Digital Security: Cybersecurity, Privacy and Trust
DS-02-2014 Access Control

European Commission | Horizon 2020 European Union funding for Research & Innovation

From Real-world Identities to Privacy-preserving and Attribute-based CREDentials for Device-centric Access Control

WP3 – Beyond the Password: Device-centric Authentication

Deliverable D3.3 "Description of DCA protocols and technology support (revised)"

Editor(s):	UPRC: Christoforos Ntantogian, Eleni Veroni
Author(s):	CNIT: Claudio Pisa, Alberto Caponi, Tooska Dargahi CSGN: George Gugulea, Mihai Togan, Nicolae Ghibu, Bogdan Chifor TID: Claudio Soriente, Jeremy Blackburn, Nicolas Kourtellis, Diego Perino UC3M: Rubén Cuevas Rumín, Ivan Vidal UPCOM: Vangelis Bagiatis, George Savvas, Kostas Flokos, Lefteris Fanos UPRC: Christoforos Ntantogian, Eleni Veroni, Panagiotis Nikitopoulos, Christos Lyvas, Christos Xenakis VERI: Arsalan Najwani, Joao Gomes, Mark Fidell WEDIA: Evangelos Kotsifakos, Anty Rentetakos, Giorgos Gonidelis, Michalis Koulinas
Dissemination Level:	Public
Nature:	Report
Version:	1.8

ReCRED Project Profile

Contract Number	653417
Acronym	ReCRED
Title	From Real-world Identities to Privacy-preserving and Attribute-based CREDentials for Device-centric Access Control
Start Date	May 1st, 2015
Duration	36 Months

Partners

University of Piraeus	University of Piraeus research center	Greece
Telefónica Investigación y Desarrollo	Telefonica Investigacion Y Desarrollo Sa	Spain
verizon	Verizon Nederland B.V.	The Netherlands
certSIGN BY UTI	Certsign SA	Romania
wedia	Wedia Limited	Greece
EXUS	EXUS Software Ltd	U.K.
upcom Bringing business and IT together	Upcom Bvba (sme)	Belgium
THE PRODUCTIZERS	De Productizers B.V.	The Netherlands
2004	Cyprus University of Technology	Cyprus
Universidad Carlos III de Madrid	Universidad Carlos III de Madrid	Spain
cnit	Consorzio Nazionale Interuniversitario per le Telecomunicazioni	Italy
BAKER & MCKENZIE	Studio Professionale Associato a Baker & Mckenzie	Italy

Document History

Version	Date	Author	Remarks
0.1	13/09/2016		Initial Table of Contents
0.2	14/10/2016		Final Table of Contents
0.3	25/11/2016		Including WEDIA, UPCOM, CSGN, TID contributions
0.4	06/12/2016		Including VERI contribution
0.5	12/12/2016		Including UPCOM enhancements
0.6	14/12/2016		Including CSGN, UPRC contributions Document restructuring
0.7	19/12/2016		Including input from WEDIA, CNIT, CSGN, VERI
0.8 – 0.10	28/12/2016		Including input from CSGN
1	12/1/2017		First integrated version Document restructuring
1.1 – 1.4	23/1/2017		Including input from CNIT, WEDIA, TID, UPCOM, VERI and UPRC
1.5	25/1/2017		Review by CNIT, TID
1.6	27/1/2017		Corrections by UPRC, review by CUT
1.7	30/1/2017		Final version
1.8	31/7/2017		Revision - addressing PO's comments

Executive Summary

This document is part of “WP3 – Beyond the Password: Device-centric Authentication” of the ReCRED project. As the revised and extended version of the Deliverable D3.1 “Description of DCA protocols and technology support”, the purpose of this report is to further describe the Device Centric Authentication (DCA) protocols, user/device and device/server interfaces, presenting the progress made so far.

The document is organized as follows:

The Introduction gives an overview of the WP3’s scope, protocols, technologies and main components. Chapter 2 focuses on the description of the user-to-device authenticators and the FIDO UAF Client. The progress regarding the implementation of device-to-service authentication components can be found in Chapter 3, including the description of gateSAFE and FIDO UAF Server. In Chapter 4, the Behavioral Authentication Authority architecture is outlined, presenting the implemented modules and databases, as a part of the device-to-service authentication. Chapters 5 and 6 include the descriptions of needed extensions to FIDO standards in the context of federated authentication and privacy-preserving attribute-based authentication, respectively. Additionally, a great part of this document, and specifically Chapter 7, is dedicated to an in-depth analysis of Open-TEE, the virtual and open-source Trusted Execution Environment of choice for ReCRED. Chapter 8 presents the privacy and security considerations in the context of DCA and Chapter 9 concludes this deliverable.

Table of Contents

List of Figures

List of Tables

Table of Abbreviations

ABAC	Attribute-Based Access Control
API	Application Programming Interface
ASM	Authenticator Specific Module
BAA	Behavioral Authentication Authority
CA	Client Application
CDR	Call Description Records
DCA	Device-Centric Authentication
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
FIDO	Fast IDentity Online
FIDO UAF	FIDO Universal Authentication Framework
HAL	Hardware Abstraction Layer
HMAC	Hashed Message Authentication Code
HTTP	Hypertext Transfer Protocol
IDC	Identity Consolidator
JSON	JavaScript Object Notation
JWT	JSON Web Token
MDM	Mobile Device Management
MITM	Man in the Middle
PABAC	Privacy-preserving ABAC
PIN	Personal Identification Number
QR	Quick Response
REE	Rich Execution Environment
RSA	Rivest, Shamir, & Adleman (public key encryption technology)
SHA	Secure Hash Algorithm
SP	Service Provider
SSO	Single-Sign-On

TA	Trusted Application
TEE	Trusted Execution Environment
TLS	Transport Layer Security
UUID	Universally Unique Identifier

1 Introduction

ReCRED's ultimate goal is to promote the user's personal mobile device to the role of a unified authentication and authorization proxy. Towards this direction, in Work Package (WP) 3 the protocols and technologies for user-friendly device-centric authentication are being established. The developed Device Centric Authentication (DCA) protocol introduces two interfaces: one between the user and a local device and one between the local device and any service on the Internet. Core part of the DCA is the modification and extension of the FIDO UAF protocol to overcome the shortcomings of the FIDO Alliance standard while maintaining interoperability. In WP3, the developed standards enable, among others, the use of behavioral biometric authentication as a secondary confirmation and the support of trusted execution environments. The Deliverable D3.3, a revised and extended version of Deliverable D3.1, documents specifically and in detail the progress made so far on the authentication mechanisms supported by ReCRED, both on device and server side, describing also the Behavioral Authentication Authority (BAA) structure and modules, as well as how the Open-TEE, the open, virtual Trusted Execution Environment (TEE) can be enhanced and exploited for ReCRED's purposes.

The application-centric authentication model, where independent services apply individual authentication methods to verify the user's identity, has been proven to pose serious usability and therefore security problems, due to the fact that users tend to choose weak username/password combinations and then reuse them for more than one online services. But thanks to the ubiquity of the smartphones, the authentication scheme nowadays seems to be shifting towards the device-centric model, where the user authenticates to a local device and that device then authenticates to online services on user's behalf. The user-to-device authentication, executed by means of face recognition and Android TrustZone authenticators for ReCRED, is modified in such way to unlock a cryptographic key on the device, which is then used in FIDO protocol for device-to-service authentication.

FIDO has gained significant support over the last couple of years in its effort to address the lack of interoperability among strong authentication devices as well as the problems users face with creating and remembering multiple usernames and passwords. FIDO Alliance [21], the organization behind FIDO, aims to change the nature of authentication by developing specifications that define an open, scalable, interoperable set of mechanisms and set a new standard for devices to securely authenticate users beyond the password era.

However, as FIDO specifications also gain traction, more and more implementations appear. FIDO Alliance, in order to guarantee the FIDO compliance and interoperability among different products and services, launched the FIDO Certification program. For an implementation to be certified as FIDO compliant, a set of tests is required to be performed and passed in the official testing session that is being held every 3 months. The most recent testing session took place on 14th and 15th of December, 2016. Before being accepted in the Interoperability testing session, a qualification round must be also passed.

Starting with November 2016, the ReCRED consortium decided to participate in the Interoperability testing session of December 2016. ReCRED successfully participated with a server implementation and on December 28th, FIDO Alliance officially certified that ReCRED UAF Server complies with FIDO UAF specifications, making it one of the very few open source FIDO® Certified UAF Servers [22] [23], that

allows online service providers to offer device-centric password-less and multi-factor authentication mechanisms, as well as attribute-based access control.

Figure 1 ReCRED UAF Server FIDO® UAF Certification

Besides the officially admittance by FIDO Alliance that a FIDO UAF implementation is compliant with FIDO standards, the Interoperability testing session has another, more useful benefit too: it ensures that Identity Providers that use the ReCRED implementation of FIDO UAF that their instance is interoperable with common FIDO Authenticators/Clients/Servers on the market. Being open source software, the ReCRED FIDO UAF implementation gives the possibility to any Identity Provider to easily integrate or even extend it.

The ReCRED UAF Server FIDO® UAF Certification is a significant milestone and we appreciate that a participation to a future testing session is important to ensure the interoperability of ReCRED FIDO UAF client stack as well, with other implementations.

2 User to device authentication

2.1 Face Recognition

ReCRED face recognition module is an Android application that is used to verify the identity of a device owner. The solution is based on OpenCV (Open Source Computer Vision), which is a popular and well-established library, providing programming interfaces to C, C++, Python and Android. OpenCV

provides three different algorithms for face recognition: Eigenfaces, Fisherfaces and Local Binary Patterns Histograms (LBPH). Our face recognition module makes use of the LBPH algorithm. OpenCV also incorporates FLANN (Fast Library for Approximate Nearest Neighbours), an open-source library that contains a collection of algorithms optimized for fast nearest neighbour search in large datasets and for high dimensional features.

The face recognition module supports two operations: the user enrolment and the user verification. More details regarding the implementation of these operations are included in deliverable D3.2 (section 2.1 Physiological Biometrics). This section also includes an evaluation of the face recognition engine and its expected results (true/false positives/negatives) under different conditions.

During the reference period of this deliverable, our research was concentrated on how this face recognition engine could be used as an authenticator, for user-to-device authentication. In that direction, a user can enrol his facial features, using his device's camera, and use these features in order to be authenticated by an Identity Provider.

The face authenticator is a bound authenticator, in the sense that there is a logical binding between this authenticator and the device it is attached to, as opposed to roaming authenticators that are not bound to a specific device and can be used by any number of devices (e.g. hardware OTP tokens).

The purpose of the face authenticator is to be able to be used in conjunction with the FIDO client, so that each time a user attempts to authenticate to an application using FIDO, their facial features are verified against a number of templates, securely stored on the device.

ReCRED's face authenticator includes the following components:

- An **enrolment service**, which listens for enrolment requests by the FIDO UAF client. Each time such a request is received, a UI is displayed, which tries to detect a face using the device's front camera. After the user's face is detected, a number of photos are captured and n-dimensional feature vectors are extracted for each photo and securely stored as facial templates. If no face is detected during a predefined amount of time, an error message is shown and an error code is returned to the FIDO UAF client.
- An **authentication service**, which listens for authentication requests by the FIDO UAF client. Each time such a request is received, again a UI is displayed, which tries to detect the user's face using the device's front camera. As soon as a face is detected, a photo of the user is captured, and its feature vector is extracted and matched against the available facial templates. According to the score of the matching and a predefined threshold, the authenticator sends a response to the FIDO client, determining whether the captured face belongs to the legitimate owner of the device (according to the enrolled templates). If no face is detected during a predefined amount of time, an error message is shown and an error code is returned to the FIDO UAF client.
- A **retraining mobile app**, which allows the user to re-enrol his facial templates. In order to open the retraining app, the user must first pass a two-factor authentication challenge (face authentication, along with a PIN challenge). After that, the user can either add new facial templates or remove existing ones. This functionality is very useful to decrease the number of false positives (by removing templates) or false negatives (by adding templates).

2.2 Android TrustZone authenticators

2.2.1 PIN, Pattern, Password (Gatekeeper)

2.2.1.1 Android Authentication Architecture

The Android operating system implements a secure mechanism for allowing access to an application's private key by separating the process in two different entities with different roles: a secure storage for the cryptographic key and an authentication mechanism that will authenticate the user to the key's secure storage. The two components are implemented by the keymaster, keystore service, Gatekeeper and Fingerprint.

The keystore service runs in the OS space and its counterpart, the keymaster, runs in the TEE zone. Both of them have the role of protecting the private key material access and disallow unauthorized and unauthenticated access to the key.

The user access to the private key is realized through the specialized components that enforce the user identification. Depending on the identification type, this is implemented through: the gatekeeper component if the user is identified by a PIN or a password, or a drawn pattern and fingerprint component if the user is identified by its fingerprint. Any other identification method that will be introduced, must implement such a component.

Gatekeeper has the role to authenticate the user to the private key and is also comprised of two applications: one that runs in the TEE zone, named gatekeeper, and one that runs in the OS zone and is called gatekeeperd (the name follows the Linux pattern name for daemons). Only the OS components are exposed to the android applications through an API. This is a common practice in the TEE implementations, to expose the functionality to the user only through specialized components that run in the OS zone.

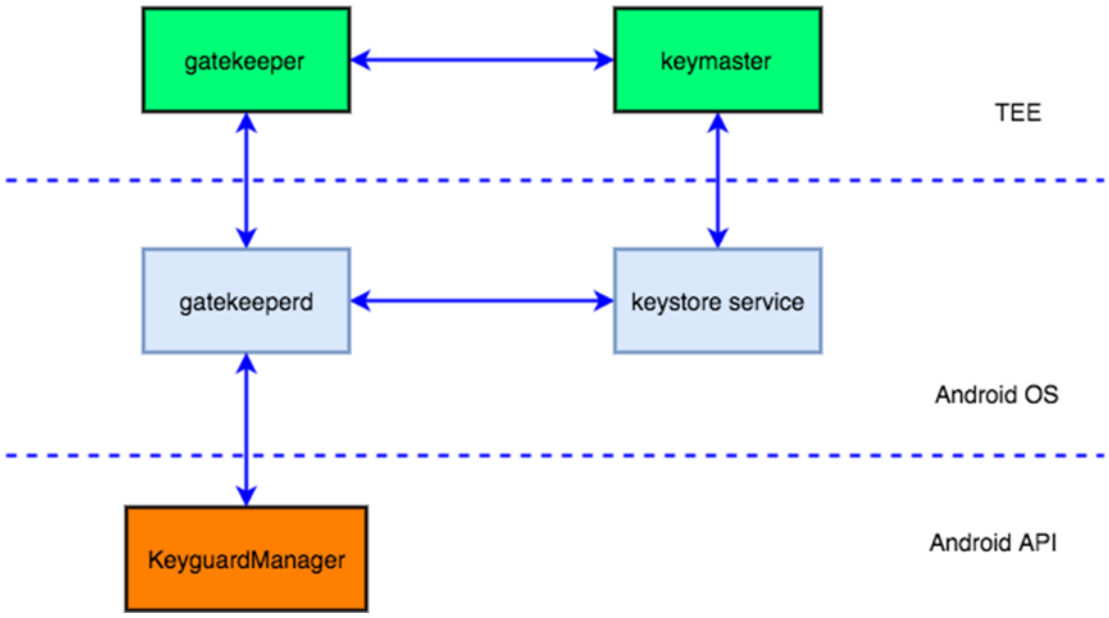

Figure 2 Android Authentication architecture

On top of the OS components (gatekeeperd), there is the API layer that bridges, through Java and Dalvik, the application access to the exposed functionality.

There is an Android common practice: not to expose the OS daemons functionality 1-1 through the Android API, but to introduce a more developer-friendly logic in between. As a result, the Android API contains a service named `KeyguardManager`. The `KeyguardManager` is available in the *android.app* package and can be instantiated by any application through the API call:

```
keyguardManager = (KeyguardManager) getSystemService(Context.KEYGUARD_SERVICE);
```

Keystore Service

The Keystore Service along with the Keymaster HAL provides a set of cryptographic primitives allowing for the implementation of protocols using access-controlled, hardware-backed keys.

KeyStore service is in android.security.KeyStore;

```
// Instantiate the keystore.
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");

// Import key.
char[] password = getPassword();
KeyStore.ProtectionParameter protParam = new KeyStore.ProtectionParameter(password);
SecretKey sk = getKey();
KeyStore.SecretKeyEntry skEntry = new KeyStore.SecretKeyEntry(sk);
KeyStore.setEntry("keyAlias", skEntry, protParam);

// Generate key.
KeyPairGenerator kpg = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC,
"AndroidKeyStore");

// Using KeystoreService to sign data.
KeyStore.Entry entry = keyStore.getEntry("keyAlias", null);
Signature s = Signature.getInstance("SHA256withECDSA");
s.initSign(((PrivateKeyEntry) entry).getPrivateKey());
```

LockSettingsService

The LockSettingsService keeps the lock pattern/password data and related settings for each Android user. The KeyguardManager uses the information stored in this service to interact with the GatekeeperService interface which allows the cryptographic material to be unlocked in the Keystore Service.

2.2.1.2 AuthTokens and User SID

AuthTokens and User SID help the entities running in separate spaces to establish user authentication through a secure channel. The User SID is randomly generated when the user first inputs his PIN, password or pattern, acts as an identifier and binds him to the cryptographic material generated or imported in the TEE. AuthTokens is a method of transporting this identifier between entities in the TEE and OS space. The AuthTokens are “signed” with a keyed SHA-256 MAC which allows the entities to recognize each other.

The following header file helps identifying the AuthToken fields. It is located in the android source code at the following folder: hardware/libhardware/include/hardware/hw_auth_token.h

```
typedef enum {
    HW_AUTH_NONE = 0,
    HW_AUTH_PASSWORD = 1 << 0,
    HW_AUTH_FINGERPRINT = 1 << 1,
    HW_AUTH_ANY = UINT32_MAX,
} hw_authenticator_type_t;

typedef struct __attribute__((__packed__)) {
    uint8_t version;  // Current version is 0
    uint64_t challenge;
    uint64_t user_id;             // secure user ID, not Android user ID
    uint64_t authenticator_id;    // secure authenticator ID
    uint32_t authenticator_type;  // hw_authenticator_type_t, in network order
    uint64_t timestamp;           // in network order
    uint8_t hmac[32];
} hw_auth_token_t;
```

The meaning of every field is as follows:

- **AuthToken Version:** meaning of the fields in the token following this one, as Android versions evolve, the fields value may be interpreted differently;
- **Challenge:** used only by the fingerprint implementation to prevent replay attacks;
- **User SID:** secure identifier described at the beginning of the chapter;
- **Authenticator ID (ASID):** All authenticators (gatekeeper or fingerprint) have their own value for the ASID that they can change according to their own requirements;
- **Authenticator Type:** 0x00 (PIN, password or pattern), 0x01 (fingerprint);
- **Timestamp:** most recent system boot (in milliseconds)
- **AuthToken HMAC:** Keyed SHA-256 MAC of all the fields except this one.

At every device boot, the TEE generates the SHA-256 MAC key and shares it through IPC with all its components (gatekeeper, keymaster, fingerprint, etc.). When the user inputs his password to the KeyguardManager, it's hash is sent to gatekeeperd which communicates with the TEE and if the password is recognized it returns the AuthToken containing the User SID binding to his cryptographic material along with the MAC of the whole AuthToken. After that, gatekeeperd sends the AuthToken to the Keystore Service which verifies the MAC with keymaster – located in the TEE. If the MAC is verified the Keystore can unlock cryptographic material associated with the User SID.

2.2.1.3 Gatekeeper Android OS Component (gatekeeperd)

Gatekeeper contains platform independent logic and corresponds to the **GateKeeperService** Java interface. It allows the **KeyguardManager** to unlock cryptographic material in the Keystore Service by fetching or saving a PIN, password, pattern, etc. from **LockSettingService** and storing/verifying it exists with the TEE component.

```
int enroll(uint32_t uid, const uint8_t *current_password_handle,
     uint32_t current_password_handle_length, const uint8_t *current_password,
     uint32_t current_password_length, const uint8_t *desired_password,
     uint32_t desired_password_length, uint8_t **enrolled_password_handle,
```

```
    uint32_t *enrolled_password_handle_length);

int verify(uint32_t uid, const uint8_t *enrolled_password_handle,
    uint32_t enrolled_password_handle_length, const uint8_t *provided_password,
    uint32_t provided_password_length, bool *request_reenroll);

uint64_t getSecureUserId(uint32_t uid);

void clearSecureUserId(uint32_t uid);
```

As can be seen from the above code, the main function of gatekeeperd is to enrol and verify a given password hash with the TEE component and supply the Keystore Service with the User SID, allowing it to unlock the given user's cryptographic material.

2.2.1.4 Gatekeeper TEE Component (gatekeeper)

The TEE gatekeeper component uses the shared secret to create an authentication attestation to send to the hardware-backed Keystore. Besides authentication attestation, it is responsible for throttling consecutive failed verification attempts.

The following header files needs to be implemented such that the gatekeeper can function along with the Android key authentication system.

The first, Gatekeeper HAL: hardware/libhardware/include/hardware/gatekeeper.h

```
int (*enroll)(const struct gatekeeper_device *dev, uint32_t uid,
       const uint8_t *current_password_handle,
       uint32_t current_password_handle_length,
       const uint8_t *current_password, uint32_t current_password_length,
       const uint8_t *desired_password, uint32_t desired_password_length,
       uint8_t **enrolled_password_handle,
       uint32_t *enrolled_password_handle_length);
```

Enrolls desired password, which should be derived from a user selected PIN or password, with the authentication factor private key used only for enrolling authentication factor data. If there was already a password enrolled, it should be provided in `current_password_handle`, along with the current password in `current_password` that should validate against `current_password_handle`

```
int (*verify)(const struct gatekeeper_device *dev, uint32_t uid,
       uint64_t challenge, const uint8_t *enrolled_password_handle,
       uint32_t enrolled_password_handle_length,
       const uint8_t *provided_password, uint32_t provided_password_length,
       uint8_t **auth_token, uint32_t *auth_token_length,
       bool *request_reenroll);
```

Verifies that `provided_password` matches `enrolled_password_handle`. Implementations of this module may retain the result of this call to attest the freshness of the authentication. On success, writes the address of a verification token to `auth_token`, usable to attest password verification to other trusted services. Clients may pass NULL for this value.

```
int (*delete_user)(const struct gatekeeper_device *dev, uint32_t uid);
```

Deletes the `enrolled_password_handle` associated with the `uid`. Once deleted the user cannot be verified anymore.

```
int (*delete_all_users)(const struct gatekeeper_device *dev);
```

Deletes all the `enrolled_password_handles` for all uid’s. Once called, no users will be enrolled on the device.

The key used to enroll and verify must never change, and should be re-derivable at every device boot.

After the Gatekeepr HAL, the following header file that needs an implementation is the Gatekeeper TEE: system/gatekeeper/include/gatekeeper/gatekeeper.h

```
void Enroll(const EnrollRequest &request, EnrollResponse *response);
void Verify(const VerifyRequest &request, VerifyResponse *response);

bool GetAuthTokenKey(const uint8_t **auth_token_key, uint32_t *length) const;
```

Retrieves the key used by GateKeeper::MintAuthToken to sign the payload of the `AuthToken`. This is not cached as it may have changed due to an event such as password change. Returns true if the key was successfully fetched.

```
void GetPasswordKey(const uint8_t **password_key, uint32_t *length);
```

The key used to sign and verify password data.

```
void ComputePasswordSignature(uint8_t *signature, uint32_t signature_length,
             const uint8_t *key, uint32_t key_length, const uint8_t *password,
             uint32_t password_length, salt_t salt) const;
```

Uses platform-specific routines to compute a signature on the provided password.

```
void GetRandom(void *random, uint32_t requested_size) const;
```

Retrieves a unique, cryptographically randomly generated buffer for use in password hashing, etc.

```
void ComputeSignature(uint8_t *signature, uint32_t signature_length,
                  const uint8_t *key, uint32_t key_length,
                  const uint8_t *message, const uint32_t length) const;
```

Uses platform-specific routines to compute a signature on the provided message.

```
uint64_t GetMillisecondsSinceBoot() const;
```

Get the time since boot in milliseconds.

```
void MintAuthToken(UniquePtr<uint8_t> *auth_token, uint32_t *length,
             int64_t timestamp, secure_id_t user_id,
             secure_id_t authenticator_id, uint64_t challenge);
```

Generates a signed attestation of an authentication event and assigns to `auth_token` `UniquePtr`. The format is consistent with that of `hw_auth_token_t`.

```
bool DoVerify(const password_handle_t *expected_handle,
       const SizedBuffer &password);
```

Verifies that handle matches password HMAC’ed with the `password_key`.

```
bool CreatePasswordHandle(SizedBuffer *password_handle, salt_t salt,
        secure_id_t secure_id, secure_id_t authenticator_id,
        const uint8_t *password,
```

```
        uint32_t password_length);
```

Populates `password_handle` with the data provided and computes HMAC.

For the TEE Gatekeeper, the primary responsibilities of a compliant implementation are:

- Adherence to the Gatekeeper HAL;
- Returned AuthTokens must be formatted according to the AuthToken specification;
- The TEE Gatekeeper must be able to share an HMAC key with Keymaster, either by requesting the key through a TEE IPC on demand or maintaining a valid cache of the value at all times.

2.2.2 Fingerprint Hardware Abstraction Layer (HAL)

If a device has a fingerprint sensor, a user can enroll one or more fingerprints and then use their fingerprints to unlock the device and use the ReCRED application. For Android based mobile devices, the Fingerprint Hardware Abstraction Layer (HAL) is used to connect to a vendor-specific library and fingerprint sensor. The implementation of the Fingerprint HAL is located in the fingerprint.h header file.

2.2.2.1 Flow

The following is a high-level flow for fingerprint matching. This flow assumes that a fingerprint already has been enrolled on the device, i.e. the vendor-specific library already has enrolled a template for the fingerprint. In case the user is not enrolled, first he/she should enroll its fingerprint by securely storing them inside the TEE. In response to a call to the authenticate function, the fingerprint sensor listens and performs the following actions:

1. When the user places a finger on the fingerprint sensor, the vendor-specific library determines if there is a match based on the current set of enrolled templates. The verification happens inside TEE as described below so that enrolled fingerprints never move outside the secure storage of TEE.
2. The result of step 1 is passed to the Fingerprint HAL, which notifies fingerprintd (the Fingerprint daemon) of a fingerprint authentication. The security of this communication channel is guaranteed using the AuthTokens that were described in section 2.2.1.

2.2.2.2 Architecture

The Fingerprint HAL interacts with the following components (Figure 3):

- **FingerprintManager API**: Interacts directly with an app in an app process.
- **FingerprintService:** A service that operates in the system process, which handles communication with fingerprintd.
- **fingerprintd** (Fingerprint daemon): A C/C++ implementation of the binder interface from FingerprintService. The fingerprintd daemon operates in its own process and wraps the Fingerprint HAL vendor-specific library.

- **Fingerprint HAL vendor-specific library:** A hardware vendor's implementation of the Fingerprint HAL. The vendor-specific library communicates with the device-specific hardware which is a TEE that will verify the correctness of the provided fingerprint template.
- **Keystore API** and **Keymaster**: These components provide hardware-backed cryptography for secure key storage in a Trusted Execution Environment (TEE). This communication is similar to the communication between the Gatekeeper and its part which is executed inside TEE. That is if the user is authenticated using his/her fingerprints, then the keys which are specific to the user and the application can be released by the keystore.

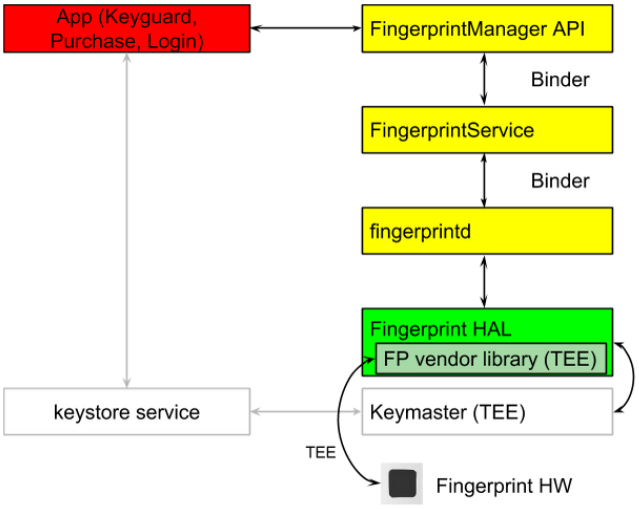

Figure 3 Fingerprint architecture in Android (taken from [1])

2.2.2.3 Major functions in the Fingerprint HAL

Below are some major functions in the /hardware/libhardware/include/hardware/fingerprint.h file.

- `enroll`: Switches the HAL state machine to start the collection and storage of a fingerprint template. As soon as enrollment is complete, or after a timeout, the HAL state machine is returned to the idle state.
- `get_authenticator_id`: Returns a token associated with the current fingerprint set.
- `enumerate`: Synchronous call for enumerating all known fingerprint templates.
- `remove`: Deletes a fingerprint template.
- `authenticate`: Authenticates a fingerprint-related operation.

2.3 FIDO UAF Client

ReCRED user to device first factor authentication is handled by the FIDO UAF (Universal Authentication Framework) module. On the user device, it is integrated as an Android Library and on other entities as a Spring Web Model-View-Controller (MVC) servlet application.

An Android Library allows packaging various Android application resources which a JAR can’t contain, such as layout files, fragments, activities, services, manifest files and much more, all compiled and bundled into an AAR archive which can easily be used and extended in various circumstances.

On the server side, the Spring Web MVC framework allows for loose coupling through dependency injection, easy endpoint definition, persistence schemes and other important development aspects such as unit testing among other things.

Using the Android Library on the device and the MVC framework on the server side allows for a modular structure of the projects, thus some modules can be reused on both the device and the server side.

The most important modules that are shared by the client and the server implementation are the cryptographic and TLV “Type-Length-Value” processing modules which allow for the generation and transport of cryptographic material between the authenticator and the server. The cryptographic module is based on the *Bouncy Castle* API, but because Android is shipped with an older version of Bouncy Castle as standard, this API had to be replaced with *Spongy Castle* which has the same API with the packages renamed from org.bouncycastle.* to org.spongycastle.* and Java Security API provider’s name changed from BC to SC.

Another important module that both implementations share is the Protocol Messages module. This module packs the TLV encoded cryptographic data for the different types of procedures defined in the FIDO UAF protocol such as registration, authentication and deregistration.

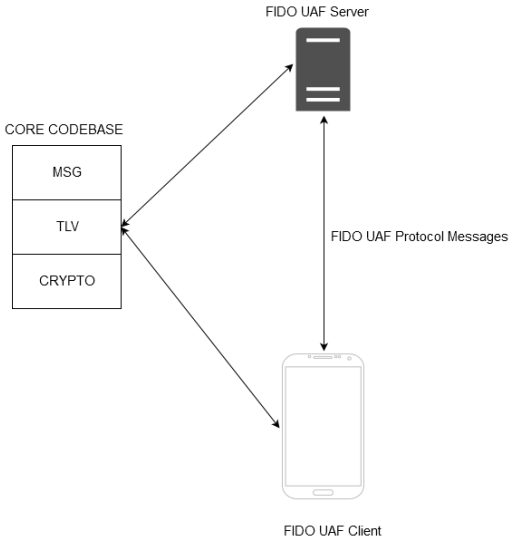

Figure 4 FIDO UAF modules

Irrespective of the protocol operation (registration, authentication and deregistration) the client stack has the same entry point in the FIDO UAF Android Library module. The module is called through an Intent which specifies the type of operation to be executed by the authenticator. After the Intent is sent, the client waits for the module response in the `onAtivityResult()` callback.

```
Intent i = new Intent("org.fidoalliance.intent.FIDO_OPERATION");
i.addCategory("android.intent.category.DEFAULT");
i.setType("application/fido.uaf_client+json");
```

```
Bundle data = new Bundle();
data.putString("message", regRequest);
data.putString("UAFIntentType", UAFIntentType.UAF_OPERATION.name());
i.putExtras(data);
……
startActivityForResult(i, REG_ACTIVITY_RES_3);
```

After receiving the Intent, the module first calls the `KeyguardManager` to receive the user consent for the operation – using the method specified at enrollment time, such as PIN, password, pattern, fingerprint, etc.

```
Intent intent = keyguardManager.createConfirmDeviceCredentialIntent("UAF","Confirm
Identity");
if (intent != null) {
   startActivityForResult(intent, REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS);
} else {
   finishWithError();
}
```

On receiving user confirmation for the operation, the module starts processing the message. Depending on the operation an ASM request is created:

```
ASMRequest<OperationIn> asmRequest = new ASMRequest<>();
asmRequest.requestType = Request.Operation;
asmRequest.asmVersion = request.header.upv;
asmRequest.args = operationIn;
asmRequest.authenticatorIndex = 1;
```

The ASM Request instructs the authenticator to perform the requested operation and return the TLV encoded cryptographic material in an `OperationOut` structure which the client appends to the protocol message and sends to the server.

During attestation, the authenticator creates keys, signs the created keys with the attestation certificates, or signs challenges received from the server with the created keys. Various signing algorithms and key encodings are used in the protocol, the following code shows the key generation for the ECDSA signing algorithm with the ‘secp256r1” curve, signing the generated key with the attestation root certificate private key and authentication challenge signing with the generated key.

Key generation:

```
public static KeyPair getKeyPair()
      throws InvalidAlgorithmParameterException,
      NoSuchAlgorithmException, NoSuchProviderException {

   ECGenParameterSpec ecGenSpec = new ECGenParameterSpec("secp256r1");
   KeyPairGenerator g = KeyPairGenerator.getInstance("ECDSA", "SC");
   g.initialize(ecGenSpec, new SecureRandom());
   return g.generateKeyPair();
}
```

Signing of generated keys:

```
private byte[] getSignature(byte[] dataForSigning) throws Exception {
   PrivateKey priv =
                KeyCodec.getPrivKey(Base64.decode(Authenticator.priv,
Base64.URL_SAFE));
    BigInteger[] signatureGen = NamedCurve.signAndFromatToRS(priv,
    SHA.sha(dataForSigning, "SHA-256"));
```

```
    boolean verify = NamedCurve.verify(
KeyCodec.getKeyAsRawBytes((ECPublicKey)KeyCodec.getPubKey(Base64.decode(Authenticat
or.pubCert, Base64.URL_SAFE))), SHA.sha(dataForSigning, "SHA-256"),
Asn1.decodeToBigIntegerArray(Asn1.getEncoded(signatureGen)));
    if (!verify) {
        throw new RuntimeException("Signatire match fail");
    }
    byte[] ret = Asn1.toRawSignatureBytes(signatureGen);

    return ret;
}
```

Authentication challenge signing:

```
private byte[] getSignature(byte[] dataForSigning) throws Exception {
    PublicKey pub =

KeyCodec.getPubKey(Base64.decode(Preferences.getSettingsParam("pub"),
Base64.URL_SAFE));
    PrivateKey priv =
KeyCodec.getPrivKey(Base64.decode(Preferences.getSettingsParam("priv"),
Base64.URL_SAFE));

    BigInteger[] signatureGen = NamedCurve.signAndFromatToRS(priv,
                SHA.sha(dataForSigning, "SHA-256"));

    boolean verify = NamedCurve.verify(KeyCodec.getKeyAsRawBytes((ECPublicKey)pub),
                SHA.sha(dataForSigning, "SHA-256"),
                Asn1.decodeToBigIntegerArray(Asn1.getEncoded(signatureGen)));
    if (!verify) {
        throw new RuntimeException("Signatire match fail");
    }
    byte[] ret = Asn1.toRawSignatureBytes(signatureGen);

    return ret;
}
```

3 Device to service authentication

3.1 Behavioral authentication factors

3.1.1 Gait-based authentication

One of the behavioral authentication factor that ReCRED uses is gait. In order to design algorithms to authenticate an individual based on gait dynamics, we have developed an Android application that periodically collects data from sensors (e.g., accelerometers, gyroscope, etc.) available on the platforms and uploads it to our server. The application was made available on the Google Play store and advertised through dedicated mailing lists, as well as among the companies and institutions of the ReCRED consortium. Our data collection campaign spanned across multiple weeks and involved more than 100 users for a total of more than 60,000 data samples. The collected data was used to train and tune a machine learning algorithm that reached an accuracy of 80%. Details on the app, the collected data, and the classifier are available in Deliverable 3.2.

3.1.2 Mobility-based authentication

We have analyzed and extracted location features from Call Description Records (CDRs) in order to use them for authentication purposes. Location features included incoming and outgoing calls, call time/date, cell towers connected to the smartphone at the time of the call. Features were used to

build user profiles and to rank an excerpt of a user (CDR) history against the profiles of all users being monitored. We used more than 220 million CDRs spanning the activity of 1.6 million users over 3 months. Our classification algorithm reached a true positive rate higher than 80% with a negligible false positive rate. Details on the data and the algorithms are available in Deliverable 3.2.

3.1.3 Browsing-based authentication

Another authentication means used in ReCRED focuses on the browsing behavior of a user. As described in Deliverable D3.2, we have collected browsing history from more than 400 users for up to 1 year across mobile and desktop devices. We have shown that the browsing signature (i.e., the pages visited by an individual and the times when those visits happened) are fairly stable over time and can be used as an additional factor in a multi-factor authentication infrastructure. Collected data was used to design a ranking algorithm that compares the browsing history excerpt of a user against the signature history of all users being monitored.

The technique detailed in Deliverable D3.2 focuses on HTTP traffic, since HTTPS conceals the actual page visited to a proxy that sits on the communication link between the browser and the web server. More accurate profiling could be achieved if one could infer the page visited by a user despite HTTPS. Given that in a TLS handshake the domain is visible thanks to the Server Name extension, the problem of inferring the page visited by a user, reduces to the problem of inferring the page visited by a user within a given domain. In the following, we show how profiling accuracy may be increased by guessing the exact page a user is browsing using traffic fingerprinting. Traffic fingerprinting is an active research area on techniques to infer information (such as the visited page on an encrypted connection) by solely observing traffic patterns at the network/transport level. Fingerprinting involves a training phase during which the profiler builds a fingerprint of each of the monitored pages. This is accomplished by fetching multiple times the monitored pages and recording features of the generated traffic such as packet size or inter-arrival times. Later, the profiler eavesdrops on the client's connection, extracts the same features from the client's traffic, and tries to match the client trace to one of the fingerprints computed during the training phase. Differences between the training data and the client (or test) data, due to, e.g., different routes or congestions are mitigated using statistical methods.

We use and adapt to our scenario the fingerprinting technique of Panchenko et al., that appeared at NDSS 2016, the most accurate web fingerprinting framework to date, that uses as features the size and the direction of each packet of a TCP connection. The classifier is, therefore, robust against differences in bandwidth or congestions along the route.

We show that webpage fingerprinting can be reasonably accurate in a “closed-world” scenario in which the profiler monitors all the pages that the client can possibly visit. This assumption is realistic in our settings because the profiler knows the website requested by the user (by looking at the Server Name in the client_hello message) and must infer which page the user is browsing within this particular website.

The features we extract from the traffic generated by downloading a page include: the number of incoming packets and the number of outgoing ones, the total size of incoming packets and the total size of outgoing ones, and a trace defined over the size and the order of the observed packets. We use an SVM classifier with an RBF kernel parametrized with γ in [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000] and c in [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000].

For each monitored website, we captured with tcpdump the traffic generated by fetching each of the 1-st level pages 50 times and measured the accuracy of the classifier using 10-fold cross validation.

3.1.3.1 *Classifier Accuracy*

For this experiment, we pick 9 popular websites, each ranking high in the respective category of alexa.com. The chosen websites are: amazon.com, rakuten.com, aarp.com, wonderhowto.com, about.com, mashable.com, slashdot.org, nbcnews.com, and reuters.com. For each website, we trained the classifier and tested its accuracy in three different scenarios. We used a PC with Mozilla Firefox with and without cache, and a mobile device with Google Chrome with cache enabled. In the latter scenario, we used the Android emulator to fetch the pages from an emulated Nexus 5 using the built-in feature of the emulator to simulate the conditions of a 3G network.

Figure 5 shows the accuracy of the classifier for the 9 websites in each one of the aforementioned scenarios.

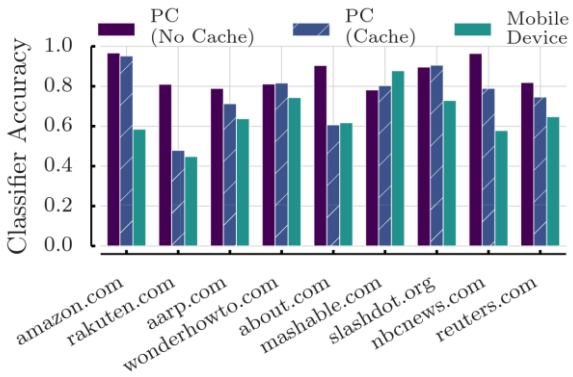

Figure 5 The accuracy of the classifier for the 9 websites

We found the lower accuracy for the PC with cache scenario when predicting pages of aarp.com (0.79) while we experienced the highest accuracy for amazon.com (0.97). Caching inevitably hinders the accuracy of the classifier by 10.3% on average, but the average accuracy never drops below 0.48. The accuracy decreases because when parts of a page are in the local cache, the traffic trace available to the classifier becomes shorter and, therefore, more likely to be confused with that of another page. In the extreme case of a page whose elements are all in the cache, the resulting trace becomes totally indistinguishable from that of any other fully cached page. The mobile phone scenario suffers from a similar issue, not due to caching only, but also because mobile versions of a site are typically simpler than their desktop counterparts, and thus they end up producing more similar traffic traces.

3.1.3.2 *Classifier Freshness*

The difference between the time when the classifier is trained and the time when pages are predicted may affect the prediction accuracy. This is especially true for very dynamic websites (e.g., news or online community websites). In this experiment, we discretize time in "epochs" and we assume that website content only changes from one epoch to the next one. If the train and the test data are collected in the same epoch, we say that the classifier is "fresh" otherwise we say that the classifier is

"stale". We define epochs as days. We train the classifier over a snapshot of the website on a given day, and we try to predict pages fetched throughout the following 6 days.

We expect a sensible difference in accuracy between a stale classifier and a fresh one for dynamic pages where content changes every day (e.g., news websites). In the case of websites with static content, the difference between a stale classifier and a fresh one are expected to be less pronounced. To verify this, we add 4 websites with mostly static content (2 corporate and 2 academic ones) to the 9 websites of the previous experiments.

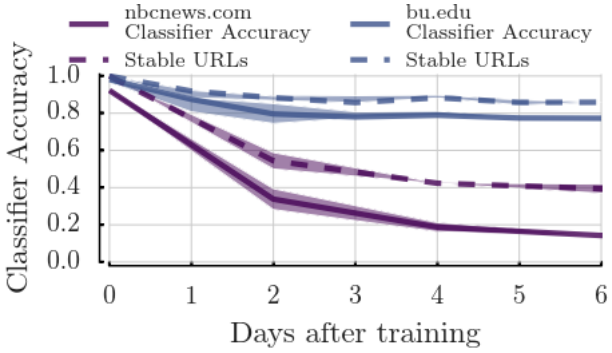

Figure 6 The effect of staleness on the accuracy of the classifier

Figure 6 shows the effect of staleness on the accuracy of the classifier for both a stateful and a dynamic website. The dashed lines represent the percentage of 1st-level pages that remain linked in the main page across days, while the solid ones represent the accuracy of the classifier. We observe the accuracy of the classifier for the dynamic website decreases rapidly while the accuracy for the static one decreases slowly during the first two days and then stabilizes around 80% accuracy. For both lines, the shadows denote the minimum and the maximum of the statistics.

3.1.3.3 Summary

User profiling through their browsing behavior requires the profiler to log visited webpages of the users being profiled. HTTP traffic allows the profiler to log visited webpages without ambiguity. HTTPS only lets the profiler learn the domain visited by a user but not the actual webpage within that domain. Profiling accuracy can benefit from techniques that infer the webpage visited by a user within a given domain, despite HTTPS. We have shown how to use traffic fingerprinting techniques to infer webpages visited by a user within an encrypted connection. Our results are promising and the technique may increase profiling accuracy for HTTPS traffic.

3.1.4 Keystroke-based authentication

An introduction on the keystroke-based authentication, the background and the related work has been provided in Deliverable D3.2. In the current deliverable, a description of the software components and the overall flow and process is made, starting with the client side, then describing the application on the user's device and following, in section 4.4, with the description for the server side on the Behavioral Authentication Authorities.

3.1.4.1 Mobile device (client) side

In this section, we discuss the way that the *b-verifier* keyboard application works on the user mobile device, capturing the necessary information and sending it to the Behavioral Authentication Authority.

The ReCRED mobile application will incorporate a custom keyboard application, the ReCRED enabled b-verifier keyboard application (BVK). The specifications of this application are described in section 3.1.4.2 "B-Verifier Mobile application technical specification".

The BVK (B-Verifier Keyboard) operation is described below.

It captures the timing between consecutive keystrokes (key-down and key-up timestamps of every key press) of the user and calculates the following measures:

1. Time between key-press (key down) and key-release (key up) of a single character.
2. **KU-KD**: Key Up-Key Down time is the time between key-up of a character and key-down of the next character.
3. **KD-KD**: Key Down- Key Down time is the time between key-down of a character and the key-down of the next character.
4. **KU-KU**: Key Up- Key Up time is the time between key-up of a character and the key-up of the next character.
5. **KD-KU**: Key Down- Key Up time is the time between key-down of a character and key-up of the next character.

We call a "sequence" every set of feature vectors (keystrokes) that is recorded in a period of time with possible pauses that are less than 3 seconds (this is configurable in the application code). Three seconds of inactivity (no keystroke) marks a new sequence.

We call a "session" the time between the appearance of the BVK and its disappearance.

The device stores in RAM the sequences and sends them to the server when the keyboard disappears (pressing the hide button, or the back button or any other button that exits the keyboard or the application) i.e. when the session is over.

A sessionID is also sent to the server.

Along with the keystroke timings the following data are captured:

- **Device ID**: Unique key – device id.
- **App ID**: The application from which the data are recorded.
- **Device scale factor** [2]:
  Supported values:
  - xlarge screens are at least 960dp x 720dp
  - large screens are at least 640dp x 480dp
  - normal screens are at least 470dp x 320dp
  - small screens are at least 426dp x 320dp
- **Device density class** [3]:
  Supported values:
  - ldpi (low) ~120dpi
  - mdpi (medium) ~160dpi

- hdpi (high) ~240dpi
- xhdpi (extra-high) ~320dpi
- xxhdpi (extra-extra-high) ~480dpi
- xxxhdpi (extra-extra-extra-high) ~640dpi

- **Text input type** [4]: The type of the input field.

The goal is to provide enough sessions to the server (Behavioral Authentication Authority-BAA) to build a typing signature/profile of the user for each application and for the device as whole.

Summarizing, the BVK app process:

1. captures keystroke timings
2. sends them to the server
3. the server stores them to a database and training or verification is performed

3.1.4.2 B-Verifier Mobile application technical specification

For the client application of the mobile device, the open source keyboard *AnySoftKeyboard* [5] is used as a basis. In order to capture the key interactions, the *date4j* [6] library is used. The *okHttp* [7] library is used for securely transmitting the data to the server.

The following features have been disabled in order to have "clean" keystroke capturing: autocomplete, suggestions, gestures.

System parameters

- The application accepts the following parameter: Keystroke pause interval: The time interval in seconds during which if no keystrokes occur the application stops counting the time between the keystrokes and considers the key sequence finished.

3.1.4.3 Testing and performance

Currently the application and the server side module have been deployed in a test environment and on two mobile devices and the collection and training process is ongoing.

Initial testing shows that the application is performing well for one user, but we are trying to engage a group of real users to test it in real life conditions. Results will be provided on the following deliverable D3.4 along with any other improvements or changes in the b-verifier application.

3.1.4.4 Future work

Apart from the extensive testing that is to be performed with real users, more sophisticated features can be also used if the ones mentioned above do not perform well enough. Such features could be:

- The rotation of the device while user is typing.
- The orientation of the device.
- Number / percentage of special characters used, mainly the backspace and punctuation keys.

3.2 QR login

QR Login permits the identity transfer between the authenticated smart-phone and a desktop. The identity transfer process is realized by using the device's camera to scan a QR code and by performing a security protocol with a server entity (QR Authenticator). The QR Login architecture was described in Deliverable D3.1 and Deliverable D3.2.

The QR Login comprises three main modules:

- client (deployed as an Android application)
- QR Authenticator server
- Service Provider

The identity transfer process has a federated authentication structure where the user is given access to the Service Provider resources after smart-phone application authenticates to the QR Authenticator module.

The sequential diagram for the QR authentication is presented in Figure 7.

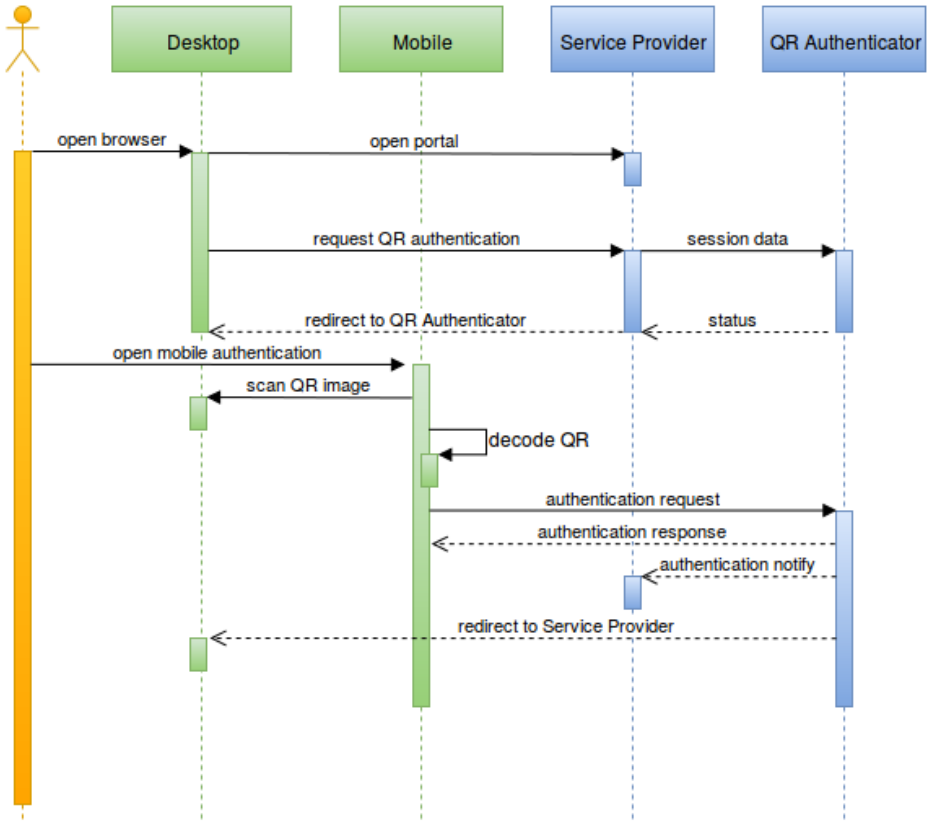

Figure 7 QR Authentication sequential diagram

The QR authentication process can be summarized in the following steps:

1. The user opens the portal using the web browser from its computer and starts a QR Authentication.

2. The Service Provider generates a token associated with the current user session and sends a QR authentication request to the QR Authenticator module. The Service Provider request must contain a policy which describes the security parameters which will be used in the user authentication.
3. The QR Authenticator verifies that the QR authentication request is from a trusted entity and generates an authentication message encoded in a QR code in order to be scanned by the mobile application.
4. The user starts the mobile application and scans the QR code image displayed in the browser.
5. The application decodes the QR code image and starts an authentication process with the QR Authenticator. The authentication steps presented in the sequence diagram are generic. The application could authenticate to the QR server by means of FIDO, mutual TLS or other mechanisms.
6. The mobile application sends the authentication message to the QR Authenticator.
7. The QR Authenticator verifies the correctness of the received QR code and expiration time and notifies the Service Provider.
8. The QR Authenticator redirects the user to the Service Provider.

The following modules comprise the QR Login software stack on the Android client side:

- Camera activity
- QR Decoder
- QR Crypto
- QR Transport

The *Camera activity* module is in charge of scanning the QR code, while displaying the camera view.

The *Camera activity* module is defined using the following declaration:

```
public class CameraActivity extends Activity implements
                SurfaceHolder.Callback, SensorEventListener, OnTouchListener
```

When the activity surface is created, the camera module is configured through the following code snippet. A callback is configured to process every image frame, in order to decode the QR code.

```
PreviewCallback previewCb = new PreviewCallback() {
      public void onPreviewFrame(byte[] data, Camera camera) {
            Camera.Parameters parameters = camera.getParameters();
            Size size = parameters.getPreviewSize();
            Image barcode = new Image(size.width, size.height, "Y800");
            barcode.setData(data);
            int result = scanner.scanImage(barcode);
            if (result != 0) {
                  try {
                        SymbolSet syms = scanner.getResults();
                        for (Symbol sym : syms) {
                              String qrCode = sym.getData();
                              /*Process QR Code*/
                        }
                  } catch (Exception e) {
                        Log.e(“QRLogin”, “Exception:” + e.getMessage);
                        /*Handle error*/
                  }
            }
```

```
        }
};
```

The QR decoder module was developed using the open-source Zbar Android library. The QR contains a JSON encoded (represented as Base64) message with the following structure:

```
{
        version: Integer,
        policy: String,
        qrCode: String,
        qrAuthURL: String,
        extension: String
}
```

The `version` field indicates the protocol version, the `policy` field indicates if a specific cryptographic operation (like a RSA or ECC signature should be applied to the QR code), the `qrCode` field contains the random code generated by the QR Authenticator module and the `qrAuthURL` contains the QR Authenticator server address. The `extension` may contain any application specific data.

After the QR code is obtained, a cryptographic operation can be applied to this value and transmitted along with the original value to a transport channel. If the transport channel between the mobile application and the QR Authenticator server is secured and authenticated (E.g.: a TLS channel with FIDO authentication) the QR code can be sent without applying any extra cryptographic operation. The following diagram describes the QR code along with the cryptographic and transport operations.

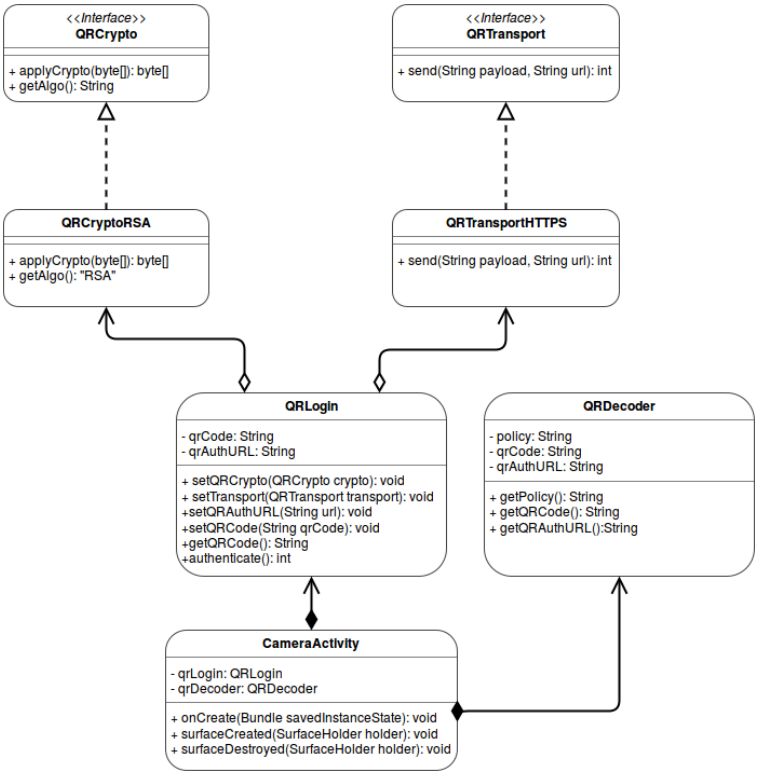

Figure 8: General architecture of the QR login

Figure 8 describes the QR authentication software modules along with the cryptographic and transport operations. The QR crypto module is in charge of applying cryptographic operations to the QR code, while the QR transport is in charge of sending the authentication message to the QR Authenticator. After the code is processed by the QR login module, the data is encapsulated in a JWT which is transmitted to the QR transport module. If no cryptographic operation is applied to the code, the JWT will contain the null algorithm in the header.

After processing the QR code, the mobile application sends the following JSON encoded message to the QR Authenticator:

```
{
	version: Integer,
	qrCode: String,
	cryptoCode: String,
	extension: String
}
```

The authentication protocol between the mobile application and the QR Authenticator may have a variable number of steps depending on the employed cryptographic parameters, being suitable not only for challenge-response protocols.

3.2.1 Service Provider

In order to provide a QR login functionality, the Service Provider module redirects the user to the QR Authenticator, after sending information regarding the session of the user to be authenticated. The Service Provider and the QR Authenticator share a secret key used to authenticate the exchanged messages by computing a HMAC encapsulated in a JWT. On the Service Provider side, after the user requests a QR login, the following message is sent to the QR Authenticator server:

```
{
	version: Integer,
	policy: String,
	userSessionID: String,
	callbackURL: String,
	extension: String
}
```

The `version` field contains the protocol version, the `policy` field indicates the security methods applied by the QR Authenticator for the authentication process, `userSessionID` contains the user-session identifier (should be a random data which uniquely identifies the authentication session) and the `callbackURL` contains the URL called by the QR Authenticator after the authentication process is finished. The `extension` field may contain any application specific data.

3.2.2 QR Authenticator

After the user is redirected to the QR Authenticator, a QR encoded authentication message is displayed in order to be scanned by the mobile application. The authentication QR code is displayed for a limited amount of time. After the authentication process is finished, the QR Authenticator notifies the Service Provider about the authentication status by sending the following JSON encoded message:

```
{
	version: Integer,
	userSessionID: String,
	status: String,
	extension: String
}
```

The `version` field indicates the protocol version, the `userSession` contains the user-session related data sent initially by the Service Provider and the `status` field contains the authentication status. The `extension` field may contain any application specific data. The message is packed in a JWT and secured with a HMAC, by using the secret key shared by the Service Provider and QR Authenticator. If the authentication is successful the user is redirected to the Service Provider, where the protected resource can be accessed.

3.3 Secure communication channel – gateSAFE

3.3.1 gateSAFE Architecture

gateSAFE enables secure access, accounting and control to both modern and legacy web applications by leveraging state of the art technologies like Transport Layer Security (TLS) and Digital Certificates.

Transport Layer Security is a suite of cryptographic protocols and signalling that provide communications security across a network in a way designed to prevent eavesdropping and tampering. As the central point of access for web applications, gateSAFE acts as a gateway, securing the communication with the client and accessing the requested resource on client's behalf.

Figure 9 depicts the general architecture of gateSAFE. The gateSAFE modules are described in the following sections.

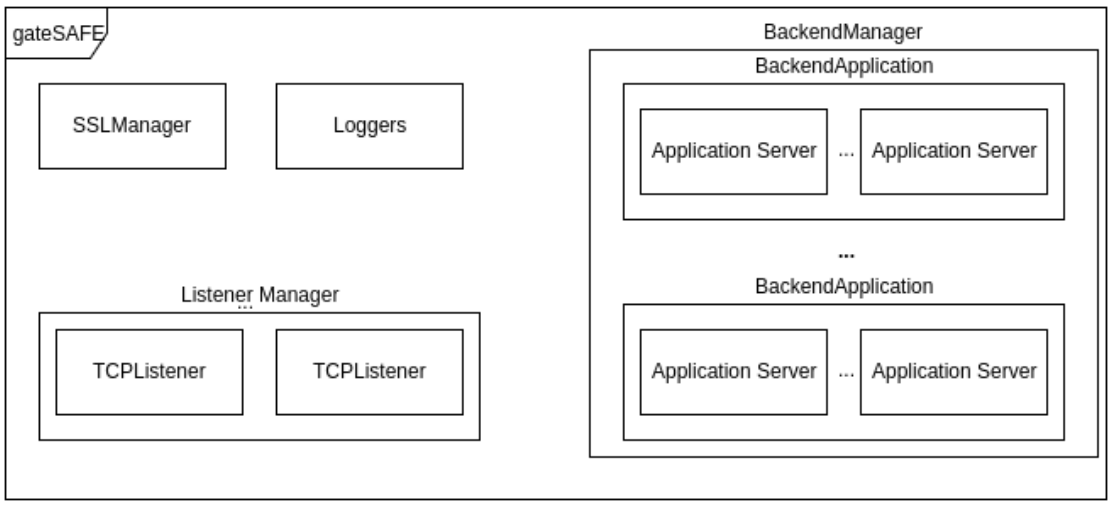

Figure 9 gateSAFE General Architecture

3.3.1.1 SSLManager

This module manages and configures SSL Contexts. It is modular and currently it has an OpenSSL Backend implemented. Depending on requirements, other cryptographic back-ends could be used. The SSL Manager takes into consideration appropriate values for implementing the SSL cryptographic engine:

- **CA File**: Configures the Certificate Authority trust chain;
- **Cipher Suite**: This module can enable or disable, in any combination, specific ciphers that are or become deprecated; this property is usually used to disable weak configuration ciphers and to enable stronger ones.
- **Certificate File**: The Digital Certificate to be used in TLS; when used with a software key, this file also keeps the private key too.
- **Crypto Engine**: In the OpenSSL implementation, the hardware devices used to safely keep the private key or to efficiently accelerate the cryptographic algorithms are called engines. Using this option gateSAFE can move the cryptographic algorithms processing overload from the computer CPU to a crypto-accelerator, or switch to a Hardware Security Module (HSM).

The gateSAFE SSL Manager has the possibility to configure the X.509 authentication mechanism to either always request a certificate from peer, optionally request the certificate, or never request, the authentication being server-side only.

gateSAFE implements three different methods of loading the Trusted Certification Authorities: from a file, from LDAP, or from a trust list. There are also two different methods for validating certificates: against Online Certificate Status Protocol (OCSP) or against Certification Revocation Lists (CRLs) that are always loaded from a LDAP server and periodically refreshed.

3.3.1.2 ListenerManager

Besides the SSL module and together with the authentication modules, the Networking module is one of the most important ones of gateSAFE. The entire networking engine implemented is asynchronous and takes advantage of the modern signalling mechanism in the Linux Kernel that enables virtually any number of networking connections. As the Networking module is layered, it offers a service for the ListenerManager: an upper layer that offers, as well, a service for accepting connections from clients (HTTP clients). The ListenerManager is one of the most important components, as it represents the entry point where the entire connection life-cycle management starts. Being an application that balances not one connection per CPU thread, but multiple connections per one thread, it is the starting point of connection management and it takes decisions of balancing the new connections. It saves the new connections to the priority queues, and after a short pre-processing, it is the WorkingManager that further handles the upper-layer logic.

It is mandatory that, through his direct components (TCP Listeners), ListenerManager processes as fast as possible the connection initiation phase, or otherwise gateSAFE application might appear, to new clients, as blocked and not responding.

3.3.1.3 TCPListener

The TCP Listener can be considered the main entry point of this software. This module sets up a TCP listener and binds to an address and port, handling incoming connections. It is non-blocking and event based, and dispatches all events received to an Event Dispatcher. The implementation works well for Linux and for Windows as well; it has a common core, the networking, and the Event Monitoring component is implemented differently for the two different kernels. This is one of the components that applies the paradigm "security-by-design". The authentication is mapped on "security levels" such that if a resource access is only accessible for authenticated users, it is mapped on a higher security level. The access of a higher security level listener to a lower one is permitted, but not the other way around. In case an unauthenticated user, from an unauthenticated listener, needs to access an authentication protected resource, that listener never performs authentication, but a redirect to the higher security level one is performed. The reason is that an unauthenticated listener is never permitted to perform any authenticated operation.

The timeout mechanism implemented at the TCP Listener ensures that no connection stalls for an unlimited period.

There is a direct binding between each TCP Listener, the instantiated backends and the SSL contexts that creates a chain of actions to be performed on the proxy-ed connection.

3.3.1.4 BackendApplication

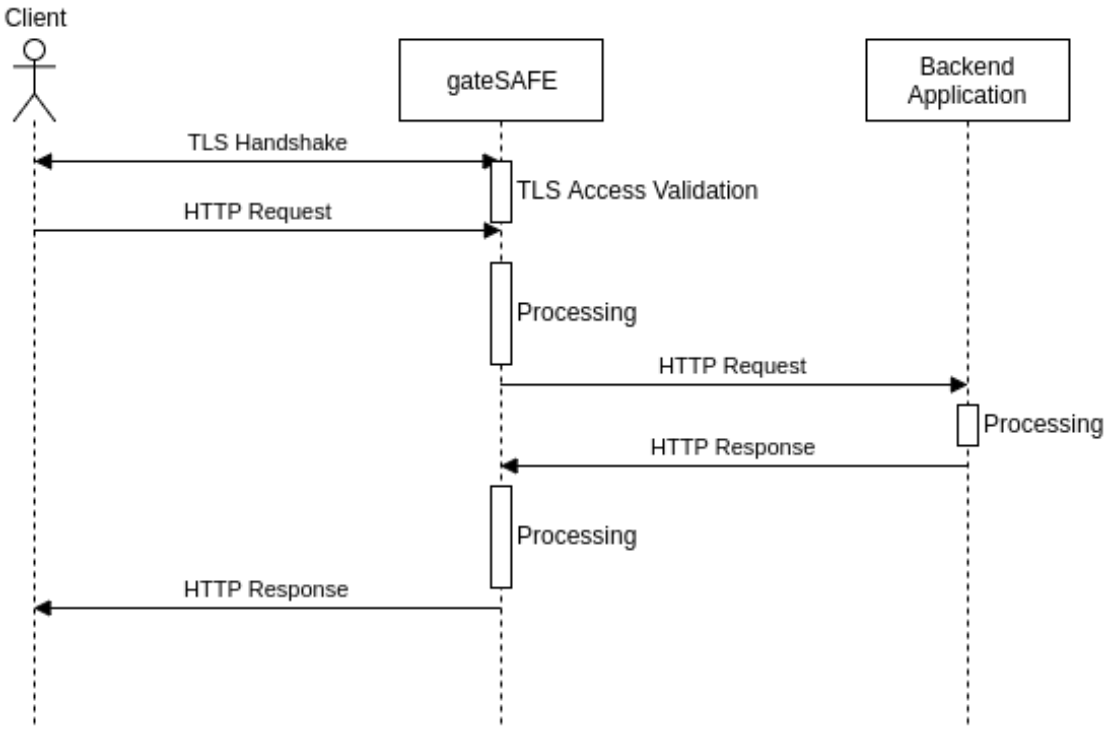

Figure 10 gateSAFE communication protocol

The concept of Backend represents, in gateSAFE, the web application that is proxy-ed. As the "Listener" represents the server part of the proxy, the Backend Application represents the client part

of it, as depicted in Figure 10. There is a mapping of 1-n between the backend application and backend connections.

In the backend application resides a complex resource selector that takes all the decisions of selecting URLs to be served to the web client.

3.3.1.5 BackendManager

This module contains a collection of BackendApplications and security policies. Interfaces with TCPListener to provide matching Backends for a specific Listener. Security policies are defined as tuples providing a mapping between the backend id and a security level.

3.3.1.6 EventDispatcher

An Event Dispatcher is used to handle events on connections. There are two types of Event Dispatchers:

- Listener Event Dispatcher: Handles TCP Listener events, allocates a `Job` instance, and associates the connection with the instance.
- Server Connection Event Dispatcher: Handles events on connections initiated by the gateSAFE, to the Backend Application Servers.

Both event dispatchers are realized implementing the IEventDispatcher interface. The interface is pretty straight-forward: on any event that happens, taking advantage on the C++ polymorphism mechanism, the corresponding implementation is called:

```
class IEventDispatcher
{
   virtual void onEvent() = 0;
};
```

3.3.1.7 Plugins Mechanism

gateSAFE is implemented using modules called plugins and using the same mechanism can also be extended. The plugins implement a specific interface, *IGS_Plugin_Request* and implements functions like *onNewRequestStart*, registering calls in the corresponding proxy-ed connection life-cycle.

```
class IGS_Plugin_Request
{
   virtual bool        onStart() = 0;
   virtual bool        onNewRequestStart() = 0;
   virtual bool        onNewRequestDone() = 0;
   virtual bool        onRequestDone() = 0;
   virtual bool        onNewResponse() = 0;
   virtual bool        onResponseDone() = 0;
   virtual bool        needReadBody() = 0;
   virtual bool        needDropBody() = 0;
   virtual bool        needGenerateResponse() = 0;
   virtual bool        waitingFor() = 0;
};
```

3.3.1.8 Job

A Job instance gets created when a connection is established from TCPListener and lives as long as the client connection exists. It implements a Finite State Machine style processing, and allows plugins to extend the functionality. All processing steps happen on an event basis coming from:

- The client connection which is active during the entire Job lifecycle;
- Multiple connections to backends, as the client generates requests and changes the URIs;

A job can go through the following states:

- **Start**: This job was just created. In this state, we are waiting to receive data initial data on the connection. A job can get into this state multiple times during its lifecycle. A callback for plugins is executed to inform the plugin that a new connection is being handled;
- **SSL Opening**: If the TCP Listener handles HTTPS, in this state we're waiting for the SSL Handshake to be done;
- **OCSP Validation**: When OCSP Validation is enabled, after completing the SSL Handshake phase, this state does OCSP Validation and determines whether the connection should be refused or not;
- **Read Request**: In this processing phase, the HTTP request is being parsed;
- **Plugin New Request Start**: After the response has been parsed, plugins get the chance to look at the request;
- **Plugin New Request Done**: Plugins can do heavy processing in the background and the Job will wait asynchronously. In this state, an event from plugins has been received that the processing has finished.
- **Plugin Need Generate Response**: Plugins are allowed to generate a response to client and even terminate connections. This is useful for accounting purposes, e.g., client has used up his credit, or a second authentication step has failed;
- **Backend Selection**: In this phase, the request is matched with a Backend and the relevant security checks are being made;
- **Plugin Backend Selection Start**, **Plugin Backend Selection Done**: After a backend has been selected, plugins are informed and get the chance to do processing;
- **Backend Open**: In this state, a connection to the relevant Backend Application Server is made;
- **Preprocess Request**: Now that the Backend Application is known, a decision whether Single Sign On (SSO) is required is made, and the relevant actions are taken;
- **Process Request**: In this state, additional headers are appended and the request headers are being rewritten for this particular Backend Application Server;
- **Request Read Body**: The HTTP Message Body is being read;
- **Write Request**: The transformed HTTP Request is being sent to the Backend Application Server;

- **Read Response**: The Backend Application Server is expected to send a response to the request message;
- **Process Response**: The response is being processed, particular headers are removed and are being changed so the entire process is transparent to the end user;
- **Write Response**: The changed response is being transferred to the client;
- **Error states**:
    - Backend not found: No backend has matched the requested client URI;
    - HTTP Error: 302, 400, 403, 404, 503, 504, the configured page error will be sent to the client.

3.3.2 Configuration Design

The configuration is based on an observer design-pattern with one central *ConfigurationManager* having each component configuring itself through the concept of *Managers* (implemented as singletons). In this way, the configuration, although is represented in a compact XML, is distributed to each component; each component that has a configuration is called to be "auto-configurable" by implementing the `IConfigurable` interface

```
class IConfigurable
{
   /// implement To receive properties from the ConfigManager;
   virtual bool    addProperty() = 0;

   /// implement to receive an "end config" message
   virtual void    onConfigEnd() = 0;

   /// implement to register "this" to the config manager
   virtual void    registerClass() = 0;
};
```

3.3.2.1 SSLManager

The XML configuration elements for *SSLManager* are:

- **engine**: OpenSSL engine to be used. Using this option, the user can switch to using a Hardware Security Module (HSM);
- **crl_refresh_period:** Refresh period for CRLs in minutes;
- **certificate_file**: The Digital Certificate to be used in TLS;
- **id**: A unique identifier for this SSL Context;
- **verify_depth**: How many levels of CAs should be verified on the Trusted Chain;
- **request_certificate**: Configures whether the client can provide a certificate, i.e. do mutual authentication;
- **require_certificate**: Configures whether the client supplied certificate is mandatory;

- **required_certificate_extension**: Access Control specific option, mandates a Certificate Extension with a specific Object Identifier (OID) and value;
- **ca_file**: Configures the Certificate Authority trust chain;
- **ciphers**: Configures which ciphers are offered and accepted in the context of TLS;
- **use_crls**: Configures whether the server should use and check CRLs;
- **ldap**: Configures LDAP for authentication;

3.3.2.2 TCPListener

The XML configuration elements for *TCPListener* are:

- **proto**: "http", "https". Specifies which protocol it should handle;
- **interface**: which interface it should bind on;
- **fqdn**: The Fully Qualified Domain Name assigned to this IP and Listener;
- **sec_level**: A security level assigned to this Listener. Used when matching security levels with users and backends.
- **use_ssl_context**: Specifies which SSL Context should be used, since a can be used by multiple Listeners;
- **backends**: A list of backends associated with this Listener;
- **default_backend**: The default backend to connect to when there's no URI match;
- **timeout_secs**: The timeout in seconds which should be used for dropping unresponsive connections;

3.3.2.3 BackendApplication

The XML configuration elements for *BackendApplication* are:

- **protocol**: "http", "https", specifies which protocol should be used to connect to the backend application;
- **id**: a unique id. Used to bind the backend application to the relevant TCPListner (backends, default_backend);
- **servers**: A list of servers for this backend application. Since the user can specify multiple servers, gateSAFE can act as a load balancer with one of the following policies (lb_policy attribute):
    - round_robin: Simplest one, always chooses the next server in a fairly way;
    - min_resp_time: Favors response time (latency) in detriment of throughput;
    - min_time_load: Favors loading time (throughput) instead of latency.

3.3.2.4 BackendApplicationServer

The XML configuration elements for *BackendApplicationServer* are:

- **hostname**
- **port**
- **max_conns**: the number of maximum established connections allowed to this server;
- **max_conc_opens**: the number of maximum pending connections allowed to this server;
- **timeout_secs**: a timeout value in seconds used for dropping unresponsive connections.

3.4 FIDO UAF Server

For device to service first factor authentication, ReCRED uses the FIDO UAF protocol. Services deploy the FIDO UAF Server to register and authenticate the device. The Spring Web MVC framework is used for implementing the server which allows for loose coupling through dependency injection, easy endpoint definition, persistence schemes and other important development aspects such as unit testing among other things.

Besides the modules shared with the client implementation described in chapter 2.4, the server side adds an extension to the cryptographic module with the **CertificateValidator** and **Notary** classes: the first validates the attestation root certificate of the authenticator and the second applies an HMAC to the server data member of the protocol messages to ensure that the response matches a given request.

A Spring REST Controller is declared for every distinct protocol operation. It defines endpoints for the protocol message request and response. This functionality is achieved with the **@RequestMapping** Java annotation which can define the endpoint and set content-type for the HTTP request and response. The HTTP POST request body is accessed with the **@RequestBody** annotation. For the HTTP response body, the **@ResponseBody** annotation is used: it automatically converts Java classes to JSON objects.

```
@RequestMapping(value = "/v1/authentication/response", method = RequestMethod.POST,
    headers = {"content-type=application/fido+uaf},
    produces = "application/fido+uaf; charset=utf-8")
```

When an endpoint is requested by the client, a function that declares the above annotations can route to the service which best suits the protocol operation. These services, i.e. the **RegistrationService** class and the **AuthenticationService** class, can both be found in the **eu.recred.fidouafsvc.service.impl.*** package. They make use of the **FetchRequestService** class to create a registration/authentication request object which the controller will automatically convert to a JSON object, and the **ProcessResponseService** to process the registration and authentication response.

FetchRequestService uses the **BCrypt** utility class from the cryptographic module to generate the server challenge for the registration and authentication request that will be signed by the client. The server data field contains this challenge, the timestamp when it was created and if the request to be

generated is for the registration operation, the username is also appended. After that an HMAC is applied to the data and base64url encoded.

The **RegistrationRequestGeneration** and **AuthenticationRequestGeneration** classes (from the eu.recred.fidouafsvc.ops package) are responsible for creating the server registration/authentication policy by fetching metadata statements from the database.

ProcessResponseService calls the **Notary.verify()** function to check the server data HMAC and verify that the response matches the request generated for it. After verifying the server data and channel binding parameters, the authenticator assertions are checked. For the registration assertion, the server checks the attestation root signature and certificate with the ones stored in the metadata statement matching the authenticator. For the signature, the verification algorithms contained in the cryptographic module are used:

```
private boolean validate(byte[] certBytes, byte[] signedDataBytes,
                         byte[] signatureBytes) throws NoSuchAlgorithmException,
        IOException, Exception {
    X509Certificate x509Certificate = X509.parseDer(certBytes);
    logger.info(" : Attestation Cert : " + x509Certificate);

    String sigAlgOID = x509Certificate.getSigAlgName();
    logger.info(" : Cert Alg : " + sigAlgOID);

    try {
        if (signatureBytes.length == 256) {
            if (!RSA.verifyPSS(x509Certificate.getPublicKey(), signedDataBytes,
                    signatureBytes)
        ……
        } else if (signatureBytes.length == 260) {
           byte[] signatureBytesRaw = new byte[256];
           System.arraycopy(signatureBytes, 4, signatureBytesRaw, 0, 256);
           if (!RSA.verifyPSS(x509Certificate.getPublicKey(), signedDataBytes,
                 signatureBytesRaw))
        ……..
        }

        BigInteger[] rs = null;
        if (signatureBytes.length == 64) {
            rs = Asn1.transformRawSignature(signatureBytes);
        } else {
            rs = Asn1.decodeToBigIntegerArray(signatureBytes);
        }
        try {
            if (!NamedCurve.verify(KeyCodec
                    .getKeyAsRawBytes((ECPublicKey) x509Certificate
                            .getPublicKey()), SHA.sha(signedDataBytes,
                    "SHA-256"), rs))
        ……
        } catch (Exception fromVerify) {
            if (!NamedCurve.verifyUsingSecp256k1(KeyCodec
                    .getKeyAsRawBytes((ECPublicKey) x509Certificate
                            .getPublicKey()), SHA.sha(signedDataBytes,
                    "SHA-256"), rs))        }

    } catch (Exception thrown) {
    ……..
    }
}
```

The signature length is checked to verify if it arrived in encoded or raw form.

Authenticator assertion verification for the authentication response is performed by the **verifySignature()** method of the **ResponseHelper** class. This method follows the same logic for signature verification as the **validate()** method presented above but with added key decoding capabilities in case that at the registration process the authenticator sent an encoded public key for signature verification.

Deregistration is handled by the **DeregistrationRequestProcessorService** class (located in the eu.recred.fidouafsvc.service.impl package). This service is called by the same controller which handles registration and creates the request based on the keyID and authenticator AAID for the username received from the client, then deletes the registration record.

```
public DeregistrationRequest[] getRequest(String username) {
    DeregistrationRequest[] request = new DeregistrationRequest[1];
    request[0] = new DeregistrationRequest();

    RegistrationRecord record =
storageDao.readRegistrationRecordUsername(username);

    request[0].header = new OperationHeader();
    request[0].header.op = "Dereg";
    request[0].header.appID = new FidoConfig().getAppId();
    request[0].header.upv = new Version(1, 0);
    request[0].authenticators = new DeregisterAuthenticator[1];
    request[0].authenticators[0] = new DeregisterAuthenticator();
    request[0].authenticators[0].aaid = record.authenticator.AAID;
    request[0].authenticators[0].keyID = record.authenticator.KeyID;

    storageDao.deleteRegistrationRecord(record.authenticator.toString());

    return request;
}
```

Another important module for the server implementation is the storage. It takes advantage of the Spring framework configuration to integrate the ORM framework (Hibernate) and database (H2).

```
jdbc.driverClassName = org.h2.Driver
jdbc.url = jdbc:h2:file:/db/fidouaf.db
jdbc.username = myuser
jdbc.password = *****

hibernate.hbm2ddl.auto = update
hibernate.dialect = org.hibernate.dialect.H2Dialect
hibernate.show_sql = true
hibernate.format_sql = true
```

After configuring the Hibernate ORM, POJO models (simple Java classes) that need persistence can use Java annotations such as **@Table**, **@Id**, **@Column**, etc. to define the table schema in the database. To persist data, the Data Access Object design pattern is used, it is implement with the **StorageDao** class (eu.recred.fidouaf.svc.dao). This class hides the Spring SessionFactory class usage which handles database queries and transactions and maps model classes to rows in the database.

4 Behavioral Authentication Authority

4.1 Architecture

The Behavioral Authentication Authority (BAA) is one of the key components of the ReCRED architecture. Its main goal is to monitor user behavior, build behavioral profiles, and finally use built profiles to accommodate user authentication requests on behalf of relaying parties. As such, the BAA uses the behavioral factors discussed in Chapter 3.1.

Figure 11 provides an overview of the BAA architecture and its interfaces with the other components of the ReCRED architecture. The ReCRED daemon component receives user's data through the Behavioral Profile (BP) Capture module, creates a behavioral profile and stores it in a BAA database for later use. The BP Verification module serves as an entry point for user's profile verification. As such, it loads user's behavioral profile from the BAA database, loads user's most recent data from an external database and then compares BP with a signature created from the data.

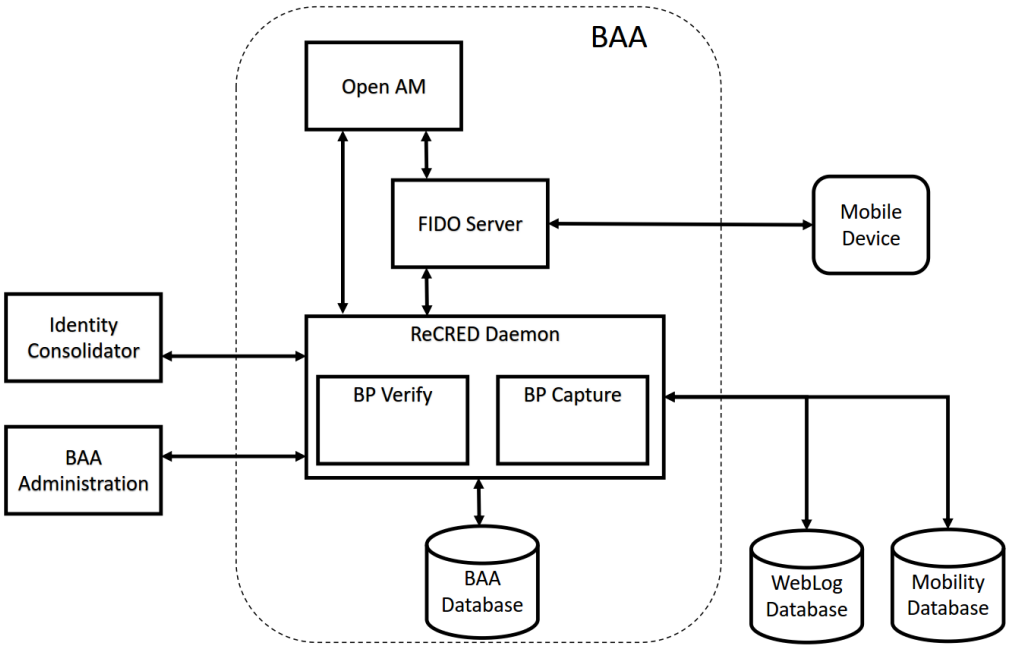

Figure 11 BAA Architecture and interfaces overview

Other modules in the diagram are used to secure connection and communication with the BAA, accept behavioral data from a mobile device, etc. In the following sections, we provide details of the BAA and we showcase it using three modalities: browsing behavior, mobility behavior and keystroke behavior. The first two modalities correspond to a BAA that monitors user behavior without requiring any changes to the user device; the last modality shows the workflow of a BAA where behavioral data is extracted from the devices using an ad-hoc application.

4.2 BAA's ReCRED daemon for Browsing Behavior

The ReCRED daemon consists of two main modules – BP Capture an BP Verify. Figure 12 depicts all modules and external entities that communicate with the ReCRED daemon in order to authenticate users based on their browsing behavior. Red boxes in the diagram denote REST API entry points, the

text shows URL address suffix. Black arrows show calls between different modules. Each arrow shows which module calls which REST API: an empty end denotes a caller; a sharp end denotes the end-point.

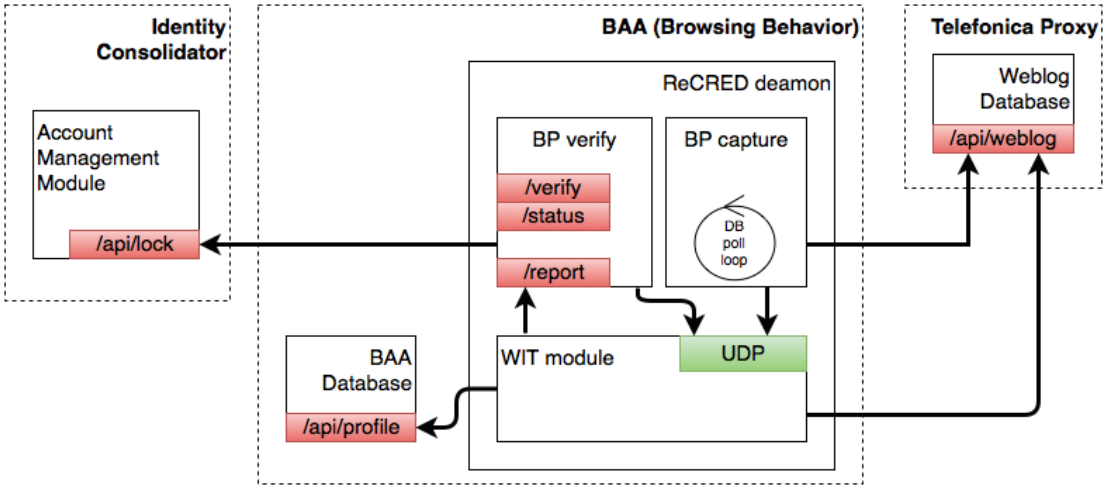

Figure 12 BAA'a ReCRED daemon for Browsing behavior and comminicating external entities

4.2.1 Description of modules

- **BAA Database**: stores behavioral profiles of different users. Offers REST API at */api/profile*.
- **Weblog Database**: stores user data and lies within a security perimeter of Telefonica. Contains a timestamped history of web page addresses visited by users. Offers REST API at */api/weblog*.
- **BP Verify**: accepts verification requests. These are passed to the WIT module which calculates similarity of the behavioral profile of a user and that user most recent behavioral (i.e., browsing) data.
- **BP capture**: periodically polls the Weblog Database and feeds WIT module with user's browsing data.
- **WIT module**: Web Identity Translator (WIT) module is used to fingerprint users based on the web addresses they visit. Accepts UDP messages, stores BP profiles into BAA Database over a REST API, queries Weblog Database for user's recent data, reports signatures similarity to BP verify module.
- **Account Management Module**: a part of the Identity Consolidator, implements locking functionality that allows locking user-defined accounts when suspicious behavioral activity is detected. The BP verify module reports deviation between a user behavioral profile and it recent behavior through REST API exposed by the AMM.

Profiles are created during a learning phase and its length is a parameter set by the WIT module. It could be time-dependent, for example a 10 days long learning phase, or it can be message-driven, so that the first 100 web browsing addresses are used for training.

Once a profile is created, it is stored in the BAA Database to achieve persistence and robustness in case the BAA has to be restarted. The BAA Database can also be used to inform users about what type of information is it stored and how their behavioral profile looks like.

There are two main modes of operation of the BAA: authentication on demand and continuous authentication. Authentication on demand is invoked externally, by third parties using */verify* or

/status APIs. Through those APIs, third parties get a confirmation that a user recent behavior corresponds to his behavioral profile. Continuous authentication is a proactive process in which the WIT module periodically checks for a discrepancy between a user recent data and his behavioral profile; when a mismatch is found, the result is reported to the Identity Consolidator's Account Management Module which implements the account locking functionality. The BAA signals suspicious activity for a user and it is up to the Account Management Module to decide whether to lock all/some of that user accounts or not.

4.2.2 Database structure

The structure of weblog database is rather simple. The internal representation is a tuple *(timestamp, user_id, web_address)*. This allows for filtering of records for particular users as well as to limit the DB query to select records according to time.

The structure of the BAA database is a histogram of web addresses visited by a user. It is represented as a tuple *(user_id, web_address, count)* where count denotes number of visits at the particular web_address during the learning phase.

4.2.3 Message flows – Learning phase

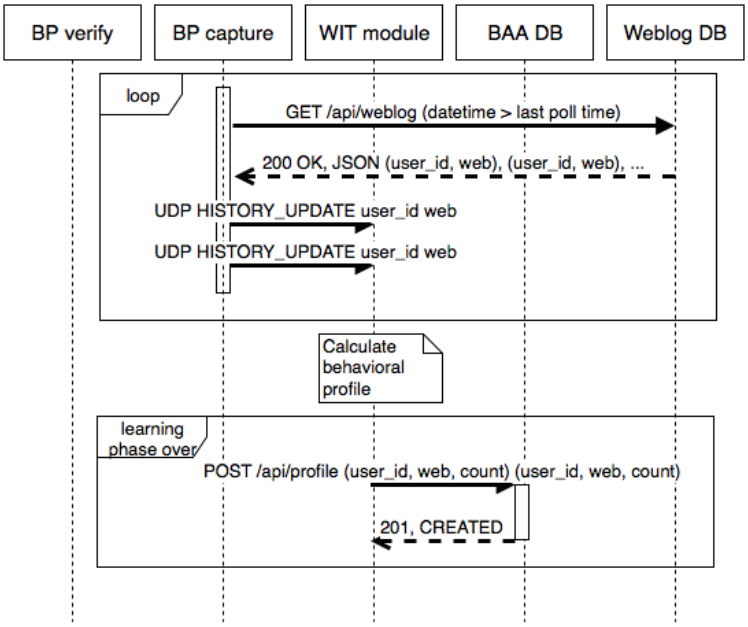

Figure 13 Message flow for the learning phase of the BAA (Browsing)

The message flow for learning phase of the BAA is depicted in Figure 13. Once a BAA instance is started, the polling of a Weblog database takes place. Data is regularly requested via REST API on the */api/weblog end-point*. Polling frequency is a configurable parameter. Only data newer than the last request is fetched; this is achieved by passing a date-time filter in the GET query string.

A response from the Weblog DB contains a list of tuples *(user_id, web_address)*. Each tuple represents one visit of a user on a particular web address. The BP Capture module then feeds those tuples into a WIT module through UDP messages in a form of *HISTORY_UPDATE user_id web.*

Once enough messages are received by the WIT module, or after a specified learning time, the behavioral profile is created for the user.

The profile is then posted to the BAA database via REST API through the POST method in which the data part contains tuples *(user_id, web, count)*. Successful creation of a database entry is confirmed by a CREATED response.

4.2.4 Message flows – Profile verification

The message flow to demonstrate how verification of a profile works is depicted in Figure 14.

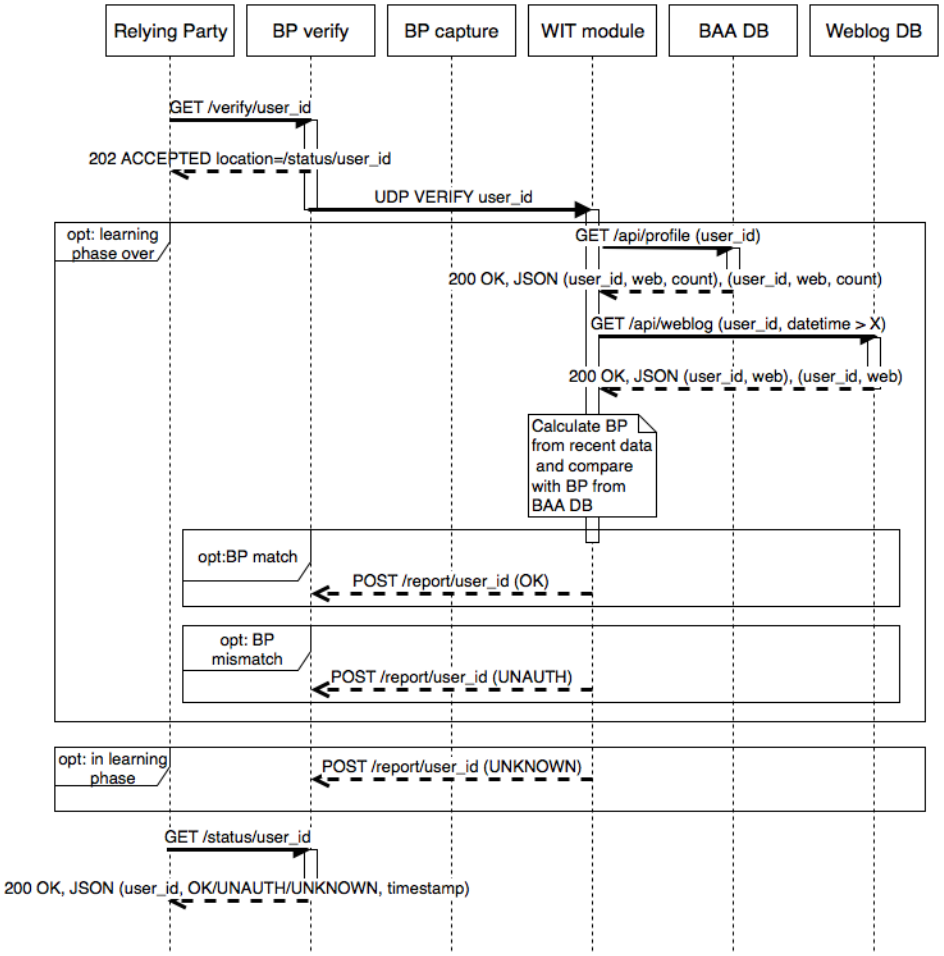

Figure 14 Message flow for profile verification (Browsing)

Any party that wants (and is allowed) to verify a user behavior posts a GET message via a REST API to the BP verify module endpoint */verify/user_id*, where *user_id* is the system-wide user identifier. The response is accompanied with JSON data that contain a UTC Unix timestamp denoting when the WIT module received the request.

Since the verification of a user behavioral profile could be a time-consuming process in which many parties and modules are involved, an immediate response is returned that confirms an accepted request and contains a location header with an URL of the result.

The verification of a profile is requested by the BP verify module by sending an UDP message *VERIFY user_id* to the WIT module. No response is awaited through the same channel. Based on the result of profile verification, the WIT module later posts a response to the BP verify module’s endpoint */report/user_id*. There are three possible answers:

- **OK** means that the user's behavioral profile corresponds to his recent history
- **UNAUTH** means that the user's behavioral profile doesn't correspond to his recent history
- **UNKNOWN** means that a learning phase is still in place for that particular user, or in case of an error during BP verification processing

To achieve behavioral profile verification, the WIT module first queries BAA DB on the */api/profile* REST API interface with a query string specifying user's *user_id*. Next, it queries the Weblog database to get recent history of user's browsing. The history extent is an internal parameter and can be changed. As such it is passed in a query string in GET method request on the */api/weblog* REST API. The comparison between user's pre-calculated profile and the recent history is carried out and the result is returned to BP verify module.

Finally, the Relying party asks for verification for a specific user via */status/user_id* REST API end-point using a GET method. The response contains *user_id*, the result of the verification and a timestamp. The relying party should compare the timestamp received in verify response with status response. If the timestamp of status response is smaller than verify response timestamp, it means that the verification process still takes place.

Note that the Relying party can skip the verify request and just query the status of user verification. The continuous BP profile verification in background (if turned on) updates automatically user verification status. Based on the timestamp, the Relying party can decide if the most recent status is fresh enough to serve as a trusted reference or if it should invoke the verify request to obtain a more recent status.

4.2.5 Modules, APIs and message descriptions

4.2.5.1 BP Verify

Sends VERIFY UDP message to the WIT module and invokes the */api/lock* message in the Identity Consolidator's Account Management Module.

4.2.5.2 /status/<user_id>

REST API, supported methods: GET

Request: `GET localhost:6701/status/<user_id>`

Response: `OK 200, {'user_id': user_id, 'status': status, 'timestamp': timestamp}`

where status is one of OK, UNKNOWN, UNAUTH and timestamp is UTC Unix timestamp of status validity

4.2.5.3 /verify/<user_id>

Accepts request to verify a user with user_id.

REST API, supported methods: `GET`

Request: `GET localhost:6701/verify/<user_id>`

Response: `ACCEPTED 202, {'timestmp': timestamp}`

where timestamp is UTC Unix timestamp on WIT module, thus an indication which timestamp in status response will mean that verification of the BP profile is done.

4.2.5.4 /report/<user_id>/<status>

Serves for response from the WIT module, to store user’s status.

REST API, supported methods: `POST`

Request: `POST localhost:6701/report/<user_id>/<status>`

where status is one of **OK**, **UNKNOWN**, **UNAUTH**

Response: `OK 200, {}`

4.2.5.5 BP capture

`localhost:6701`

Listens on port 6701. No API exposed. Sends HISTORY_UPDATE UDP messages to the WIT module and polls the Weblog database by sending GET messages to weblog DB’s */api/weblog* end-point.

4.2.5.6 WIT module

`localhost:9999`

Listens on port 9999 and accepts UDP datagrams. Accepted messages are:

- **HISTORY_UPDATE**
- **VERIFY**
- **ECHO** (for testing purposes)

Sends GET messages to the BAA database */api/profile* and to the Weblog database */api/profile*.

4.2.5.7 Weblog Database

`localhost:5688`

Listens on port 5688.

4.2.5.8 /api/weblog

Accepts requests to get user’s browsing data (web address).

REST API, supported methods: `GET`, (`POST`, `PATCH` and `DELETE` for debugging purposes)

Request: `GET localhost:5688/api/weblog/`

4.3 BAA's ReCRED daemon for Mobility Behavior

The ReCRED daemon that authenticates users based on mobility patterns consists of two main modules – BP Capture an BP Verify. Figure 15 depicts all modules and external entities that communicate with the ReCRED daemon in order to authenticate users based on their mobility patterns.

Red boxes in the diagram denote REST API entry points, the text shows URL address suffix. Black arrows show calls between different modules. Each arrow shows which module calls which REST API: an empty end denotes a caller; a sharp end denotes the end-point.

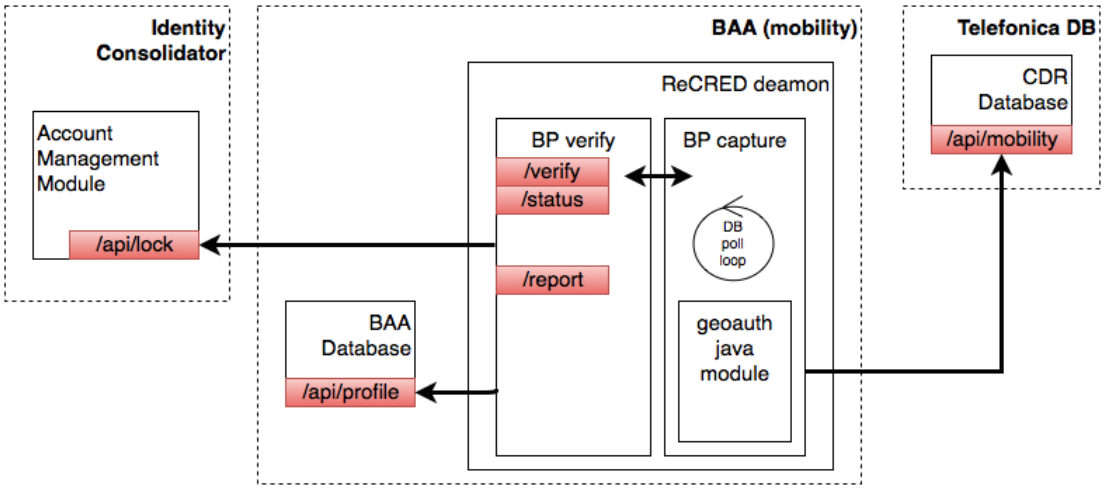

Figure 15 BAA's ReCRED daemon for Mobility behavior and comminicating external entities

4.3.1 Description of modules

- **BAA Database**: stores behavioral profiles of different users. Offers REST API at */api/profile*.
- **CDR Database (Mobility Database)**: database with users Call Data Records in a security perimeter of Telefonica. Contains a time stamped history of calls placed or received or text messages sent and received. Offers REST API at */api/mobility*.
- **BP Verify**: accepts verification requests. These are passed to the BP Capture module which computes the similarity of a user behavioral profiles and that user most recent behavioral data.
- **BP Capture**: periodically polls the Mobility Database and feeds it with user's mobility data.
- **geoauth java module**: a set of Java functions, wrapped in one *geoauth* package. It consists of four functions:
    - *CDRsLocationVectorExtended* parses mobile CDRs and construct location vectors per user (aka behavioral mobility profile). The mobility profile is stored in BAA Database.
    - *CDRsVector RandomPairwiseFixedComparisonExtended* computes cosine similarity between vector of a user-set provided with X random users from the population. Similarity score is then used for later use.
    - *CDRsVectorSameUserComparisonExtended* is used to compute cosine similarity between the historical profile and current fingerprint, obtained from recent data, of the same user.

- *CDRsRankSimilarityExtended* computes the rank of the user when comparing his self-similarity score with random other user's pairwise scores. The final rank is then used to decide on authentication: if user's rank scores best among other users, then the user is considered as being different enough from all other users and thus is authenticated.

- **Account Management Module**: a part of the Identity Consolidator, implements locking functionality that allows locking user-defined accounts when suspicious behavioral activity is detected. The BP verify module reports deviation between a user behavioral profile and its recent behavior through the REST API exposed by the AMM.

As per the browsing-based functionality, the duration learning phase can be parametrized based on time or number of events. Once a profile is calculated, it is stored in BAA Database to achieve persistence and robustness.

Also, here there are two main modes of operation: authentication on demand and continuous authentication. Authentication on demand is invoked externally, by third parties using */verify* or */status* APIs. The purpose is to get confirmation that a user's recent behavior corresponds to his behavioral profile. Continuous authentication is a pro-active process in which the BP Capture module periodically checks for a discrepancy between a user recent data and his behavioral profile. Mismatches are reported to the Identity consolidator where the Account Management Module decides whether to lock accounts or not.

4.3.2 Database structure

The structure of the CDR (mobility) database is rather simple. The internal representation is a tuple *(timestamp, user_id, cdr)*. This allows for filtering for records for particular users as well as to limit the DB query to select records according to time.

The structure of the BAA database for storing behavioral profile is a vector of visited cells together with a corresponding *tf-idf* value per user. It is represented as a tuple *(user_id, caller_vector, callee_vector)* where *caller_vector* denotes vector of cells and *tf-idf* values for calls in which the user is a caller; *callee_vector* has a similar meaning except that the user is a callee.

4.3.3 Message flows – Learning phase

The message flow for the learning phase of the BAA is depicted in Figure 16.

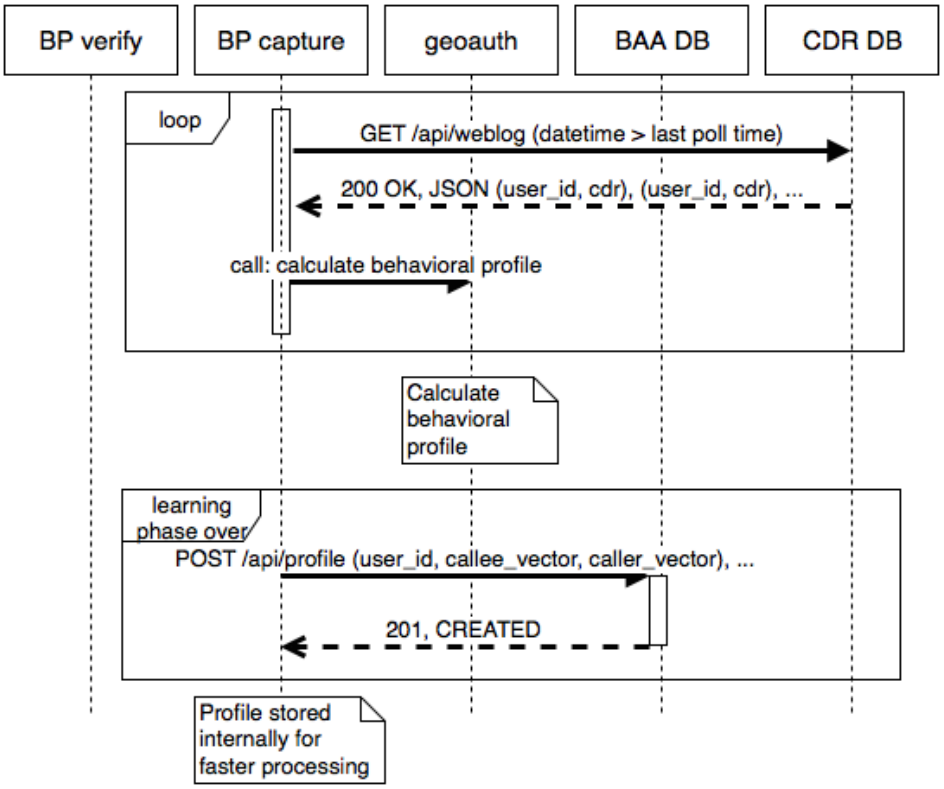

Figure 16 Message flow for the learning phase of BAA (Mobility)

Once a BAA instance is started, a fingerprinting of all users in the database starts.

The response from CDR DB contains a list of tuples *(user_id, CDR).* Each tuple represents one call or text message placed/received/sent. The BP Capture module then feeds those tuples into the *geoauth* module through a file. The *geoauth* module returns a user behavioral profile, which is a vector of visited cell-ids together with their corresponding *tf-idf* values.

The profile is then posted to the BAA Database via REST API through the POST method in which the data part contains tuples *(user_id, caller_vector, callee_vector)*. Successful creation of a database entry is confirmed by a CREATED response.

Once the learning phase is over, a regular polling of a database takes place. Data is regularly requested via REST API on the */api/mobility* end-point (polling frequency can be adjusted). Only data newer than the last request is fetched, by passing a datetime filter in the GET query string.

4.3.4 Message flows – Profile Verification

The message flow for profile verification is depicted in Figure 17.

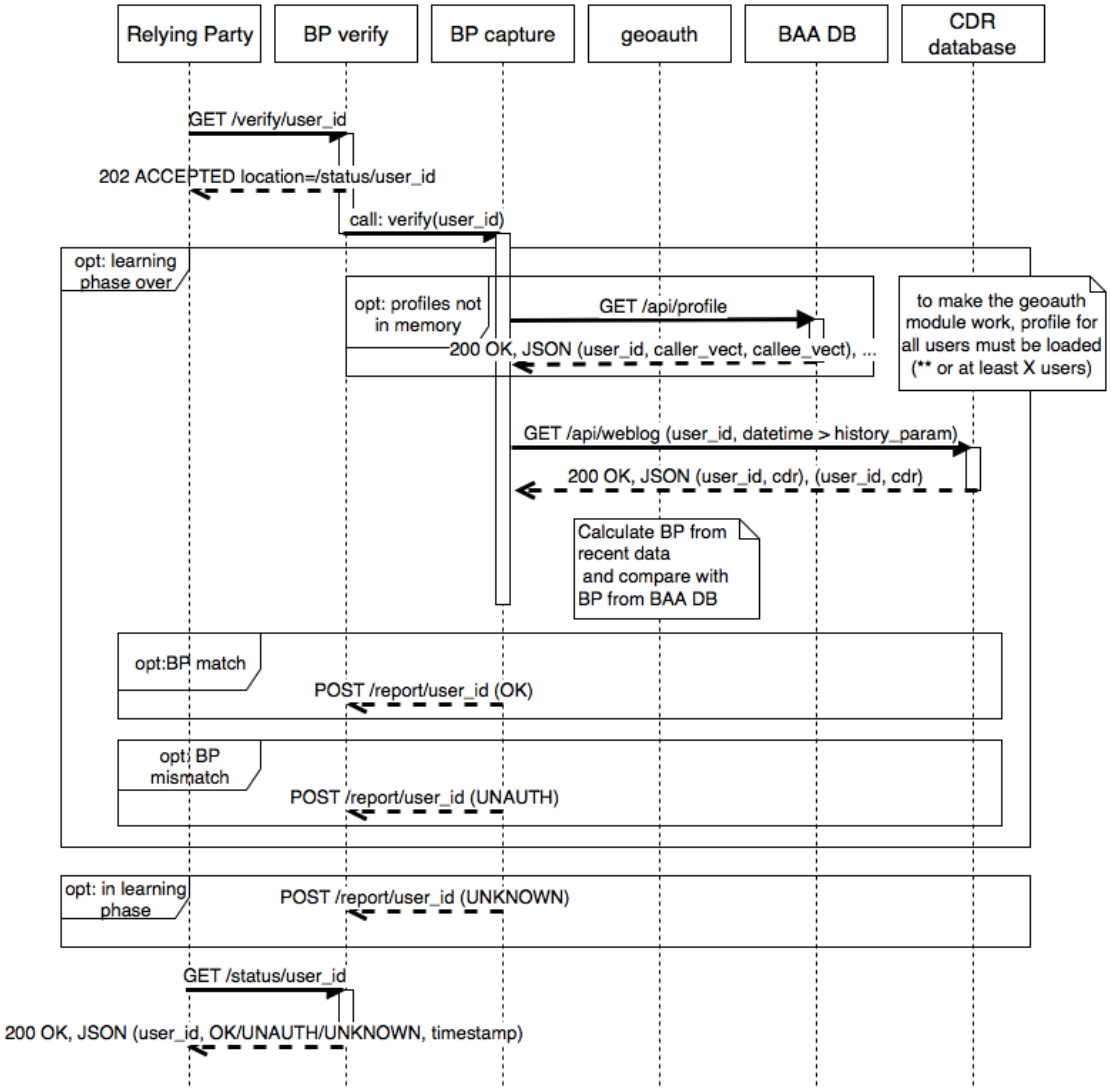

Figure 17 Message flows for profile verification (Mobility)

Any party that wants (and is allowed) to verify a user behavior posts a GET message via a REST API to the BP Verify module endpoint */verify/user_id*, where *user_id* is the system-wide user's identifier. The response is accompanied with JSON data that contain UTC Unix timestamp denoting when the WIT module received the request.

Since the verification of user's behavioral profile could be a time-consuming process in which many parties and modules are involved, an immediate response is returned that confirms an accepted request and it contains a location header with an URL of the result.

The verification of profile is requested by BP Verify module by internally calling verify function of BP Capture module. The call ends by spawning a verification thread because the verification process might be time-consuming. The BP capture later posts a response to BP verify module's endpoint */report/user_id*. There are three possible answers:

- **OK** means that user's behavioral profile corresponds to his recent history
- **UNAUTH** means that user's behavioral profile doesn't correspond to his recent history
- **UNKNOWN** means that a learning phase is still in place for that particular user, or in case of any error during BP verification processing

To achieve behavioral profile verification, the BP capture module queries the BAA DB on the */api/profile/mobility* REST API interface. For the geoauth module it is crucial to have a profile of all users. In order to avoid the overhead of loading the behavior profiles for all users each time, the profiles are requested once and then stored in the BP Capture module's memory. Next, the BP Capture queries the Mobility database to get last user's mobility patterns in form of raw CDRs. The history extent is an internal parameter and can be adjusted. As such it is passed in a query string in GET method request on /api/mobility REST API. The comparison between a user profile and similar representation of the recent history is carried out and the result is returned to the BP Capture module.

Finally, the Relying party asks for status of a user via */status/user_id* REST API end-point using a GET method. The response contains *user_id*, status of verification and a timestamp. The Relying party should compare a timestamp received in verify response with status response. If the timestamp of status response is smaller than the verify response timestamp, it means that the verification process still takes place.

Note that the Relying party can skip the verify request and just query the status of a user. The continuous BP profile verification in background (if turned on) updates automatically user verification statuses. Based on the timestamp, the Relying party can decide if the most recent status is fresh enough to serve as a trusted reference or if it better invokes the verify request to obtain more recent status.

4.3.5 Geoauth module functionality and workflow

In Figure 18 the functionality of geoauth module is explained and its workflow is demonstrated on an example of verifying a particular user. Two steps are necessary: learning and verification. Learning phase starts immediately after BAA's first run. At this time, all CDRs for all users are processed into a historical mobility profile. Next, the picture assumes the verification of a user with mobile number 577. The first step is to query the Mobility Database and obtain a recent history of that user activity. From this snapshot, the recent userprofile is calculated. Using the historical behavioral profile of all users, calculated during the learning phase, the next step is to compare the user recent profile with all historical ones. This results in a similarity score. Finally, a ranking is carried from which a decision is made.

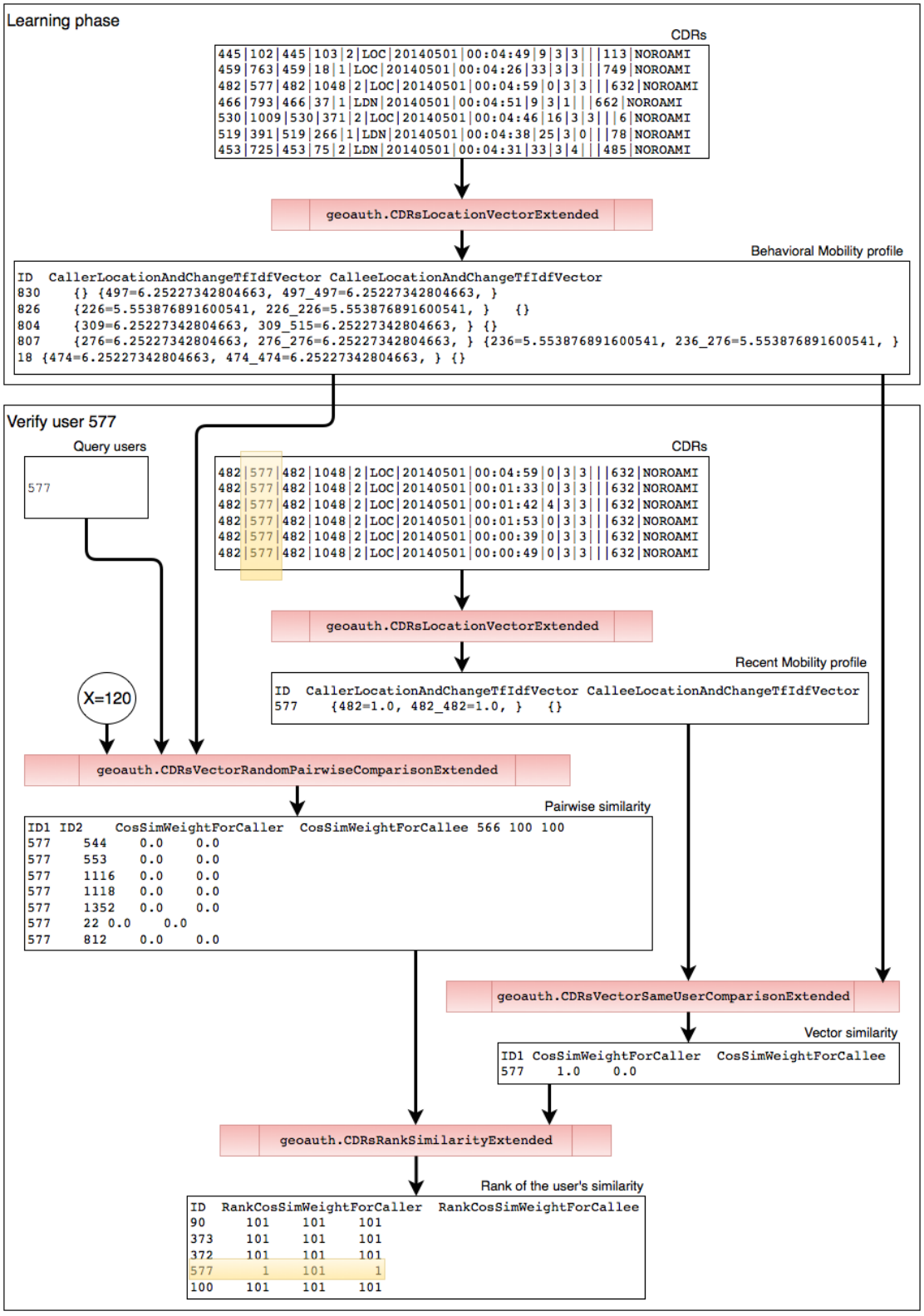

Figure 18 Geoauth module functionality and workflow

4.3.6 Modules, APIs and message descriptions

4.3.6.1 BP Verify

`localhost:6702`

Listens on port 6702. Handles verification.

4.3.6.2 /status/<user_id>

REST API, supported methods: `GET`

Request: `GET localhost:6702/status/<user_id>`

Response: `OK 200, {'user_id': user_id, 'status': status, 'timestamp': timestamp}`

where status is one of **OK**, **UNKNOWN**, **UNAUTH** and timestamp is the UTC Unix timestamp of status validity.

4.3.6.3 /verify/<user_id>

Accepts request to verify user with *user_id*.

REST API, supported methods: `GET`

Request: `GET localhost:6702/verify/<user_id>`

Response: `ACCEPTED 202, {'timestmp': timestamp}`

where timestamp is the UTC Unix timestamp when the request has been accepted, thus an indication which timestamp in status response will mean that verification of the BP profile is done.

4.3.6.4 /report/<user_id>/<status>

Serves for response from verification thread to store user’s status.

REST API, supported methods: `POST`

Request: `POST localhost:6702/report/<user_id>/<status>`

where status is one of **OK**, **UNKNOWN**, **UNAUTH**.

Response: `OK 200, {}`

Multiple verification responses can be passed through a */report* API via a POST method with JSON *{‘user_id’:user_id, ‘status’:status}*

4.3.6.5 BP capture

localhost:6702

Listens on port 6702. No API exposed. Polls CDR database by sending GET messages to mobility DB’s /api/mobility end-point.

4.3.6.6 geoauth

A set of Java functions. All communication is done through multiprocessing and data in various steps are exchanged using temporary files.

4.3.6.7 Mobility Database

`localhost:5689`

Listens on port 6702.

4.3.6.8 /api/mobility

Accepts requests to get user’s CDR data.

REST API, supported methods: `GET`, (`POST`, `PATCH` and `DELETE` for debugging purposes)

Request: `GET localhost:5689/api/mobility/`

4.4 BAA’s ReCRED daemon for Typing-Keystrokes Behavior

In this section, we describe the logic behind the server component of the b-verifier application, described in Section 3.1.4, that is running in the Behavioral Authentication Authority (BAA) for capturing and verifying the user’s typing behavior.

The server that receives the keystrokes from the BVK is the Behavioral Authentication Authority that is connected to the User Device, the ID consolidator and the Service Providers that want to authenticate the user with a second factor behavioral authentication.

The BAA runs a training phase and a verification phase for every device/user and the procedure is the following.

The BAA receives the user keystrokes and runs the following process.

1. Receives the session (set of keystroke sequences).
2. It checks whether the specific *deviceID* has already a typing signature associated with it (for the specific application – appID and for the device in general).
    a. If yes, then the incoming session is sent to the verification component (go to step 4).
    b. If no, then the incoming session will participate to the training phase. It is stored in the database in the appropriate table/file.
3. The **training component** checks if there are enough samples (keystrokes), based on the respective parameter of the configuration, in order to start the creation of the user signature.
    a. If the samples are less than those needed, it performs no action.
    b. If the samples are adequate, then it reads all the training keystrokes and produces the signature for the user. This is done by aggregating the training records, calculating the average and standard deviation of the feature vectors.
    c. The user signature is stored in a separate table, along with the creation timestamp and information needed for the verification.

4. The **verification component** aggregates the session that is to be verified and produces the session signature, using the same process that is used to produce the signature of the user.
5. The aggregated session is then compared with the user signature (using the Euclidean distance metric) and a matching score is produced.
6. The produced scores are stored in the database, along with the session signature, in a separate table that is to be used to answer the authentication request of the Service Provider.

Note that this process of building user typing signatures is done per device and per appID. A cross-application score is also generated every time a signature for a new appID is created.

So, the BAA performs training for every appID that uses the BVK, creates a signature for each one of them and holds the latest N scores for every appID.

A cross-application signature is also calculated and the N latest scores of the comparison of this signature and the latest session are also stored.

These scores are to be used either when a service provider requests a second factor authentication, or to notify the ID consolidator (IDC) if an unusual behavior is noticed from the BAA.

The IDC can then decide to lock some or all user accounts, to notify the user or take no action, according to the user preferences.

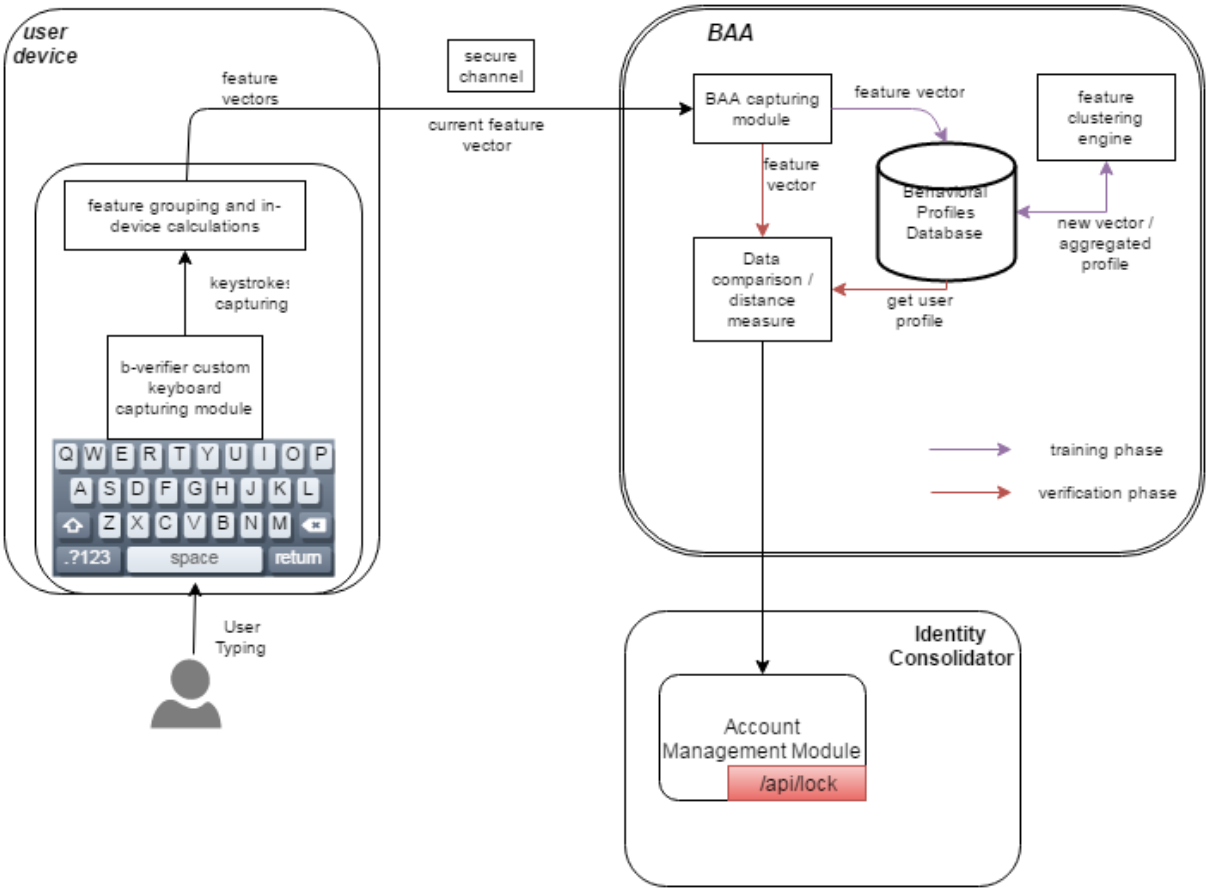

Figure 19 B-Verifier client and server modules

4.4.1 /api/session

Consumes a JSON object that contains the keystrokes of a session. The JSON object should look like:

```
{"session_id": "e74dc7a5d911567b43d14c52f197cd", "device_id": "xxx", "scale_factor": "normal", "density_class": "XXHDPI", "app_id": "com.ogden.memo", "input_type": "search_form(example)", "orientation": "Portrait", "KeystrokesList": [[{"keyCode": -5, "keyDownToKeyDown": 0, "keyDownToKeyUP": 0, "keyUptoKeyDown": 0, "keyUptoKeyUp": 0, "keydownTime": 1478773093970, "keyupTime": 1478773093976}, ...], ...]}
```

REST API, supported methods: `POST`

Request: `POST wildfly.wediacloud.net:8080/ReCRED-KeystrokeDynamics/api/session`

Response: `OK 200, {}`

4.4.2 /api/device/<id>

Accepts request to delete data for the specified device id (i.e. retrain).

REST API, supported methods: `DELETE`

Request: `DELETE wildfly.wediacloud.net:8080/ReCRED-KeystrokeDynamics/api/device/<id>`

Response: `OK 200, {}`

The API calls that are related to the service provided will be soon implemented, while the database related API calls are described in the appropriate section, 4.5.

4.4.3 Retraining

User has the option to set an expiration date for his typing signatures. This is because the user can be more familiar with his device and thus, his typing signature can be changed. This setting is stored in the IDC. When this date is reached, the IDC requests a full reset of the typing profile of the user on the BAA and the BAA deletes all the signatures collected so far and the verification scores produced and starts the process from the beginning (collecting signatures and training the system).

4.4.4 BAA and Identity Consolidator

When a user tries to access a service (e.g. e-banking service) he provides a PIN or password to the service. If this is correct the service can also ask the Behavioral Authentication Authority (BAA) if the user behaves as normal, regarding his typing style.

The service provider contacts the BAA sending the device id that has requested the access.

The BAA queries the database and retrieves all the related scores to this device id for every application.

If there are no typing signatures yet, the BAA responds that it is not possible to answer about that device id.

If there are more than one signatures, the BAA will have either to send them all to the service provider, or, preferably to calculate an average score from all the applications. Weights may be applied to this calculation to promote the recently calculated scores against the older ones.

The BAA sends this final score (a percentage) to the service provider.

The service provider decides if this score is acceptable based on a threshold. This threshold depends on the criticality of the service or the specific action the user is requesting and is for the service provider to decide.

The authentication attempts are stored in a log in the ID Consolidator.

4.5 Behavioral Profiles Database

All the user behavioral profiles are stored in the behavioral profiles database. The behavioral profiles are created by the input that is provided by the behavioral capturing modules on the BAAs and on the user's devices and stored as such.

The behavioral profiles database provides input to the Behavioral profile verification module for the purposes of making sure that the captured profile matches the stored profile. More specifically, a REST API has been implemented, providing the following services:

- **Create a new behavioral profile**
  - The BAA can call a REST method (POST) in order to create and store a new behavioral profile resource in the database.
  - The BAA can store the following types of behavioral profiles: gait templates, keystroke templates, mobility signatures, browsing signatures.
  - Upon successful creation of the behavioral profile, the REST API returns a location header with a link to the newly-created profile, along with the HTTP status `201 CREATED`.
  - The REST API authenticates the BAA and checks that it is authorized to store the specific behavioral profile to the database. Appropriate HTTP statuses are returned in case of authentication (e.g. `401 UNAUTHORIZED`) or authorization failures (e.g. `403 FORBIDDEN`).
  - The REST API checks if the resource already exists and returns the HTTP status `409 CONFLICT` if it does.
- **Retrieve a Stored Behavioral Profile**
  - The BAA can call a REST method (GET) in order to read a behavioral profile stored in the database, including the identifier of the profile's owner and the type of behavioral profile (gait, keystroke, mobility, browsing).
  - Upon successful retrieval of the resource, it is returned in an appropriate representation (JSON), along with the HTTP status `200 OK`.
  - The REST API authenticates the BAA and checks that it is authorized to read the specific behavioral profile from the database. Appropriate HTTP statuses are returned in case of authentication (e.g. `401 UNAUTHORIZED`) or authorization failures (e.g. `403 FORBIDDEN`).
  - The REST API returns appropriate HTTP statuses in other error cases (e.g. `400 BAD REQUEST` or `404 NOT FOUND`).

- **Update a Stored Behavioral Profile**
  - The BAA can call a REST method (PUT) in order to update a behavioral profile stored in the database, including the identifier of the resource to be updated.
  - Upon successful update of the resource, the REST API returns the HTTP status `200 OK`.
  - The REST API authenticates the BAA and checks that it is authorized to update the specific behavioral profile. Appropriate HTTP statuses are returned in case of authentication (e.g. `401 UNAUTHORIZED`) or authorization failures (e.g. `403 FORBIDDEN`).
  - The REST API returns appropriate HTTP statuses in other error cases (e.g. `404 NOT FOUND`).
- **Delete a Stored Behavioral Profile**
  - The BAA can call a REST method (DELETE) in order to delete a behavioral profile stored in the BPDB, including the identifier of the resource to be deleted.
  - Upon successful deletion of the resource, the REST API returns the HTTP status `200 OK`.
  - The REST API authenticates the BAA and checks that it is authorized to delete the specific behavioral profile. Appropriate HTTP statuses are returned in case of authentication (e.g. `401 UNAUTHORIZED`) or authorization failures (e.g. `403 FORBIDDEN`).
  - The REST API returns appropriate HTTP statuses in other error cases (e.g. `404 NOT FOUND`).

4.5.1 Weblog Profiles

The following attributes are stored for each weblog profile.

Name	Type	Index	Nullable	PK
id	Integer	true	false	true
user_id	String (20)	true	false	false
web	String (100)	false	false	false
dt	DateTime	false	false	false

The REST API for the mobility profiles includes the calls described in the following subsections.

4.5.1.1 Create new Weblog Profile

A POST request is sent to `../weblog/create`, which consumes a JSON message in the request body, like for example: `{ "userid":"test","web":"test","date":"22-11-2016 22:24:00"}`

All three keys (*userid, web, date*) are mandatory, so if anything is omitted a `400 BAD REQUEST` status will be returned as a response. The date value is of the form `(dd - MM - yyyy HH:mm:ss)`.

If the call is successful, a `201 CREATED` status is returned as a response, along with a LOCATION header similar to this one:

`../ weblog/create/5804810bfb3fef608a843946`, where *5804810bfb3fef608a843946* is the id of the newly created resource.

If a resource already exists for the given userid, a `409 CONFLICT` status is returned as a response.

4.5.1.2 Retrieve a Stored Weblog Profile

A GET request is sent to `../weblog/{userid}`, where *userid* is the userid of the resource to be retrieved.

If the call is successful, a `200 OK` status is returned as a response, along with the resource in JSON format in the response body, like for example: `{"id": "5804810bfb3fef608a843946", "userid": "test", "web": "test", "date": "22-11-2016 22:24:00"}`

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.1.3 Update a Stored Weblog Profile

A PUT request is sent to `../weblog/update/{id}`, where *id* is the id of the weblog resource to be updated, for example *5804810bfb3fef608a843946*.

The request consumes a JSON message in the request body like for example `{"userid":"12345","web":"newtest","date":"11-11-2016 09:25:00"}`. The *userid* key is not mandatory and will not be considered even if sent, since it should not be updated.

If the call is successful, a `200 OK` status is returned as a response and the resource is updated with the new web and date values.

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.1.4 Delete a Stored Weblog Profile

A DELETE request is sent to `../weblog/{id}`, where id is the id of the weblog resource to be deleted, for example *5804810bfb3fef608a843946*.

If the call is successful, a `200 OK` status is returned as a response and the resource is deleted from the database.

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.2 Mobility Profiles

The following attributes are stored for each mobility profile.

Name	Type	Index	Nullable	PK
id	Integer	true	false	true
user_id	String (20)	true	false	false
value	Integer	false	false	false
count	Integer	false	false	false

The REST API for the mobility profiles includes the calls described in the following subsections.

4.5.2.1 Create new Mobility Profile

A POST request is sent to `../mobility/create`, which consumes a JSON message in the request body, like for example: `{"userid": test", "value": 32, "count": 52}`

All three keys (*userid, value, count*) are mandatory, so if anything is omitted a `400 BAD REQUEST` status will be returned as a response.

If the call is successful, a `201 CREATED` status is returned as a response, along with a LOCATION header similar to this one:

`../mobility/create/5804810bfb3fef608a843946`, where *5804810bfb3fef608a843946* is the id of the newly created resource.

If a resource already exists for the given userid, a `409 CONFLICT` status is returned as a response.

4.5.2.2 Retrieve a Stored Mobility Profile

A GET request is sent to `../mobility/{userid}`, where *userid* is the userid of the resource to be retrieved.

If the call is successful, a `200 OK` status is returned as a response, along with the resource in JSON format in the response body, like for example: `{"id": "5804810bfb3fef608a843946", "userid": "test", "value": 32, "count": 52}`

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.2.3 Update a Stored Mobility Profile

A PUT request is sent to `../mobility/update/{id}`, where id is the id of the mobility resource to be updated, for example *5804810bfb3fef608a843946*.

The request consumes a JSON message in the request body like for example `{"userid":"12345", "count": 700, "value": 700}`. The *userid* key is not mandatory and will not be considered even if sent, since it should not be updated.

If the call is successful, a `200 OK` status is returned as a response and the resource is updated with the new value and count values.

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.2.4 Delete a Stored Mobility Profile

A DELETE request is sent to `../mobility/{id}`, where *id* is the id of the mobility resource to be deleted, for example *5804810bfb3fef608a843946*.

If the call is successful, a `200 OK` status is returned as a response and the resource is deleted from the database.

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.3 Gait Profiles

The following attributes are stored for each gait profile.

Name	Type	Index	Nullable	PK
id	Integer	true	false	true
user_id	String (20)	true	false	false
x_value	Integer	false	false	false
y_value	Integer	false	false	false
z_value	Integer	false	false	false

The REST API for the mobility profiles includes the calls described in the following subsections.

4.5.3.1 Create new Gait Profile

A POST request is sent to `../gait/create`, which consumes a JSON message in the request body, like for example: `{"userid": test", "x-value": 10, "y-value": 20, "z-value": 30}`

All keys (*userid, x-value, y-value, z-value*) are mandatory, so if anything is omitted a `400 BAD REQUEST` status will be returned as a response.

If the call is successful, a `201 CREATED` status is returned as a response, along with a LOCATION header similar to this one:

`../gait/create/5804810bfb3fef608a843946`, where *5804810bfb3fef608a843946* is the id of the newly created resource.

If a resource already exists for the given userid, a `409 CONFLICT` status is returned as a response.

4.5.3.2 Retrieve a Stored Gait Profile

A GET request is sent to `../gait/{userid}`, where *userid* is the userid of the resource to be retrieved.

If the call is successful, a `200 OK` status is returned as a response, along with the resource in JSON format in the response body, like for example: `{"id": "5804810bfb3fef608a843946", "userid": "test", "x-value": 10, "y-value": 20, "z-value": 30}`

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.3.3 Update a Stored Gait Profile

A PUT request is sent to `../gait/update/{id}`, where *id* is the id of the gait resource to be updated, for example *5804810bfb3fef608a843946*.

The request consumes a JSON message in the request body like for example `{"userid":"12345", "x-value": 10, "y-value": 20, "z-value": 30}`. The *userid* key is not mandatory and will not be considered even if sent, since it should not be updated.

If the call is successful, a `200 OK` status is returned as a response and the resource is updated with the new value and count values.

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.3.4 Delete a Stored Gait Profile

A DELETE request is sent to `../gait/{id}`, where *id* is the id of the gait resource to be deleted, for example *5804810bfb3fef608a843946*.

If the call is successful, a `200 OK` status is returned as a response and the resource is deleted from the database.

If the resource is not found, a `404 NOT FOUND` status is returned as a response.

4.5.4 Keystrokes Profiles

The Behavioral Profiles Database stores the three main data structures, regarding the keystrokes (b-verifier):

1. Training sets, per device / application / session, which include the following attributes:

Name	Type	Index	Nullable	PK
session	String	true	false	true
device_id	String (20)	true	false	true
app_id	String (20)	false	false	true
scale_factor	String (20)	false	false	false
density_class	String (20)	false	false	false
input_type	String (20)	false	false	false
orientation	String (20)	false	false	false
keydown_timestamp	Long	false	false	false
keyup_timestamp	Long	false	false	false
stroke_single_up_to_down	Integer	false	false	false
stroke_diff_down_to_down	Integer	false	false	false
stroke_diff_down_to_up	Integer	false	false	false
stroke_diff_up_to_down	Integer	false	false	false
stroke_diff_up_to_up	Integer	false	false	false

2. Signatures, per device and per application, which include the following attributes:

Name	Type	Index	Nullable	PK
device_id	String (20)	true	false	true
app_id	String (20)	true	false	true
date_time	Timestamp	false	false	false
stroke_single_up_to_down_avg	Double	false	false	false
stroke_diff_down_to_down_avg	Double	false	false	false
stroke_diff_down_to_up_avg	Double	false	false	false
stroke_diff_up_to_down_avg	Double	false	false	false
stroke_diff_up_to_up_avg	Double	false	false	false
stroke_single_up_to_down_std	Double	false	false	false
stroke_diff_down_to_down_std	Double	false	false	false
stroke_diff_down_to_up_std	Double	false	false	false

stroke_diff_up_to_down_std	Double	false	false	false
stroke_diff_up_to_up_std	Double	false	false	false

3. Verifications, per device and per application, which include the following attributes:

Name	Type	Index	Nullable	PK
device_id	String (20)	true	false	true
app_id	String (20)	true	false	true
date_time	Timestamp	false	false	true
score	Double	false	false	false
stroke_single_up_to_down_avg	Double	false	false	false
stroke_diff_down_to_down_avg	Double	false	false	false
stroke_diff_down_to_up_avg	Double	false	false	false
stroke_diff_up_to_down_avg	Double	false	false	false
stroke_diff_up_to_up_avg	Double	false	false	false
stroke_single_up_to_down_std	Double	false	false	false
stroke_diff_down_to_down_std	Double	false	false	false
stroke_diff_down_to_up_std	Double	false	false	false
stroke_diff_up_to_down_std	Double	false	false	false
stroke_diff_up_to_up_std	Double	false	false	false

The REST API for the mobility profiles includes the calls described in the following subsections.

4.5.4.1 Create new Training Set

A POST request is sent to `../bverifier/trainingSet/create`, which consumes a JSON message in the request body, like for example:

```
{"session": "test", "device_id": "5455", "app_id": "1244", "scale_factor":
"test", "density_class": "test", "input_type": "abc", "orientation":
"portrait", "keydown_timestamp": 1234567890, "keyup_timestamp": 1234567890,
"stroke_single_up_to_down": 10, "stroke_diff_down_to_down": 10,
"stroke_diff_down_to_up": 10, "stroke_diff_up_to_down", 10,
"stroke_diff_up_to_up": 10}
```

All keys are mandatory, so if anything is omitted a `400 BAD REQUEST` status will be returned as a response.

If the call is successful, a `201 CREATED` status is returned as a response

If a resource already exists for the given composite key (*session, device_id, app_id*), a `409 CONFLICT` status is returned as a response.

4.5.4.2 Retrieve Stored Training Sets

A GET request is sent to `../bverifier/trainingSet/{device_id}`, where *device_id* is the id of the device for which the respective training sets will be retrieved.

If the call is successful, a `200 OK` status is returned as a response, along with the resource in JSON format in the response body, like for example:

```
{["session": "test", "device_id": "5455", "app_id": "1244", "scale_factor":
"test", "density_class": "test", "input_type": "abc", "orientation":
"portrait", "keydown_timestamp": 1234567890, "keyup_timestamp": 1234567890,
"stroke_single_up_to_down": 10, "stroke_diff_down_to_down": 10,
"stroke_diff_down_to_up": 10, "stroke_diff_up_to_down", 10,
"stroke_diff_up_to_up": 10], […]}
```

If no training sets are found, a `404 NOT FOUND` status is returned as a response.

4.5.4.3 Delete Stored Training Sets

A DELETE request is sent to `../bverifier/trainingSet/{device_id}`, where *device_id* is the id of the device for which the respective training sets will be deleted.

If the call is successful, a `200 OK` status is returned as a response and the respective resources (all the training sets with the given *device_id*) are deleted from the database.

If no training sets are found, a `404 NOT FOUND` status is returned as a response.

4.5.4.4 Create new Signature

A POST request is sent to `../bverifier/signature/create`, which consumes a JSON message in the request body, like for example:

```
{"device_id": "5455", "app_id": "1244", "date_time": 2016-01-01 00:00:00,
"stroke_single_up_to_down_avg": 10, "stroke_diff_down_to_down_avg": 10,
"stroke_diff_down_to_up_avg": 10, "stroke_diff_up_to_down_avg", 10,
"stroke_diff_up_to_up_avg": 10, "stroke_single_up_to_down_std": 10,
"stroke_diff_down_to_down_std": 10, "stroke_diff_down_to_up_std": 10,
"stroke_diff_up_to_down_std", 10, "stroke_diff_up_to_up_std": 10}
```

All keys are mandatory, so if anything is omitted a `400 BAD REQUEST` status will be returned as a response.

If the call is successful, a `201 CREATED` status is returned as a response

If a resource already exists for the given composite key (*device_id, app_id, date_time*), a `409 CONFLICT` status is returned as a response.

4.5.4.5 Retrieve Stored Signatures

A GET request is sent to `../bverifier/signature/{device_id}`, where *device_id* is the id of the device for which the respective signatures will be retrieved.

If the call is successful, a `200 OK` status is returned as a response, along with the resource in JSON format in the response body, like for example:

```
{["device_id": "5455", "app_id": "1244", "date_time": 2016-01-01 00:00:00,
"stroke_single_up_to_down_avg": 10, "stroke_diff_down_to_down_avg": 10,
```

```
"stroke_diff_down_to_up_avg": 10, "stroke_diff_up_to_down_avg", 10,
"stroke_diff_up_to_up_avg": 10, "stroke_single_up_to_down_std": 10,
"stroke_diff_down_to_down_std": 10,  "stroke_diff_down_to_up_std": 10,
"stroke_diff_up_to_down_std", 10, "stroke_diff_up_to_up_std": 10], […]}
```

If no signatures are found, a `404 NOT FOUND` status is returned as a response.

4.5.4.6 Delete Stored Signatures

A DELETE request is sent to `../bverifier/signature/{device_id}`, where *device_id* is the id of the device for which the respective signatures will be deleted.

If the call is successful, a `200 OK` status is returned as a response and the respective resources (all the signatures with the given *device_id*) are deleted from the database.

If no signatures are found, a `404 NOT FOUND` status is returned as a response.

4.5.4.7 Create new Verification

A POST request is sent to `../bverifier/verification/create`, which consumes a JSON message in the request body, like for example:

```
{"device_id": "5455", "app_id": "1244", "date_time": 2016-01-01 00:00:00,
"score": 80, "stroke_single_up_to_down_avg": 10,
"stroke_diff_down_to_down_avg": 10,  "stroke_diff_down_to_up_avg": 10,
"stroke_diff_up_to_down_avg", 10, "stroke_diff_up_to_up_avg": 10,
"stroke_single_up_to_down_std": 10, "stroke_diff_down_to_down_std": 10,
"stroke_diff_down_to_up_std": 10, "stroke_diff_up_to_down_std", 10,
"stroke_diff_up_to_up_std": 10}
```

All keys are mandatory, so if anything is omitted a `400 BAD REQUEST` status will be returned as a response.

If the call is successful, a `201 CREATED` status is returned as a response

If a resource already exists for the given composite key (*device_id, app_id, date_time*), a `409 CONFLICT` status is returned as a response.

4.5.4.8 Retrieve Stored Verifications

A GET request is sent to `../bverifier/verification/{device_id}`, where *device_id* is the id of the device for which the respective verifications will be retrieved.

If the call is successful, a `200 OK` status is returned as a response, along with the resource in JSON format in the response body, like for example:

```
{["device_id": "5455", "app_id": "1244", "date_time": 2016-01-01 00:00:00,
"score": 80, "stroke_single_up_to_down_avg": 10,
"stroke_diff_down_to_down_avg": 10,  "stroke_diff_down_to_up_avg": 10,
"stroke_diff_up_to_down_avg", 10, "stroke_diff_up_to_up_avg": 10,
"stroke_single_up_to_down_std": 10, "stroke_diff_down_to_down_std": 10,
```

```
"stroke_diff_down_to_up_std": 10, "stroke_diff_up_to_down_std", 10,
"stroke_diff_up_to_up_std": 10], […]}
```

If no verifications are found, a `404 NOT FOUND` status is returned as a response.

4.5.4.9 *Delete Stored Verifications*

A DELETE request is sent to `../bverifier/verification/{device_id}`, where *device_id* is the id of the device for which the respective verifications will be deleted.

If the call is successful, a `200 OK` status is returned as a response and the respective resources (all the verifications with the given *device_id*) are deleted from the database.

If no verifications are found, a `404 NOT FOUND` status is returned as a response.

5 FIDO extensions for federated identities

5.1 Federated authentication

OpenID Connect (OIDC), the federation protocol, is an enhanced version of OAuth 2.0. OpenID Connect deals with the identity of an end-user by permitting OIDC Providers to verify the identity of an end-user based on the authentication performed by an Authentication Server (Identity Provider). In addition to that, it also allows OIDC clients to request identity attributes of an end-user from OpenID Connect Provider (OP) in a REST-like fashion.

OpenID Connect provides flexibility to OIDC clients to dictate how the authentication should be performed by OpenID Provider (OP), by explicitly specifying the assurance level (acr) or authentication methods (amr) in the authentication request. In the ReCRED project, the authentication context class reference (acr) and authentication methods references (amr) parameters of the OIDC protocol have been used to support FIDO authentication.

5.2 Implementation approach using OpenAM

The core of the solution for implementing federation is OpenAM. This is a web-based open-source solution that provides authentication, authorization, entitlement, and federation services. OpenAM out-of-the-box supports 25 authentication methods and offers the flexibility to create custom authentication modules based on the JAAS (Java Authentication and Authorization Service) open standard. In addition to that, OpenAM's federation services allow federated members to securely share identity information with each other.

This platform can be extended to allow authentication using custom modules. One such module has been created for the ReCRED project that authenticates a user against a FIDO server and then, if successful, returns a success status to the OpenAM server to continue with the authentication process. Details of the way in which the ForgeRock OpenAM server may be extended are available here: https://backstage.forgerock.com/docs/openam/13.5/dev-guide#sec-auth-spi.

In terms of functionality, the module first asks for a username and in response provides the FIDO endpoint. After that, it requests authentication ID from the FIDO client as a call-back. The authentication ID which module receives is then checked with the FIDO server to know the status of the FIDO authentication. If authentication succeeds, then OpenID Provider (OP) provides an authorization code to the OIDC client, which OIDC client can in turn use to request access_token and id_token from the OIDC provider.

Figure 20 depicts a sequence diagram showing the interactions between the Authentication module, client and the FIDO server:

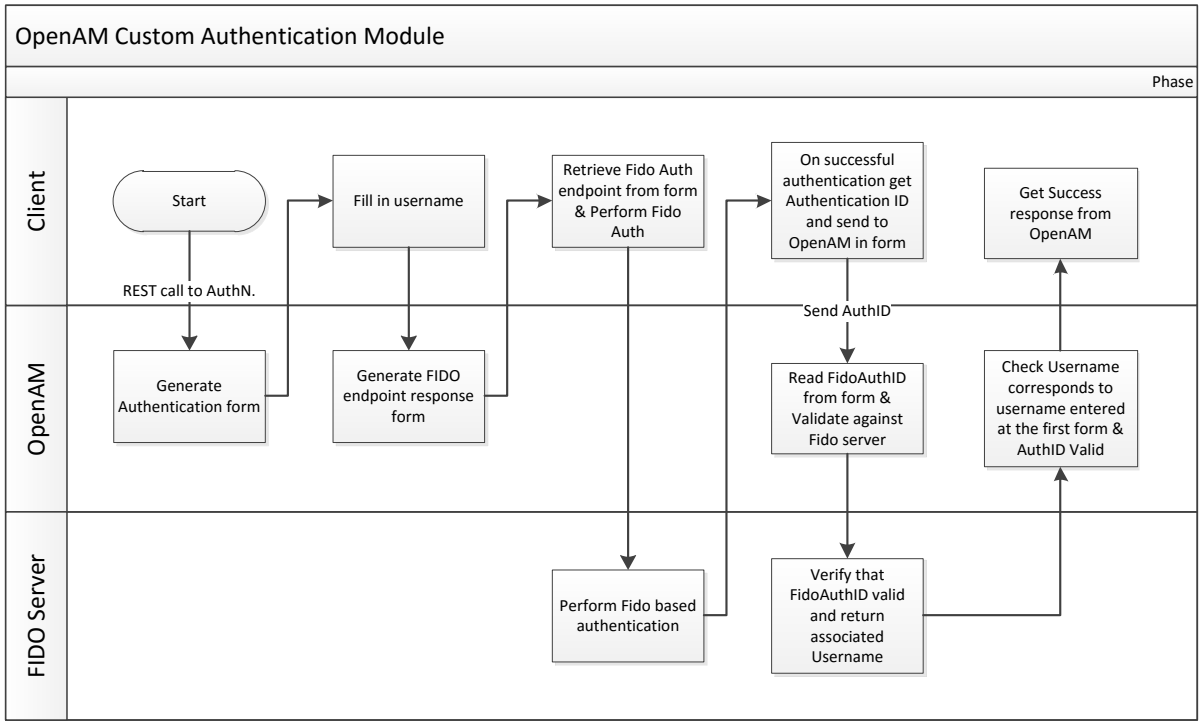

Figure 20 Interactions between Authentication module, client and FIDO server

The module that has been created has the following states:

- **STATE_BEGIN**
- **STATE_AUTH_BEGIN**
- **STATE_AUTH_RETURN**
- **STATE_ERROR**
- **LOGIN_SUCCESS**

The server side holds one of these states while the user is interacting with the platform (Figure 21). The first state, a dummy state for updating the values on the server and setting up state, quickly passes the user to STATE_AUTH_BEGIN which returns a JSON form for the client to complete that contains the user’s username. The handler for this event is called after the client has submitted the JSON form with the user’s username. If the username is valid (not blank), it moves to STATE_AUTH_RETURN. This presents the client application again with a JSON form that must be filled with the authentication ID that has been returned by the FIDO client application after authentication. This authentication response is then submitted in the JSON form back to the custom authentication module on the

OpenAM server. While in the STATE_AUTH_RETURN state, the server validates this Authentication response ID, with the user’s username and session information, against the FIDO server. If these values are correct, then the server moves to the state of LOGIN_SUCCESS. If at any stage during this process an error is encountered, then the server state moves to STATE_ERROR and the client application receives an error payload.

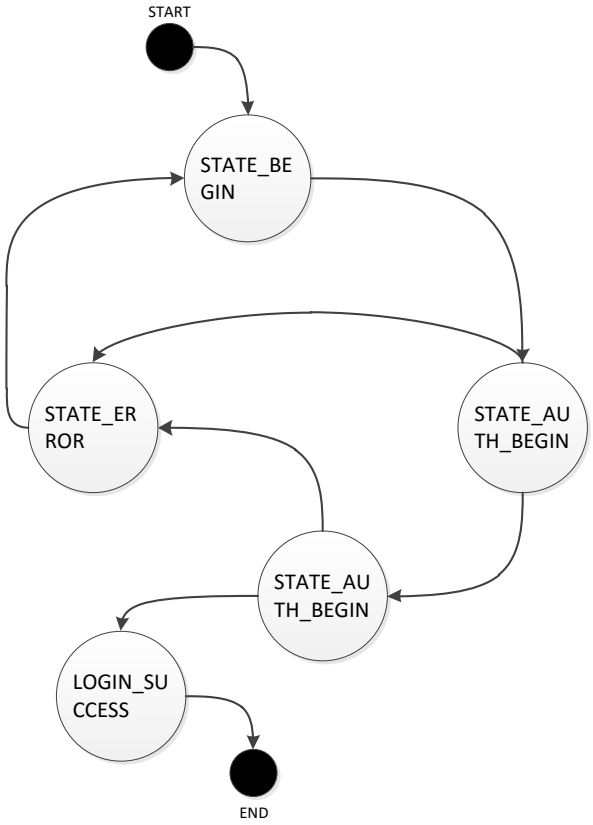

Figure 21 Authentication module states

To integrate this custom FIDO authentication module in OpenAM, an Ansible role has been created (Variables are environment dependent):

```
- name: Copy amAuthRecredFidoAuth.xml
  copy:
    src=amAuthRecredFidoAuth.xml
    dest=/home/{{ tomcat_user }}/amAuthRecredFidoAuth.xml
- name: Create service
  shell: "{{ frSSOAdmCmd }} create-svc -u amadmin -f {{ frOpenAMPasswordFile
}} -X /home/{{ tomcat_user }}/amAuthRecredFidoAuth.xml"
- name: Register module
  shell: "{{ frSSOAdmCmd }} register-auth-module -u amadmin -f {{
frOpenAMPasswordFile }} -a com.verizon.iam.openam.RecredFidoAuth"
- name: Copy recredfido-1.0-SNAPSHOT.jar
  copy:
    src=recredfido-1.0-SNAPSHOT.jar
    dest={{ vzSoftwareLocation }}/{{ tomcat_folder }}/webapps/openam/WEB-
INF/lib/
- name: Copy amAuthRecredFidoAuth.properties
  copy:
    src=amAuthRecredFidoAuth.properties
```

```
            dest={{ vzSoftwareLocation }}/{{ tomcat_folder }}/webapps/openam/WEB-
        INF/classes/
        - name: Copy RecredFidoAuth.xml
          copy:
            src=RecredFidoAuth.xml
            dest={{ vzSoftwareLocation }}/{{ tomcat_folder
        }}/webapps/openam/config/auth/default/
        - name: Create auth instance
          shell: "{{ frSSOAdmCmd }} create-auth-instance -m ReCredFido -t
        RecredFidoAuth -u amadmin -f {{ frOpenAMPasswordFile }} -e /"
        - name: Create auth cfg
          shell: "{{ frSSOAdmCmd }} create-auth-cfg -e / -m RecredFIDOnew -u
        amadmin -f {{ frOpenAMPasswordFile }}"
        - name: Update auth cfg entr
          shell: "{{ frSSOAdmCmd }} update-auth-cfg-entr -m RecredFIDOnew -e / -u
        amadmin -f {{ frOpenAMPasswordFile }} -a 'ReCredFido|REQUIRED'"
        - name: Stop Tomcat
          command: "nohup {{ vzSoftwareLocation }}/{{ tomcat_folder
        }}/bin/shutdown.sh"
          become: yes
          become_user: "{{ tomcat_user }}"
        - pause: seconds=50
        - name: Start Tomcat
          command: "nohup {{ vzSoftwareLocation }}/{{ tomcat_folder
        }}/bin/startup.sh"
          become: yes
          become_user: "{{ tomcat_user }}"
        - name: Wait for tomcat to start
          wait_for: port={{ http_port }}
               - pause: seconds=50
```

5.3 FIDO UAF Server functionalities for OpenAM

Apart from the standard implementation, FIDO UAF Server adds two specific functionalities for a better integration in OpenAM architectures. The two extensions have the role to obtain the authentication status for a specific user and to obtain the last authentication timestamp as well.

5.3.1 Last Authentication Time Extension

For a given username, the lastAuth endpoint returns the last authentication timestamp.

Endpoint: *GET /v1/lastAuth/{username}*

Returns: JSON Object

```
{
       "timestamp": long
}
```

If the user isn't authenticated the timestamp will have a value of -1.

```
@RequestMapping(value = "/lastAuth/{username}", method = RequestMethod.GET)
 public String lastAuth(@PathVariable(value = "username")String username) {
      return processAuxRequests.getLastAuth(username);
 }
public String getLastAuth(username) {
     JsonObject response = new JsonObject();
     try {
         response.addProperty("timestamp",
storageDao.readByUsername(username).timestamp;
     } catch (Exception e) {
         response.addProperty("timestamp", -1);
     }
```

```
     return response.toString();
}
```

The *getLastAuth()* method of the **ProcessAuxRequestsService** (eu.recred.fidouafsvc.service.impl) class tries to retrieve a registration record associated with the given username from the database, if the registration record doesn't exit an exception will be thrown and on that condition the timestamp will have a value of -1. If the record is found the authentication timestamp will be returned.

In the **AuthenticationResponse** class (eu.recred.fidouafsvc.ops) every time the user successfully authenticates with the server, the registration record associated with his username is updated with the current timestamp and updated in the database.

5.3.2 Authentication Id Extension

After every successful authentication, the server sends an *authenticationId* in the response. This *authenticationId* is used by OpenAM to verify with the server if the user is really authenticated and did not send a false claim.

Endpoint: *GET /v1/isAuthenticated/{authenticationId}*

Returns: JSON Object

```
{
       "authenticated": boolean
       "username": string
       "timestamp": long
}
```

If the authenticationId is not associated with any username the "authenticated" boolean will be false, which will indicate to the calling party to stop processing the object.

At every successful authentication the server will generate a random *authenticationId* (using the java.security.SecureRandom class) and store it with the associated username in the database. When the server receives the authentication id, it retrieves the username associated with it from the database and fetches the *RegistrationRecord* containing the authentication timestamp.

eu.recred.fidouafsvc.service.impl.ProcessAuxRequestsService

```
public getAuthenticated(String authenticationId) {
     JsonObject response = new JsonObject();
       if (authenticationId == null || authenticationId.isEmpty()) {
           response.addProperty("authenticated", false);
           return response.toString();
       }
       String username = storageDao.getAuthenticated(authenticationId);
       if (username != null) {
          try {
              response.addProperty("timestamp",
storageDao.getByUsername(username).timestamp);
          } catch (Exception e) {
             response.addProperty("authenticated", false);
             return response.toString();
         }
         response.addProperty("authenticated", true);
         response.addProperty("username", username);
    } else {
       response.addProperty("authenticated", false);
    }
```

```
    return response.toString();
}
```

6 FIDO extensions for ABAC and anonymous credentials

Privacy-Preserving Attribute-Based Access Control (PABAC) is emerging as a means for reliably authenticating users to services while preserving their privacy. Idemix and U-Prove are among the most well-known mechanisms, and are being integrated in the ReCRED architecture components mainly through the activities of WP5, WP4 and WP3.

One way of taking advantage of FIDO in the ABAC architecture is employing the FIDO protocols in the authentication phase between the User Device and the Issuer. Indeed, in this phase, the issuer needs to know the identity of the user in order to disclose a cryptographic credential with her attributes. Thus, using FIDO would allow for a reliable and password-less PABAC credential issuance. However, this would be far from bringing to the User-centric mobile device authentication world the advantages of PABAC.

A tighter integration of PABAC mechanisms in FIDO allows instead users to authenticate through their devices to Online Services while preserving their privacy, while online services can attract more users to their platforms.

We here propose an integration of PABAC in the FIDO UAF protocol. An architectural overview is provided in Figure 22.

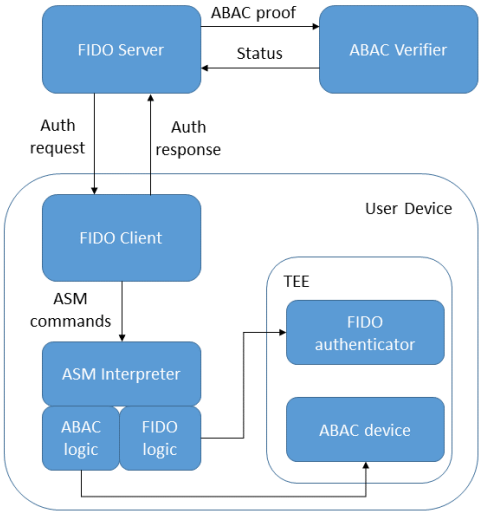

Figure 22 PABAC-FIDO Proposed Integrated Architecture

Figure 23 summarizes the FIDO UAF authentication process as defined by the FIDO specification [8]. Please note that the “RP Web App and Web Server” component referenced in the FIDO specification corresponds to the *Identity Provider* in the ReCRED architecture, and thus the following diagrams have been adapted to reflect this mapping.

For the integration that we are proposing, we assume that the PABAC credential issuance phase, which is not described here, has taken place before the authentication phase. Moreover, we assume that using a well-established special constant value for the username and public key triggers the PABAC-FIDO mechanism. This allows for the coexistence of the “normal” FIDO authentication mechanism along with the PABAC-FIDO “credential show” mechanism.

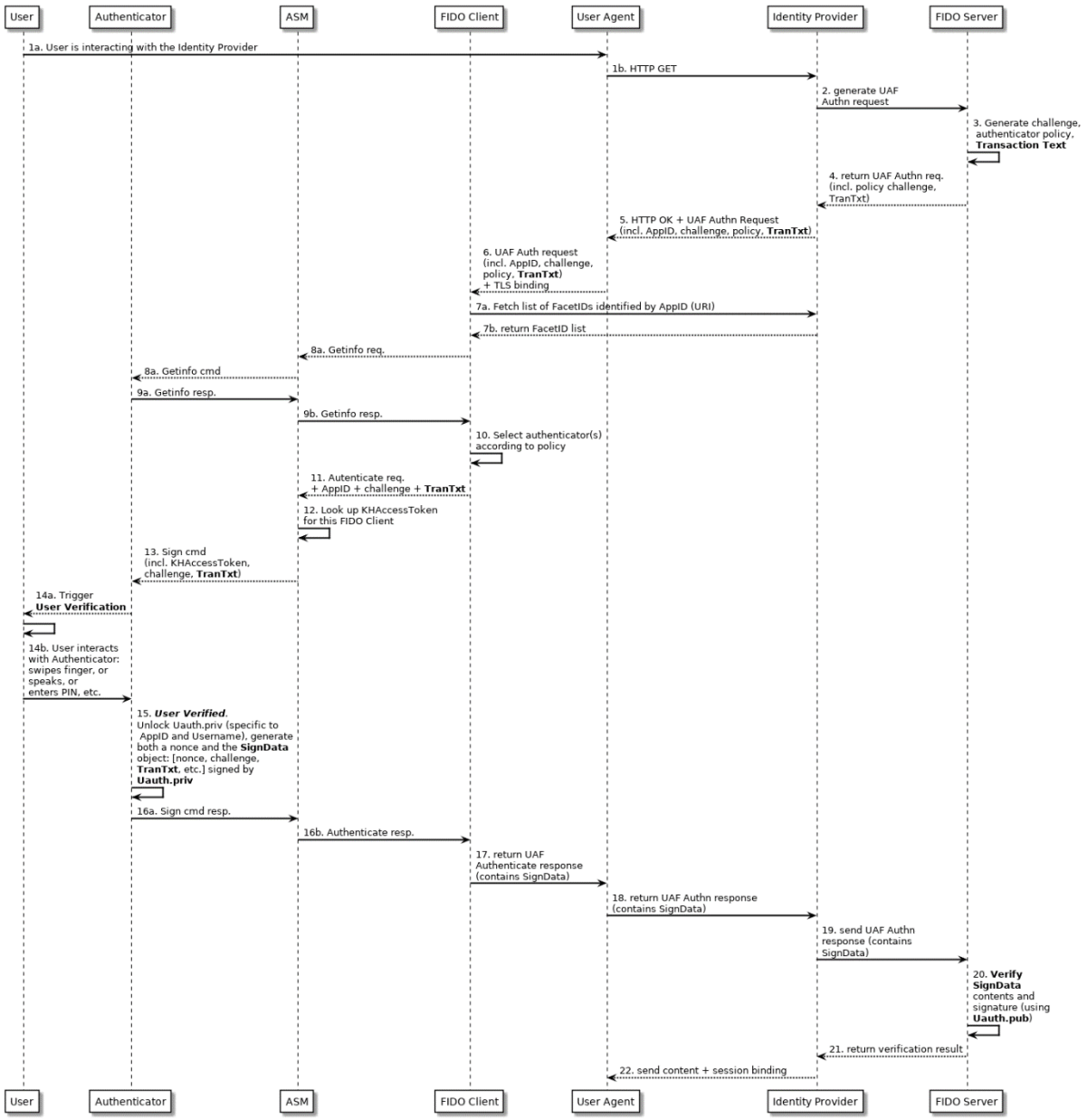

Figure 23 Authentication process from the FIDO UAF specification

Starting from the standard FIDO authentication mechanism depicted in the figure, we substitute the public-key based identification with a privacy-preserving attribute-based authentication: instead of the public-key cryptographic operations in steps 15 and 20, we employ a privacy-preserving attribute proving. Furthermore, the FIDO policy in steps 3, 4, 5, 6, 10 is replaced with a PABAC policy.

The modified protocol is depicted in Figure 24 below.

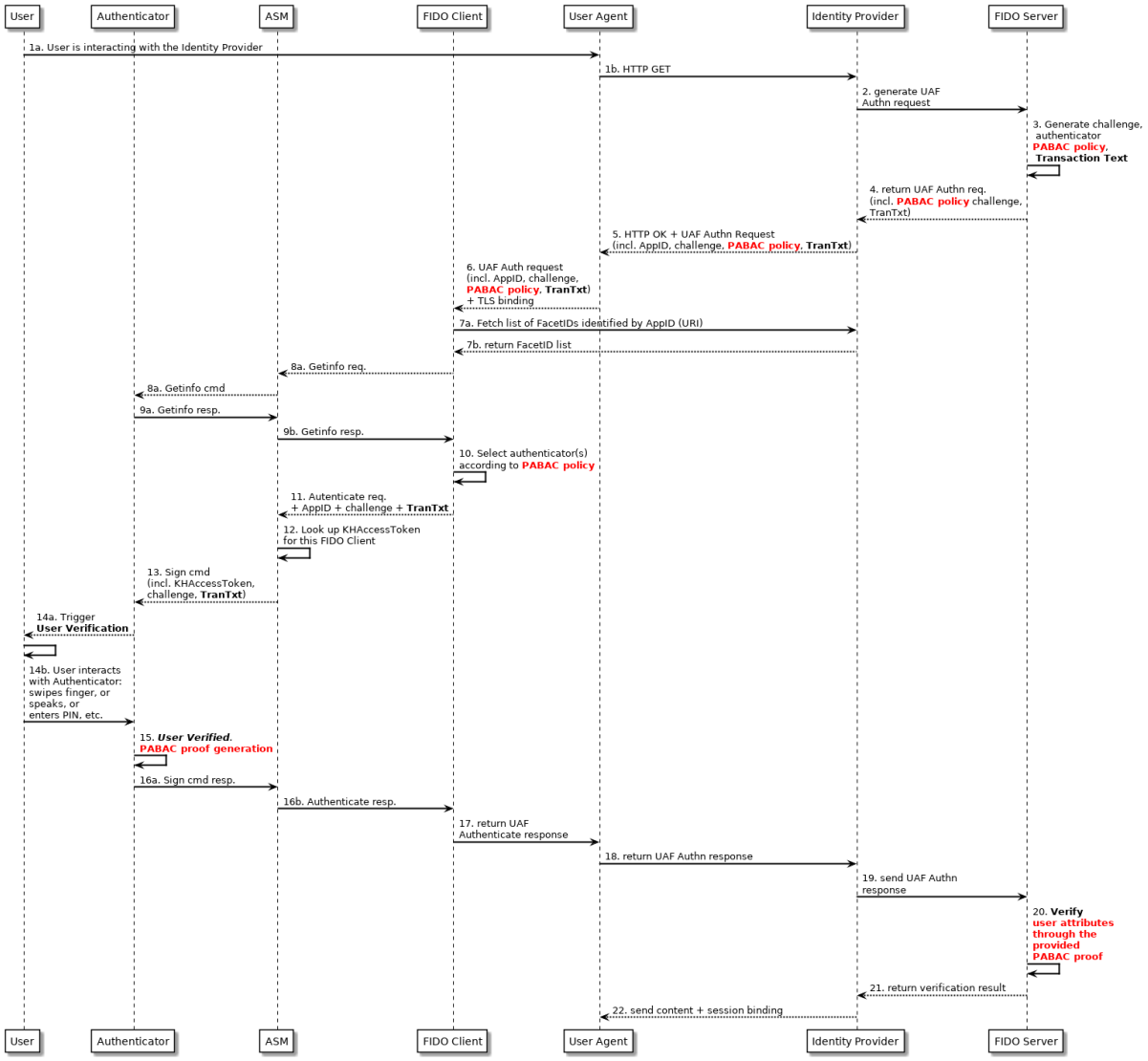

Figure 24 PABAC-FIDO integrated authentication protocol - proposed changes to the FIDO UAF specification are highlighted in red

When the user tries to access an unauthorized Web resource, the FIDO server replies with an `AuthenticationRequest` message as described in [8]. The FIDO server transmits to the client the required PABAC attributes through a FIDO extension as defined in the following data structures.

```
Dictionary AuthenticationRequest {
        required OperationHeader header;
        required ServerChallenge challenge;
        Transaction[ ] transaction;
        required Policy policy;
}
Dictionary Policy {
        required MatchCriteria[ ][ ] accepted;
        MatchCriteria disallowed;
}
Dictionary Extension {
        required DOMString id;          /* Bind to 'PABAC attribute' */
        required DOMString data;         /* Required attribute encoded as base64 */
        required boolean fail_if_unknown;     /* Bind to true */
}
```

After the required attributes reach the software authenticator module through FIDO ASM, the PABAC proof is generated and sent back to the server serialized in a FIDO extension which is encapsulated in the FIDO `AuthenticatorSignAssertion` structure.

```
Dictionary AuthenticatorSignAssertion {
              required DOMString assertionScheme;
              required DOMString assertion;
              Extension[ ] exts;         /*Serialized PABAC proof*/
}
```

The latter structure is embedded in the FIDO `AuthenticationResponse` dictionary which is processed by the server.

7 Open-TEE – An Open Virtual Trusted Execution Environment

7.1 Introduction

The Trusted Execution Environment (TEE) is an emerging technology that comes built in the latest mobile devices. It provides an isolated execution environment for executing Trusted Applications with elevated security mechanisms. TEE is separated from the Rich Execution Environment (REE) in which normal operating systems are running. By keeping confidential information and limiting its manipulations within the TEE, even if the REE is compromised, it is still hard for critical information breaches to occur on the TEE.

TEE was first defined by Open Mobile Terminal Platform as a set of hardware and software components which had to meet predefined security requirements. A major drawback of this first approach, was the absence of a publicly available specification and the limited interaction that developers could have with this technology. Typically, most of the mobile device vendors provided access to the API but the licenses were too expensive for regular users that just needed to experiment with this new technology. In July 2010, GlobalPlatform announced their own definition of the TEE and made the API publicly available. Since then, many implementations of the standardized TEE have been proposed that enable users to implement and execute applications which utilize the enhanced security features provided by TEE.

7.2 TEE & REE

7.2.1 Background

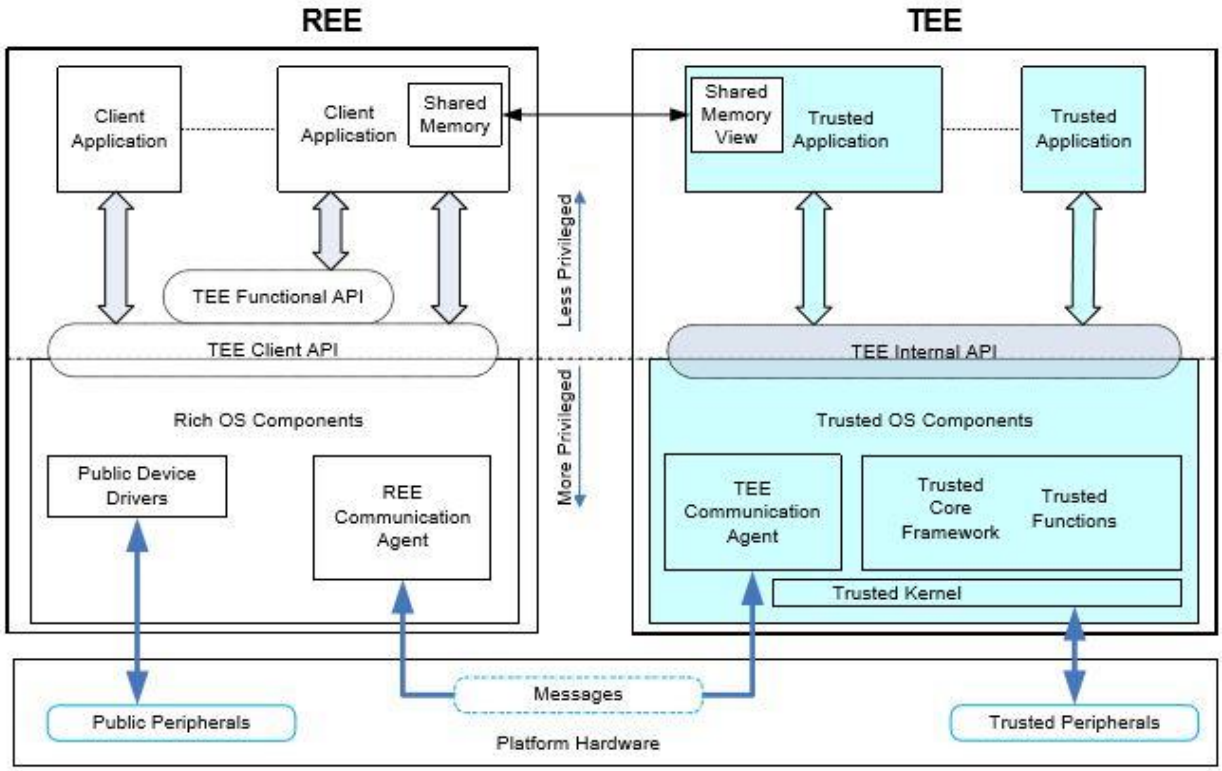

Figure 25 REE & TEE general architecture

As stated earlier, TEE is an isolated execution environment which offers the benefits of platform boot integrity, device identification/authentication, integrity protection and secure storage services for the REE. The REE represents a conventional operating system (REE OS) which normally provides rich features compared to the TEE OS. However, the REE is vulnerable to different kinds of attacks, such as malware. In order to protect sensitive private information against these attacks, it is considered best practice to keep this information safely in a separate container, in case the REE is compromised. The TEE provides the isolation for such a secure container. In addition, custom Trusted Applications (TA) can run within the TEE. The combination of secure storage and constrained operations on it defined by the TAs can keep the private information securely inside of the TEE, without exposing it to the REE.

7.2.2 TEE Implementations

The major hardware component that enables TEE in mobile devices is the ARM TrustZone [9]. As a feature, TrustZone can be seen as a special kind of virtualization of CPU state, memory, interrupt signals and I/O data with the purpose of isolation. In ReCRED, mobile devices that support the ARM TrustZone will benefit in terms of security, since the related ReCRED mobile application will leverage the security properties of TEE to isolate parts of the code and protect it from malicious software.

Another TEE hardware implementation which is worth mentioning is Intel Software Guard Extensions [10] (Intel SGX). The latter is a TEE technology, which allows an application, or its sub-component, to run inside an isolated execution environment, called an enclave. Intel SGX protects the enclave against any malicious software, including operating system, hypervisor, and low-level firmware which attempts to compromise its integrity or steal its secrecy. Intel SGX main goal is to protect workstations

(such as laptops, servers and desktops) that their hardware is based on an Intel CPUs. Thus, Intel SGX is the counterpart of ARM TrustZone for Workstations and its main intention is to secure cloud computing and Server side infrastructures.

Apart from hardware, a specific software implementation (i.e., OS, kernel, etc.) is required to enable TEE. Here we will focus on ARM based TEE software implementations, to enhance the security of the ReCRED mobile application. The existing ARM based TEE implementations can be categorized into three groups, according to their license type: (a) commercial, (b) open-source and (c) dual commercial/open-source. A list of the available TEE implementations, is presented below:

- Commercial implementations:
  - **Kinibi**, a commercial implementation from Trustonic [11]
  - **securiTEE**, a commercial implementation from Solacia [12]
- Open-source implementations:
  - **OP-TEE**, an open source implementation under BSD license maintained by Linaro [13]
  - **TLK**, an open-source implementation from NVIDIA under BSD license [14]
  - **T6**, and open-source implementation and research topic under GPL license [15]
  - **Open-TEE**, an open source implementation and research project from the University of Helsinki and sponsored by Intel. Provided under an Apache license [16].
- Dual commercial/open-source implementations:
  - **SierraTEE**, an implementation from Sierraware available both under commercial and GPL-licensing [17].

From the above implementations, we will focus more on **Open-TEE**, which is a software virtualization solution of a real Trusted Execution Environment (TEE) that has many benefits compared to the other TEE implementations. It emulates the TEE components and provides the entire GlobalPlatform TEE API functionality. The main goal of Open-TEE is to enable programmers during the early stages of the development process of Trusted Applications, to easily adopt the TEE functionality, regardless of the choice of the real TEE implementation. The Trusted Applications can later be compiled and run for any target TEE implementation that complies with the GlobalPlatform specifications. As its code is purely C, Open-TEE also provides a Java Native Interface which developers can use when developing mobile applications.

Open-TEE will be the Trusted Execution Environment of choice for ReCRED in order to enable the TEE code to be easily ported to any commercial or non-commercial TEE implementation. A list of the benefits of Open-TEE is presented below:

- Active support and development
  - Open TEE is supported by Intel Collaborative Research Institute for Secure Computing [19] and Secure Systems group [20]
- Software solution - Hardware Independent
  - Open-TEE does not need any specific hardware and run in any Linux environment as it's a "virtual" TEE.
- GlobalPlatform API compliant - Application can be compiled for any actual hardware TEE
  - Open-TEE is compliant with GlobalPlatform specifications
  - The Open-TEE Client and Trusted Application can be compiled with the specific flags of any commercial TEE.
- Runs in Linux and Android

  - As TEE mainly targets mobile devices in order to enhance the security, Open-TEE can run also on Android Devices. However, it's worth mentioning again that Open-TEE running on Android cannot be a substitute of a hardware-based TEE.
- Easy to use
  - Open-TEE sidesteps the complicated functionality of developing application in a hardware based TEE. With the virtual solution approach, it helps the developers easily develop and debug client and trusted applications.

Finally, it is worth mentioning that similarly to Open-TEE, there is also the Open-SGX platform, which provides an environment for developers to write applications that leverage the Intel SGX technology.

7.2.3 GlobalPlatform Specification

While there exist several proprietary TEE systems, GlobalPlatform [18] is working to standardize the TEE specification. Standardizing the TEE is crucial for a wide number of applications, such as mobile wallets, NFC payment implementations, premium content protection and bring your own device (BYOD) initiatives.

The following documents, which describe the GlobalPlatform TEE specification, are currently available on their website:

- **TEE Client API Specification v1.0** outlines the communication between applications running in a mobile OS and trusted applications residing in the TEE.
- **TEE Systems Architecture v1.0** explains the hardware and software architectures behind the TEE.
- **TEE Internal API Specification v1.0** specifies how to develop trusted applications.
- **TEE Secure Element API Specification v1.0** specifies the syntax and semantics of the TEE Secure Element API. It is suitable for software developers implementing trusted applications running inside the TEE which need to expose an externally visible interface to client applications.
- **Trusted User Interface API Specification v1.0** specifies how a trusted UI should facilitate information that will be securely configured by the end user and securely controlled by the TEE.
- **TEE TA Debug Specification v1.0** specifies the GlobalPlatform TEE debug interfaces and protocols.

The GlobalPlatform API is separated in two main components: the Client API and the Internal API. The Client API includes the functions that run in REE. The whole program running in REE is called Client Application (CA). Each Client Application is associated with a Trusted Application (TA) that is executed inside the TEE and makes use of Internal API.

7.3 Client API

7.3.1 Design Principles

The key design principles of the TEE Client API are presented below:

C language: C is the common denominator for practically all of the application frameworks and operating systems hosting Client Applications. Support for alternative language bindings – such as a Java API – may be considered in the future.

Blocking functions: Most Client Application developers are familiar with synchronous functions which block, waiting for the underlying task to complete, before returning to the calling code. An asynchronous interface is hard to design, hard to port in rich OS environments, and is generally difficult to use for developers familiar with synchronous APIs. In addition, it is assumed that multi-threading support is available on all target platforms; this is required for Implementations to support cancellation of blocking API functions.

Source-level portability: In Client API, there is no requirement on binary compatibility. Client Application developers will need only to recompile their code against an appropriate implementation-defined version of the TEE Client API headers in order to function correctly on that implementation.

Client-side memory allocations: Wherever possible, the design of the TEE Client API has placed the responsibility for memory allocation on the calling Client Application code. This gives the Client developer a choice on memory allocation location, enabling simple optimizations such as stack-based allocation or enhanced flexibility using placements in static global memory or thread-local storage.

Aim for zero-copy data transfer: The features of the TEE Client API are chosen to maximize the possibility of zero-copy data transfer between the Client Application and the Trusted Application, provided that the host operating system and hardware implementation can support it. This minimizes communications overhead and improves software efficiency, especially on cached processors where data copies are an expensive operation because of the cache pollution they cause. However, short messages can also be passed by copy, which avoids the overhead of sharing memory.

Support memory sharing by pointers: The TEE Client API will be used to implement higher-level APIs, such as cryptography or secure storage, where the caller will often provide memory buffers for input or output data using simple C pointers. The TEE Client API must allow efficient sharing of this type of memory, and as such does not rely on the Client Application being able to use bulk memory buffers allocated by the TEE Client API.

Specify only communication mechanisms: The Client API focuses on defining the underlying communications channel. It does define the format of the messages (type: *int, float, char*) which pass over the channel, or the protocols used by specific Trusted Applications.

7.3.2 Communication and Data Exchange Scheme between CA and TA

The main goal of the Client API is to define the communication and data exchange scheme between REE and TEE. The following operations are typically performed during the communication process:

1. Communication establishment between (REE) CA and TEE.
2. Communication establishment between CA and TA.
3. Shared Memory Registration (optional).
4. Invoke a Command optionally passing a shared memory reference and/or a message value.
5. Shared Memory Release (optional).
6. Communication termination between CA and TA.
7. Communication termination between (REE) CA and TEE.

Once the connection between CA and TA is successful, the Client Application can exchange data and request from the Trusted Application to perform specific actions, such as cryptographic and secure storage operations. The specific operations that the Trusted Application is able to perform will be

described in the Internal API specification in the next section. Upon successful execution of the requested operations, the connection between CA and TA is gracefully terminated.

Technically, the 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 remaining of this section describes the TEE structures and functions needed to perform a typical operation inside TEE.

7.3.2.1 TEE Client API data structures

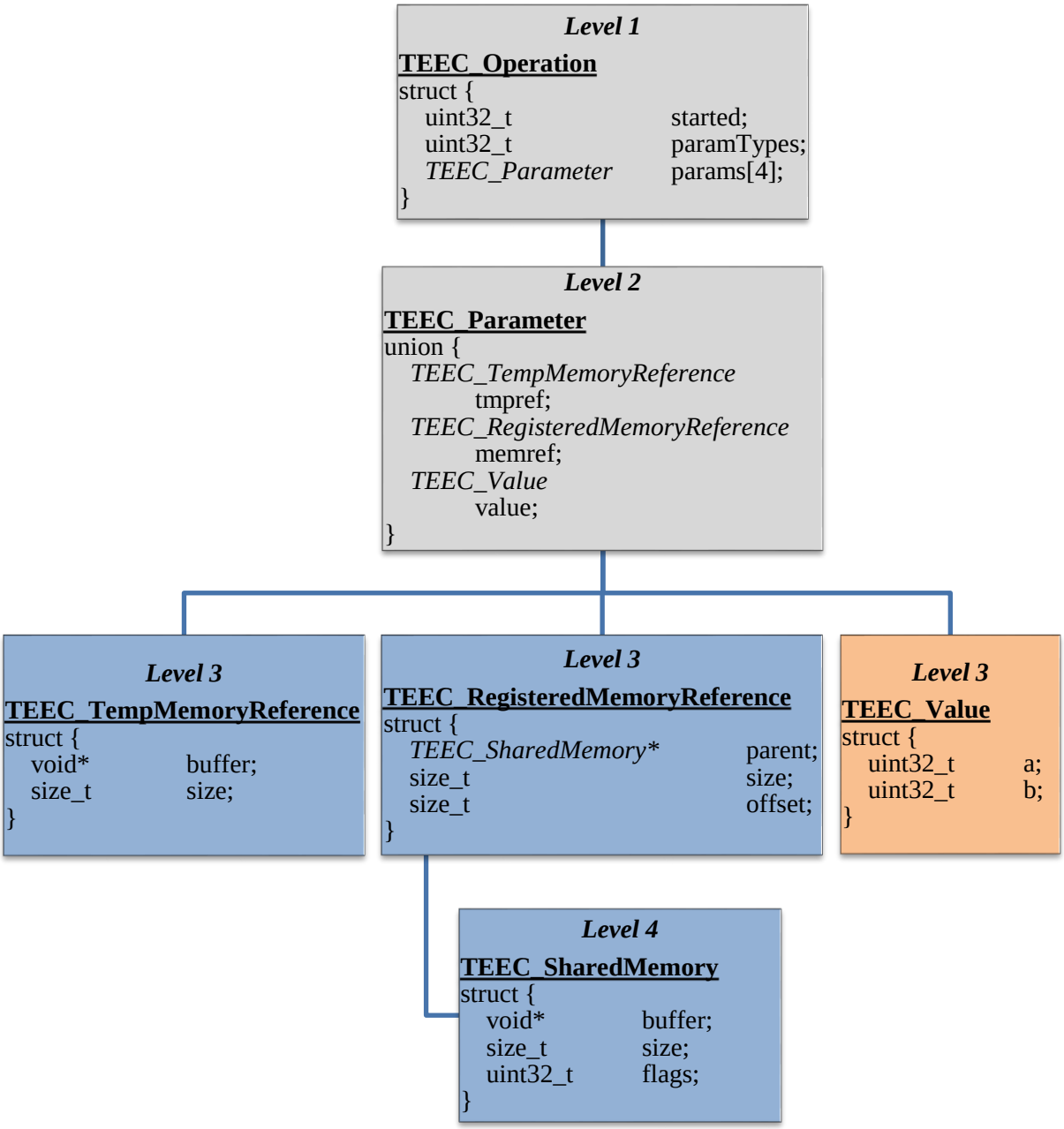

Figure 26 Hierarchical model of TEE Client API data structures

Data transfer between a CA and a TA may be accomplished by using the message passing option or the shared memory option. The suitable method should be selected, according to the size of the data that will be transferred: if the data is small enough (i.e. simple integer parameters) then message passing should be used, while for bigger data the shared memory approach should be considered for better performance.

1. Shared memory: a block of memory in a CA is shared with a TA to avoid memory copy back and forth. Android OS permits two different processes to share the same block of memory within the same application sandbox using a special Kernel feature called *ashmem*, in order minimize computational overhead. Since CA and TA are two Android processes and the data exchange between them is critical, TEE exploits this feature.

2. Message Passing: a pair of integer values is passed (copied) from a CA to a TA in the form of a message.

Figure 26 highlights the relationship between TEE data types. The appropriate data types for shared memory option are marked with blue color, while green color indicates those types that should be used for message passing. At the top level (Level 1) there is the *TEEC_Operation* data type that holds information about an operation. Each operation holds some parameters (*TEEC_Parameter*) (Level 2). These can be *TEEC_TempMemoryReference*, *TEEC_RegisteredMemoryReference* or *TEEC_Value* further analyzed in Level 3. Level 4 is the last level containing the *TEEC_SharedMemory* data type.

- `TEEC_Parameter`: the smallest unit for an operation. It can be either a *TEEC_TempMemoryReference*, *TEEC_RegisteredMemoryReference* or *TEEC_Value*.
- `TEEC_Value`: the simplest method to exchange a pair of integer values between a CA and a TA, without the need to register a shared memory. The value pair must be encapsulated into a parameter.
- `TEEC_TempMemoryReference`: the reference to a memory region which will be temporarily registered for the duration of the operation.
  `TEEC_RegisteredMemoryReference`: the reference to a registered shared memory. Once a shared memory is registered, in order to refer the shared memory to the TA, the CA must encapsulate it into a registered memory reference.

7.3.2.2 TEE Client API functions

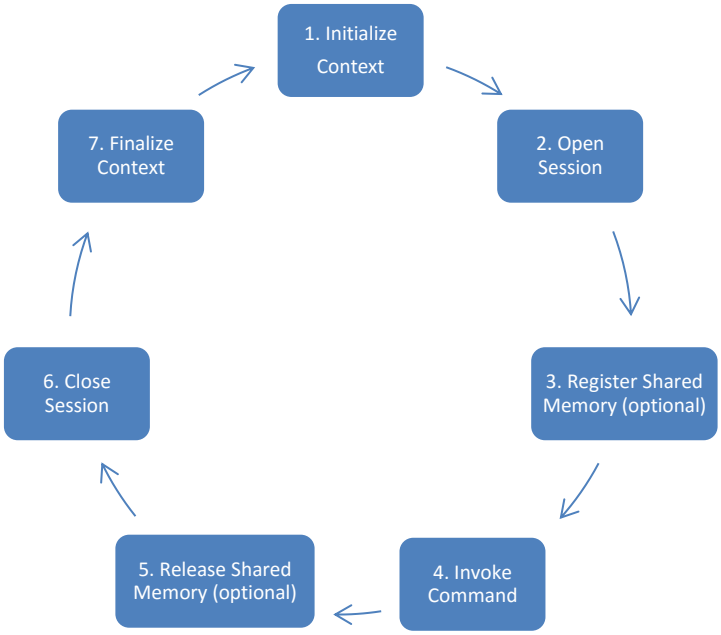

Figure 27 Communication and data exchange functions defined in TEE Client API

Figure 27 depicts the necessary steps that need to be performed for a typical TEE operation to be accomplished. These steps are explained below:

1. Initialize context

TEE context is the communication layer between the CA and a remote TEE; it does not refer to a specific TA. In order to establish a new TEE context, the *TEEC_InitializeConext* function should be used.

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

TEEC_InitializeContext function parameters

- `name`: a zero terminated string that describes the TEE to connect to. If this parameter is set to *NULL*, the default TEE is selected.
- `context`: a logical container linking a Client Application with a particular TEE. Its content is entirely implementation-defined

2. Open session

TEE session is an abstraction of the communication layer between a CA and a TA on top of a valid TEE context; thus, TEE session is only valid within a TEE context. Communications with a TA must occur using a valid TEE session. In order to establish a new TEE session, the *TEEC_OpenSession* function should be used. However, the CA must have enough information about the TA that it wants to interact, prior to establishing a TEE session. This information is mandatory for invoking the *TEEC_OpenSession* function.

```
TEEC_OpenSession (
TEEC_Context*    context,
```

```
TEEC_Session*     session,
const TEEC_UUID* destination,
uint32_t          connectionMethod,
const void*       connectionData,
TEEC_Operation*   operation,
uint32_t*         returnOrigin)
```

TEEC_OpenSession function parameters

- `context`: a pointer to an initialized TEE Context.
- `session`: a pointer to a Session structure to open.
- `destination`: a pointer to a structure containing the UUID of the destination Trusted Application.
- `connectionMethod`: the method of connection to use. It used to determine access control permissions to functionality provided by, or data stored by, the Trusted Application.
- `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 Trusted Application, or NULL if no Parameters are to be exchanged or if the operation cannot be cancelled.
- `returnOrigin`: a pointer to a variable which will contain the return origin. This field may be NULL if the return origin is not needed.

3. Register Shared Memory

Once all the data structures related to shared memory are defined, the Client Application can request to register a specific memory space to be used as Shared Memory. This is accomplished with the *TEEC_RegisterSharedMemory* function. The parameters of this function are a valid *TEE_Context* and the *TEEC_SharedMemory* data structure

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

TEEC_RegisterSharedMemory function parameters

`TEEC_SharedMemory`: the shared memory in the GlobalPlatform TEE Client API specification is a scheme to enable data transfer between a CA and a TA. Firstly, a block of memory within the CA must be registered as a shared memory against a specific TA. Then the TA can also operate on it. Notice that the shared memory is initiated by the CA, thus the actual memory space containing the shared memory belongs to the CA. Usually TAs have limited available memory space. Furthermore, it is dangerous to permit CAs to operate on the memory space of TAs from the perspective of security concerns. Hence, forcing the shared memory location to be owned by a CA is justified. The memory that is pre-allocated for the TAs is restricted for critical operations inside the TEE exclusively.

4. Invoke Command

Prior to invoking a command, a TA must pre-define a list of commands that a CA can invoke. Once a session to a target TA is initiated, the CA may invoke a command by specifying a command id which is

associated with a pre-agreed operation. For each different registered command, it is possible that there are input or output parameters involved (such as shared memory buffers). The input and output parameters are encapsulated in the *TEEC_Operation* structure. A valid *TEEC_Context* and *TEEC_Session* must be passed to this API call along with the command id and an optional *TEEC_Operation* structure.

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

5. Release Shared Memory

When a registered shared memory structure is no longer needed, the memory space should be released. This may be accomplished by using the *TEEC_ReleaseSharedMemory*, function which releases the *TEEC_SharedMemory* data structure, passed as an argument.

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

6. Close session

When no further communication is needed with the TA, the matching session should be closed to release the used resources. This task may be carried out by the *TEEC_CloseSession* function in the GlobalPlatform TEE Client API, which receives a single input argument *TEEC_Session* (the session to be closed). There is no return value to indicate the result of the close session operation. Thus, after calling this function, the *TEEC_Session* should not be reused.

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

7. Finalize a context

If there is no need to further interact with the TEE, all shared resources must be released and the context must be finalized. In GlobalPlatform TEE Client API, the *TEEC_FinializeContext* function must be called by providing a valid context handle initialized previously

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

7.3.3 Communication Sequence between REE and TEE

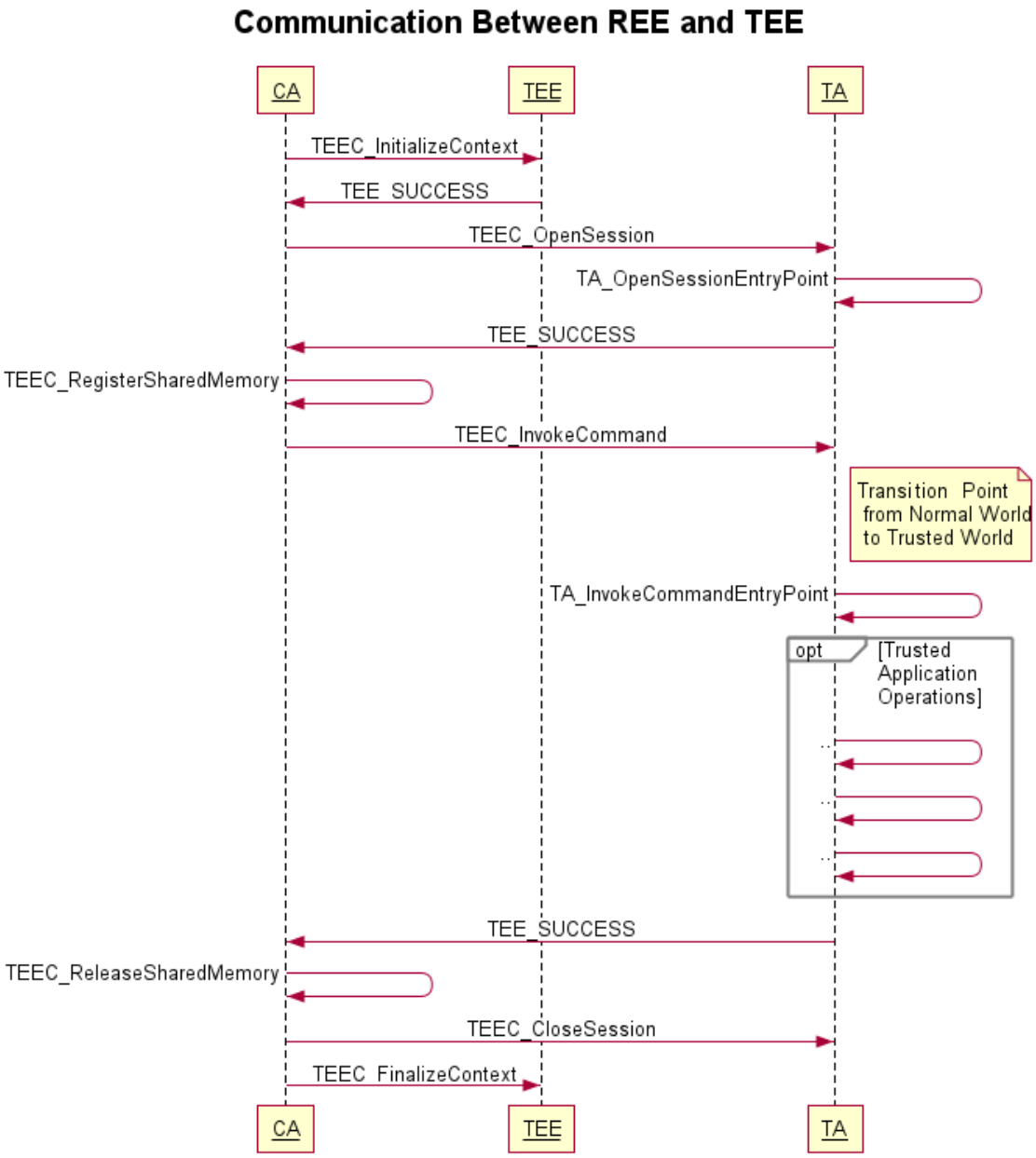

Figure 28 Communication sequence between REE and TEE

Figure 28 depicts the communication scheme between the REE (CA) and TEE (TA) including the GlobalPlatform Client API function calls and some GlobalPlatform Internal API Calls.

1) **TEEC_InitializeContext**: The CA initiates a context with the TEE.
2) When the new context has successfully opened the *TEEC_SUCCESS* message is returned to the CA.
3) **TEEC_OpenSession**: The CA opens a Session with the corresponding TA
4) **TA_OpenSessionEntryPoint**: When the *TEEC_OpenSession* is called from the REE, the Function TA_OpenSessionEntryPoint is executed in the TEE, to notify the TA that a new client is connecting.

5) When the new session has successfully opened the *TEEC_SUCCESS* message is returned.
6) **TEEC_RegisterSharedMemory**: After the successful connection establishment with the TEE (*TEEC_Context*) and the Trusted Application (*TEEC_Session*) the shared memory space is allocated. This action is CA driven as mentioned before.
7) **TEEC_InvokeCommand**: The CA invokes a command to be executed in the TA. At this point the program flow changes from Normal World to Trusted World.
8) **TA_InvokeCommandEntryPoint**: When this function is called, the TA is informed that a command has been invoked by the CA. Then, it executes the trusted function calls such as key generation, cryptographic function, secure storage operation and critical arithmetic calculations that are supported be the TEE GlobalPlatform internal API specifications. The detailed analysis of the supported functions will be presented in the next chapter. When a trusted operation has finished successfully, a *TEE_SUCCESS* message is return to the CA. In different case an error message is returned. The Trusted Application also makes use of the allocated shared memory that is accessible from both CA and TA.
9) **TEEC_ReleaseSharedMemory**: After the completion of the trusted operation in the secure world (TEE) the program flow switches back to normal world (REE). The pre-allocated registered shared memory is released.
10) **TEEC_CloseSession**: The CA closes the opened session with the TA
11) **TEEC_FinalizeContext**: The CA closes the context with the TEE.

7.4 Internal API

The TEE GlobalPlatform Internal API supports Trusted Storage and a wide range of cryptographic and arithmetic operations that can be implemented inside the Trusted Application. These operations are executed in the Trusted World, taking advantage of the elevated security mechanisms that the TEE offers.

7.4.1 Trusted Storage API for Data Objects and Keys

The first group of functions that the GlobalPlatform TEE Internal API supports is associated with the Trusted Storage feature. This storage may be accessed only from within the TEE and can't be accessed by any Client Application. The size in MB of the trusted storage is vendor dependent. Each one pre-allocates a protected memory space where only the TEE can operate.

The Trusted Storage supports securely storing Transient Objects and Persistent Objects. Transient Objects are held in memory and are automatically wiped and reclaimed when they are closed or when the TA instance is destroyed. On the other hand, Persistent Objects are persisted in TEE Trusted Storage even after disconnecting CA from the TEE. A Persistent Object can contain a Cryptographic Key Object, a Cryptographic Key-Pair Object or a user defined Data Object. On the other hand, a Transient Object can contain only a Cryptographic Key Object or a Cryptographic Key-Pair Object.

The hierarchy of the Persistent Trusted Storage is presented in Figure 29. Starting from the outer circle:

- The TEE can have many Trusted Applications.
- Each TA has access to a set of Trusted Storage Spaces.
- A Trusted Storage Space contains Persistent Objects which are manipulated through opaque Object Handles.
- This handle points to a Cryptographic Key Object, a Cryptographic Key-Pair Object, or a Data Object.

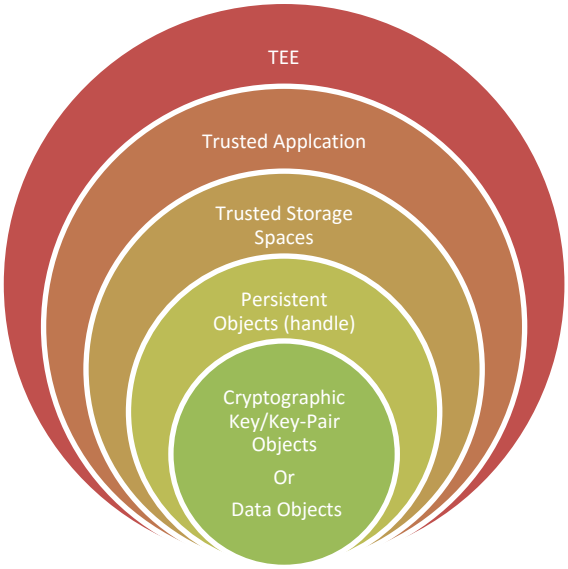

Figure 29 Hierarchy of the Persistent Trusted Storage

Each Persistent Object has a type, which precisely defines the contents of the object. For example, there are object types for AES keys, RSA key-pairs, data objects, etc. All persistent objects have an associated Data Stream. Data objects have only a data stream. Persistent cryptographic key/key-pair objects have a data stream, Object Attributes, and metadata. Transient Objects contain only attributes and no data stream:

- **Data Stream:** contains the actual data of the Persistent Object. It can be loaded into a buffer (in the TA memory space) or accessed as a stream.
- **Object Attributes:** used to store the key material in a structured way. For example, an RSA key-pair has an attribute for the modulus, the public exponent, the private exponent, etc. used for storing small amount of data (typically a few tens or hundreds of bytes).
- **Metadata:** includes information about a cryptographic key: (a) key size in bits and (b) key usage flags which define the operations permitted with the key as well as whether the sensitive parts of the key material can be retrieved by the TA or not.

7.4.1.1 Persistent Objects

User defined data and cryptographic keys can be persisted into the Persistent Trusted Storage using the *TEE_CreatePersistentObject* function. Later, in order to access this data, the *TEE_ReadObjectData* function may be used. When the data have been read, the Persistent Object should be closed using the *TEE_CloseObject* to free any used resources. Figure 30 depicts the process of creating, reading and closing Persistent Data Objects.

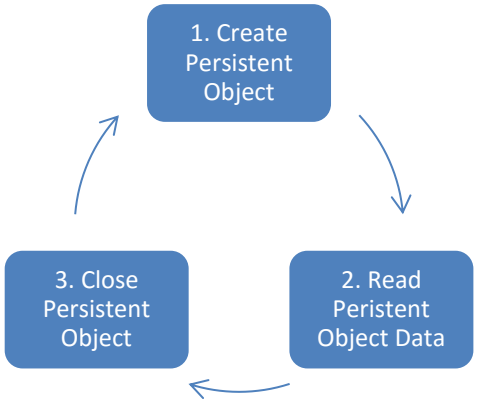

Figure 30 Basic steps for creating, reading and closing Persistent Data Objects

1. Create Persistent Object

```
TEE_Result TEE_CreatePersistentObject(
uint32_t  storageID,
[in(objectIDLength]  void* objectID,
size_t objectIDLen,
uint32_t  flags,
TEE_ObjectHandle  attributes,
inbuf]  void*  initialData,
size_t initialDataLen,
[out]  TEE_ObjectHandle*  object);
```

Parameters

- `storageID`: The storage to use. It must be TEE_STORAGE_PRIVATE
- `objectID`, `objectIDLen`: The object identifier. Note that this cannot reside in shared memory.
- `flags`: The flags which determine the settings under which the object is opened
- `attributes`: A handle on a transient object from which to take the persistent object attributes. Can be *TEE_HANDLE_NULL* if the persistent object contains no attribute, for example if it is a pure data object.
- `initialData`, `initialDataLen`: The initial data content of the persistent object
- `object`: A pointer to the handle, which contains the opened handle upon successful completion. If this function fails for any reason, the value pointed to by object is set to TEE_HANDLE_NULL. When the object handle is no longer required, it must be closed using a call to the *TEE_CloseObject* function.

2. Read Object Data

```
TEE_ReadObjectData(
TEE_ObjectHandle  object,
[out] void* buffer,
size_t size,
[out] uint32_t* count)
```

Parameters

- `object`: The object handle.
- `buffer`: A pointer to the memory which, upon successful completion, contains the bytes read.
- `size`: The number of bytes to read.

- `count`: A pointer to the variable which upon successful completion contains the number of bytes read.

3. Close Object

```
TEE_CloseObject( TEE_ObjectHandle object)
```

Parameters

- `object`: Handle on the object to close. If set to TEE_HANDLE_NULL, does nothing

Persistent Objects are persisted in TEE Trusted Storage even after disconnecting a CA from the TEE. In order to gain access to a previously created Persistent Object, the *TEE_OpenPersistentObject* function should be used. Then the contents of the object may be read by using the aforementioned *TEE_ReadObjectData* function. The *TEE_CloseAndDeletePersistentObject* function closes and permanently deletes the Persistent Object from the TEE Trusted Storage.

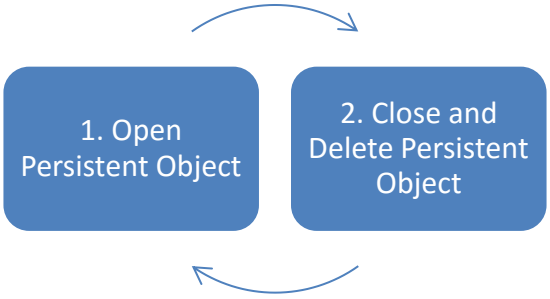

Figure 31 Open and delete a persistent object

4. Open Persistent Object

```
TEE_OpenPersistentObject(
uint32_t  storageID,
[in(objectIDLength)] void* objectID,
size_t objectIDLen,
uint32_t  flags,
[out] TEE_ObjectHandle* object);
```

Parameters

- `storageID`: The storage to use. It must be TEE_STORAGE_PRIVATE.
- `objectID`, `objectIDLen`: The object identifier. Note that this buffer cannot reside in shared memory.
- `flags`: The flags which determine the settings under which the object is opened.
- `object`: A pointer to the handle, which contains the opened handle upon successful completion. If this function fails for any reason, the value pointed to by object is set to *TEE_HANDLE_NULL*. When the object handle is no longer needed, it must be closed using a call to the *TEE_CloseObject* function.

5. Close and Delete Persistent Object

```
TEE_CloseAndDeletePersistentObject( TEE_ObjectHandle object );
```

Parameters

- `object`: Handle on the object to close. If set to *TEE_HANDLE_NULL*, does nothing.

7.4.1.2 Transient Objects and Cryptographic Key Objects

A Transient Object can be used to hold a cryptographic object (key or key-pair). The first step towards creating a Transient Object, is to allocate the memory required for storing the Object, using the *TEE_AllocateTransientObject* function. The uninitialized Transient Object can then be populated with data by utilizing the *TEE_PopulateTransientObject* function. Typically, most of the times, this function is called internally by another cryptographic function that generates the appropriate Cryptographic Key. The allocated resources may be cleared by calling the *TEE_FreeTransientObject* function.

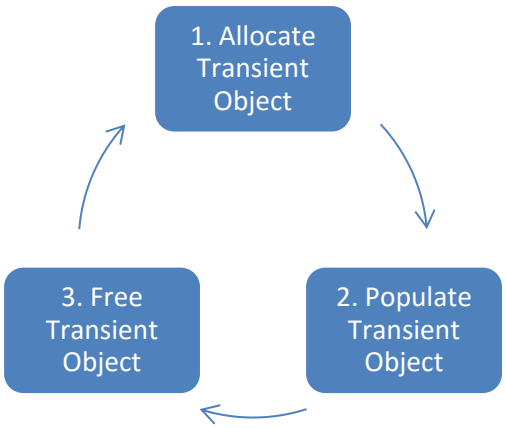

Figure 32 Workflow for Allocating, Populating and Freeing a Transient Object

1. Allocate Transient Object

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

Parameters

- `objectType`: Type of uninitialized object container to be created.
- `maxObjectSize`: Size of the object. The interpretation of this parameter depends on the object type and is defined in Table 5-7 above.
- `object`: Filled with a handle on the newly created key container.

The *objectType* parameter may have one of the values presented in Table 1.

TEE_TYPE_AES
TEE_TYPE_DES
TEE_TYPE_DES3
TEE_TYPE_HMAC_MD5
TEE_TYPE_HMAC_SHA1
TEE_TYPE_HMAC_SHA224
TEE_TYPE_HMAC_SHA256
TEE_TYPE_HMAC_SHA384
TEE_TYPE_HMAC_SHA512
TEE_TYPE_RSA_PUBLIC_KEY
TEE_TYPE_RSA_KEYPAIR
TEE_TYPE_DSA_PUBLIC_KEY
TEE_TYPE_DSA_KEYPAIR
TEE_TYPE_DH_KEYPAIR
TEE_TYPE_GENERIC_SECRET

Table 1 List of key types supported by TEE Internal API

2. Populate Transient Object

```
TEE_PopulateTransientObject(
TEE_ObjectHandle objectType,
[in] TEE_Attribute* attrs,
uint32_t attrCount
)
```

Parameters

- `object`: Handle on an already created transient and uninitialized object.
- `attrs`, `attrCount`: Array of object attributes.

3. Free Transient Object

```
TEE_FreeTransientObject(
TEE_ObjectHandle object
)
```

Parameters

- `object`: Handle on the object to free.

A major feature of Trusted Storage API is the support for securely generating cryptographic keys. A cryptographic key can be a Persistent Object or a Transient Object based on the needs of the application. Cryptographic keys are initiated as Transient Objects and can be alternated to Persistent later. Prior to generating a new cryptographic key, a new Transient Object should be allocated as a placeholder for the key. The key generation process is carried out by the *TEE_GenerateKey* API function of the GlobalPlatform Internal API. This function generates random cryptographic keys and securely stores them in a persistent object.

4. Generate Key

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

Parameters

- `object`: Handle on an uninitialized transient key to populate with the generated key
- `keySize`: Requested key size. Must be less than or equal to the maximum size of the object container.
- `params`, `paramCount`: Parameters for the key generation.

Once the key generation process is completed, the cryptographic key can be used in the appropriate cryptographic operation presented in the next section.

7.4.2 TEE Cryptographic Operations

Cryptographic operations are the most important feature of the GlobalPlatform TEE Internal API. It supports Digests, Symmetric ciphers, Message Authentication Codes (MACs), Authenticated Encryption (AE), Asymmetric Encryption Schemes, Asymmetric Signature Schemes and Key Exchange Algorithms. These functions can be used by any Trusted Application.

The Trusted Application can perform the supported cryptographic operations using the provided API. First, an operation is allocated using the *TEE_AllocateOperation* function, where the operation mode is defined (encryption, decryption). With *TEE_SetOperationKey* function the previously generated key is bound with the appropriate operation. Then, the execution of the cryptographic operation occurs, using the appropriate cryptographic function. Figure 33 depicts the workflow of an encryption operation example, using the *TEE_AssymetricSignDigest* cryptographic function.

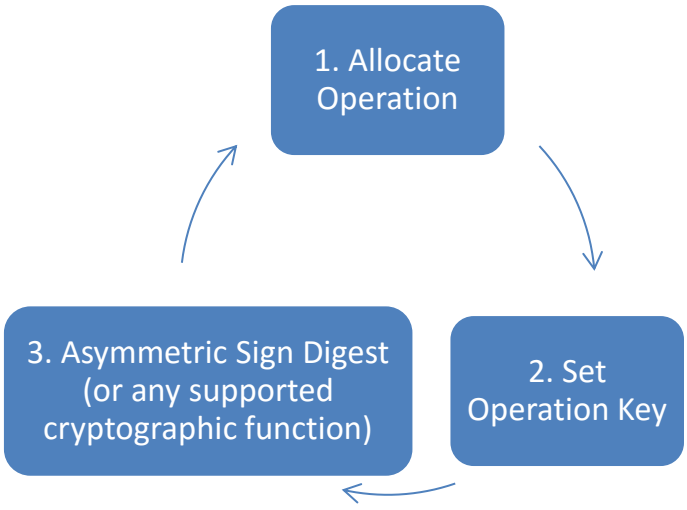

Figure 33 Workflow of performing an indicative cryptographic operation

The *TEE_AssymetricSignDigest* function signs a message digest within an asymmetric operation. A list of the supported encryption algorithms is presented in Table 2.

Digests	MD5 SHA-1 SHA-256 SHA-224 SHA-384 SHA-512
Symmetric ciphers	DES Triple-DES with double-length and triple-length keys AES
Message Authentication Codes (MACs)	DES-MAC AES-MAC AES-CMAC HMAC with one of the supported digests
Authenticated Encryption (AE)	AES-CCM with support for Additional Authenticated Data (AAD) AES-GCM with support for Additional Authenticated Data (AAD)
Asymmetric Encryption Schemes	RSA PKCS1-V1.5 RSA OAEP
Asymmetric Signature Schemes	DSA RSA PKCS1-V1.5 RSA PSS
Key Exchange Algorithms	Diffie-Hellman

Table 2 Encryption algorithms supported in TEE Internal API

1. Allocate Operation

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

Parameters

- `operation`: Reference to generated operation handle.
- `algorithm`: One of the cipher algorithms enumerated in previous section.
- `mode`: The operation mode.
- `maxKeySize`: Maximum key size in bits for the operation.

2. Set Operation Key

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

Parameters

- `operation`: Operation handle.

- `key`: A handle on a key object

3. Asymmetric Sign Digest

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

Parameters

- `operation`: Handle on the operation, which must have been suitably set up with an operation key
- `params`, `paramCount`: Optional operation parameters
- `digest`, `digestLen`: Input buffer containing the input message digest
- `signature`, `signatureLen`: Output buffer written with the signature of the digest

7.4.3 TEE Arithmetic Operations

All asymmetric cryptographic functions are implemented by using arithmetical functions, where operands are typically elements of finite fields or in mathematical structures containing finite field elements. The Cryptographic Operations API should hide the complexity of the mathematics that is behind these operations. A developer who needs some cryptographic service does not need to know anything about the internal implementation.

However, in practice developers may face the following difficulties: the API does not support the desired algorithm; or the API supports the algorithm, but with the wrong encodings, options, etc. The purpose of the TEE Arithmetical API is to provide building blocks so that the developer can implement missing asymmetric cryptographic algorithms. In other words, the arithmetical API can be used to implement a plug-in into the Cryptographic Operations API. Allowing the possibility of expanding the Cryptographic Operations API means that some of its functions can be left as optional to implement.

The Arithmetic Operations API specification mandates that all functions work with input and output *TEE_BigInt* data types, which are within the interval [-2^M+1, 2^M-1], where (M ≥ 2048) is an implementation-defined number of bits.

In Table 3, a limited list of the supported operations is presented, while the full description of them is beyond the scope of this document.

Category	Function	Description
Basic	TEE_BigIntAdd	Returns the summary of two TEE_BigInt.
	TEE_BigIntSub	Returns the subtraction of two TEE_BigInt.
	TEE_BigIntNeg	Returns the negative of a TEE_BigInt.
	TEE_BigIntMul	Returns the multiplication of two TEE_BigInt.
	TEE_BigIntSquare	Returns the square of a TEE_BigInt.
	TEE_BigIntDiv	Returns the integer part of the division between two TEE_BigInt.
Logical	TEE_BigIntCmp	Compares two TEE_BigInt.

	TEE_BigIntShiftRight	Returns the result of |BigInt| >> bits operation.
	TEE_BigIntGetBit	Returns an exact bit of the natural binary representation of a TEE_BigInt.
	TEE_BigIntGetBitCount	Returns the magnitude of a TEE_BigInt.
Modular	TEE_BigIntMod	Returns the integer remaining of the division between two TEE_BigInt.
	TEE_BigIntAddMod	Returns the integer remaining of the division between two TEE_BigInt, added to a third TEE_BigInt.
	TEE_BigIntSubMod	Returns the integer remaining of the division between two TEE_BigInt, subtracted to a third TEE_BigInt.
	TEE_BigIntMulMod	Returns the integer remaining of the division between two TEE_BigInt, multiplied to a third TEE_BigInt.
	TEE_BigIntSquareMod	Returns the integer remaining of the division between two TEE_BigInt, multiplied to the first TEE_BigInt.
Other	TEE_BigIntRelativePrime	Determines whether the greatest common divisor between two TEE_BigInt is equal to 1.
	TEE_BigIntIsProbablePrime	Performs a probabilistic primality test on a TEE_BigInt.

Table 3 Arithmetic Operations supported in TEE Internal API

7.5 API usage & examples

For better understanding of the TEE Client and Internal APIs, several examples will be presented in this section. The first example performs encryption using an asymmetric operation. The second example performs secure storage.

7.5.1 Key Generation and Digital Signature

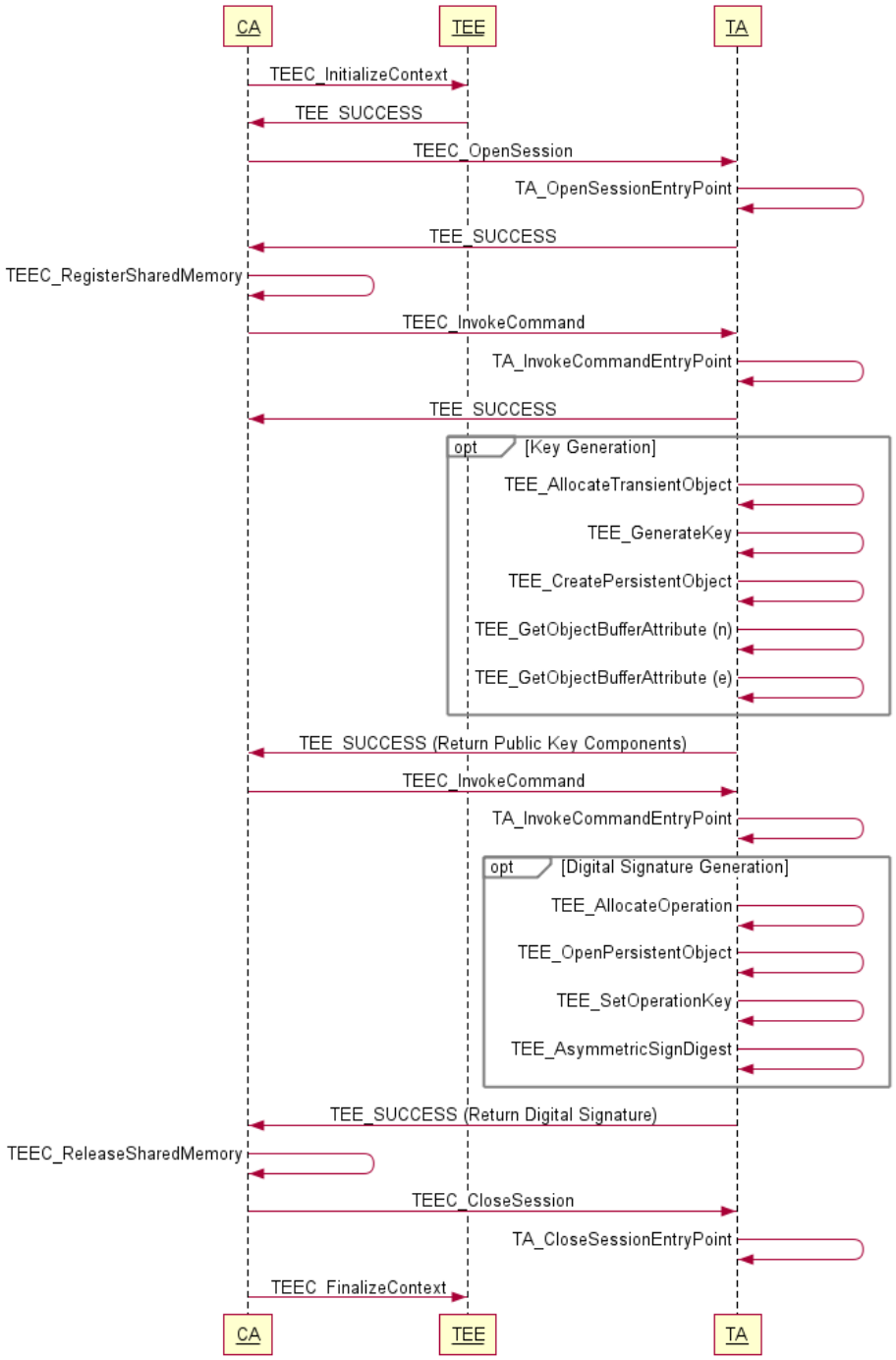

Figure 34 Asymmetric encryption example

In this example (Figure 34), the use of Key Generation and Digital Signature Encryption features is presented. The goal is to (a) generate a new RSA key pair and (b) create a digital signature by signing the contents of a buffer with the private key. Simultaneously with the key generation, the components of the public key (modulus, public exponent) are passed to the Client Application through shared buffer so anyone can validate the signature. The communication API functions have been presented in the previous chapters, thus won’t be further analyzed in detail here.

Key Generation

The CA is executed in the Normal World. First, it establishes a connection with the TEE and presents the two available operations that the user can choose from: (1) the generation of an RSA key pair and (2) the generation of a digital signature using the previously generated key. Figure 35 depicts the main menu of the application.

Figure 35 Application main menu

The user chooses the generation of a digital signature (1). At this point, the program flow switches from REE to TEE (Trusted World), executing the operations depicted in Figure 36 using the GlobalPlatform Internal API.

Figure 36 Operations performed for generating a digital signature key

7.5.2 RSA Key Pair Generation

The RSA Key Pair Generation consists of the following GP Internal API functions.

1. *TEE_AllocateTransietObject*
2. *TEE_GenerateKey*
3. *TEE_CreatePersistentObject*
4. *TEE_GetObjectBufferAttribute*
5. *TEE_CloseObject*

7.5.2.1 TEE_ AllocateTransientObject

The following example code allocates memory for a Transient Object that will hold a RSA Key pair of 1024-bit size. The complete list of the supported key types is presented in Table 1.

ReCRED Key Generation Code

```
TEE_ObjectHandle rsa_keypair;
size_t key_size2 = 1024;
TEE_AllocateTransientObject(TEE_TYPE_RSA_KEYPAIR, key_size2, &rsa_keypair);
```

7.5.2.2 TEE_GenerateKey

Once the Transient Object has been allocated uninitialized, the actual key generation is handled by the *TEE_GenerateKey* function of the GlobalPlatform Internal API.

ReCRED Key Generation Code

```
TEE_GenerateKey(rsa_keypair, key_size2, NULL, 0);
```

7.5.2.3 TEE_CreatePersistentObject

The generated RSA Key pair is currently held in a Transient Object but should be stored in a Persistent Object for later use. Persistent Objects reside in the TEE secure storage. An example of a Persistent Object creation is presented later in the Secure Storage example.

7.5.2.4 TEE_GetObjectBufferAttribute

Once the Key generation process has been completed, the Object consists of the following attributes.

1. TEE_ATTR_RSA_MODULUS
2. TEE_ATTR_RSA_PUBLIC_EXPONENT
3. TEE_ATTR_RSA_PRIVATE_EXPONENT
4. TEE_ATTR_RSA_PRIME1
5. TEE_ATTR_RSA_PRIME2
6. TEE_ATTR_RSA_EXPONENT1
7. TEE_ATTR_RSA_EXPONENT2
8. TEE_ATTR_RSA_COEFFICIEN

In order to export the Public Key components, the *TEE_GetObjectBufferAttribute* function should be used, as shown below.

ReCRED Key Generation Code

```
TEE_GetObjectBufferAttribute(RSA_keypair_object,
TEE_ATTR_RSA_MODULUS, n, &n_len);

TEE_GetObjectBufferAttribute(RSA_keypair_object,
TEE_ATTR_RSA_PUBLIC_EXPONENT, e, &e_len)

exp_res = params[1].memref.buffer;
memset(exp_res, '\0', params[1].memref.size);
memcpy(exp_res, n,512);

exp_res = params[2].memref.buffer;
memset(exp_res, '\0', params[2].memref.size);
memcpy(exp_res, e, 512);
```

The modulus and the public exponent are exported from the key pair and are passed to the shared memory location (*params[1].memref.buffer, params[2].memref.buffer*) that can be accessed from the CA. The value of the generated public key is depicted in Figure 37.

```
TEEC_InvokeCommand
Transition ---> Secure World !!!
Operation Successful
Transition ---> Normal World !!!
 Modulus - n / Public key: c2d6174ac7bcfe0a5f84884eaa76ce9c3e0415178886da815dc1b62a0fe7d5c4ed93a27e
6ecfc351288009c2381730563b36dfe01ec79bba4c8678c663d2a710697f3790dda18af951a843aa31b868285f59a6293d0
3e91fa9b0ea38c846fff4ef6e19c529ccaced2b5a985e89501b1bd718f9a99a999e7befa278ae3fb7dded
Public Exponent - e : 010001
```

Figure 37 Generated Public RSA Key

7.5.3 Digital Signature

Once the key generation process has completed successfully, the newly generated key can be used to create digital signatures. This time the user chooses the second option from the main menu. The generated digital signature is depicted in Figure 38.

```
============================================================
Select one of the operations below:
Generate RSA Key Pair  = 1
Generate Digital Signature  = 2
============================================================
2
Digital Signature Operation
TEEC_InvokeCommand
Transition ---> Secure World !!!
Operation Successful
Transition ---> Normal World !!!
Digital Signature: 70e8566cc7f2d639dfbe4033fcb9103b8155238815bfc663025df3a395511f00b9e2d1b7d2a2c50b
03a6dd9e86c4be5796a0d6bf4be924cae732ab24b08a4b3d56dd773d49b61526ad8d70e4ab1d6b87820c813c9de23f29e38
6c334c1d906420ce5cd954dab7554226a747896e1d5260b976c3aabe5f9f9b40410b2a971e4ec
```

Figure 38 The generated digital signature in hex representation

For the digital signature generation process, the GlobalPlatform Internal API functions depicted in Figure 39 are executed.

Figure 39 Internal API functions executed for the digital signature generation process

1. TEE_AllocateOperation
2. TEE_OpenPersistentObject
3. TEE_SetOperationKey
4. TEE_AssymetricSignDigest

7.5.3.1 TEE_AllocateOperation

The *TEE_AllocateOperation* function allocates a handle for a new cryptographic operation and sets the mode and algorithm type. The PKCS1_v1.5 RSA protocol suite is used with a 1024-bits RSA key

ReCRED Digital Signature Code

```
TEE_OperationHandle handle = (TEE_OperationHandle)NULL;

TEE_AllocateOperation(&handle,
TEE_ALG_RSAES_PKCS1_V1_5,
TEE_MODE_SIGN,
1024);
```

7.5.3.2 TEE_OpenPersistentObject

During the Key generation process, the RSA key was stored inside TEE Secure Storage using the *TEE_CreatePersistentObject* function. In order to use this cryptographic key, the Persistent Object’s attributes should be retrieved. This is accomplished using the *TEE_OpenPersistentObject* function which opens a handle on an existing Persistent Object. This handle can be used to access the Persistent Object’s attributes and data stream.

ReCRED Digital Signature Code

```
char rsa_keypair_id[] = "rsa_keypair_object_id1";
static TEE_ObjectHandle RSA_keypair_object;

TEE_OpenPersistentObject(TEE_STORAGE_PRIVATE,
rsa_keypair_id,
sizeof(rsa_keypair_id),
flags_open,
&RSA_keypair_object);
```

7.5.3.3 TEE_SetOperationKey

This function is highly related with the *TEE_AllocateOperation* function. In the current step, the previously allocated cryptographic key for the operation is set. These two functions are typically used together.

ReCRED Digital Signature Code

```
TEE_SetOperationKey(handle, RSA_keypair_object);
```

7.5.3.4 TEE_AssymetricSignDigest

The contents of the buffer are signed with the private key in order to generate the actual digital signature. The application signs the contents of the shared memory (*params[2].memref.buffer*) and copies the digital signature result to the shared memory (*params[3].memref.buffer*), in order for the CA to have access.

ReCRED Digital Signature Code

```
TEE_AsymmetricSignDigest (
handle,
NULL,
0,
params[2].memref.buffer,
params[2].memref.size,
params[3].memref.buffer,
(uint32_t *)&params[3].memref.size);
```

7.5.4 Secure Storage

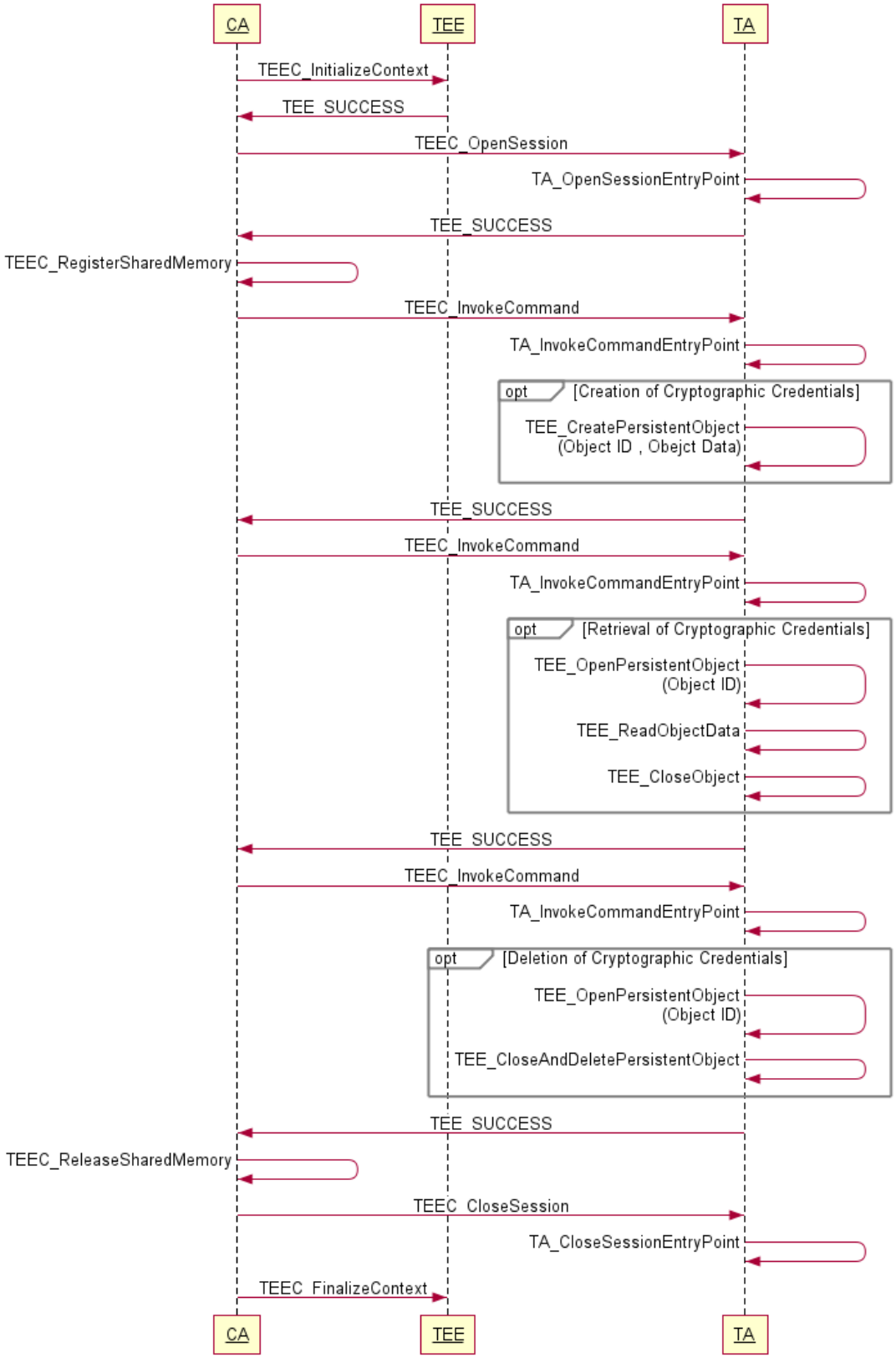

Figure 40 Secure Storage example

The following example (Figure 40) demonstrates the Secure Storage capabilities that the TEE offers. The basic communication steps followed in this example are the same as in the previous example and thus will not be analyzed further.

Step 1

The CA is executed in the Normal World. First, it performs a connection to the TEE and displays the three available operations that the user can choose from: (1) the creation, (2) the retrieval and (3) the deletion of a Persistent Object. Figure 41 depicts the main menu of Trusted Storage Application.

Figure 41 Trusted Storage Application main menu

Step 2

As this is the first execution of the secure storage ReCRED application, the first action should be to create a persistent object. So the user chooses the first operation.

After that, the user is prompted (Figure 42) to insert the Object ID which is a uniqe identifier of the Persistent Object. In ReCRED application, the Object IDs are the users’ cryptographic credentials.

Figure 42 Secure Storage Application prompts for the Object ID

The CA presents all the available objects that are stored inside the TEE. The user is prompted to enter the Object Data. In this example, the user inserts *"password1234@"* in the Object Data field (Figure 43).

Figure 43 The user inserts custom data in the Object Data field

The Object ID and Object Data that the user inserted, are placed in the shared memory location and can be accessed from the CA and the TA.

Step 3

At this point, the *TEEC_InvokeCommand* function is executed from the CA and the program flow switches to Trusted World (TEE), informing the TA to securely store the object (Figure 44).

Figure 44 Application execution switches from Normal World to Trusted World

Figure 45 depicts the GlobalPlatform Internal API functions that are called from the Trusted Application side. The remaining of this chapter further analyzes these function calls.

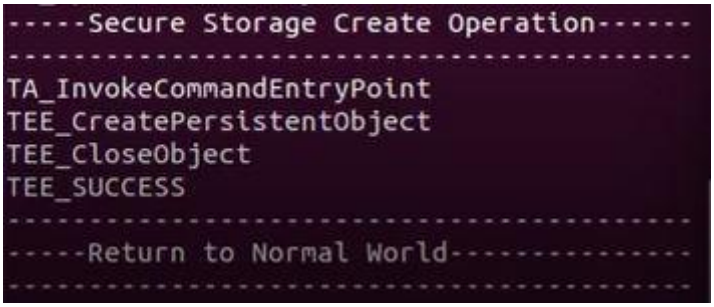

Figure 45 Internal API functions executed for the Secure Storage process

7.5.4.1 TEE_CreatePersistentObject

ReCRED Secure Storage Code

```
TEE_ObjectHandle persistent_object = (TEE_ObjectHandle)NULL;
uint32_t objectID = params[1].value.a;
uint32_t objectIDsize = sizeof(objectID);
uint32_t flags_create = TEE_DATA_FLAG_ACCESS_WRITE_META |
TEE_DATA_FLAG_ACCESS_WRITE | TEE_DATA_FLAG_ACCESS_WRITE |
TEE_DATA_FLAG_OVERWRITE;
```

```
TEE_CreatePersistentObject(TEE_STORAGE_PRIVATE,
&objectID,
objectIDsize,
flags_create,
NULL,
params[0].memref.buffer,
256,
&persistent_object);

TEE_CloseObject(persistent_object);
```

Step 4

Once the Persistent Object (Object ID and Object Data) has been created and stored inside TEE Secure Storage, the user has two options: either to retrieve the Object from TEE or to delete the previously inserted Object. When the user chooses to retrieve an object (selection 2 from the menu) the complete list of Object IDs is displayed (Figure 46). Then the user inserts the object ID that needs to be retrieved. In this example the Object ID five (5) is selected, as depicted in Figure 46.

```
============================================================
Select one of the operations below:
Create   Persistent Object   = 1
Retrieve Persistent Object   = 2
Delete   Persistent Object   = 3
============================================================
2
Retrieve Operation
Available Objects Presented by Object ID
5
6
0
0
0
Insert Object ID
5
```

Figure 46 The application displays the list of Object IDs and prompts the user to insert her selection

Step 5

At this point the *TEEC_InvokeCommand* is executed from the CA and the program flow switches to Trusted World (TEE), informing the TA to securely store the new inserted object (Figure 47).

```
TEEC_InvokeCommand
Retrieving Object ID and Object Data from Secure World
Transition ---> Secure World !!!
Operation Successful
Transition ---> Normal World !!!
Object ID  5
Object Data : password1234@
```

Figure 47 The application switches to Trusted World and displays the contents of the Persistent Object

Figure 48 displays the GlobalPlatform Internal API functions that are called from the Trusted Application side. These functions have already been presented in the previous chapter. Then, the Object ID and Object Data are returned to the CA through the shared memory feature.

1. TEE_OpenPersistentObject
2. TEE_ReadObjectData
3. TEE_CloseObject

```
----------------------------------------
----Secure Storage Retrieve Operation-----
----------------------------------------
TA_InvokeCommandEntryPoint
TEE_OpenPersistentObject
TEE_ReadObjectData
TEE_CloseObject
TEE_SUCCESS
----------------------------------------
-----Return to Normal World--------------
----------------------------------------
```

Figure 48 Internal API functions executed for retrieving a Persistent Object from Trusted Storage

7.5.4.2 TEE_OpenPersistentObject & TEE_ReadObjectData

ReCRED Secure Storage Code

```
TEE_ObjectHandle persistent_object = (TEE_ObjectHandle)NULL;
uint32_t objectID = params[1].value.a;
uint32_t objectIDsize = sizeof(objectID);
uint32_t flags_open = TEE_DATA_FLAG_ACCESS_WRITE | TEE_DATA_FLAG_ACCESS_WRITE
| TEE_DATA_FLAG_ACCESS_READ | TEE_DATA_FLAG_ACCESS_WRITE_META;

    char dataRead[256];
    uint32_t count;
    persistent_object = (TEE_ObjectHandle)NULL;


    result = TEE_OpenPersistentObject(
                TEE_STORAGE_PRIVATE,
                &objectID,
                objectIDsize,
                flags_open,
                &persistent_object);

    result = TEE_ReadObjectData(
                persistent_object,
                dataRead,
                256,
                &count);


    void *exp_res = NULL;
    exp_res = params[3].memref.buffer;
    memset(exp_res, '\0', params[3].memref.size);
    memcpy(exp_res, dataRead, 256);
```

8 Privacy and security considerations

The privacy and security section is considering the threat model, limitations and mitigation techniques, in reference with the DCA components documented in the previous chapters.

8.1 Biometric Identity

Despite the many advantages of biometrics over other traditional authentication methods (e.g. passwords and tokens), there are still some privacy and security considerations that also need to be addressed.

ReCRED takes advantage of two different physical biometrics for user-to-device authentication: fingerprints and face, described in sub-chapters 2.2.2 and 2.1 respectively.

Fingerprints are part of the native Android TrustZone authenticators, therefore most of the security issues are addressed directly by the Android OS. More specifically, ReCRED only stores an encrypted, binary representation of fingerprint templates. These templates are safely stored inside the user device and are never transferred to any cloud-based central storage, such as the ID Consolidator.

Fingerprints also have some native advantages over other physical biometrics. Fooling a fingerprint scanner would require a lot of effort and expertise (e.g. use of advanced methods in order to replicate the finger). Furthermore, they are not totally irrevocable, in the sense that even if a fingerprint is compromised some other finger can be used. Finally, since only binary templates of fingerprints are

stored, even if such a template is compromised, the hacker will not have access to the actual features and no private info will be derived.

Regarding face recognition, no actual user photos are stored. As soon as the n-dimensional feature vectors are extracted, the user photos are immediately deleted. Moreover, all the matching is based on binary facial templates comparisons. All the feature vectors are stored on the device internal storage, which means that they are private to the application and cannot be accessed by the user or other applications, unless the device is rooted. If the user uninstalls the application all the stored vectors are deleted from the device.

Face recognition was chosen as an additional biometrics authenticator, mostly because it is easy to use, it is fast, it does not require any specialized sensors (only a decent embedded camera) and it comes more natural to the users. On the other hand, face recognition admittedly has some native disadvantages. More importantly, face detection systems are easier to be spoofed, they result more false positives or even negatives and they work poorly (or even not at all) under poor lighting conditions.

8.2 Behavioral Authentication as a Second Factor

For privacy reasons, the BAA does not store user behavior events but rather a user profile that is extracted from such events, as it was described in Chapter 4.

For example, the BAA stores number of incoming/outgoing calls rather than CDRs, frequency visits to internet domains rather than complete URLs, or typing features rather than the actual text a user types. Nevertheless, the BAA database should be encrypted and backed-up securely to avoid malicious intrusion and data leakage.

Regarding mobility- and browsing-based authentication, no information is ever stored on the device.

The B-verifier application does not store any user typing signature or patterns in the user device. The typing features are sent to the BAA server where they are processed and evaluated.

Communication between the device and the BAA as well as the BAA and the databases where network events are stored (e.g., CDRs) is always secured (e.g., by using HTTPS or IPSec).

Finally, the user has to accept an agreement for using the ReCRED platform as a secure service and thus will be notified about the use of their data.

8.3 QR Login

In order to ensure the QR Authentication security several prerequisites must be fulfilled. To mitigate MITM attacks and eavesdropping all the HTTP messages (exchanged between the Service Provider and the QR Authenticator) must be encrypted using a TLS secure channel (the used TLS protocol version must be v1.2 and only in case it is not available, v1.1 should be used). Regarding the server messages authenticity, the JWT (JSON Web Token) exchanged between the QR Authenticator and the Service Provider must be signed using a HMAC algorithm (the minimum recommended hash function is SHA-256). It is also assumed that the QR Authenticator and the Service Provider are sharing a secret key for HMAC computation and validation.

In order to mitigate web session hijacking attacks, the Service Provider session token will be a nonce, unrelated with the web session ID: both the session token and the QR code will be random byte strings with a minimum length of 32 bytes.

The message sent from the mobile application to the QR Authenticator must be transmitted over a secure and authenticated channel. It is assumed that the mobile application is already authenticated to the QR Authenticator module (using FIDO, two-way TLS or other mechanism) before scanning the QR code. Taking this into consideration, the QR code sent from the mobile application to the QR Authenticator will be signed only if the authentication protocol requires it (for instance if using two-way TLS it is not necessary to sign the message).

8.4 FIDO and Federated Authentication

In order to achieve a sound security level, the following recommendations regarding algorithms and key lengths should be considered when implementing FIDO:

- *ServerChallenge* is defined as a byte array that contains random data. This data is generated by the FIDO UAF Server, sent to the Client and further included in a response message to protect from Reply Attacks.

The length of such an array is defined to be at least 8 bytes and no more than 64 bytes. 8 bytes should ensure a minimum security level and 64 bytes should ensure a SHA-512 compatibility

- Authenticators are recommended to use more sources of random seeds to increase the quality of random numbers generated (RFC4086)
- The FIDO UAF Protocol relies on TLS to ensure messages privacy. Protocol messages require a channel binding structure that binds security elements from the communication channel into protocol messages. The *ChannelBinding* structure is added by the client and verified by the server to protect from MITM attacks. The following bindings are supported (RFC5929):
    - TLS channel ID
    - Server End Point
    - TLS server certificate
    - TLS Unique
- The supported key type used in *ChannelID* is EC (elliptic curve) and more specifically, the P-256 EC key type as specified by NIST
- The server endpoint of TLS connection must be at the Relying Party and the client endpoint must be the FIDO UAF Client or the User Agent
- The used TLS protocol version must be v1.2 and only in case it is not available, v1.1 should be used. Insecure algorithms in TLS protocol should also be avoided (e.g. MD5, RC4, SHA1) – NIST recommendation (SP800-131A)
- A usage of SHA-1 in digital signatures generation context becomes deprecated in NIST standards (SP800-131A) and ENISA recommendations (Algorithms, Key Sizes and Parameters Report, 2013) the minimum recommended hash function is SHA-256.

8.5 Authentication and Consent Modules

The authentication management module will receive the user’s identification, such as username, phone number, email address and be responsible for coordinating the authentication process for the user against Identity Providers. The communication between the user’s browser or application and the Authentication module will be secured via TLS. The communication with the Service Providers (relying party) will support the OpenID Connect Authorization code flow to reduce the exposure of the user’s identity token.

No user information will be stored in the Identity Consolidator’s Authentication module, instead all the data will be stored in the ReCRED Storage API which will in turn be queried by the authentication module.

When a user want to get access to a relying party, the relying party will redirect the user to an OpenID Connect Provider in order to get access to identity attributes that the service would like. The user will be presented with the details of this request and will be able to provide consent or refuse access to their data thereby enabling the end user to manage their own privacy requirements through consent. Once access to these attributes has been granted, the use may later return to the consent management interface and revoke this access. Although the relying party would have been able to “cache” the previous version of the attribute, the RP will no longer be able to retrieve modifications or updates to the user’s data.

8.6 FIDO Extensions for ABAC and Anonymous Credentials

FIDO aims at reliably identifying the user in order to authorize her. However, user identification may not always be needed or desirable, as it may jeopardize the user’s privacy. Our proposed extension to the FIDO UAF protocol, described in Chapter 6, aims at retaining the benefits of FIDO, especially in terms of usability, while employing anonymous credentials to implement attribute-based access control. Moreover, our proposed approach does not change substantially the FIDO UAF protocol, thus we do not foresee any additional security threats.

8.7 Open-TEE

As it was discussed in Chapter 7, Open-TEE will be the Trusted Execution Environment of choice for ReCRED in order to enable the TEE code to be easily ported to any commercial or non-commercial TEE implementation. While a specific software implementation (i.e., OS, kernel, etc.) is required to enable TEE, Open-TEE is a software virtualization solution that manages to emulate the TEE components and provide the entire GlobalPlatform TEE API functionality, regardless of the choice of the real TEE implementation.

Being hardware-independent and GlobalPlatform compliant, Open-TEE Client and Trusted Applications can be compiled for any TEE and run on Android devices as well. Therefore, we cannot raise any privacy and security considerations for Open-TEE, as it is a virtual and not a real environment. Android-related TEE considerations were documented in Deliverable D3.1, sub-chapter 5.5.

9 Conclusions

Deliverable D3.3 “Description of DCA protocols and technology support (revised)” is the third deliverable of WP3. It revises and extends the protocols related to Device-Centric Authentication (DCA) that have been established in Deliverable D3.1, documenting in detail the implementation progress made so far.

After giving an overview of the current infrastructure of all authentication mechanisms, we described the status of the FIDO UAF Client and Server. By integrating FIDO UAF in ReCRED’s protocol stack we aim to achieve a strong, password-less authentication scheme that preserves user privacy while maintaining interoperability. The communication channel gateSAFE, also mentioned in this document, enables secure access, accounting and control by leveraging technologies like Transport Layer Security and Digital Certificates. Introducing gateSAFE in ReCRED allows us to take advantage of the gateSAFE features like TLS endpoint and to further extend gateSAFE to support the new authentication protocols that are more privacy preserving for the end-user, like FIDO UAF, OAuth and U-Prove.

Another key component of the ReCRED architecture, the Behavioral Authentication Authority, was presented in this deliverable. BAA’s main goal is to monitor user behavior, build behavioral profiles, and use these profiles to accommodate user authentication requests on behalf of relying parties. The behavioral profiles are created by the input that is provided by the behavioral capturing modules on the BAAs and on the user's devices and are later being stored in the behavioral profiles database, that were also described in this document.

Open-TEE, as the Trusted Execution Environment of choice for ReCRED, was detailed in D3.3 too. Open-TEE is a software virtualization solution of a real Trusted Execution Environment (TEE) that has many benefits compared to the other TEE implementations, as it emulates the TEE components and provides the entire GlobalPlatform TEE API functionality. A number of cryptographic operation examples were presented, as well as the Secure Storage functionality.

In Chapter 8, the related privacy and security considerations that were raised regarding the previously described DCA components, were documented.

10 References

[1] https://source.android.com/security/authentication/fingerprint-hal.html
[2] https://developer.android.com/reference/android/util/DisplayMetrics.html
[3] https://developer.android.com/guide/practices/screens_support.html
[4] https://developer.android.com/reference/android/text/InputType.html
[5] https://github.com/AnySoftKeyboard/AnySoftKeyboard
[6] http://www.date4j.net/
[7] http://square.github.io/okhttp/
[8] https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-uaf-protocol-v1.1-rd-20161005.html
[9] ARM TrustZone Architecture, 2015, http://www.arm.com/prodcuts/processors/technologies/TrustZone/index.php
[10] Intel SGX, https://software.intel.com/en-us/sgx
[11] Kinibi https://www.trustonic.com/products/kinibi
[12] SecuriTEE, http://www.sola-cia.com/en/securiTee/product.asp
[13] OP-TEE https://wiki.linaro.org/WorkingGroups/Security/OP-TEE
[14] TLK, http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=summary
[15] T6, https://github.com/liwenhaosuper/t6
[16] Open-TEE, https://open-tee.github.io/
[17] SierraTEE, https://www.sierraware.com/open-source-ARM-TrustZone.html [retrieved 9/1
[18] GlobalPlatforms Specifications, http://www.globalplatform.org/specificationsdevice.asp [retrieved 9/1/2017]
[19] http://www.icri-sc.org/icri-sc/institute/ [retrieved 9/1/2017]
[20] https://se-sy.org [retrieved 9/1/2017]
[21] https://fidoalliance.org/about/overview/
[22] https://fidoalliance.org/certification/fido-certified/
[23] https://www.certsign.ro/certsign_en/noutati/gatesafe-uaf-module-certsign