Πανεπιστήμιο Πειραιώς

University of Piraeus

M.Sc. Digital Systems Security

ANALYSIS AND IMPLEMENTATION OF THE FIDO PROTOCOL IN A TRUSTED ENVIRONMENT

Supervisor: Christos Xenakis
(xenakis@unipi.gr)

Author: Anna Angelogianni
(annaaggelogianni@ssl-unipi.gr)

University of Piraeus 6.7.2018

Πανεπιστήμιο Πειραιώς
University of Piraeus

ACKNOWLEDGMENTS

I would like to express my very great appreciation to my supervisor Professor Christos Xenakis for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously has been very much appreciated.
I would also like to thank Dr. Christoforos Ntantogian, for his advice and assistance in keeping my progress on schedule.
My grateful thanks are also extended to Mr. Panagiotis Nikitopoulos for his help in the implementation part, our conversations helped me evolve as a programmer.
I would also like to extend my thanks to all my professors during this Master that helped me grow both as a professional and as a human.
Last but not least, I wish to thank my parents and my friends for their support and encouragement throughout my studies.

Πανεπιστήμιο Πειραιώς
University of Piraeus

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

The increasing use of online accounts has created the need for access control and security. Different authentication techniques have been proposed over the years but the passwords have failed to be replaced yet. FIDO protocol proposes a new authentication scheme that guarantees both security and usability. Nevertheless, for every protocol to be secure, trusted hardware is also needed for the storage of private keys. Therefore, this thesis explores both FIDO and TEE and proposes a way to combine them both to a proven secure scheme.

CHAPTER 1: INTRODUCTION

1.1 The problem

The extended use of online services has resulted in the vast adoption of passwords as authentication schemes, aiming to create a secure environment for all parties involved.
Unfortunately, the idea of passwords has many problems such as:

- The easier the password is for the owner to remember it, the easier it will also be for the attacker to guess.
- Passwords are stolen from the servers.
- Users are entering their credentials into untrusted apps.
- Users have problems remembering the different passwords used for different services with different password policies.

In general terms, users do not know where they enter their passwords and servers do not know if the client is who he says.
It is crucial to develop strong authentication schemes that combine security with usability.

1.2 The solution

FIDO (Fast Identity Online) proposes a strong authentication scheme in which the user is authenticated to the device and the latter is authenticated to the server using a challenge-response scheme and public key cryptography.

1.3 The benefits

- User credentials are now stored on the user's device in a trusted environment.
- Server only stores the public key of the user authentication process.
- Users do not have to remember complex passwords (convenience & security).
- Users can select the authentication mechanism of their preference (PIN, biometrics, etc.) and use it for different services.
- Authentication keys are different for different services.
- FIDO protocol can be combined with existing technologies and it is highly extensible.
- Both the server and the client are protected.

CHAPTER 2: FIDO UAF, U2F AND FIDO 2

There are two key protocols within FIDO: FIDO UAF and FIDO U2F.

2.1 FIDO UAF Overview

The Universal Authentication Protocol (UAF) allows online services to offer password-less and multi-factor security. There are two basic parties involved in the UAF protocol: the server and the client.

CLIENT

The client side includes the *FIDO UAF client* which implements:

(a) the client side of FIDO UAF protocol,

(b) the *FIDO Authenticator* which creates the key for the cryptographic challenge, combining for example a fingerprint input with the supported crypto algorithms, and

(c) the *Authenticator Specific Model (ASM)* which permits the communication of the FIDO UAF client with the Authenticator.

The *FIDO Authenticator* could be embedded on user's device or external hardware (that can be used in more devices). The matcher, which performs the user verification process, is a part of the authenticator. Tampering with the matcher could crucially affect the security of the protocol and therefore, it is recommended to run this module in a trusted environment. According to the FIDO UAF specifications, the envisaged methods for user verification are PIN and biometric-based. Nevertheless, FIDO UAF also supports location and pattern-based verification.
In the implementation of PIN/passcode-based user verification methods, it is important to specify the base of the numeric system (e.g. 10), the minimum length of the PIN, the maximum attempts before the authenticator blocks this method and the wait time after blocking.

RELYING PARTY

The Relying Party includes:

(a) the *Web Server* containing the service in which the user wants to be authenticated, and

(b) the *FIDO server* which ensures that only trusted applications are being used.

The FIDO Server can cryptographically verify that user's FIDO authenticator is indeed trusted and compliant with FIDO protocol using a process called *authenticator attestation*.

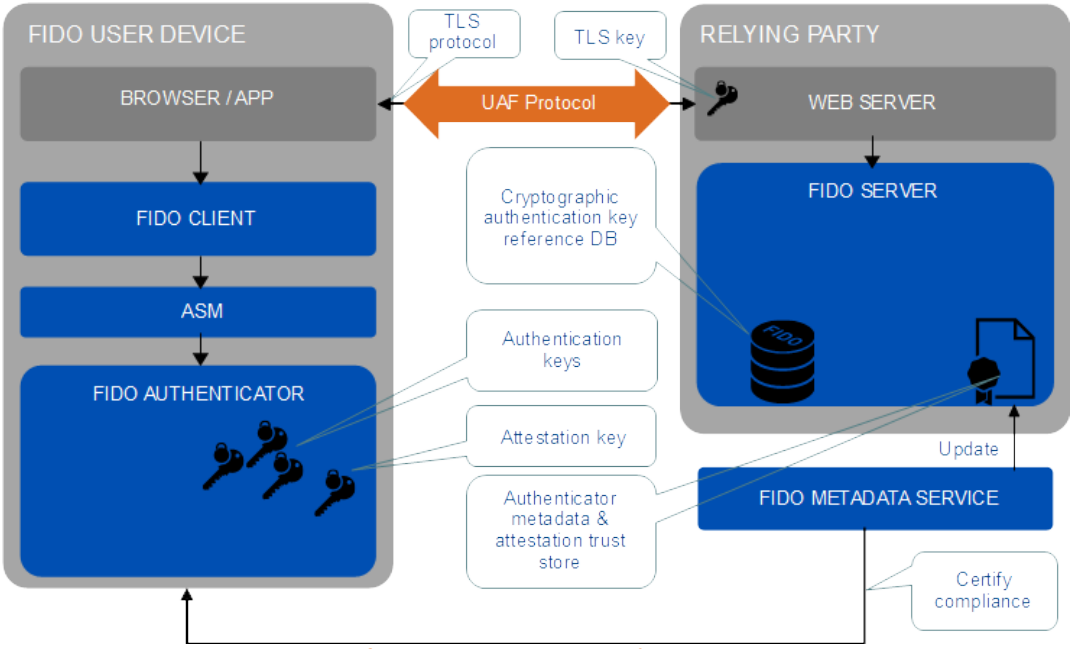

figure 1: FIDO UAF Protocol Overview

The communication between the client and the server is achieved using the TLS protocol. The communication between the client and the FIDO client, as well as between the FIDO client and the authenticator is achieved using the appropriate API (UAF API, ASM API).

2.2 FIDO UAF User Verification Methods

The following are the user verification methods supported on user's local device by FIDO UAF.

- Fingerprint
- Passcode
- Voiceprint
- Face print
- Location
- Eye print
- Pattern
- Handprint

In the UAF protocol specifications document, the envisaged user verification methods are PIN and biometric based.

FIDO UAF also supports silent authenticators which do not requiring any types of user verification or user presence check.

2.3 FIDO UAF Attestation Types

FIDO UAD proposes 3 types of attestation

1. Basic full

   In which a group of authenticators that share some common characteristics (i.e. same model), possess an attestation certificate and an attestation private key which they use to sign the registration object.

2. Basic surrogate

   In which the key registration object is signed using *Uauth.priv* key. It does not provide any cryptographic proof of the authenticators security characteristics. It is used if the authenticator is not able to have an attestation private key.

3. ECDAA

   Which proves the trust in the authenticator using Direct Anonymous Attestation cryptographic scheme (DAA) with Elliptic Curves.

   It is a more secure alternative than the basic full attestation which uses "group keys" and it combines security with privacy. In ECDAA if the key is stolen it does not affect other authenticators.

   Another solution to group keys is the use of individual keys combined with a Privacy-CA. However, involving a third party could involve other risks such as threats on user's privacy and high availability requirements on behalf of the Privacy-CA.

2.4 FIDO UAF Protocol Conversations

The core FIDO UAF protocol consists of the following conversations between the FIDO UAF Client and the FIDO Server:

1. Authenticator Registration
2. User Authentication
3. Transaction Confirmation
4. Authenticator Deregistration

The enrollment of the user in the authenticator does not concern FIDO.

1) AUTHENTICATOR REGISTRATION

The registration process allows the Relying Party to verify the authenticity of the FIDO Authenticator and register it among with the user's account. Once an authenticator has been validated, the Relying Party can assign a unique identifier number (*aaid*) to the authenticator that can be used in future communication between the two parties.

- ***Registration request and registration reply***

From a cryptographic point of view, after the registration request is being send by the FIDO Client, the Server will reply with a message containing the following parameters: *username*, *policy*, *appID*, *challenge*.
The *username* refers to the parameter that helps the Authenticator distinguish the different users (different keys) of the same application (or website), the *policy* refers to the Relying party's set of criteria concering the acceptable authenticators, the *appID* refers to the parameter used by UAF Client to determine if the application is authorized to use UAF protocol and the *challenge* refers to a random value send to protect against replay attacks.

- ***Key registration request***

After receiving the registration message, the FIDO UAF Client decides whether to proceed or not by discovering all the authenticators available that satisfy the Relying Party's policy using Authenticator Specific Module (ASM).
The FIDO UAF Client will also check the *appID* by asking from the Relying Party for the *facet* list which contains all the approved applications (or websites).
If there the authenticator(s) and the *appID* are indeed approved, the process continues with the UAF Client computing the *final challenge parameter (fcp)* which derives from the server *challenge*, the *appID*, some other data **and** the *KHAccessToken* which derives from *appID*, *personaID* (an identifier provided by ASM used to associate different registrations), *ASMToken* and the *callerID* (the ID the platform has assigned to the calling FIDO UAF Client). The FIDO UAF Client will finally send the *username*, the *hashed fcp* and the *hashed KHAccessToken* to the authenticator.

- ***Key registration reply***

The Authenticator after receiving these values, will prompt the user for authentication and generate afterwards a set of keys, a public and a private user authentication key *(Uauth.pub, Uauth.priv)* that will store in its secure storage and associate with the *username* and the *KHAccessToken*.
The Authenticator will create the *Key Registration Data (KRD)* which contains the *hashed fcp*, the *Uauth.pub*, the *aaid* (which is a unique identifier assigned to all FIDO Authenticators that share the same characteristics), the *attestation certificate* (related to related to the Attestation Key whose chain up to the Attestation Root Certificate proves trust by the FIDO Alliance) and some other values and signs it, using its *attestation private key*.
The key registration is subsequently send to the UAF Client and then to the FIDO Server which can verify whether the authenticator is trusted by the *aaid* and the *attestation certificate* and stores the *Uauth.pub* key in a database to authenticate the user in the future.

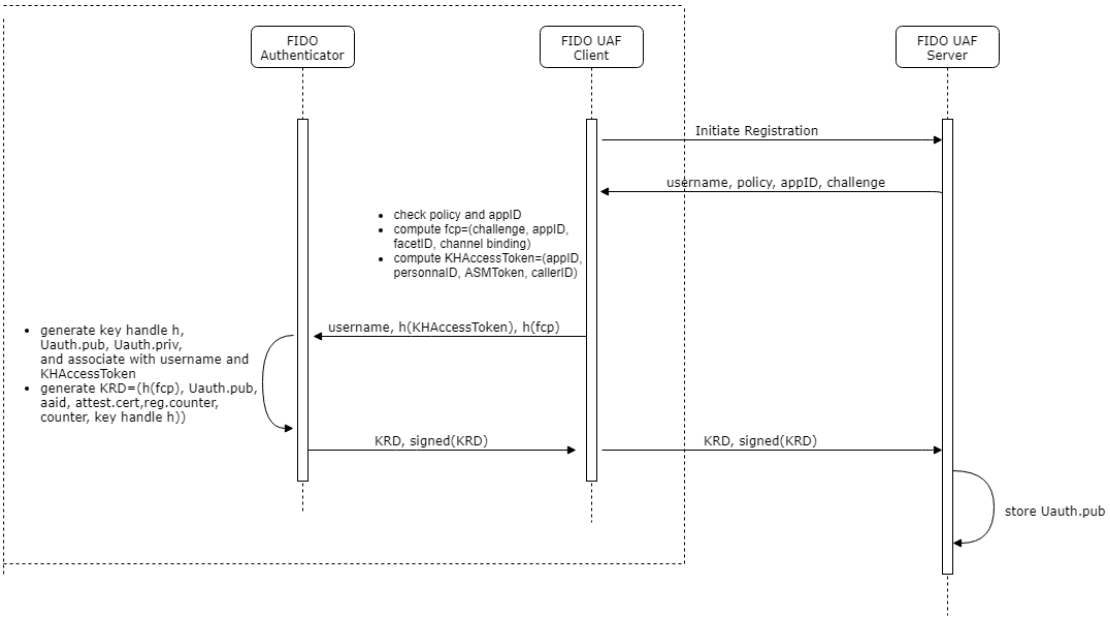

figure 2: FIDO UAF Registration Messages Flow

2) USER AUTHENTICATION & TRANSACTION CONFIRMATION

The user authentication process (and the transaction confirmation) is based on a cryptographic challenge-response scheme in which the user is prompt by the FIDO Server to be verified to the FIDO Authenticator which was used in the registration process.
Cryptographic analysis:

- ***Authentication Request and Authentication message***

From a cryptographic point of view, after the authentication request is being send by the FIDO Client, the Server will reply with a message containing the following parameters: the authenticator *policy*, the *appID* and the server *challenge*.

- ***Key Authentication request***

The FIDO UAF Client will check the *appID* and the *policy* to determine whether the application (or website) is trusted and whether UAF Authenticators meet the requirements by the Server's policy. The FIDO Client will afterwards compute the *fcp* and the fcp's hash as well as the *KHAccessToken* that will send to the UAF Authenticator.

- ***Key Authentication reply***

The UAF Authenticator will verify that the UAF Client is authorized to ask authentication for the specific user based on the *KHAccessToken*. In order to unlock the *Uauth.priv* key, the user will be triggered to enter a PIN or his fingerprint. The Authenticator will subsequently create the *SignedData* object which contains the *hashed fcp* and some other values and is signed by using the *UAuth.priv* key which is specific for the *appID* and the *username*.
The client will send this message to the FIDO Server that can cryptographically verify the response by using *Uauth.pub*.

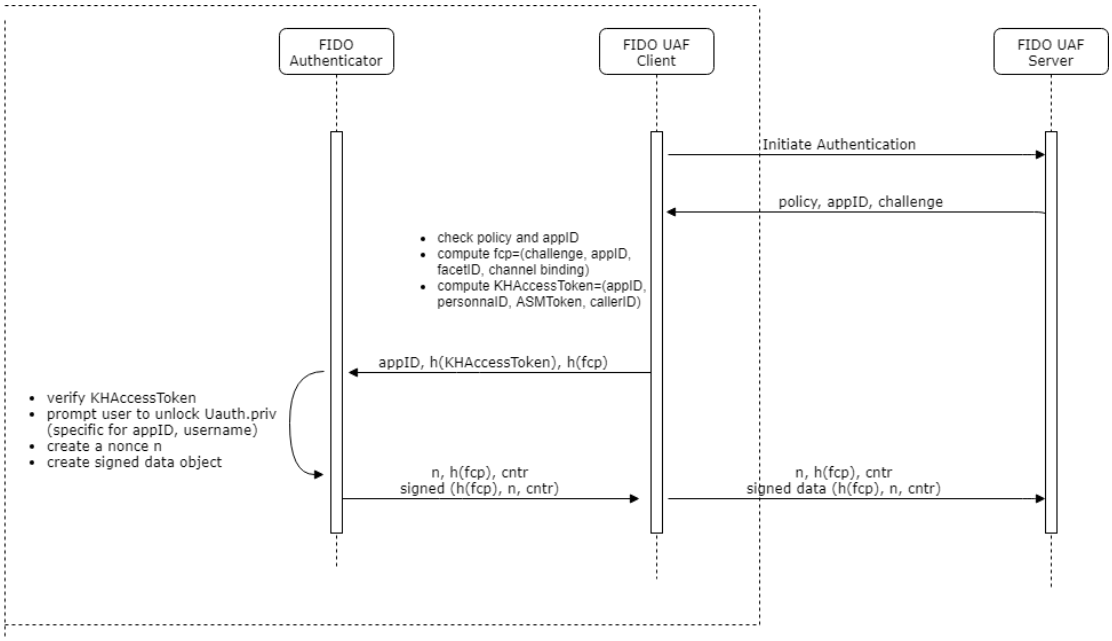

figure 3: FIDO UAF Authentication Messages Flow

3) AUTHENTICATOR DEREGISTRATION

Deregistration is required when the user account is removed at the relying party. The relying party can trigger the deregistration by asking the authenticator to delete the associated UAF credentials that are bound to the user account.
Cryptographic analysis:

- ***Deregister request***

The FIDO Client needs to be logged in to the Relying Party. The later, will send back to the FIDO Client a deregistration request containing the authenticators to be deleted.

- ***Deregister Authenticator request***

The FIDO Client will ask the Authenticator to delete the keys related to the Relying party by indicating the associated *aaid* and *keyid* (the id of the credentials).

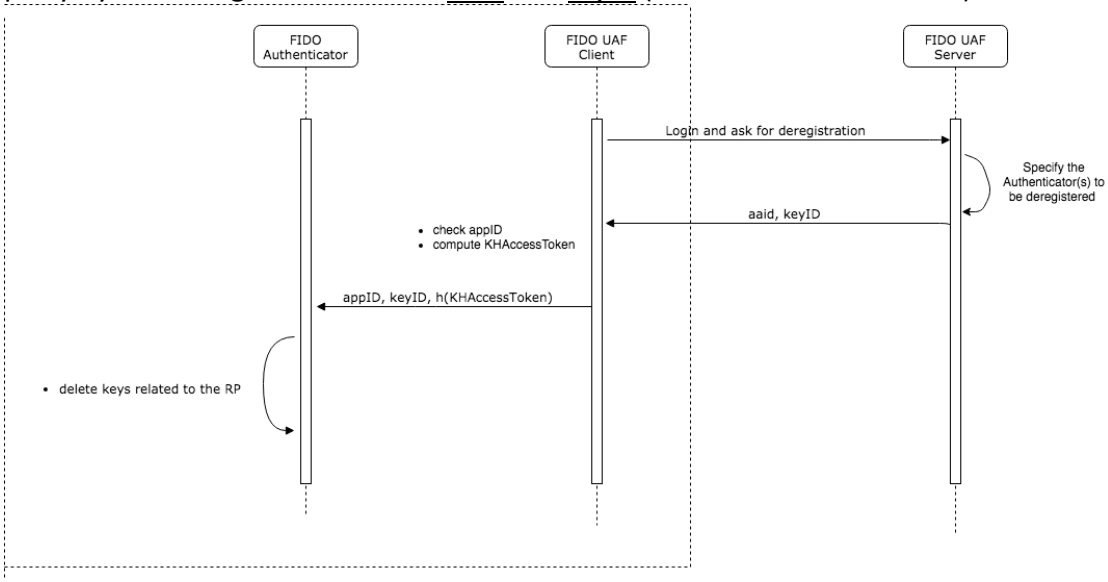

figure 4: FIDO UAF Deregistration Messages Flow

2.5 FIDO UAF v1.1 vs v1.2

document	FIDO UAF 1.1	FIDO UAF 1.2
1. Architectual Overview	-	-
2. Protocol Specification		▪ ***3.1.8 Client Data dictionary*** Alternative to fcp structure to support CTAP2 and WebAuthn Contains: challenge, origin (similar to facetID), hashAlg, token binding (similar to channel binding), extensions ▪ ***4.2.2 Revealing KeyIDs*** Advice concerning when keyID should be revealed to protect against attacks.
3. UAF Client API Transport	▪ ***4.5.1 TODO note:*** What does it occur and what should RP do when authenticator access is denied	▪ ***6.1.1 Android FIDO Client*** Should => Must *FIDO UAF Clients running on Android version 5 or later must not declare this permission and they also must not declare the related "uses-permission".*
4. UAF ASM API		▪ ***5.2 Java ASM API for Android*** ▪ ***5.3 C++ ASM API for iOS*** ▪ ***5.4 Windows ASM API*** ▪ ***6 CTAP2 Interface*** Which allows an authenticator to be used as external from FIDO2 or WebAuthn protocol. This section specifies the how the ASM should process the information received via FIDO CTAP2 Interface to the FIDO Authenticator. ▪ ***6.1 authenticatorMakeCredential*** ▪ ***6.2 authenticatorGetAssertion*** ▪ ***6.3 authenticatorGetNextAssertion*** ▪ ***6.4 authenticatorCancel*** ▪ ***6.5 authenticatorReset*** ▪ ***6.6 authenticatorGetInfo***
5. UAF Authenticator Commands		▪ ***6.1.3 Command Response*** Adds more information about RGB display ▪ ***6.3.4 Status Code & 6.4.4 Command Description***

		Bound authenticators can implement different binding method for the keys.
6. UAF APDU	-	-
7. Metadata Statement		▪ ***3.1 Authenticator Attestation GUID (AAGUID)*** An id assigned by the manufacturer that Indicates the type of the authenticator ▪ ***3.10 Extension Descriptor dictionary*** Tag parameter is added in the dictionary which refers to the tag of the extension ▪ ***3.11 Alternative Descriptions dictionary*** Which contains the description in different languages ▪ ***4 Metadata keys*** The following parameters are added to the Metadata Statement dictionary: -legalHeader -alternativeDescription -authenticationAlgorithm -publicKeyAlgEncodings -cryptoStrength -operatingEnv
8. Metadata Service		▪ ***3.1.2 Status Report dictionary*** The following parameters are added to the Status Report dictionary: -certificationDescriptor -certificateNumber -certificationPolicyVersion -certificationRequirementsVersion ▪ ***3.1.5 Metadata TOC Payload dictionary*** The following parameter is added to the Metadata TOC Payload dictionary: -legalHeader
9. Registry	-	-
10. UAF Registry		▪ ***5.1 User Verification Method Extension*** ▪ ***5.2 User ID Extension*** ▪ ***5.5 User Verification Caching*** This extension allows the RP to find out how long ago the user was authenticated.
11. AppID and FacetsID		▪ ***3.1.3.1 Dictionary TrustedFacetList***

	Which is an array of TrustedFacets (already defined)
12. ECDAA Algorithm	This specification includes the fixes of the issue regarding the Diffie-Hellman oracle w.r.t. the secret key of the TPM and regarding the potential privacy violations by fraudulent TPMs as proposed in [CCDLNU2017-DAA]. ▪ ***2.2 Architecture Overview*** It is clarified that the ECDAA-Join operation takes place between the Authenticator and the ECDAA Issuer which is the authenticator vendor. There are also some differences concerning the calculation of some values in ECDAA-Join which affect -ECDAA-Join Algorithm -ECDAA-Join Split between Authenticator and ASM -ECDAA-Join Split between TPM and ASM
13. Security Reference	▪ ***3 Attack Classification*** Attack classes are specified [AC1, AC2, AC3, AC4, AC5, AC6] ▪ ***4 FIDO Security Goals*** [SG-16] Assessable level of security ▪ ***5 FIDO Security Measures*** [SM-16] Use of strong, modern Cryptographic Primitives [SM-17] Resistance to Side Channel Attacks [SM-18] Resistance to Injected Faults in Cryptographic Functions [SM-19] Bounded Probability of a Birthday Collision. [SM-20] Individual authenticators are indistinguishable provided authenticators [SM-21] Authentication and replay-resistance [SM-22] Certified FIDO Authenticators fully described by the vendor, and tested [SM-23] Key Handles containing a key are cryptographically linked with the Authenticator

[SM-24] Design, implementation and manufacture of certified FIDO Authenticators
[SM-25] Depending on the certification level, certified authenticators are required
[SM-26] Input Data Validation
[SM-27] Protection of user verification reference data and biometric data.
[SM-28] Resistance to Fault Injection Attacks
[SM-29] Resistance to Remote Timing Attacks

- ***7 Threats to Client Side***

T-1.1.2 Homograph Mis-Authentication
T-1.4.15 Compromised the internal PRNG state and the entropy source
T-1.4.16 Compromised entropy source after successful seeding during initialization
T-1.4.17 Compromised the internal PRNG state, but not the entropy source
T-1.4.18 Bad Key Generation
T-1.4.19 Local external side channel attacks
T-1.4.20 Internal side channel attacks
T-1.4.21 Error injection during key or signature generation
T-1.4.22 Birthday Paradox Collision
T-1.4.23 Privacy Reduction
T-1.4.24 Covert Channel
T-1.4.25 Substitution of Protected Information
T-1.4.26 Compromise of Protected Information
T-1.4.27 Signature or registration counter non-monotonicity
T-1.4.28 Hostile ASM / Client
T-1.4.29 Debug Interface
T-1.4.30 Fault induced by malformed input
T-1.4.31 Fault Injection Attack
T-1.4.32 Remote Timing Attacks
T-2.2.2 Linking through compromised Relying Party database
T-5.1.3 Physical Attack on a User Presence Authenticator
T-5.1.4 Physical Attack

14. Glossary	The following definitions have been added to the Glossary: -ECDAA -Test of User Presence -User Presence Check It is also added in the definition of FIDO Authenticator that the *"Authenticators specify in the Metadata Statement whether they have exclusive control over the data being signed by the Uauth key."*

2.6 FIDO UAF and TEE, SE, TPM

FIDO proposes a secure implementation through Trusted Execution Environment (TEE) and Secure Element (SE). Trusted Platform Module (TPM) does not yet support the FIDO UAF attestation model. *(These technologies are explained in detail in the 3rd chapter of this document)*

The authenticator might be implemented in separate hardware or trusted execution environments. The specifications of the protocol do not oblige the use of TEE or SE, nevertheless they underline the importance of protecting some specific components suggesting that the optimum solution is the TEE or SE, depending on the component. It is suggested that the authenticator should be implemented in a TEE (using a special "Trustlet", trusted application running inside TEE to perform the UAF operations) and communicate with an SE within the Authenticator, where all the important keys could be stored.

Generally, it is important to protect the keys and the functions that produce the keys whose security is crucial for the security of the whole protocol.

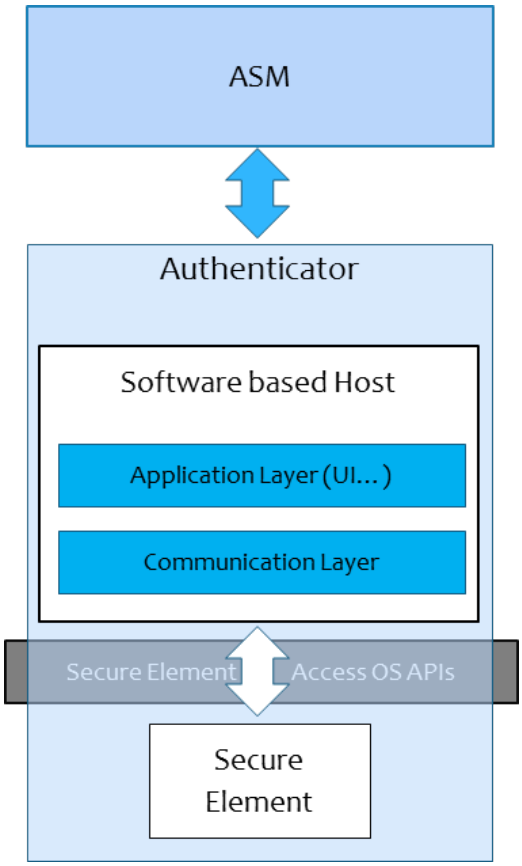

figure 5: FIDO Authenticator's Internal Architecture

The important functions and keys that could be implemented in a trusted environment are listed in the following table:

Secure Element	• Attestation private key [highly recommended] • Matcher • Crypto kernel • User Verification Model
TEE	• Matcher • Crypto kernel • User Verification Model • Transaction Confirmation Display (implemented with Trusted UI) • Wrap.sym key • UAuth.priv keys

	• Liveness Detection/Presentation Attack Detection (in the case of PIN based matching)
Trusted Computing Base	• Facet Mechanism

Matcher: By definition, the matcher component is part of the authenticator. This does not impose any restrictions on the authenticator implementation, but implementers need to make sure that there is a proper security boundary binding the matcher and the other parts of the authenticator together.
Crypto kernel: The crypto kernel is a module of the authenticator implementing cryptographic functions (key generation, signing, wrapping, etc) necessary for UAF, and having access to UAuth.priv, Attestation Private Key and Wrap.sym (symmetrically encrypted key handles).
User Verification Module: If the User Verification Module is inside the Host, for example in the context of the TEE, the UserVerificationToken shall be generated in the Host and not in the SE.
This specification doesn't specify how exactly user verification must be performed inside the authenticator. Verification is considered to be an authenticator, and vendor, specific operation. However, it is proposed how the vendor User Verify command could be bound to UAF Register and Sign command by using a UserVerificationToken.

2.7 FIDO U2F Overview

The Universal Second Factor (U2F) protocol allows online services to augment the security of their existing password infrastructure by adding a strong second factor to user login. The user logs in using a *username* and password as before but can also present a second factor device any time he chooses.
FIDO supports software-based techniques but suggests the use of the appropriate hardware. Hardware supporting U2F is compatible with modern devices without the need of additional drivers.

CLIENT

The client side includes of the *FIDO client* and the *FIDO U2F device*. The FIDO client is typically a web browser which relays the messages between the FIDO U2F device (or U2F token) and the Relying Party.
The FIDO U2F device is responsible for the generation of U2F tokens which provide cryptographic assertions used by the Relying Parties to verify their authenticity. U2F Tokens are typically small special-purpose devices that aren't directly connected to the Internet.

RELYING PARTY

The Relying Party includes the *Web Server* containing the service in which the user wants to be authenticated and the *FIDO server* which ensures that only trusted applications are being used.
FIDO Server can cryptographically verify that user's FIDO U2F device is indeed trusted and compliant with the FIDO protocol.
The communication between the Relying Party and the FIDO client is achieved using the appropriate JavaScript API. FIDO also standardizes the form of the messages exchanged between the FIDO Client and the U2F device sent over NFC, Bluetooth or USB.

2.8 FIDO U2F Protocol Conversations

The U2F protocol supports 2 operations: *registration* and *authentication*.

1) REGISTRATION

The registration operation introduces the relying party to a freshly-minted key pair produced by the U2F device. The browser implementation (using the appropriate JavaScript code) can ensure that the user is aware of this dialogue.
Cryptographic analysis:

- ***Registration request***

The FIDO Client contacts the relying party to obtain the *challenge* and creates the *hashed challenge parameter* which also includes other *client data*: *type, origin* (facetID of the caller)*, channel id public key*. The FIDO Client will send the *challenge parameter* among with the *appID* to the U2F device.

- ***Registration response***

The U2F device, after ensuring user's presence, will perform some cryptographic operations to generate the *user public* and *private key* and the *key handle* which facilitates the identification of the generated keypair. Afterwards, the U2F device will create the response message to the FIDO Client which includes the following parameters: *user public key, key handle length, key handle, attestation certificate* and the *signature* of the *appID, challenge parameter, key handle and user public key*. The FIDO Client will forward this message to the Relying Party which will store the user *public key* and the *key handle*.

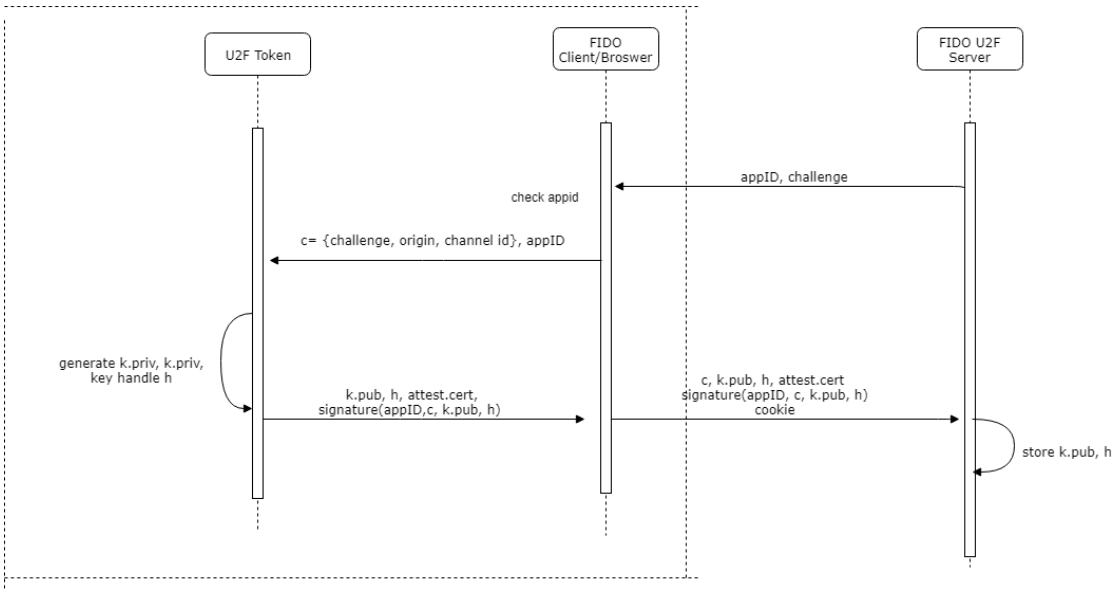

figure 6: FIDO U2F Registration Messages Flow

2) AUTHENTICATION

The authentication process proves possession of a previously-registered key pair in order to verify that the U2F device is already registered to the service thus, it is

trusted. In the authentication process the user could be asked to verify its presence ex. by pushing a button, before the U2F device signs the *challenge*.
Cryptographic analysis:

- ***Authentication request***

The FIDO Client contacts the relying party to obtain the *challenge* and creates the *hashed challenge parameter* which also includes other *client data: type, origin, channel id public key.* The FIDO Client can examine whether the U2F device is registered by sending the *challenge parameter* among with the *appID*, the *key handle* and the *key handle length* to the U2F device.

- ***Authentication response***

The U2F device will retrieve the key pair using the *key handle* and create the response message to the FIDO Client, which includes a *counter* and a *signed object* containing the *appID*, the *counter* and the *challenge* parameter.
The FIDO Client will forward these values along with the *challenge parameter* to the Relying Party, which can verify the validity of the signature using the *user public key* obtained during registration.

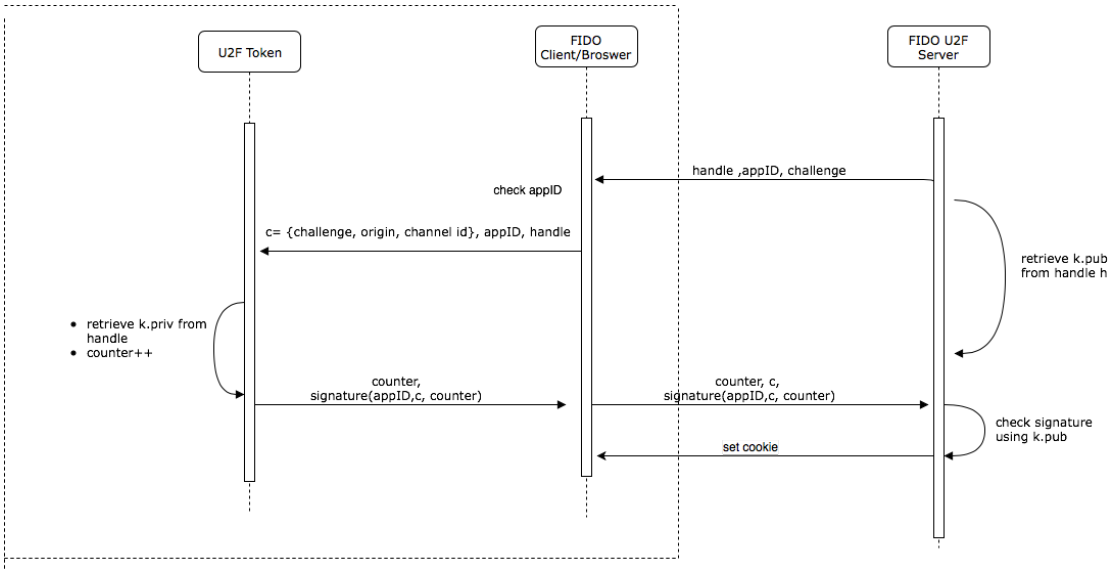

figure 7: FIDO U2F Authentication Messages Flow

2.9 FIDO 2 Overview

The FIDO 2 protocol is a combination of 3 protocols:

- FIDO UAF/U2F
- FIDO CTAP (for external authenticators)
- WebAuthn (JavaScript API)

FIDO 2 comes to help the integration of the FIDO protocol (especially the FIDO UAF) on user's machines. In the new version of the protocol the FIDO Server is a universal server which implements the server's side FIDO protocol and communicates with the

metadata service. In addition, the browser implements a JavaScript API which facilitates the communication with the FIDO Client. The FIDO Client now is implemented by the OS platform which, for the time being, it's Android or Windows. Therefore, the idea of a Web Authentication API is introduced in the new specifications that allow the Relying Party to communicate with the authenticator through the client (browser and OS).
FIDO 2 refers to the registration (make credential) and authentication (get assertion) process. Authenticator management actions such as credential deletion (deregistration) is the responsibility of a user interface and is deliberately omitted from the API exposed to scripts.
The ***FIDO CTAP*** document describes the communication between FIDO Clients and external authenticators.
The ***WebAuthn*** is the W3C candidate recommendation for the implementation of FIDO in browsers.
The protocol messages exchanged bear a great resemblance to the previous FIDO UAF and U2F protocol specifications. The scheme follows the logic of UAF protocol messages exchanged. The external authenticator can be a USB, NFC or Bluetooth device as it was on the U2F specifications.
The below scenarios will further explain the adoption of FIDO 2:

- Registration using a phone

  The user will sign in to his existing account with the authentication method he was already using and select to register this device on this specific webpage or application. Then the user will enter his authentication method i.e. fingerprint and complete the authenticator registration step.

- Authentication on laptop

  The user will enter the website by his broswer and select to sign in using his previously registered mobile phone. The user will see a message on his mobile phone to select the account that he wants to enter (if more than one) and then enter his authentication method i.e. fingerprint.

  If this step is completed successfully the user will be able to access the webpage from his browser on his laptop.

2.10 FIDO 2 Protocol Conversations

1) REGISTRATION - AUTHENTICATOR_MAKE_CREDENTIAL

The registration refers to the enrollment of the specific authenticator to the Relying Party. The bears great resemblance to UAF and U2F.
The authenticator will generate a key pair and store the public value in the Relying Party's server in order to be authenticated.
Cryptographic analysis:

- ***Registration request***

The FIDO Client contacts the relying party to obtain the *challenge* the *userInfo* and the *relyingPartyInfo*. The Client will create the *clientData* parameter and forward it to the authenticator among with the *challenge* the *userInfo* and the *relyingPartyInfo*.

- ***Registration response***

The Authenticator after receiving these values, will prompt the user for authentication and generate afterwards a set of keys, a public and a private user authentication key that will store in its secure storage and create the *attestationObject* which contains information about the key and attestation type and format and will return it to the Client. The Client will forward the *attestationObject* among with the *credentialId* and the *ClientData*

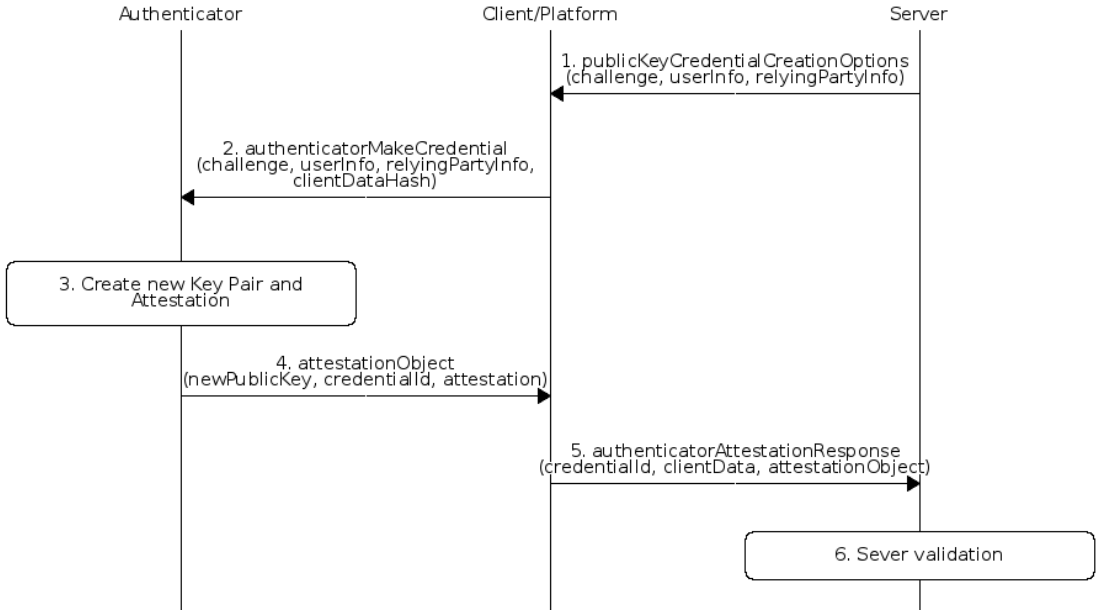

figure 8: FIDO 2 Registration Messages Flow

source: https://developers.yubico.com/FIDO2/

2) AUTHENTICATION– AUTHENTICATOR_GET_ASSERTION

The authentication process proves possession of a previously-registered key pair in order to verify that the device is already registered to the service thus, it is trusted. In the authentication process the user could be asked to verify its presence ex. by pushing a button, before the device signs the *challenge*.
Cryptographic analysis:

- ***Authentication request***

The FIDO Client contacts the relying party to obtain the *challenge* and creates the *clientData* object which sends to the Authenticator among with the *relyingPartyInfo*.

- ***Authentication response***

The authenticator will retrieve the key pair and send to the Client the *signature* among with the *credentialId* and the *authData*. The Client will forward to the server the information received from the authenticator plus the *clientData* object.

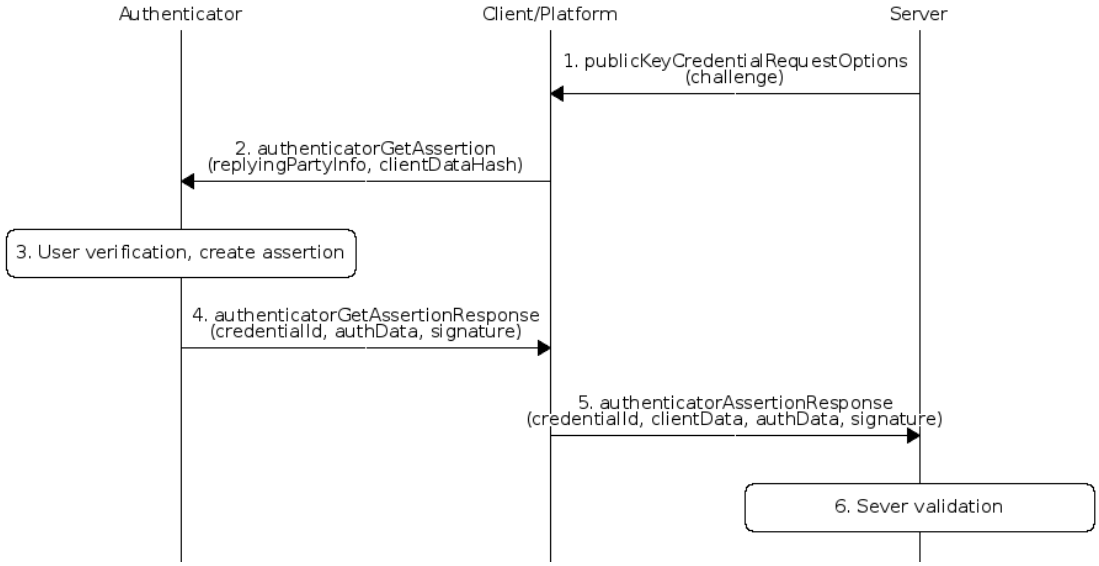

figure 9: FIDO 2 Authentication Messages Flow

source: https://developers.yubico.com/FIDO2/

2.11 FIDO 2 Client: Microsoft Edge and Windows Hello

WebAuthn API in Microsoft Edge enables web applications to use Windows Hello biometrics for user authentication. Using Web Authentication combined with Windows Hello, the server sends down a plaintext challenge to the browser. Once Microsoft Edge is able to verify the user through Windows Hello, the system will sign the challenge with a private key previously provisioned for this user and send the signature back to the server.

When you use the makeCredential method, Microsoft Edge will first ask Windows Hello to use face or fingerprint identification to verify that the user is the same user as the one logged into the Windows account. Once this step is completed, 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. If the user doesn't have a TPM enabled device, these keys will be stored in software. These credentials are created per origin, per Windows account, and will not be roamed because they are tied to the device. This means that you'll need to make sure the user registers to use Windows Hello for every device they use.

Once the getAssertion call is made, Microsoft Edge will show the Windows Hello prompt, which will verify the identity of the user using biometrics. After the user is verified, the challenge will be signed within the TPM and the promise will return with an assertion object that contains the signature and other metadata for you to send to the server:

2.12 FIDO 2 Client: Browsers

- Firefox

- Chrome
- Microsoft Edge

2.13 FIDO 2 Client: OS

- Windows
- Android
- Mac OS (via 3rd party development)

2.14 FIDO 2 vs FIDO UAF and U2F

Authenticators that only support the FIDO U2F Attestation Statement Format have no mechanism to store a user handle, so the returned userHandle will always be null.

CHAPTER 3: TRUST EXECUTION ENVIRONMENT

GlobalPlatform defines a TEE as a secure area in the main processor in a smart phone (or any connected device) that ensures sensitive data is stored, processed, and protected in an isolated, trusted environment.[1]
GlobalPlatform with its alliances have documented the specifications of this technology.
REE
The Rich Execution Environment is comprising at least one Rich OS and all other components of the device (SoCs, other discrete components, firmware, and software) which execute, host, and support the Rich OS. WARNING: In the previous version of the Global Platform specification document the REE was considered to be everything outside of the Trust Execution Environment under consideration. In the new definition, other entities are acknowledged.
TEE
An execution environment that runs alongside but isolated from a Rich Execution Environment (REE). A TEE meets certain security capabilities and requirements: It protects from general software attacks and can, therefore, resist a set of defined threats. In general terms, the TEE offers an execution space that provides a higher level of security than a Rich OS, although the TEE is not as secure as an SE, the security it offers is sufficient for most applications.
A typical board level chipset architecture of a mobile device, which consist a Printed Circuit Board (PCB) and several components, is depicted in the image below:

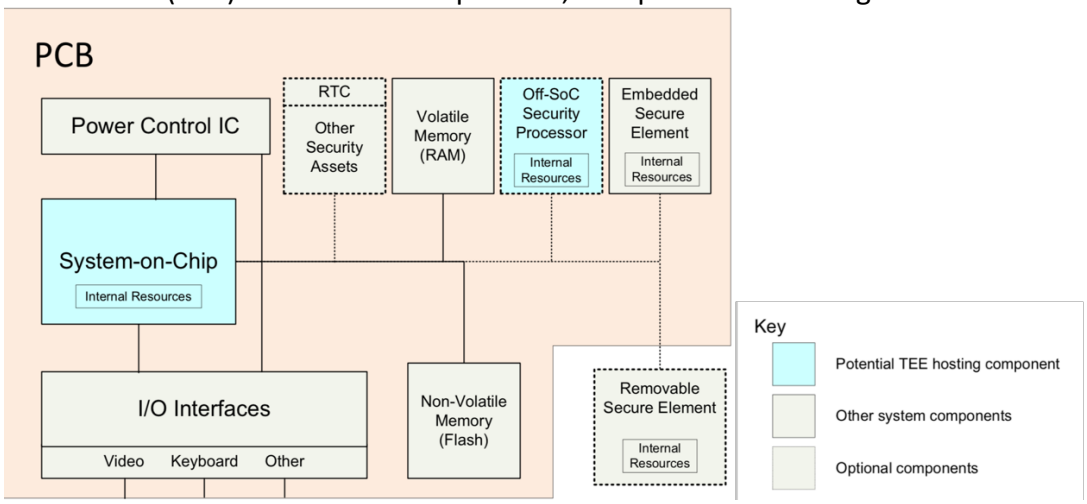

figure 10: PCB Architecture

REE has access to the untrusted resources, which can be implemented on-chip (Soc) or off-chip in other components on the PCB. The REE cannot access the trusted resources. This access control is enforced through physical isolation, hardware logic based isolation, or cryptographic isolation methods. The only way for the REE to get access to trusted resources is via any API entry points or services exposed by the TEE

1 https://www.globalplatform.org/certification/TEE_Security_Certification_Presentation-FINAL1.pdf

and accessed through, for example, the TEE Client API.[2] There is a REE Communication Agent which provides REE support for messaging between the Client Application and the Trusted Application.

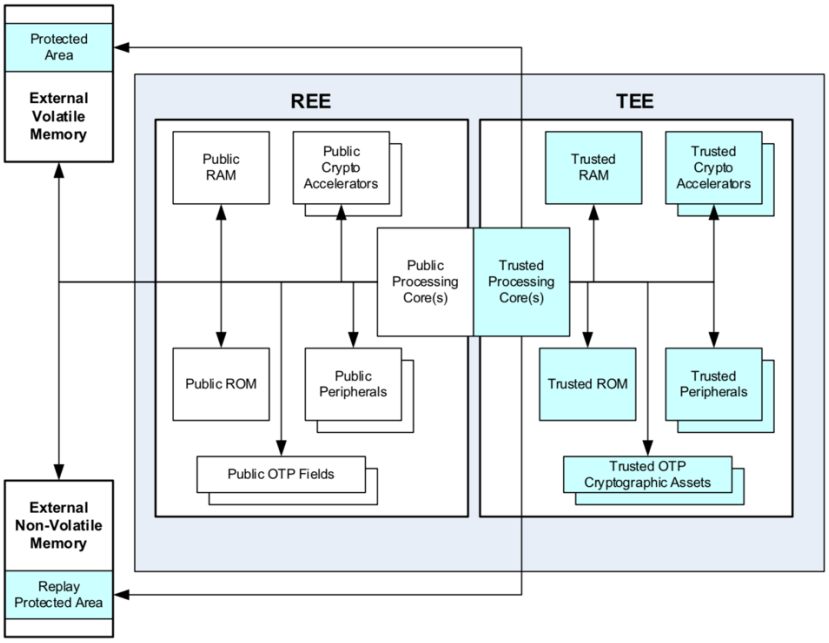

figure 11: REE and TEE

There is no specific implementation architecture for these components (TEE, REE, SE). The TEE implementation could be either hardware or software.
Trusted OS components consist of:

- The Trusted Core Framework (also part of TEE Internal Core API) which provides OS functionality to Trusted Applications.
- Trusted device drivers which aid the communication with trusted peripherals.

There is also the TEE Communication Agent which works with the REE Communication Agent to safely transfer the messages between the Client Application (CA) and the Trusted Application (TA). The CA will create a session in order to communicate with the TA.
(This will be explained more thoroughly in the Client API paragraph)
Each TA has a TA interface which encompass a set of entry point functions that the Trusted Core Framework implementation calls to inform the TA about life-cycle changes (ex. creates an instance) and to relay communication between Clients and the TA (ex. notifies the instance that a new client is connecting or when it invokes commands). Once the Trusted Core Framework has called one of the TA entry points, the TA can make use of the TEE Internal Core API to access the facilities of the Trusted OS.

[2] GPD_TEE_SystemArch_v1.1_Public_Release.pdf

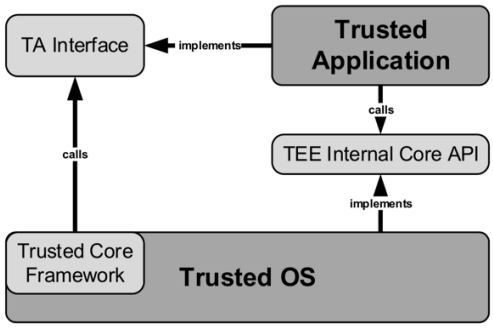

figure 12: TEE Internal Functions

An overview of TEE Architecture (and the respected documents from GlobalPlatform specifications) can be depicted in the following figure:

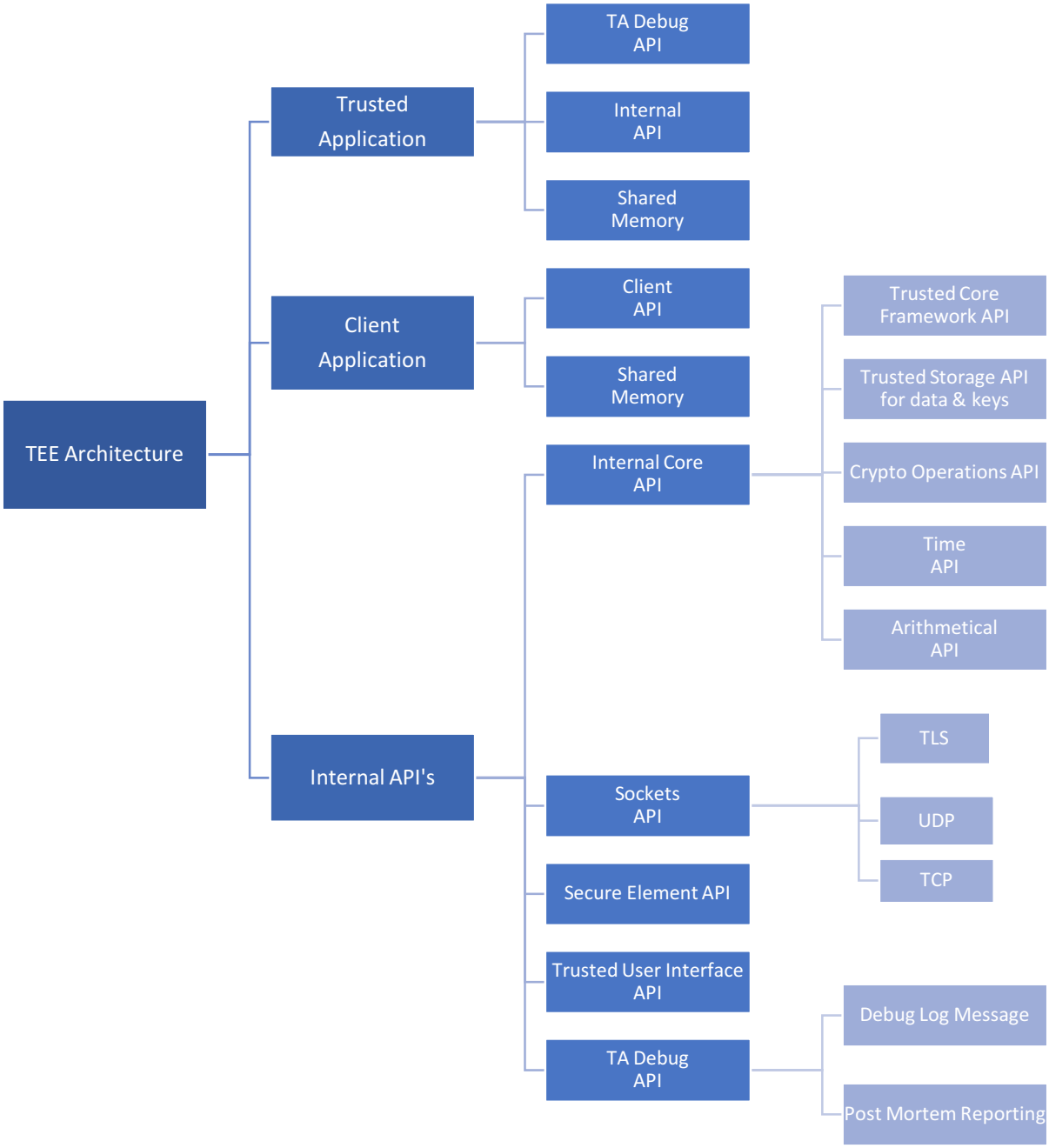

figure 13: Global Platform Specifications

3.1 TEE Client API: Shared Memory and Functions

The Client API provides a communication channel between the CA and the TA. Within the trusted environment this specification identifies two distinct classes of component: the hosting code of the TEE itself, and the Trusted Applications which run on top of it. There is no definition of the expected implementation in the TEE specification document.
Within the REE this specification identifies three distinct classes of component:

- The CA which make use of the TEE Client API.
- The TEE Client API library implementation.
- The communications stack which is shared amongst all CA, and whose role is to handle the communications between the REE and the TEE.

In a **session**, the logical connection exists between a CA and a specific TA. A Session is opened by the CA within the scope of a particular TEE Context.
When creating a new Session, the CA must identify the TA's which it wishes to connect to using the Universally Unique Identifier (UUID) of the TA. The open session operation allows an initial data exchange to be made with the TA, if this is required in the protocol between the CA and the TA.
The Session MAY be opened using a specific connection method that can carry additional connection data, such as data about the user or user-group running the CA, or about the CA itself.
When a CA creates a **session** with a TA, it connects to an **instance** (which can be one for all sessions or a different one for each session) of that TA and invokes **commands** (a message including a Command Identifier and Operation Parameters to initiate an operation) using the Client API. It is up to the TA to define the combinations of commands and their parameters that are valid to execute.
A TA instance has physical memory address space which is separated from the physical memory address space of all other TA instances. The TA instance memory space contains the TA instance heap and writable global static data.
All code executed in a TA is executed by **tasks** which keep a record of execution history. Tasks MUST be created every time the Trusted OS calls an entry point of the TA.

1) SHARED MEMORY

The shared memory used to transfer data between CA and TA can be either existing in the CA memory which is subsequently registered with the TEE Client API or memory which is allocated on behalf of the CA using the TEE Client API.
When possible the implementation of the communications channel beneath the TEE Client API should try to directly map Shared Memory in to the TA memory space, enabling true zero-copy data transfer. In cases when zero-copy data transfer is not possible (ex. TEE and CA do not have access to the same physical memory system) the specification determines synchronization points aiming to synchronize the TEE Client API and the shared memory block.
The memory buffer used in an operation may be released immediately after its completion and the TA must not be able to access it.

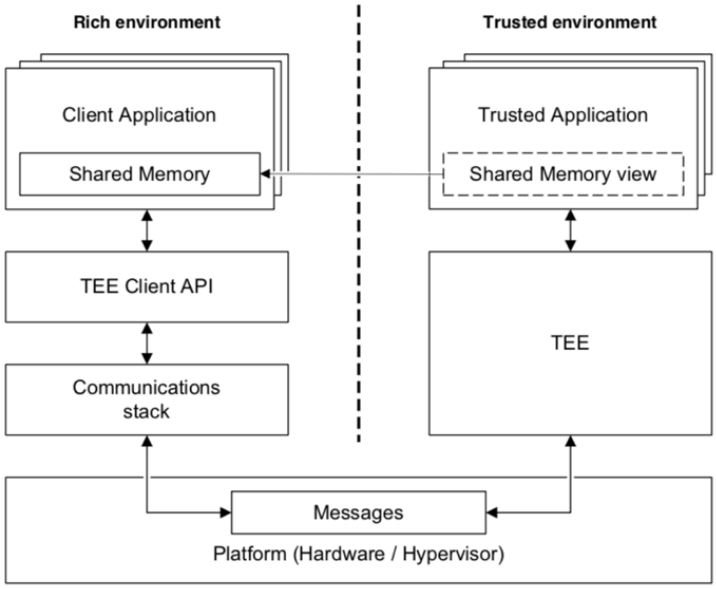

figure 14: TEE scheme

2) TEE CLIENT API FUNCTIONS

TEE Client API defines a set of C functions and structures which enable the developers to perform the required steps for establishing a connection and exchanging data between CA and TA. The following ones are used perform a typical operation inside TEE:

1. **TEEC_InitializeContext**

   The TEEC_Context is the main logical container which links a CA with a particular TEE.

   This function initializes a new TEE Context, forming a connection between CA and the TEE identified by the string identifier name.

```
TEEC_Result TEEC_InitializeContext(
      const char*   name,
      TEEC_Context* context)
```

   name: refers to the name of TEE connected to

   context: a TEEC_Context structure

2. **TEEC_FinalizeContext**

   This function finalizes an initialized TEE Context, closing the connection between the CA and the TEE. The CA MUST only call this function when all Sessions inside this TEE Context have been closed and all Shared Memory blocks have been released.

```
void TEEC_FinalizeContext(
    TEEC_Context* context)
```

3. **TEEC_RegisterSharedMemory**

   This function registers a block of existing CA memory as a block of Shared Memory within the scope of the specified TEE Context, in accordance with the parameters which have been set by the CA inside the sharedMem structure.

```
TEEC_Result TEEC_RegisterSharedMemory(
    TEEC_Context*      context,
    TEEC_SharedMemory* sharedMem)
```

sharedMem: MUST point to the Shared Memory structure defining the memory region to register

4. **TEEC_AllocateSharedMemory**
   This function allocates a new block of memory as a block of Shared Memory within the scope of the specified TEE Context, in accordance with the parameters which have been set by the CA inside the sharedMem structure.

```
TEEC_Result TEEC_AllocateSharedMemory(
    TEEC_Context*      context,
    TEEC_SharedMemory* sharedMem)
```

5. **TEEC_ReleaseSharedMemory**
   This function deregisters or deallocates a previously initialized block of Shared Memory.

```
void TEEC_ReleaseSharedMemory (
    TEEC_SharedMemory* sharedMem)
```

6. **TEEC_OpenSession**
   The TEEC_Session is a logical container which links a CA with a particular TEE.
   This function opens a new Session between the CA and the specified TA.

```
TEEC_Result TEEC_OpenSession (
    TEEC_Context*    context,
    TEEC_Session*    session,
    const TEEC_UUID* destination,
    uint32_t         connectionMethod,
    const void*      connectionData,
    TEEC_Operation*  operation,
    uint32_t*        returnOrigin)
```

session: a pointer to a Session structure to open
destination: a pointer to a structure containing the UUID (which is used to uniquely identify the TA) of the destination TA
connectionMethod: the method of connection to use
connectionData: any necessary data required to support the connection method chosen
operation: a pointer to an Operation containing a set of Parameters to exchange with the TA
returnOrigin: a pointer to a variable which will contain the return origin

7. **TEEC_CloseSession**
   This function closes a Session which has been opened with a TA.
   All Commands within the Session MUST have completed before calling this function.

```
void TEEC_CloseSession (
    TEEC_Session* session)
```

8. **TEEC_InvokeCommand**

   This function invokes a Command within the specified Session. The set of commands depend on the TA.

```
TEEC_Result TEEC_InvokeCommand(
    TEEC_Session*     session,
    uint32_t          commandID,
    TEEC_Operation*   operation,
    uint32_t*         returnOrigin)
```

   commandID is an identifier that is used to indicate which of the exposed Trusted Application functions should be invoked

9. **TEEC_RequestCancellation**

   This function requests the cancellation of a pending open Session operation or a Command invocation operation. Also, in error events that the client dies, it MUST seem as a cancelation event to the TA.

   As this is a synchronous API, this function must be called from a thread other than the one executing the TEEC_OpenSession or TEEC_InvokeCommand function.

   It is unsure whether the operation will be cancelled by the time the function returns while at the same time the TA MAY ignore the cancellation request

```
void TEEC_RequestCancellation(
    TEEC_Operation* operation)
```

The consecutive functions come in pairs:

- TEEC_InitializeContext / TEEC_FinalizeContext
- TEEC_OpenSession / TEEC_CloseSession
- TEEC_RegisterSharedMemory / TEEC_ReleaseSharedMemory
- TEEC_AllocateSharedMemory / TEEC_ReleaseSharedMemory

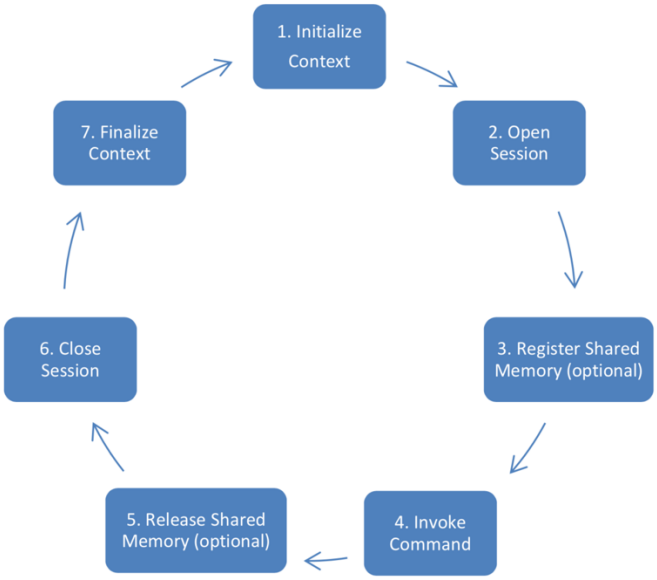

figure 15 ReCRED_D3.3_Description_of_DCA_protocols_and_technology_support

3.2 Trusted User Interface API (TUI)

The Trusted User Interface (TUI) API permits the display of screens to the user and achieves three objectives:

- Secure Display: Information displayed to the user cannot be accessed, modified, or obscured by any software within the REE or by an unauthorized application in the TEE.
- Secure Input: Information entered by the user cannot be derived or modified by any software within the REE or by an unauthorized application in the TEE.
- Secure Indicator: The user can be confident that the screen displayed is actually a screen displayed by a TA.

TUI can be used in authentication transactions (ex. PIN entry or username and password) or message functionalities (ex. Transaction confirmation).

The peripherals that are related to the User Interface (such as touchscreen or keyboard) must be wired to the device. When the TA reserves the resources for the TUI, the peripherals must not be accessible by the REE. TUI screens must be displayed in the foreground, that is the reason why it is recommended to be close to full screen size.

There is a security indicator which indicates when TEE is used so that the is informed of the level of trust.

TUI also supports as input images in Portable Network Graphics (PNG) format or text, characters of the ASCII table Characters in the interval [Unicode (U+0020) - Unicode (U+007D)].

TUI sessions must be terminated in events related to power management (ex. Device turn off or reset).

3.3 TEE Internal Core API

TEE Client API defines a set of C functions and structures which enable the developers to perform the required steps for establishing a connection and exchanging data between CA and TA. The following ones are used perform a typical operation inside TEE:

1. **TA_CreateEntryPoint**
   This is the Trusted Application constructor. It is called once and only once in the life-time of the Trusted Application instance. If this function fails, the instance is not created

```
TEE_Result TA_EXPORT TA_CreateEntryPoint (void);
```

2. **TA_DestroyEntryPoint**
   This is the Trusted Application destructor. The Trusted Core Framework calls this function just before the Trusted Application instance is terminated. The Framework MUST guarantee that no sessions are open when this function is called.

```
void TA EXPORT TA DestroyEntryPoint (void);
```

3. **TA_OpenSessionEntryPoint**
   This function is called whenever a client attempts to connect to the Trusted Application instance to open a new session. If this function returns an error, the connection is rejected and no new session is opened.

```
TEE_Result TA_EXPORT TA_OpenSessionEntryPoint (
uint32_t paramTypes,
[inout]          TEE_Param params[4],
[out][ctx]  void** sessionContext);
```

4. **TA_CloseSessionEntryPoint**
   This function is called when the client closes a session and disconnects from the Trusted Application instance. The Implementation guarantees that there are no active commands in the session being closed. The session context reference is given back to the Trusted Application by the Framework.

```
void TA_EXPORT TA_CloseSessionEntryPoint (
 [ctx] void*     sessionContext);
```

5. **TA_InvokeCommand**
   This function is called whenever a client invokes a Trusted Application command. The Framework gives back the session context reference to the Trusted Application in this function call.

```
TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint (
      [ctx]       void*       sessionContext,
                  uint32_t          commandID,
                  uint32_t          paramTypes,
      [inout]     TEE_PARAM         params[4]);
```

- ***Other Functions Used for the implementation:***

Operation

1. **TEE_AllocateOperation**

   The TEE_AllocateOperation function allocates a handle for a new cryptographic operation and sets the mode and algorithm type. If this function does not return with TEE_SUCCESS then there is no valid handle value

```
TEE_Result TEE_AllocateOperation (
      TEE_OperationHandle*   operation,
      uint32_t               algorithm,
      uint32_t               mode,
      uint32_t               maxKeySize);
```

2. **TEE_SetOperationKey**

   The TEE_SetOperationKey function programs the key of an operation; that is, it associates an operation with a key. The key material is copied from the key object handle into the operation. After the key has been set, there is no longer any link between the operation and the key object. The object handle can be closed or reset and this will not affect the operation. This copied material exists until the operation is freed using TEE_FreeOperation or another key is set into the operation.

```
TEE_Result TEE_SetOperationKey (
      TEE_OperationHandle    operation,
      TEE_ObjectHandle       key);
```

3. **TEE_AsymmetricSignDigest**

   The TEE_AsymmetricSignDigest function signs a message digest within an asymmetric operation. Note that only an already hashed message can be signed.

   This function can be called only with an operation of the following EC algorithms:

   - TEE_ALG_ECDSA_SHA1 (if supported)
   - TEE_ALG_ECDSA_SHA224 (if supported)
   - TEE_ALG_ECDSA_SHA256 (if supported)
   - TEE_ALG_ECDSA_SHA384 (if supported)
   - TEE_ALG_ECDSA_SHA512 (if supported)

```
TEE_Result TEE_AsymmetricSignDigest(
      TEE_OperationHandle    operation
[in]  TEE_Attribute*         params,
      uint32_t               paramCount,
[inbuf] void*                digest,
      size_t                 digestLen,
[outbuf]void*                signature,
      size_t*                signatureLen);
```

4. **TEE_FreeOperation**

The TEE_Free Operation function deallocates all resources associated with an operation handle. After this function is called, the operation handle is no longer valid. All cryptographic material in the operation is destroyed.

```
void TEE_FreeOperation (TEE_OperationHandle operation);
```

Object

1. **TEE_AllocateTransientObject**

The TEE_AllocateTransientObject function allocates an uninitialized transient object, i.e. a container for attributes. Transient objects are used to hold a cryptographic object (key or key-pair). The object type and the maximum key size MUST be specified so that all the container resources can be pre-allocated. As allocated, the container is uninitialized. It can be initialized by subsequently importing the object material, generating an object, deriving an object, or loading an object from the Trusted Storage.

```
TEE_Result TEE_AllocateTransientObject(
      uint32_t                objectType,
      uint32_t                maxObjectSize,
[out] TEE_ObjectHandle*object);
```

2. **TEE_GenerateKey**

The TEE_GenerateKey function generates a random key or a key-pair and populates a transient key object with the generated key material. The size of the desired key is passed in the keySize parameter and MUST be less than or equal to the maximum key size specified when the transient object was created.

```
TEE_Result TEE_GenerateKey(
      TEE_ObjectHandle        object,
      uint32_t                keySize,
[in]  TEE_Attribute*          params,
      uint32_t                paramCount);
```

3. **TEE_GetObjectBufferAttribute**

The TEE_GetObjectBufferAttribute function extracts one buffer attribute from an object. The attribute is identified by the argument attributeID

```
TEE_Result TEE_GetObjectBufferAttribute(
            TEE_ObjectHandle        object
            uint32_t                attributeID,
[outbuf]    void*                   buffer,
            size_t*                 size);
```

4. **TEE_FreeTransientObject**

The TEE_FreeTransientObject function deallocates a transient object previously allocated with TEE_AllocateTransientObject. After this function has been called, the object handle is no longer valid and all resources associated with the transient object MUST have been reclaimed

```
void TEE_FreeTransientObject(TEE_ObjectHandle object);
```

3.4 TEE Implementations

The implementations available refer to the processor used: Intel, AMD and ARM.

- Intel

  Intel's SGX TEE implementation for its processors.

- ARM

  TrustZone specifications are analysing the set of codes for ARM processors which are found in most mobile devices due to the improved power consumption.

- AMD

  Platform Security Processor (PSP)

Other implementations:

- OpenTEE (the one used at the present project as a testing environment)
  An open source implementation and research project from the University of Helsinki and sponsored by Intel. Provided under an Apache license.
- OP-TEE
  An open source implementation under BSD license, originally from STMicroelectronics, now owned and maintained by Linaro.

CHAPTER 4: TEE-FIDO IMPLEMENTATION

The implementation designed contains 3 parts:

- A local socket server which for the FIDO protocol represents the Relying Party
- The Client's Application
- The Client's Trusted Application within the TEE

Open-TEE project was used to run and debug the TEE part.
The scheme presented below describes all the steps of this implementation:

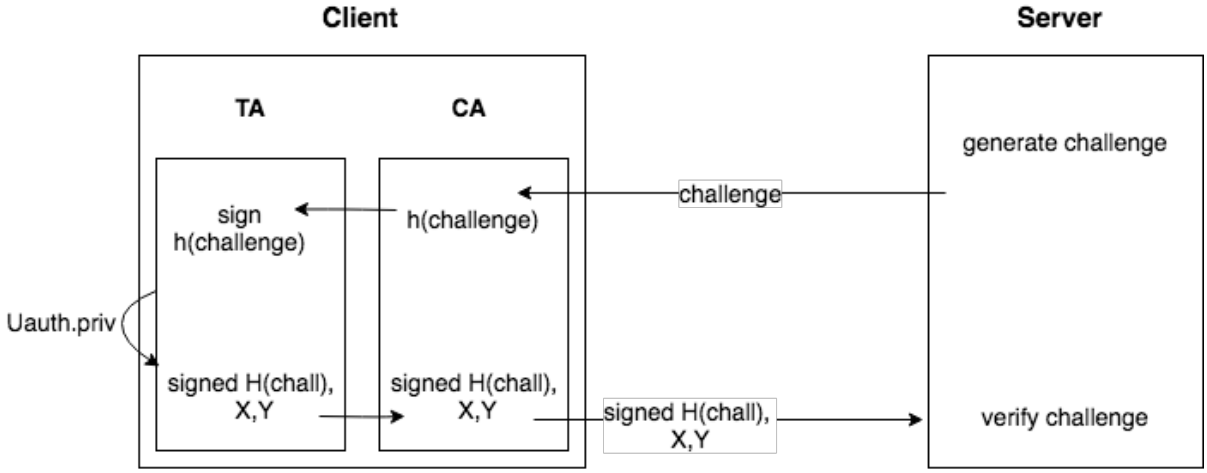

figure 16: FIDO in TEE implementation scheme

The server sends a challenge to the Client Application which will forward oti to the Trusted Application to be signed with ECDSA NIST P256 curve. The Trusted Application will send back the signed challenge and the public keys X, Y.
The results of the implementation are depicted in the images below:

```
anna@anna-VirtualBox:~/Open-TEE/gcc-debug$ ./ellipticcurve_ca
         buffer value here is: 12345sffdsdfsdf
         TA returned signed challenge
70       op params[1] signed challenge output here is: 304402201C87C9FBDA857E1EA
AA624F5075709AC97E219C1E70C0B1C41C91A8C73194109022078B3A19D847D58B3E83B755524C8D
CD4F686B6DE4868A6CF22FCB5D8F9648DBF
```

figure 17: CA implementation result

```
anna@anna-VirtualBox:~/Open-TEE/gcc-debug$ tail -f /var/log/syslog
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so: INFO  : convertNumToStr : 32
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so:  [OK] : ECDSA_sig : - with public key x size 32 value 15AEA0
AA43459CACBE6EFA0477FA2302BE2E2F1461D46D1BF9EA944D946AF036
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so: INFO  : convertNumToStr : 32
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so:  [OK] : ECDSA_sig : - with public key y size 32 value FEB44B
877017662194E33261033C7436741222116D824892BC0B798E7E1731F4
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so: INFO  : convertNumToStr : 32
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so:  [OK] : ECDSA_sig : - with private key size 32 value 2B4419B
579A7B9A33C73DC2104B99F881C1C10A8F0BD6E8C312C6D1B2F371141
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so:  [OK] : ECDSA_sig : ---digest is: 256 value 12345sffdsdfsdf
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so:  [OK] : ECDSA_sig : ---signature is: 71 value 0E#002 3y◆D%◆#
016s◆#010◆(◆ð◆◆◆◆Z}n#024◆K◆#025◆$(n◆#002!
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so:  [OK] : ECDSA_sig : --- WITH KEY SIZE 256
Jun 20 19:00:55 anna-VirtualBox libellipticcurve/home/anna/Open-TEE/gcc-debug/TAs/l
ibellipticcurve_ta.so: /home/anna/Open-TEE/TAs/ellipticcurve_ta/ellipticcurve_ta.c:
TA_CloseSessionEntryPoint:88  Closed Session: Returning attestation reply
^Z
```

figure 18: TA Implementation result

The code can be found in the repository: https://github.com/AnnaAnge/ECDSA_OpenTEE

CHAPTER 5: OTHER FIDO IMPLEMENTATIONS

- eBay
- ReCred

5.1 eBay

eBay's implementation in Java has defined every dictionary presented in the protocol specifications and the register, authentication and deregistration operation. It has 3 main files: the server, the core part of the protocol and the client which is an android app performing the operations from the user's device.
The guidelines in BuildingAndRunningUAFServerUsingMaven (CLIonly) explain thoroughly all steps needed to run the server.

5.2 ReCRED

The overall architecture of the project is descriped in the following scheme:

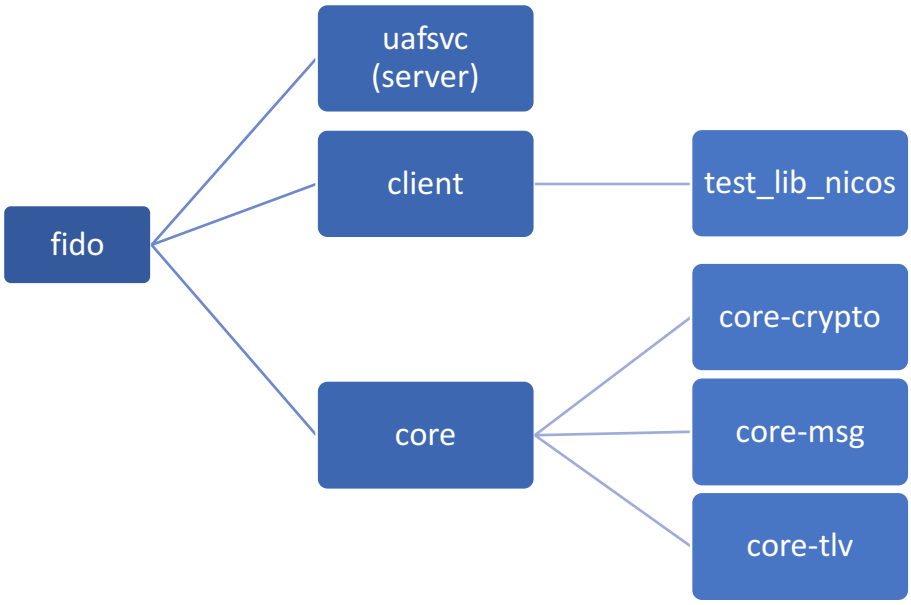

figure 19: ReCRED code overview

i. UAFsvc differs from ebay's UAF server. The code has is written in MVC pattern (model-view-controller).
ii. Client is a demo app.
iii. Core resembles ebay's project.

- In *core-crypto* every cryptographic operation is defined
- In *core-msg* all FIDO UAF dictionaries are defined according to the specifications.
- In *core-tlv* consist of more details of the protocol that were not previously defined in crypto or msg.

The project is build using Spring (Java) framework.
For the server mariadb is used.

Recred documentation

The ReCRED projects supports Registration, Authentication and Deregistration according to FIDO UAF v1.1 protocol specification. A great deal of this project is based on eBay's implementation.

REFERENCES

1. FIDO Specifications UAF v1.1: https://fidoalliance.org/specs/fido-uaf-v1.1-ps-20170202/FIDO-UAF-COMPLETE-v1.1-ps-20170202.pdf
2. FIDO Specifications UAF v1.2: https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/FIDO-UAF-COMPLETE-v1.2-rd-20171128.pdf
3. FIDO Specifications U2F v1.2: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/FIDO-U2F-COMPLETE-v1.2-ps-20170411.pdf
4. FIDO2 Specifications:
    a. https://fidoalliance.org/specs/fido-v2.0-rd-20170927/fido-overview-v2.0-rd-20170927.html
    b. https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-web-api-v2.0-ps-20150904.html
    c. https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html
    d. https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-signature-format-v2.0-ps-20150904.html
    e. https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-client-to-authenticator-protocol-v2.0-id-20180227.html
5. W3C Specifications: https://www.w3.org/TR/webauthn/
6. TEE Specifications: https://globalplatform.org/specs-library/?filter-committee=tee
7. Open TEE: https://ieeexplore.ieee.org/document/7345308/
8. Open TEE Project: https://open-tee.github.io
9. eBay UAF implementation: https://github.com/eBay/UAF
10. https://github.com/MicrosoftDocs/edge-developer/blob/master/microsoft-edge/dev-guide/device/web-authentication.md
11. Christoforos Panos, Stefanos Malliaros, Christoforos Ntantogian, Angeliki Panou, Christos Xenakis, “A Security Evaluation of FIDO’s UAF Protocol in Mobile and Embedded Devices”, International Tyrrhenian Workshop on Digital Communication (TIWC 2017), Palermo, Italy, September 2017
12. Christos Xenakis, Christoforos Ntantogian, Ioannis Stavrakakis, “A Network-Assisted Mobile VPN deployment for securing users data in UMTS", Computer Communications, Elsevier, vol. 31, No. 14, pp. 3315-3327 September 2008.
13. Eleni Darra, Christoforos Ntantogian, Christos Xenakis, Sokratis Katsikas, "A Mobility and Energy-aware Hierarchical Intrusion Detection System for Mobile ad hoc Networks," In Proc. 8th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2011), Toulouse France, August 2011
14. Christoforos Panos, Christoforos Ntantogian, Stefanos Malliaros, Christos Xenakis, "Quantifying, Analyzing and Evaluating Blackhole Attacks in Infrastructure-less Networks", Computer Networks, Elsevier, Vol. 113, February 2017, pp: 94-110