AMD White Paper

FAST IDENTITY ONLINE(FIDO): PASSWORD-LESS AUTHENTICATION

Summary

Password based security has become less efficient in recent times due to more sophisticated phishing attacks and challenges faced by users to create and manage numerous passwords. Also, the centralized storage of millions of user credentials (including passwords) in a cloud or a website server can provide a single point of target for hackers, which can greatly increase the cost of a single security breach.

Online attacks like Man-in-the-Middle (MitM) and Man-in-the-Browser (MitB) can pose a substantial threat to online authentication by secretly modifying the communication between user and server. Online services are beginning to initiate stricter password rules, but these can pose challenges for users.

The FIDO Alliance[1] and other leading firms came together to enhance cybersecurity by putting together a credential protocol based on public key cryptography designed to mitigate security threats like MitM and MitB. FIDO specified a unique solution by replacing passwords with biometrics to help create a better user experience by storing the user credentials in the user devices instead of in centralized storages.

The FIDO Alliance[1] is comprised of more than 250 organizations, including major platform and browser providers such as Microsoft®, Google, and Firefox®. and aims to provide an interoperable ecosystem for password-less online transactions used by various applications and websites.

This paper discusses FIDO's password-less authentication in the Windows® platform and examines the capability of the Windows 10 and Edge browser being used to complete the FIDO transaction on an AMD processor-based laptop using a Synaptics fingerprint sensor.

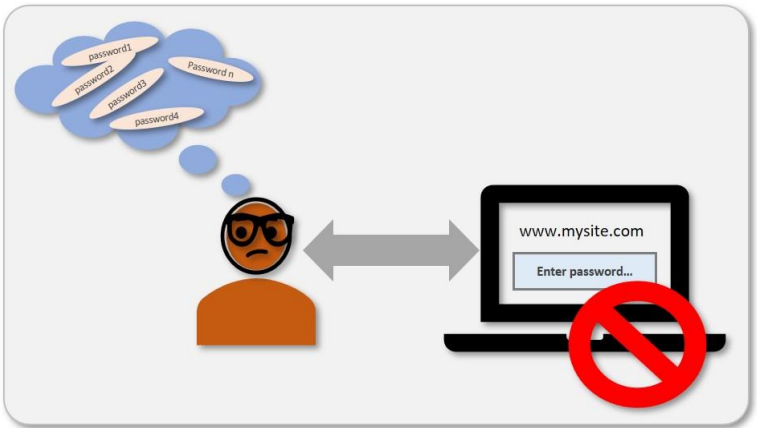


Figure 1: Password Challenges

Authors

Maneesh Jhinger
Senior Member Technical Staff
maneesh.jhinger@amd.com

Pooja Kuntal
Senior Software Engineer
pooja.kuntal@amd.com

Contents

FIDO Introduction

FIDO specifications define the abstract application programming interfaces (APIs) for the FIDO protocol implementation, helping to provide interoperability to developers to assist with creating single secure user accounts operating from different platforms, browsers, and apps running on different hardware. The open source FIDO Alliance provides generic system management functionality which can help reduce costs associated with the deployment, application development, testing and debug cycle.

In traditional credential management, a password is created during registration/signup and stored at an online site server (relying party). In case of authentication/login, user provides the password and relying party (RP) server carries out the authentication process by matching it against stored password. FIDO is designed to eliminate the concept of storing user credential at online-site server. It can provide a new mechanism to establish a secure and robust authentication channel by creating a public-private cryptographic key pair which can bind the user and the local user device with the relying party. User credential match can be done using either a built-in authenticator (e.g. biometric sensor) or a plug-in authenticator (e.g. USB/smartcard) or an external/roaming authenticator (e.g. mobile). User enrollment to an authenticator, for example registering user fingerprints to a biometric authenticator, is outside the scope of FIDO protocol and is an authenticator specific process.

FIDO Registration Overview

- RP application in user device initiates Registration of a user's account at Relying party.
- Relying party discovers the presence of FIDO authenticators in the device using RP application, selects the RP compliant authenticators and communicates back to RP server. Server sends a "challenge" to kick start the registration.
- User is prompted to choose an available FIDO authenticator that matches the relying party's acceptance policy.
- User unlocks the FIDO authenticator using a fingerprint sensor, a button on a second–factor device, securely–entered PIN or other methods.
- User device creates a unique public-private key pair binding the local device, relying party and user's account.
- Public key along with signed "challenge" is sent back to the relying party, whereas private key and biometric credentials are stored in the user device.

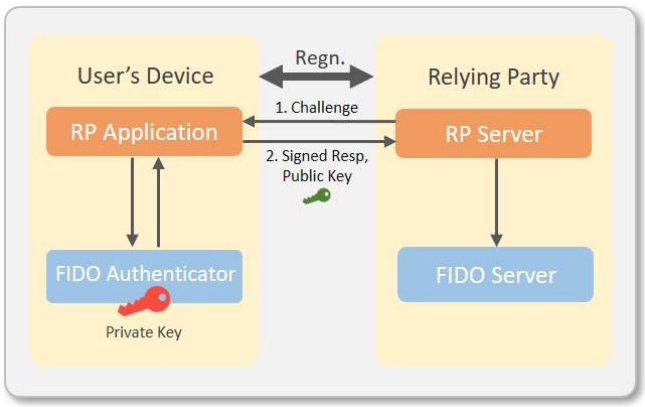


Figure 2: FIDO Registration Overview

FIDO Authentication Overview

- User initiates authentication by requesting relying party.
- RP server responds by sending a "challenge" back to the user device.
- User is prompted to select an authenticator based on RP policy.
- User unlocks the FIDO authenticator using the same method as it has used while registration.
- User device uses the "user's account identifier" provided by the relying party to select the correct private key stored securely in the device and then signs the "challenge" sent by RP server using private key.
- User device sends the signed challenge back to the RP server, which verifies the signature of the "challenge" with the stored public key corresponding to user and device.
- RP server then allows the user to proceed with the transaction.

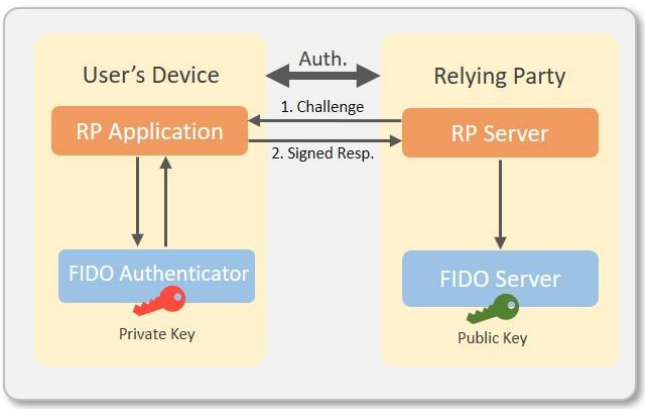


Figure 3 : FIDO Authentication Overview

FIDO Protocols

The FIDO Alliance[1] defines three sets of technology agnostic specifications: Universal Authentication Framework (UAF), Universal Second Factor (U2F) and Client to Authenticator Protocol (CTAP). U2F protocol is designed to strengthen the existing password-based authentication mechanism by introducing second factor authentication using plug-in or out-of-device (roaming) authenticator, UAF protocol is designed to provide password-less experience with devices using built-in or plug-in authenticators, and CTAP is designed to extend the UAF and U2F functionality by providing the password-less authentication using out-of-device (roaming) authenticator.

Universal Second Factor (U2F)

U2F specification has an upper layer with cryptographic core protocol and a lower layer specifying user to U2F device communication for an allowed transport protocol (e.g. USB, NFC, BLE etc.). The registration involves the traditional user-name/password, but the service can prompt the user to present a second factor device. An online-site specific key pair will be generated to bind the U2F device with the site. Second factor authentication will be done by either pressing a button on a USB device or using NFC device (U2F) to allow site-specific key pair to be used to complete the authentication by signing the challenge sent by RP.

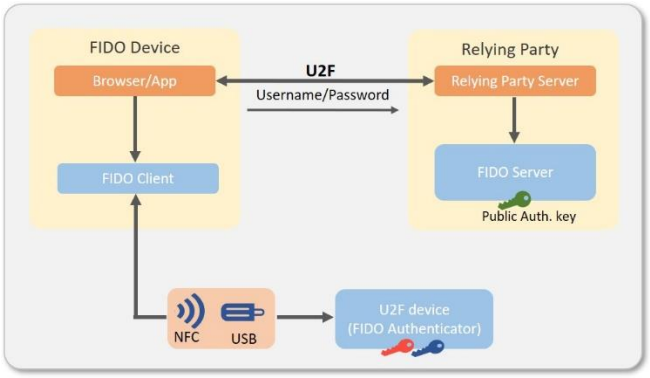


Figure 4: U2F Architecture

Universal Authentication Framework (UAF)

The non-normative UAF architecture proposes a generic FIDO client which may be embedded in a browser or app. FIDO client is designed to communicate with "Authenticator Specific Module" (ASM) which may be platform-specific (e.g. for Windows or Android) but is authenticator-vendor agnostic. ASM in turn can connect the client to vendors specific authenticators using generic API's. UAF can also allow experiences that combine multiple authentication mechanisms, such as fingerprint + PIN. In case of plug-in authenticators, the user device and authenticator can establish connection using USB protocol.

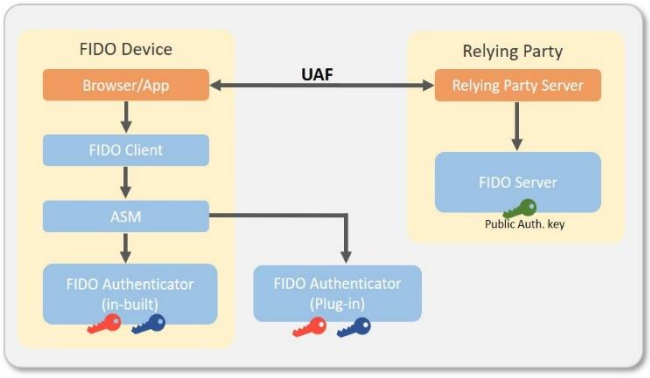


Figure 5: UAF Architectue

Client to Authenticator Protocol (CTAP)

CTAP can provide a way to establish communication between platform/client with roaming authenticator. CTAP is designed to enable external devices such as mobile handsets or FIDO Security Keys to serve as authenticators. The protocol defines an "Authenticator API", as a message encoding method to handle the encoded message sent-to or received-from roaming authenticator over the chosen transport protocol and a "Transport-Specific Binding" to describe message bindings with the chosen transport protocol (e.g. NFC, BLE, USB).

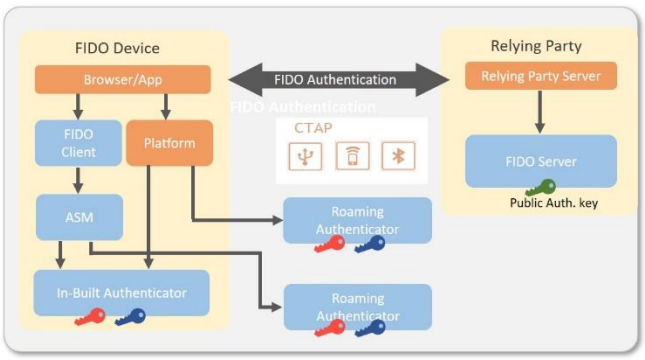


Figure 6: Client to Authenticator Protocol

FIDO2.0: Standardization of FIDO protocols

The FIDO Alliance[1] collaborated with "World Wide Web Consortium" (W3C)[2,3], which is an international community for developing open standards for web, to standardize FIDO2 implementation as a set of "Web Authentication API's". The "Web Authentication API's", also called as **WebAuthn**[2,3], is supported by all major existing browsers. It provides an interface for cryptographic public-key authentication of users for web-based applications and services. To support FIDO2, **WebAuthn**[2,3] implements an extension of existing "Credential Management API" that store username-password combinations. The existing "Credential Management API"[4] define a framework for handling the credentials of two types, namely *PasswordCredential* and *FederatedCredential*. The *PasswordCredintial* is password based and *FederatedCredential* is provided by a federated identity which is trusted by website to authenticate a user. The **WebAuthn**[2,3] defines a third credential type, *PublicKeyCredential*, which uses cryptographically attested credentials for strong user authentication. To accommodate *PublicKeyCredential*, the **WebAuthn**[2,3] extends the existing JavaScript methods of *navigator.credentials.create()* and *navigator.credentials.get()*, to accept a *publicKey* parameter. During registration, the *navigator.credentials.create()* method generates public-private key pair and binds them with the user account, user device and relying party. During transaction, *navigator.credentials.get()* method is used for obtaining an authentication result.

FIDO **WebAuthn** Registration

navigator.credentials.create(), creates new credentials, either for registering a new account or for associating a new asymmetric key pair credentials with an existing account.

1. RP script embedded in web page or RP app sends the request to RP server for registration.
2. Server sends back the user info, RP info and an array of bytes as challenge. The parameters received from the server will be passed on to the *navigator.credentials.create()* as arguments which synchronously returns a "Promise" to be resolved to a *PublicKeyCredential* structure on completion of registration process.
3. *navigator.credentials.create()* implemented in "Web Auth client" module creates client data by determining *rpid*, *challenge*, *tokenBinding*. and generates the *clientDataHash* . It then calls the authenticator's *authenticatorMakeCredential()* with the received parameters as arguments along with the *clientDataHash*.
4. Authenticator creates asymmetric key pair and perform attestation:
   a. Authenticator or its user agent prompts for obtaining user consent. The prompt displays user name and RP name, user then verify itself using PIN or Biometrics.
   b. Authenticator creates new asymmetric key pair. The *privateKey* along with *rpid*, *userHandle* and credential.id (a probabilistically-unique byte sequence generated by authenticator) becomes a "Public Key Credential Source" which binds key pair with RP and User. The public key portion of this is called *credentialPublicKey*.
   c. A "attestation object" is created containing authenticator data which includes credential.id and *credentialPublicKey*, and an attestation statement (in authenticator chosen format).
5. Authenticator returns "attestation object" and *clientData* to the "Web Auth client".
6. The *navigator.credentials.create()* "Promise" (step 2) resolves to an *PublicKeyCredential* structure, which has a *PublicKeyCredential.rawId* that is the globally unique credential id and *AuthenticatorAttestationResponse* structure containing *clientDataJSON* and the "attestation object" received from authenticator. The *PublicKeyCredential* is sent back to the server by RP scripts.
7. Server carries out verification of client data and "attestation object".

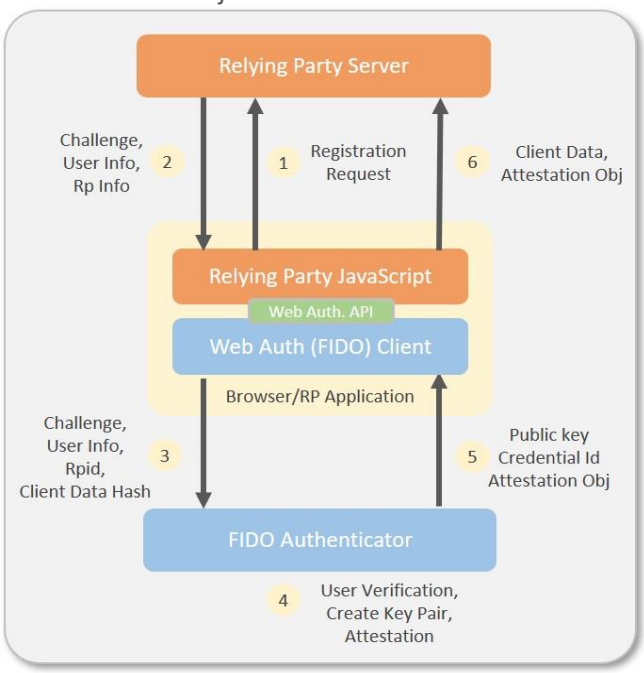


Figure 7 : FIDO WebAuthn Registration

FIDO **WebAuthn** Authentication

navigator.credentials.get(), uses an existing set of credentials to authenticate to a service, either logging a user in or as a form of second-factor authentication.

Authentication process will execute following operation:

1. Web page requests to server for authentication.
2. Server sends a challenge to "Relying Party JavaScript". Script invokes *navigator.creation.get()* with the parameters received from server which returns a "Promise" to be resolved to a *PublicKeyCredential* structure containing an *AuthenticatorAssertionResponse* on completion of authentication process.
3. *navigator.credentials.get()* implemented in "Web Auth client" module calls *authenticatorGetCredential()* on Authenticator:
   a. It validates input parameters and fill in origin and relying party id.
   b. Parameters of *navigator.credentials.get()* are passed to authenticator, along with *clientDataHash* (hash of client data).
4. Authenticator creates an Assertion:
   a. The authenticator finds a credential for this service that matches the Relying Party ID.
   b. Authenticator or its user agent prompts for obtaining user consent. The prompt displays user name and RP name, user can verify itself using PIN or Biometrics.
5. The authenticator creates a new assertion by signing over the *clientDataHash* and *authenticatorData* with the *privateKey* generated for the account during the registration call.
6. The authenticator returns the *authenticatorData* and assertion signature back to the client.
7. The client resolves the "Promise" (step 2) to a *PublicKeyCredential* structure containing *AuthenticatorAssertionResponse*, which include *signature* obtained using *privateKey*. RP script sends the response to the server
8. Server verifies the signature using public key and authenticates the transaction.

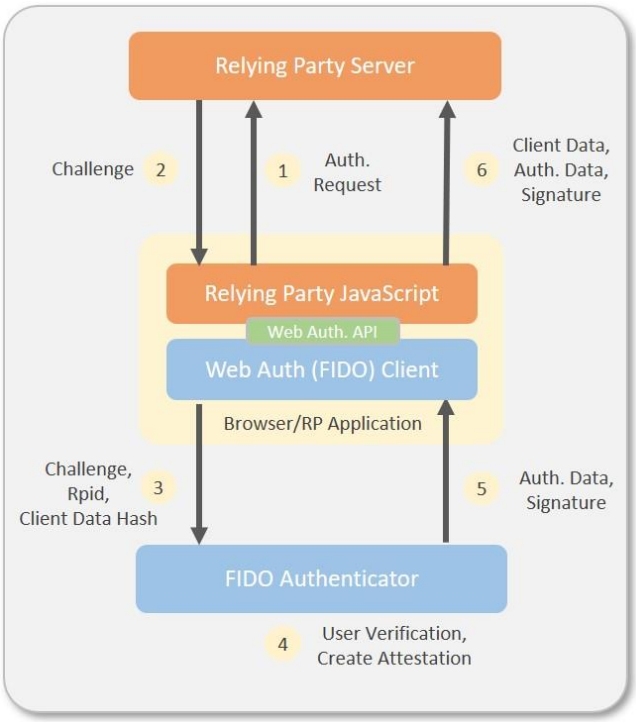


Figure 8: FIDO WebAuthn Authentication

Microsoft Support for FIDO

Microsoft supports FIDO in Windows 10[9] by enabling security features like Windows Hello, Microsoft Passport and providing web authentication APIs support in Microsoft Edge. Windows Hello and Microsoft passport are designed to map to the "Authenticator" and "Authenticator Specific Module" (ASM) of the FIDO architecture respectively.

Microsoft Passport

Microsoft has revamped Passport for new PKI credential based multi factor authentication to provide full support for FIDO transactions. Microsoft Passport is designed to work with Trusted Platform Module (TPM2.0) for enhanced security solution.

When a Windows user registers with relying party, MS passport generates a public/private key pair. The private key is stored in the TPM and can be accessed only with the help of Windows Hello, which performs the user verification by matching user's biometrics, PIN, etc. The keys are encrypted and protected by the TPM. Key pair will be associated with the user account and local device. User needs to enroll each device used to access relying party's services.

Microsoft Passport is designed to function like a virtual smart card. A smart card must be registered with a service and has a private key locked within it which is used to sign the challenge in case of any transaction with that service. The Microsoft Passport credential essentially works in a similar manner.

Windows Hello

Windows Hello is designed to enhance security by facilitating multi-factor authentication using biometrics, PIN, etc. that is used to unlock a private key stored in the secure TPM chip. Windows Hello is Microsoft software, or middleware in Windows 10, that is used to support biometrics for authentication. Previously, OEMs used to provide their own solution to support biometric authentication. FIDO transactions are designed to carry out user verification before creating/unlocking key-pair. Windows Hello looks for registered biometrics or PIN for user verification during FIDO transaction, but if built-in biometrics device is not available, it prompts to insert security key into the USB port to complete user verification.

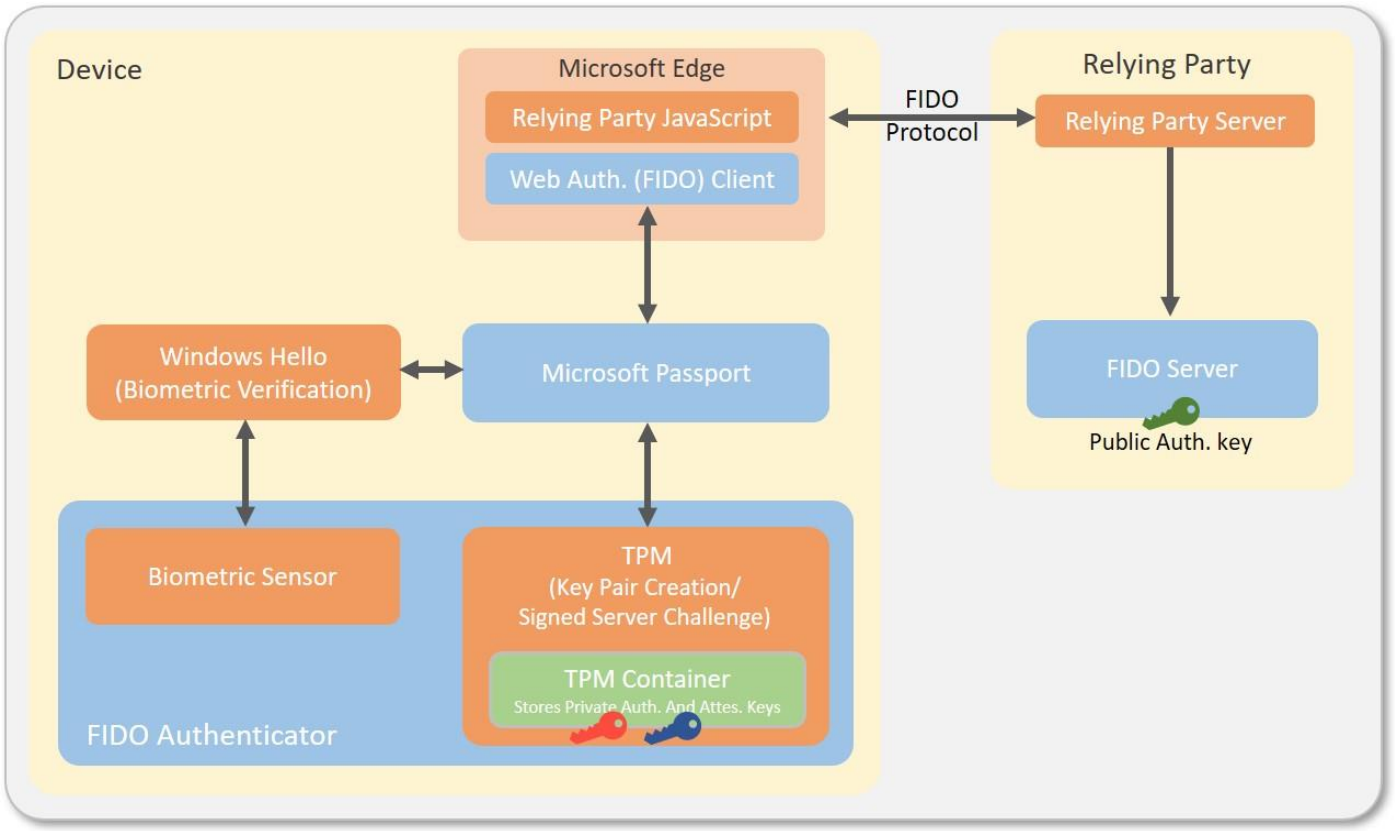


Figure 9 : FIDO2.0 in Windows 10

FIDO Registration in Windows10

In Microsoft Edge, when web authentication API *navigator.credential.create()* is invoked by relying party script, the browser connects to Microsoft Passport which uses Trusted Platform Module (TPM) for key pair creation and prompts Windows Hello for user verification.

1. Relying Party server sends registration challenge and *CredentialCreateOption* to RP JavaSscript running on user device.
2. RP JavaScript executes *navigator.credential.create()* using params passed by server.
3. Browser/Webauth Client invokes Microsoft Passport's *RequestCreateAsync()*.
4. *RequestCreateAsync()* API does following :
   a. Prompts a Window Hello dialog to request the user's PIN or biometric.
   b. Requests the TPM chip to create the private and public key, store the result and returns Public key to server.

FIDO Authentication in Windows 10

In Microsoft Edge, when web authentication API *navigator.credential.get()* is invoked by relying party script, the browser connects to Microsoft Passport which prompts Windows Hello for user verification before unlocking private key. It connects to TPM to sign RP server challenge using private key after user verification.

1. Relying Party server sends authentication challenge and *CredentialRequestOption* to script.
2. Relying party scripts executes *navigator.credential.get()* using params passed by server.
3. Client invokes Microsoft Passport's *RequestSignAsync()* API to sign the challenge sent from server. It triggers the OS to request the user's PIN or biometrics through Windows Hello. After successful user identification, challenge gets signed in TPM.
4. Client sends signed challenge back to server.

Web Authentication Demo – HP EliteBook (with AMD Ryzen™ processor) with Windows 10

In the following project we demonstrate the capability of Microsoft Passport and Windows Hello to support Web Authentication API and provide a standard implementation for FIDO2 architecture. By mimicking and hardcoding the RP script to communicate to "locally hosted relying party", we are able to demonstrate the user registration and authentication. Below are the setup and demo details:

Setup Configuration

1. Platform: Windows 10 RS5 (v_43.17713.1000.0)
2. Browser: Microsoft Edge
3. Enabled Web Authentication API in Edge.
4. Configured Microsoft Account and registered it with Windows Hello biometrics.

Demo Development

1. Hosted a html page on localhost and embedded relying party script which invokes *navigator.credential.create()* API to register the user and *navigator.credential.get()* to authenticate registered user.
2. When *navigator.credential.create()* executes in Edge, It prompts user verification using Windows Hello UI to use PIN/Biometrics verify that the user is the same user as the one logged into the Windows account.
3. After successful verification, Microsoft Passport will generate a public/private key pair and store the private key in the Trusted Platform Module (TPM), the dedicated crypto processor hardware used to store credentials
4. When *navigator.credential.get()* is executed, user verification happens using Windows Hello and later the challenge will be signed within the TPM and the promise will return with an assertion object that contains the signature and other metadata.

Conclusion

Microsoft supports the FIDO2 specification and FIDO based secured transactions can be done on the Windows 10 platform. Biometric sensor along with TPM acts as platform authenticator. Microsoft Passport, along with Windows Hello, can help connect the dots from browser to platform authenticator. In this paper, we successfully demonstrated FIDO support in Windows 10 platform using Synaptics Fingerprint (FP) sensor for biometric user verification.

References

1. https://fidoalliance.org/
2. https://www.w3.org/TR/webauthn/
3. https://github.com/w3c/webauthn/
4. https://www.w3.org/TR/credential-management-1/
5. https://docs.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/hello-identity-verification
6. https://blogs.windows.com/business/2015/02/13/microsoft-announces-fido-support-coming-to-windows-10
7. https://docs.microsoft.com/en-us/microsoft-edge/dev-guide/device/web-authentication
8. https://www.slideshare.net/SanjeevVermaPhD/fidoalliance
9. http://docs.microsoft.com/en-us/microsoft-edge/dev-guide/windows-integration/web-authentication

Appendix A: Web Authentication APIs Input/Output parameters

navigator.credential.create()

Input: It takes dictionary '***PublicKeyCredentialCreationOptions***' as Input parameter:

```
dictionary PublicKeyCredentialCreationOptions
{
    required PublicKeyCredentialRpEntity                    rp;
    required PublicKeyCredentialUserEntity                  user;
    required BufferSource                                   challenge;
    required sequence<PublicKeyCredentialParameters>        pubKeyCredParams;
    unsigned long                                           timeout;
    sequence<PublicKeyCredentialDescriptor>                 excludeCredentials = [];
    AuthenticatorSelectionCriteria                          authenticatorSelection;
    AttestationConveyancePreference                         attestation = "none";
    AuthenticationExtensionsClientInputs                    extensions;
};
```

Output : It returns a Promise of type interface '***PublicKeyCredential***' which is derived from '***Credential***' Interface:

```
interface PublicKeyCredential : Credential
{
    readonly attribute ArrayBuffer                  rawId;
    readonly attribute AuthenticatorResponse        response;
    AuthenticationExtensionsClientOutputs           getClientExtensionResults();
};
interface Credential
{
    readonly attribute USVString                    id;
    readonly attribute DOMString                    type;
};
```

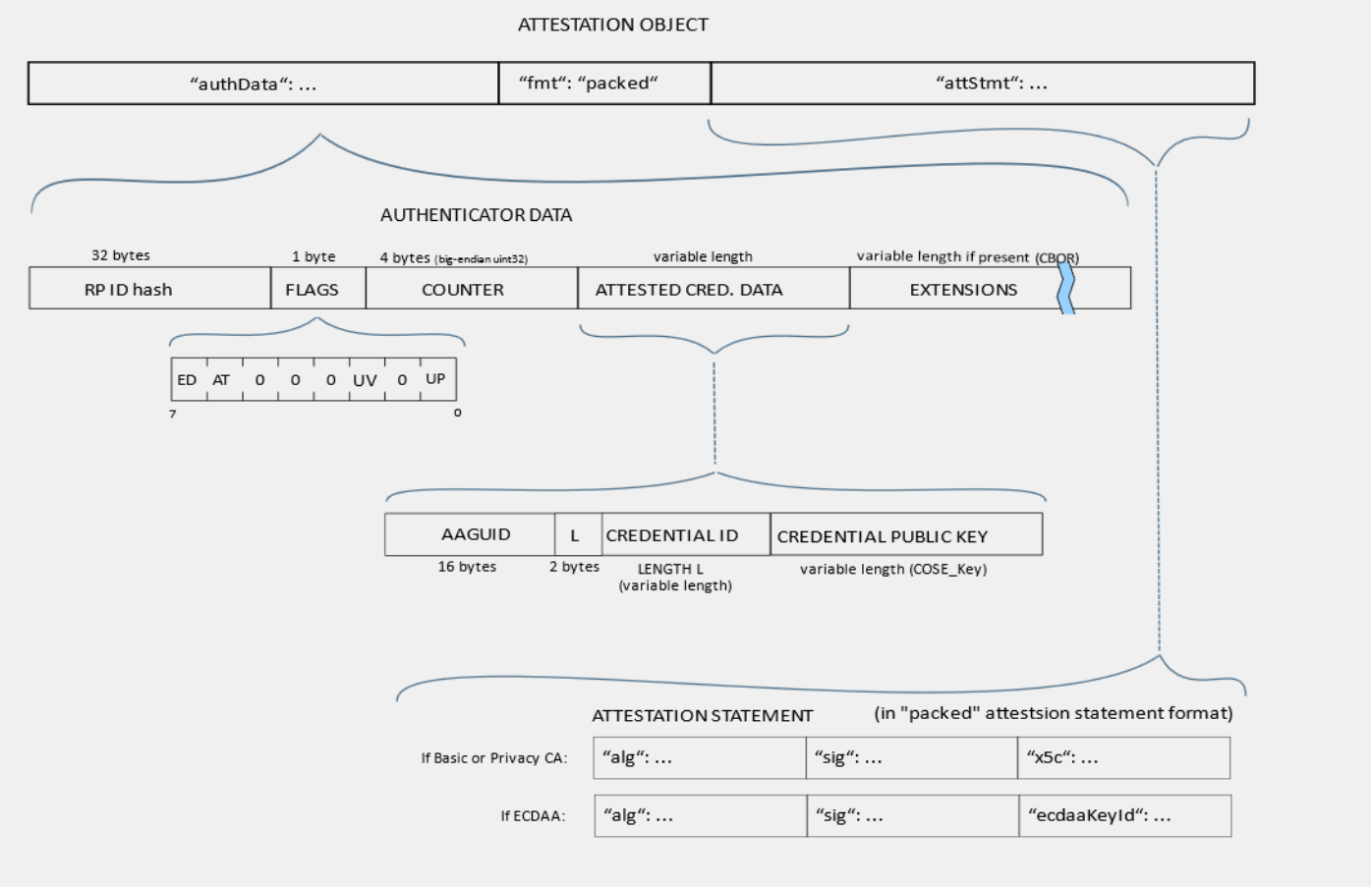
AuthenticatorResponse will contain '***AuthenticatorAttestationResponse***' in response of create call which is derived from ***AuthenticatorResponse*** interface:

```
interface AuthenticatorAttestationResponse : AuthenticatorResponse
{
    readonly attribute ArrayBuffer                  attestationObject;
};
interface AuthenticatorResponse
{
    readonly attribute ArrayBuffer                  clientDataJson;
};
```

ClientDataJson is JSON serialization of the ***CollectedClientData*** passed to the authenticator by the client in its call to either *navigator.credential.create()* or *navigator.credential.get()*.

```
dictionary CollectedClientData
{
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    TokenBinding                tokenBinding;
};
```

Attestation object contains authenticator data and attestation statement as shown in below figure:



navigator.credential.get()

Input : It takes dictionary '***PublicKeyCredentialRequestOptions***' as Input parameter:

```
dictionary PublicKeyCredentialRequestOptions
{
    required  BufferSource                          challenge;
    unsigned long                                   timeout;
    USVString                                       rpId;
    sequence<PublicKeyCredentialDescriptor>         allowCredentials = [];
    UserVerificationRequirement                     userVerification = "preferred";
    AuthenticationExtensionsClientInputs            extensions;
};
```

Output : It returns a Promise of type interface '***PublicKeyCredential***' which is derived from '***Credential***' Interface:

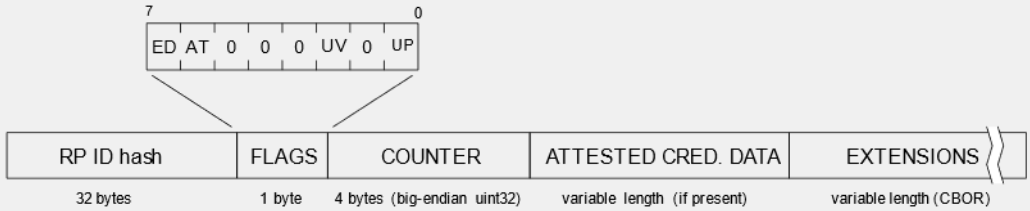
```
interface PublicKeyCredential : Credential
{
    readonly attribute  ArrayBuffer                 rawId;
    readonly attribute  AuthenticatorResponse       response;
    AuthenticationExtensionsClientOutputs           getClientExtensionResults();
};

interface Credential {
    readonly attribute  USVString                   id;
    readonly attribute  DOMString                   type;
};
```

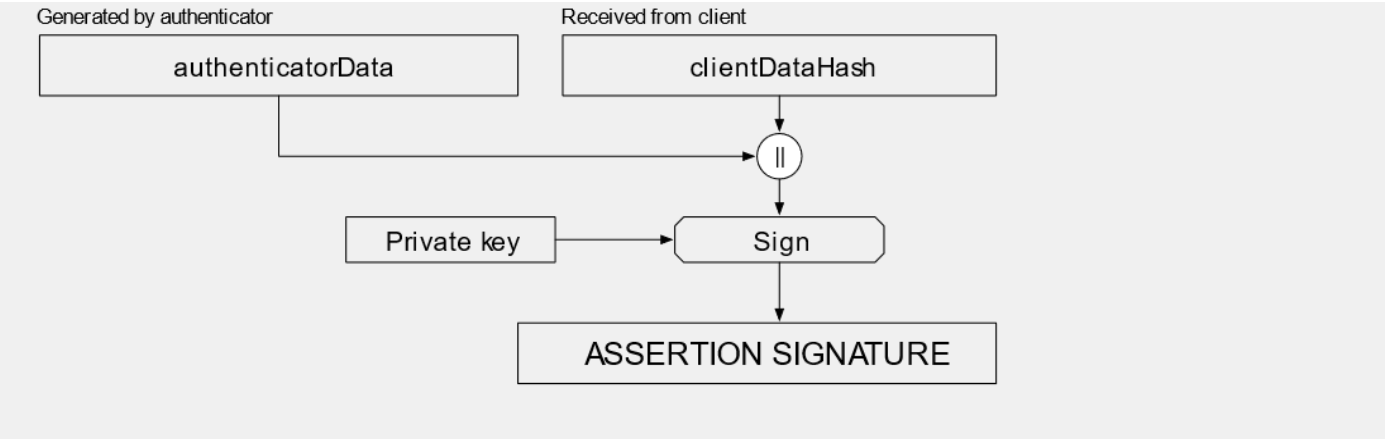
AuthenticatorResponse will contain '***AuthenticatorAssertionResponse***' in response of create call which is derived from ***AuthenticatorResponse*** interface.

```
interface AuthenticatorAssertionResponse : AuthenticatorResponse
{
    readonly attribute ArrayBuffer    authenticatorData;
    readonly attribute ArrayBuffer    signature;
    readonly attribute ArrayBuffer    userHandle;
};
interface AuthenticatorResponse {
    readonly attribute ArrayBuffer    clientDataJson;
};
```

The *authenticatorData* structure encodes contextual bindings made by the authenticator. These bindings are controlled by the authenticator itself and derive their trust from the Relying Party's assessment of the security properties of the authenticator. The authenticator data structure is a byte array of 37 bytes or more, as follows:



The *signature* is the assertion signature of the concatenation of *authenticatorData* and hash using the privateKey of *selectedCredential* as shown in figure below:



DISCLAIMER:

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.
The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.
AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.
AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS DOCUMENT IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.
The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.