFAuthnrMetadata] Davit Baghdasaryan, Brad Hill. FIDO Universal Authentication Framework Authenticator Metadata. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-uaf-authnr-metadata-v1.0-rd-20140209.pdf

[UAFProtocol] Rolf Lindemann, Davit Baghdasaryan, Eric Tiffany. FIDO Universal Authentication Framework Protocol. Version v1.0-rd-20140209, FIDO Alliance, February 2014. See http://fidoalliance.org/specs/fido-uaf-protocol-v1.0-rd-20140209.pdf

[FIDORegistry] Rolf Lindemann, Davit Baghdasaryan, Brad Hill. FIDO Universal Authentication Framework Registry of Predefined Values. Version v1.0-rd-20140209, FIDO Alliance. February 2014. See http://fidoalliance.org/specs/fido-uaf-reg-v1.0-rd-20140209.pdf

Other References:

[ANDROID] The Android™ Operating System, Google Inc., the Open Handset Alliance and the Android Open Source Project. Work in progress. See: http://developer.android.com/

[CHANNELID] Transport Layer Security (TLS) Channel IDs, D. Balfanz, Work in progress. (draft-balfanz-tls-channel-id-00)

[CORS] Cross-Origin Resource Sharing, A. van Kesteren, Ed., World Wide Web Consortium, January 2014

[DOMTR] Document Object Model (DOM) Technical Reports, World Wide Web Consortium. Work in progress.

[ECMA-262] ECMAScript Language Specification, Editing 5.1, A. Wirfs-Brock, Editor. Ecma International, June 2011.

Unofficial HTML version available at http://es5.github.com/.

[ECMA-404] The JSON Data Interchange Format, ECMA International, October 2013

[JWT] JSON Web Token (JWT), draft-ietf-oauth-json-web-token, M. Jones, Work in progress.

[OPENIDCONNECT] OpenID Connect, OpenID Foundation, Work in progress.

[PNG] Portable Network Graphics (PNG) Specification (Second Edition) Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E), D. Duce, Ed., World Wide Web Consortium, November 2003

1208 **[RFC2119]** Key words for use in RFCs to Indicate Requirement Levels (RFC2119), S.
1209 Bradner, March 1997

1210 **[RFC2397]** The "data" URL scheme (RFC2397), L. Masinter, August 1998

1211 **[RFC2560]** X.509 Internet Public Key Infrastructure Online Certificate Status Protocol –
1212 OCSP (RFC2560), M. Myers et al, June 1999

1213 **[RFC2616]** Hypertext Transfer Protocol – HTTP/1.1 (RFC2616), R. Fielding et al, June
1214 1999

1215 **[RFC2818]** HTTP over TLS (RFC2818) , E. Rescorla, May 2000

1216 **[RFC4120]** The Kerberos Network Authentication Service (V5) (RFC4120), C. Neuman
1217 et al, July 2005

1218 **[RFC5280]** Internet X.509 Public Key Infrastructure Certificate and Certificate Revoca-
1219 tion List (CRL) Profile (RFC5280), D. Cooper et al, May 2008

1220 **[RFC5849]** The OAuth 1.0 Protocol (RFC5849), E. Hammer-Lahav, Ed., April 2010

1221 **[RFC5929]** Channel Bindings for TLS (RFC 5929), J. Altman et al, July 2010

1222 **[RFC6125]** Representation and Verification of Domain-Based Application Service Iden-
1223 tity within Internet Public Key Infrastructure using X.509 (PKIX) Certificates in the Con-
1224 text of Transport Layer Security (TLS) (RFC6125), P. Saint-Andre et al, March 2011

1225 **[RFC6265]** HTTP State Management Mechanism (RFC6265), A.Barth, April 2011

1226 **[RFC6454]** The Web Origin Concept (RFC6454), A. Barth, December 2011

1227 **[RFC6749]** The OAuth 2.0 Authorization Framework (RFC6749), D. Hardt, Ed., October
1228 2012

1229 **[SAML1_1]** Security Assertion Markup Language (SAML) v.1.1, OASIS, October 2003

1230 **[SAML2]** Security Assertion Markup Language (SAML) v2.0, OASIS, March 2005

1231 **[SOP]** Same-Origin Policy, Mozilla Developer Network, January 2014

1232 **[SP800-131A]** NIST Special Publication 800-131A, Transitions: Recommendations for
1233 Transitioning the Use of Cryptographic Algorithms and Key Lengths, E. Barker et al, Na-
1234 tional Institute of Standards and Technology, January 20111

1235 **[TLS]** The TLS Protocol Version 1.0 (RFC 2246), Version 1.1 (RFC 4346), Version 1.2
1236 (RFC 5246)

1237 **[WebIDL]** Web IDL, World Wide Web Consortium, work in progress.

1238 **[WEBMESSAGING]** HTML5 Web Messaging, I. Hickson, World Wide Web Consortium,
1239 work in progress.

1240 **[XHR]** XMLHttpRequest, J. Aubourg et al, World Wide Web Consortium, work in
1241 progress.