Ezio Mobile SDK V4.9

Programmer's Guide

gemalto
security to be free

All information herein is either public information or is the property of and owned solely by Gemalto NV. and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Gemalto's information.

This document can be used for informational, non-commercial, internal and personal use only provided that:

- The copyright notice below, the confidentiality and proprietary legend and this full warning notice appear in all copies.
- This document shall not be posted on any network computer or broadcast in any media and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided "AS IS" without any warranty of any kind. Unless otherwise expressly agreed in writing, Gemalto makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Gemalto reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Gemalto hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Gemalto be liable, whether in contract, tort or otherwise, for any indirect, special or consequential damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Gemalto does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Gemalto be held liable for any third party actions and in particular in case of any successful attack against systems or equipment incorporating Gemalto products. Gemalto disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that result from any use of its products. It is further stressed that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service or loss of privacy.

June 18, 2019

Contents

Preface

About This Guide

This document explains the features an application needs to provide in order to match the functional and security requirements expected for a mobile banking application based on Ezio Mobile SDK. It gives an overview of the API introduced by the SDK and details on the main features of the SDK with their relevant code snippets for both Java and C interfaces. This guide also provides simple troubleshooting tips that solve most of the integration problems with the SDK.

Who Should Read this Guide

This guide is intended for Gemalto customers /application developers who are responsible for developing on and integrating with Ezio Mobile SDK. This guide uses the term "application developer" to refer to the people implementing or customizing the product and "user" to refer to the mobile phone end-user of the mobile SDK solution.

This guide includes many references to *Ezio Mobile SDK V4.9 Security Guidelines*. These references are in the form:

- GENxx
  For general security guidelines, platform independent.
- ANDxx
  For Android specific practices.
- IOSxx
  For iOS specific practices.

For More Information

The following table contains the complete list of documents for Ezio Mobile SDK V4.9.

Document	Description
Ezio Mobile SDK V4.9 Overview	This serves as an introduction to the Ezio Mobile SDK product. It includes information on the package contents, an overview of the system, and a description of the features and services provided by the SDK.
Ezio Mobile SDK V4.9 Release Notes	This contains detailed release information such as new features, issues fixed, known issues, devices and OS versions tested.
Ezio Mobile SDK V4.9 Migration Guides	This set of documents contains the necessary steps when migrating from an earlier versions to *Ezio Mobile SDK V4.9*.
Ezio Mobile SDK V4.9 Security Guidelines	This contains best and recommended security practices that an application developer should follow.
Ezio Mobile SDK V4.9 API Documentation	This document details the interfaces provided by the Ezio Mobile SDK libraries on Android and iOS platforms

Document	Description
Ezio Mobile SDK V4.9 Programmer's Guide	This guide details the usage of the Ezio Mobile SDK when extending the features, functionalities, and interfaces of Ezio Mobile solution.

You may also refer to the following list of links and documents:

- *Chip Authentication Program - Functional Architecture 2007*(MasterCard)
- Android PRNG Fix
- HOTP RFC
- TOTP RFC
- OCRA RFC
- *Gemalto Dynamic Signature (DS) Specification 1.8*
- *Gemalto Generic SWYS Specification 2.2*
- Android, Invalid Key Exception
- Android Q Privacy

Contact Us

For contractual customers, further help is provided in the Gemalto Self Support portal at http://support.gemalto.com or you can contact your Gemalto representative.
Gemalto makes every effort to prevent errors in its documentation. However, if you discover any errors or inaccuracies in this document, please inform your Gemalto representative.

Introduction

This chapter provides a brief overview of Ezio Mobile SDK.

About Ezio Mobile SDK

Ezio Mobile is an enterprise authentication solution for e-banking and e-commerce functions such as one-time password (OTP) management, challenge-responses, transaction data signing, data protection, and out-of-band (OOB) communication. It utilizes the user's mobile device as the security platform. The solution consists of mobile applications based on the Ezio Mobile SDK (software development kit).
For the detailed product overview, refer to *Ezio Mobile SDK V4.9 Overview*.

Installing Ezio Mobile SDK

Ezio Mobile SDK must be installed as part of your development platform. The first step is to unzip the contents of the Ezio Mobile SDK release onto the host platform. The steps required to build an application based on the Ezio Mobile SDK are listed under their target platforms. To test an application based on Ezio Mobile SDK, various servers are required depending on the feature under the test (for example, an authentication server to verify an OTP).

Each platform provides different versions of Ezio Mobile SDK:

- A debug version with additional debug information to simplify application development.
- A release version where the security configuration is enforced by the version (for example, TLS configuration). As the debugger detection is applied to some features (for example, secure storage), it will halt the execution when a debugger is attached to the application process. In addition, for the Android release version, anti-hooking feature has been included where it crashes the application when hooking is detected. This feature is disabled by default. For users who want to enable the anti-hooking feature, refer to Antihooking Detection.

  Note:

  Approximately 2% of the devices on the field may potentially be detected as being hooked.

- iOS-specific: A debug_nocoverage version which is similar to the debug version but without code coverage data. By using this version, application is not forced to enable code coverage.

Ezio Mobile SDK provides the following libraries:

- Android: In `debug/`, and `release/`, different types of libraries are provided.
  - Java library

    `libidpmobile.jar`

    You can configure the `build.gradle` file to link the JAR file of debug and release versions correctly:

    ```
    dependencies {
        debugCompile files("${your-ezio-root-
    dir}/android/debug/libidpmobile.jar")
        releaseCompile files("${your-ezio-root-
    dir}/android/release/libidpmobile.jar")
    }
    ```

  - Native libraries

    Since 4.6, the sensitive codes were moved from Java to C which includes a shared library `libidp-shared.so`. Ezio Mobile SDK does not support MIPS ABIs as there is no device using this architecture. Since 4.7, Ezio Mobile SDK ceases the support for armeabi ABI.

    For a default flavor of the Ezio Mobile SDK delivery package, the native libraries include:

```
├── arm64-v8a
│   ├── libidp-shared.so
├── armeabi-v7a
│   ├── libidp-shared.so
├── x86
│   ├── libidp-shared.so
└── x86_64
    ├── libidp-shared.so
```

You can copy these native libraries to the `jniLibs` directory of debug and release versions respectively, or configure your `build.gradle` file as the following codes to link the debug and release version of native libraries correctly:

```
android {
    ...
    sourceSets {
        debug {
            jniLibs.srcDirs += ["${your-ezio-root-dir}/android/debug"]
        }
        release {
            jniLibs.srcDirs += ["${your-ezio-root-dir}/android/release"]
       }
    }
    ...
}
```

- iOS
    - `debug/EzioMobile.framework`
    - `debug_nocoverage/EzioMobile.framework`
    - `release/EzioMobile.framework`

On Android Platform

Ezio Mobile SDK library for Android must be included as part of the classpath when building for Android. Android Support Library must be downloaded from Android SDK Manager and imported as a library in the Android project. A typical Android project must have a link or copy of the library in the project's libs directory before issuing a build command from Ant. In Eclipse and Android Studio, a project can link the library by including it as an external JAR file in the project's Java build path. Ezio Mobile SDK for Android requires android.support.v4 library version 23.0 or later so as to support the keypad in Secure Input Service which is `android.support.v4.app.DialogFragment`. The activity which displays the SecurePinPad needs to extend `android.support.v4.app.FragmentActivity`. This library is also required to support Android M for runtime permission handling.

Native codes are embedded into SDK to enhance the root detection functionality. Besides the JAR files, dynamically linked library files (with extension '.so') are included in the delivery. These files are to be copied or linked to the project. For Eclipse user, copy all the folders (named by its architecture) to the project's libs directory. For Android Studio users, copy them to the `src/main/jniLibs/ directory`.

Note:

Application developer using Android Studio version 2.0 or later, needs to disable the following settings:

- For Mac:
  Android Studio > **Preferences** > **Build/Execution/Deployment** > **Instant Run** > **Enable Instant Run to hot swap code/resource on deploy (default enabled)**
- For Windows:
  File > **Settings** > **Build, Execution, Deployment** > **Instant Run** > **Enable Instant Run to hot swap code/resource on deploy (default enabled)**

Otherwise, the application will crash due to anti-hooking detection in Ezio Mobile SDK V4.8. Alternatively, for development purposes, you can unset the `HookingDetectionListener` to skip the hook detection.

Warning:

- Ezio Mobile SDK uses some immutable characteristics of the platform and the application's package name to initialize its environment. By default, the application that uses Ezio Mobile SDK is sealed to the device on which the Ezio Mobile SDK services are used the first time.
- Since 4.6, Ezio Mobile SDK had ceased support for Android 2.x and 3.x. This means that Gemalto does not validate the SDK on Android 2.x, 3.x and no longer gurantees the functionalities of the SDK on Android versions earlier than 4.0. Since 4.7, Ezio Mobile SDK had ceased support for Android versions earlier than 4.4. You have to perform your own validation, if required.
- With the use of Dexguard encryption features in Ezio Mobile SDK, Dexguard will try to write temporary files during encryption/decryption at runtime. Application integrator must include the following codes in your application before initializing Ezio Mobile SDK.

```
System.setProperty("java.io.tmpdir",
      getDir("files",
      Context.MODE_PRIVATE).getPath());
```

JNA Usage

As the JNA library (version 4.5) is used since EZIO Mobile SDK 4.6, you can use any JNA version which is 4.5 and later. However, other versions of JNA are not guaranteed to work with EZIO Mobile SDK.

1. Configure JNA in your project build.gradle

```
compile 'net.java.dev.jna:jna:4.5.0'
```

2. Include the **libjnidispatch.so** shared library for all the Android ABIs that your project supports.
    a. Navigate to https://github.com/java-native-access/jna/releases.
    b. In Version 4.5.0, download the ZIP file.
    c. Unzip the package, navigate to the `jna-4.5.0/dist/` directory. `libjnidispatch.so` for different ABIs can be extracted from respective JAR file.
        - arm64-v8a – android-aarch64.jar
        - armeabi-v7a – android-armv7.jar
        - x86 – android-x86.jar
        - x86_64 – android-x86-64.jar

APK Size Reduction

The native shared libraries will have some impact on the APK size. One suggestion to reduce the size of your APK is to only include the ABIs that your project supports. For some other techniques to reduce the APK size, refer to https://developer.android.com/topic/performance/reduce-apk-size.html.

On iOS Platform

The following frameworks have to be added to your project in order to link your application with Ezio Mobile SDK:

- Foundation.framework
- SystemConfiguration.framework
- UIKit.framework
- LocalAuthentication.framework
- Security.framework
- AVFoundation.framework
- CoreMedia.framework
- Accelerate.framework

Also, add the following dynamic library to your project:

- `libc++.dylib`
- `libsqlite3.0.dylib`

Add the following property into the info.plist file.

- Privacy - Camera Usage Description

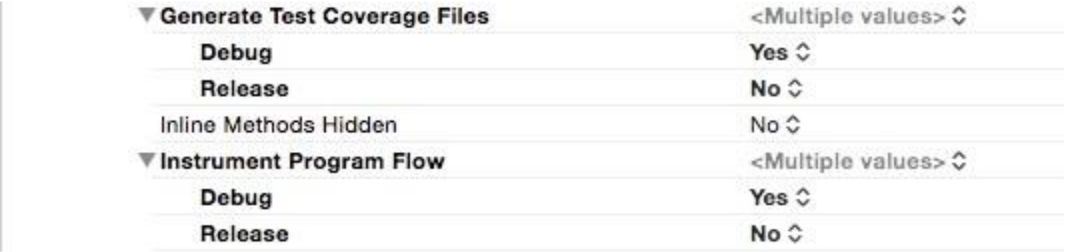
The compilation option –ObjC must also be added in the *other linker flags* item under the Xcode project build settings including –ObjC –all_load. In addition the following security guidelines must be enforced for security when setting the application project:

- For IOS06, remove the symbols from Xcode output.
- For IOS07, enable Xcode compiler security and obfuscation options.

To be able to use the debug version of Ezio Mobile framework and prevent errors during the build, you may want to use the debug_nocoverage framework or set two options in the **Build Settings** section of your project:

- Generate Test Coverage Files: **YES** for the **Debug** version, **NO** for the **Release** version.
- Instruments Program Flow: **YES** for the **Debug** version, **NO** for the **Release** version.

Figure 1 Option for Build Settings



▼ **Generate Test Coverage Files**	<Multiple values> ⇕
Debug	**Yes** ⇕
Release	**No** ⇕
Inline Methods Hidden	No ⇕
▼ **Instrument Program Flow**	<Multiple values> ⇕
Debug	**Yes** ⇕
Release	**No** ⇕

Application Requirements

Ezio Mobile SDK imposes requirements on the application during its development. This chapter lists the general requirements of the application. Refer to the subsequent chapters for specific feature requirements.

Security Guidelines

A key step to ensure the security of the application is to adhere to the guidelines in *Ezio Mobile SDK V4.9 Security Guidelines*. These are generally the most important ones:

- GEN01: Code obfuscation
- GEN05: No debug symbols
- GEN09: AND01. IOS01. Sensitive data leaks.

User Interface

The application is responsible for displaying all UIs unless specifically stated. A typical example of a user interaction requesting the user for the PIN or displaying a message pushed by the bank. Hence, the application has to enforce these security guidelines related to the user interface:

- GEN08: PIN sanitization.
- GEN12: Wiping assets.
  - AND07: Clearing assets before a TLS session (for example, provisioning)
  - IOS05: Clearing assets.
- GEN14: Use a trusted keyboard.
- GEN16: Use separate channels for sensitive data.
- AND01: Sensitive data leaks.
- IOS01: Sensitive data leaks.

Network

The application is responsible for the sending and receiving of data over a network connection. In specific cases such as when creating a token, Ezio Mobile SDK handles the network connection itself. The application must enforce these security guidelines related to network connections to secure the network connection:

- GEN02: Communication over HTTP is forbidden.

- GEN03: Use POST, not GET.
- GEN04: Reject invalid SSL certificates.
- GEN16: Use separate channels for sensitive data.

On Android

The following permissions are requested by the application:

Table 1 Network Permissions

Permissions
android.permission.INTERNET
android.permission.ACCESS_NETWORK_STATE

Obfuscation

Android Obfuscation

It is highly recommended that your application be obfuscated with ProGuard/DexGuard prior to delivery. Ezio Mobile SDK has applied obfuscation and advanced class/string encryption internally. In order to obfuscate your application on top of Ezio Mobile SDK, the following ProGuard configuration needs to be included in your ProGuard configuration.

Note:

In the EZIO Mobile SDK delivery package, a recommended configuration "proguard-project.pro" is provided for ProGuard .

Additional configuration may be required for Dexguard users for resource related configuration. Refer to *GuardSquare* documentation and standard DexGuard configurations provided on application resources. An example of the configuration is shown in the following. The configuration may differ when different DexGuard version is used.

```
-dalvik
-android


# Ezio native libs
-keepresourcefiles lib/**/libjnidispatch.so
-keepresourcefiles lib/**/libidp-shared.so
# for faceID users
-keepresourcefiles lib/**/libopenblas.so
-keepresourcefiles lib/**/libfid.so
-keepresourcefiles assets/data/**.ndf


# adapt resources etc...
```

Warning:

Use `flattenpackagehierarchy` util for package flattening in order to hide the sensitive packages of Ezio SDK.

Logs are stripped off from the release version of Ezio Mobile SDK.
To strip the logs from the application level, include the following codes in your ProGuard/DexGuard configuration:

```
-assumenosideeffects class android.util.Log{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
}
```

For better obfuscation, it is highly recommended to apply the following:

```
# Add the following if you wish not to see the non-obfuscated class name in the
stackTrace
-renamesourcefileattribute SourceFile
```

To keep track of your obfuscation mapping and debug purposes, apply the following:

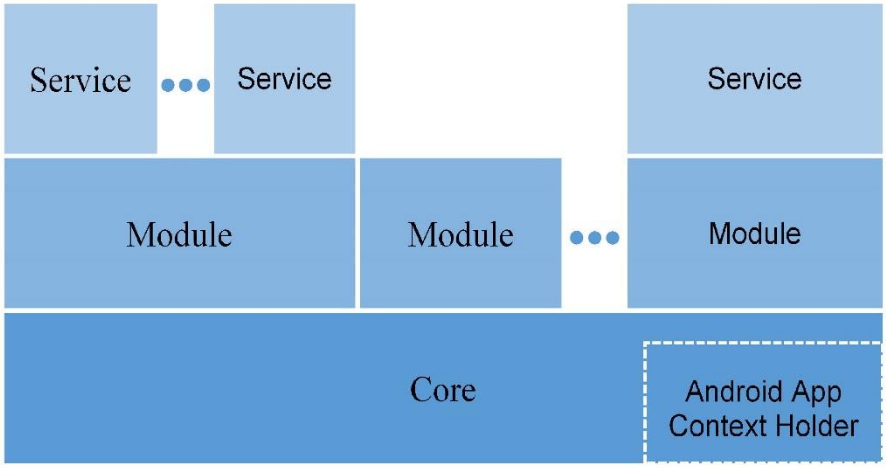
```
# Keep this file as private since it contains sensitive information
# Ezio SDK will be obfuscated further when building the application, the mapping
file
will be needed in order to report stackTrace for trouble-shooting later on.
-printmapping yourMappingFile.txt
```

Design

This chapter provides a high-level design description of Ezio Mobile SDK. It covers the purpose of using the SDK and naming scheme of the major components of the API, the relevant platform security enhancements, and any other constraints imposed by the SDK. The general hierarchy of the product is shown in the following figure.

Ezio Mobile SDK consists of a core module with an optional Android application context holder, and several modules and services.

Figure 2 Hierarchy of Modules and Services in Ezio Mobile SDK



Where:

- Core Module
  The core module defines the entry point of Ezio Mobile SDK and the foundation of the essential features. It is also the point where the modules are configured by specifying the configuration data associated with the modules. Refer to Core Module for more details.
  The core module on Android has an additional API, Android application context holder that must be set up by the application. The application must provide its Android application context instance to Ezio Mobile SDK. This context is used to access Android APIs that require a context instance.
- Module
  A module defines a high-level feature set. It can be a standalone and can composed of one or more services. The following chapters cover each module in detail.
- Service
  A service defines a specialized feature of a module. If no specialized features are needed, a module will not expose the services.

Platform Security Enhancements

Android Secure Random Fixes

Ezio Mobile SDK automatically fixes the security issues on affected Android versions whose random number generation did not receive cryptographically strong values. The Google-provided fix installs a Linux PRNG-based SecureRandom implementation as the default provider PRNGFIX. The effect is that new `SecureRandom()` and `SecureRandom.getInstance(SHA1PRNG)` will return a SecureRandom object backed by Linux PRNG-based SecureRandom.

Note:

If the fix cannot be applied, then the list of security providers will not be updated.

Application Constraints

Thread Safety

Ezio Mobile SDK APIs are not thread-safe unless explicitly stated. Therefore, special care must be taken by the application to ensure that concurrent access or modification to any SDK object or data is explicitly protected by the application. If multiple threads use the SDK's features, then it is recommended to check the pertinent concurrency features of each platform.

SDK Data Migration

Ezio Mobile SDK cannot be downgraded to an older version. Once the application is executed on the device, the same application cannot be updated to a version that uses an earlier version of Ezio Mobile SDK. If an earlier version is to be installed, ensure that the persistent storage is cleared and you have to re-provision the credentials.

By default, Ezio Mobile SDK is set to automatically update itself to a later version, unless otherwise specified. When the application is updated, the credentials are automatically migrated by the new version of Ezio Mobile SDK.

Note:

Special care must be taken for tokens created with custom fingerprint data. The application must provide the same custom fingerprint data that was used when the token was created. Otherwise, the token will be rendered permanently unusable by the updated application.

The following table summarizes the different scenarios when updating Ezio Mobile SDK:

Table 2 Supported Version Upgrades

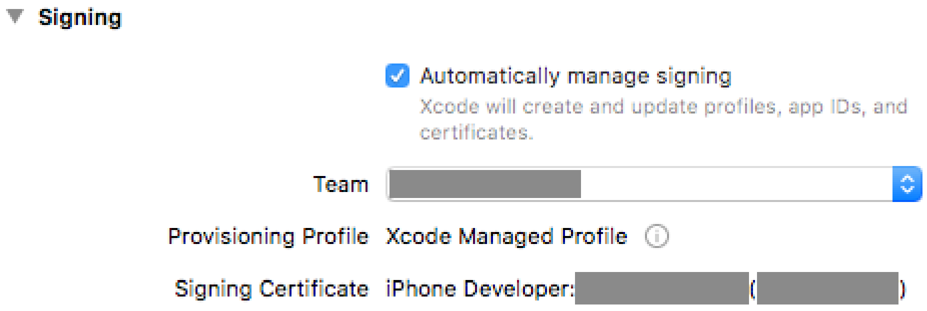
From/To	3.1	3.2	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9
3.0	Yes	Yes	Yes	Yes	Yes	Yes	Yes*	Yes*	Yes*	Yes*	Yes*	Yes*
3.1		Yes	Yes	Yes	Yes	Yes	Yes	Yes*	Yes*	Yes*	Yes*	Yes*
3.2			Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.0				Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

From/To	3.1	3.2	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9
4.1					Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.2						Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.3							Yes	Yes	Yes	Yes	Yes	Yes
4.4								Yes	Yes	Yes	Yes	Yes
4.5									Yes	Yes	Yes	Yes
4.6										Yes	Yes	Yes
4.7											Yes	Yes
4.8												Yes

Where * indicates the version upgrade should work theoretically. However, the migration test has not been performed for these cases.

Note:

When using iOS Simulator, apart from having the application bundle id consistent in the old and updated version, ensure that the certificate signing identity is also consistent for both app versions. This is because even if the simulator does not require signing, part of the certificate information is used in one of the keychain properties. Such constraint is not an issue when using an actual device because the device disallows the upgrading of an app signed with a different certificate.
On Xcode, Go to **Project** > **Target** > **General** > **Signing**, set the settings to either automatic or manual, as long as it is consistent in both versions.



▼ **Signing**

☑ Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team [redacted]

Provisioning Profile Xcode Managed Profile ⓘ

Signing Certificate iPhone Developer: [redacted] ([redacted])

Migrating from Ezio Mobile SDK Version 3 Onwards

Application data based on Ezio Mobile SDK version 3 and onwards is automatically upgraded and requires no explicit migration by the application.

Migrating from Version 2

For Android apps based on Ezio Mobile SDK version 2, it is mandatory to migrate the application data to support the protection provided by Password Manager. Refer to Migrating from Ezio Mobile SDK Version 2 on Android for more details.
For iOS apps, no migration procedure is needed.

Security Detection Service

This chapter describes how to set up the `SecurityDetectionServices` to report the security warnings to the application.

Anti-Hooking Detection

This section shows the usage of Anti-Hooking Detection callback functionality. Anti-Hooking detection is only available on iOS from version 4.7. The SDK provides the hooking detection technique which crashes the application if this feature is enabled and hooking is detected. Since version 4.4, you are able to catch `IdpHookingException` on Android to avoid the application from crashing.
In addition, the SDK has provided the callback functionality in which you can choose to continue (ignore) or to abort (fail) when hooking is detected. By default, the Anti-Hooking detection is disabled on version 4.7. To enable this functionality, `setHookingDetectionListener` is required.
Since version 4.8, the Anti-Hooking detection is enabled by default in the SDK.
`setHookingDetectionListener()` is not mandatory to be called, without listener, the SDK will throw an exception directly when a hook is detected.
On Android:

```
// Since 4.8, the Anti-Hooking detection is enabled by default in SDK.
setHookingDetectionListener() is not mandatory anymore, without listener, SDK will
throw an exception when hook is detected.
// This has to be called before Initializing IdpCore
SecurityDetectionService.setHookingDetectionListener(new HookingDetectionListener()
{
 @Override
 public boolean onHookingDetected() {
    // Return true to continue or false to abort
    // Application developer can choose to warn the user
    // By returning false, SDK terminates the execution and report the error.
    return false;
 }
});
// Removes the listener so that the SDK aborts the operation and
// throws Exception whenever hooking is detected
SecurityDetectionService.clearHookingDetectionListener();
```

Warning:

Usage of the callback will potentially raise another attack point for the attacker. The callback implementation has to be obfuscated. Refer to Android Obfuscation for the new configuration.
If the application uses callback, you have to set it before initializing the IdpCore.

On iOS:

```
// Since 4.8, the Anti-Hooking detection is enabled by default in SDK.
EMHookingDetectionSetListener() is not mandatory anymore, without listener, SDK will
report an internal generic error when hook is detected.
EMHookingDetectionPolicy otphookingDetectionListener() {
    // Return EMHookingDetectionPolicyIgnore to continue.
    // Return EMHookingDetectionPolicyFail to throw an exception
    // whenever hooking is detected.
    return EMHookingDetectionPolicyFail;
}

// This has to be called before Initializing EMCore if you want to react to the
hooking.
EMHookingDetectionSetListener(&otphookingDetectionListener);

// Removes the listener so that the SDK aborts the hooking detection
EMHookingDetectionClearListener();
```

Anti-Hooking Detection Error Reporting

Error reporting is only applicable when hooking detection feature is enabled and the `onHookingDetected()` callback returns a `false` value, or hooking is detected when there is no listener set. The error reporting feature is only available on iOS from version 4.8.

On Android:

In Ezio Mobile Android SDK 4.8, the following API calls will raise exception:

- `IdpCore.configure(...)`
- `FaceAuthVerifier.authenticateUser()`

Since Ezio Mobile Android SDK 4.6, instead of throwing `IdpHookingException`, the SDK throws a `PasswordManagerException` with an error code IdpResultCode.PASSWORD_MANAGER_STORAGE_NOT_OPEN on all the SDK features that requires a PasswordManager login. The exception has an error message that reports a generic storage exception.

- Secure Storage: `propertyStorage.open`
- One Time Password: createToken, getToken, APIs on Token where `PasswordManagerException` is thrown.
- Out-Of-Band Communication: All server calls such as register/unregister, fetchMessage/sendMessage, get/update/clear Notification profile when `OobException` is thrown.
- FaceID authentication: initialization, enroller and verifier features.

The same exception can be thrown if the application is not logged in to Password Manager. The SDK integrator must be able to differentiate:

- If a device is detected as hooked.
- If the application did not perform `passwordManager.login.`

To achieve the best security, the integrator should perform a code review and logged in to Password Manager before using any of the password protected features (such as SecureStorage, Out-Of-Band, One Time Password, FaceID).

Warning:

For security considerations, do not provide any hint at the point where hooking is detected. It implies that when the API `onHookingDetected` is invoked, the SDK integrator should not log or notify the user about the hooking detection. If needed, Log/Notification should be performed only at the point where `PasswordManagerException` with the above result code is caught.

On iOS:
Since Ezio Mobile SDK 4.8, instead of throwing an exception, the SDK will propagate an error message that reports a generic unknown keychain error message.

- Token: createToken, getToken, change PIN
- One Time Password: get an one time password.

SDK will report a generic invalid password error message for operations inside these modules:

- SecureStorage
- OOB

Warning:

For security considerations, do not provide any hint at the point where hooking is detected. It implies that when the API `onHookingDetected` is invoked, the SDK integrator should not log or notify the user about the hooking detection. If needed, Log/Notification should be performed only at the point where an API returns a generic unknown keychain error message or invalid password error message.

Anti-Debugging Detection

This section shows the usage of Native Anti-Debugging detection functionality on Android device. Release version of the SDK provides native debugging detection, and will crash the application if this feature is enabled and debugging is detected.This feature is is available since version 4.6 and it is not enabled by default. The application needs to call the following API to enable this feature.
This native anti-debugging detection feature is by default enabled since Ezio Mobile SDK V4.8, the following API still can be called in the application to disable/enable the debugger detection.
On Android:

```
// This has to be called before Initializing IdpCore
// This feature is only effective on Release version
// since 4.8, the native debugger detection by default is enabled, no need to call
this API anymore unless application wants to disable it.(by set the parameter to
false)
SecurityDetectionService.setDebuggerDetection(true);
```

For iOS, the anti-debugger detection feature is only supported on real devices, which is not applicable on the simulator. And in Xcode settings, `GCC_INSTRUMENT_PROGRAM_FLOW_ARCS` are to be set to `NO`.

Warning:

Usage of this API call will potentially raise another attack point for the attacker. This API call has to be obfuscated. Refer to Android Obfuscation for the new configuration.

Emulator Detection

This section shows the usage of emulator detection on Android only. Android emulator detection service is supported since Ezio Mobile SDK V4.8.
When the emulator is detected, the application will exit.

Warning:

Emulator detection is only available in the release version of Ezio Mobile SDK.

Potential Overlay Attack App Detection

Note:

This section is only supported on Android only from Ezio Mobile SDK V4.9.0.

This feature is used to detect if there are apps installed (non-system app) which will potentially do overlay attack to Ezio based app, and return the app info.

And the threat applications which have "Enabled accessibility service" and "Draw over the apps" permissions are treated as potentially overlay attack apps.
The Ezio based application has to call the following API to get the Info of the Apps which potentially does an overlay attack.

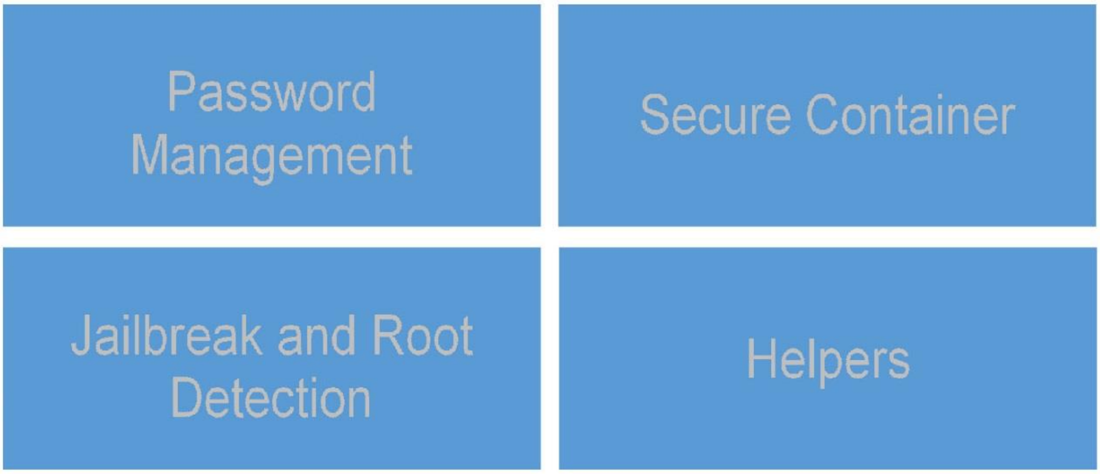
```
// call API to check if potential overlay attack app detected
// Set of ApplicationID(application package name) which have been detected as
potential overlay attack apps if found will be returned.
Set<String> potentialOverlayAttackAppInfo =
SecurityDetectionService.getPotentialOverlayAttackAppInfo();
```

After this API is called, if apps are detected, a set of Application ID of the Apps will be returned.
The application developer can use this Application ID to get the application name or application info to warn the user about the security risk.
To avoid the prompting of potential overlay attack apps to user, the application can maintain a whitelist to save the app names.

Core Module

This chapter describes how to configure and use the core module. The core module serves as the entry point of Ezio Mobile SDK and is the foundation of essential features. The data associated with the other modules are also specified here.

Figure 3 Main Core Features



Application Requirements

This section describes the preparation steps to be done before using the core module.

Android Application Context

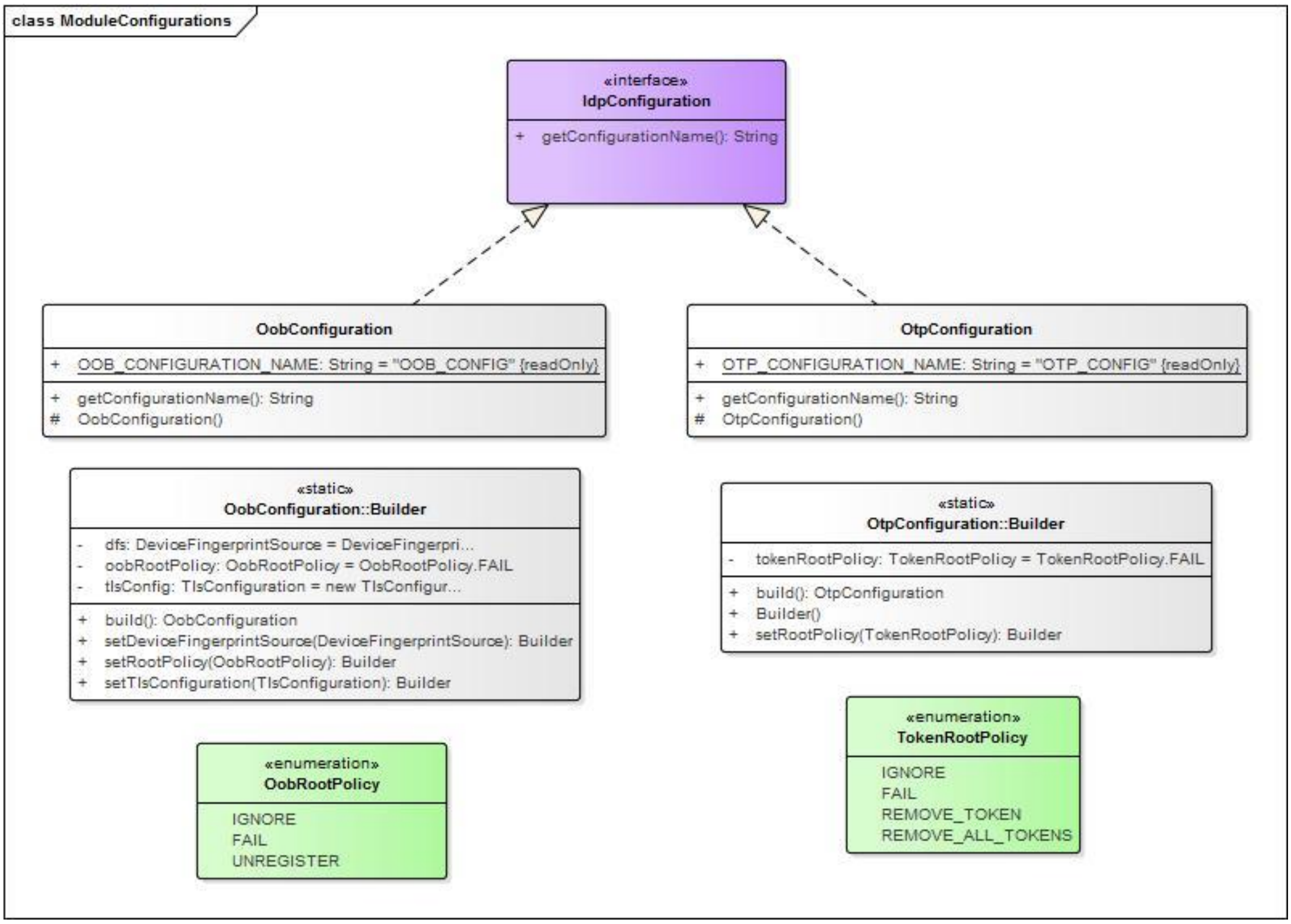
The application context instance of an Android application must be provided to Ezio Mobile SDK. The context is used to access Android APIs that require a context instance. This must be set before using any part of the SDK and it can only be set once.

```
ApplicationContextHolder.setContext(appContext);
```

Configuration

Ezio Mobile SDK has to be configured before it can be used. Each module has a corresponding configuration data that applies to the module and its feature sets, and is immutable at runtime.

Figure 4 Configurations of the Modules in SDK


class ModuleConfigurations
«interface»
IdpConfiguration
+ getConfigurationName(): String
OobConfiguration
+ OOB_CONFIGURATION_NAME: String = "OOB_CONFIG" {readOnly}
+ getConfigurationName(): String
# OobConfiguration()
OtpConfiguration
+ OTP_CONFIGURATION_NAME: String = "OTP_CONFIG" {readOnly}
+ getConfigurationName(): String
# OtpConfiguration()
«static»
OobConfiguration::Builder
- dfs: DeviceFingerprintSource = DeviceFingerpri...
- oobRootPolicy: OobRootPolicy = OobRootPolicy.FAIL
- tlsConfig: TlsConfiguration = new TlsConfigur...
+ build(): OobConfiguration
+ setDeviceFingerprintSource(DeviceFingerprintSource): Builder
+ setRootPolicy(OobRootPolicy): Builder
+ setTlsConfiguration(TlsConfiguration): Builder
«static»
OtpConfiguration::Builder
- tokenRootPolicy: TokenRootPolicy = TokenRootPolicy.FAIL
+ build(): OtpConfiguration
+ Builder()
+ setRootPolicy(TokenRootPolicy): Builder
«enumeration»
OobRootPolicy
IGNORE
FAIL
UNREGISTER
«enumeration»
TokenRootPolicy
IGNORE
FAIL
REMOVE_TOKEN
REMOVE_ALL_TOKENS

These configurations are built and passed to the SDK when the core module is configured:

- On Android:
  Preload the library to reduce the time spend on core configuration:

```
public class MyApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();
    // This is required before loading any Ezio classes.
    // It is required by Dexguard to perform decryption of java classes nad
native code
    System.setProperty("java.io.tmpdir", getDir("files",
Context.MODE_PRIVATE).getPath());
    // Preload loads the native libraries in UI thread
    // For Security purposes, the loading must be completed before going to the
Ezio SDK
    // If this is not called, the loading will be performed while initializing
IdpCore
    IdpCore.preLoad();
  }
}
```

Note:

The preLoad function does not improve the overall performance of the SDK. Without calling preLoad, the loading will be executed into IdpCore.configure(..) as shown in the following snippet. This gives the application the flexibility to call it from the Application onCreate() so it must be completed before the first activity shows up.

To configure the core module:

```
// Build the configuration for the OTP module
OtpConfiguration otpConfig = new OtpConfiguration.Builder()
          .setRootPolicy(TokenRootPolicy.IGNORE)
          .build();
IdpCore core = IdpCore.configure(otpConfig);
```

- On iOS:

```
// Build the configuration for the OTP module
EMOtpConfiguration *otpConfig = [EMOtpConfiguration
              configurationWithJailbreakPolicy:EMTokenJailbreakPolicyIgnore];

EMCore *core = [EMCore
              configureWithConfigurations: [NSSet setWithObjects:otpConfig,
nil]];
```

Retrieving Ezio Mobile SDK Version

You can retrieve the version of the current Ezio Mobile SDK using the respective functions:

- On Android:

```
String version = IdpCore.getVersion();
```

- On iOS:

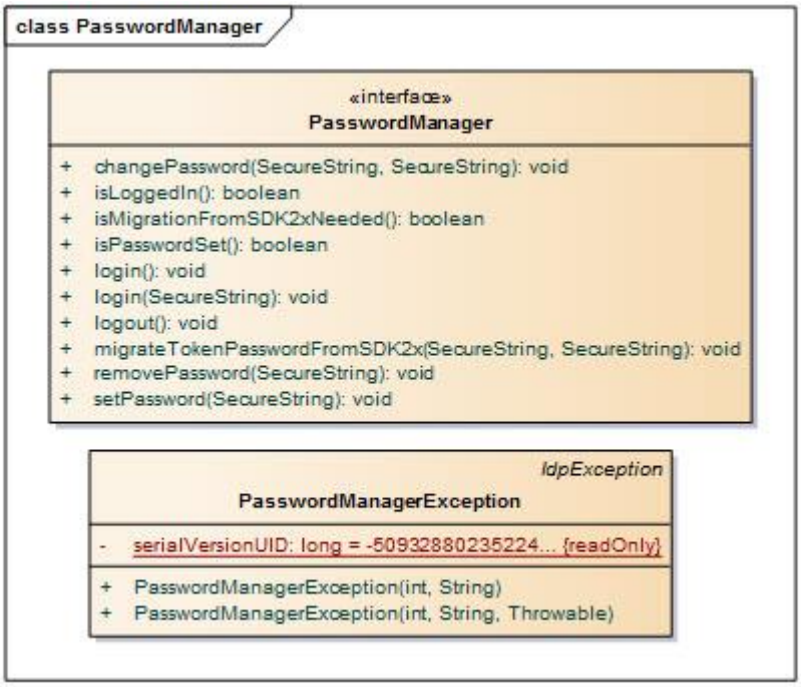
```
NSString* version = [EMCore version];
```

On these platforms, the value returned by the core module uses a three-part version number: <major>.<minor>.<patch>.

Password Manager

This feature manages the password that protects the SDK's persistent data. An application may choose to use this feature with or without a password, but in all cases a login must be explicitly done when needed. Ezio Mobile SDK API will indicate if it requires the LoggedIn state in its relevant section.A login is valid until an application explicitly logs out or exits. Therefore, it is recommended to logout as soon as possible.

Figure 5 Password Manager Class Diagram


class PasswordManager
«interface»
PasswordManager
+ changePassword(SecureString, SecureString): void
+ isLoggedIn(): boolean
+ isMigrationFromSDK2xNeeded(): boolean
+ isPasswordSet(): boolean
+ login(): void
+ login(SecureString): void
+ logout(): void
+ migrateTokenPasswordFromSDK2x(SecureString, SecureString): void
+ removePassword(SecureString): void
+ setPassword(SecureString): void
IdpException
PasswordManagerException
- serialVersionUID: long = -50932880235224... {readOnly}
+ PasswordManagerException(int, String)
+ PasswordManagerException(int, String, Throwable)

Login without a Password

This option is used when the password is not set. An application will still call the login method to authorize access to the protected data when it is necessitated by an API.

Warning:

As this variant is not secured, avoid using it whenever it is possible. The usage of a password is highly recommended.

- On Android:

```
// Login to password manager
try {
 core.getPasswordManager().login();
} catch (PasswordManagerException e) {
  // handle error
}
```

- On iOS:

```
// Login to password manager
NSError* error = NULL;
BOOL result = [[core passwordManager] login: &error];
if (!result) {
// Handle error
}
```

Managing the Password

By default, Password Manager is in the insecure no-password state. An application can query the state of Password Manager to check if a password has been set, set the password, change an existing password, or remove the password and go back to the passwordless login. The following series of code snippets demonstrate each of the actions.

Setting the Password

This is only valid if Password Manager did not use a password. This example also demonstrates how to check if a password is set.

- On Android:

```
if (!passwordManager.isPasswordSet()) {
    // Increase security by using a password login
    passwordManager.setPassword(password)
} catch (PasswordManagerException e) {
    // Handle error
}
```

- On iOS:

```
NSError* error = NULL;
if (![passwordManager isPasswordSet:&error]) {
    if (error) {
        // Handle error
    }
    BOOL result = [passwordManager setPassword:aPassword error:&error];
    if (!result) {
        // Handle error
    }
}
```

Changing the Password

You can only change the password if Password Manager uses a password.

- On Android:

```
try {
    passwordManager.changePassword(oldPassword, newPassword);
} catch (PasswordManagerException e) {
    // Handle error
}
```

- On iOS:

```
NSError* error = NULL;
BOOL result = [passwordManager changePassword:oldpassword
newPassword:newPassword error:&error];
```

```
if (!result) {
    // Handle error
}
```

Removing the Password

This is only valid if Password Manager uses a password. This reverts Password Manager back to a no-password login state.

Warning:

It is not recommended to work without a password as removing the password protection weakens your account's security.

- On Android:

```
try {
    passwordManager.removePassword(password);
} catch (PasswordManagerException e) {
    // Handle error
}
```

- On iOS:

```
NSError* error = NULL;
BOOL result = [passwordManager removePassword:aPassword error:&error];
if (!result) {
 // Handle error
}
```

Logging In with a Password

When an application is logged in with a password, access to APIs with protected data is authorized. The source of the password, typically comes from the user of the application.

- On Android:

```
SecureString password =
core.getSecureContainerFactory().fromString("MyPassword");
try {
 passwordManager.login(password);
} catch (PasswordManagerException e) {
 // Handle error
}
```

- On iOS:

```
id<EMSecureString> aPassword = [[core secureContainerFactory]
secureStringFromString:@"MyPassword"];
NSError *error = NULL;
BOOL result = [passwordManager loginWithPassword:aPassword error:&error];
if (!result) {
 // Handle error
}
```

Migrating from Ezio Mobile SDK Version 2 on Android

The application calls a migration procedure when it is run for the first time from version 2 of the SDK. The migration process takes the password used to protect the token data in version 2 and protects the data using the same password from Password Manager. If no password is used in version 2, then the migration is executed using the passwordless variant.

Warning:

The existing tokens will not work until they are migrated.

```
// Check and migrate if needed
try {
    if (passwordManager.isMigrationFromSDK2xNeeded()) {
        // Need migration, get information from user...
        SecureString oldPassword = getOldPasswordFromUser();
        SecureString newPassword = getNewPasswordFromUser();
        // Proceed with migration
        passwordManager.migrateTokenPasswordFromSDK2x(
                oldPassword, newPassword);
    }
} catch (PasswordManagerException e) {
    // handle error
}
```

Resetting Password Manager

This feature is used to reset Password Manager in cases where the user forgets the password.

Note:

Password Manager is used to protect the other features in the SDK. Resetting Password Manager causes other SDK features to be non-usable. Refer to the following Android and iOS instructions for more details.

- On Android:
  Password Manager is used to protect One-Time Password, Out-of-Band Communication, Secure Storage and Face Authentication features. Once it is reset, the existing data of the protected features are no

longer usable. The application needs to invoke the corresponding "reset" API for the protected features to clean up.

-
  - One-Time-Password: Re-provisioning is required to obtain a new token.
  - Out-of-Band Communication: Registration of new client is required.
  - Secure Storage: Existing data is removed and needs to be created again.
  - Face Authentication: Existing enrolled face template is no longer usable. Re-enrolment is required.

  Note:

  This does not consume additional license.

  ```
  try {
     passwordManager.reset();
  } catch (PasswordManagerException e) {
        // handle error
     }
  ```

- On iOS:
  Password Manager is used to protect Out-of-Band Communication and Secure Storage features. Once it is reset, the existing data of these features are no longer usable. PasswordManager reset will automatically clean up the data for Out-of-Band and Secure Storage.
  - Out-of-Band Communication: Registration of new client is required.
  - Secure Storage: Existing data is removed and needs to be created again.

```
NSError* error;
[passwordManager reset:&error];
// check error
```

Device Fingerprint

Fingerprinting is a security mechanism to limit the usage of app data to a specific device, also known as anti-cloning mechanism. It effectively seals any data with device information, service or software-based identifiers as well as the application-defined custom data. The following section describes the recommended usage and constraints of the fingerprint types.
Typically, these are the two processes which are concerned with the device fingerprint usage:

- Data Sealing: The process of sealing the data with device fingerprint, which takes place when data is written to the persistent storage.
- Data Retrieval: The process of retrieving the data from the persistent storage by providing the device fingerprint.

Device fingerprint must be consistent for the successful retrieval of data.

Device Fingerprint Sources

The device fingerprints are collected from the device/software. There are three device fingerprint sources that can be combined:

- Device information
  This information is linked to the device itself (not the operating system running on the device). If this is used and the user tries to restore the application and its data on another device, then all data on the other device becomes invalid. This information can be configured by a global flag `DeviceSourceBinding` in the core configuration which can be disabled by the application. `DeviceSourceBinding` is enabled by default. From Ezio Mobile SDK V4.9.2 onwards, this information is based on an auto generated file, while on previous version, it uses the IMEI/MEID/ESN information. Refer to *Migration for Android Q Upgrade* for more information.
- Soft information
  The information is linked to the software part of the device. Updating the operating system has no impact on it.
- Service information (Android only)
  From Ezio Mobile SDK V4.9.2 onwards, this information is **deprecated** and will be ignored due to Android Q Privacy Policy enforcement. Device Binding can only be configured through `IdpCore.configure` API using `DeviceSourceBinding` flag. However, this information is still required in the Data Migration process. Refer to chapter *Migration for Android Q Upgrade.* On previous version before V4.9.2, this device fingerprint source uses the IMSI information.

The device fingerprints sources are supplied at the time of data sealing and data retrieval. However, in the case of OTP token usage, the list of device fingerprints sources used will be saved during the provisioning and automatically configured at the time of retrieval.

Custom Fingerprint Data

It is highly recommended to use the custom fingerprint data, as it provides another layer of security over the stored data. It is up to the application developer to determine the type of data to use.

Warning:

If custom fingerprint data is used, the same custom fingerprint data must be provided to retrieve the stored data. This also implies that if custom fingerprint data has not been used previously in Ezio SDK V3.X, the later versions should not be changed to use the custom fingerprint data, as this will cause invalid fingerprint error when using features such as OOB.

Constraints

On Android

From Ezio SDK V4.9.2, Device Fingerprint does not requires any Android Permission. The permission of READ_PHONE_STATE is only required for data migration from previous version which uses `DeviceSourceBinding` or Fingerprint Type SERVICE.

- If `DeviceSourceBinding` is enabled, the user cannot restore the data after reinstalling an application even on the same device.

- If Type.SOFT is used, the user cannot use the restored data after doing a factory reset and reinitialized it to the same device. Likewise, the user cannot back up the application and restore it on a different device.

Note:

- When android.permission.READ_PHONE_STATE is used in Android M, if the application is targeting Android SDK version 23 and later, a pop-up message will prompt end user to allow the permission to read the phone's data. In order to bind the application data to the end user's handset only, the end user must allow this permission.

The following code snippet shows the fingerprint types used:

```
// Device Source Binding i.e. disable the use of DEVICE source globally
IdpCore.configure(false, activationCode, configs);
DeviceFingerprintSource fingerprintSource;

// The default fingerprint sources i.e. All available fingerprint sources
fingerprintSource = new DeviceFingerprintSource();

// Specifying explicitly the device fingerprint sources and custom data
fingerprintSource = new DeviceFingerprintSource(customData, Type.SOFT);
```

On iOS

iOS supports the following fingerprint types represented by the EMDeviceFingerprintType enum:

Table 3 Fingerprint Type Supported by iOS

Fingerprint Type	EMDeviceFingerprintType Value
DEVICE (replaced by DeviceSourceBinding flag)	NA
SOFT	EMDeviceFingerprintTypeSoft

Note:

- If DeviceSourceBinding flag is disabled, you can back up the fingerprint data and restore it to another device from an encrypted backup (iTunes, or through Clould Keychain.
- Data sealed with device fingerprint cannot be unlocked if the keychain for the application is deleted on the device.
- In the usage of OTP Token, since Device Fingerprint Sources usage is used and saved at the time when the token is provisioned, if the application have tokens provisioned in SDK 2.X with DEVICE source included or provisioned in SDK 3.X to 4.3, these tokens will not work after a backup/restore even if DeviceSourceBinding is disabled.

The following code snippet shows the usage of fingerprint type EMDeviceFingerprintTypeSoft:

```
// Device Source Binding i.e. disable the use of DEVICE source globally
[EMCore configureWithActicationCode: activationCode configurations:configs
useDeviceSourceBinding:NO];

EMDeviceFingerprintSource *fingerprintSource;
// The default fingerprint sources i.e. All available fingerprint sources
fingerprintSource = [EMDeviceFingerprintSource defaultDeviceFingerprintSource];
```

```
// Specifying explicitly the device fingerprint sources and custom data
fingerprintSource = [[EMDeviceFingerprintSource alloc]
       initWithCustomData:customFp
       deviceFingerprintType:[NSSet setWithObject:
                @(EMDeviceFingerprintTypeSoft)]];
```

Jailbreak and Root Detection

This feature detects if an application is running on a jailbroken (iOS) or rooted (Android) OS. An application can perform the detection in two ways:

- Calling an API – explained in the code below.
- Configuring a policy to apply for checks in the SDK. Documented in One-Time Password and Out-of-Band Communication.

The following code snippets are used to check if the OS is jailbroken/rooted. An application needs to take appropriate action according to the jaibreak status returned by SDK.

- On Android:

  ```
  RootStatus rootStatus = detector.getRootStatus();
  if (rootStatus == RootStatus.ROOTED) {
   // Exit, or display warning, etc.
  }
  ```

  The Android root detection API is also protected by the anti-hooking technique.

- On iOS:

  ```
  EMJailbreakStatus jailbreakStatus = EMJailbreakDetectorGetJailbreakStatus();
  if (jailbreakStatus == EMJailbreakStatusJailbroken) {
   // Exit, or display warning, etc.
  }
  ```

Note:

Do not wrap the function EMJailbreakDetectorGetJailbreakStatus() in an Objective-C method for convenience purposes, as it will undermine the security feature of this API.

Secure Container

Secure containers for common data types encapsulate sensitive data and are used extensively throughout Ezio Mobile SDK. It is recommended to use these APIs when handling sensitive data such as registration codes, passwords, and PIN. You can reducing the lifetime of these data in memory as much as possible by calling the wipe API. Each data type is instantiated via a factory.

The application must enforce these security guidelines: GEN12. Wiping assets.

Creating Secure Container Types

A factory must be used to create a secure container type. The API determines the type of secure container to be used.
The following example demonstrates the creating of a secure string from a byte array.
Other types of secure containers are also created in the similar manner.

- On Android:

```
// The second arg wipes the rawPassword data
SecureString password = core.getSecureContainerFactory()
         .fromByteArray(rawPassword, true);
```

- On iOS:

```
id<EMSecureContainerFactory> scf = [core secureContainerFactory];
// The second arg wipes the rawPassword data
id<EMSecureString> password = [scf secureStringFromData:rawPassword
              wipeSource:YES];
```

Wiping Secure Container Data

When data inside secure container is no longer needed, you can call the wipemethod to clear the data from the memory.

Note:

The data inside secure container will be invalid once its wipe method is called. Similarly, any references to the inner data of a secure container (that is, dataValue or stringValue) will also be invalid when the secure container has been wiped.

- On Android:

```
password.wipe();
```

- On iOS:

```
[password wipe];
```

Helpers

Helpers are general utilities used to aid in the development of an application.

Creating a RSA Public Key

This is a method to create a RSA public key type from raw RSA key components.

- On Android:

```
PublicKey key = Tools.generatePublicKeyFromByteArray(modulus, exponent);
```

- On iOS:
  Not applicable due to lack of a common public key type.

Getting UTC Time

A method to return the current UTC time in milliseconds. This allows an application to manually synchronize with the server if required (for example, operations based on the current time).

- On Android:

```
long utc = Tools.getUtc();
```

- On iOS:

```
unsigned long long utc = [EMOtpTools UTC];
```

Multi-Authentication

This module defines the feature set of various authentication services.
Different authentication modes are now available since Ezio Mobile SDK V4.0 such as PIN authentication and BioFingerprint authentication. These modes provide multiple ways to authenticate the users before accessing encrypted user assets such as when generating an OTP.
The authentication mode identifies their service and the related authentication input represents the input value. Modes can be used independently of each other, for example if the biometric authentication mode is active on a token, then either the PIN (active by default) or biometric can be used to authenticate the user.
The authentication module is the entry point to use the authentication services.

- On Android:

```
// Create AuthenticationModule. It's the entry point for all authentication
related
   // features.
AuthenticationModule authModule = AuthenticationModule.create();
```

- On iOS:

```
// Create EMAuthModule. It’s the entry point for all authentication related
// features.
EMAuthModule* authModule = [EMAuthModule authModule];
```

The application developers should particularly pay attention to two interfaces:

- AuthMode
  AuthMode interface represents a type of authentication. It is used to specify the authentication mode. It is used mainly for activation/deactivation operations, for example, `activateAuthMode(AuthMode)`.
- AuthInput
  AuthInput interface represents credentials that can be used for protected operations such as one-time password computation, for example, `getOtp(AuthInput)`. AuthInput is created from the respective authentication services by providing a user authentication such as user PIN or user biometric fingerprint.

Each authentication service has its own set of requirements on the usage. Refer to One-Time Password for detailed usage.

PIN Authentication

This module defines the feature for PIN authentication. PIN authentication mode is the primary authentication mode used by the SDK which authenticates against user knowledge (the PIN). It cannot be activated or deactivated since it is always enabled.

Note:

When the PIN authentication is active, it does not imply that the multi-authentication mode is enabled. To use different authentication modes, the token needs to be migrated to support the multi-authentication mode. The different authentication modes need to be activated before you can start using them.

- On Android:

```
PinAuthInput pinAuthInput = null;
try {
 AuthenticationModule authenticationModule = AuthenticationModule.create();
 PinAuthService pinAuthService = PinAuthService.create(authenticationModule);
 pinAuthInput = pinAuthService.createAuthInput(userPin);

 otpDevice.getOtp(pinAuthInput);
}
catch (IdpException e) {
...
} finally {
 // Wipe it immediately. If it is to be used multiple times
 // (consecutive time-based OTP), wipe it when application goes to background
 // e.g. onPause()
 pinAuthInput.wipe();
}
```

- On iOS:

```
EMAuthModule* authModule = [EMAuthModule authModule];
EMPinAuthService* pinAuthService = [EMPinAuthService
serviceWithModule:authModule];
id<EMPinAuthInput> pinAuthInput = [pinAuthService
createAuthInputWithData:userPin
error:&error];
if (error) {
    // Handle error
}
...
id<EMSecureString> otp = [otpDevice otpMode2WithAuthInput:pinAuthInput
error:&error];
if (error) {
 // Handle error
}
// Wipe it immediately. If it is to be used multiple times
// (consecutive time-based OTP), wipe it when application goes to background
// You can also put the line in a finally block to make sure it is wiped
[pinAuthInput wipe];
```

Biometric Authentication

This section defines the feature set for biometric authentications. Ezio Mobile SDK introduces functions that make use of these APIs to provide the biometric authentication that complements the PIN authentication services.

Application Requirements

Fingerprint Permission

For Android, Fingerprint Permission is required to use this feature in your app. Ensure that USE_FINGERPRINT permission \[ANDFP\] is added in the Android manifest file as follows:

```
<uses-permission android:name="android.permission.USE_FINGERPRINT"/>
This constant was deprecated in API level 28. Applications should request
USE_BIOMETRIC instead. But this permission is still required to support API level <
28.


<uses-permission android:name="android.permission.USE_BIOMETRIC"/>
Added in API level 28. Allows an app to use device supported biometric modalities.


Note: Both the permission works for API level 28 and above.
```

Biometrics Authentication Service Initialization

In order to access the biometrics functions, the BiometricsAuthService object must first be created.

- On Android:

```
// Create AuthenticationModule. It's the entry point for all authentication
related features.
AuthenticationModule authModule = AuthenticationModule.create();

// Create an object that represents fingerprint authentication service
BioFingerprintAuthService bioFPAuthSvc =
BioFingerprintAuthService.create(authModule);
```

- On iOS: Create AuthenticationModule. It's the entry point for all authentication related features.

```
// Create AuthenticationModule. It’s the entry point for all authentication
related features.
EMAuthModule* authModule = [EMAuthModule authModule];

// Create an object that represents fingerprint authentication service
EMBioFingerprintAuthService* bioFpAuthSvc = [EMBioFingerprintAuthService
serviceWithModule:authModule];//Deprecated in Ezio 4.6
```

Note:

Since Ezio Mobile SDK V4.6, `EMBioFingerprintAuthService` API is deprecated, it is replaced with two new APIs `SystemBioFingerprintAuth` and `SystemFaceAuth` in order to support new Apple Face ID.
Calling the `BioFingerprintAuthService` in the device that has Apple Face ID will trigger the Apple Face ID authentication.

```
// Create an object that represents fingerprint authentication service
EMSystemBioFingerprintAuthService* systemBioFpAuthSvc =
[EMSystemBioFingerprintAuthService serviceWithModule:authModule];

// Create an object that represents face authentication service
EMSystemFaceAuthService* systemFaceAuthSvc = [EMSystemFaceAuthService
serviceWithModule:authModule];
```

Device Checks

The following checks have to be performed before Ezio Mobile SDK is able to provide the biometric authentication.

1. Check the biometric sensor.
   The device must have a supported biometric sensor. The following code snippets are used to check if the device has a biometric sensor.

   – On Android:

   ```
   boolean fpSupported = bioFPAuthSvc.isSupported();
   ```

   – On iOS:

   ```
   BOOL fpSupported = [systemBioFpAuthSvc isSupported:&error];
   //or
   BOOL faceSupported = [systemFaceAuthSvc isSupported:&error];
   ```

2. Check the configuration of the biometric data.
   Device must have at least one biometric data registered on the device. The following code snippet is used to check if the device has one or more biometric data configured.

   – On Android:

   ```
   boolean fpConfigured = bioFPAuthSvc.isConfigured();
   ```

- On iOS:

```
BOOL fpCongfigured = [systemBioFpAuthSvc isConfigured:&error];
//or
BOOL fpCongfigured = [systemFaceAuthSvc isConfigured:&error];
```

The SDK will perform the preventive check internally for features that require biometric data regardless whether the checks have been done explicitly.

Upgrading to Multi-Authentication Mode

By default, tokens are authenticated via the PIN mode. If the device supports fingerprint/face(iOS) authentication, then the token is upgraded to support the multi-authentication mode using the token's PIN, before fingerprint/face(iOS) authentication mode can be activated and used.

Note:

The correct PIN must be presented when the token is upgraded to support multi-authentication mode. An OTP verification is recommended before the token upgrade.

The following code snippets demonstrate how to upgrade the token to multi-authentication mode on the different platforms:

- On Android:

```
try {
    // Create an object that represents pin authentication service.
    PinAuthService pinAuthService = PinAuthService.create(authModule);

    // Create an PinAuthInput object that represents pin authentication.
    // sPin should be the String PIN input obtained from the user.
    PinAuthInput pinAuthInput = pinAuthService.createAuthInput(sPin);

    // In order to activate fingerprint mode, we need to first upgrade the
token to // support multi auth mode.
    // Pass object that represents pin authentication functionality as
parameter

    // Upgrade token object.
    if (!token.isMultiAuthModeEnabled()) {
        token.upgradeToMultiAuthMode(pinAuthInput);
    }
}
catch (IdpException e) {
    // Handle exceptions flow here
}
```

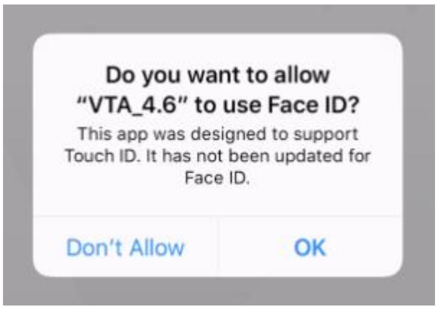
- On iOS:

```
// Create an object that represents pin authentication service.
EMPinAuthService* pinAuthService = [EMPinAuthService
serviceWithModule:authModule];
```

```
// Create an PinAuthInput object that represents pin authentication.
// sPin should be the String PIN input obtained from the user.
id<EMPinAuthInput> pinAuthInput = [pinAuthService createAuthInputWithData:sPin
error:&error];

if (error) {
    // Handle error
}
// Use the pinAuthInput to verify an OTP before continue here.
// In order to activate biometric mode, we need to first upgrade the token to
// support multi auth mode.
// Pass object that represents pin authentication functionality as parameter

// Upgrade token object.
if (![token isMultiAuthModeEnabled]) {
    BOOL result = [token upgradeToMultiAuthMode:pinAuthInput error:&error];
    if (!result) {
       // Handle error
    }
}
// Wipe the pin
[pinAuthInput wipe];
```

iPhone X Face ID permission

For iPhone X with iOS 11.X, Face ID permission alert is shown by iOS is as follows.



Do you want to allow "VTA_4.6" to use Face ID?

This app was designed to support Touch ID. It has not been updated for Face ID.

The message shown in this alert can be overridden by an app maker by including the `NSFaceIDUsageDescription` key in the app's `Info.plist` file and providing the custom or empty string for this key.
The default text for this key by OS is "This app was designed to use Touch ID and may not fully support Face ID".

Biometric Face Authentication Mode for iOS

Face Authentication Mode Activation Check

The following code snippets are used to check if the face authentication mode is activated.

- On iOS:

```
BOOL isActive = [token isAuthModeActive:[systemBioFaceAuthSvc authMode]];
```

Face Mode Activation

Once the multi-authentication mode has been enabled on the token, face authentication mode can be activated.
The following code snippets demonstrate the activation of face mode.

- On iOS:

```
// Check the mode is activated or not
if (![token isAuthModeActive:[systemBioFaceAuthSvc authMode]]) {
    // Activate face authentication mode
    // The token’s pin must be passed in order to activate face authentication
mode
    BOOL result = [token activateAuthMode:[systemBioFaceAuthSvc authMode]
usingActivatedInput:pinAuthInput error:&error];
    if (!result) {
        // Handle error
    }
}
```

Face Mode Deactivation

The following code snippets demonstrate the deactivation of face mode.

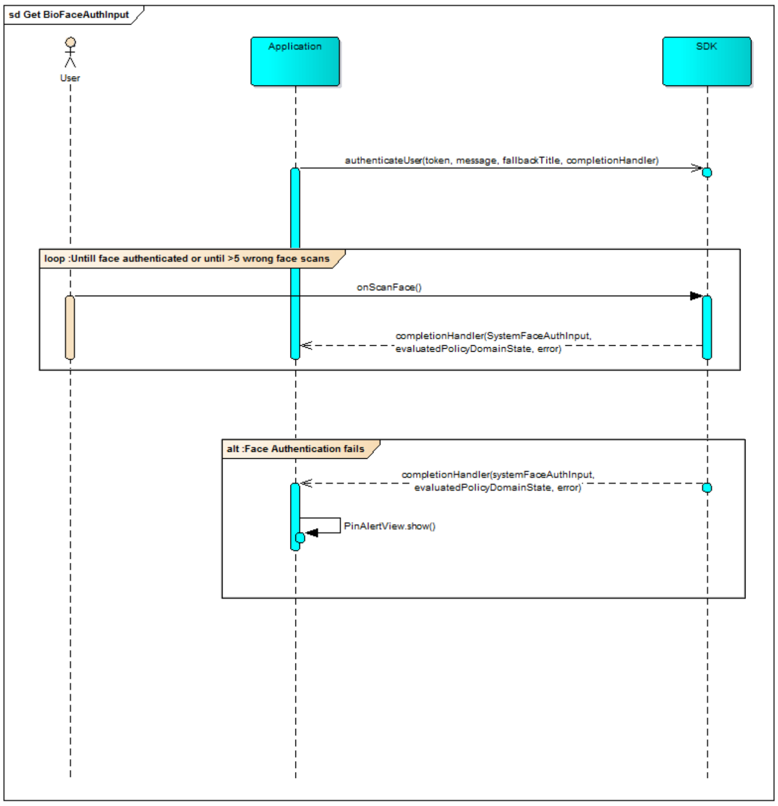
- On iOS:

```
// check if token is in multi auth mode & if face mode is active
if ([token isMultiAuthModeEnabled] &&
    [token isAuthModeActive:[systemBioFaceAuthSvc authMode]]) {
    BOOL result = [token deactivateAuthMode:[systemBioFaceAuthSvc authMode]
error:&error];
    if (!result) {
        // Handle error
    }
}
```

Using the Face Authentication

Get FaceAuthInput

Figure 6 Authentication with Biometric Face


sd Get BioFaceAuthInput
User
Application
SDK
authenticateUser(token, message, fallbackTitle, completionHandler)
loop :Untill face authenticated or until >5 wrong face scans
onScanFace()
completionHandler(SystemFaceAuthInput, evaluatedPolicyDomainState, error)
alt :Face Authentication fails
completionHandler(systemFaceAuthInput, evaluatedPolicyDomainState, error)
PinAlertView.show()

The first time a user is authenticated with a face id, the data will be stored in an encrypted cache. Refer to Cache Authentication State for a more detailed explanation.

The following code snippets demonstrate how to implement the face id authentication for the first time.

- On iOS:

```
EMSystemFaceAuthContainer* bioFaceContainer = [EMSystemFaceAuthContainer
containerWithNativeFaceAuthService:faceAuthSvc];

// Invoke method to start fingerprint authentication
// The message will be displayed on Touch Id
[bioFaceContainer authenticateUser:token
            withMessage:@"Authenticate User"
          fallbackTitle:@""
                 completionHandler:^(id<EMSystemFaceAuthInput> faceAuthInput,
NSData *evaluatedPolicyDomainState, NSError *error) {
                     //handle input and error
                 }];
```

Reusing SystemFaceAuthInput Object

By default, `SystemFaceAuthInput` has cache enabled such that when the application requires additional face id authentication, the user does not have to authenticate again (see Cache Authentication State).

The following code snippets demonstrate how it is done.

- On iOS:

```
id<EMOathDevice> otpDevice = [[[EMOathFactory alloc] init]
createSoftOathDeviceWithToken:token

settings:settings

error:&error];
if (error) {
    // Handle error
}

// You can get multiple otp with the same bioFpAuthInput, skipped the error
handling
id<EMSecureString> otp = [otpDevice hotpWithAuthInput:faceAuthInput
error:&error];
id<EMSecureString> otp2 = [otpDevice hotpWithAuthInput:faceAuthInput
error:&error];

[faceAuthInput wipe];
```

PIN Fallback

In the event that the face id authentication fails, users will still be able to obtain their OTP by using the fallback mechanism. The system will prompt the users to enter the device passcode. User can handle the fallback explicitly to use PIN authentication , so that the customized UI will pop up and the application PIN is used. The following codes demonstrate the usage of PIN fallback API:

```
EMSystemFaceAuthContainer* faceContainer = [EMSystemFaceAuthContainer
containerWithNativeFaceAuthService:faceAuthSvc];
// Authenticate user by Face ID with PIN fallback feature.
// By using this API, the application could customize the action upon error cases.
// This API only supports iOS 9 and above
BOOL isIos9AndAbove = [[UIDevice currentDevice].systemVersion intValue] >= 9;
if (isIos9AndAbove) {
    [faceContainer authenticateUser:token
                            withMessage:@"Authenticate User"
                          fallbackTitle:@"Enter PIN"
                      completionHandler:^(id<EMSystemFaceAuthInput> faceAuthInput,
NSData *evaluatedPolicyDomainState, NSError *error) {
                          if(error) {
                              // User pressed fallback button
                              if ([error code] ==
EM_STATUS_AUTHENTICATION_CANCELED_USER_FALLBACK) {
                                  // Handle fallback here (Provide customized UI for
PIN input etc).
                              }
                              // Handle other errors
                          } else {
                              // face authentication is successful
                              localFaceAuthInput = faceAuthInput;
                              // Compare evaluatedPolicyDomainState with previous
saved evaluatedPolicyDomainState
                              // to determine whether the database of authorized
face has been updated.
                              if (![evaluatedPolicyDomainState
isEqualToData:previousEvaluatedPolicDomainState]) {
                                  // Database of authorized face has been updated.
                              }
                          }
                      }];
}
```

Note:

- As this feature is only supported for iOS 9 and later, ensure that an OS version check is performed before using this API.
- An empty string can be provided as parameter fallbackTitle to turn off the PIN fallback feature

Supportability API

For devices whose TouchID or FaceID have been configured correctly by the user, `[EMSystemFaceAuthService isSupported]` works normally as expected.

However, if the biometric authentication is not configured, or when the user has denied access to this biometric mode, the underlying iOS Local Authentication does not provide information as to what biometric type is available on the device, hence the SDK will not be able to know whether the device supports TouchID, FaceID or none of them.

For devices released after 2017 (that is, iPhone X), the default scenario assumes that it will support native FaceID. However this may not be entirely accurate because some devices may support only TouchID but not FaceID.

In this case, this API allows the application to override this behaviour by implementing the `setIsSupportedFallback` block and determine the hardware support based on MachineID, a hardcoded string inherent on the hardware model. For details on the available machine IDs, refer to the “Hardware strings” value in https://en.wikipedia.org/wiki/List_of_iOS_devices.

```
[EMSystemFaceAuthService setIsSupportedFallback:^BOOL(NSString *machineID) {
    if ([machineID isEqualToString:@"iPhone11,3"]) {        //Hypothetical future
iPhone 11
        return YES;
    }else if(([machineID isEqualToString:@"iPhone11,6"])){  //Hypothetical future
iPhone 11
        return YES;
    }else{
        return NO;
    }
} error:&error];
```

Note:

The `fallback` block will only be invoked when the biometric is not configured and that the device machine ID is not recognized as of this writing (that is, devices later than 2017).

Backup/Restore

For iOS, after backup/restore, the biometric authentication mode will automatically be deactivated from all the tokens. The application developer has to provide a way for the user to reactivate the biometric authentication mode.

Face Authentication Callbacks

As callback mechanism is used in order to control and support the flow of face authentication, the error codes provided during the authentication are handled accordingly. The API returns the evaluated policy domain state of face ID which is used to determine whether the database of the authorised face has been updated. The following code snippet demonstrates on how this is done.

```
EMSystemFaceAuthContainer* faceContainer = [EMSystemFaceAuthContainer
containerWithNativeFaceAuthService:faceAuthSvc];
// Authenticate user by face ID with PIN fallback feature.
// By using this API, the application could customize the action upon error cases.
// This API only supports iOS 9 and above
BOOL isIos9AndAbove = [[UIDevice currentDevice].systemVersion intValue] >= 9;
if (isIos9AndAbove) {
   [faceContainer authenticateUser:token
                       withMessage:@"Authenticate User"
                     fallbackTitle:@"Enter PIN"
                 completionHandler:^(id<EMSystemFaceAuthInput> faceAuthInput, NSData
*evaluatedPolicyDomainState, NSError *error) {
                     if(error) {
                         // User pressed fallback button
                         if ([error code] ==
EM_STATUS_AUTHENTICATION_CANCELED_USER_FALLBACK) {
                             // Handle fallback here (Provide customized UI for PIN
input etc).
                         }
                         // Handle other errors
                     } else {
                         // face authentication is successful
                         localFaceAuthInput = faceAuthInput;
                         // Compare evaluatedPolicyDomainState with previous saved
evaluatedPolicyDomainState
                         // to determine whether the database of authorized face has
been updated.
                         if (![evaluatedPolicyDomainState
isEqualToData:previousEvaluatedPolicDomainState]) {
                             // Database of authorized face has been updated.
                         }
                     }
                 }];
}
```

Face Authentication Cache State

Ezio Mobile SDK provides a mechanism to cache or preserve the authentication state of the application. This way, the end users will not be prompted again for their face ID the next time an authentication is required, as long as the state is valid. The application developers should manage the lifetime of this authentication state securely. To invalidate this state, call wipeon the `EMSystemFaceAuthInput` object.

```
[faceAuthInput wipe];
```

Note:

To ensure the integrity of the data, wipe the face data immediately as soon as the application becomes inactive, that is, when it goes to the background mode, terminates and so on.

Face Authentication Error Codes

This section contains a list of error codes and their corresponding exceptions that may happen during the face authentication.The full list of error codes can be found in the `em_status.hclass`.

Table 4 Error Codes on iOS

Error Code	Description
810: EM_STATUS_AUTHENTICATION_FAILED	User cannot be authenticated.
811: EM_STATUS_AUTHENTICATION_CANCEL	User cancelled the face ID authentication.
812: EM_STATUS_AUTHENTICATION_ALREADY_REGISTERED	Bio-face mode is already activated for this token.
113: EM_STATUS_UNKNOWN_KEYCHAIN_ELEMENT	Unknown keychain item.
114: EM_STATUS_UNEXPECTED_KEYCHAIN_ERROR	An error occurs during the keychain operation.

For more information on UX Differences between iOS System FaceID and TouchID see UX Differences Between TouchID and FaceID (iOS).

Face Authentication Security Considerations

When using the biometric face service, the attack surface of the mobile application is not the same as that for the PIN. Templates and assets that are linked to user face are protected and stored locally, unlike the PIN which does not store any derived data. Therefore, risks such as theft threats or forensic analysis are not the same as when only PIN is used. Resistance of this authentication method relies only on the platform resistance.

Due to the Face ID feature on iOS, the device passcode has the same security role as the PIN for tokens. Anyone who knows the device passcode is able to enroll the face and generate a valid OTP with the tokens on which biometric face has previously been activated.

However, as biometric authentications are managed entirely by the system on iOS 8.X onwards, Ezio Mobile SDK on iOS does not detect the change in the biometric state. As for iOS9.X and later, `evaluatedPolicyDomainState` can be retrieved by the application to determine if there is a change in the biometric state.

Biometric Fingerprint Authentication Mode

Fingerprint Authentication Mode Activation Check

The following code snippets are used to check if the fingerprint mode is activated.

- On Android:

```
// get object that represents fingerprint authentication functionality
BioFingerprintAuthMode bioFPAuthMode = bioFPAuthSvc.getAuthMode();
boolean bIsAuthModeActive = token.isAuthModeActive(bioFPAuthMode);
```

- On iOS:

```
BOOL isActive = [token isAuthModeActive:[systemBioFpAuthSvc authMode]]'
```

Fingerprint Mode Activation

Once the multi-authentication mode has been enabled on the token, fingerprint authentication mode can be activated.

The following code snippets demonstrate the activation of fingerprint mode.

- On Android:

```
try {
    // Get object that represents fingerprint authentication functionality
    BioFingerprintAuthMode bioFPAuthMode = bioFPAuthSvc.getAuthMode();

    // Activate fingerprint authentication mode
    // The token's pin must be passed in order to activate fingerprint mode
    token.activateAuthMode(bioFPAuthMode, pinAuthInput);
}
catch (IdpException e) {
   // Handle exceptions flow here
}
```

- On iOS:

```
// Check the mode is activated or not
if (![token isAuthModeActive:[systemBioFpAuthSvc authMode]]) {
    // Activate fingerprint authentication mode
    // The token’s pin must be passed in order to activate fingerprint mode
    BOOL result = [token activateAuthMode:[systemBioFpAuthSvc authMode]
               usingActivatedInput:pinAuthInput
                     error:&error];
    if (!result) {
        // Handle error
    }
}
```

Fingerprint Mode Deactivation

The following code snippets demonstrate the deactivation of fingerprint mode.

- On Android:

```
// check if token is in multi auth mode & if fingerprint mode is active
if (token.isMultiAuthModeEnabled() &&
    token.isAuthModeActive(bioFPAuthMode)) {
    token.deactivateAuthMode(bioFPAuthMode);
}
```

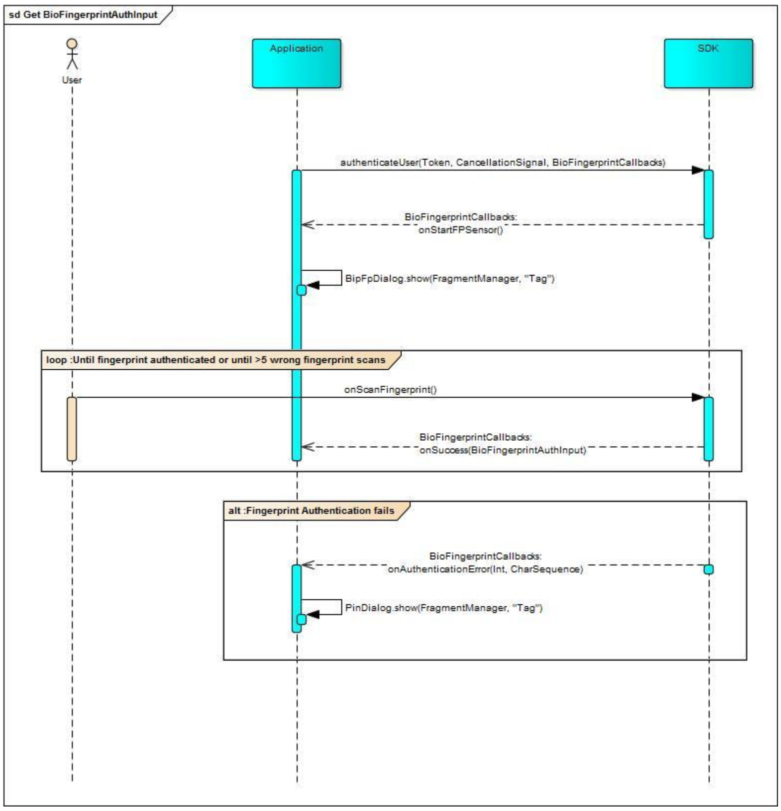
- On iOS:

```
// check if token is in multi auth mode & if fingerprint mode is active
if ([token isMultiAuthModeEnabled] &&
    [token isAuthModeActive:[systemBioFpAuthSvc authMode]]) {
    BOOL result = [token deactivateAuthMode:[systemBioFpAuthSvc authMode]
error:&error];
    if (!result) {
        // Handle error
    }
}
```

Using the Fingerprint Authentication

Get BioFingerprintAuthInput

Figure 7 Authentication with Biometric Fingerprint


sd Get BioFingerprintAuthInput
User
Application
SDK
authenticateUser(Token, CancellationSignal, BioFingerprintCallbacks)
BioFingerprintCallbacks:
onStartFPSensor()
BipFpDialog.show(FragmentManager, "Tag")
loop :Until fingerprint authenticated or until >5 wrong fingerprint scans
onScanFingerprint()
BioFingerprintCallbacks:
onSuccess(BioFingerprintAuthInput)
alt :Fingerprint Authentication fails
BioFingerprintCallbacks:
onAuthenticationError(Int, CharSequence)
PinDialog.show(FragmentManager, "Tag")

The first time a user is authenticated with a biometric fingerprint, the data will be stored in an encrypted cache. Refer to Cache Authentication State for a more detailed explanation.
The following code snippets demonstrate how to implement the fingerprint authentication for the first time.

- On Android:

```
try {
  BioFingerprintAuthMode bioFPAuthMode = bioFPAuthSvc.getAuthMode();
  cancellationSignal = new CancellationSignal();


  // check if fingerprint mode is activated
  if (token.isAuthModeActive(bioFPAuthMode)) {
    // Invoke method to start fingerprint authentication
    bioFPAuthSvc.getBioFingerprintContainer().authenticateUser(token,
cancellationSignal, bioFPCallbacks);
  }
}
catch (ActivationException e) {
  ...
}
catch (IdpException e) {
  ...
}
```

- On iOS:

```
EMSystemBioFingerprintContainer *bioFpContainer =
[EMSystemBioFingerprintContainer
containerWithBioFingerprintAuthService:systemBioFpAuthSvc];
// Invoke method to start fingerprint authentication
// The message will be displayed on Touch Id
[bioFpContainer authenticateUser:token
                     withMessage:message
                   fallbackTitle:@""
               completionHandler:^(id<EMSystemBioFingerprintAuthInput>
bioFpAuthInput, NSData *evaluatedPolicyDomainState, NSError *error) {
                      if (error) {
                        // Handle error
                      }
                  }];
```

Reusing BioFingerprintAuthInput Object

By default, BioFingerprintAuthInput has cache enabled such that when the application requires additional fingerprint authentication, the user does not have to authenticate again (see Cache Authentication State).
The following code snippets demonstrate how it is done.

- On Android:

```
// A device is the object that generates OTPs.
// Notice that in this snippet we will create OATH TOTP
GemaltoOathDevice device =
oathService.getFactory().createGemaltoOathDevice(oathToken);

// getTotp receives as parameter authInput object. In case of fingerprint
// authentication it should be object of BioFingerprintAuthInput type
SecureString otp = device.getTotp(bioFPAuthInput);
```

- On iOS:

```
id<EMOathDevice> otpDevice = [[[EMOathFactory alloc] init]
       createSoftOathDeviceWithToken:token
                        settings:settings
                   error:&error];
if (error) {
  // Handle error
}

// You can get multiple otp with the same bioFpAuthInput, skipped the error
handling
id<EMSecureString> otp = [otpDevice hotpWithAuthInput:bioFpAuthInput
error:&error];
id<EMSecureString> otp2 = [otpDevice hotpWithAuthInput:bioFpAuthInput
error:&error];

[bioFpAuthInput wipe];
```

PIN Fallback

In the event that the fingerprint authentication fails, users will still be able to obtain their OTP by using the PIN authentication, and the fallback will be handled accordingly.

For Android, the device passcode is not required by the system during a PIN fallback. However, the iOS system will prompt the users to enter the device passcode. Ezio Mobile SDK provides an API to overwrite this system behavior in which you can handle the fallback explicitly so that the customized UI will pop up and the application PIN is used. The following codes demonstrate the usage of PIN fallback API:

```
EMSystemBioFingerprintContainer* bioFpContainer = [EMSystemBioFingerprintContainer
containerWithBioFingerprintAuthService:systemBioFpAuthSvc];
// Authenticate user by Touch ID with PIN fallback feature.
// By using this API, the application could customize the action upon error cases.
// This API only supports iOS 9 and above
BOOL isIos9AndAbove = [[UIDevice currentDevice].systemVersion intValue] >= 9;
if (isIos9AndAbove) {
    [bioFpContainer authenticateUser:token
                         withMessage:@"Authenticate User"
                       fallbackTitle:@"Enter PIN"
                   completionHandler:^(id<EMSystemBioFingerprintAuthInput>
authInput,NSData *evaluatedPolicyDomainState, NSError *authError) {
                       if(authError) {
                           // User pressed fallback button
                           if ([authError code] ==
EM_STATUS_AUTHENTICATION_CANCELED_USER_FALLBACK) {
                                // Handle fallback here (Provide customized UI for
PIN input etc).
                           }
```

```
                               // Handle other errors
                      } else {
                           // Biofingerprint authentication is successful
                           localBioFpAuthInput = authInput;
                      }
                }];
}
```

Note:

- As this feature is only supported for iOS 9 and later, ensure that an OS version check is performed before using this API.
- An empty string can be provided as parameter `fallbackTitle` to turn off the PIN fallback feature

Backup/Restore
For iOS, after backup/restore, the biometric authentication mode will automatically be deactivated from all the tokens. The application developer has to provide a way for the user to reactivate the biometric authentication mode.

Fingerprint Authentication Callbacks (Android)

A callback mechanism is used to control and support the flow of fingerprint authentication. `BioFingerprintAuthenticationCallbacks` is implemented to handle the various control flows of the fingerprint sensor authentication. The `BioFingerprintAuthenticationCallbacks` abstract class extends the `FingerprintManager.AuthenticationCallback` class.
For more details, refer to the Android documentation FMAC. This document contains the following methods that are used for receiving information if the user passes the phone's (sensor) validation, and as well as to update the UI View.

- `public void onAuthenticationError(int errorCode, CharSequence errString)`
- `public void onAuthenticationHelp(int helpCode, CharSequence helpString)`
- `public void onAuthenticationSucceeded()`
- `public void onAuthenticationFailed()`

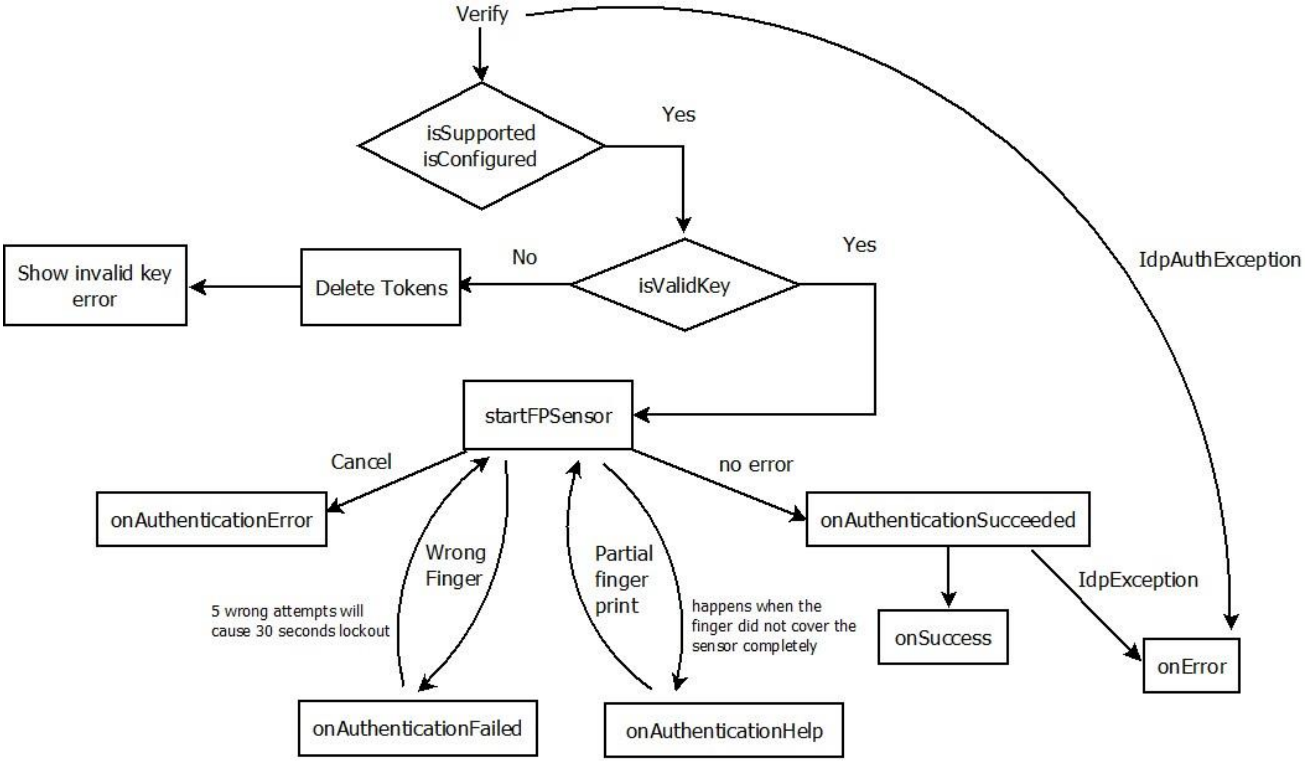
In order to control an internal flow (if the token data has been read correctly and all data that is necessary for OTP creation is available), the following methods callbacks class is extended in the BioFingerprintAuthenticationCallbacksclass.

```
public abstract class BioFingerprintAuthenticationCallbacks extends
FingerprintManager.AuthenticationCallback {
/ All objects were initialized correctly
public abstract void onSucceess(BioFingerprintAuthInput bioFPAuthInput);
// Called when fingerprint sensor starts working
public abstract void onStartFPSensor();
// Error was received - what went wrong during authentication (including
// cryptographic) data initialization and usage
public abstract void onError(IdpException exception);
```

Detailed explanation of when each callback method is triggered is shown in the flow diagram in the next section.

Lifecycle Callbacks

Figure 8 Lifecycle Callbacks



Implementing BioFingerprintAuthenticationCallbacks

The `BioFingerprintAuthenticationCallbacks` methods are implemented in order to receive the fingerprint data. The following code snippet demonstrates how this is done.

```
// BioFpFragment is a DialogFragment to prompt users to scan their fingerprint.
BioFpFragment bioFpFragment;
BioFingerprintAuthenticationCallbacks bioFPCallbacks = new
BioFingerprintAuthenticationCallbacks() {
    // Fingerprint authenticated successfully.
    // BioFingerprintAuthInput object will be passed here.
    @Override
    public void onSucceess(BioFingerprintAuthInput bioFingerprintAuthInput) {
        if (bioFpFragment != null) {
            bioFpFragment.dismiss();
        }

        // invoke method to get OTP using the fingerprint data
        getOTP(bioFingerprintAuthInput);
    }

    // The fingerprint sensor has been activated and is actively scanning
    // for fingerprint input.
    @Override
    public void onStartFPSensor() {
        // Display the DialogFragment to prompt user for fingerprint.
        bioFpFragment.show(getSupportFragmentManager(), "BioFPFragment");
    }
    @Override
    public void onError(IdpException e) {
        // Handle any error here.
    }

    @Override
    public void onAuthenticationError(int i, CharSequence charSequence) {
        // For the list of int code and it’s representation, please refer to the
        // official Android documentation.
        //
https://developer.android.com/reference/android/hardware/fingerprint/FingerprintMana
ger.html
    }

    @Override
    public void onAuthenticationHelp(int i, CharSequence charSequence) {
    }

    @Override
    public void onAuthenticationSucceeded() {
    }

    @Override
    public void onAuthenticationFailed() {
    }
};
```

Fingerprint Authentication Callbacks (iOS)

As callbacks mechanism is used in order to control and support the flow of fingerprint authentication, the error codes provided during the authentication are handled accordingly.

In Ezio Mobile SDK V4.1.1, the authenticate user API is enhanced by returning the Touch ID evaluated policy domain state. This feature is used to determine whether the database of the authorized fingerprints has been updated. The following code snippet demonstrates on how callbacks are handled.

```
EMSystemBioFingerprintContainer* bioFpContainer = [EMSystemBioFingerprintContainer
containerWithBioFingerprintAuthService:bioFpAuthSvc];

// Authenticate user by Touch ID with PIN fallback feature and have the ability to
know
// the database of authorized fingerprints state (evaluatedPolicyDomainState).
// By using this API, the application could customize the action upon error cases.
// This API only supports iOS 9 and above
BOOL isIos9AndAbove = [[UIDevice currentDevice].systemVersion intValue] >= 9;
if (isIos9AndAbove) {
    [bioFpContainer authenticateUserV2:token
                           withMessage:@"Authenticate User"
                         fallbackTitle:@"Enter PIN"
                     completionHandler:^(id<EMSystemBioFingerprintAuthInput>
authInput, NSData *evaluatedPolicyDomainState, NSError *authError) {
                         if(authError) {
                             // User pressed fallback button
                             if ([authError code] ==
EM_STATUS_AUTHENTICATION_CANCELED_USER_FALLBACK) {
```

```
                                 // Handle fallback here (Provide customized UI for
PIN input etc).
                             }

                             // Handle other errors
                         } else {
                             // Biofingerprint authentication is successful
                             localBioFpAuthInput = authInput;
                             // Compare evaluatedPolicyDomainState with previous
saved evaluatedPolicyDomainState
                             // to determine whether the database of authorized
fingerprints has been updated.
                             if (![evaluatedPolicyDomainState
isEqualToData:previousEvaluatedPolicDomainState]) {
                                 // Database of authorized fingerprints has been
updated.
                             }
                         }
                     }];
}
```

Fingerprint Authentication Cache Authentication State

Ezio Mobile SDK provides a mechanism to cache or preserve the authentication state of the application. This way, the end users will not be prompted again for their biometric fingerprints the next time an authentication is required, as long as the state is valid. The application developers should manage the lifetime of this authentication state securely. To invalidate this state, call `wipe` on the `BioFingerprintAuthInput` object.

- On Android:

```
bioFPAuthInput.wipe();
```

- On iOS:

```
[bioFPAuthInput wipe];
```

Note:

To ensure the integrity of the data, wipe the fingerprint data immediately as soon as the application becomes inactive, that is, when it goes to the background mode, terminates and so on.

Fingerprint Authentication Cancellation Signal (Android)

Misuse of the cancellation signal can cause an impact on the mobile device (such as temporary instability and even soft-resets). Therefore, it is mandatory to cancel each cancellation object, and ensure that the same object is not cancelled twice.

The following code snippet is recommended for handling a cancellation object:

```
...
...
```

```
// CancellationSignal object should be instantiated per authenticateUser call.
CancellationSignal cancellationSignal = new CancellationSignal();

if (token.isAuthModeActive(bioFPAuthMode)) {
    bioFPAuthSvc.getBioFingerprintContainer().authenticateUser(token,
cancellationSignal, bioFPCallbacks);
}
...
...

/**
 * BioFPFragmentDialog Callback Method
 * Handle dialog cancellation here
 */
@TargetApi(23)
@Override
public void onCancel() {
    if (cancellationSignal != null && !cancellationSignal.isCanceled()) {
        cancellationSignal.cancel();
    }
    cancellationSignal = null;
}
```

Fingerprint Authentication Error Codes

This section contains a list of error codes that may occur on Android and iOS platforms.

- On Android
  This section highlights a few selected error codes and their corresponding exceptions that may happen during the fingerprint authentication. The full list of error codes can be found in the `BioFingerprintResultCode` class.

Table 5 Error Codes on Android

Error Code	Description
6202: NO_FINGERPRINTS_REGISTERED	Indicates that fingerprint authentication cannot be used due to non-enrolled fingerprints.
6204: INVALID_KEY_EXCEPTION	Indicates invalid keys (invalid encoding, wrong length, and uninitialized). For more details, refer to the official Android, InvalidKeyException documentation.
6215: NO_CACHE	Indicates that no cache is created yet. To create a cache, the user has to first authenticate with a fingerprint before the encrypted fingerprint data can be used. Cache data will be wiped off on the closure of the application.
6216: PERMANENTLY_KEY_INVALIDATED_EXCEPTION	This code is used when `KeyPermanentlyInvalidatedException` is caught. `KeyPermanentlyInvalidatedException` is caught when the key can no longer be used because it has been permanently invalidated. This only occurs for keys which are authorized to be used only if the user has been authenticated. Such keys are permanently and irreversibly invalidated once the secure lock screen is disabled (that is, reconfigured to None, Swipe or other mode which does not authenticate the user) or when the secure lock screen is forcibly reset (for example, by Device Admin). Additionally, keys configured to require user authentication to take place for every key, are also permanently invalidated once a new fingerprint is enrolled or once no more fingerprints are enrolled. For more details, refer to the official KeyPermanentlyInvalidatedException documentation.

- iOS
  This section highlights a few selected error codes and their corresponding exceptions that may happen during the fingerprint authentication. The full list of error codes can be found in `em_status.h`.

Table 6 Error Codes on iOS

Error Code	Description
810: EM_STATUS_AUTHENTICATION_FAILED	User cannot be authenticated.
811: EM_STATUS_AUTHENTICATION_CANCEL	User cancels the Touch Id authentication.
812: EM_STATUS_AUTHENTICATION_ALREADY_REGISTERED	Bio-fingerprint mode is already activated for this token.
113: EM_STATUS_UNKNOWN_KEYCHAIN_ELEMENT	Unknown keychain item.
114: EM_STATUS_UNEXPECTED_KEYCHAIN_ERROR	An error occurs during the keychain operation.

For more information on UX Differences between iOS System FaceID and TouchID see UX Differences Between TouchID and FaceID (iOS).

Fingerprint Authentication Security Considerations

When using the biometric fingerprint service, the attack surface of the mobile application is not the same as that for the PIN. Templates and assets that are linked to user fingerprints are protected and stored locally, unlike the PIN which does not store any derived data. Therefore, the risks such as theft threats or forensic analysis are not the same as when only PIN is used. Resistance of this authentication method relies only on the platform resistance.

- On Android:
  Android M biometric fingerprint implementation does not differentiate between the enrolled fingerprints. Authentication to any of the enrolled fingerprints will allow the user to be correctly authenticated. For example, generating a valid one-time password.
  This may introduce new security risks as the newly enrolled biometric fingerprints may not belong to the same end user. Ezio Mobile SDK adds security to the biometric authentication by invalidating authenticated objects when the system's biofingerprint state is altered. When this happens, reactivating via PIN is required.

- On iOS:
  Due to Touch ID feature on iOS, the device passcode has the same security role as the PIN for tokens. Anyone who knows the device passcode is able to enroll its fingerprint and generate a valid OTP with the tokens on which biometric fingerprint has previously been activated.
  On iOS version 8.x and 9.x, several fingerprints can be enrolled but it is not possible to know which fingerprint has been used for the authentication. This is similar to Ezio Mobile SDK on Android. However, as authentications by TouchID are managed entirely by the system on iOS 8.X, Ezio Mobile SDK on iOS does not detect the change in the biofingerprint state. As for iOS 9.X and later, evaluatedPolicyDomainState can be retrieved by the application to determine if there is a change in the biofingerprint state.

One-Time Password

This module defines the feature set for one-time password (OTP). The module defines the basic types that are common across most OTP services. Each service defines and implements OTP algorithms. An OTP is consumed by a backend server and it typically authenticates a user or a transaction.

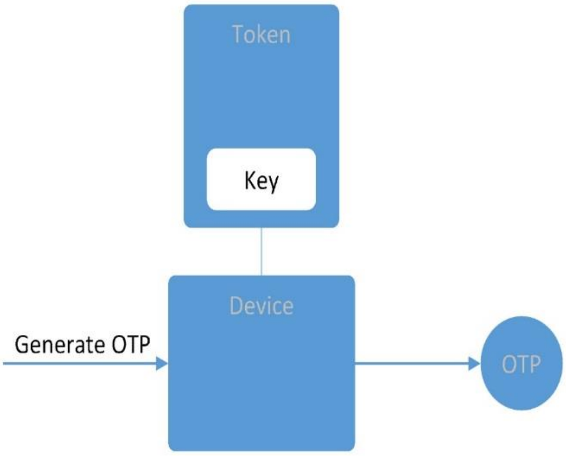
A common usage for OTPs is in online banking. A bank distributes tokens and devices to all its customers. Each token contains a secret key plus a state specific to a customer while each device uses the token's secret key to compute OTPs.

Note:

In many solutions, a token and a device are combined into a single form factor.

This module uses a naming scheme and data flow that models real world tokens and devices. Therefore, the flow is analogous to a user generating an OTP by the inserting of smart card token into a reader device.

Figure 9 Conceptual Relationship between Token, Device and OTP



Application Requirements

Configuration

An application must configure the core module with an OTP configuration. Otherwise instantiating the one-time password module will fail.

- On Android:

```
// Sample OTP configuration for
// Token policy set to ignore (allow the app to continue when device is rooted)
// Use the default configuration of PIN rule.
```

```
OtpConfiguration otpConfig = new OtpConfiguration.Builder()
                        .setRootPolicy(TokenRootPolicy.IGNORE)
                        .build();
IdpCore core = IdpCore.configure(otpConfig);
```

- On iOS:

```
// Build the configuration for the OTP module
EMOtpConfiguration *otpConfig = [EMOtpConfiguration
configurationWithJailbreakPolicy:EMTokenJailbreakPolicyIgnore];
EMCore *core = [EMCore configureWithConfigurations:[NSSet
setWithObjects:otpConfig,
                                    nil]];
```

User Inputs

As specified in User Interface, an application is responsible for displaying all UIs unless specifically noted otherwise. As the OTP module do not provide any UI, all the input data comes from an application's UI. This includes the input data such as the registration codes for provisioning, and OTP challenges.

Network

As specified in Network, in general an application is responsible for sending and receiving of data over a network connection. However, the OTP module handles all the network connections when creating the tokens (a.k.a provisioning). See Provisioning Configuration for more information.

Login to the Password Manager (Android Only)

An Android application must login via the Password Manager for most OTP related functions. This step is required due to the lack of a feature which is similar to iOS's keychain.

Tokens and Token Management

Token is the base type that represents a user's credentials, state and metadata examples of each are the secret key, counter, and identifier respectively) and Token Manager is the base type that specifies how to list, remove and retrieve the tokens. Most OTP services are built upon these base types to enhance static type checking. And there are optional types which provide additional operations that you can add on to. A token's credentials are used by a device type (defined by the relevant service) to generate OTPs.

Token Persistent Storage

Tokens are persistently stored in an application's private directory and great lengths have been taken to make it hard to read and reverse engineer. Therefore they cannot be accessed by any other applications regardless of whether this SDK is used.

Note:

There is no limitation imposed on the number of tokens that can be stored.

- Android Storage Locations
  The data is stored in the following locations within the application's installation directory:

  - The databases directory.
  - The internal files directory.
- iOS Storage Locations
  The data is stored in the app documents directory in (<APPHOME>/Documents) within the application's installation directory.

Provisioning Configuration

Provisioning is the process of creating a token and saving it to the mobile device. During a typical production, provisioning is done against a provisioning server, where the user's credentials which include the secret key and other supporting data are moved from the server to the device. In some cases such as during an early development phase, a token can also be created via tools instead of a real provisioning server.
A token which is created using a Token Manager is specific to each token type. Refer to the respective token's service documentation for additional behavior and limitations of its Token Manager. In general, the token provisioning process involves communicating with a provider (such as the server, and application) to provision the token's data.
The following sections show the provisioning configurations generated from the configuration builders.

EPS Configuration Builder

The EPS Configuration Builder directly communicates with the EPS server and manages the protocol and security of the communication. An application is responsible for providing the following:

- Registration Code
  A code produced by EPS during an user enrollment. Typically, this is provided by the user but it can be directly from a server depending on the design of the system.
- Mobile Provisioning Protocol Version
  The expected protocol version to use when communicating with the EPS. A service's Token Manager documents the versions it supports.

Table 7 MPP Versions

MPP Version	MPP Version Minimum EPS RSA Key Length
V1	1536 bits
V2	1536 bits
V3	2048 bits

- Server URL and Security Arguments
  The security arguments are related to the public key of the server.
- TLS configuration
  Specific aspects of the TLS protocol may be configured. By default, all insecure permits are disabled. The insecure permits could only be used in debug mode for test purpose. Using them in release mode of the SDK will generate an exception.

Warning:

Insecure permits of TLS Configuration should only be used on debug mode. Usage of insecure permits on release version of the SDK will generate an exception.

- Cipher Suites
  On Android, Ezio Mobile SDK limits to a list of TLS cipher suites which are considered secure. Both the client and the server must support one of the following ciphers listed in the table. There are no weak ciphers on iOS, as such no action is required on the server side.

Table 8 TLS Cipher Suites

RFC
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

RFC
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384

- HTTP Request Headers (Optional)
  Additional HTTP headers placed in the request are sent to EPS. There are no limits to the number of customized HTTP headers but for security reasons, the following headers are not allowed:

  Table 9 Prohibited Headers

Headers Prohibited in Customization			
Accept	Accept-Charset	Accept-Encoding	Accept-Datetime
Accept-Language	Cache-Control	Connection	Content-Length
Content-Type	Date	Expect	From
Host	If-Match	If-Modified-Since	If-None-Match
If-Range	If-Unmodified-Since	Max-Forwards	Origin
Pragma	Range	Referer	TE
Upgrade	Via	Warning	

The following security guidelines must be enforced during application development:

- GEN16: Use separate channels for sensitive data.
- GEN18: Ensure application origin.

The following codes demonstrates how to configure the EPS's TLS connection:

- On Android:

```
// Optional Step: Configure the EPS's TLS connection.
// There are two options configurable:
```

```
// (1) Timeout;
// (2) Permits (debug only).
// For example, the following connection will timeout after 1 second and
// will allow insecure HTTP URLs to be used.
// Those permits such as insecure connections must be used only in debug
// mode for test purpose and not allowed in production release.
TlsConfiguration tlsConf = new TlsConfiguration(1000,
    TlsConfiguration.Permit.INSECURE_CONNECTIONS);

// Create a map to contain the customized headers for the provisioning
Map < String, SecureString > headers = new HashMap < String, SecureString > ();
// Add all necessary customized headers into the map
headers.put("Session-Id",
    core.getSecureContainerFactory().fromString("123456"));
headers.put("Token-Id",
    core.getSecureContainerFactory().fromString("123456"));

// build Cap provisioning configuration for EPS provisioning
ProvisioningConfiguration provisioningConfig = new EpsConfigurationBuilder(
        core.getSecureContainerFactory().fromString(rc), epsAddress,
        MobileProvisioningProtocol.PROVISIONING_PROTOCOL_V1, rsaKeyId,
        rsaKeyExponent, rsaKeyModulus).setTlsConfiguration(tlsConf)
    .setProvisioningRequestHeaders(headers).build();
```

- On iOS:

```
EMProvisioningConfiguration* epsConfiguration = [EMProvisioningConfiguration
      epsConfigurationWithURL:epsAddress
          rsaKeyId:rsaKeyId
        rsaExponent:rsaKeyExponent
        rsaModulus:rsaKeyModulus
         registrationCode:regCode
   provisioningProtocol:EMMobileProvisioningProtocolVersion1
      optionalParameters:^(EMEpsConfigurationBuilder *builder) {
// configure optional parameters
// Optional Step: Configure the EPS's TLS connection with timeout and permits.
// For example, the following connection will timeout after 1 second
// and will allow insecure HTTP URLs to be used.
// Those permits must be used only in debug mode for test purpose and
// not allowed in production release.
builder.tlsConfiguration = [[EMTlsConfiguration alloc]
   initWithInsecureConnectionAllowed:YES
 selfSignedCertAllowed:NO
      hostnameMismatchAllowed:NO
         timeout:1];
    // Set customer headers
    NSMutableDictionary *headers = [NSMutableDictionary dictionary];
 [headers setObject:[[core secureContainerFactory]
secureStringFromString:@"123456"] forKey:@"Session-Id"];
 [headers setObject:[[core secureContainerFactory]
secureStringFromString:@"123456"] forKey:@"Token-Id"];
 builder.headers = headers;
}];
```

Dskpp Configuration Builder

- Configuration builder
  This provisioning configuration directly communicates with a Dskpp server and manages the protocol and security of the communication. An application is responsible for providing the following:

  – Dskpp provisioning protocol version
    The expected Dskpp provisioning protocol version to use when communicating with server.

    Note:

    DskppProvisioningProtocol.PROVISIONING_PROTOCOL_V1 only supports Dskpp provisioning base on Gemalto proprietary server SPA.

  – User credentials and url
    The user id, the enrollment password, and the server url. OR a unique base64 string containing these 3 parameters.Typically it is user provided but could come directly from a server depending on the design of the system.

  – Callback function
    The callback object to be called during provisioning.

  – DskppCallbackHandler (Android only)
    Defines the thread in which the dskpp callback will be executed. If it is null, then the callbacks will be called directly by the background thread handling the ongoing operation (from which the callback is triggered).

  – Dskpp TLS configuration
    Specific aspects of the TLS protocol may be configured. By default, all insecure permits are disabled. The unencrypted (http) connection permit INSECURE_CONNECTIONS could only be used in debug mode for test purpose. Using it in release mode of the SDK will generate an exception. Optionally Certificate list can be passed in the Dskpp TLS Configuration to enable pinning of public keys. The certificate list will be validated if Self signed certificate is permitted.

    Note:

    - INSECURE_CONNECTIONS permit of TLS Configuration should only be used on debug mode. Usage of insecure permits on release version of the SDK will generate an exception.
    - Certificate list is recommended to be set, if Permit.SELF_SIGNED_CERTIFICATES is passed.
    - Only OATH token is supported by dskpp provisioning protocol.

- On Android:

```
// build Dskpp provisioning configuration using server url, user id and
enrollment password
ProvisioningConfiguration provisioningConfig = new DskppConfigurationBuilder(
        serverUrl, userId, enrollmentPassword,
        provisioningProtocol, dskppCallback, handler, wipeUserIdPassword)
        .setTlsConfiguration(new DskppTlsConfiguration())
        .setApplicationName(appName)
        .setApplicationVersion(appVersion)
        .setParametersVersion(parametersVersion)
        .setCapability(capability)
        .setRootStatus(rootStatus)
        .setSecurityLevel(securityLevel)
        .setDeviceFriendlyName(deviceFriendlyName)
        .setDeviceFormFactor(deviceFormFactor)
        .setPushCapable(pushCapable)
        .setListToReceive(listToReceive)
        .setListToSend(listToSend)
        .build();
// Alternatively, the provisioning configuration can be build with base64
string
// build Dskpp provisioning configuration using base64 string
ProvisioningConfiguration provisioningConfigFromBase64String = new
DskppConfigurationBuilder(
        base64String, provisioningProtocol, dskppCallback, handler,
wipePinBase64String).build();
```

- On iOS:

```
// build Dskpp provisioning configuration using server url, user id and
enrollment
// password.
// Pass nil for optional parameters if there's nothing to set.
EMProvisioningConfiguration *dskppConfiguration = [EMProvisioningConfiguration
                 dskppConfigurationWithUrl:url
                                       uid:userId
                                  password:enrollmentPassword
                             dskppCallback:dskppCallback
                      provisioningProtocol:EMMobileProvisioningProtocolVersion3
               shouldWipeUserIdAndPassword:YES
                        optionalParameters:^(EMDskppConfigurationBuilder
*builder) {
builder.tlsConfiguration = [[EMDskppTlsConfiguration alloc] init];
builder.applicationName = _appName;
builder.applicationVersion = _appVersion;
builder.parametersVersion = _parameterVersion;
builder.capability = _capability;
builder.jailbreakStatus = _jailbroken;
builder.securityLevel = _securityLevel;
builder.deviceFriendlyName = _deviceFriendlyName;
builder.deviceFormFactor = _deviceFormFactor;
builder.pushCapable = _pushCapable;
builder.listToReceive = listToReceive;
builder.listToSend = listToSend;
}];
// Alternatively, the provisioning configuration can be built with base 64
string
// Build Dskpp provisioning configuration using base64 string
EMProvisioningConfiguration *dskppConfigurationBase64 =
[EMProvisioningConfiguration
              dskppConfigurationWithBase64:base64String
                    dskppCallback:dskppCallback
             provisioningProtocol:EMMobileProvisioningProtocolVersion3
```

```
                shouldWipeBase64Data:YES
                  optionalParameters:nil];
```

Warning:

For *deviceFriendlyName*, do not use specific value that can be linked to actual user

- Dskpp callback
  The results of the DSKPP Provisioning is passing through callback functions. Different callback functions are available to implement:
  - onServerPinRequested callback
    Triggered during provisioning if token Pin policy is set to SERVER_PIN. Application need to provide server pin and user pin to SDK. These 2 pins will be from the user.
  - onLocalPinRequested callback
    Triggered during provisioning if token Pin policy is set to USER_PIN or NO_PIN. Application need to provide user pin to SDK. When USER_PIN policy is used, ask for the PIN from the end user. This PIN will be used for OTP generation from the SDK. When NO_PIN policy is used, the pin is normally generate by the application and stored for the token. The PIN will be used for OTP generation from the SDK.
  - onPolicyStringProvided callback
    Triggered after completion of provisioning. Contains the policy string for the token.
  - onPolicyExtensionCallback callback
    Triggered when there are extensions to policies.
  - onKeyProvisioned callback
    Triggered when provisioning is finished. The key information is available for application in this function.
  - On Android:

```
DskppProvisioningCallback dskppCallback = new DskppProvisioningCallback()
{
    @Override
    public void onServerPinRequested(String tokenName, DskppPinPolicy
dskppPinPolicy, DskppServerPinCallback serverPinCallback) {
        //called when SERVER_PIN policy is used for the token
        PinAuthInput serverPin = null;
        PinAuthInput userPin = null;
        //create UI to get server pin and user pin from user
        //...
        //if pin entry is cancelled
        serverPinCallback.onPinEntryCancelled();
        //if pin is entered
        serverPinCallback.onPinEntered(serverPin, userPin);
    }
    @Override
    public void onLocalPinRequested(String tokenName, DskppPinPolicy
dskppPinPolicy, DskppLocalPinCallback localPinCallback) {
        //called when USER_PIN or NO_PIN policy is used for the token
        PinAuthInput userPin = null;
        // according to the pin policy, either create UI to get user pin
from user or get pin from application
        //...
        //if pin entry/generation is cancelled
        localPinCallback.onPinEntryCancelled();
        //if pin is entered/generated
        localPinCallback.onPinEntered(userPin);
    }
    @Override
    public void onPolicyStringProvided(String tokenName, SecureString
policyString) {
        //the policy string from server is available here for application
        //...
    }
    @Override
    public boolean onPolicyExtensionProvided(String tokenName,
SecureString XML, List<Pair<String,SecureString>> fields) {
        //called when there are extensions to policies
        //the xml string passed from this function contains the DSKPP
extensions supplied according to list to receive in Dskpp provisioning
configuration
        //...
        return true;
    }
    @Override
    public void onKeyProvisioned(String tokenName, DskppKeyInformation
keyInfo) {
        //the key info from server is available here for application
        //...
    }
};
```

– On iOS:

```
//Instantiate callback for DSKPP provisioning
EMDskppProvisioningCallback *callback = [[EMDskppProvisioningCallback
alloc] init];
callback.onLocalPinRequestBlock = ^(NSString *tokenName,
                                    EMDskppPinPolicy *dskppPinPolicy,
                                    id<EMDskppLocalPinCallback>
localPinCallback) {
    // called when USER_PIN or NO_PIN policy is used for the token


    id<EMPinAuthInput> userPin = nil;
    // according to the pin policy, either create UI to get user pin from
user or
    // from application
    //...

    // if pin entry/generation is cancelled
    [localPinCallback cancelLocalPinEntry];

    // if pin is entered/generated
    [localPinCallback localPinEntryCompletedWithPin:userPin];
};


callback.onServerRequestBlock = ^(NSString *tokenName,
                                 EMDskppPinPolicy *dskppPinPolicy,
                                 id<EMDskppServerPinCallback>
serverPinCallback) {
    //called when SERVER_PIN policy is used for the token

    id<EMPinAuthInput> serverPin = nil;
    id<EMPinAuthInput> userPin = nil;
    //create UI to get server pin and user pin from user
    //...

    //if pin entry is cancelled
    [serverPinCallback cancelPinEntries];

    //if pin is entered
    [serverPinCallback pinEntriesCompletedWithServerPin:serverPin
userPin:userPin];
};


callback.onPolicyStringProvidedBlock = ^(NSString *tokenName,
                                        id<EMSecureString>policyString) {
    //the policy string from server is available here for application
    //...
};


callback.onPolicyExtensionProvidedBlock = ^(NSString *tokenName,
                                           id<EMSecureString> XML,
                                           NSMutableDictionary *fields) {
    //called when there are extensions to policies
    //the xml string passed from this function contains the DSKPP
extensions supplied
    //according to list to receive in Dskpp provisioning configuration
    //...

    return YES;
};

callback.onKeyProvisionedBlock = ^(NSString *tokenName,
                                  EMDskppKeyInformation *keyInfo) {
    //the key info from server is available here for application
    //...
};
```

Offline Configuration Builder

This provisioning configuration enables an application to provision a token during the initial development when a working EPS is not yet available. Its purpose is exclusively used for testing and debugging purposes and should not be used in a production environment. See section Application Development Resourceson how to generate the input data, sample input data expected results, and usage.

- On Android:

```
// Setup the offline token characteristics, session key and provisioning
// response should have been generated by the host tool 'tokenbuilder'.
ProvisioningConfiguration offlineTokenConfig = new
OfflineTokenConfigurationBuilder(
 MobileProvisioningProtocol.PROVISIONING_PROTOCOL_V3,
 sessionKey, provisioningResponse).build();
```

- On iOS:

```
// Setup the offline token characteristics, session key and provisioning
// response should have been generated by the host tool 'tokenbuilder'.
EMProvisioningConfiguration *offlineTokenConfig = [EMProvisioningConfiguration
 offlineTokenConfigurationWithSessionKey:sessionKey
     withHmacKey:hmacKey
   provisioningResponse:provisioningResponse
   provisioningProtocol:EMMobileProvisioningProtocolVersion2];
```

Clear Text Secret Configuration Builder

This provisioning configuration enables an application to provision a token from a cleartext secret key. Its purpose is to allow arbitrary provisioning. Since the secret key is in cleartext format, an application is responsible for its security until the token is created.

- On Android:

```
/**
 * Initialize a new ClearTextSecreTokenConfiguration mandatory parameters
 *     pin and secret  * and optional parameter, userTokenId
 */
ProvisioningConfiguration clearTextSecretTokenConfig = new
ClearTextSecretTokenConfigurationBuilder(
     pinAuthInput, secret, true).setUserTokenId(userTokenId).build();
```

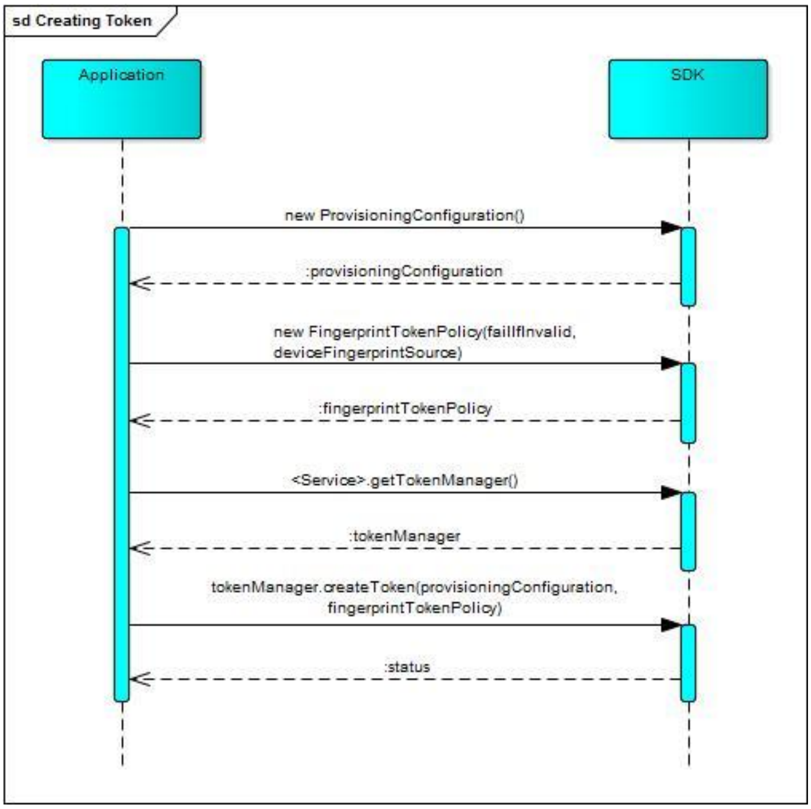
- On iOS:

```
/**
 * Initialize a new ClearTextSecreTokenConfiguration mandatory parameters
 *     pin and secret  * and optional parameter, userTokenId
 */
EMProvisioningConfiguration *clearTextConfig = [EMProvisioningConfiguration
       clearTextSeedImportConfigurationWithPin:pin
                                         secret:secret
                            optionalParameters:^
   (EMClearTextSecretTokenConfigurationBuilder *builder) {
     builder.userTokenId = userTokenId;
 }];
```

Creating Token (Provisioning)

To create a token, you have to specify the token name, the device fingerprint policy the device fingerprint data source of the token is bound to and its failure behavior, and the provisioning configuration.

Figure 10 Provisioning Sequence


sd Creating Token
Application
SDK
new ProvisioningConfiguration()
:provisioningConfiguration
new FingerprintTokenPolicy(failIfInvalid, deviceFingerprintSource)
:fingerprintTokenPolicy
<Service>.getTokenManager()
:tokenManager
tokenManager.createToken(provisioningConfiguration, fingerprintTokenPolicy)
:status

During the creation of a token, the Token Manager seals the token credentials with the configured fingerprint sources and policy for the anti-cloning mechanism.

Note:

Check that the configuration is made according to the intended behavior before sealing the token credentials. Once it is sealed, it cannot be changed.

Internally, the SDK maintains a digest of fingerprint sources for the token. During the access of these fingerprint sources, if there is a mismatch with the digest, the SDK will either generate an error or compute an incorrect OTP, depending on the policy. The digest is kept as small as possible to avoid having stop conditions. Therefore, there is 1 in 216 or 1 in 65536 chance of not detecting a mismatch.

Note:

The application must be able to reproduce the device fingerprint data it used to create the token. The SDK does not provide information about these sources. Therefore if the app has used the API that provides a custom fingerprint data during the creation, it must use the same API and pass the same custom data during the token access procedure to compute for OTP.

The token is created by a service-specific component TokenManager. The following example shows the usage of TokenManager with CAP service.

- On Android:

```
// Configure the device fingerprint to use service and soft data
// along with some custom data provided by the application as fingerprint data
DeviceFingerprintSource fingerprintSource =
         new DeviceFingerprintSource(customData, Type.SERVICE, Type.SOFT);
// Set the policy to generate an error when fingerprint is incorrect
DeviceFingerprintTokenPolicy fingerprintTokenPolicy =
         new DeviceFingerprintTokenPolicy(true, fingerprintSource);


// Sample token creation using CAP token
CapTokenManager capTokenManager = capService.getTokenManager();
try {
     CapToken token = capTokenManager.createToken(tokenName,
                                                  provisioningConfig,
                                                  fingerprintTokenPolicy); }
catch (PasswordManagerException e) {  // Need to login first, handle error }
catch (IdpException e) {  // handle error }
```

- On iOS:

```
// Configure the device fingerprint to not specify any fingerprint sources.
// In this case only the device information is used as fingerprint source,
// since it's enforced by SDK internally. Custom data defined by the
// application to be used as fingerprint data can also be passed here.
EMDeviceFingerprintSource *fingerprintSource = [[EMDeviceFingerprintSource
alloc]
                                                initWithCustomData:customFp];
// Set the policy to generate an error when fingerprint is incorrect
EMDeviceFingerprintTokenPolicy *fingerprintTokenPolicy =
        [[EMDeviceFingerprintTokenPolicy alloc]
                        initWithDeviceFingerprintSource:fingerprintSource
                                          failIfInvalid:YES];
// Sample token creation using CAP token
id<EMCapTokenManager> capTokenManager = [capService tokenManager:&error];
[capTokenManager createTokenWithName:tokenName
           provisioningConfiguration:provisioningConfig
        deviceFingerprintTokenPolicy:fingerprintTokenPolicy
                   completionHandler:^(id<EMCapToken> token, NSError *error) {
                       // do some logic with token or error
                   }];
```

Note:

- The provisioning process is not thread-safe as the SDK does not support concurrent provisioning, and this may lead to database error. To provision more than one token, ensure that the provisioning invocations are performed in sequential order.
- The token name is unique. Before calling `createToken`, check that the name of the token to be created does not exist.

Removing Token

Use the respective TokenManager to remove a token. Removing the token also removes all of the token's persistent data.

- On Android:

```
// remove a token
// If a token identified by the name is not of the same type as the token
manager
//    the token is not removed and returns false.
capTokenManager.removeToken(name);
```

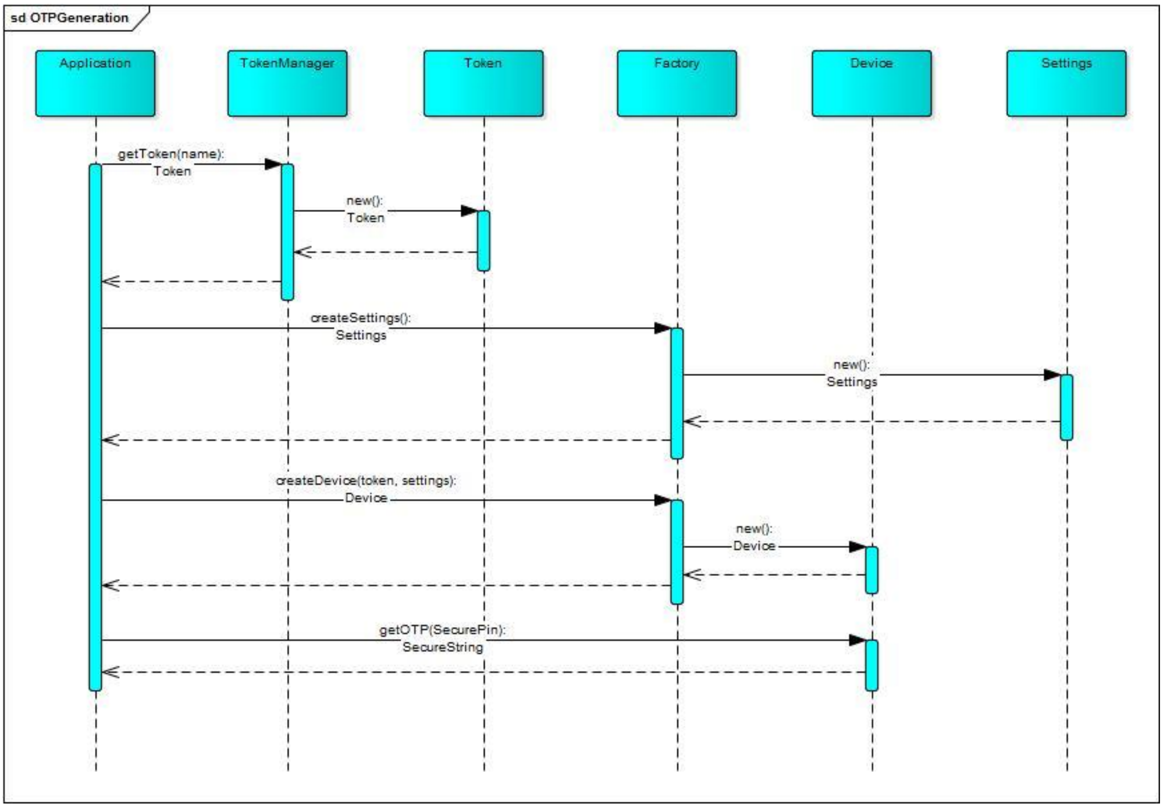
- On iOS:

```
// remove a token
// If a token identified by the name is not of the same type as the token
manager
//    the token is not removed and returns false.
[capTokenManager removeTokenWithName:name error:&error\];
```

OTP Generation

This section shows the general flow for the generation of OTP. For flows and code snippets specific to a service for a token type, refer to their respective services from Chip Authentication Program (CAP) Service to Gemalto Proprietary Formatting Dynamic Signature (GPF DS) Service.

Figure 11 Generating an OTP



- On Android:

```
CapTokenManager capTokenManager = capService.getTokenManager();

SecureString otp = null;
 try {
  // The customData must be provided if the token had been provisioned with
one.  CapToken token = capTokenManager.getToken(tokenName, customData);
  // Create the OTP device associated with the token type.
  // CAP device for this particular example.   // Alternatively, a variant with
custom settings can be used.
  CapDevice device = capService.getFactory().createCapDevice(token);

   // Generate OTP using appropriate API.
   otp = device.getOTPMode2(pinAuthInput);
 } catch (PasswordManagerException e) {
   // Need to login first, handle error
 } catch (IdpException e) {
   // handle error
}
```

- On iOS:

```
id<EMCapTokenManager> capTokenManager = [capService tokenManager:&error];

// The customData must be provided if the token had been provisioned with one.
```

```
id<EMCapToken> token = [capTokenManager tokenWithName:tokenName
                                fingerprintCustomData:customData
                                                error:&error];
// Create the OTP device associated with the token type.
// CAP device for this particular example.
// Alternatively, a variant with custom settings can be used
id<EMCapDevice> device = [[capService capFactory]
createCapDeviceWithToken:token

error:&error];
// Generate OTP using appropriate API.
id<EMSecureString> otp = [device otpMode2WithAuthInput:pinAuthInput
error:&error];
```

Changing the PIN

Ezio Mobile SDK provides an operation to allow you to change the user's PIN any time after the provisioning process. The SDK uses the same methods and mathematical calculations to generate an OTP regardless of whether the PIN is correct. Only the server will do the verification on whether the generated OTP is correct. In other words, the Ezio Mobile SDK does not distinguish between valid and invalid PINs. This eliminates offline brute force attacks on the PIN because generated OTPs are verified through the server. However, the PIN changing process must be implemented with care regarding the specific security concerns present in this process. Refer to *Ezio Mobile SDK V4.8 Security Guidelines* for more information.
The complete PIN change service is a multi-step process requiring interactions between four actors:

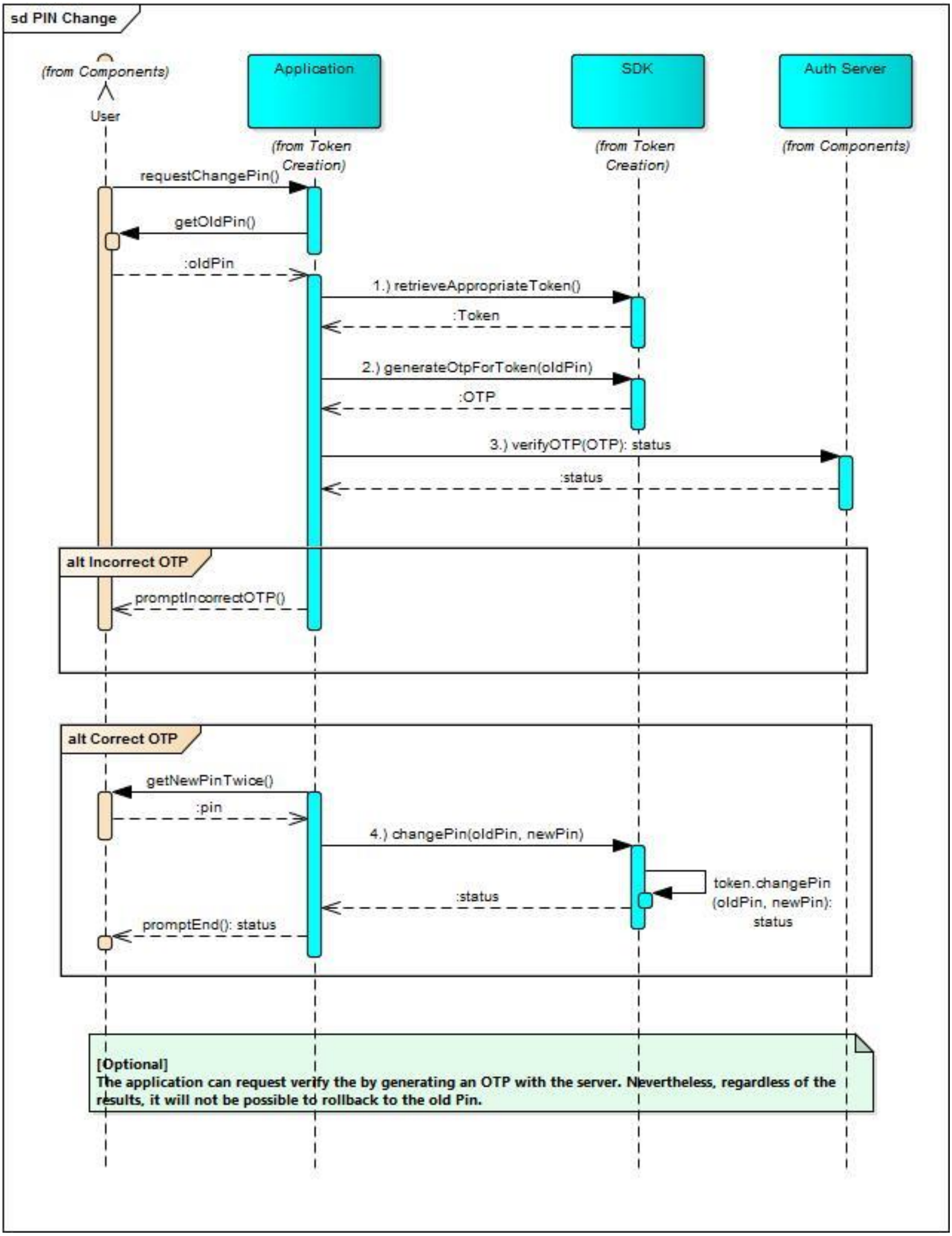
- The end user.
- The application.
- Ezio Mobile SDK
- The Ezio server.

For functional security requirements, the applications are to validate new PIN entries against the PIN rules to ensure that it is secure enough, prompting the user of inherent security risks if the PIN can be easily guessed. Furthermore, the only way to verify that the old PIN is correct is to generate an OTP using that PIN and validate against the Ezio server. To do so, the application needs to go through the complete OTP generation flow detailed in its respective token type. The complete PIN-change sequence diagram is presented in the following figure.

Note:

- The SDK is unable to determine whether the old PIN is valid. The application has to verify with the server by sending an OTP based on the old PIN. Changing the PIN with an incorrect old PIN results in an irrecoverable corruption of the token data.
- Neither the SDK nor the application stores any information about the user PIN.
- Do not store any PIN or derived information such as the PIN's digest value with salt as this exposes the PIN to an offline brute force attack and can ruin the security model of the solution.

Figure 12 PIN Change Sequence



[Optional]
The application can request verify the by generating an OTP with the server. Nevertheless, regardless of the results, it will not be possible to rollback to the old Pin.

- On Android:

```
// Change PIN
token.changePin(oldPin, newPin);
```

- On iOS:

```
// Change PIN
[token changePinWithAuthInput:oldPin newPin:newPin error:&error];
```

Chip Authentication Program (CAP) Service

This service is used to create OTPs based on the Mastercard's Chip Authentication Program (CAP) specification. In this service, a CAP Token is equivalent to an EMV smartcard while a CAP device is an equivalent to a card reader.
The following CAP modes and their basic purposes are provided as follows:

- Mode 1: Produces an OTP from a challenge and an optional amount and currency. Commonly used in a "buy" operation.
- Mode 2: Produces an OTP without any input data. Commonly used in a "code"operation.
- Mode TDS: Produces an OTP with arbitrary but strictly formatted input data. Commonly used in a "sign" operation.
- Mode 3: Produces an OTP from a challenge. Commonly used in a "login"operation.

Unconfigurable CAP Values

The initial value for CAP counter (ATC) of a new CAP Token is always 0 and cannot be configured.

CAP Tokens and Token Management

CAP introduces its own token and Token Manager types in order to enable the static type checking for CAP-specific operations, to add APIs for additional features, and to specify limitations. Refer to the following sections for details on the various usage.

Getting CAP Token Manager

The application retrieves an instance of the CAP Token Manager from its service.
The Token Manager only operates on CAP tokens and cannot be used to access other token types.

- On Android:

```
// Login the password manager
try {
   core.getPasswordManager().login();
} catch (PasswordManagerException e) {
 // handle error
}
// create OTP module
```

```
OtpModule otpModule = OtpModule.create();
// create CAP service
CapService capService = CapService.create(otpModule);
// Retrieve the token manager
CapTokenManager capTokenManager = capService.getTokenManager();
```

Note:

You have to login to the Password Manager (Android Only)

- On iOS:

```
// create OTP module
EMOtpModule *otpModule = [EMOtpModule otpModule];
// create CAP service
EMCapService *capService = [EMCapService serviceWithModule:otpModule];
// Retrieve the token manager
id<EMCapTokenManager> tokenManager = [capService tokenManager:&error];
```

Limitations of CAP Tokens

Clear Text Secret Configuration in Clear Text Secret Configuration Builder is not supported. Any attempt to use this configuration will result in a runtime error.

Generating CAP OTPs

This section describes how to use a typical token to create a device that generates CAP OTPs.
It is the responsibility of the application developers and system integrators to ensure that the correct settings and modes are used for the application.

CAP Device Settings

Application developper may choose to use either the default settings or custom CAP settings.
The former is done by simply creating a CAP device using a method without arguments for settings. The default settings for each CAP parameter can be found from the Android and iOS API docs.
The latter is demonstrated in the following.

Note:

The settings must be the same between the application and the backend authentication server.

- On Android:

```
// Device settings for CAP
SoftCapSettings capSettings = capService.getFactory().createSoftCapSettings();

// Set the Card risk management Data Object List 1 (CDOL)
capSettings.setCdol(new byte\[\]\{(byte) 0x9f, (byte) 0x02,
        (byte) 0x06, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x9f,
        (byte) 0x03, (byte) 0x06, (byte) 0x00, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x9f, (byte) 0x1a, (byte) 0x02, (byte) 0x00,
        (byte) 0x00, (byte) 0x95, (byte) 0x05, (byte) 0x80,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x5f, (byte) 0x2a, (byte) 0x02, (byte) 0x00,
        (byte) 0x00, (byte) 0x9a, (byte) 0x03, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x9c, (byte) 0x01,
        (byte) 0x00, (byte) 0x9f, (byte) 0x37, (byte) 0x04,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x82, (byte) 0x02, (byte) 0x10, (byte) 0x00,
        (byte) 0x9f, (byte) 0x36, (byte) 0x02, (byte) 0x00,
        (byte) 0x00, (byte) 0x9f, (byte) 0x52, (byte) 0x06,
        (byte) 0xa5, (byte) 0x00, (byte) 0x03, (byte) 0x04,
        (byte) 0x00, (byte) 0x00\});

// Set the Cryptographic Identifier (CID)
capSettings.setCid((byte) 0x80);

// Set the Issuer Application Data (IAD)
capSettings.setIad(new byte\[\]\{(byte) 0x00, (byte) 0x00,
        (byte) 0xA5, (byte) 0x00, (byte) 0x03, (byte) 0x04,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,\});

// Set Issuer Authentication Flags (IAF)
capSettings.setIaf((byte) 0x80);

// Set the Issuer Proprietary Bitmap (IPB)
capSettings.setIpb(new byte\[\]\{(byte) 0x00, (byte) 0x00,
        (byte) 0xFF, (byte) 0xFF, (byte) 0xFF, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00,\});

// Set the padding used when computing a CAP MAC
capSettings.setMacPadding(new byte\[\]\{(byte) 0x80, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00,
        (byte) 0x00, (byte) 0x00, (byte) 0x00\});

// Create cap device using token and cap settings
CapDevice device = capService.getFactory().createSoftCapDevice(token,
capSettings);
```

- On iOS:

```
// Device settings for Cap
id<EMMutableSoftCapSettings> capSettings = [capFactory mutableSoftCapSettings];

// Set the Card risk management Data Object List 1 (CDOL)
const unsigned char cdol[] =
 {
    0x9f, 0x02, 0x06, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x9f,
    0x03, 0x06, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x9f, 0x1a,
    0x02, 0x00, 0x00, 0x95, 0x05,
    0x80, 0x00, 0x00, 0x00, 0x00,
    0x5f, 0x2a, 0x02, 0x00, 0x00,
    0x9a, 0x03, 0x00, 0x00, 0x00,
    0x9c, 0x01, 0x00, 0x9f, 0x37,
    0x04, 0x00, 0x00, 0x00, 0x00,
    0x82, 0x02, 0x10, 0x00, 0x9f,
    0x36, 0x02, 0x00, 0x00, 0x9f,
    0x52, 0x06, 0xa5, 0x00, 0x03,
    0x04, 0x00, 0x00,
};
[capSettings setCdol:[NSData dataWithBytes:cdol length:sizeof(cdol)]];

// Set the Cryptographic Identifier (CID)
[capSettings setCid:0x80];

// Set the Issuer Application Data (IAD)
const unsigned char iad[] = {
    0x00, 0x00, 0xA5, 0x00, 0x03,
    0x04, 0x00, 0x00, 0x00, 0x00,
};
[capSettings setIad:\[NSData dataWithBytes:iad length:sizeof(iad)]];

// Set the Issuer Authentication Format (IAF)
[capSettings setIaf:0x80];

// Set the Issuer Proprietary Bitmap (IPB)
const unsigned char ipb[] = {
    0x00, 0x00, 0xFF, 0xFF, 0xFF,
    0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00,
    0x00,
};
[capSettings setIpb:\[NSData dataWithBytes:ipb length:sizeof(ipb)]];

// Set padding used when computing a CAP MAC
const unsigned char default_padding[] =
{
    0x80, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00
};
[capSettings setMacPadding:[NSData dataWithBytes:default_padding
                                          length:sizeof(default_padding)]];

// Create cap device using token and cap settings
id<EMCapDevice> device = [capFactory createSoftCapDeviceWithToken:token
                                                         settings:capSettings
                                                            error:&error];
```

CAP Modes

The CAP mode has to be determined at the point of generating an OTP.
An overview of the modes is provided in Chip Authentication Program (CAP) Service. The following example demonstrates how to generate an OTP using mode 1.

- On Android:

```
// Application request challenge to user, store it in 'challenge'
// ...

if (device.isMode1AmountRequired()) {
    // Application request amount from user, store it in 'amount'
    // ...
}

if (device.isMode1CurrencyRequired()) {
    // Application request currency from user, store it in 'currency'
    // ...
}

SecureString otp = device.getOtpMode1(authInput, challenge, amount, currency);
```

- On iOS:

```
// Application request challenge to user, store it in 'challenge'
// ...

if ([device isMode1AmountRequired\])
{
    // Application request amount from user, store it in 'amount'
    // ...
}

if ([device isMode1CurrencyRequired\])
{
    // Application request currency from user, store it in 'currency'
    // ...
}

id<EMSecureString> otp = [device otpMode1WithAuthInput:pin
                                             challenge:challenge
                                                amount:amount
                                          currencyCode:currencyCode
                                                 error:&error];
```

CAP Helpers

OTP Formatting

This transforms an OTP into groups of digits separated by spaces for display purposes in order to improve the end user's experience. Positions of spaces within the OTP value are described in *Ezio Mobile SDK V4.8 API Documentation*

- On Android:

```
SecureString formattedOtp = OtpTools.formatOtp(OtpTools.FormatType.CAP, otp);
```

- On iOS:

```
id<EMSecureString> formattedOtp = \[EMOtpTools
formatOtpWithType:EMFormatTypeCap otp:otp\];
```

Prefix Dynamic Signature with Token Sequence Number as Decimal Digits

Ezio Mobile SDK provides a function to prefix a token sequence number called GIDV in the previous versions of SDK (1.x). The application appends the token sequence number after a GpfDs (Gemalto Proprietary Format Dynamic Signature) operation.

Note:

Use this method only for tokens provisioned with ppv1.

- On Android:

```
// Add the token sequence number to the OTP, without left padding with zero
SecureString appendedOtp = CapTools.prefixSequenceNumberToOtp(otp, capToken);
```

- On iOS:

```
id<EMSecureString> appendedOtp = \[EMCapTools prefixSequenceNumberToOtp:otp
token: capToken];
```

OATH Service

An application uses this service to create OTPs based on the Open AutTHentication algorithms from RFC 4226 (HOTP RFC, 6238 TOTP RFC and 6287 OCRA RFC. An OATH Token contains the secret key and state while an OATH device generates OTPs using a token.
The OATH modes and their basic usage are provided as follows:

- HOTP – Produces an OTP from a shared secret and counter. This is commonly used for simple login OTPs. This services extends the RFC by supporting HMAC-SHA-256.

Note:

Ezio Mobile SDK also supports the unofficial HOTP with SHA-256 digest algorithm.

- TOTP – Produces an OTP from a shared secret and time. This is commonly used for simple login OTPs (for example, Facebook or Google's 2FA).
- OCRA – OATH Challenge-Response Algorithm. This is commonly used for complex OTPs such as challenge/response or transaction signing.

Note:

Since TOTP is a time-based passcode, it is sometimes possible for the server and client to drift (by having their clocks de-synchronized). This can lead to errors during the generation of OTPs, especially if the drift is significant. If the mobile device is set to automatically synchronize with a time server (which is usually the case), this time drift is not an issue since the device is not allowed to drift much. However, if the mobile device is not automatically synchronized with a time server then this may lead to an invalid generation of TOTP (for example, a user may have manually set his device time and let it drift away or set the device to a completely incorrect time).

OATH vs Gemalto OATH

The OATH specifications, in particular OCRA, are open-ended regarding their configuration and intended use cases. This allows many possible configurations and use cases but it also makes it harder to understand and implement. Gemalto helps to alleviate this problem by providing the standard OATH implementation and a Gemalto OATH specification that simplifies the configuration and specific use cases of OATH.

Unconfigurable OATH Values

These are the OATH values which are not configurable:

- The OATH counter for a new OATH token always starts from 0.
- The T0 value is always 0.

OATH Tokens and Token Management

OATH has its own token and Token Manager types to enable static type of checking for OATH-specific operations, adding APIs for additional features, and specifying limitations. Refer to the following sections for the details.

Getting OATH Token Manager

An application retrieves an instance of OathTokenManager from OathService. This token manager only operates on OATH tokens and cannot be used to access any other token type.

- On Android:

```
// Login the password manager
try {
     core.getPasswordManager().login();
} catch (PasswordManagerException e) {
     // handle error
}
```

```
// create OATH service
OathService oathService = OathService.create(otpModule);

// Retrieve the token manager
OathTokenManager oathTokenManager = oathService.getTokenManager();
```

Refer to Login to the Password Manager (Android Only) for more information.

- On iOS:

```
// create OATH service
EMOtpModule *otpModule = [EMOtpModule otpModule];
EMOathService *oathService = [EMOathService serviceWithModule:otpModule];

// Retrieve the token manager
id<EMOathTokenManager> tokenManager = [oathService tokenManager:&error];
```

Getting OATH Tokens

Using the OATH Token Manager, an application retrieves an OATH token using its name and optionally its custom fingerprint data. Refer to Device Fingerprintfor more information.

- On Android:

```
//get the token with the specified name
OathToken token1 = oathTokenManager.getToken(name);

//get the token with the specified name and custom fingerprint data
OathToken token2 = oathTokenManager.getToken(name, fingerPrintCustomData);
```

- On iOS:

```
//get the token with the specified name
id<EMOathToken> token1 = [tokenManager tokenWithName:tokenName error:&error];


//get the token with the specified name and custom fingerprint data
id<EMOathToken> token2 = [tokenManager tokenWithName:tokenName
                               fingerprintCustomData:customData error:&error];
```

Creating and Using Multi-Seed OATH Tokens

By default, tokens are used to generate OTPs from a single shared secret key. However, it is possible to create a token that generates OTPs from one of two selectable keys. OTPs are sent to an authentication server to authenticate the user's action. The key to use is selected by an application from the token's instance and the key to use is defined by the application (for example, use key 0 for low risk operations such as logins and key 1 for high risk operations such as signing transactions).

Note:

- The OATH service does not impose any restrictions on the usage of the keys.
- Do not use this capability on arbitrary provisioning requests. When creating a token with this capability, the provisioning response must be associated with a dual seed enrollment request.

This capability limits the provisioning response to the following settings:

Table 10 Limitations on Provisioning Response

Field	Limitation	Reason
Token Type	OATH only	Due to technical constraints, only OATH tokens are supported.
Provisioning Protocol	v3 only	The first version supporting OATH.
Key Size	32 bytes only	Dual seed tokens by design have two keys per provisioning response and provisioning protocol v3 supports a maximum size of 64 bytes for all keys. Therefore, individual key sizes are 32 bytes or smaller. **Note:** 20-byte keys are currently not supported.

This capability limits the usage of the token in the following ways:

Table 11 Limitations on Usage of Token

Limitation	Reason
No event based algorithms	Due to the technical constraints, Ezio Mobile SDK only supports one counter per token but dual seed tokens require two counters per token (one for each key). Therefore no event based algorithms are permitted.
Single PIN	A single PIN is used to protect the two keys and a PIN change operation applies to both keys.

- On Android:
  The following example shows how to create a multi-seed OATH token and how to select a key to be used for generating an OTP.

```
// Specify TokenCapability to be DUAL_SEED, also make sure that corresponding
// DUAL_SEED token was actually enrolled through EPS
oathTokenManager.createToken(name, epsConfiguration,
TokenCapability.DUAL_SEED);

DualSeedSoftOathToken token =
(DualSeedSoftOathToken)oathTokenManager.getToken(name);
// Required step before using 'token': select a key (e.g. the second key)
token.selectKey(1);
```

- On iOS:
  The following example shows how to create a multi-seed OATH token and how to select a key to be used for generating an OTP.

```
// Specify TokenCapability to be DUAL_SEED, also make sure that corresponding
// DUAL_SEED token was actually enrolled through EPS
[oathTokenManager createTokenWithName:tokenName
     provisioningConfiguration:provisioningConfig
              capability:EMTokenCapabilityDUAL_SEED
         completionHandler:^(id<EMOathToken> oathToken, NSError *error) {
        // do some logic with token or error

   id<EMDualSeedSoftOathToken> token = (id<EMDualSeedSoftOathToken>)
[oathTokenManager tokenWithName:tokenName error:&err];
   // Required step before using 'token': select a key (e.g. the second key)
   [token selectKeyIndex:1];
}];
```

Limitations of OATH Tokens

These are the limitations of the OATH tokens:

- Mobile Provisioning Protocol (MPP) versions 1 and 2 are not supported. These versions of the protocol do not support the OATH token type.
- Token key length must be greater than or equal to the hash algorithm when creating an OATH device. An exception is thrown when the key length fails to comply with the following algorithms:

Table 12 Available Algorithms

Key Length	Accepted Hash Algorithm(s)
20	HMAC-SHA1
32	HMAC-SHA1, HMAC-SHA256
64	HMAC-SHA1, HMAC-SHA256, HMAC-SHA512

Generating OATH OTPs

The earlier sections explain how to get an OATH token. This section describes how to use a token to create a fully configurable device that generates OATH OTPs in a generic way.

OATH Device Settings

You can either choose to use the default settings or custom OATH settings. The former is done by simply creating an OATH Device using a method that takes no settings argument. The latter is demonstrated as follows:

Note:

The settings have to be the same between the application and the backend authentication server.

- On Android:

```
SoftOathSettings settings = oathService.getFactory().createSoftOathSettings();


// Configure OCRA algorithm (first method)
settings.setOcraOtpLength(8);
settings.setOcraMaximumChallengeQuestionLength(8);
settings.setOcraCounterUsed(false);
settings.setOcraSessionLength(-1);
settings.setOcraChallengeQuestionFormat(SoftOathSettings.OcraChallengeQuestionF
ormat.ALPHANUMERIC);
settings.setOcraHashAlgorithm(SoftOathSettings.OathHashAlgorithm.SHA256);
settings.setOcraTimeSettings(
         SoftOathSettings.OathTimestepType.NONE, 0, 0);
settings.setOcraPasswordHashAlgorithm(SoftOathSettings.OcraPasswordHashAlgorith
m.NONE);


// ... add any customized settings as required


// Configure OCRA algorithm (second method)
settings.setOcraSuite(secureContainerFactory.fromString("OCRA-1:HOTP-SHA256-
8:QA08"));
```

- On iOS:

```
id<EMMutableSoftOathSettings> settings = [[oathService oathFactory]
                                          mutableSoftOathSettings];
// configure oath settings (1st method)
[settings setOcraOtpLength:8]; [settings
setOcraMaximumChallengeQuestionLength:8];
[settings setOcraCounterUsed:NO];
[settings
setOcraChallengeQuestionFormat:EMOcraChallengeQuestionFormatAlphanumeric];
[settings setOcraHashAlgorithm:EMOathHashSHA256];
[settings setOcraTimeSettingsWithTimeStep:EMOathTimestepTypeNone timestepSize:0
startTime:0];
[settings setOcraPasswordHashAlgorithm:EMOcraPasswordNone];


// configure oath settings (2nd method)
[settings setOcraSuite:[secureContainerFactory secureStringFromString:@"OCRA-
1:HOTP-SHA256-8:QA08"]];
```

HOTP, TOTP, OCRA Algorithms

The OATH operation to be used have to be determined for generating an OTP. An overview of the modes is provided in the beginning of chapter 8.6. This example demonstrates how to generate an OTP using the OCR algorithm.

- On Android:

```
SecureString otp = device.getOcraOtp(authInput, serverQuestion, null, null,
null);;
```

- On iOS:

```
id<EMSecureString> otp = [device ocraOtpWithAuthInput:pin
                                serverChallengeQuestion:serverQuestion
                                clientChallengeQuestion:nil
                                          passwordHash:nil
                                               session:nil
                                                 error:&error];
```

Generating Gemalto OATH OTPs

This section describes how to use a token to create a Gemalto-specific and configured device that generates OATH OTPs in specified ways.

The Gemalto OATH device provides a set of methods to easily implement services with most of the settings predefined:

- HOTP
- TOTP
- OCRA event-based challenge/response
- OCRA time-based challenge/response
- OCRA event-based signature

For these methods, the hash algorithm used to compute an OTP is always SHA-1 and it cannot be modified for this device.

Gemalto OATH Device Settings

You can choose to use either the default settings or custom Gemalto OATH settings. Only the OTP length and the maximum size of challenge can be changed as follows:

Table 13 Available Customization for Gemalto OATH Device

Parameters	Allowed Values
OTP Length ("x" in the OCRA-suite)	6 or 8
Maximum Challenge length ("nn" in the OCRA-suite)	6 or 8

The following table provides the parameter values and the corresponding OCRA-suite for each method.

Table 14 Parameters for Gemalto Device Signature Methods

Method	Challenge Format	Time Step	Ocra-suite
OCRA event-based challenge/response	Numerical	N/A	OCRA-1:HOTP-SHA1-x:C-QNnn
OCRA time-based challenge/response	Numerical	30s	OCRA-1:HOTP-SHA1-x:QNnn-T30S
OCRA event-based signature	Hexadecimal	N/A	OCRA-1:HOTP-SHA1-x:C-QH40
OCRA time-based signature	Hexadecimal	30s	OCRA-1:HOTP-SHA1-x:QH40-T30S

Note:

For Gemalto device signature methods:

- All data provided to the signature method are internally concatenated to a signing buffer using "~" as a separator. This signing buffer will be hashed using SHA-1 and the resulting value is used as the challenge in OCRA computation.
- "~" (0x7E) is a reserved character in the data and any containing "~" is rejected. The maximum challenge length is always of 40 hexadecimal nibbles.

HOTP, TOTP, Challenge/Response, Signature Methods

The Gemalto OATH operation to be used has to be determined for generating an OTP. An overview of the modes is provided in Gemalto OATH Device Settings.

The following example demonstrates how to generate an OTP using OCRA time-based challenge/response:

- On Android:

```
SecureString otp = device.getOcraTimeChallengeResponse(authInput, challenge);
```

- On iOS:

```
id<EMSecureString> otp = [device
ocraTimeChallengeResponseWithAuthInput:authInput
                                 challengeQuestion:challenge
                                             error:&error];
```

Generating Dynamic Code Verification (DCVX2)

Dynamic CVX2 is based on TOTP. A minor specification change is used to generate 3-4 digits OTP. The provisioning process is similar to OATH token provisioning. Typically dynamic CVX2 changes every 20 minutes. However it is made configurable for the application developer to set the expiry time of DCVX2.

Generating DCV and setting DCV OATH Device Settings

An application can choose to use either the default settings or custom DCV OATH settings. The former is done by simply creating a DCV OATH Device using a method that takes no settings argument. From DCV OATH device, DCV and the time expiration for DCV can be fetched.

Note:

The settings must be the same between the application and the backend authentication server.

- On Android:

```
SoftDcvOathSettings settings =
oathService.getFactory().createSoftDcvOathSettings();

// Configure DCV Setting
settings.setDcvHashAlgorithm(SoftOathSettings.OathHashAlgorithm.SHA1);
settings.setDcvTimeSetting(20, SoftOathSettings.OathTimestepType.MINUTES);
settings.setDcvStartTime(0);
settings.setDcvLength(3);

// Create DCV Oath device using dcv setting
DcvOathDevice device = oathService.getFactory().createSoftDcvOathDevice(token,
settings);

// Use the device created to get the DCV by passing pin input
SecureString dcv = device.getDcv(pinAuthInput);

// Expiration time for DCV can be retrieved by using the device
int time = device.getLastDcvLifespan();
```

- On iOS:

```
id<EMMutableSoftDcvOathSettings> settings = [oathFactory
mutableSoftDcvOathSettings];

// Configure DCV Settings
[settings setDcvHashAlgorithm:EMOathHashSHA1];
[settings setDcvTimeSettingsWithTimestepSize:20
timestepType:EMOathTimestepTypeMinutes];
[settings setDcvStartTime:0];
[settings setDcvLength:3];

// Create DCV Oath device using DCV settings
id<EMDcvOathDevice> device = [oathFactory
createSoftDcvOathDeviceWithToken:token
settings:settings error:nil];

// Use the device created to get the DCV by passing pin input
id<EMSecureString> dcv = [device dcvWithAuthInput:authInput error:nil];

// Expiration time for DCV can be retrieved by using the device
NSInteger time = [device lastDcvLifespan];
```

Dynamic Signature (DS) Formatting Service

This service is used to format the transaction data so that it can be signed by another OTP service. This is known as the dynamic signature which provides a mechanism for dynamic behavior that allows a bank to respond to threats in real-time. Unlike other OTP services, the DS Formatting Service itself does not generate any OTPs. However, its output is used by other OTP services to generate OTPs.

A DS Formatting Device computes a dynamic signature in either the unconnected or connected mode. For more information on the unconnected mode, refer to *Gemalto Dynamic Signature (DS) Specification 1.8* and *Gemalto Generic SWYS Specification 2.2* for the connected mode. If the former is used, then a challenge code generated by the bank is required to determine how to proceed with the operation. If the latter is used, then the bank is required to communicate directly to the mobile application on how to proceed with the operation.

For the unconnected mode, a challenge code is generated by the bank and enter into the mobile application by the user. Based on the challenge code, various predefined but customizable templates in this service can be triggered. A total of 24 templates are defined and are tailored to various situations. Each template contains primitives which represent dialogues to prompt the user to enter or accept specific information.

For the connected mode, an application is informed by a backend server the primitive to prompt the user with. Unlike in unconnected mode, templates are never used. Instead, the backend server sends the list of primitives to the application which then prompts the user to enter or accept the information specified. The application is responsible for all backend communication.

Understanding DS Template and Primitives

Before a dynamic signature is computed, the data to sign must be built. This section documents the basic types templates and primitives.

Templates

Every unconnected mode operation begins with a challenge generated by a server. This challenge may or may not encode a DS template that indicates the primitives to be used in order to compose the signature. If the challenge does not encode a template then the application does not display any dialogs to the user. If the challenge encodes a template, then the application will display one or more dialogs requesting information from the user. For example, a template that represents a payment request will request for the destination account number followed by the payment amount from the user.

Note:

Templates are not used in the connected mode.

Customization of Templates and Primitives

Ezio Mobile SDK comes preconfigured with a set of templates and primitives. For banks that use the default set of templates, no further customization is required (excluding the UI). However, if a bank requires a different set of templates and primitives then Ezio Mobile SDK can be customized via a JSON configuration. The exact content of JSON configuration is expected to be defined by the bank and not the application.

Note:

The predefined `PrimitiveTags` are only relevant if the default JSON file is used. If the JSON file is customized and the primitive tag values are altered or new primitive tag values are added, do not use this helper class.

- On Android:

```
// create DsFormatting service
DsFormattingService dsFormattingService =
DsFormattingService.create(otpModule);
// templates and primitives can be customized via a JSON configuration file.
final String CUSTOMIZED_DS_CONFIGURATION = "/dsConfiguration.json";
InputStream configuration = DsFormattingFactory.class
       .getResourceAsStream(DsFormattingFactory.DEFAULT_DS_CONFIGURATION);

dsFormattingService.getFactory().createDsDevice(token, configuration);
```

`DsFactoryprovides` a method `createDsDevice` that takes an `InputStream` that provides the customized DS configuration data.

- On iOS:

```
// create DsFormatting service
EMDsFormattingService *dsFormattingService = [EMDsFormattingService
serviceWithModule:otpModule];
EMDsFormattingFactory *dsFactory = [dsFormattingService dsFactory];

// templates and primitives can be customized via a JSON configuration file.
NSBundle *bundle = [NSBundle mainBundle];
NSString *path = [bundle pathForResource:@"dsConfiguration" ofType:@"json"];
```

```
id<EMDsFormattingDevice> device = [dsFactory createDsDeviceWithToken:token
dsConfigurationPath:path error:nil];
```

Where `EMDsFactory` provides a method `createDsDeviceWithToken:dsConfigurationPath:error:` that takes the path of a customized DS configuration file as input.

Primitives

Primitives represent information and data that is used to compute the dynamic signature. A primitive may request the user to confirm a predefined message, enter information such as an amount, or confirm the arbitrary text. For unconnected mode, the template provides a list of primitives to use for the operation. For the connected mode, the application receives a list of primitives from the bank.

Text Primitives in Connected Mode

The text primitives can only be used in connected mode. There are currently only two text primitives:

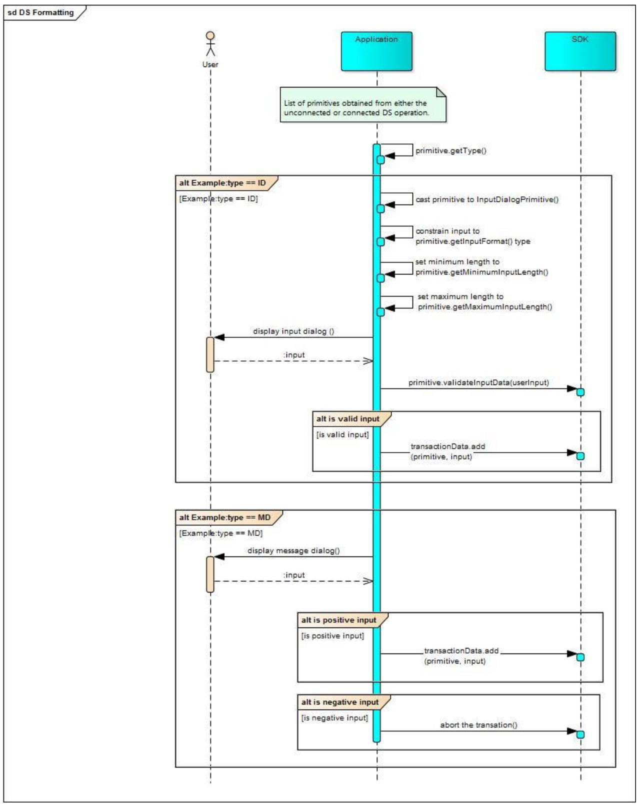
- Free Text
  Arbitrary text to display to the user before generating the signature. If the mobile application is required to be compliant with the signatures from dedicated hardware devices, the size of the text value has to match that of the hardware device. Currently there is no limit to the value size.
- Hidden Text
  Arbitrary binary value. For example, a hash-message that has no meaning to a user can be used to bind a transaction to a certain place/time but will still be included in the signing data.

Typically, all text primitive values are received directly from the backend server when the process begins.

Generating DS Transaction Data

Before a dynamic signature is computed, the data to sign must be built. This section documents how to generate the data to sign using one of the two modes. The following figure shows the steps that are common to both modes.

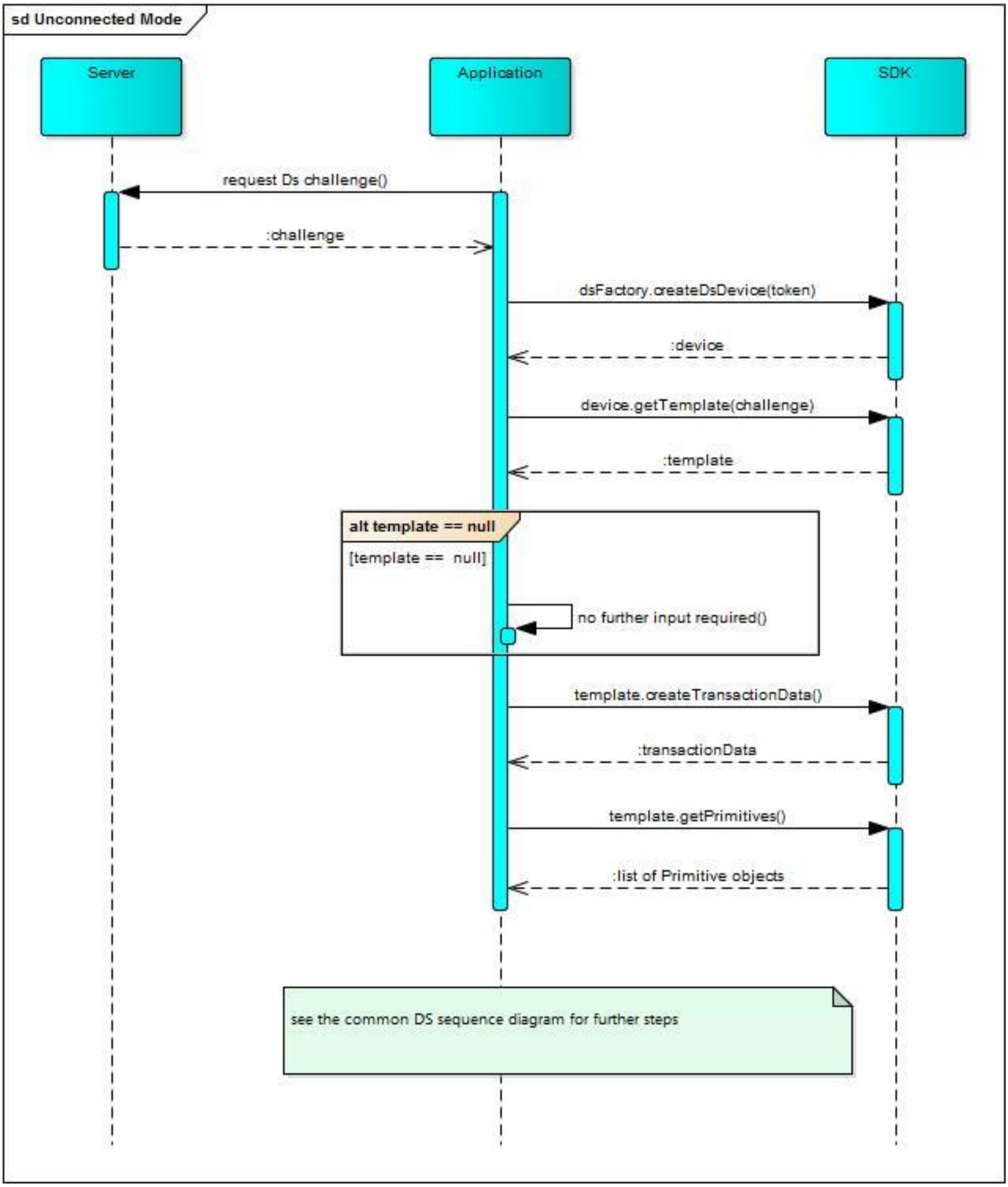
Figure 13 Common DS Transaction


sd DS Formatting
User
Application
SDK
List of primitives obtained from either the unconnected or connected DS operation.
primitive.getType()
alt Example:type == ID
[Example:type == ID]
cast primitive to InputDialogPrimitive()
constrain input to primitive.getInputFormat() type
set minimum length to primitive.getMinimumInputLength()
set maximum length to primitive.getMaximumInputLength()
display input dialog ()
:input
primitive.validateInputData(userInput)
alt is valid input
[is valid input]
transactionData.add (primitive, input)
alt Example:type == MD
[Example:type == MD]
display message dialog()
:input
alt is positive input
[is positive input]
transactionData.add (primitive, input)
alt is negative input
[is negative input]
abort the transation()

Unconnected Mode

The sequence diagram for the unconnected mode is presented as follows:

Figure 14 Unconnected Mode DS Transaction


sd Unconnected Mode
Server
Application
SDK
request Ds challenge()
:challenge
dsFactory.createDsDevice(token)
:device
device.getTemplate(challenge)
:template
alt template == null
[template == null]
no further input required()
template.createTransactionData()
:transactionData
template.getPrimitives()
:list of Primitive objects
see the common DS sequence diagram for further steps

- On Android:
  To determine the template to be used in the dynamic signature process, the challenge received from the server is entered into a DS device.

```
// Use the DsFactory to create a DsDevice.
DsFormattingDevice device =
dsFormattingService.getFactory().createDsDevice(token);


// Get the template
Template template = device.getTemplate(challenge);
```

If the template `is` null, then the application may skip the remaining code snippets and goes directly to one of the formatting options. The template contains questions and messages that are represented by the `Primitive` interface. A question is represented by the `InputPrimitive` interface and requires input from the user. `InputPrimitive` contains methods that describe the expected input and a method to validate input. `DsTransactionData` is a container for pairs of primitives and values. Each of the template's primitive must be added to the `DsTransactionData` object with the corresponding validated user input.

```
// Get the questions/messages included in this template
List<Primitive> primitives = template.getPrimitives();

// Create a new DS transaction data object.
DsTransactionData transactionData = template.createTransactionData();

for (Primitive prim : primitives) {
    PrimitiveType type = prim.getType();
    // Based on the primitive type different messages/questions
    // are presented to the user.
    if (type == PrimitiveType.ID) {
       InputDialogPrimitive idp = (InputDialogPrimitive) prim;

       // Get user input based on input format
       InputFormat oolean mat = idp.getInputFormat();
       int maxLen = idp.getMaximumInputLength();
       int minLen = idp.getMinimumInputLength();

        // Verify that the input is valid.
        // If not, the application must check formatting,
        // length, etc. Requirements are not broken.
        boolean valid = idp.validateInputData(userInput);

      if (valid) {
          // Add the primitive (idp) and it's input (userInput)
          // to the DS transaction data.
          transactionData.add(idp, userInput);
      }
    } else if (type == PrimitiveType.MD) {
      // ...
      // Abort the transaction if user gives negative response in
      // Message Dialog.
    } else {
        // ...
        // Handle other PrimitiveType values (e.g. Numeric Input
        // Dialog must display in appropriate format).
    }
}
```

The above code snippet shares the logic for each primitive type. `Primitive.getTag()` and `Primitive.getName()` can be used to develop functions for each individual primitive. An example is to display the appropriate message such as "Account number" for each primitive.

- On iOS:
  To determine the template to be used in the dynamic signature process, the challenge received from the server is entered into a DS device.

```
// Use the EMDsFactory to create a EMDsDevice.
id<EMDsDevice> device = [EMDsFactory createDsDeviceWithToken:token];


// Get the template
id<EMTemplate> template = [device templateFromChallenge:challenge];
```

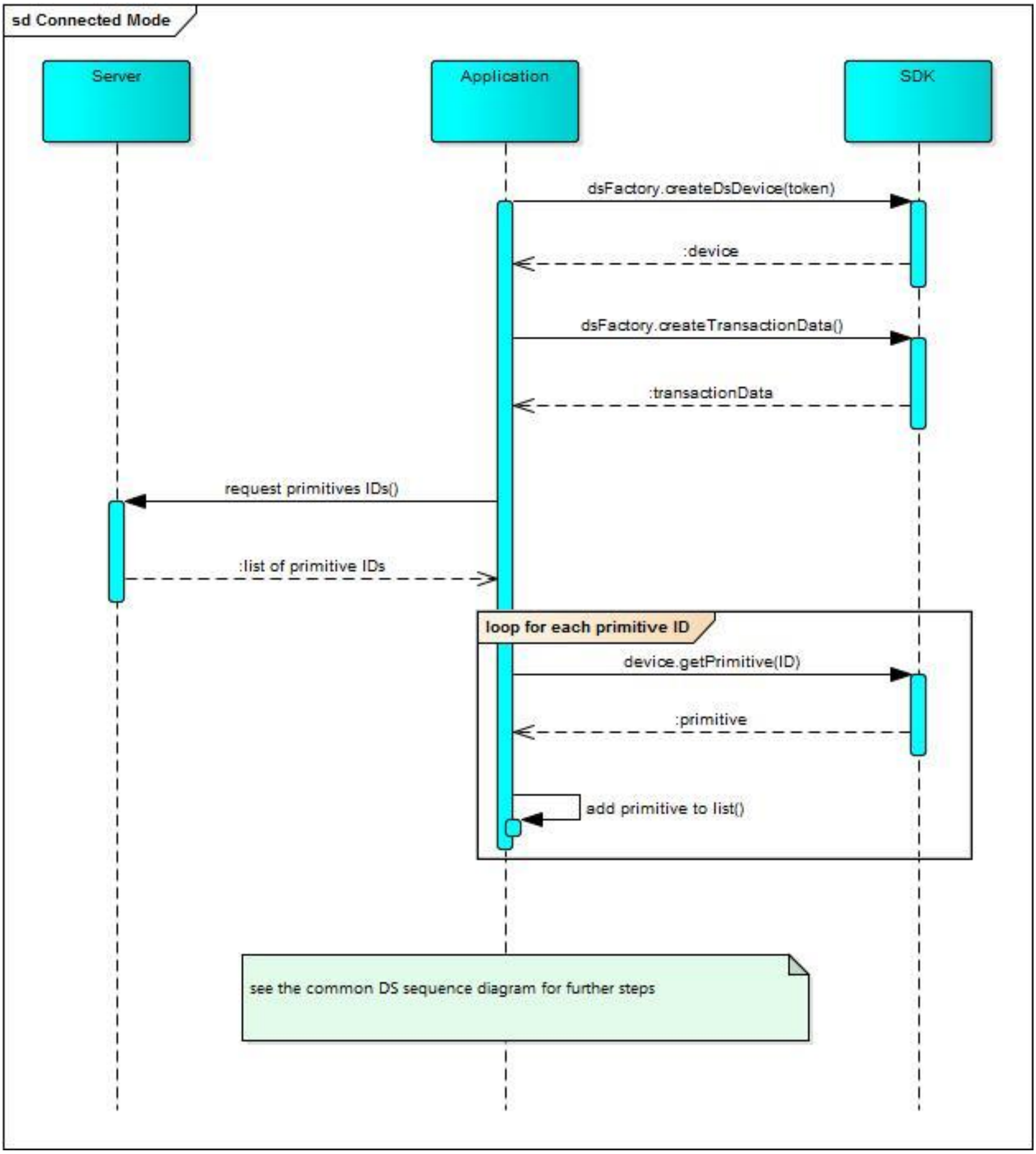
If the template is `nil`, then the application may skip the remaining code snippets and go directly to one of the formatting options. The template contains questions and messages that are represented by the `EMPrimitive` protocol. A question is represented by the `EMInputPrimitive` protocol and requires input from the user. `EMInputPrimitive` contains methods that describe the expected input and a method to validate the input. `EMDsTransactionData` is a container for pairs of primitives and values. Each of the template's primitive must be added to the `EMDsTransactionData` object with the corresponding validated user input.

```
// Get the questions/messages included in this template
NSArray *primitives = [template primitives];

// Create a new DS transaction data object.
id<EMDsTransactionData> transactionData = [template createTransactionData];

for(id<EMPrimitive> prim in primitives)
{
     EMPrimitiveType type = prim.primitiveType;

    // Based on the primitive type different messages/questions
    // are presented to the user.
    if (type == EMPrimitiveTypeID)
    {
       id<EMInputPrimitive> idp = (id<EMInputPrimitive>)prim;
       // Get user input based on input format
       EMPrimitiveInputFormat inputFormat = idp.inputFormat;
       NSInteger maxLen = idp.maximumInputLength;
       NSInteger minLen = idp.minimumInputLength;
       // Verify that the input is valid.
       // If not, the application must check formatting,
       // length, etc. requirements are not broken.
       BOOL valid = [idp validateInputData:userInput];


       if (valid)
       {
           // Add the primitive (idp) and it’s input (userInput)
           // to the DS transaction data.
           [transactionData addInputPrimitive:idp value:userInput];
         }
     }
     else if (type == EMPrimitiveTypeMD)
     {
         // Abort the transaction if user gives negative response in
         // Message Dialog.
      }
     else
     {
         // ...
         // Handle other EMPrimitiveType values (e.g. Numeric Input
         // Dialog must display in appropriate format).
     }
}
```

Connected Mode

Sequence diagram of the connected mode is presented as follows:

Figure 15 Connected Mode DS Transaction



- On Android:
  The application first receives the list of primitives directly from the bank since there is no template. The following code snippet assumes that the application receives the three primitive names. The first is an input primitive that takes a time value. The second is a text primitive that signs the text that the application displays to the user. The third is a message primitive which the application displays to the user. Each primitive is added to the transaction data with its corresponding value, if any, and then signed by one of the DS formatting modes presented in DS Formatting Helpers.

```
// Create a DSTransactionData without any template
DsTransactionData td = DsFactory.createDsTransactionData();


// Aggregate primitives and their value, if any, into the transaction data
td.add((InputPrimitive)device.getPrimitive("INPUT_TIME"), timeValue);
td.add((TextPrimitive)device.getPrimitive("FREE_TEXT"), freeTextValue);
td.add(device.getPrimitive("MESSAGE_SIGNING"));
```

- On iOS:
  The application first receives the list of primitives directly from the bank since there is no template. The following code snippet assumes that the application receives the three primitive names. The first is an input primitive that takes a time value. The second is a text primitive that signs the text that the application displays to the user. The third is a message primitive which the application displays to the user. Each primitive is added to the transaction data with its corresponding value, if any, and then signed by one of the DS formatting modes presented later in DS Formatting Helpers.

```
// Create a new DS transaction data object.
id<EMDsTransactionData> td = [EMDsFactory createDsTransactionData];


// Aggregate primitives and their value, if any, into the transaction data
id<EMInputPrimitive> inputTime =
      (id<EMInputPrimitive>) [device primitiveWithName:@"INPUT_TIME"];
[td addInputPrimitive:inputTime value:timeValue];


id<EMTextPrimitive> freeText =
      (id<EMTextPrimitive>) [device primitiveWithName:@"FREE_TEXT"];
[td addTextPrimitive:freeText value:freeTextValue];


[td addMessagePrimitive: (id<EMMessageDialogPrimitive>)
[device primitiveWithName:@"MESSAGE_SIGNING"]];
```

Target Formats

The service supports two types of dynamic signatures:

- Gemalto Proprietary Formatting (GPF) where signatures are generated via a GPF DS service.
- CAP Mode 2 TDS Formatting where signatures are generated via the CAP service.

These two types of dynamic signatures may either use the unconnected or connected mode to create the data that composes the signature. However, the forms differ by using different devices to generate the final signature.

Using Gemalto Proprietary Formatting

Gemalto Proprietary Formatting (GPF) uses the DS Transaction Data directly. No conversion is needed. See Gemalto Proprietary Formatting Dynamic Signature (GPF DS) Service for more details.

Using CAP Mode 2 TDS Formatting

Creating a CAP OTP requires the conversion of the DS Transaction Data into a CAP compatible form. When all primitives in the template have been added to the transaction data, the DS Formatting Device creates the CAP parameters from the challenge, DS domain and transaction data. Next, the parameters are fed into the CAP mode 2 TDS algorithm. Once computed, the OTP is passed on a backend server for authentication.

- On Android:

```
// Create the DS Formatting Device
DsFormattingService dsFormattingService =
DsFormattingService.create(otpModule);
DsFormattingFactory dsFactory = dsFormattingService.getFactory();
DsFormattingDevice dsFormattingDevice = dsFactory.createDsDevice(token);


// Transform the signature parameters and SIGN domain into
// CAP Mode 2 TDS parameters.
DsCapParameters capParams = dsFormattingDevice.getCapParameters(challenge,
         Template.DomainType.SIGN,
         DsCapParameters.DsCapMode.MODE2_TDS,
         transactionData);


// Use the CapFactory to create a CapDevice.
CapDevice capDevice = capService.getFactory().createCapDevice(token);


// Creates an OTP using the transaction data computed in 8.1 and transformed
// above into CAP parameters.
SecureString otp = capDevice.getOtpMode2Tds(pinAuthInput,
                                            capParams.getDataToSign());
```

- On iOS:

```
// Create the DS Formatting Device
EMDsFormattingService *dsFormattingService = [EMDsFormattingService
                                              serviceWithModule:otpModule];
EMDsFormattingFactory *dsFormattingFactory = [dsFormattingService
                                              dsFactory];
id<EMDsFormattingDevice> dsFormattingDevice = [dsFormattingFactory
                                    createDsDeviceWithToken:token
                                    error:&error];

// Transform the signature parameters and SIGN domain
// Into CAP Mode TDS parameters
EMDsCapParameters *capParameters = [dsFormattingDevice
                                    capParametersWithChallenge:challenge
                                    domain:EMDomainTypeSign
                                    dsCapMode:EMDsCapMode2Tds
transactionData:dsTransactionData];


// Create the cap device
EMCapService *capService = [EMCapService serviceWithModule:otpModule];
```

```
EMCapFactory *capFactory = [capService capFactory];
id<EMCapDevice> capDevice = [capFactory createCapDeviceWithToken:token
                                                           error:&error];
// Generate the signature
id<EMSecureString> otp = [capDevice otpMode2TdsWithAuthInput:pinAuthInput
                                                   dataToSign:[capParameters
                                                              dataToSign]
                                                        error:&error];
```

DS Formatting Helpers

Get Dynamic Signature template ID

Part of the dynamic signature process is the computing of the template ID from the challenge. From the template ID, the application can determine the questions to get the user to complete the dynamic signature transaction. Ezio Mobile SDKprovides a function that computes a template ID from a challenge.

- On Android:

```
try {
    int templateId =
DsFormattingTools.getDynamicSignatureTemplateId(challenge);
} catch (InvalidDigitChecksumException e) {
     //challenge have incorrect check digit
}
```

- On iOS:

```
NSInteger templateId = [EMDsFormattingTools
getDynamicSignatureTemplateIdWithChallenge:challenge];
```

Verify Challenge Check Digit

The challenge includes a checksum mechanism (depending on the server policy) in order to notify the user that a typing mistake is made. Ezio Mobile SDK provides a function that verifies the checksum for a given input.

- On Android:

```
if ( ! DsFormattingTools.checkDigit(DsFormattingTools.VerifyType.VERHOEFF,
challenge) ) {
    // Wrong checksum
    // ...
}
```

- On iOS:

```
if (![EMDsFormattingTools checkDigitWithType:EMVerifyTypeVerhoeff
challenge:challenge])
{
     // Wrong checksum...
}
```

Gemalto Proprietary Formatting Dynamic Signature (GPF DS) Service

This service is used to create dynamic signatures (OTPs) based on Gemalto's proprietary formatting algorithm. Unlike other services that generate OTPs, GPF DS does not define its own token type but instead uses the CAP tokens. Therefore, a CAP token can be used interchangeably to generate CAP OTPs or GPF dynamic signatures.

The GPF DS modes and their basic usage are provided as follows:

- Code mode which produces an OTP equivalent to CAP mode 2.
- Dynamic Signature mode which produces an OTP from DS Transaction Data.

Dependencies

This service is dependent on the following OTP services:

- CAP service
- DS Formatting Service

Generating Dynamic Signature OTPs

This section describes how to use a CAP token to create a GPF DS Device that generates OTPs in a generic way.

GPF DS Device Settings

Either the default settings or custom GPF DS settings can be used to generate dynamic signatures. This process is similar to the configuring of a CAP Device (see also Chip Authentication Program (CAP) Service). This service introduce an extra setting to specify the minimum OTP length.

Dynamic Signature Codes

The GPF DS operation to be used has to be determined before generating an OTP. In general, an application using a GPF DS Device is expected to generate an OTP using the Dynamic Signature operation. The following code snippet shows how to generate a DS OTP after processing the primitives associated with the transaction.

- On Android:

```
// create GpfDs service
GpfDsService gpfDsService = GpfDsService.create(otpModule);
// Use the GpfDsFactory to create a GpfDsDevice.  GpfDsDevice gpfDsDevice =
gpfDsService.getFactory().createGpfDsDevice(capToken);
```

```
// Creates a signature (OTP) in the SIGN domain using transaction data
SecureString otp = gpfDsDevice.getDynamicSignature(pinAuthInput, challenge,
Template.DomainType.SIGN, transactionData);
```

- On iOS:

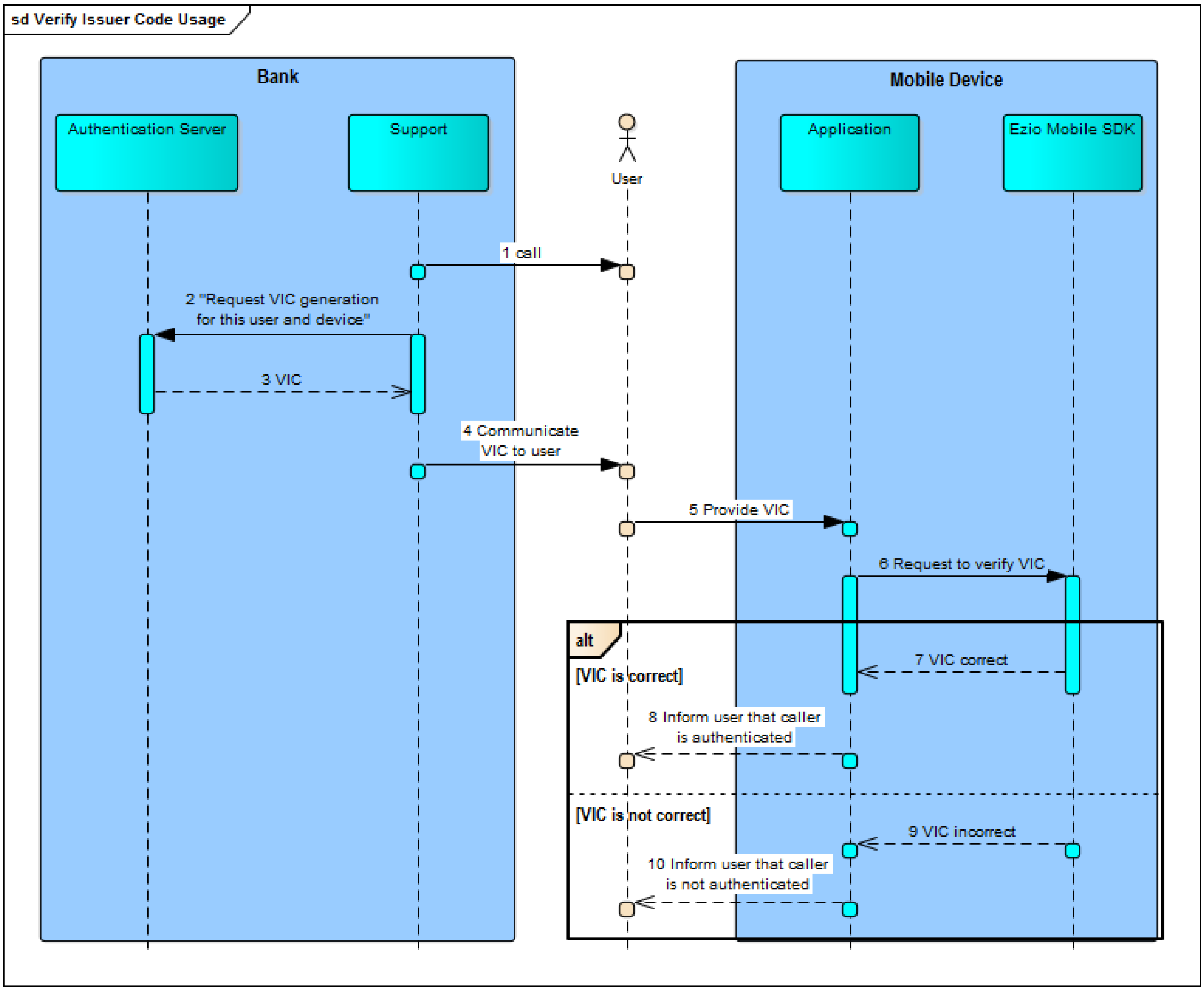
```
// create GpfDs service
EMGpfDsService *gpfDsService = [EMGpfDsService serviceWithModule:otpModule];

// Create GpfDs device
EMGpfDsFactory *factory = [gpfDsService gpfdsFactory];
id<EMGpfDsDevice> device = [factory createGpfDsDeviceWithToken:token
error:&error];

// Creates a signature (OTP) in the SIGN domain using transaction data
id<EMSecureString> otp = [device dynamicSignatureWithAuthInput:pinAuthInput
                                 challenge:challenge
                                 domain:domain
                                 transactionData:transactionData
                                 error:&error];
```

Verify Issuer Code (VIC) Service

The Verify Issuer Code (VIC) feature enables an application that uses the Ezio Mobile SDK to authenticate the server that holds end-user credentials.

The following high-level flow diagram describes how this feature can be used to authenticate a bank's support service in case a bank customer needs to be called. Not included in the diagram are usage details such as PIN entry and creation of VIC device in the Ezio Mobile SDK.

Figure 16 A High Level View of Verify Issuer Code Feature Usage


sd Verify Issuer Code Usage
Bank
Authentication Server
Support
User
Mobile Device
Application
Ezio Mobile SDK
1 call
2 "Request VIC generation for this user and device"
3 VIC
4 Communicate VIC to user
5 Provide VIC
6 Request to verify VIC
alt
[VIC is correct]
7 VIC correct
8 Inform user that caller is authenticated
[VIC is not correct]
9 VIC incorrect
10 Inform user that caller is not authenticated

1. Support service calls the end-user of the bank mobile service.
2. Support service can request the authentication server that supports VIC feature to generate a code for this end-user and device.
3. Authentication server provides VIC value to the support service.
4. Support service can communicate the VIC to the end-user.
5. End-user starts a verification process from its bank mobile application and provides the VIC value.
6. Application uses Ezio Mobile SDK to check VIC value.
7. If the VIC is correct, Ezio Mobile SDK informs the application that the code was correct.
8. If the VIC is correct, the application informs the end-user that the caller is authenticated.
9. If the VIC is incorrect, Ezio Mobile SDK informs the application that the code is incorrect.
10. If the VIC is incorrect, the application informs the end-user that the caller is not authenticated.

VIC Feature Usage

Activation

By default the feature is not enabled in Ezio Mobile SDK. Application needs to provide an activation code when establishing the Ezio Mobile context to be allowed to use it.
Only Gemalto can provide activation codes.You must request an activation code from your Gemalto representative.

VIC Characteristics

The VIC feature uses event-based OTP computation. The value is always 8 digits long, first two digits are the server counter value, the remaining 6 digits are the OTP digits.
During VIC verification, the Ezio Mobile SDK acts as an authentication component that checks an OTP value. Internally, each token with VIC capability enabled has two specific counters:

- A try counter (VICTC) that is decreased each time a wrong VIC is used. Once VICTC reaches zero, the feature is blocked for this token. This means that there is no more way to unblock the feature. VICTC is reset to "10" each time a correct VIC value is checked. When the token is created, the VICTC is initialized to "10".
- A VIC Application Transaction Counter (VICATC), which is a monotonic counter, is initialized at zero with a maximum value of 99 and must be compliant with the server that has generated the VIC value:
  - If VICATC is greater or equal to the one of the server, then the VIC value will be rejected (VICTC is decremented also in this case).
  - If VICATC is strictly lower than the one of the server, then the VIC value will be checked. In case the VIC value is correct, VICATC will be updated to the server’s value. Due to this update mechanism, the VICATC may not have a regular progression and less than 99 verifications can be allowed. Each time the server generates a VIC value, its internal counter is incremented by one. If several VICs are generated by the server, and the last one is used for a successful verification, the Ezio Mobile SDK will resynchronize its VICATC by more than one increment.

VIC Tokens and Token Management

VIC introduces its own token and Token Manager types. Their main purpose is to enable static type checking for VIC specific operations, add APIs for additional features, and specify limitations. The following subsections describe these in detail.

Getting VIC Token Manager

An application retrieves an instance of the Token Manager from its service.This Token Manager only operates on VIC tokens and cannot be used to access any other token types.

- On Android

```
// create VIC service
VicService vicService = VicService.create(otpModule);
// Retrieve the token manager
VicTokenManager vicTokenManager = vicService.getTokenManager();
```

- On iOS

```
// create VIC service
EMVicService* vicService = [EMVicService serviceWithModule:otpModule];
// Retrieve the token manager
id<EMVicTokenManager> vicTokenManager = [vicService tokenManager:&error];
```

Getting VIC Tokens

Using the VIC Token Manager, an application retrieves a VIC token by using its name and optionally its custom fingerprint data.

- On Android

```
//get the token with the specified name
SoftVicToken token1 = vicTokenManager.getToken(name);
//get the token with the specified name and custom fingerprint data
SoftVicToken token2 = vicTokenManager.getToken(name, fingerPrintCustomData);
```

- On iOS

```
//get the token with the specified name
id<EMVicToken> vicToken = [vicTokenManager tokenWithName:tokenName
error:&error];
//get the token with the specified name and custom fingerprint data
id<EMVicToken> vicToken = [vicTokenManager tokenWithName:tokenName
fingerprintCustomData:fingerPrintCustomData error:&error];
```

Limitations of VIC Tokens

The application developer must be aware of the following limitations:

- Only Mobile Provisioning Protocol (MPP) versions 1 and 2 are supported.
- VIC functionality is only available with the VAS server.

Verifying VIC

No settings are required on VIC. The application will need to create a **VicDevice**, provide the code to be verified, usually from an issuer such as the bank.

- On Android

```
// get the device
VicDevice device = vicService.getFactory().createVicDevice(vicToken);
// Verify the issuer code
boolean isVerified = device.verifyIssuerCode(codeToVerify, authInput);
```

- On iOS

```
// get the device
id<EMVicDevice> device = [[vicService vicFactory]
createVicDeviceWithToken:vicToken error:&error];
// Verify the issuer code
BOOL isVerified = [vicDevice verifyIssuerCode:codeToVerify authInput:authInput
error:&error];
```

OTP Helpers

OTP Scrambling

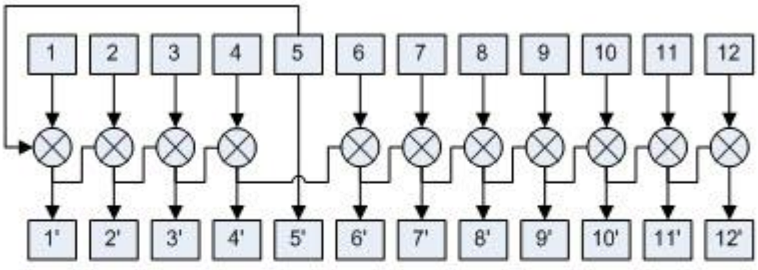
This is a scrambling function used to increase the visual randomness of OTPs. The post-treatment can be used by an application which is in compliance with the Ezio server policy. As this function does not enhance the security in OTP, the value of the OTP must be greater than five digits.

Note:

This process does not change the length of OTPs.

The following figure shows how the scrambling algorithm works. It takes the fifth digit of the OTP, and uses this digit as Initial Value (IV), and then making a chained modulus 10 additions for each digit in the response, bypassing digit 5, which is copied without modification.

Figure 17 Scrambling Algorithm Used for OTPs



The algorithm can be described as: n=(n+IV) mod 10

- On Android:

```
// unscrambled response: 630 243 848
// initial Value (IV): 4
// scrambled response: 033 548 608
SecureString scrambleOtp = OtpTools.scrambleOtp(ScrambleType.VAS, otp);
```

- On iOS:

```
id<EMSecureString> scrambleOtp = [EMOtpTools
scrambleOtpWithType:EMScrambleTypeVas otp:otp];
```

OTP Left-Padding with Zeros

Ezio Mobile SDK provides a method to left-pad the OTP with zeros only if the specified length is longer than the generated OTP length.

- On Android:

```
SecureString paddedOtp = OtpTools.padOtpWithZero(otp, 12);
```

- On iOS:

```
id<EMSecureString> paddedOtp = [EMOtpTools padOtpWithZero:otp minLength:12];
```

Application Development Resources

Generating Offline Provisioning Data

A tool (eps-tokenbuilder) to generate sets of test tokens for various authentication servers is included into the delivery package. The test values are exported in a format compliant with the provisioning configuration defined in Provisioning Configuration. The credentials to import is defined by a session key and a provisioning response value.

The following extract gives an example on the two test tokens generated by tokenbuilder that can be used in an offline provisioning.

```
<?xml version="1.0" 'encoding="UTF-8" standalone="yes"?>
<provisioningData>
  <provisioningProtocol>PP_V2</provisioningProtocol>
  <tokenList>
       <token>
  <pin>00273943</pin>
  <provisioningResponse>54B6EA...276ACA</provisioningResponse>
  <sessionKey>269219... D9F1A2</sessionKey>
  <tokenId>0001001000909998:56</tokenId>
</token>
<token>
  <pin>25448807</pin>
  <provisioningResponse>51C77C...22DF59</provisioningResponse>
  <sessionKey>F67CD6...877AD2</sessionKey>
  <tokenId>0001001000909999:56</tokenId>
</token>
  </tokenList>
</provisioningData>
```

The mobile application uses the PIN provided, session key and provisioning response for each token as resources for application development.

Sample Offline Provisioning Data Inputs

This section describes the sample inputs and outputs for offline provisioning that can be used to facilitate application development and testing.

The integration of the application with Ezio Mobile SDK can be test in isolation. Neither the EPS server nor authentication server is required when the following samples are used.

The following table lists the sample configurations created to allow application developers to evaluate their integration with Ezio Mobile SDK.

Table 15 Sample Offline Provisioning Configuration Data

Configuration Name	*PP Version*	PIN	Session Key	HMAC Key	Response	Plaintext Device Key
CAP.DS.v1	1	0445	0xFB, 0xDC, 0x5D, 0x5D, 0x75, 0x19, 0xAB, 0x3E, 0xBA, 0x61, 0xFD, 0xC7, 0x92, 0xE6, 0x3E, 0x08, 0x0D, 0x3D, 0xB6, 0x25, 0x92, 0x5B, 0x1F, 0x5B	N/A	0x55, 0x87, 0x97, 0x53, 0x4F, 0x49, 0x1C, 0xF0, 0xB2, 0xDD, 0x1F, 0xDD, 0xAF, 0x95, 0xFD, 0x74, 0xB3, 0xC5, 0xDD, 0x50, 0x30, 0xF5, 0x1A, 0xF7	0xA0, 0x14, 0xBD, 0x15, 0xFC, 0xD7, 0x83, 0x32, 0x09, 0x6F, 0x99, 0x42, 0xEF, 0xEE, 0x81, 0x09
CAP.v3	3	6917	0xB8, 0xDF, 0x18, 0xD0, 0x91, 0x53, 0x2B, 0x37, 0xED, 0x5B, 0x81, 0x7E, 0xFC, 0xD8, 0x62, 0x05	0x0C, 0x80, 0x89, 0xF3, 0x8C, 0x32, 0xD0, 0x87, 0x38, 0x59, 0x21, 0x0D, 0x6B, 0xE4, 0x09, 0x96	0x31, 0xC4, 0xFD, 0xD1, 0xE5, 0x80, 0x91, 0xA1, 0x82, 0xE8, 0x38, 0x1F, 0x84, 0x9C, 0x7C, 0xE3, 0xB9, 0x3E, 0xEE, 0xA1, 0xD9, 0xB4, 0xA8, 0x0C, 0x65, 0x54, 0x24, 0x84, 0x47, 0x30, 0x7A, 0xF2, 0xDC, 0xAD, 0xC2, 0x7F, 0x91, 0xDA, 0x5D, 0xA4, 0xF5, 0xB4, 0x8C, 0x7A, 0x15, 0x97, 0xE6, 0xAC, 0x0D, 0x61, 0xAB, 0x5A, 0x74, 0x70, 0x09, 0x25, 0x54, 0x07, 0xAC, 0xF5, 0x5B, 0x63, 0xF1, 0xF7, 0x49, 0x01, 0x5C, 0x89, 0xBD, 0x19, 0x19, 0x48, 0xF3, 0xCF, 0xAD, 0xE4, 0xE4, 0x18, 0x09, 0x45	0xC6, 0xF0, 0x16, 0x42, 0x55, 0x01, 0x83, 0x73, 0x6C, 0xA6, 0xB7, 0x27, 0xFE, 0x50, 0x19, 0xEE
OATH.v3	3	9444	0xA4, 0x40, 0xAB, 0x04, 0xF6, 0xD3, 0xB5, 0xD9,	0xAE, 0xD7, 0x72, 0x0B,	0xC9, 0xFD, 0x6D, 0x94, 0x92, 0xCA, 0xE0, 0x46, 0x78, 0xB5, 0x73, 0x0E, 0xC9,	0xC3, 0x3A, 0xCD, 0x2B, 0xEB, 0xDC, 0xB9, 0xDF, 0xE0, 0x4F, 0x06, 0x9E,

Configuration Name	*PP Version*	PIN	Session Key	HMAC Key	Response	Plaintext Device Key
			0x85, 0x55, 0xDC, 0x43, 0x88, 0x5E, 0x7D, 0xDF	0x75, 0xCC, 0x08, 0xCD, 0xAB, 0x2D, 0xB3, 0x7B, 0xF3, 0xE4, 0x4C, 0xC8	0x4E, 0xBB, 0xA6, 0xFA, 0x04, 0x02, 0x6F, 0x2E, 0x22, 0xC7, 0x90, 0xE0, 0x5D, 0x93, 0x7D, 0x6A, 0x9A, 0xB2, 0xA7, 0xA5, 0x54, 0x0D, 0x50, 0x84, 0x3B, 0x4A, 0x8A, 0x78, 0xED, 0xD1, 0x2A, 0x19, 0x61, 0x9A, 0x39, 0x40, 0xDC, 0x4A, 0x47, 0x1C, 0xB0, 0x16, 0x40, 0xDC, 0xC9, 0xEA, 0x81, 0xED, 0xA4, 0x42, 0x73, 0x1F, 0x84, 0x13, 0xF5, 0x0C, 0xC9, 0x88, 0xF1, 0x56, 0x88, 0x46, 0x7E, 0x61, 0x69, 0xCC, 0x49, 0x2D, 0xAE, 0xB7, 0x79, 0x33, 0x1C, 0x64, 0x09, 0xCD, 0xB2, 0xF6, 0x9A, 0x48, 0x4E, 0xE8, 0x9C	0x11, 0x57, 0xAD, 0x1C, 0x6B, 0x71, 0xC5, 0x7D
OATH_DUAL_SEED.v3	3	1234	0xA4, 0x40, 0xAB, 0x04, 0xF6, 0xD3, 0xB5, 0xD9, 0x85, 0x55, 0xDC, 0x43, 0x88, 0x5E, 0x7D, 0xDF	0xAE, 0xD7, 0x72, 0x0B, 0x75, 0xCC, 0x08, 0xCD, 0xAB, 0x2D, 0xB3, 0x7B, 0xF3, 0xE4, 0x4C, 0xC8	0x4A, 0x70, 0x13, 0x61, 0x45, 0x2D, 0x92, 0x93, 0x12, 0x06, 0x7D, 0x9E, 0x81, 0x18, 0xE5, 0xAE, 0x84, 0x70, 0xEC, 0x02, 0xF7, 0x66, 0x85, 0x06, 0x7A, 0x98, 0xA7, 0x5B, 0xB9, 0x11, 0x0C, 0x5C, 0xAB, 0xEB, 0x11, 0x53, 0xB1, 0x17, 0xE9, 0x61, 0x34, 0xEF, 0x97, 0x54, 0x64, 0x69, 0x2F, 0xE9, 0x15, 0xA6, 0x89, 0x30, 0xA0, 0x05, 0x2F, 0xB4, 0xCE, 0x1B, 0x66, 0xFC, 0xD5, 0x1A, 0x11, 0x30, 0xED, 0xFB, 0x5D, 0xFF, 0x7F, 0xF8, 0x4D, 0x9F, 0xD3, 0xAB, 0xF6, 0x27, 0x96, 0xB3, 0xCF, 0xFA, 0x9B, 0x1D, 0x4B, 0x77, 0xE6, 0x80, 0xE0, 0x03, 0x1A, 0x64, 0x8C,	key[0] = 0x4F, 0x70, 0x0C, 0xCB, 0x05, 0xA8, 0x58, 0x9B, 0x35, 0xAB, 0x6D, 0xC1, 0xD4, 0xB4, 0x39, 0xE7, 0x90, 0x63, 0xD1, 0xE6, 0xCA, 0xC7, 0x7A, 0xB9, 0x32, 0x4D, 0x5D, 0x65, 0x2E, 0x84, 0x25, 0xAE key[1] = 0xBB, 0x49, 0xA4, 0xE8, 0xEC, 0x61, 0x11, 0x67, 0x65, 0x47, 0x25, 0x8F, 0x45, 0x45, 0xDA, 0x7C, 0x02, 0xAC, 0x4B, 0x64, 0x5D, 0x3C, 0x8E, 0xDB, 0xBE, 0x0B, 0xF5, 0x4A, 0x16, 0x70, 0xDD, 0x80

Configuration Name	*PP Version*	PIN	Session Key	HMAC Key	Response	Plaintext Device Key
					0x90, 0x27, 0x68, 0x48, 0xDE, 0x02, 0xD3, 0x4D, 0xF5, 0xF6, 0xB3, 0xD3, 0x70, 0x01, 0x29, 0x8C, 0x8C, 0x7A, 0x4F, 0x5F, 0x39, 0x3B, 0x9C, 0xE6, 0x23, 0x62, 0xC7, 0x41, 0x7F, 0xDC, 0x6A, 0x76, 0x9B, 0x82, 0x93, 0x62, 0x54	

CAP and DS

The following table lists the offline token configuration names and the settings required for the CAP and DS devices.

Configuration Name	Setting	Value
CAP.DS.v1	CDOL	0x9F, 0x02, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x9F, 0x03, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x9F, 0x1A, 0x02, 0x00, 0x00, 0x95, 0x05, 0x80, 0x00, 0x00, 0x00, 0x00, 0x5F, 0x2A, 0x02, 0x00, 0x00, 0x9A, 0x03, 0x00, 0x00, 0x00, 0x9C, 0x01, 0x00, 0x9F, 0x37, 0x04, 0x00, 0x00, 0x00, 0x00, 0x82, 0x02, 0x10, 0x00, 0x9F, 0x36, 0x02, 0x00, 0x00, 0x9F, 0x52, 0x06, 0xA5, 0x00, 0x03, 0x04, 0x00, 0x00
	_CID	0x80
	_IAD	0x06, 0x01, 0x0A, 0x03, 0x04, 0x00, 0x00, 0x00
	_IAF	0xC0
	_IPB	0x00, 0x00, 0x00, 0x7F, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
CAP.v3	CDOL	0x9F, 0x02, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x9F, 0x03, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x9F, 0x1A, 0x02, 0x00, 0x00, 0x95, 0x05, 0x80, 0x00, 0x00, 0x00, 0x00, 0x5F, 0x2A, 0x02, 0x00, 0x00, 0x9A, 0x03, 0x00, 0x00, 0x00, 0x9C, 0x01, 0x00, 0x9F, 0x37, 0x04, 0x00, 0x00, 0x00, 0x00, 0x82, 0x02, 0x10, 0x00, 0x9F, 0x36, 0x02, 0x00, 0x00, 0x9F, 0x52, 0x06, 0xA5, 0x00, 0x03, 0x04, 0x00, 0x00
	CID	0x80
	_IAD	0x00, 0x00, 0xA5, 0x00, 0x03, 0x04, 0x00, 0x00, 0x00, 0x00
	_IAF	0x80
	_IPB	0x00, 0x00, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

The following table lists the operation and inputs used to generate the sample OTPs.

Operation	Input	Value
CAP Mode 1	Challenge	00002011

Operation	Input	Value
	_Amount	5000
	_Currency	USD
CAP Mode 2		
CAP Mode 2 TDS	TDS 1	1111
	_TDS 2	222
	_TDS 3	33
	_TDS 4	4
CAP Mode 3	Challenge	00002011
GPF DS Code		
GPF DS Signature	Challenge	117233
	_Domain	sign
	_Transaction Data: INPUT_TO_ACCOUNT (0xDF04)	31240012345
	_Transaction Data: INPUT_AMOUNT (0x9F03)	0000001200.00

The following tables lists the configuration name, operation, counter value, and sample OTP the application developer can verify against. Each operation is performed in the order listed to generate the given OTP. If any operation is skipped, then all the subsequent OTPs will not be correct.

Configuration Name	Operation	Counter	OTP
CAP.DS.v1	CAP Mode 1	0	99398
	_CAP Mode 2	1	177212
	_CAP Mode 2 TDS	2	199230
	_CAP Mode 3	3	272746
	_GPF DS Code	4	0389219
	_GPF DS Signature	5	990427886
CAP.v3	CAP Mode 1	0	106673
	_CAP Mode 2	1	162841
	_CAP Mode 2 TDS	2	229034
	_CAP Mode 3	3	306369

Configuration Name	Operation	Counter	OTP
	_GPF DS Code	N/A	N/A
	_GPF DS Signature	N/A	N/A

OATH

The following table lists the offline token configuration names and the settings required for the OATH device.

Table 16 Offline Token Configuration Names and Settings for OATH Devices

Configuration Name	Setting	Value
OATH.v3	HOTP length	8
	TOTP length	8
	TOTP timestep size	30
	TOTP start time	Current time in seconds (see Getting UTC Time)
	OCRA counter used	True
	OCRA challenge question format	Numeric
	OCRA OTP length	8
	OCRA maximum challenge question length	9
	OCRA time settings	SECONDS, 5, Current time in seconds (see Getting UTC Time)
OATH_DUAL_SEED.v3	HOTP length	N/A
	TOTP timestep size	30
	TOTP start time	Current time in seconds (see 6.7.2 Getting UTC Time)
	OCRA counter used	False
	OCRA challenge question format	Numeric
	OCRA OTP length	8
	OCRA maximum challenge question length	8
	OCRA time settings	SECONDS, 30, Current time in seconds (see Getting UTC Time)
	OCRA Hash Algorithm	SHA256

The following table lists the operation and inputs used to generate the sample OTPs.

Table 17 Operations and Inputs to Generate Sample OTPs

Operation	Input	Value
HOTP		
TOTP		
OCRA	Server Challenge Question	91234512
	_ Client Challenge Question	-

Operation	Input	Value
	_ Password Hash	-
	_ Session	-
Gemalto HOTP		
Gemalto TOTP		
Gemalto Event Signature	List index 0	1111
	_ List index 1	222
	_ List index 2	33
	_ List index 3	4
Gemalto Event Challenge / Response	Challenge	91234512
Gemalto Time Signature	List index 0	1111
	_ List index 1	22
	_ List index 2	33
	_ List index 3	4
Gemalto Time Challenge / Response	Challenge	91234512

The following tables lists the configuration name, operation, counter value, time and sample OTP the application developer can verify against. Each operation is performed in the order listed to generate the given OTP. If any operation is skipped, then all the subsequent counter based OTPs will not be correct.

Note:

For time-based Gemalto operations, the operating system's time has to be manually set to the value listed. However, since setting the time manually and generating the OTP may take more than 30 seconds, five sample OTPs are generated for application developer to check against.

Table 18 Expected OTP Result

Configuration Name	Operation	Counter	Time (UTC)	OTP
OATH.v3	HOTP	0		09810777
	TOTP		Current Time	09810777
	OCRA	1		77872738
	Gemalto HOTP	2		15061273
	Gemalto TOTP		07 Nov 2013 13:17:18 GMT	09633232
			07 Nov 2013 13:17:30 GMT	12009573
			07 Nov 2013 13:18:00 GMT	32370445
			07 Nov 2013 13:18:30 GMT	40001204

Configuration Name	Operation	Counter	Time (UTC)	OTP
			07 Nov 2013 13:19:00 GMT	37687095
	Gemalto Event Signature	3		43939470
	Gemalto Event Challenge / Response	4		46035548
	Gemalto Time Signature		07 Nov 2013 13:19:32 GMT	16643824
			07 Nov 2013 13:20:00 GMT	76085823
			07 Nov 2013 13:20:30 GMT	61225402
			07 Nov 2013 13:21:00 GMT	26866087
			07 Nov 2013 13:21:30 GMT	89507976
	Gemalto Time Challenge / Response		07 Nov 2013 13:22:00 GMT	00843946
			07 Nov 2013 13:22:30 GMT	36524886
			07 Nov 2013 13:23:00 GMT	90656244
			07 Nov 2013 13:23:30 GMT	56957557
			07 Nov 2013 13:24:00 GMT	59021580
OATH_DUAL_SEED.v3 ▪ Event based not supported by token capability ▪ Limited set for brevity	TOTP (key[0])		Current Time	79764307
	_TOTP (key[1])		Current Time	07461270
	_OCRA (key[0])		Current Time	15695377
	_OCRA (key[1])		Current Time	63976105

Behavior of Jailbroken and Rooted Devices

The configuration of the OTP module includes a policy setting on how a Token Manager manages its tokens on jailbroken/rooted devices. Refer to Device Fingerprint for more information.

This policy indirectly alters the jailbroken and rooted behavior in a predefined way.

The following table summarizes all the possible behaviors:

Table 19 Policies Used in Token Management

Policy	Creating a Token	Getting a Token
Ignore	–	–

Policy	Creating a Token	Getting a Token
Fail	Sequence is aborted.Exception/error code is raised.	Sequence is aborted.Exception/error code is raised.
Remove Token	Sequence is aborted. The token is removed and an exception/error code is raised.	Sequence is aborted. The token is removed and an exception/error code is raised.
Remove All Tokens	Sequence is aborted. All the tokens and all the respective services are removed and an exception/error code is raised.	Sequence is aborted. All the tokens and all the respective services are removed and an exception/error code is raised.

Exceptions/errors thrown from operations on root or jailbreak detection can be found in the API documentation.

OTP Module Reset

Note:

OTP Module reset is currently only available on Android platform.

This is used to remove all the tokens that have been created.

On Android:

```
try {
 otpModule.reset();
} catch (IdpException e) {
// handle error
}
```

MSP Parser

Mobile signing protocol (MSP) is a protocol to transfer data for OTP generation and data signing.

This module describes how to use the MSP Parser to parse and read the information from mobile signing protocol frames. Currently, the following modes of MSP frames are supported:

- CAP Mode 1
- CAP Mode 2
- CAP Mode 2 Tds
- CAP Mode 3
- OATH HOTP
- OATH TOTP
- OATH OCRA

Frames with signing and obfuscation are supported by MSP frame parser for you to enhance the security. Frames with partial input mode is also supported to ensure that the user reads the data present on the screen of the device. The user is prompted to complete some fields with important values such as part of an amount.

Application Requirements

Configuration

The application has to configure the Core with an MSP configuration apart from OTP configuration and OOB configuration. Otherwise, instantiating this module will fail. To use the signature and obfuscation feature, configure the signature key and obfuscation key in the MSP configuration.

```
To fetch the signature key and the obfuscation key from their respective arrays, the
valid index range is from 1 to 15.
```

- On Android:

```
// Create MspSignatureKey
MspSignatureKey signKey = new MspSignatureKey(
SIGN_PUB_KEY,
SIGN_P_KEY,
SIGN_Q_KEY,
SIGN_G_KEY
);
// Configuration with signature keys and obfuscation key
MspConfiguration mspConfig = new MspConfiguration.Builder()
  .setSignatureKeys(Arrays.asList(signKey))
  .setObfuscationKeys(Arrays.asList(OBFUSC_MK_KEY2))
  .build();

IdpCore core = IdpCore.configure(deviceBinding, activationCode, mspConfig);
```

- On iOS:

```
//Signature keys and obfuscation keys
NSData *obfusMkKeyData = [[NSData alloc] initWithBytes:OBFUSC_MK_KEY
length:sizeof(OBFUSC_MK_KEY)];
NSData *signPubKeyData = [[NSData alloc] initWithBytes:SIGN_PUB_KEY
length:sizeof(SIGN_PUB_KEY)];
NSData *signPKeyData = [[NSData alloc] initWithBytes:SIGN_P_KEY
length:sizeof(SIGN_P_KEY)];
NSData *signQKeyData = [[NSData alloc] initWithBytes:SIGN_Q_KEY
length:sizeof(SIGN_Q_KEY)];
NSData *signGKeyData = [[NSData alloc] initWithBytes:SIGN_G_KEY
length:sizeof(SIGN_G_KEY)];


NSMutableArray *obfuscationKeys = [NSMutableArray array];
[obfuscationKeys addObject:obfusMkKeyData];


EMSignatureKey *signatureKey = [[EMSignatureKey alloc] initWith:signPubKeyData
p:signPKeyData q:signQKeyData g:signGKeyData];
NSMutableArray *signatureKeys = [NSMutableArray array];
[signatureKeys addObject:signatureKey];

//Msp configuration with signature and obfuscation keys
EMMspConfiguration *mspConfiguration = [EMMspConfiguration
configurationWithObfuscationKeys:obfuscationKeys signatureKeys:signatureKeys];
// add configs
NSSet* configs = \[NSSet setWithObject:mspConfiguration];


\\Configure the core
EMCore configureWithActivationCode:activationCode configurations:configs];
```

In cases where the signature and obfuscation feature are not used, you can use the default MSP configuration: \\ \\

- On Android:

```
// Use default Msp configuration
MspConfiguration mspConfig = new MspConfiguration.Builder().build();

IdpCore core = IdpCore.configure(deviceBinding, activationCode, mspConfig);
```

- On iOS:

```
//Msp configuration with default configuartion
EMMspConfiguration *mspConfiguration = [EMMspConfiguration
defaultConfiguration];


NSSet* configs = [NSSet setWithObject:mspConfiguration];


//Configure the core [EMCore configureWithActivationCode:activationCode
configurations:configs];
```

Create MSP Parser

You have to create an MSP parser in order to parse the MSP frame. The following snippets demonstrates how to create the parser.

- On Android:

```
MspModule mspModule = MspModule.create();
MspService mspService = MspService.create(mspModule);
MspFactory mspFactory = mspService.getFactory();
MspParser mspParser = mspFactory.createMspParser();
```

- On iOS:

```
EMMspModule *mspModule = [EMMspModule mspModule];
EMMspService *mspService = [EMMspService serviceWithModule:mspModule];
EMMspFactory *mspFactory = [mspService mspFactory];
id<EMMspParser> mspParser = [mspFactory createMspParser];
```

Parse OATH Frame

Developers need to parse the MSP frame to generate the OTP. For example, for a OOB transaction signing request, you can get the MSP frame by parsing the MSP frame byte data of the request.
The following snippets demonstrate the parsing of MSP frame and generation of OTP for OATH (HOTP, TOTP, and OCRA).

- On Android:

```
// Parse Msp frame
  try {
   MspFrame mspFrame = mspParser.parse(mspFrameBytes);
   MspData mspData = mspParser.parseMspData(mspFrame);
   if (mspData.getBaseAlgo() == MspBaseAlgorithm.BASE_OATH) {
   // Cast to MspOathData
   MspOathData mspOathData = (MspOathData) mspData;
   int mode = mspOathData.getMode();
   switch (mode) {
   case MspOathData.MSP_OATH_HOTP:
     // Compute Oath HOTP
     otp = oathDevice.getHotp(pinAuthInput);
     break;
   case MspOathData.MSP_OATH_TOTP:
     // Compute Oath TOTP
     otp = oathDevice.getTotp(pinAuthInput);
   case MspOathData.MSP_OATH_OCRA:
     // Create device depending on OCRA suite
     SoftOathSettings oathSettings = oathFactory.createSoftOathSettings();
     oathSettings.setOcraSuite(OCRA_SUITE);
     ocraDevice = oathFactory.createSoftOathDevice(ocraToken, oathSettings);

     // Parse Msp data
     SecureByteArray serverChallenge = mspOathData.getOcraServerChallenge() ==
null ? null : mspOathData.getOcraServerChallenge().getValue();
     SecureByteArray clientChallenge = mspOathData.getOcraClientChallenge() ==
null ? null : mspOathData.getOcraClientChallenge().getValue();
     SecureString passwordHash = mspOathData.getOcraPasswordHash() == null ?
null : mspOathData.getOcraPasswordHash().getValue();
     SecureString session = mspOathData.getOcraSession() == null ? null :
mspOathData.getOcraSession().getValue();

     // Compute Oath Ocra OTP
     otp = ocraDevice.getOcraOtp(pinAuthInput, serverChallenge,
clientChallenge, passwordHash, session);
   default:
     break;
   }
 }
  } catch (IdpException e) {
 // Handle exception here ...
  }
```

- On iOS:

```
    //Parse the mspFrame from the transaction response (which is a secure byte
array) to a msp Frame object
    id<EMMspFrame> mspFrame = [mspParser parse:mspFramePayload
                                         error:&error];

    //get the payload data from the frame
    id<EMMspData> mspData = [mspParser parseMspData:mspFrame
```

```
                                              error:&error];
    if(!error)
    {
        //Check if the base algorithm is OATH
        if (([mspData baseAlgo] == EM_MSP_BASE_ALGO_OATH))
        {
            // Cast the msp data to oath data
            id<EMMspOathData> oathData = (id<EMMspOathData>) mspData;
            id<EMSecureString> otp;
            //Check mode and generate otp
            switch ([oathData mode]) {
                case EM_MSP_OATH_HOTP:
                    //HOTP
                    otp = [oathDevice hotpWithAuthInput:pinHotp error:&error];
                    break;
                case EM_MSP_OATH_TOTP:
                    //TOTP
                    otp = [oathDevice totpWithAuthInput:pinTotp error:&error];
                    break;
                case EM_MSP_OATH_OCRA:
                    //OCRA
                    //We need to set the ocra suite in the oath settings
depending on the OCRA policy
                    [oathSettings setOcraSuite:[secureContainerFactory
secureStringFromString:OCRA_SUITE]];
                    //And then get the device based on the updated oathSettings
and the corresponding ocra token
                    oathDevice = [oathFactory
createSoftOathDeviceWithToken:ocratoken

settings:oathSettings

error:&error];
                    //get the otp with the ocra auth input
                    otp = [oathDevice ocraOtpWithAuthInput:pinOcra
                                   serverChallengeQuestion:[oathData
ocraServerChallenge] ? [[oathData ocraServerChallenge] value] : nil
                                   clientChallengeQuestion:[oathData
ocraClientChallenge] ? [[oathData ocraClientChallenge] value] : nil
                                              passwordHash:[oathData
ocraPasswordHash] ? [[oathData ocraPasswordHash] value] : nil
                                                   session:[oathData
ocraSession] ? [[oathData ocraSession] value] : nil
                                                     error:&error];
                    break;
                default:
                    break;
            }
        }
    } else {
        // Handle error here
    }
```

Parse CAP Frame

You have to parse the MSP frame to generate the OTP. For example, for a OOB transaction signing request, you can get the MSP frame by parsing the MSP frame byte data of the request.
The following snippets demonstrate the parsing of MSP frame and generation of OTP for CAP (Mode1, Mode2, Mode2TDS, and Mode3).

- On Android:

```
// Parse Msp frame
 try {
          MspFrame mspFrame = mspParser.parse(mspFrameBytes);
          MspData mspData = mspParser.parseMspData(mspFrame);
          if (mspData.getBaseAlgo() == MspBaseAlgorithm.BASE_OATH) {
               // Cast to MspOathData
               MspOathData mspOathData = (MspOathData) mspData;
               int mode = mspOathData.getMode();
               switch (mode) {
                   case MspOathData.MSP_OATH_HOTP:
                       // Compute Oath HOTP
                       otp = oathDevice.getHotp(pinAuthInput);
                       break;
                   case MspOathData.MSP_OATH_TOTP:
                       // Compute Oath TOTP
                       otp = oathDevice.getTotp(pinAuthInput);
                   case MspOathData.MSP_OATH_OCRA:
                       // Create device depending on OCRA suite
                       SoftOathSettings oathSettings =
oathFactory.createSoftOathSettings();
                       oathSettings.setOcraSuite(OCRA_SUITE);
                       ocraDevice = oathFactory.createSoftOathDevice(ocraToken,
oathSettings);

                           // Parse Msp data
                       SecureByteArray serverChallenge =
mspOathData.getOcraServerChallenge() == null ? null :
mspOathData.getOcraServerChallenge().getValue();
                       SecureByteArray clientChallenge =
mspOathData.getOcraClientChallenge() == null ? null :
mspOathData.getOcraClientChallenge().getValue();
                       SecureString passwordHash =
mspOathData.getOcraPasswordHash() == null ? null :
mspOathData.getOcraPasswordHash().getValue();
                       SecureString session = mspOathData.getOcraSession() ==
null ? null : mspOathData.getOcraSession().getValue();
// Compute Oath Ocra OTP
                       otp = ocraDevice.getOcraOtp(pinAuthInput,
serverChallenge, clientChallenge, passwordHash, session);
                  default:
                       break;
               }
           }
       } catch (IdpException e) {
           // Handle exception here ...
       }
```

- On iOS:

```
//Parse the mspFrame from the transaction response (which is a secure byte
array) to a msp Frame object
id<EMMspFrame> mspFrame = [mspParser parse:mspFramePayload
                                          error:&error];

//get the payload data from the frame
     id<EMMspData> mspData = [mspParser parseMspData:mspFrame
                                               error:&error];
if(!error)
{
     //Check if the base algorithm is OATH
      if (([mspData baseAlgo] == EM_MSP_BASE_ALGO_OATH))
         {
             // Cast the msp data to oath data
             id<EMMspOathData> oathData = (id<EMMspOathData>) mspData;
             id<EMSecureString> otp;
             //Check mode and generate otp
             switch ([oathData mode]) {
                 case EM_MSP_OATH_HOTP:
                     //HOTP
                     otp = [oathDevice hotpWithAuthInput:pinHotp error:&error];
                     break;
                 case EM_MSP_OATH_TOTP:
                     //TOTP
                     otp = [oathDevice totpWithAuthInput:pinTotp error:&error];
                     break;
                 case EM_MSP_OATH_OCRA:
                     //OCRA
                     //We need to set the ocra suite in the oath settings
depending on the OCRA policy
                     [oathSettings setOcraSuite:[secureContainerFactory
secureStringFromString:OCRA_SUITE]];
                     //And then get the device based on the updated
oathSettings and the corresponding ocra token
                     oathDevice = [oathFactory
createSoftOathDeviceWithToken:ocratoken
               settings:oathSettings

error:&error];
                     //get the otp with the ocra auth input
                     otp = [oathDevice ocraOtpWithAuthInput:pinOcra
                                    serverChallengeQuestion:[oathData
ocraServerChallenge] ? [[oathData ocraServerChallenge] value] : nil
clientChallengeQuestion:[oathData ocraClientChallenge] ? [[oathData
ocraClientChallenge] value] : nil
passwordHash:[oathData ocraPasswordHash] ? [[oathData ocraPasswordHash] value]
: nil
                                                    session:[oathData
ocraSession] ? [[oathData ocraSession] value] : nil
error:&error];
                     break;
                 default:
                     break;
             }
         }
     } else {
         // Handle error here
     }
```

Parse CAP Partial Input Frame

For MSP frame with partial input, ensure that the following checks are done:

- Check for the incompleteness of the field.
- Prompt the user to enter the missing fields.
- Update the missing bytes and generate the OTP.

The following code snippets demonstrate the above steps for a CAP Mode1 partial input frame.

- On Android:

```
try {
    if (mspData.getBaseAlgo() == MspBaseAlgorithm.BASE_CAP) {
        // Cast to MspCapData
        MspCapData mspCapData = (MspCapData) mspData;
        if (mspCapData.getMode() == MspCapData.MSP_CAP_MODE1) {
           // Parse MspCapData mode 1 fields
           MspField amountField = mspCapData.getCapAmount();
           // Check amount field is completed or not
           if (amountField.isComplete() ==
MspField.FieldCompleteness.INCOMPLETE) {
              // Prompt the user to complete the missing fields
              // Use the field's missing and visible bytes for UI display
              Map<Integer,Byte> userInput =
displayUIandGetUserInput(amountField.getMissingBytes(),amountField.getVisibleBy
tes());

              //...
              // Once the user completes the fields update the missing fields
in amount data
              if (amountField.updateMissingBytes(userInput)) {
                  // Update succeed
                  // Read updated amount
                  SecureString updatedAmount = amountField.getValue();
                  // Read other fields
                  SecureString mode1challenge =
mspCapData.getCapChallenge().getValue();
                  Currency currency =
Currency.getInstance(mspCapData.getCapCurrency().getValue().toString());
                  // Compute Cap mode 1 OTP
                  otp = capDevice.getOtpMode1(pinAuthInput, mode1challenge,
updatedAmount, currency);
               } else {
                   // Update failed, handle the error here
               }
           }
        }
    }
} catch (IdpException e) {
    // Handle exception here
}
```

- On iOS:

```
    //Check if the base algorithm is Cap and the mode is Mode 1
if (([mspData baseAlgo] == EM_MSP_BASE_ALGO_CAP) && ([mspData mode] ==
EM_MSP_CAP_MODE1))
{
    // Cast the msp data to cap data
    id<EMMspCapData> capData = (id<EMMspCapData>) mspData;

    //Check if the amount is incomplete
    //The amount is the partial input field hence it will not be complete
    if (![[capData capAmount] isComplete])
    {
        //Prompt the user to complete the missing fields
        //Get the missing bytes and the visible bytes. This will be a
dictionary of the index and the value
        [[capData capAmount] visibleBytes];
        [[capData capAmount] missingBytes];

        // Once the user completes the fields, update the missing fields in
amount data
        NSMutableDictionary* fillingBytes = [NSMutableDictionary new];// This
will be a dictionary of the user input value and its corresponding index.
        //update missing bytes
        if([[capData capAmount] updateMissingBytes:fillingBytes])
        {
            // Update succeed
            //generate otp
            otp = [capDevice otpMode1WithAuthInput:pinCap
                                          challenge:[[capData capChallenge]
value]
                                             amount:[[capData capAmount] value]
                                       currencyCode:[[[capData capCurrency]
value] stringValue]
                                              error:&error];

        } else {
            // Handle error here
        }
    }
}
```

Get Token Based on User Token Id (UTI)

If the MSP frame has UTI defined, you have to get the token based on UTI before computing the OTP. The following snippets demonstrate the fetching of the UTI based token from the Oath frame.

Note:

Ensure that the UTI is unique for each token, otherwise, it will potentially return the wrong token.

- On Android:

```
try {
    MspFrame mspFrame = mspParser.parse(mspFrameBytes);
    MspData mspData = mspParser.parseMspData(mspFrame);
    if (mspData.getBaseAlgo() == MspBaseAlgorithm.BASE_OATH) {
        int uti = ((MspOathData) mspData).getUserTokenId();
        if (uti != 0) {
            // Valid uti, get token with Uti
            SoftOathToken oathToken =
oathTokenManager.getTokenWithUserTokenId(uti);
        } else {
            // Handle error here
        }
    }
} catch (IdpException e) {
    // Handle exception here
}
```

- On iOS:

```
//Parse the mspFrame to a msp Frame object
id<EMMspFrame> mspFrame = [mspParser parse:mspFramePayload error:&error];

  //get the payload data from the frame
id<EMMspData> mspData = [mspParser parseMspData:mspFrame error:&error];
if (!error)
{
    //get the UTI from the msp data
    int uti = [mspData userTokenId];
    // If we have the user token identifier (uti) in the frame data then use
the following snippet to get the oath token
    if (uti != 0)
    {
        id<EMSoftOathToken> token = (id<EMSoftOathToken>)[oathTokenManager
tokenWithUti:uti error:&error];
    }
} else {
 // Handle error here
}
```

Out-of-Band Communication

This module defines the feature set for an Out of Band (OOB) communication solution between a backend server and its clients. This link allows the following:

- The server can send instructions and transaction details directly to the mobile application.
- The mobile application can respond to the server and approve or decline an operation.

There are two main advantages offered by the solution:

- Usability improvement
  User only needs to confirm the operation from the mobile application.

  Note:

  It is not required to have the transaction data entered on the mobile application, or OTP and transaction signature entered on the website.

- Security enhancement
  As the transaction channel (typically, the web browser) is completely separated from the verification channel (mobile application) and the data communication on this channel is done in an encrypted way, the security is greatly strengthened. In events where the transaction channel is compromised, the operations transacting on the verification channel will not be affected.

In general, the OOB solution is combined with a push notification system to allow the server to trigger notifications on the mobile phone. The server notifies the user that an operation (authentication or transaction) is waiting for user's confirmation. It is not mandatory for the system to work with this feature but helps to provide a better user experience. While the OOB solution role is to allow a mobile application to authenticate identity or validate transactions executed in another primary channel, that primary channel is not limited to web browsers. For example, it is possible to use the OOB solution to validate operations done with a native mobile banking application— either running on the same device or a different device.

Application Requirements

Configuration

The application has to configure Core with an OOB configuration. Otherwise, instantiating this module will fail.

- On Android:

```
// Configure OOB configuration in Core
OobConfiguration oobConfig = new
OobConfiguration.Builder().setDeviceFingerprintSource(deviceFingerPrintSource).
setRootPolicyOobConfiguration.OobRootPolicy.IGNORE).setTlsConfiguration(tlsConf
iguration).build();

ApplicationContextHolder.setContext(appContext);
IdpCore core = IdpCore.configure(activationCode, oobConfig);
```

```
// Initializing OOB module OobModule
oobModule = OobModule.create();

// Create oobManager OobManager
oobManager = oobModule.createOobManager(oobUrl,
                                        domain,
                                        applicationId,
                                        rsaExponent,
                                        rsaModulus)
```

- On iOS:

```
// Configure core
EMOobConfiguration* oobsConfiguration = [EMOobConfiguration
configurationWithOptionalParameters:^(EMOobConfigurationBuilder
*configurationBuilder) {
    configurationBuilder.tlsConfiguration = aTlsConfiguration;
}];


NSSet* configs = [NSSet setWithObject:oobsConfiguration];
EMCore* core = [EMCore configureWithActivationCode:activationCode
                                    configurations:configs];

// Initializing OOB module
EMOobModule* oobModule = [EMOobModule oobModule];

// Create oobManager
id<EMOobManager> manager = [oobModule createOobManagerWithURL:anUrl
                                                       domain:aDomain
                                                applicationId:anApplicationId
                                                  rsaExponent:aRsaExponent
                                                   rsaModulus:aRsaModulus
                                                        error:&error];
```

Login to the Password Manager

An application must login via the Password Manager in order to use the OOB module. Ensure that you are logged out of the Password Manager once you are finished using the OOB functionalities.

Server Registration

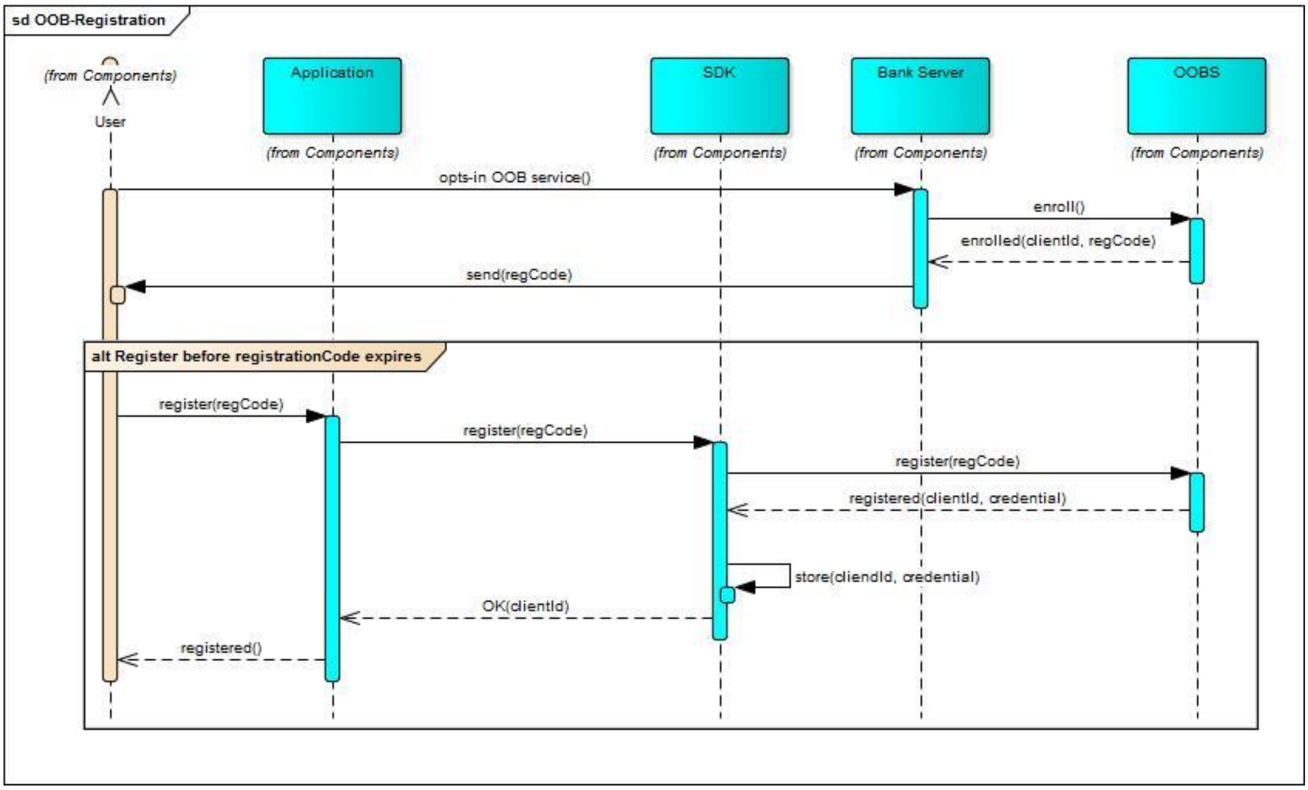
The first step is to register with an OOB server. During the registration, a client ID is created by the server and returned to the application. This client ID is used by the server to identify the registered user and is provided to access all other functionalities of the OOB APIs.

Registering to OOB Server

It requires a user registration code from the provider's server for the registration. This code is not generated by the OOB module and is typically obtained from the server. Once obtained, an application can register to an

OOB server with the user ID, user alias and registration code. Once user has registered successfully, a client ID is received. Refer to the following flow diagram on the registration process.

Figure 18 OOB Registration Sequence



An application can perform multiple registrations; each registration returns a unique clientID. This allows an application to provide a multi-client feature. It is the responsibility of the application to persist and manage the clientID(s) returned by the registration process. Data is to be transmitted to the server through HTTPS and Proprietary protocol.

- On Android:

```
// get the OOB registration Manager
OobRegistrationManager oobRegManager = oobManager.getOobRegistrationManager();


// get the OOB registration manager
OobRegistrationRequest oobRegistrationRequest = new OobRegistrationRequest(
                                   userId,
                                   userAlias,
                                   RegistrationMethod.REGISTRATION_CODE,
                                   registrationParameter);
// register user asynchronously
oobRegManager.register(oobRegistrationRequest, new OobRegistrationCallback() {
     @Override
     public void onOobRegistrationResponse(OobRegistrationResponse response) {
         // check the response
         if (response.isSucceeded()) {
             // display registration successful
         } else {
             // display registration fail
         }
     }
});
```

Note:

It is also possible to provide a Notification Profile during the registration phase. For more information on notification profiles, see Notification Profile.

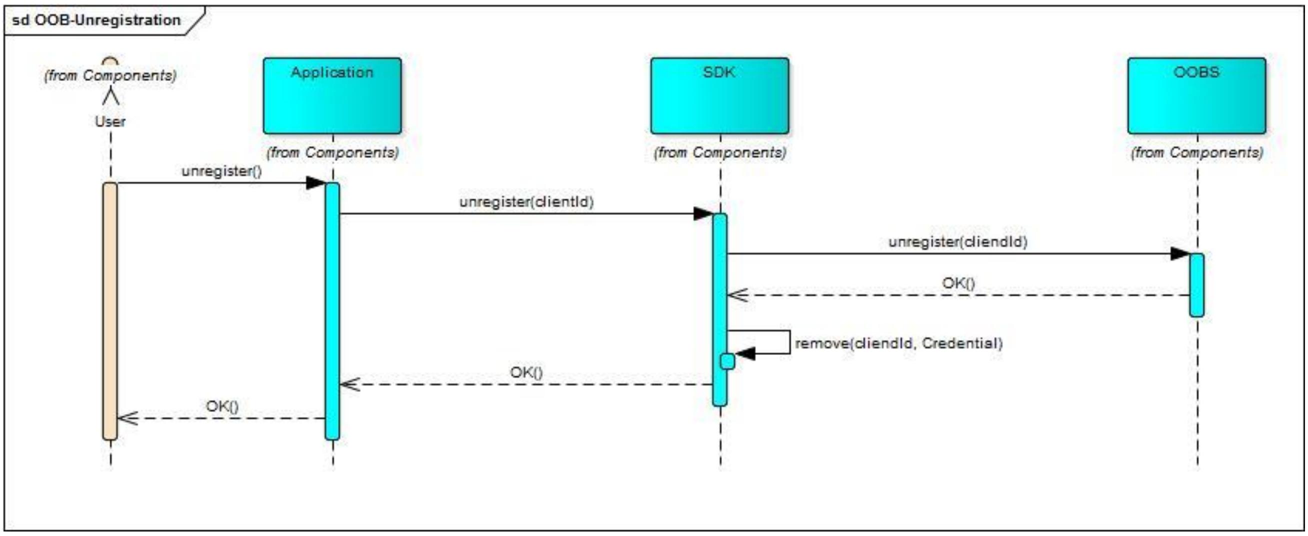
- On iOS:

```
// get the OOB registration manager
id<EMOobRegistrationManager> oobRegistrationManager = [manager
oobRegistrationManager];

// create the registration request
EMOobRegistrationRequest *oobRegistrationRequest = [[EMOobRegistrationRequest
alloc]
                                  initWithUserId:userId
                              userAliasForClient:userAlias

registrationMethod:EMOobRegistrationMethodRegistrationCode
                           registrationParameter:registrationParameter
                                           error:nil];

// register user asynchronously
[oobRegistrationManager registerWithRequest:oobRegistrationRequest
completionHandler:^(id<EMOobRegistrationResponse> aResponse, NSError *anError)
{
    // Check the NSError object first
    // ...

    // Get the result from response
    NSString* resultMessage = [aResponse resultMessage];
    if ([aResponse resultCode] == EMOobResultCodeSuccess) {
        // display registration successful
    }
    else {
        // display registration fail
    }
 }];
```

Unregistering from OOB Server

The unregistering process removes the clientID from the OOB server. Once unregistered, it is not possible to use the OOB functionalities for this client until a registration transaction is successfully executed again. Refer to the following flow diagram for the unregistration process.

Figure 19 OOB Unregistration Sequence Diagram



- On Android:

```
// get the OOB unregistration manager
OobUnregistrationManager oobUnregistrationManager =
oobManager.getOobUnregistrationManager(clientId);


// do the unregistration from OOB asynchronously
oobUnregistrationManager.unregister(new OobUnregistrationCallback() {

    @Override
    public void onOobUnregistrationResponse(final OobResponse response) {
         // check the response
         if (response.isSucceeded()) {
             // display unregistration successful
         } else {
             // display unregistration fail
         }
     }
 });
```

- On iOS:

```
// get the OOB unregistration manager
id<EMOobUnregistrationManager> oobUnregistrationManager = [manager
oobUnregistrationManagerWithClientId:clientId];

// oobRequestParameter is optional, APIs without oobRequestParameter is
available
[oobUnregistrationManager unregisterWithRequestParameter:oobRequestParameter
completionHandler:^(id<EMOobResponse> aResponse, NSError *error) {
    // Check the NSError object first
    // Get the result from response
    NSString* resultMessage = [aResponse resultMessage];
```

```
        if ([aResponse resultCode] == EMOobResultCodeSuccess){
        }else{
           // Handle error code
        }
    }];
```

Notification Profile

When the backend authentication server has a message waiting for the end user, it needs a way to let the user know whether a message is pending. This is done through the notification system. The notification system is configured via the user notification profile. Each user registered with the OOB solution has a notification profile attached to them.
A notification profile is made of zero or more channel/end point pairs (called `OobNotificationProfileItem` in the API), where:

- A channel is a means of sending a notification to an end user. The nature and names of the channels are dependent on the final customization of the solution and are left undefined (for example, the native push channel is passed transparently by the OOB library) for added flexibility.
- An end point is the value associated with the channel for a given user and is therefore dependent on the channel.

These are a few examples of what a notification profile item may contain:

Table 20 Example of a Notification Profile

Channel	End Point Definition	End Point Example
SMS	An MSISDN for sending SMS to.	5555555555
NATIVE_PUSH	A registration Id from GCM or device token from APNS	APA91bHgP3QU_v-z98i0KLk7Z2 [...]
EMAIL	An email address	user@email.com

Optionally, the notification profile can be set during the registration. The OOB service provides both synchronous and asynchronous methods for its notification profile management functionalities.

Retrieving the Notification Profile

The notification profile manager allows an application to retrieve the notification profile as a list of notification profile item.

- On Android:

```
    // get the OOB notification profile manager
    OobNotificationManager notificationProfileManager = oobManager
        .getOobNotificationManager(clientId);

    // Getting the notification profile
    notificationProfileManager.getNotificationProfiles(new
    OobGetNotificationProfileCallback() {
        @Override
        public void onGetNotificationProfileResult(OobNotificationProfilesResponse
    response) {
            if (response.isSucceeded()) {
                List<OobNotificationProfile> notificationProfile =
    response.getNotificationProfiles();
                // get profile success
            } else {
                // get profile fail
            }
        }
    });
```

- On iOS:

```
// get the OOB notification profile manager
id<EMOobNotificationManager> oobNotificationProfileManager = [manager
oobNotificationManagerWithClientId:clientId];

// Getting the notification profile
[oobNotificationProfileManager notificationProfilesWithCompletionHandler:
     ^(id<EMOobNotificationProfilesResponse> aResponse, NSError *error) {
       // Check the NSError object first
       // ...

       if ([aResponse resultCode] == EMOobResultCodeSuccess) {
           NSArray *notificationProfile = [aResponse notificationProfileList];
           // get profile success
        }
         else {
            // get profile fail
         }
    }];
```

Setting the Notification Profile

The notification profile manager allows an application to set the notification profile by providing a specific list of notification profile items.

Note:

User's current notification profile will be replaced by the new list of notification profile items.

- On Android:

```
// updating the notification profile
OobNotificationProfile item1 = new OobNotificationProfile("SMS", endpoints1);
OobNotificationProfile item2 = new OobNotificationProfile("PUSH", endpoints2);
List<OobNotificationProfile> newNotificationProfile = new
ArrayList<OobNotificationProfile>();
newNotificationProfile.add(item1);  newNotificationProfile.add(item2);

notificationProfileManager.setNotificationProfiles(newNotificationProfile,
    new OobSetNotificationProfileCallback() {
        @Override
        public void onSetNotificationProfileResult(final OobResponse response)
{
            if (response.isSucceeded()) {
                // update profile success
            } else {
                // update profile fail
            }
        }
    });
```

- On iOS:

```
// updating the notification profile
EMOobNotificationProfile *oobNotificationProfileItem1 =
[[EMOobNotificationProfile alloc] initWithChannel:@"SMS"

endPoint:@"123456789"];
EMOobNotificationProfile *oobNotificationProfileItem2 =
[[EMOobNotificationProfile alloc] initWithChannel:@"PUSH"

endPoint:@"A1B2C3D4"];
NSArray* notificationItems =
@[oobNotificationProfileItem1,oobNotificationProfileItem2];

[oobNotificationProfileManager setNotificationProfiles:notificationItems
       completionHandler:^(id<EMOobResponse> aResponse, NSError *error) {
            // Check the NSError object first
            // ...

            if ([aResponse resultCode] == EMOobResultCodeSuccess) {
            // update profile success
            }
            else {
                // update profile fail
            }
 }];
```

Clearing the Notification Profile

The notification profile manager allows the application to clear the notification profiles. This method leaves the clientID without a notification profile.

- On Android:

```
notificationProfileManager.clearNotificationProfiles(new
OobClearNotificationProfileCallback() {
    @Override
    public void onClearNotificationProfileResult(final OobResponse response) {
        if (response.isSucceeded()) {
            // clear profile success
        } else {
            // clear profile fail
        }
    }
});
```

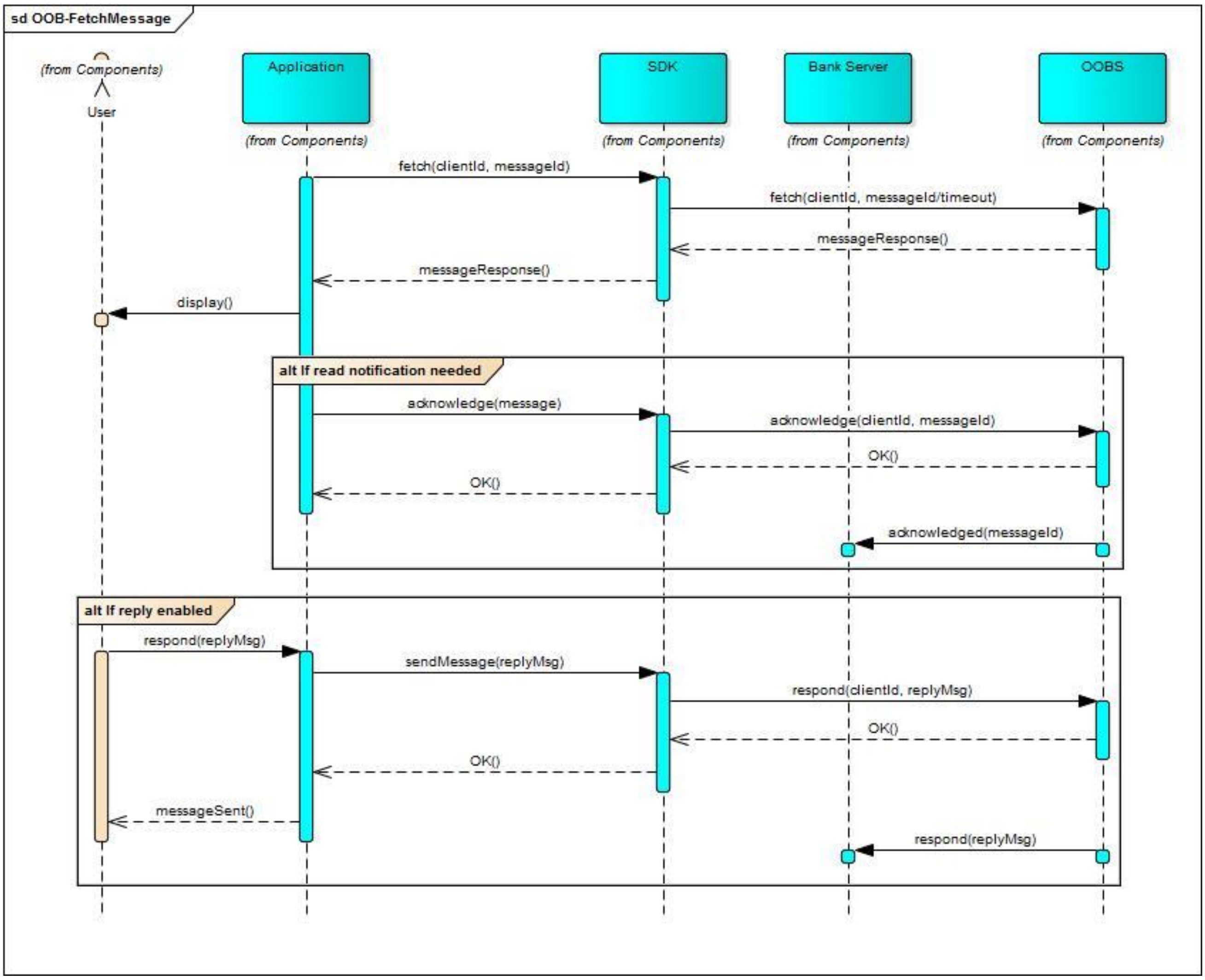
- On iOS:

```
// delete an existing profile
[oobNotificationProfileManager clearNotificationProfilesWithCompletionHandler:
    ^(id<EMOobResponse> aResponse, NSError *error) {
        // Check the NSError object first
        // ...

        if ([aResponse resultCode] == EMOobResultCodeSuccess) {
            // clear profile success
        }
        else {
             // clear profile fail
        }
     }];
```

Messages

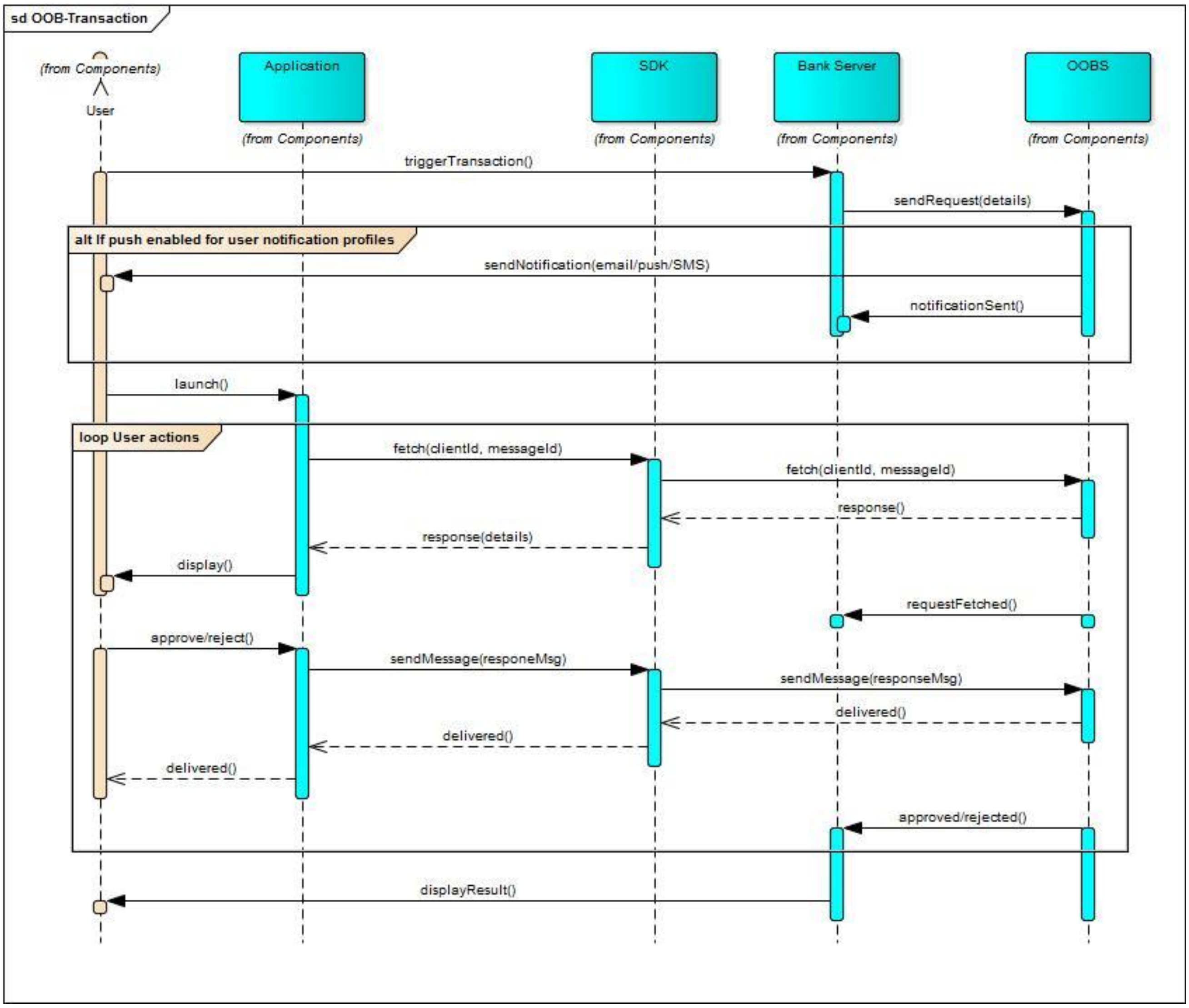
The main goal of the OOB is to securely exchange messages between a mobile client and a backend server. For the client side, there are multiple messaging functions supported including the fetching of a message, acknowledging and/or replying to a message (depends on the requirement of the message fetched), sending a message and verifying a transaction request. The following figure describes the scenario of a client fetching a message and making the corresponding response, if required.

Figure 20 OOB Message Fetching Sequence



The following figure describes the scenario of a transaction verification via the OOB messaging mechanism.

Figure 21 OOB Transaction Verification Sequence



All the resources needed to manage communication are available through the `OobMessageManager` object. The incoming messages can be of any of the following types:

- Provider to user messages (`USER_MESSAGE`) used to convey messages with a recipient and an originator, a subject, an attachment, and so on. For example, a simplified e-mail.
- Transaction verification request (`TRANSACTION_VERIFY`) used to request the end user to provide a response to a specific question.
- Generic messages (`GENERIC`) used by the backend to send certain contents directly to the application without any processing done by Ezio Mobile SDK.

Outgoing messages can be any of the following types:

- From user to provider messages (`USER_MESSAGE`) to send a response to the message previously received from a provider or to send a new message to a provider.
- Transaction verification response (`TRANSACTION_VERIFY`) used to send a response to a transaction verification request.
- Generic outgoing messages (`GENERIC`) used by the mobile client to send certain contents directly to the backend server without any processing done by Ezio Mobile SDK.

- Error messages (`ERROR`) used by the mobile client to send an error message to the backend server. For example, if a received message from the backend server uses a locale not supported by the client.

All the APIs provided by the out-of-band (OOB) message manager comes with both synchronous and asynchronous modes. As the messaging operation may take time to execute (depending on network condition), it is recommended to perform password operations in a background thread. In other words, you may either choose to use the asynchronous APIs or, to place the synchronous API calls into a background thread on their own.

Fetching a Message

Ezio Mobile SDK provides both synchronous and asynchronous methods to fetch and then retrieve the messages from the backend server.

- On Android:

```
// get the message manager from the OOB manager
OobMessageManager oobMessageManager = oobManager.getOobMessageManager(clientId,
providerId);


// fetch a message from the OOB server, wait at most 60 seconds
OobFetchMessageResponse response = oobMessageManager.fetchMessage(60);
if (response.isSucceeded()) {
    // fetch message success
    OobIncomingMessage oobIncomingMessage = response.getOobIncomingMessage();
} else {
    // fetch message fail
}
```

- On iOS:

```
// get the message manager from the OOB manager
id<EMOobMessageManager> oobMessageManager =  [oobManager
oobMessageManagerWithClientId:clientId providerId:providerId];

// fetch a message from the OOB server, wait at most 60 seconds
[oobMessageManager fetchWithTimeout:60 completionHandler:
     ^(id<EMOobFetchMessageResponse> aResponse, NSError *anError) {

        // Check the NSError object first
        // ...
        // Get the result from response
        if ([aResponse resultCode] == EMOobResultCodeSuccess) {
            // fetch message success
            id<EMOobIncomingMessage> oobIncomingMessage = [aResponse
oobIncomingMessage];
        }
        else {
            // fetch message fail
        }
 }];
```

Send a Message Originated by Mobile Client

User to provider messages can be generated directly by the mobile client, without the need of an initial provider message.

The following code snippet provides an example of a user-to-provider create message and the sending of the message.

- On Android:

```
// get the message manager from the OOB manager
OobMessageManager oobMessageManager = oobManager.getOobMessageManager(clientId,
providerId);

// get a message from the OOB server, wait at most 60 seconds
OobUserToProviderMessage message =
oobMessageManager.createUserToProviderMessage(
   "en", /* Locale */
   new Date(),
      core.getSecureContainerFactory().fromString("Purchase"), /* Subject */
   "John Smith", /* From */
   "John Doe", /* To */
   "John Smith", /* ReplyTo */
   "edd31e60-2a40-4dfc-86b4-acdef8578158", /* ThreadId */
   "text/plain", /* contentType */
   messageContent, /* Content */
   null, /* Attachments */
   null /* Meta data */
);
 // Send the message to the OOB server
OobMessageResponse response = oobMessageManager.sendMessage(message);
if (response.isSucceeded()) {
    // send message success
} else {
   // send message fail
}
```

- On iOS:

```
// get the message manager from the OOB manager id<EMOobMessageManager>
oobMessageManager =
    [oobManager oobMessageManagerWithClientId:clientId providerId:providerId];

// get a message from the OOB server, wait at most 60 seconds
id<EMSecureString> aSubject = nil;
id<EMOobUserToProviderMessage> message =
    [oobMessageManager
         createUserToProviderMessageWithLocale:@"en"
                                    createTime:[[NSDate date]
timeIntervalSince1970\]
                                       subject:aSubject
                                          from:@"John Smith"
                                            to:@"John Doe"
                                       replyTo:@"John Smith"
                                      threadId:@"edd31e60-2a40-4dfc-86b4-
acdef8578158"
                                   contentType:@"text/plain"
                                       content: [[context secureDataFactory]

secureStringWithString:@"SHVtbW0sIHlvdSBhcmUgdmVyeSBjdXJpb3VzIDotKQ=="]
                                   attachments:nil
                                          meta:nil];

// Send the message to the OOB server
[oobMessageManager sendWithMessage:message completionHandler:
    ^(id<EMOobMessageResponse> aResponse, NSError *anError) {
        // Check the NSError object first
        // ...

        // Get the result from response
        if ([aResponse resultCode] == EMOobResultCodeSuccess) {
            // send message success
        }
        else {
            // send message fail
        }
}];
```

Acknowledge to a Message

The backend server may request an acknowledgment from the mobile client. In this case, a specific property is set in the message.
The following code snippet shows how to check if the backend server has requested an acknowledgment to its message.

- On Android:

```
// fetch a message from the OOB server, wait at most 60 seconds
OobFetchMessageResponse response = oobMessageManager.fetchMessage(60);
if (response.isSucceeded()) {
    // fetch message success
    OobIncomingMessage oobIncomingMessage = response.getOobIncomingMessage();

    if (oobIncomingMessage.isAcknowledgmentRequested()) {
        // ... do any application treatment before acknowledge ...
        OobResponse acknowledgeResponse = oobMessageManager
            .acknowledgeMessage(oobIncomingMessage);
        if (response.isSucceeded()) {
            // acknowledge message success
        } else {
            // acknowledge message failed
        }
    }
} else {
    // fetch message fail
}
```

- On iOS:

```
// fetch a message from the OOB server, wait at most 60 seconds
[oobMessageManager fetchWithTimeout:60 completionHandler:
        ^(id<EMOobFetchMessageResponse> aResponse, NSError* anError) {

        // Check the NSError object first
        // ...


        // Get the result from response
        if ([aResponse resultCode] == EMOobResultCodeSuccess) {
            id<EMOobIncomingMessage> oobIncomingMessage = [aResponse
oobIncomingMessage];
            if([oobIncomingMessage isAcknowledgmentRequested]) {
                // ... do any application treatment before acknowledge ...
                [oobMessageManager acknowledgeWithMessage:oobIncomingMessage
                                        completionHandler:^(id<EMOobResponse>
aResponse, NSError* anError) {
                    // Check the NSError object first
                    if ([aResponse resultCode] == EMOobResultCodeSuccess) {
                       // acknowledge message success
                    }
                   else {
                       // acknowledge message failed
                   }
                }];
            }
        }
        else {
               // Handle error code
        }
    }];
```

Response to a Message

Ezio Mobile SDK provides a simple way to generate a response from the provider to a user. A provider-to-user incoming message contains a factory to create a response with certain predefined message fields.

- On Android:

```
// fetch a message from the OOB server, wait at most 60 seconds
OobFetchMessageResponse response = oobMessageManager.fetchMessage(60);
if (response.isSucceeded()) {
    // fetch message success
    OobIncomingMessage oobIncomingMessage = response.getOobIncomingMessage();

    if (oobIncomingMessage.getMessageType() ==
OobIncomingMessageType.USER_MESSAGE) {
        OobProviderToUserMessage providerToUser = (OobProviderToUserMessage)
response;
        OobUserToProviderMessage message = providerToUser.createResponse(
                providerToUser.getLocale(), /* Locale */
                new Date(),
                core.getSecureContainerFactory().fromString("Response"), /*
Subject */
                null, /* From */
                "text/plain", /* ContentType */
                messageContent, /* Content */
                null, /* Attachments */
                null /* Meta data */ );

        OobMessageResponse sendResponse =
oobMessageManager.sendMessage(message);
        if (sendResponse.isSucceeded()) {
             // send message success
        } else {
             // send message fail
        }
    }
} else {
     // fetch message fail
}
```

- On iOS:

```
// fetch a message from the OOB server, wait at most 60 seconds
[oobMessageManager fetchWithTimeout:60 completionHandler:
     ^(id<EMOobFetchMessageResponse> aResponse, NSError *anError) {
    // Check the NSError object first
    // ...

    // Get the result from response
    if ([aResponse resultCode] == EMOobResultCodeSuccess)
    {
        id<EMOobIncomingMessage> oobIncomingMessage = [aResponse
oobIncomingMessage];
        if([oobIncomingMessage messageType] ==
EMOobIncomingMessageTypeUserMessage) {
            id<EMOobProviderToUserMessage> providerToUser =
(id<EMOobProviderToUserMessage>) oobIncomingMessage;
            id<EMOobUserToProviderMessage> response = [providerToUser
                  createResponseWithLocale:[ providerToUser locale]
                           createTime:[[NSDate date] timeIntervalSince1970]
                              subject:@"Purchase"
                                 from:@"John Smith"
                          contentType:@"text/plain"
                              content:[[context secureDataFactory]

secureStringWithString:@"SHVtbW0sIHlvdSBhcmUgdmVyeSBjdXJpb3VzIDotKQ=="]
                          attachments:nil
                                 meta:nil];

            [oobMessageManager sendWithMessage:response
completionHandler:^(id<EMOobMessageResponse> aResponse, NSError *anError) {
                // Check the NSError object first
                // ...

                if ([aResponse resultCode] == EMOobResultCodeSuccess) {
                    // send message success
                }
                else {
                    // send message fail
                }
            }];
        }
    }
    else {
             // Handle error code
    }
}];
```

Response to a Transaction Signing Message

When the provider sends a transaction signing message, the end user has to return a response either to approve or reject the transaction.

- On Android:

```
// fetch a message from the OOB server, wait at most 60 seconds
oobMessageManager.fetchMessage(60, new OobFetchMessageCallback() {
    @Override
    public void onFetchMessageResult(OobFetchMessageResponse response) {
        if (response.isSucceeded()) {
            // fetch message success
            OobIncomingMessage oobIncomingMessage =
response.getOobIncomingMessage();

            if
(oobIncomingMessage.getMessageType().equals(OobIncomingMessageType.TRANSACTION_
SIGNING)) {
                OobTransactionSigningRequest oobTransactionSigningRequest =
(OobTransactionSigningRequest) response;
                SecureByteArray mspFrame =
oobTransactionSigningRequest.getMspFrame();

                // – Generate OTP starts –
                // Refer the MSP Parser module for generating the OTP from MSP
Frame  //otp = ...
                // – Generate OTP ends –

                OobTransactionSigningResponse oobTransactionSigningResponse =
null;
                try {
                oobTransactionSigningResponse = oobTransactionSigningRequest
createResponse(OobTransactionSigningResponse.

OobTransactionSigningResponseValue.ACCEPTED,

otp, meta);
                } catch (OobException e) {
                   //handle OobException
                }
                finally {
                }

                oobMessageManager.sendMessage(oobTransactionSigningResponse,
new OobSendMessageCallback() {
                    @Override
                    public void onSendMessageResult(final OobMessageResponse
response) {
                        if (response.isSucceeded()) {
                            // send message success
                        } else {
                             // send message fail
                        }
                    }
                });
            }
        } else {
            // fetch message fail
        }
    }
});
```

- On iOS:

```
// Get a message from the OOB server, wait at most 60 seconds
[oobMessageManager fetchWithTimeout:60
completionHandler:^(id<EMOobFetchMessageResponse> aResponse, NSError *anError)
{
    // Check the NSError object first
    // ...
    // Get the result from response
    if ([aResponse resultCode] == EMOobResultCodeSuccess)
    {
        id<EMOobIncomingMessage> oobIncomingMessage = [aResponse
oobIncomingMessage];
        if([[oobIncomingMessage messageType]  isEqual:
EMOobIncomingMessageTypeTransactionSigning])
        {
            id<EMOobTransactionSigningRequest> transactionSigningRequest =
(id<EMOobTransactionSigningRequest>)oobIncomingMessage;
            id<EMSecureByteArray> mspFrame =
transactionSigningRequest.mspFrame;

            //  --- Generate OTP starts ---
            //  Refer the MSP Parser module for generating the OTP from MSP
Frame
            //otp = ...
            //  --- Generate OTP ends ---

            id<EMOobTransactionSigningResponse> response =
            [transactionSigningRequest
createWithResponse:EMOobTransactionSigningResponseValueAccepted otp:otp
meta:meta];
            [oobMessageManager sendWithMessage:response
completionHandler:^(id<EMOobMessageResponse> aResponse, NSError *anError) {
                // Check the NSError object first
                // ...

                // Get the result from response
                if ([aResponse resultCode] == EMOobResultCodeSuccess)
                {
                   // Do something
                }
                else
                {
                   // Handle error code
                }
            }];
        }
    }
    else
    {
     // Handle error code
    }
}]
```

Response to a Transaction Verification Message

When the provider sends a transaction verification message, the end user has to send a response to approve or reject the transaction.

- On Android:

```
// fetch a message from the OOB server, wait at most 60 seconds
OobFetchMessageResponse response = oobMessageManager.fetchMessage(60);
if (response.isSucceeded()) {
    // fetch message success
    OobIncomingMessage oobIncomingMessage = response.getOobIncomingMessage();

    if (oobIncomingMessage.getMessageType() ==
OobIncomingMessageType.TRANSACTION_VERIFY) {
        obTransactionVerifyResponse oobTransactionVerifyResponse =
          ((OobTransactionVerifyRequest)oobIncomingMessage).createResponse(
              OobTransactionVerifyResponseValue.ACCEPTED,
              null);
        OobMessageResponse sendResponse = oobMessageManager
            .sendMessage(oobTransactionVerifyResponse);
        if (sendResponse.isSucceeded()) {
            // send message success
        } else {
            // send message fail
        }
    }
} else {
     // fetch message fail
```

- On iOS:

```
// fetch a message from the OOB server, wait at most 60 seconds
[oobMessageManager fetchWithTimeout:60 completionHandler:
    ^(id<EMOobFetchMessageResponse> response, NSError *anError) {

        // Check the NSError object first
        // ...

        if ([response resultCode] == EMOobResultCodeSuccess) {
            id<EMOobIncomingMessage> oobIncomingMessage = [response
oobIncomingMessage];
            if([oobIncomingMessage messageType] ==
EMOobIncomingMessageTypeTransactionVerify) {
                id<EMOobTransactionVerifyResponse> response =
[(id<EMOobTransactionVerifyRequest>)oobIncomingMessage

createWithResponse:EMOobTransactionVerifyResponseValueAccepted meta:meta];

                [oobMessageManager sendWithMessage:response
completionHandler:^(id<EMOobMessageResponse> aResponse, NSError *anError) {
                    // Check the NSError object first
                    // ...
                    if ([aResponse resultCode] == EMOobResultCodeSuccess) {
                        // send message success
                    }
                    else {
                        // send message fail
                    }
                }];
            }
        }
        else {
            // Handle error code
        }
 }];
```

Message with Custom MIME Type

The out-of-band module provides the pluggability for customized MIME type. This allows the application maker to implement customized message and its handler/parser. By registering the OOB module, the incoming message byte array will be de-serialized by the customized handler so the message can be casted to the customized message type. Outgoing message are then serialized by the customized handler into byte array.

A registry of message handlers is set up in the OOB module. The message types predefined by the SDK will be registered the first time the `OobMessageHandlerRegistry` is created. Each message type and message handler is uniquely identified by its MIME type. To avoid having conflicts with the SDK, the following MIME type are reserved by the SDK:

```
message/vnd.gemalto.ezio.oob.ErrorReport_1.0+json
message/vnd.gemalto.ezio.oob.VerifyTransaction_1.0+json
message/vnd.gemalto.ezio.oob.VerifyTransactionResponse_1.0+json
message/vnd.gemalto.ezio.oob.UserMessage_1.0+json
```

As there will be more MIME types to be added to the SDK in the future, it is strongly recommended to avoid using the gemalto.ezio.oob in the prefix of your MIME type. The MIME type is used as the identifier for the message handler in the `OobMessageHandlerRegsitry`.
To enable a customized MIME type, perform the following steps:

1. Implement your custom message. It should be extended from `OobIncomingMessageBase` or `OobOutgoingMessageBase`, and it should implement the `OobIncomingMessage` or `OobOutgoingMessageinterface`.
2. Implement your custom message handler. It should implement `OobIncomingMessageHandler` or `OobOutgoingMessageHandler` interface or both if the same MIME type is used for your outgoing and incoming message.
3. Register your message handler to the registry before your message is fetched and sent.

The following code snippets guide you through a simple example which implements a customized fund transfer message.

- On Android:
  Firstly, the implementation of the message—getters of message type, MIME type and wipe have to be implemented, where:
  - The other fields are content-specific to your message.
  - The MIME type message specifies how your message is parsed whereas the message type is mainly used to check the type of message before doing a casting.

```
public class IncomingFundTransferMessage extends OobIncomingMessageBase
implements OobIncomingMessage {
    public static final String CUSTOM_MIME_TYPE =
"message.mimetype.custom";
    public static final String CUSTOM_MESSAGE_TYPE = "CUSTOM";

    private String recipient;
    private int transactionAmount = 0;
    private String from;
    private String transactionCurrency = "SGD";

    public IncomingFundTransferMessage() { }

    // getters/setters
    @Override
    public String getMessageMIMEType() {
        return CUSTOM_MIME_TYPE;
    }

    @Override
    public String getMessageType() {
        return CUSTOM_MESSAGE_TYPE;
    }

    @Override
    public void wipe() {
        // Do your wiping here if needed
        }
    }
```

- Secondly, the implementation of the message handler—Serialization is required for the outgoing message and de-serialization is required for the incoming message.

```
public class FundTransferMessageHandler implements OobOutgoingMessageHandler,
OobIncomingMessageHandler {
    @Override
    public OobIncomingMessage deserialize(String mimeType, byte[] content)
throws OobException {
        IncomingFundTransferMessage msg = new IncomingFundTransferMessage();
        try {
            JSONObject obj  = new JSONObject(new String(content));
            String recipient  = obj.getString("recipient");
            String from = obj.getString("from");
            int transactionAmount = obj.getInt("transactionAmount");
            String transactionCurrency = obj.getString("transactionCurrency");
            msg.setFrom(from);
            msg.setRecipient(recipient);
            msg.setTransactionAmount(transactionAmount);
            msg.setTransactionCurrency(transactionCurrency);
        } catch  (JSONException e)  {
            throw new OobException(e,  e.getMessage());
        }
        return msg;
    }
    @Override
    public byte[] serialize(OobOutgoingMessage message) throws OobException {
        OutgoingFundTransferMessage msg = (OutgoingFundTransferMessage)
message;
        JSONObject obj = new JSONObject();
        try {
            obj.put("recipient", msg.getRecipient());
            obj.put("from", msg.getFrom());
            obj.put("transactionAmount", msg.getTransactionAmount());
            obj.put("transactionCurrency", msg.getTransactionCurrency());
        } catch  (JSONException e)  {
            throw new OobException(e,  e.getMessage());
        }
        return obj.toString().getBytes();
    }
}
```

Lastly, to register your message handler with the registry.

Note:

The message handler can only be registered once. Ensure that it is already registered before adding or removing the handler from the registry.

```
// Register handler
OobMessageHanlderRegistry reg = OobMessageHanlderRegistry. getInstance ();
if (!reg().isRegistered(IncomingFundTransferMessage.CUSTOM_MIME_TYPE)) {
    reg.register(IncomingFundTransferMessage.CUSTOM_MIME_TYPE,
                        new FundTransferMessageHandler());
}
…
// Un-register handler
if (reg.isRegistered(IncomingFundTransferMessage.CUSTOM_MIME_TYPE)) {
    reg.unregister(IncomingFundTransferMessage.CUSTOM_MIME_TYPE);
}
```

- On iOS:
  Firstly, the implementation of the message—getters of message type, MIME type and wipe have to be implemented, where:
  - The other fields are content-specific to your message.
  - Message MIME type specifies how your message is parsed whereas the message type is mainly used to check the type of message before doing a casting.

The implementation of a .h file:

```
#define CUSTOM_MIME_TYPE        @"mesage.mimetype.custom"
#define CUSTOM_MESSAGE_TYPE     @"CUSTOM"

@interface IncomingFundTransferMessage : EMOobIncomingMessageBase
<EMOobIncomingMessage>

@property (nonatomic, strong) NSString* recipient;
@property (nonatomic, strong) NSString* from;
@property (nonatomic, strong) NSString* transactionCurrency;
@property (nonatomic) int transactionAmount;

@end
```

The implementation of a .m file:

```
#import "IncomingFundTransferMessage.h"

@implementation EMOobCustomIncomingFundTransferMessage

@synthesize recipient = _recipient;
```

```
@synthesize from = _from;
@synthesize transactionCurrency = _transactionCurrency;
@synthesize transactionAmount = _transactionAmount;

// your constructor or other methods

- (NSString *)messageMIMEType
{
    return CUSTOM_MIME_TYPE;
}
- (NSString *)messageType
{
    return CUSTOM_MESSAGE_TYPE;
}

-(void)wipe
{
    // Do your wiping here if needed
}
@end
```

Secondly, the implementation of the message handler—serialization is required for the outgoing message and de-serialization is required for the incoming message. The message handler implementation of a .h file.

```
#import <Foundation/Foundation.h>
#import "EMOobIncomingMessageHandler.h"
#import "EMOobOutgoingMessageHandler.h"
#import "EMOObIncomingMessage.h"
#import "EMOObOutgoingMessage.h"

@interface FundTransferMessageHandler : NSObject <EMOobIncomingMessageHandler,
EMOobOutgoingMessageHandler>

@end
```

The message handler implementation of a .m file.

```
#import "FundTransferMessageHandler.h"
#import "IncomingFundTransferMessage.h"
#import "OutgoingFundTransferMessage.h"
@implementation FundTransferMessageHandler
- (id<EMOobIncomingMessage>)deserializeWithMIMEType:(NSString *)mimeType
withContent:(NSData *)content withError:(NSError **)error
{
    id json = [NSJSONSerialization JSONObjectWithData:content options:0
error:error];
    IncomingFundTransferMessage* message = [[IncomingFundTransferMessage
alloc] init];
```

```
     message.recipient = [json objectForKey:@"recipient"];
     message.from = [json objectForKey:@"from"];
     message.transactionCurrency = [json objectForKey:@"transactionCurrency"];
     message.transactionAmount = [[json objectForKey:@"transactionAmount"]
intValue];

     return message;
}

- (NSData *)serializeMessage:(id<EMOobOutgoingMessage>)message
withError:(NSError **)error
{
     OutgoingFundTransferMessage* msg = (OutgoingFundTransferMessage*)message;
     NSMutableDictionary *jsonDict = [[NSMutableDictionary alloc] init];
     [jsonDict setValue:msg.recipient forKey:@"recipient"];
     [jsonDict setValue:msg.from forKey:@"from"];
     [jsonDict setValue:@(msg.transactionAmount) forKey:@"transactionAmount"];
     [jsonDict setValue:msg.transactionCurrency forKey:@"transactionCurrency"];

     NSData* messageContent = [NSJSONSerialization dataWithJSONObject:jsonDict
options:0 error:error];
     return messageContent;
}
@end
```

Lastly, to register your message handler with the registry.

Note:

The message handler can only be registered once. Ensure that it is already registered before adding or removing the handler from the registry.

```
// Register handler
EMOobMessageHandlerRegistry* reg = [EMOobMessageHandlerRegistry
sharedInstance];
if(![reg isRegistered:CUSTOM_MIME_TYPE])
{
     [reg registerMIMEType:CUSTOM_MIME_TYPE
withOobMessageHandler:[[FundTransferMessageHandler alloc] init]];
}
…
// Un-register handler
if([reg isRegistered:CUSTOM_MIME_TYPE])
{
 [reg unregisterMIMEType:CUSTOM_MIME_TYPE];
}
```

Multiple OOB Server Configurations

Ezio Mobile SDK allows user to configure multiple instances of OOB server in one application. Each OobManager could have its specific OOB server configuration.

Note:

Tls configuration, root policy and device fingerprint source are configured globally on core so it will be applied to all of OobManager. All these global configurations are optional, so if they are not configured, Ezio Mobile SDK will use the default values.

- On Android:

```
// Create Oob Global Configuration on Module level,
// this configuration can only be done once
OobConfiguration oobConfigGlobal = new OobConfiguration.
Builder().setTlsConfiguration(tlsConfiguration)
  .setRootPolicy(policy)
  .setDeviceFingerprintSource(getDeviceFingerPrint()).build();


IdpCore core = IdCore.configure(oobConfigGlobal);


// Create OobManager with Oob Server Configuration 1
OobManager oobManager1 = OobModule.createOobManager(new URL(OOB_CLIENT_URL),
              OOB_DEFAULT_DOMAIN,
              OOB_DEFAULT_APP_ID,
              publicKey);


// Create OobManager with Oob Server Configuration 2
OobManager oobManager2 = OobModule.createOobManager(new URL(OOB_CLIENT_URL_2),
              OOB_DEFAULT_DOMAIN_2,
              OOB_DEFAULT_APP_ID_2,
              publicKey_2);
```

- On iOS:

```
// Create Oob Global Configuration on Module level
// this configuration can only be done once
EMOobConfiguration *oobConfiguration = [EMOobConfiguration
configurationWithOptionalParameters:^(EMOobConfigurationBuilder
*configurationBuilder)
{
    configurationBuilder.deviceFingerprintSource = [[EMDeviceFingerprintSource
alloc] initWithDeviceFingerprintType:[NSSet setWithObject:
@(EMDeviceFingerprintTypeSoft)]];
    configurationBuilder.jailbreakPolicy = jailbreakPolicy;
    configurationBuilder.tlsConfiguration = [[EMTlsConfiguration alloc] init];
}];

[EMCore configureWithActivationCode:activationData configurations:[NSSet
setWithObject:oobConfiguration]];

EMOobModule _module = [[EMOobModule alloc] init];

// Create OobManager with Oob Server Configuration 1
id<EMOobManager> oobManager = [_module createOobManagerWithURL:[NSURL
URLWithString:conf.oobClientUrl]
                                                         domain:conf.domain

applicationId:conf.applicationId
                                                    rsaExponent:conf.rsaExponent
                                                     rsaModulus:conf.rsaModulus
                                                          error:&error];

// Create OobManager with Oob Server Configuration 2
id<EMOobManager> oobManager2 = [_module createOobManagerWithURL:[NSURL
URLWithString:conf.oobClientUrl2]
                                                          domain:conf.domain2

applicationId:conf.applicationId2

rsaExponent:conf.rsaExponent2

rsaModulus:conf.rsaModulus2
                                                           error:&error];
```

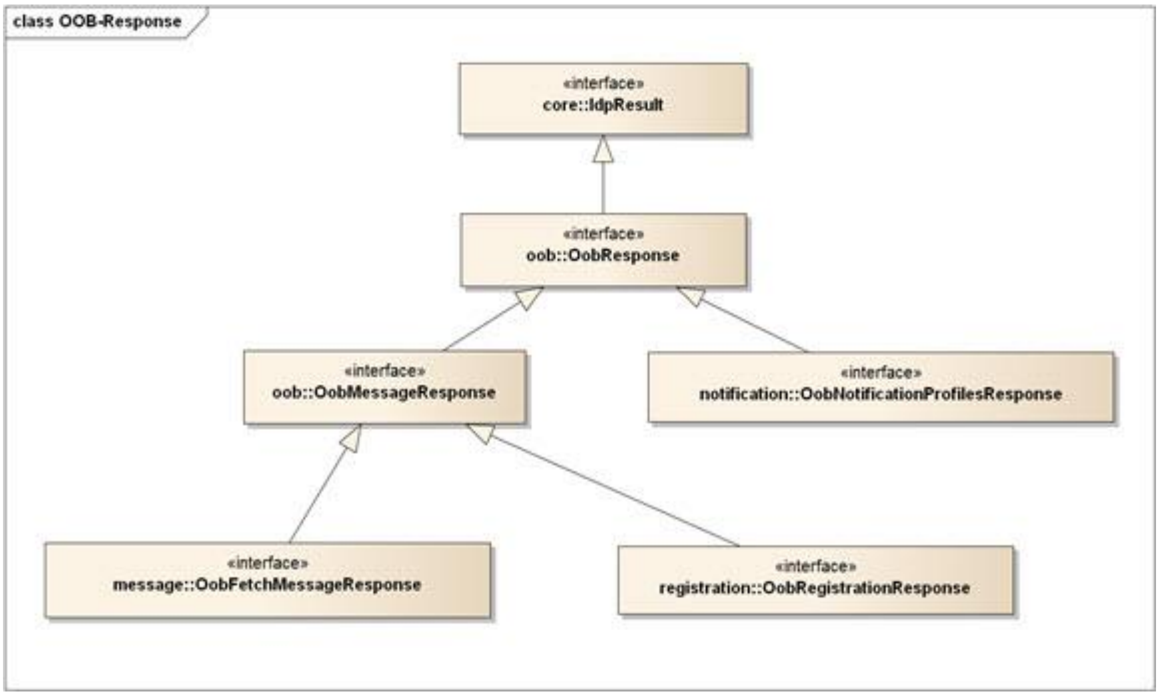
Understanding the Results

The results of the operations are reported to the client via result objects. Result objects have generic properties on all supported platforms:

- Result code
- Result domain
- Result message which contains the technical description of the result code (not localizable).
- User info which contains additional unstructured data giving much more technical details about the problem which occurred
  - The user info is a key-value pairs dictionary.
  - The content (keys) is defined by the subsystems.
  - It is intended merely for the debugging and support use cases, that is, the client application should not act/depend on this user info.

Each Ezio Mobile SDK operation can be either a success or failure. The reason for the failure can be distinguished by the result code. The following figure shows the structure of the results object on Android. A similar structure is developed for iOS.

Figure 22 Hierarchy for OobResponse Objects



The error handling slightly differs between the two platforms, refer to the subsequent chapters for more details of the respective platform.
On Android, the Android response object system is built based on the IdpResult interface exposing a few common properties:

- Code: The result code of the operation.
  - `int getCode();`
  - All possible codes are defined by `OobResultCode`.
- Domain: The source of the result
  - `int getDomain();`
  - All known domains are defined by `OobResultDomain`.
- Message: The technical message detailing the result.
  - `String getMessage();`
  - It is not localized.
- Exception: The exception which caused the failure
  - `Exception getException();`
  - Not all results are caused by an exception so the application has to be able to handle the null value anytime.
  - It is intended mostly for the debugging purposes as it can be used to print the stack trace.
- User info: The additional technical data that contain more details about the occurrence of the problem.
  - `Object getUserInfo(final String key);`To obtain the specified user info data item.
  - `Set<String> getUserInfoKeys();`
    To iterate all the user info data items that are currently available.

Subsequently, the result objects are further handled the same way as how the same base interface is implemented. Error delivered by the callback parameter of the OOB API registration call is as follows:

```
// register user asynchronously
oobRegistrationManager.register(oobRegistrationRequest, new
OobRegistrationCallback(){
  @Override
  public void onOobRegistrationResponse(OobRegistrationResponse response) {
  // check the response
  if (response.isSucceeded()) {
   // display registration successful
  } else {
 // handle the fail or exception here
    // (e.g. open the dialog, log the errors)
 if (response.getException() != null) {
     Log.e("EZIO REGISTRATION EXCEPTION", response.getMessage(),
response.getException());
 } else {
   Log.e("EZIO REGISTRATION FAIL", "Domain: " + response.getDomain().toString());
   Log.e("EZIO REGISTRATION FAIL", "Message:" + response.getMessage());   }
     // retrieve all the extra user info from the result object
      Set<String> userInfoKeys = response.getUserInfoKeys();
      for (String key : userInfoKeys) {
     Log.e(key, response.getUserInfo(key).toString());
       }
   // get one specific extra user info - OOB status message in this case
      String oobStatusMessage = response.getUserInfo(OobResponse.
     USER_INFO_KEY__OOB_STATUS_MESSAGE).toString();
   // continue to show the message e.g. in the error dialog
   }
  }
});
```

Note:

The `OobRegistrationResponse` implementes the IdpResult interface.

On iOS, in the out-of-band part, the result object in iOS contains a `NSError` class that is available in Foundation framework. An `NSError` object encapsulates richer and more extensible error information than using only an error code or error string. The core attributes of an `NSError` object—an error domain (represented by a string), a domain-specific error code, and a user info dictionary containing the application-specific information.

- code: The result code of the operation.
  - @property (readonly) NSInteger code;
  - All possible codes are defined by the EMOobErrorMessageCode enumeration.
- domain: The source of the result.
  - @property (readonly, copy) NSString *domain;
  - All known domains are defined in EMErrorDomain.h
- userInfo: The dictionary which contains the technical message detailing the result.
  - @property (readonly, copy) NSDictionary *userInfo;
  - It is not localized.

Error messages are delivered by the NSError object and call back parameter.

```
// register user asynchronously
[oobRegistrationManager registerWithRequest:oobRegistrationRequest
completionHandler:^(id<EMOobRegistrationResponse> aResponse, NSError *anError) {

        // Check the NSError object first
        if (anError != NULL) {
            NSLog(@"Error code: %ld", (long)[anError code]);
            NSLog(@"Error message: %@", [anError description]);
            NSLog(@"Error domain: %@", [anError domain]);

            // Retrieve all extra user info from the error
            NSDictionary* usrInfo = [anError userInfo];
            for(NSString * key in usrInfo) {
                NSLog(@"Key: %@  Message: %@", key, [usrInfo valueForKey:key]);
            }

            // Get one specific user info
            NSString * oobExceptionMessage = [usrInfo valueForKey:
@"OOB_EXCEPTION"];
        }
        else {
            if ([aResponse resultCode] == EMOobResultCodeSuccess) {
                 // display registration successful
            }
            else {
                 // Get the result from response
                NSLog(@"EZIO REGISTRATION FAIL: %@", [aResponse resultMessage]);
           }
        }
    }];
```

Request with Custom Headers

To extend the out-of-band features with custom headers, request parameter objects are used. These can be injected into all individual requests.

Note:

The out-of-band server does not handle any custom headers, they are handled explicitly by the integrator of the backend.

The custom headers feature applies to:

- Registration/unregistration
- Messaging functionalities
- Notification profile management

Refer to for more information.
Hence, all APIs communicating with the server now include a parameter `customHeader`.

- On Android:

```
// Setup custom headers
Map<String, SecureString> header = new HashMap<>();
header.put(headerKey, headerValue);
OobRequestParameter requestParameter = new
OobRequestParameter.Builder().setCustomHeader(header).build();
```

- On iOS:

```
// Setup custom headers
EMOobRequestParameter *oobRequestParameter =
    [EMOobRequestParameter
configurationWithOptionalParameters:^(EMOobRequestParameterBuilder
*paramBuilder) {
        NSString *customHeader = @"customHeader";
        id<EMSecureString> customHeaderValue = [secureContainerFactory
secureStringFromString:@"Value"];
        NSDictionary *dic = [NSDictionary
dictionaryWithObject:customHeaderValue forKey:customHeader];
        paramBuilder.customHeaders = dic;
    }];
```

OOB Module Reset

Note:

OOB Module reset is currently available only on Android platform.

This is used to remove the data associated with the given application ID. The clients registered under this application ID can no longer be used after the reset.
On Android:

```
otpModule.reset(applicationId);
```

Secure Storage

This chapter defines the feature set to the secure storing of the arbitrary application data. An application stores the data in this module using key-value pairs. Secure Storage supports the creation of multiple storages with unique identifiers, with each storage storing multiple key-values.

Application Requirements

Logging In to the Password Manager

The application has to be logged in via the Password Manager in order to use the SecureStorage module. Log out from the Password Manager once you are finished using the SecureStorage functionalities.

Storing Data

The Secure Storage module is the entry point to the usage of the secure storage. It provides access to the Secure Storage Manager which in turn provides the functionality to create and destroy storages. The storage is protected by a password, as well as a device fingerprint.

Creating Storage Instance

Creating or opening a storage instance, you have to first fetch the Property Storage by identifier and open it. The device fingerprint is also can be configured at the time the storage instance is created. If the device fingerprint is not configured, the default fingerprint source settings will be applied.

- On Android: SERVICE and SOFT. See constraints on Android
- On iOS: SOFT. See constraints on iOS

Note:

- The database is only created the first time the data is written to it.
- See the Password Manager requirement in Password Manager.

- On Android:

```
SecureStorageModule secureStorageModule = SecureStorageModule.create();
SecureStorageManager ssManager = secureStorageModule.getSecureStorageManager();
try {
     // Log in to passwordManager to unlock secureStorage feature
     passwordManager.login(password);
     // user define a finger print source
     DeviceFingerprintSource dfs= DeviceFingerprintSource.DEFAULT;
     // get the reference to PropertyStorage
     PropertyStorage storage = ssManager.getPropertyStorage(identifier, dfs);
    ...
     // open the storage
     storage.open();
     ...
  } catch (IdpSecureStorageException sse) {
     // Handle the exception } catch (PasswordManagerException pme) {
     // Handle the exception } catch (DeviceFingerprintException dfe) {
 // Handle the exception
}
```

- On iOS:

```
// Init the secure storage module and get secure storage manager
EMSecureStorageModule* ssModule = [EMSecureStorageModule secureStorageModule];
id<EMSecureStorageManager> secureStorageManager = [ssModule
secureStorageManager];

// Log in to passwordManager to unlock secureStorage feature
[passwordManager loginWithPassword:aPassword error:&error];

// get property storage using given parameters
id<EMPropertyStorage> storage = [secureStorageManager
propertyStorageWithIdentifier:identifier error:&error];

// open the storage
[storage open:&error];
```

Reading, Writing and Deleting Data

Property Storage is a type of storage that represents a single storage DB. It is created from Secure Storage Manager. The following example continues the example in the previous section. For security reasons, remember to close the storage and logout after using it. When the stored data is not needed anymore, the storage can be destroyed from Secure Storage Manager. This will wipe off all the data in the storage.

- On Android:
  Reading, writing and deleting key-value pairs:

```
try {
    // Write data to the storage
    storage.writeProperty(key, data, false);

    // Read data from the storage
    SecureByteArray readData = storage.readProperty(key);

    // Delete data from the storage
    storage.deleteProperty(key);
} catch (IdpSecureStorageException sse) {
     // Handle the exception
} catch (PasswordManagerException pme) {
     // Handle the exception
} catch (DeviceFingerprintException dfe) {
     // Handle the exception
}
```

Closing and destroying the storages, and logging out of Password Manager:

```
try {
    // Close the storage
    storage.close();

    // Destroy the storage which wipes all data
    ssManager.destroyPropertyStorage(identifier);

    // Logout
    passwordManager.logout();

} catch(PasswordManagerException pme){
    // Handle the exception
} catch(IdpSecureStorageException sse){
    // Handle the exception
}
```

Resetting a storage regardless of the state of Password Manager or Storage's. This is used when the user forgets the password to the Password Manager or the application has entered an unrecoverable state: DeviceFingerprint is altered.

```
try {
    // Destroy the storage which wipes all data
    secureStorageModule.reset(identifier);

} catch(IdpSecureStorageException sse){
    // Handle the exception
}
```

- On iOS:
  Reading, writing and deleting key-value pairs:

```
// write data to storage
[storage setProperty:data forKey:key wipeValue:YES error:&error];

// read data from storage
id<EMSecureByteArray> data = [storage propertyForKey:key error:&error];

// delete data from the storage [
storage removePropertyForKey:key error:&error];
```

Closing and destroying storages, and logging out of Password Manager:

```
// close the storage
[storage close:&error];

// destroy the storage
[secureStorageManager destroyPropertyStorageWithIdentifier:identifier
error:&error];

// logout
[passwordManager logout:&error];
```

SecureStorage Module Reset

SecureStorage Module reset is provided on Android platform only at the moment. It removes the data associated with the given storage identifier.
On Android:

```
try {
 secureStorageModule.reset(identifier);
} catch (IdpStorageException e) {
// handle error
}
```

User Interface

This chapter defines the feature set for the user interface.

Secure Input Service

Secure Input service provides a single visual view used for the secure input of data. The purpose of the Secure Input service is to ensure that the management and manipulation of sensitive data are done in a controlled and secure way, where:

- "Controlled" means that the input data in the memory is protected and copies are managed to ensure that when it is no longer needed or when the reference goes out of scope, it is wiped off from the memory.
- "Secure" means that the protection mechanisms are in place to defeat or mitigate keylogger, overshoulder attacks, memory dump and screen capture. Secure Input makes it more difficult to carry out targeted attacks to the user.

Note:

- The secure input service is an optional feature which is not enabled by default. Contact your Gemalto sales representatives for the activation code to enable this feature.
- Two sets of API are available: SecureInputBuilder and SecureInputBuilderV2. SecureInputBuilder API has been marked as deprecated and will be removed in the future release but it is strongly recommended to use the SecureInputBuilderV2 API.

Application Requirements

Activation Code

By default, this feature is not enabled in Ezio Mobile SDK. Hence, the application have to provide an activation code when configuring the SDK in order to use it. Only Gemalto can provide the activation codes, refer to your Gemalto representative for more information.

Secure Input Builder Version 1 API

Although SecureInputBuilder has been marked as deprecated in Ezio Mobile SDK V4.4, these APIs are still available for backward compatibility. It is strongly recommended to use the SecureInputBuilderV2 API.

Customizing the PIN Pad V1 API

During the initialization, the following elements can be configured as parameters or through the API offered by the SDK.

Table 21 Customizing the PIN Pad Elements

Configuration Element	Description
Scramble / Unscramble Mode	Determines if the position of the characters is randomized or in the standard layout.
Double Password Entry Mode	Determines if the SDK shows two text input fields instead of the standard single field. This mode is specially designed for users changing their PINs through mobile devices, used when validating the new PIN. The input data in the two fields need to be identical. The validation of the two passwords has to be done by SDK user.
Show Secure Pin Pad as Dialog or View (Android Only)	Configures whether the keypad is shown as a Dialog (Popup) or as a View (FullScreen).
Character Choices	The character set indicating the possible characters that can be displayed. Only digits are currently supported.
Button Spacing	The spacing between the buttons.
Background Color	The background color of the keypad.
Character Color Normal	The character color on the buttons.
Character Color Selected	The character color on the buttons in selected state.
Character Color Disabled	The character color on the buttons in disabled state (special buttons only).
Button Gradient Normal Start	The start and end of the gradient color for the button in RGBA format. If the start and end color are same, the button appears in a single color.
Button Gradient Normal End	
Button Gradient Selected Start	The start and end of the gradient color for the button when tapped, in RGBA format. If the start and end color are same, the button appears in a single color.
Button Gradient Selected End	
Button Gradient Disabled Start	The start and end of the gradient color for the button in disabled state, in RGBA format. If start and end color are same, the button appears in a single color.
Button Gradient Disabled End	
Text Color	The color of the text label in RGBA format. This color can also be used to display the asterisk characters in the text input field.
Text Box Border Color Unfocussed	The color of the text border when it is unfocused or in an unselected condition.

Configuration Element	Description
Text Box Border Color Focussed	The color of the text border when it is focused or in selected condition.
First Text Label	The text displayed on the label above the input field(s).
Second Text Label	
Number of Rows	The grid array size for the keypad, indicating the number of rows and columns to be displayed.
Number of Columns	
Minimum Input Length	The minimum length of the input text to enter.
Maximum Input Length	The maximum length of the input text to enter.
Automatic enabled OK	Determines if the **OK** button is enabled when the minimum length of the input text is reached, otherwise the **OK**button is enabled all the time.
Automatic OK	Determines if the **OK** button is automatically triggered when the maximum length of the input text is reached after the last digit is entered, otherwise the **OK** button is only triggered when the user taps it manually (the **OK** button is not displayed in this case).
Visible Button Presses	Determines if the button is highlighted when tapped. If it is disabled, no highlighting effect appears. This is set to "true" by default.
Pin Pad Width	The width of the PIN pad.
Pin Pad Height	The height of the PIN pad.
Pin Pad Dialog Width (Android Only)	Sets the ratio for the width of the PIN Pad dialog with respect to the screen width. This method only applies for PIN Pad shown as dialog.
Pin Pad Dialog Height (Android Only)	Sets the ratio for the width of the PIN Pad dialog with respect to the screen height. This method only applies PIN pad shown as dialog.
Encoding	Sets the secure PIN Pad encoding format.

Warning:

When isAutomaticOK is set to "true", the minimum input length and maximum input length have to be explicitly set to be equal. Otherwise an exception will be thrown.

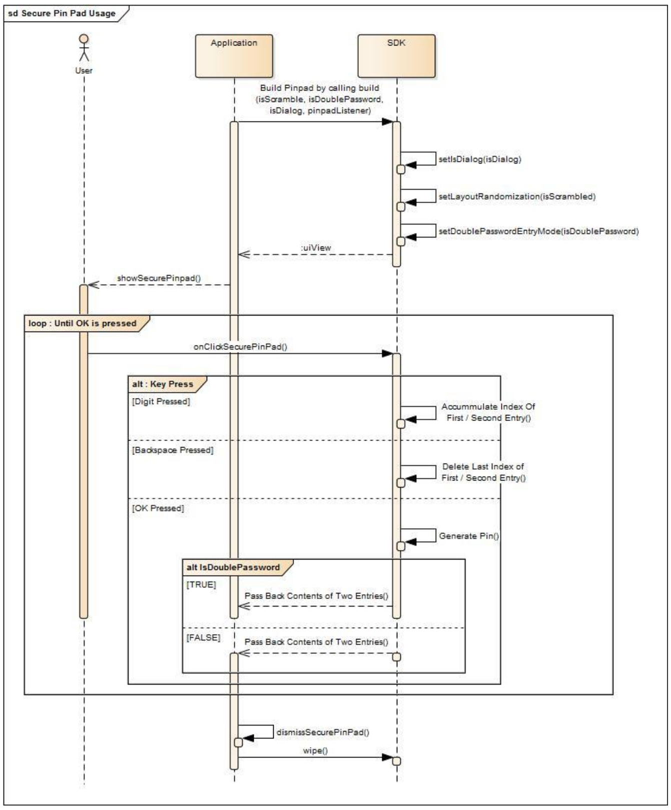
Implementing the PIN Pad

The methods and functions for initializing and customizing the PIN pad vary with different platforms.

Sequence of PIN Pad Usage

The following figure shows the interaction between the different actors of the PIN pad usage. The methods and functions for initializing and customizing the PIN pad vary with different platforms.

Figure 23 Sequence of Secure PIN Pad Use Case


sd Secure Pin Pad Usage
User
Application
SDK
Build Pinpad by calling build (isScramble, isDoublePassword, isDialog, pinpadListener)
setIsDialog(isDialog)
setLayoutRandomization(isScrambled)
setDoublePasswordEntryMode(isDoublePassword)
:uiView
showSecurePinpad()
loop : Until OK is pressed
onClickSecurePinPad()
alt : Key Press
[Digit Pressed]
Accummulate Index Of First / Second Entry()
[Backspace Pressed]
Delete Last Index of First / Second Entry()
[OK Pressed]
Generate Pin()
alt IsDoublePassword
[TRUE]
Pass Back Contents of Two Entries()
[FALSE]
Pass Back Contents of Two Entries()
dismissSecurePinPad()
wipe()

Android Interface

As shown in the sequence diagram in Figure 1, SecureInputBuilder needs to be initialized first.
The single instance of the Secure Input builder can be retrieved from the SecureInputServiceinstance.

```
UiModule uiModule = UiModule.create();
SecureInputService secureInputService = SecureInputService.create(uiModule);
SecureInputBuilder pinpadBuilder = secureInputService.getSecureInputBuilder();
```

If the application supports orientation change, in the application manifest file, add the following codes to the Activities which uses secure PINpad. This prevents the activity from restarting when the screen orientation changes or keyboard availability changes.

```
android:configChanges="orientation|keyboardHidden"
```

SecurePinpadListener needs to be implemented in the same class that displays secure PINpad. A sample is shown as follows:

```
public class MainActivity extends FragmentActivity implements SecurePinpadListener
```

The application can get the callback value from onFinish method after implementing SecurePinpadListener. This overriding method will be called when the user has finished entering the PIN on the keypad. In the single password mode, the secondPin is null. The callback value can be used to change the PIN or to generate OTP.

Note:

The PINs returned by secure PINpad are presented as PinAuthInput objects, and the clear text password cannot be extracted from a PinAuthInput object.

```
@Override
public void onFinish(PinAuthInput firstPin, PinAuthInput secondPin) {
    // Call the wipe method to clear the some sensitive data from the pinpad,
    // If you need to display another pinpad inside this callback method,
    // the wipe() must be called before you call the
    // pinpadBuilder.build(...) method.
    // The other configuration like color and text label won’t be changed here.
    pinpadBuilder.wipe();
    // dismiss pinpad, you can write your own dismiss method.
    // You has to dismiss it since the dialogFragment cannot be reused.
    dismissPinpad();
    // Now you can use the pin object by passing it to the sdk.
    // To use it again (e.g. in use case of changing PIN),
    // you have to wipe and dismiss the current keypad,
    // and then create a new dialogFragment from pinpad object.
}
```

Ensure that the PIN objects and the Secure Pinpad instances are wiped before dismissing the secure PINpad.

Note:

If the application uses more than one instance of secure PINpad, you have to wipe off any existing instance first before initializing a new one. Shared PIN pad resources are cached globally which may result in unexpected behavior when instances exist at the same time.

There are two ways to create and show the secure PINpad in Android. The secure PINpad could be shown as a Dialog or a View. The orientation of the PIN pad follows the setting of the activity that displays the secure PINpad.

SecurePinPad as Dialog

You have to set IS_DIALOG to "true" in the application when creating the SecurePinPadBuilder.

The following arguments are mandatory when creating DialogFragment:

```
// Show double pin mode, enter 2 pins in one keypad
boolean IS_DOUBLE_PIN = false;
// Show scramble, randomly arrange the input characters
boolean IS_SCRAMBLE = true;
// Show as Dialog (Pop Up)
Boolean IS_DIALOG = true;
```

These are some of the optional configurations that can be set to customize the secure PINpad. All these settings have to be done before DialogFragment is created.

```
// all the settings has to be done before creating the dialogFragment
pinpadBuilder.setScreenBackGroundColor(getResources().getColor(R.color.pinpad));
pinpadBuilder.setDialogHeightToScreenRatio(0.6);
pinpadBuilder.setDialogWidthToScreenRatio(0.6);
pinpadBuilder.setTextBorderFocusColor(Color.parseColor("#7700FF00"));
pinpadBuilder.setTextBorderUnfocusColor(Color.parseColor("#FF0000FF"));


// more settings...
```

Creating and displaying the DialogFragment:

```
DialogFragment dialogFragment = pinpadBuilder.build(IS_SCRAMBLE, IS_DOUBLE_PIN,
IS_DIALOG, MainActivity.this);


// android.support.v4.app.FragmentManager
FragmentManager mFragmentManager = getSupportFragmentManager();


// Show the pinpad
dialogFragment.show(mFragmentManager, "SECURE PIN");
```

SecurePinPad as View

You have to set set IS_DIALOG to "false" in the application when creating the secure PINpad.

The following code snippet shows the mandatory settings:

```
// Show double pin mode, enter 2 pins in one keypad
boolean IS_DOUBLE_PIN = false;

// Show scramble, randomly arrange the input characters
boolean IS_SCRAMBLE = true;

// Show as Dialog (View)
Boolean IS_DIALOG = false;
```

Since secure PINpad is a Fragment, it has to be contained in a FrameLayout provided by the application.

```
FrameLayout mFrameLayout = (FrameLayout)findViewById(R.id.pinpad_framelayout);
mFrameLayout.setVisibility(View.VISIBLE);
```

The PIN pad does not scale automatically to fit its container. The application has to provide the width and height when the DialogFragment is first created. Ideally, the PIN pad is set to match the dimension of the FrameLayoutthat is provided by the application. In the Android application, the actual dimension of a UI component is not readily available during onCreate() or onStart(). In fact, it will only be available after the onMeasure and onLayout events have completed. In order to catch the onLayout event, OnGlobalLayoutListener is used as follows:

```
ViewTreeObserver viewTreeObserver = mFrameLayout.getViewTreeObserver();
if (viewTreeObserver.isAlive()) {
    viewTreeObserver.addOnGlobalLayoutListener(new OnGlobalLayoutListener() {
 @Override
 public void onGlobalLayout() {


 // remove the listener to avoid receiving further event
    mFrameLayout.getViewTreeObserver().removeGlobalOnLayoutListener(this);
            // set the dimensions
    int height = mFrameLayout.getHeight();
    int width = mFrameLayout.getWidth();
    pinpadBuilder.setKeypadHeight(height);
    pinpadBuilder.setKeypadWidth(width);


 // additional settings...
 pinpadBuilder.setTextBorderFocusColor(Color.parseColor("#7700FF00"));
 pinpadBuilder.setTextBorderUnfocusColor(Color.parseColor("#FF0000FF"));


 // build the fragment
 dialogFragment = pinpadBuilder.build(IS_SCRAMBLE,
   IS_DOUBLE_PIN, IS_DIALOG,
   MainActivity.this);
      FragmentTransaction fragmentTransaction =
   mFragmentManager.beginTransaction();
      fragmentTransaction.add(mFrameLayout.getId(),dialogFragment);
      fragmentTransaction.commit();
 }
    });
}
```

iOS Interface

Except for the layout configuration, the secure PINpad use case on iOS is similar to that of Android. The PIN pad is automatically configured with a set of default values created by Ezio Mobile SDK. It can be further customized before it is displayed.

An instance of the secure PINpad can be retrieved from the UI module that has been initialized with a valid activation code:

```
EMUIModule *uiModule = EMUIModule uiModule];
EMSecureInputService *secureInputService = [EMSecureInputService
                                            serviceWithModule:uiModule];
id<EMSecureInputBuilder> securePinpadBuilder = [secureInputService
                                                secureInputBuilder];
```

The following code snippet demonstrates a custom configuration that can be performed:

```
securePinpadBuilder setScreenBackgroundColor:EMUIColorFromRGBA(0xFFFFFFFF)];
[securePinpadBuilder setButtonCharacterColorNormal:EMUIColorFromRGBA(0x000000FF)];
[securePinpadBuilder setButtonCharacterColorDisabled:EMUIColorFromRGBA(0x000000FF)];
[securePinpadBuilder setButtonSpacing:2];
[securePinpadBuilder setButtonGradientNormalStart:EMUIColorFromRGBA(0xDDDDDDFF)];
[securePinpadBuilder setButtonGradientNormalEnd:EMUIColorFromRGBA(0xCCCCCCFF)];
[securePinpadBuilder setButtonGradientDisabledStart:EMUIColorFromRGBA(0xDDDDAAFF)];
[securePinpadBuilder setButtonGradientDisabledEnd:EMUIColorFromRGBA(0xCCCCA0FF)];
```

```
[securePinpadBuilder setLabelTextColor:EMUIColorFromRGBA(0x000000FF)];
[securePinpadBuilder setTextBoxBorderFocussedColor:[UIColor greenColor]];
[securePinpadBuilder setTextBoxBorderUnFocussedColor:[UIColor blueColor]];
[securePinpadBuilder setMinimumInputLength:4];
[securePinpadBuilder setMaximumInputLength:8]
```

The current version of secure PINpad on iOS does not allow the customization of size. The PIN pad takes up the whole screen to display. However, the orientation of the device is supported and the PIN pad automatically rotates if the orientation is supported by the application.

Finally, the PIN pad is displayed and the callback is handled as demonstrated in the following:

```
//Display text
[pinpadBuilder setTextLabel:@"Enter your PIN"];

// Create the pin pad
UIViewController *pinPad = [pinpadBuilder
                            buildWithScrambling:YES
                            doublePin:NO
                            onFinishBlock:^(id<EMPinAuthInput> firstPin,
                                            id<EMPinAuthInput> secondPin) {

                // Dismiss the keypad
                [self.navigationController popViewControllerAnimated:YES];

                // Generate OTP
                id<EMSecureString> otp;
                otp = [self generateOTPWithPinAuthInput:firstPin\];

                // Wipe the pinpadBuilder for security reason
                [pinpadBuilder wipe];
           }];

// Push the pin pad to navigation controller
[self.navigationController pushViewController:pinPad animated:YES\];
```

The PIN pad is returned as a UIViewController object that manages the lifecycle of the PIN pad itself. The presentation of UIViewControlleris decided by the application developer using either pushed into a UINavigationController stack or shown as a modal view controller.

Same as the callback in Android, the clear text PINs returned by the secure PINpad cannot be extracted from the EMSecurePin objects.

The secure PINpad object is wiped off when it gets deallocated by the autorelease pool. However, in order to enhance security, it is recommended to call wipe prematurely when the no secure PINpad object longer needed.

Secure Input Builder Version 2 API

SecureInputBuilderV2 API is added since the Ezio Mobile SDK V4.4 release. With these APIs, you can:

- Turn off secure keypad default top screen and develop the customized top screen.
- Further customize the appearance of the secure keypad.

- Implement a dialog mode in iOS.
- Implement an embedded mode in iOS.

Customizing the PIN Pad V2 API

During the initialization, the following elements can be configured as parameters or through the API offered by the SDK.

Table 22 Customizing the PIN Pad Elements

Configuration Element	Description	Note
Scramble / Unscramble Mode	Determines if the position of the keys is randomized or in the standard layout.	
Double Input Field Mode	Determines if the SDK shows two input fields instead of the standard single input field. This mode is specially designed for users changing their PINs through mobile devices, used when validating the new PIN. The input data in the two fields need to be identical. The validation of the two passwords has to be done by SDK user.	
Show Secure Pin Pad as Dialog or Full Screen	Configures whether the keypad is shown as a dialog or as full screen.	Dialog mode with customized top screen is not supported on Android.
Encoding	Sets the secure keypad encoding format.	
Enable top screen	Configures whether the default top screen of secure keypad is displayed.	
Keys	The character set indicating the possible keys that can be displayed. Only alphanumeric characters are currently supported.	
Key Font	Sets the font of keys.	For Android, this attribute applies to keys and subscripts. Keys and subscripts share the same font. For iOS, this attribute applies to keys only, use `setSubscriptFont` to change the font of subscripts.
Key Font Size	Sets the font size of keys.	Applicable to Android only. For iOS, the font size is already included in UIFont.
Key Color	Configures the color of keys. Different colors could be specified for selected and normal state.	
Subscript	Sets the list of subscript to be displayed next to keys on the keypad.	For iOS, the subscript is set together with main keys.
Subscript Font	Sets the font of subscripts.	Applicable to iOS only.
Subscript Font Size	Sets the font size of subscripts.	Applicable to Android only. For iOS, the font size is already included in UIFont.
Subscript Color	Configures the subscript color. Different colors could be specified for selected and normal state.	

Configuration Element	Description	Note
Distance Between Key and Subscript	Set the Distance Between Key and Subscript	Configures the distance between key and subscript. Accepts value [0,4].
Visible Button Presses	Determines if the button is highlighted when tapped. If it is disabled, no highlighting effect appears. This is set to false by default. Applicable to all types of buttons.	If set to false, the button selected state of `setButtonBackgroundColor` will not work.
Button Border Width	Sets the border width of buttons. Accept value is (0, 10].	
Button Background Color	Sets the background color for all types of buttons. Different colors can be specified under normal, selected and disable state.	
Button Background Image	Sets the background image for all types of button.	
Button Background Image Opacity	Sets the image opacity for all types of buttons under normal and disable state (selected state is not supported).	Applicable to Android only. For iOS, the opacity attribute is already included in UIColor.
Button Highlight Color Opacity	Sets the color opacity for all type of button under selected state.	
Button Gradient Color	Set the keypad button character gradient start and end color values in different state.	Added in 4.8.0
Screen Background Color	Sets the background color of the keypad top screen.	
Screen Background Image	Sets the background image of keypad top screen.	
Delete Button Text	Sets the text for Delete button.	
Delete Button Font	Sets the font for Delete button.	
Delete Button Font Size	Sets the font size for Delete button.	Applicable to Android only, for iOS the font size is already included in UIFont.
Delete Button Image	Sets the image for Delete button.	
Delete Button Image Opacity	Sets the image opacity for Delete button under normal and disable state. Note that selected state is not supported.	
Delete Button Text Color	Sets the text color for Delete button under normal, selected and disable state.	
Delete Button Gradient Color	Set the Delete button character gradient start and end color values in different state.	Added in 4.8.0

Configuration Element	Description	Note
OK Button Behavior	Configures the OK button behavior: NONE, CUSTOM, AUTOMATICALLY_ENABLED and ALWAYS_ENABLED. Refer ton behavior table Table 2.	
Ok Button Text	Sets the text for **OK** button.	
Ok Button Font	Sets the font for **OK** button.	
Ok Button Font Size	Sets the font size for **OK** button.	Applicable to Android only, for iOS the font size is already included in UIFont.
Ok Button Image	Sets the image for **OK** button.	
Ok Button Image Opacity	Sets the image opacity for **OK** button under normal and disable state. Note that selected state is not supported.	
Ok Button Text Color	Sets the text color for **OK** button under normal, selected and disable state.	
OK Button Gradient Color	Set the OK button character gradient start and end color values in different state.	Added in 4.8.0
Button Border Color	Sets the border color of all the buttons.	Deprecated in 4.6.0. The API is split into `Sets Keypad Frame Color` and `Sets Keypad Grid Gradient Colors`.
Keypad Frame Color	Sets the frame color of the keypad.	Added in 4.6.0
Keypad Grid Gradient Colors	Sets the grid gradient start and end color for the keypad grids.	Added in 4.6.0
Swap Ok and Delete Button	Swaps the position of **OK** and delete button. By default, the **OK** button is on the left and **Delete** button is on the right.	
Delete Button Always Enable	Sets whether the **Delete** button should always be enabled and even when there is nothing to delete.	Added in 4.6.0
Input Field Font Size	Sets the font size of the aterisk character in the input field.	
Input Field Border color	Sets the color of the input field border Different colors could be specified for focused and unfocused state.	
Input Field Background Color	Sets the color of the input field background. Different colors could be specified for focused and unfocused state.	
First Text Label	Sets the text displayed on the label above the input field(s).	
Second Text Label		
Label Color	Sets the text color of label. This color is shared with the asterisk characters in the input field.	

Configuration Element	Description	Note
Label Font Size	Sets the font size of label.	
Label Alignment	Sets the label text position: left, center or right.	
Logo Image and Position	Sets the image and position of logo (left, center or right).	
Logo Bar Background Color	Sets the color of logo bar.	Applicable to iOS only.
Number of Rows	Sets the grid array size for the keypad, indicating the number of rows and columns to be displayed.	
Number of Columns		
Minimum Input Length	Sets the minimum length of the input text to enter.	
Maximum Input Length	Sets the maximum length of the input text to enter.	
Pin Pad Width	Sets the width of the PIN pad.	Applicable to Android only.
Pin Pad Height	Sets the height of the PIN pad.	Applicable to Android only.
Pin Pad Height Ratio	Sets the ratio of keypad with respect to the entire keypad screen. Accepted value: from 0.25 to 0.5.	This value is ignored when top screen is disabled for Android, the keypad will always occupy the entire fragment.
Pin Pad Dialog Height to Screen Ratio	Sets the height ratio of the keypad with respect to keypad screen. Thidd Applicable to dialog mode.	Applicable to Android only.
Pin Pad Dialog Width to Screen Ratio	Sets the width ratio of the keypad with respect to keypad screen. Applicable to dialog mode.	Applicable to Android only.
Enable/Disable Navigation Bar	Show / Hide navigation bar.	Applicable to iOS only.
Keypad view rectangle in Portrait mode	Keypad view rectangle in Portrait mode.	Applicable to iOS only. Added in 4.8.0.
Keypad view rectangle in Landscape mode	Keypad view rectangle in Landscape mode.	Applicable to iOS only. Added in 4.8.0.
Validate Configuration	Validates whether the keypad configuration is correct.	Applicable to iOS and Android, Android added in 5.0.

The following table explains the **OK** Button behaviour:

Table 23 Ok Button Behavior

OkButtonBehavior	OK Button Visible	Behavior	Constraint
NONE	NO	OK button is internally simulated when the Minimum Text Length = Maximum Text Length and the input reaches that text length.	minimumInputLength must be equal to maximumInpputLength
CUSTOM	YES, Enabled by default	OK button is internally simulated when the Minimum Text Length = Maximum Text Length and the input reaches that text length.	minimumInputLength must be equal to maximumInpputLength
AUTOMATICALLY_ENABLED	YES, Grayed out by default	OK button is enabled only if input length >= Minimum Text Length.	
ALWAYS_ENABLED	YES. Enabled by default	No OK button simulation.	

Callback Methods

The following tables explain the callback methods that enable users to communicate with the internal states of secure key pad.

Table 24 Android Interface SecurePinpadListenerV2

Method	Remark
void onKeyPressedCountChanged(int newCount, int inputField);	For custom top screen, this will be called when the user press the keypad button. The top screen UI is updated according to the newCount and inputField arguments, that is, to update the number of currently typed PIN digits for the selected input filed. For default top screen, you can ignore this callback method.
void onInputFieldSelected(int inputField)	For custom top screen, this will be called when the user interacts with the secure input resulting in a change of the input field focus. For default top screen, you can ignore this callback method.
void onOkButtonPressed()	This is called when the user interacts with the secure input tapping the OK button.
void onDeleteButtonPressed()	This is called when the user interacts with the secure input tapping the Delete button.
void onFinish(PinAuthInput firstPin, PinAuthInput secondPin)	This method is used to get the callback from keypad when the user has finished entering the PIN(s) and tapping the OK button. As the PIN is presented as a secure data object, it is not possible to extract the clear text password from it.
void onError(String errorMessage)	This is called when errors occurred in the secure input.

Table 25 iOS Interface EMSecureInputCustomUiDelegate

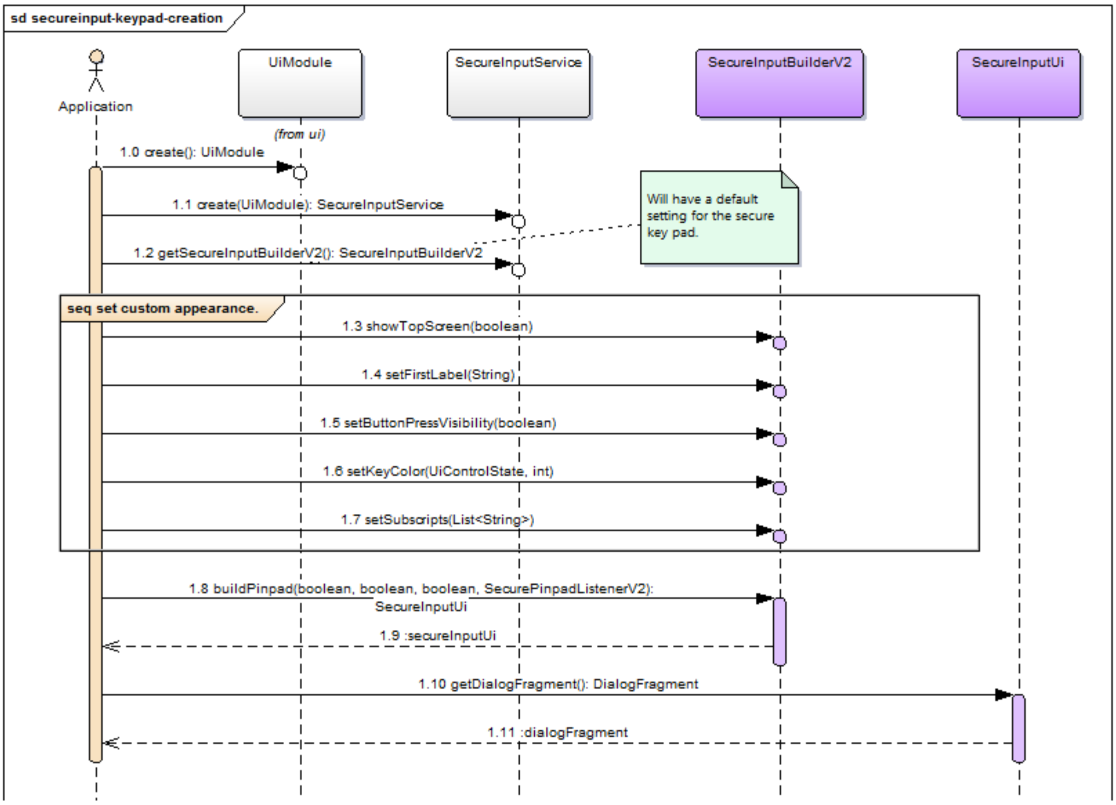
Method	Remark
(void)secureInputUi:(id)caller keyPressedCountChanged:(NSInteger)count forInputField:(NSInteger)inputFieldIndex;	Secure input UI key pressed count changed event.

Method	Remark
(void)secureInputUi:(id)caller selectedInputFieldChanged:(NSInteger)inputFieldIndex;	Secure input UI selected input field changed event.
(void)secureInputUiOkButtonPressed:(id)caller;	Secure input UI on OK button pressed.
(void)secureInputUiDeleteButtonPressed:(id)caller;	Secure input UI on Delete button pressed.

Sequence of PIN Pad Usage

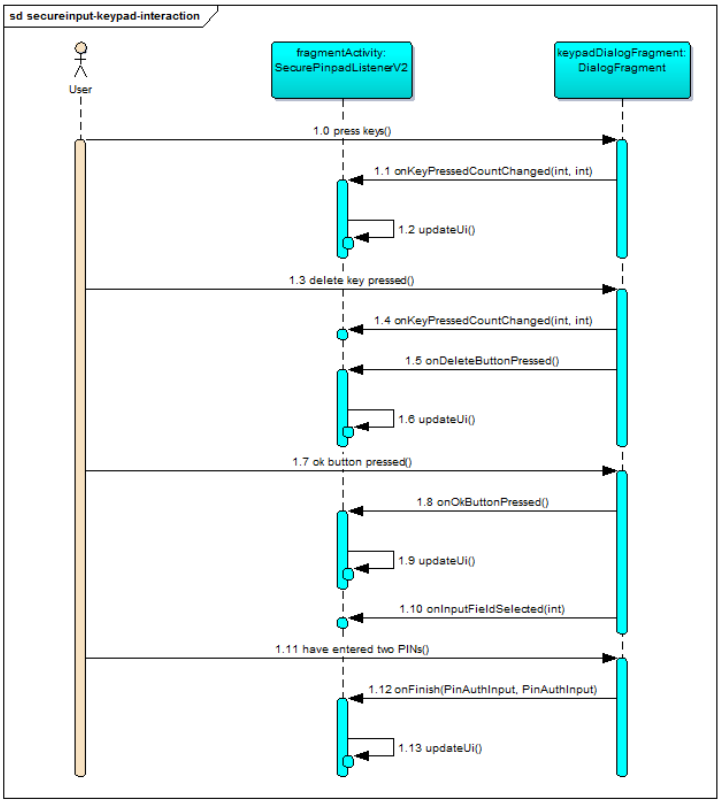
The following diagram shows the creation of secure key pad. The methods and functions for initializing and customizing the PIN pad vary from different platforms.

Figure 24 Sequence of Secure Keypad Creation


sd secureinput-keypad-creation
Application
UiModule
(from ui)
SecureInputService
SecureInputBuilderV2
SecureInputUi
1.0 create(): UiModule
1.1 create(UiModule): SecureInputService
1.2 getSecureInputBuilderV2(): SecureInputBuilderV2
Will have a default setting for the secure key pad.
seq set custom appearance.
1.3 showTopScreen(boolean)
1.4 setFirstLabel(String)
1.5 setButtonPressVisibility(boolean)
1.6 setKeyColor(UiControlState, int)
1.7 setSubscripts(List<String>)
1.8 buildPinpad(boolean, boolean, boolean, SecurePinpadListenerV2): SecureInputUi
1.9 :secureInputUi
1.10 getDialogFragment(): DialogFragment
1.11 :dialogFragment

The following diagram shows the user's interaction with secure key pad. The methods and functions for interacting with the secure keypad vary from different platforms.

Figure 25 Sequence of Secure Keypad Interaction


sd secureinput-keypad-interaction
User
fragmentActivity:
SecurePinpadListenerV2
keypadDialogFragment:
DialogFragment
1.0 press keys()
1.1 onKeyPressedCountChanged(int, int)
1.2 updateUi()
1.3 delete key pressed()
1.4 onKeyPressedCountChanged(int, int)
1.5 onDeleteButtonPressed()
1.6 updateUi()
1.7 ok button pressed()
1.8 onOkButtonPressed()
1.9 updateUi()
1.10 onInputFieldSelected(int)
1.11 have entered two PINs()
1.12 onFinish(PinAuthInput, PinAuthInput)
1.13 updateUi()

Android Interface

As shown in the sequence diagram Figure 1, SecureInputBuilderV2 needs to be initialized first.
The instance of the Secure Input builder can be created from the SecureInputService instance.

```
UiModule uiModule = UiModule.create();
SecureInputService secureInputService = SecureInputService.create(uiModule);
SecureInputBuilderV2 pinpadBuilderV2 = secureInputService.getSecureInputBuilderV2();
```

SecurePinpadListenerV2needs to be implemented in the same class that displays secure PIN pad. A sample is shown as follows:

```
public class MainActivity extends FragmentActivity implements SecurePinpadListenerV2
```

The application can get the callback value from onFinish method after implementing SecurePinpadListenerV2. This overriding method will be called when the user has finished entering the PINs on the keypad. In the single password mode, the secondPin is null. The callback value can be used to change the PIN or to generate OTP.

Note:

The PINs returned by secure PIN pad are presented as PinAuthInputobjects, and the clear text password cannot be extracted from the PinAuthInput object.

```
@Override
public void onFinish(PinAuthInput firstPin, PinAuthInput secondPin) {
    // Call the wipe method to clear the some sensitive data from the pinpad,
    // If you need to display another pinpad inside this callback method,
    // the wipe() must be called before you call the
    // pinpadBuilderV2.build(...) method.
    // The other configuration like color and text label won't be changed here.
pinpadBuilderV2.wipe();

    // dismiss pinpad, you can write your own dismiss method.
    // You has to dismiss it since the dialogFragment cannot be reused.
dismissPinpad();

    // Now you can use the pin object by passing it to the sdk.
    // To use it again (e.g. in use case of changing PIN),
    // you have to wipe and dismiss the current keypad,
    // and then create a new dialogFragment from pinpad object.
}
```

Ensure that the PIN objects and the Secure Pinpad instances are wiped off before dismissing the secure PIN pad.

Note:

Compared to SecureInputBuilder, SecureInputBuilderV2 is designed not to be a singleton instance, every time the method getSecureInputBuilderV2() is called, a new secure PIN pad instance will be created.

There are two ways to create and show the secure PIN pad on Android. The secure PIN pad could be shown as a dialog or full screen activity.

SecurePinPad as Dialog

Default dialog mode

You have to set IS_DIALOG to "true" in the application when creating the SecurePinPadBuilderV2.

The following arguments are mandatory when creating DialogFragment:

```
// Show double pin mode, enter 2 pins in one keypad
boolean IS_DOUBLE_PIN = false;

// Show scramble, randomly arrange the input characters
boolean IS_SCRAMBLE = true;

// Show as dialog (Pop Up)
Boolean IS_DIALOG = true;
```

These are some of the optional configurations that can be set to customize the secure PIN pad. All these settings have to be done before DialogFragment is created.

```
// Display secure keypad default top screen
pinpadBuilderV2.showTopScreen(true);
// all the settings has to be done before creating the dialogFragment
pinpadBuilderV2.setScreenBackGroundColor(getResources().getColor(R.color.pinpad));
pinpadBuilderV2.setDialogHeightToScreenRatio(0.8);
```

```
pinpadBuilderV2.setDialogWidthToScreenRatio(0.8);

// more settings...
```

Creating and displaying the secure PIN pad DialogFragment:

```
// Build the secure PIN pad
SecureInputUi secureInputUi = pinpadBuilderV2.build(IS_SCRAMBLE, IS_DOUBLE_PIN,
IS_DIALOG, MainActivity.this);

// Get an instance of the DialogFragment
DialogFragment dialogFragment = mSecureInputUi.getDialogFragment();

// android.support.v4.app.FragmentManager
FragmentManager mFragmentManager = getSupportFragmentManager();

// Show the pinpad
dialogFragment.show(mFragmentManager, "SECURE PIN");
```

Note:

- The custom top screen is currently not supported in the dialog mode.
- The default top screen must be enabled in order to use dialog mode.

SecurePinPad as Full Screen

Default full screen mode

You have to set IS_DIALOG to "false" in the application when creating the secure PINpad. The following code snippet shows the mandatory settings:

```
// Set false to show single pin mode.
boolean IS_DOUBLE_PIN = false;

// Show scramble, randomly arrange the input characters
boolean IS_SCRAMBLE = true;

// Show as full screen
Boolean IS_DIALOG = false;
```

These are some of the optional configurations that can be set to customize the secure PIN pad. All these settings have to be done before DialogFragment is created.

```
// Display secure keypad default top screen
pinpadBuilderV2.showTopScreen(true);

// all the settings has to be done before creating the dialogFragment

// Main keys must be set before the subscripts
pinpadBuilderV2.setKeys(mMainKeys);
```

```
pinpadBuilderV2.setKeyFontSize(30);
pinpadBuilderV2.setKeyColor(SecureInputBuilderV2.UiControlState.NORMAL,darkBlueColor
);
pinpadBuilderV2.setKeyColor(SecureInputBuilderV2.UiControlState.SELECTED,darkBlueCol
or);
pinpadBuilderV2.setKeyColor(SecureInputBuilderV2.UiControlState.DISABLED,darkBlueCol
or);

// Max and min length settings
pinpadBuilderV2.setMaximumAndMinimumInputLength(pinLength, pinLength);

// more settings...
// ...

// Build the PIN pad
mSecureInputUi = pinpadBuilderV2.buildPinpad(isScrambled, isDoublePassword,
isDialog, ExampleBuilderV2CustomFullScreenActivity2.this);
```

Since secure PIN pad is a subclass of DialogFragment, it has to be contained in a FrameLayout provided by the application.
For example, the XML layout file should be similar as follows:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="#123456"
tools:context="com.gemalto.ezio.mobile.sdk.example.v2.defaultmode.ExampleBuilderV2De
faultFullScreenActivity" >
    <FrameLayout android:id="@+id/frame_layout_v2_default_full_screen"
       android:layout_width="fill_parent" android:layout_height="fill_parent">
    </FrameLayout>
</RelativeLayout>
```

The PIN pad does not scale automatically to fit its container. The application has to provide the width and height when the DialogFragment is first created. For full screen mode, the PIN pad must be set to match the dimension of the FrameLayout that is provided by the application. You have to implement the addOnGlobalLayoutListener interface to get the width and height of the frame layout in runtime as in the following sample code:

```
mFragmentManager = getSupportFragmentManager();

/* Frame Layout to hold the Keypad fragment. */
final FrameLayout frameLayout = (FrameLayout)
findViewById(R.id.frame_layout_v2_custom_full_screen);
mDialogFragment = mSecureInputUi.getDialogFragment();
FragmentTransaction fragmentTransaction = mFragmentManager.beginTransaction();
fragmentTransaction.add(frameLayout.getId(), mDialogFragment);
fragmentTransaction.commit();

// Update keypad height and width.
ViewTreeObserver viewTreeObserver = frameLayout.getViewTreeObserver();
if (viewTreeObserver.isAlive()) {
    viewTreeObserver.addOnGlobalLayoutListener(new
ViewTreeObserver.OnGlobalLayoutListener() {

        @Override
        public void onGlobalLayout() {
            if (android.os.Build.VERSION.SDK_INT < 16) {
                Utils.removeLayoutListenerPre16(frameLayout.getViewTreeObserver(),
this);
            } else {
                Utils.removeLayoutListenerPost16(frameLayout.getViewTreeObserver(),
this);
            }

            int height = frameLayout.getHeight();
            int width = frameLayout.getWidth();
                    pinpadBuilderV2.setKeypadHeight(height);
            pinpadBuilderV2.setKeypadWidth(width);

                    mDialogFragment = mSecureInputUi.getDialogFragment();
            FragmentTransaction fragmentTransaction =
mFragmentManager.beginTransaction();
            fragmentTransaction.detach(mDialogFragment);
            fragmentTransaction.attach(mDialogFragment);
            fragmentTransaction.commit();
        }
    });
}
```

For custom top screen in full screen mode, the method pinpadBuilderV2.showTopScreen(**false**); must be called to hide the default top screen. You have to configure your XML layout according to the UI requirement. See the example project that is included into the delivery package for more details.

iOS Interface

Except for the layout configuration, the secure keypad use case on iOS is similar to that of Android. The keypad is automatically configured with a set of default values created by Ezio Mobile SDK. It can be further customized before it is displayed. An instance of the secure input service can be retrieved from the UI module that has been initialized with a valid activation code.

An instance of the secure input service can be retrieved from the UI module that has been initialized with a valid activation code. An instance of the secure input builder object can be retrieved from the UI secure input service.

```
id<EMSecureInputBuilderV2> secureInputBuilderV2;

EMUIModule *uiModule = [EMUIModule uiModule];
EMSecureInputService *secureInputService = [EMSecureInputService
                                            serviceWithModule:uiModule];

secureInputBuilderV2 = [secureInputService secureInputBuilderV2];
```

Note:

The secure input builder object is to be kept alive till the end of the secure keypad life cycle.

The following code snippet demonstrates a custom configuration that can be performed:

```
[secureInputBuilderV2 setScreenBackgroundColor:EMUIColorFromRGBA(0xFFFFFFFF)];
[secureInputBuilderV2 setInputFieldBorderColor:[UIColor greenColor]
forState:EMSecureInputUiControlFocused];
[secureInputBuilderV2 setInputFieldBorderColor:[UIColor blueColor]
forState:EMSecureInputUiControlUnfocused];
// ...
```

Validate the UI configuration once the settings is done.

```
// Validate UI configuration
[secureInputBuilderV2 validateUiConfiguration];
```

Finally, the keypad is displayed and the callback is handled as follows:

```
// Build secure input UI
id<EMSecureInputUi> secureInputUi;
secureInputUi = [secureInputBuilderV2 buildWithScrambling:YES isDoubleInputField:NO
isDialog:NO onFinishBlock:^(id<EMPinAuthInput> firstPin, id<EMPinAuthInput>
secondPin) {
    // Dismiss the keypad and delete the secure input UI
    [self.navigationController popViewControllerAnimated:YES];

    // Generate OTP and display
    [self _generateOTPAndDisplay:firstPin];

    // Wipe pinpad builder for security purpose.
    [secureInputBuilderV2 wipe];
    secureInputBuilderV2 = nil;
}];

// Push in secure input UI view controller to the current view controller
[self.navigationController pushViewController:[secureInputUi viewController]
animated:YES];
```

To support screen rotation, application have to override the willAnimateRotationToInterfaceOrientation:duration: as shown on the code snippets:

```
-
(void)willAnimateRotationToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceO
rientation duration:(NSTimeInterval)duration
{
    [super willAnimateRotationToInterfaceOrientation:toInterfaceOrientation
duration:duration];
    if ([self.secureInputUi viewController]) {
        [[self.secureInputUi viewController]
willAnimateRotationToInterfaceOrientation:toInterfaceOrientation duration:duration];
    }
```

```
 // Update UI frame
}
```

Note:

The application developer has to override `willAnimateRotationToInterfaceOrientation:duration:` method to support screen rotation. This method is deprecated in iOS 8.0 and will be fixed it in the next secure PIN pad release.

The PIN pad is returned as a `EMSecureInputUi` object that manages the life cycle of the PIN pad itself. The `EMSecureInputUi` contains `viewController` property can be used by the application developer by either pushing into a `UINavigationController` stack or shown as a modal view controller.

Same as the callback in Android, the clear text PINs returned by the secure PIN pad cannot be extracted from the EMPinAuthInput objects.

The secure input builder object is wiped off when it gets deallocated by the autorelease pool. However, in order to enhance security, it is recommended to call wipe prematurely when the no secure PINpad object longer needed.

To customize the upper portion of the UI, call the hide top screen function before validating the UI configuration:

```
[secureInputBuilderV2 showTopScreen:NO];
```

Create the customization UI and handle the custom UI callback respectively. See Callback Methods for all the callback details.

```
// Create custom top view
UIView *customTopView = [[UIView alloc] init];
[self.view addSubview:customTopView];


// Set the custom UI delegate
[secureInputUi setCustomUiDelegate:self];
```

To change the selected input field index:

```
// Change the selected input field
[secureInputUi setSelectedInputFieldIndex:1];
```

Note:

See the example project that is included into the delivery package for more details about customizing the upper portion of the UI.

SecurePinPad as Dialog

To display the secure PINpad in dialog mode, set the isDialog parameter to YES when building the secure input UI. The default secure PINpad configuration is only optimized for full screen mode, it is recommended to customize the look and feels before constructing the secure PINpad.

```
// Customize configurations...

// Build secure input UI
id<EMSecureInputUi> secureInputUi;
secureInputUi = [secureInputBuilderV2 buildWithScrambling:YES isDoubleInputField:NO
isDialog:YES onFinishBlock:^(id<EMPinAuthInput> firstPin, id<EMPinAuthInput>
secondPin) {
    // Pop the secure input UI view controller's view from the parent view

    // Do something...

    // Wipe pinpad builder for security purpose.
    [secureInputBuilderV2 wipe];
    secureInputBuilderV2 = nil;
}];

// Add secure input UI view controller's view ontop of overlay view with dimmed
background
```

Note:

You have to customize the background behind the secure PINpad. For example, adding a dimmed black background.

The `SecureInputUi` object contains `secureKeypadViewRectproperty` that could be used to position the secure PINpad on the screen.

```
// Position the secure PINpad to the center of the screen.
CGRect secureKeypadViewRect = [secureInputUi secureKeypadViewRect];

CGRect frame = CGRectZero;
frame.size.width = secureKeypadViewRect.size.width;
frame.size.height = secureKeypadViewRect.origin.y +
secureKeypadViewRect.size.height;
frame.origin.y += (self.view.frame.size.width - frame.size.width) / 2;
frame.origin.y += (self.view.frame.size.height - frame.size.height) / 2;
[secureInputUi viewController].view.frame = frame;
```

SecurePinPad as Full Screen

To display the secure PINpad in full screen mode, set the isDialogparameter to NO when building the secure input UI.

```
// Build secure input UI
id<EMSecureInputUi> secureInputUi;
secureInputUi = [secureInputBuilderV2 buildWithScrambling:YES isDoubleInputField:NO
isDialog:NO onFinishBlock:^(id<EMPinAuthInput> firstPin, id<EMPinAuthInput>
secondPin) {
    // Dismiss the keypad and delete the secure input UI
    [self.navigationController popViewControllerAnimated:YES];
    // Generate OTP and display
    [self _generateOTPAndDisplay:firstPin];

    // Wipe pinpad builder for security purpose.
    [secureInputBuilderV2 wipe];
    secureInputBuilderV2 = nil;
}];

// Push in secure input UI view controller to the current view controller
[self.navigationController pushViewController:[secureInputUi viewController]
animated:YES];
```

SecurePinPad as Embedded Mode

To display the secure PINpad in embedded mode, set the `keypadViewRectInPortrait` and `setKeypadViewInLandscape` to customize the configurations. This is for application developer who is in charge of developing the top part of the screen.

```
CGRect portraitRect;
CGRect landscapeRect;
[secureInputBuilderV2 setKeypadViewRectInPortrait:portraitRect];
[secureInputBuilderV2 setKeypadViewRectInLandscape:landscapeRect];
// Customize configurations...

// Build secure input UI
id<EMSecureInputUi> secureInputUi;
secureInputUi = [secureInputBuilderV2 buildWithScrambling:YES isDoubleInputField:NO
isDialog:NO onFinishBlock:^(id<EMPinAuthInput> firstPin, id<EMPinAuthInput>
secondPin) {
    // Pop the secure input UI view controller's view from the parent view

    // Do something...

    // Wipe pinpad builder for security purpose.
    [secureInputBuilderV2 wipe];
    secureInputBuilderV2 = nil;
}];

// Create top view
UIView *customTopView = [[UIView alloc] init];
[self.view addSubview:customTopView];

// Set the custom UI delegate (if needed)
[secureInputUi setCustomUiDelegate:self];

// Add secure input UI keypadView to the screen
[self.view addSubView:[secureInputUi keypadView]];
```

Known Limitations

This section describes the known limitations on both Android and iOS platforms that you have to be aware of when integrating the secure PIN pad into application.

Android Known Limitations

This section describes the known limitations that currently exist on Android platform.

- Custom dialog mode is not supported.
- `setKeypadWidth()` and `setKeypadHeight()` must be called before `buildPinpad()` for full screen mode.
- For default top screen mode, there is no API to customize the width and length of input field.

iOS Known Limitations

This section describes the known limitations that currently exist on iOS platform.

- Key and subscript is not horizontally aligned.
- Key is not vertically aligned when the subscript is set.
- No API to set label and input field font.
- No API to set input field text character.
- Overlapping issues when all maximum values for all fields are set in full screen and dialog mode.
- Background image which is set to default aspect ratio does not cover the entire background area.
- Secure keypad top element is not vertically aligned if the keypad controller is presented modally as a view controller for navigation controller.

Error Management and Bug Reporting

This chapter introduces error management and bug reporting in Ezio Mobile SDK. This chapter covers the following use cases:

- Reporting the brief set of actionable result codes to the client application.
- Supplying more technical details about the problems for debugging and technical support.

Error Check

Both the Android and iOS platforms provide an error check so that the developers are able to handle these errors in their applications to display the appropriate message to the end user. While the Android uses the `Exception` class to prompt the application developer to catch them, the iOS uses `NSError` from Foundation frameworks and application developers' to check the `NSError` object accordingly.

Android Interface

In Ezio Mobile SDK, `IdpException` extends the `Exception` class. While using the APIs, the developer will be prompted to catch and handle the exception. `IdpException`, together with its subclasses such as `IdpNetworkException`, `IdpStorageException`, `DeviceFingerprintException` and `PasswordManagerException`, comes with the error domain, error code, error message and its original cause (`exception.getCause()`) if there is any. However, there are certain exceptions which also imply the type of error by its class name.

There are three ways to handle the exceptions. As indicated in the *Javadoc*, a method from the API may throw more than one exception.

The following API is an example that throws a few exceptions:

```
/**
* Gets an existing token.
*
* @param name
*             the name of the token  * @return the token, <code>null</code> if not
found
* @throws com.gemalto.idp.mobile.core.root.RootPolicyException
*              when the
*              {@link TokenRootPolicy}
*              requires this exception to be thrown when the physical device
*              is rooted.
* @throws IdpException
*              this is generic exception, you can directly catch this
*              exception or catch each specific exception below.
* @throws IdpStorageException
*              when database operation failed, or when the name is already
*              used by an existing token.
* @throws DeviceFingerprintException
*              when the token's fingerprint checksum does not match.
* @throws PasswordManagerException
*              when the TOKEN domain is not logged in (using one of the
*              password managers)
*/ <T extends CapToken> T getToken(String name) throws IdpStorageException,
PasswordManagerException,
         DeviceFingerprintException;
```

- To catch each exception separately:

```
try{
     CapToken capToken = capTokenManager.getToken(tokenName);
     ...
} catch (IdpStorageException se){
     // exception handling } catch (DeviceFingerprintException fe){
     // exception handling } catch (PasswordManagerException pe){
     // exception handling }
```

- To catch `IdpException` in general:

```
try{
     CapToken capToken = capTokenManager.getToken(tokenName);
     ... } catch (IdpException e){
     // exception handling
     int domain = e.getDomain();
     int code = e.getCode();
     String errorMessage = e.getMessage();
     ...
}
```

- To catch a mix of exceptions

```
try{
     CapToken capToken = capTokenManager.getToken(tokenName);
     ...
} catch (IdpStorageException se){
     // exception handling
} catch (IdpException e){
     // exception handling
}
```

iOS Interface

Methods that expects an error comes with an NSError parameter.

```
/**
  Creates a mode 2 OTP.
  The typical use case of this mode is for simple authentication.
  @param pin The pin
  @param error object that describes the problem. If you are not concerned with
   possible errors, you can pass in 'nil'.  @return The mode 2 OTP

*/
- (id<EMSecureString>)otpMode2WithAuthInput:(id<EMAuthInput>)authInput
                                       error:(NSError **)error;
```

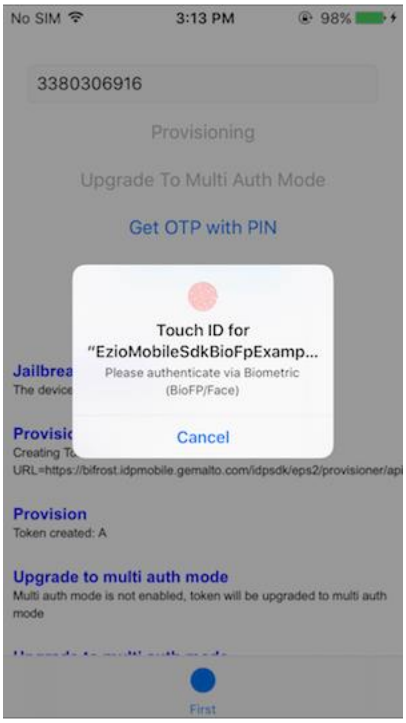
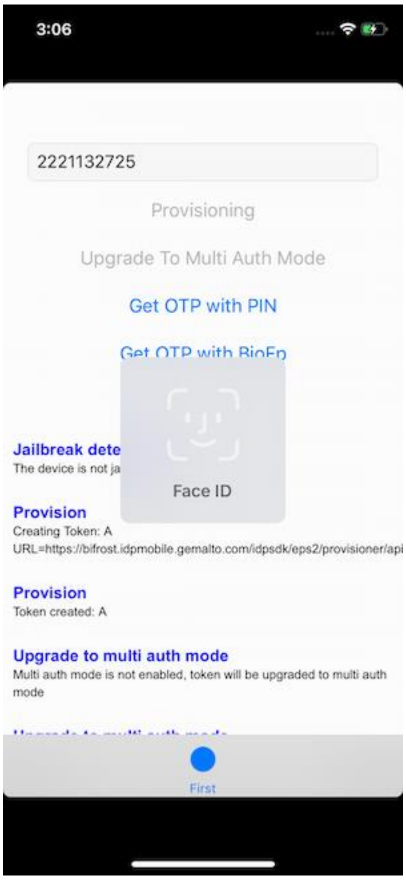
If error is not `nil`, the developer will need to check the details of the error.

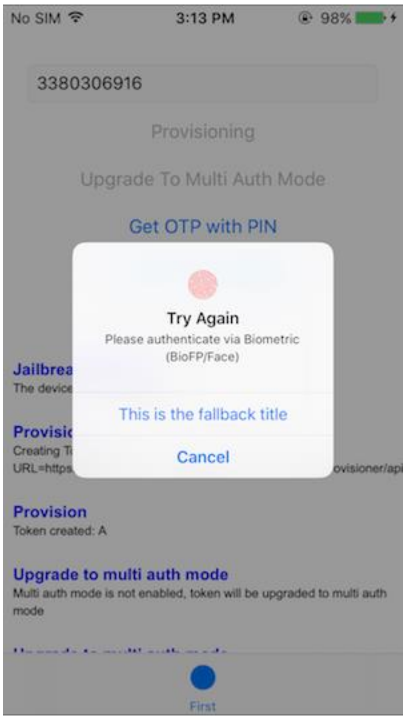
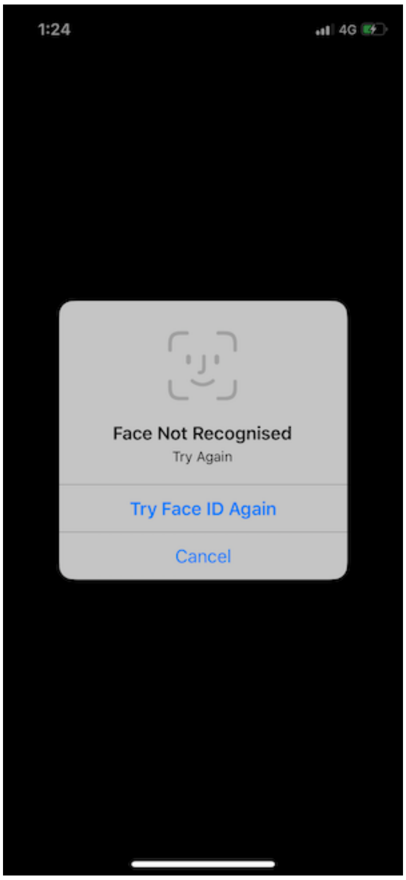
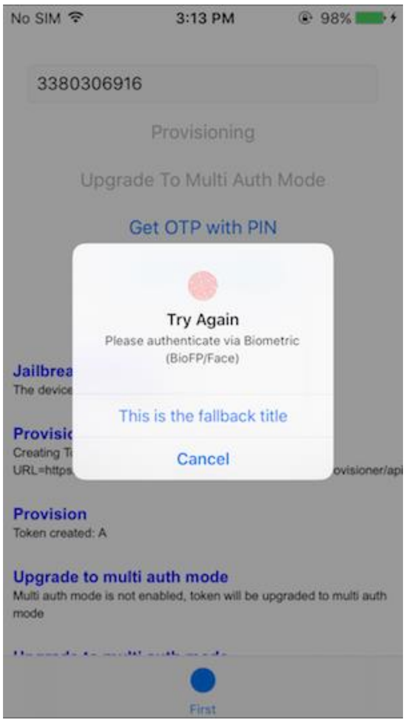
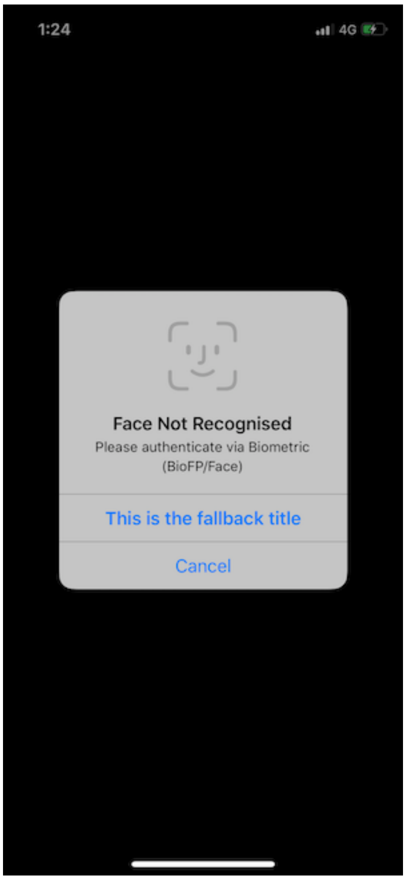
```
NSError *error;
id<EMSecureString> otp = [capDevice otpMode2WithAuthInput:pin error:&error];
if(error){
NSDictionary* info = [error userInfo];
// check info to get the error details
}
```

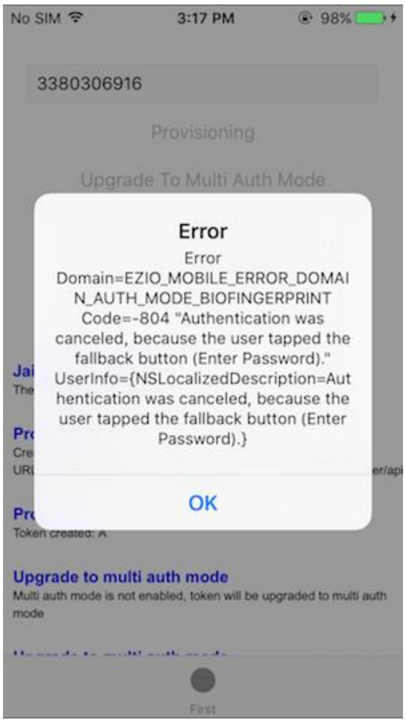
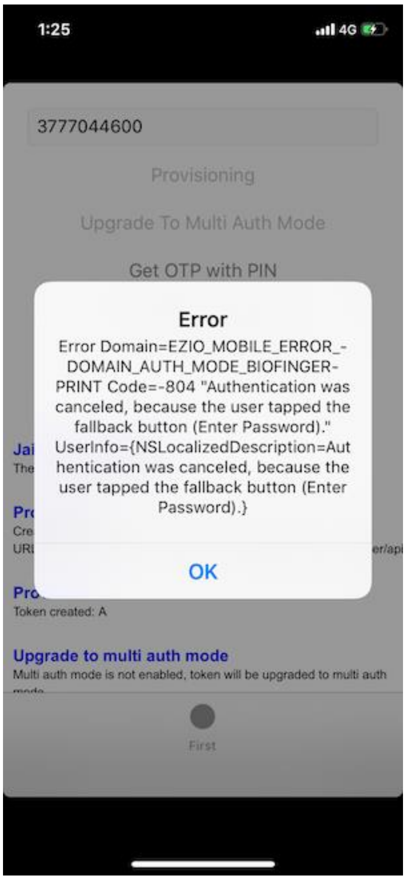
UX Differences Between TouchID and FaceID (iOS)

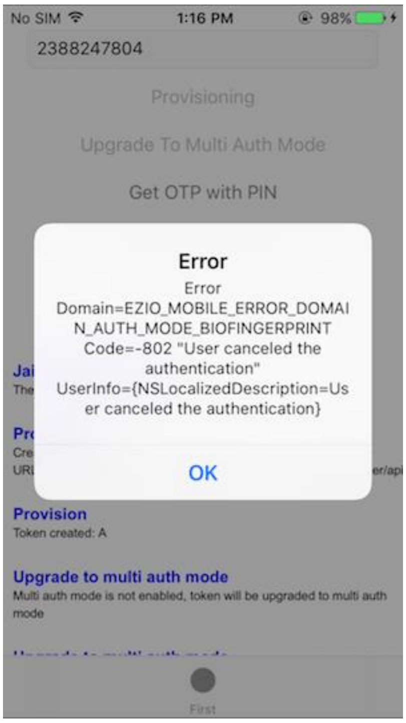
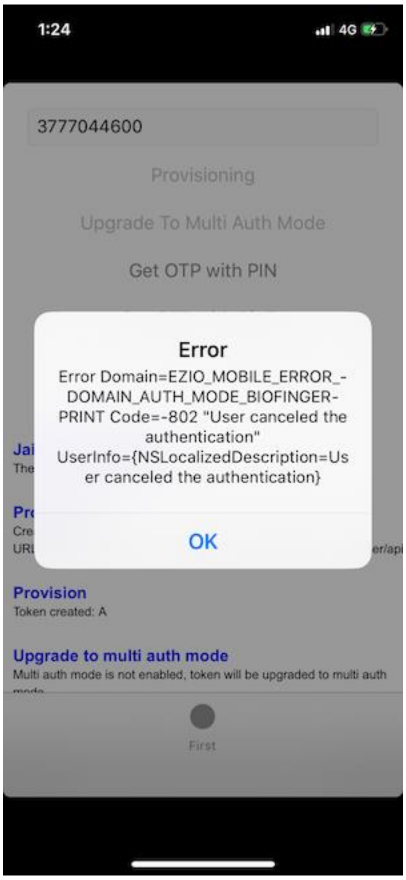
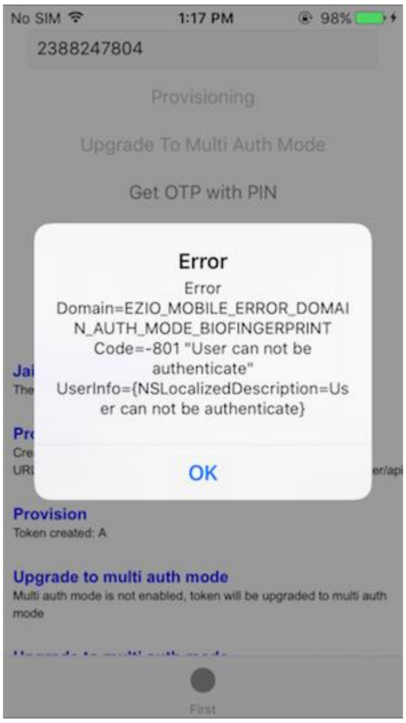
This section describes the UX differences between the error behavior of `EMBioFingerprintAuthService`, `EMSystemBioFingerprintAuthService` and `EMSystemFaceAuthService` when executed on TouchID devices and FaceID devices (iPhone X).

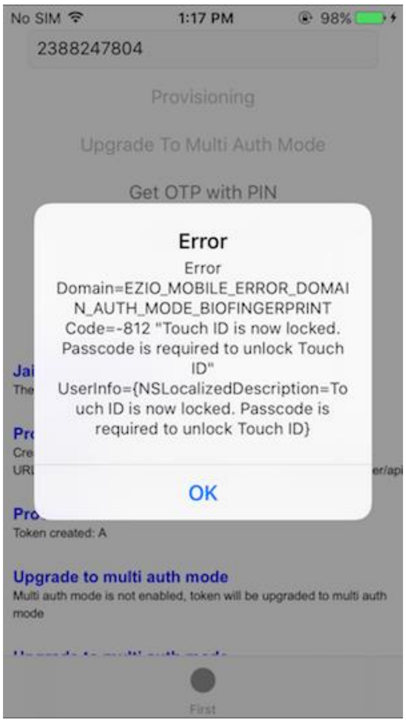
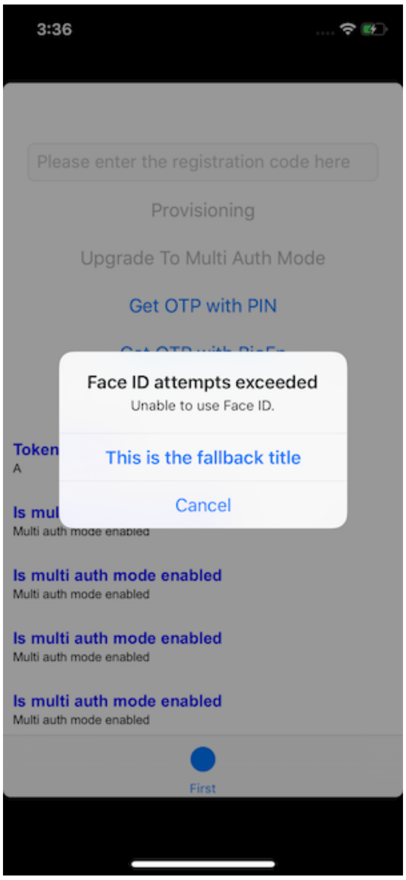
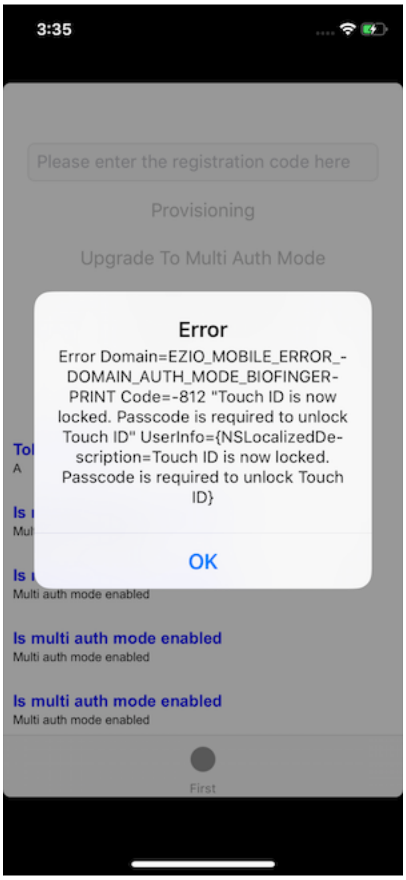
TouchID	FaceID
Authentication Prompt. For BioFP, a prompt is displayed for the user to place their finger onto the sensor while FaceID looks instantaneous, providing no Cancel option.	

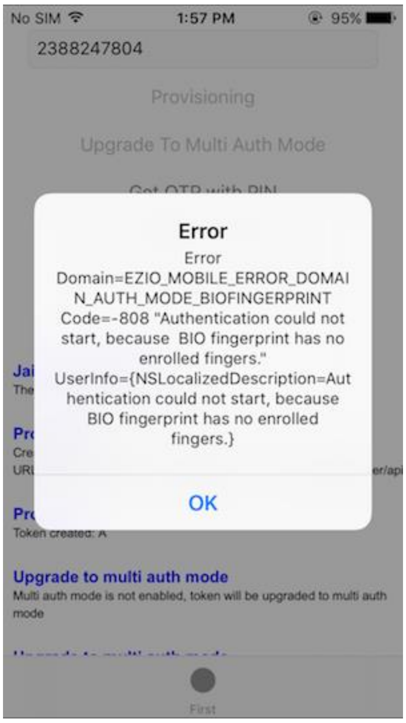
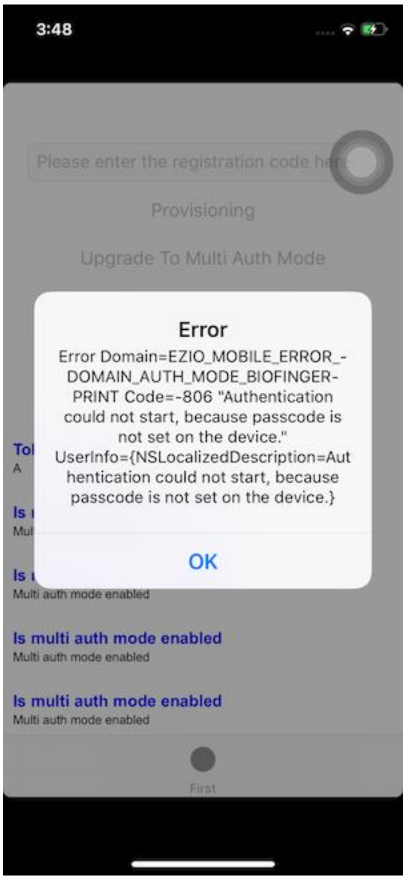
TouchID	FaceID
 No SIM 3:13 PM 98% 3380306916 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Touch ID for “EzioMobileSdkBioFpExamp... Please authenticate via Biometric (BioFP/Face) Cancel Jailbrea The device Provisic Creating To URL=https://bifrost.idpmobile.gemalto.com/idpsdk/eps2/provisioner/api Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First	 3:06 2221132725 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Get OTP with BioFp Face ID Jailbreak dete The device is not ja Provision Creating Token: A URL=https://bifrost.idpmobile.gemalto.com/idpsdk/eps2/provisioner/api Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First
After First Authentication Failure. The fallback button is displayed immediately after the first failure for BioFP. Meanwhile, for FaceID, an option to retry is shown instead.	

TouchID	FaceID
 No SIM 3:13 PM 98% 3380306916 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Try Again Please authenticate via Biometric (BioFP/Face) This is the fallback title Cancel Jailbrea The device Provisic Creating T URL=https ovisioner/api Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First Fallback button appears immediately after first authentication failure.	 1:24 4G Face Not Recognised Try Again Try Face ID Again Cancel
After Subsequent Failures. The fallback button on FaceID is displayed only after 2 failed attempts.	
 No SIM 3:13 PM 98% 3380306916 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Try Again Please authenticate via Biometric (BioFP/Face) This is the fallback title Cancel Jailbrea The device Provisic Creating T URL=https ovisioner/api Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First Fallback button appears after first failed attempt.	 1:24 4G Face Not Recognised Please authenticate via Biometric (BioFP/Face) This is the fallback title Cancel

TouchID	FaceID
	Fallback button appears after 2 failed attempts.
Tapping Fallback Button. Both have the same error code.	
 No SIM 3:17 PM 98% 3380306916 Provisioning Upgrade To Multi Auth Mode Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-804 "Authentication was canceled, because the user tapped the fallback button (Enter Password)." UserInfo={NSLocalizedDescription=Authentication was canceled, because the user tapped the fallback button (Enter Password).} Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode 	 1:25 4G 3777044600 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-804 "Authentication was canceled, because the user tapped the fallback button (Enter Password)." UserInfo={NSLocalizedDescription=Authentication was canceled, because the user tapped the fallback button (Enter Password).} Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth
Cancelled Authentication. Both have the same error code.	

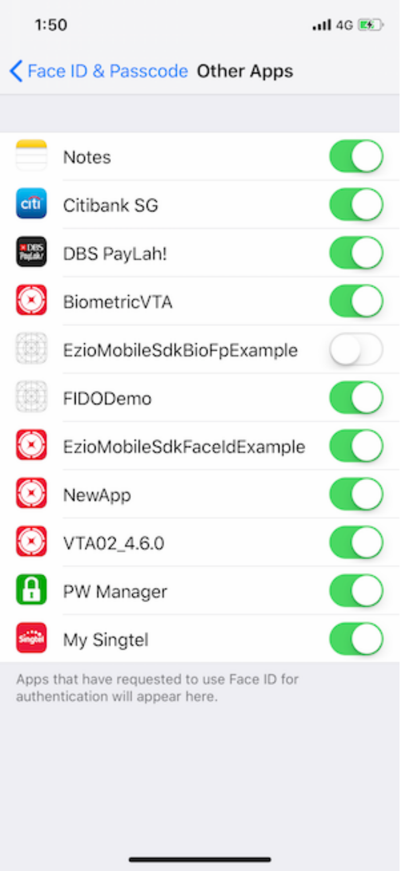
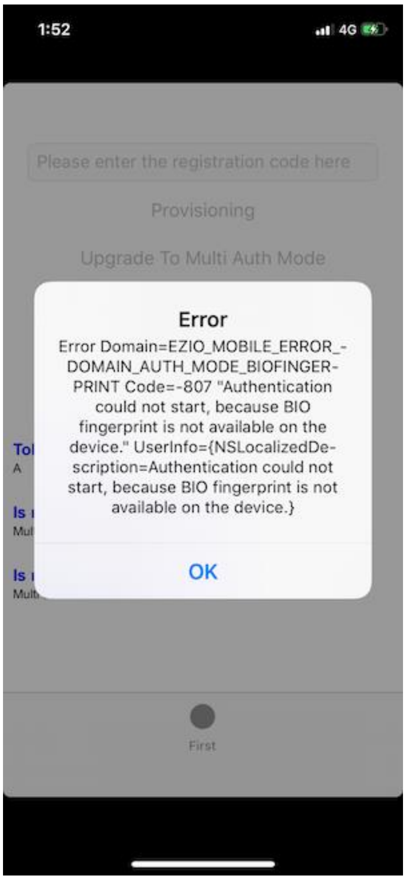
TouchID	**FaceID**
 No SIM 1:16 PM 98% 2388247804 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN **Error** Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-802 "User canceled the authentication" UserInfo={NSLocalizedDescription=User canceled the authentication} OK Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First	 1:24 4G 3777044600 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN **Error** Error Domain=EZIO_MOBILE_ERROR_-DOMAIN_AUTH_MODE_BIOFINGER-PRINT Code=-802 "User canceled the authentication" UserInfo={NSLocalizedDescription=User canceled the authentication} OK Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth First
Authentication Failure. For failed authentication (when the maximum number of attempts has been reached), BioFP returns error code 801:	
 No SIM 1:17 PM 98% 2388247804 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN **Error** Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-801 "User can not be authenticate" UserInfo={NSLocalizedDescription=User can not be authenticate} OK Provision Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First	After the failed attempts, only the Cancel and/or Fallback button will be available which are both "Cancel" use cases. Failed attempts will be logged by the system for the maximum number of attempts made.
After Maximum Allowable Failed Attempts Have Exceeded	

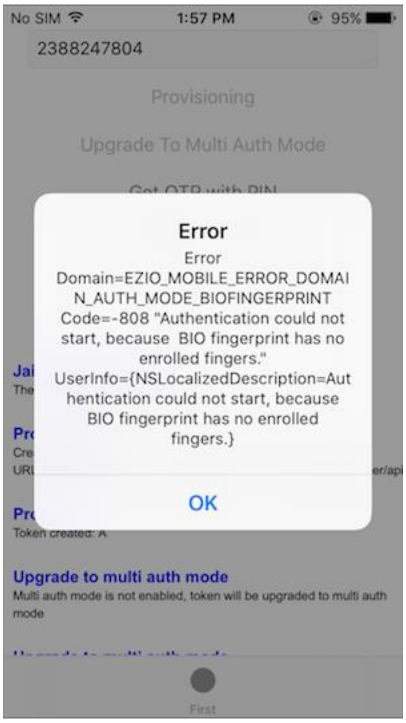
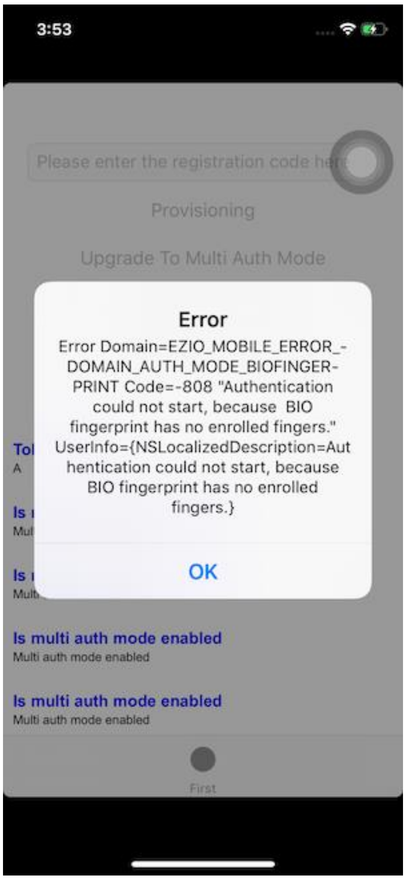
TouchID	FaceID
 No SIM 1:17 PM 98% 2388247804 Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-812 "Touch ID is now locked. Passcode is required to unlock Touch ID" UserInfo={NSLocalizedDescription=Touch ID is now locked. Passcode is required to unlock Touch ID} OK Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First	 3:36 Please enter the registration code here Provisioning Upgrade To Multi Auth Mode Get OTP with PIN Face ID attempts exceeded Unable to use Face ID. This is the fallback title Cancel Token A Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled First  3:35 Please enter the registration code here Provisioning Upgrade To Multi Auth Mode Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-812 "Touch ID is now locked. Passcode is required to unlock Touch ID" UserInfo={NSLocalizedDescription=Touch ID is now locked. Passcode is required to unlock Touch ID} OK Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled First
Device Passcode Is Disabled	

TouchID	FaceID
 Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-808 "Authentication could not start, because BIO fingerprint has no enrolled fingers." UserInfo={NSLocalizedDescription=Authentication could not start, because BIO fingerprint has no enrolled fingers.} OK No SIM 1:57 PM 95% 2388247804 Provisioning Upgrade To Multi Auth Mode Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First	 Error Error Domain=EZIO_MOBILE_ERROR_-DOMAIN_AUTH_MODE_BIOFINGER-PRINT Code=-806 "Authentication could not start, because passcode is not set on the device." UserInfo={NSLocalizedDescription=Authentication could not start, because passcode is not set on the device.} OK 3:48 Please enter the registration code here Provisioning Upgrade To Multi Auth Mode Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled First

FaceID Access by App Has Been Turned Off

(Settings > FaceID & Passcode > Turn OFF)

TouchID	**FaceID**
Not Applicable	 1:50 Face ID & Passcode Other Apps Notes ☑ Citibank SG ☑ DBS PayLah! ☑ BiometricVTA ☑ EzioMobileSdkBioFpExample ☐ FIDODemo ☑ EzioMobileSdkFaceIdExample ☑ NewApp ☑ VTA02_4.6.0 ☑ PW Manager ☑ My Singtel ☑ Apps that have requested to use Face ID for authentication will appear here.  1:52 Please enter the registration code here Provisioning Upgrade To Multi Auth Mode Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-807 "Authentication could not start, because BIO fingerprint is not available on the device." UserInfo={NSLocalizedDescription=Authentication could not start, because BIO fingerprint is not available on the device.} OK First
Fingerprints or Face Image Has Been Removed From Settings	

TouchID	FaceID
 Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-808 "Authentication could not start, because BIO fingerprint has no enrolled fingers." UserInfo={NSLocalizedDescription=Authentication could not start, because BIO fingerprint has no enrolled fingers.} No SIM 1:57 PM 95% 2388247804 Provisioning Upgrade To Multi Auth Mode OK Token created: A Upgrade to multi auth mode Multi auth mode is not enabled, token will be upgraded to multi auth mode First	 Error Error Domain=EZIO_MOBILE_ERROR_DOMAIN_AUTH_MODE_BIOFINGERPRINT Code=-808 "Authentication could not start, because BIO fingerprint has no enrolled fingers." UserInfo={NSLocalizedDescription=Authentication could not start, because BIO fingerprint has no enrolled fingers.} 3:53 Please enter the registration code here Provisioning Upgrade To Multi Auth Mode OK Is multi auth mode enabled Multi auth mode enabled Is multi auth mode enabled Multi auth mode enabled First

Unchecked Errors

Ezio Mobile SDK throws runtime exception when there is a programmer's error (for example, null parameter which breaks the method contract), unrecoverable error (invalid configuration), as well as errors in which the SDK intends to crash the application (for example, Rooted device detected).

Note:

It is not recommended to catch runtime exceptions (`NSException` for iOS).

Bug Reporting

This chapter provides the guidelines for submitting helpful bug reports on Ezio Mobile SDK.
Include the following information in addition to describing the specifics of the problem when submitting the bug report:

- Device model
- OS version
- Ezio Mobile SDK version
- Error logs

The most useful information in a bug report is the error message and stack trace. This information is easy to retrieve when testing an application based on Ezio Mobile SDK but may be difficult to access when it is generated in the field. Therefore, it is recommended that a mechanism is implemented in the application for forwarding errors to a server.
The following options are suggested for forwarding the errors:

- Network report
  In the event of an unexpected error, the application sends the error log and any additional information directly to the company's server (for example, HTTP POST).
- Bug report button

  In the event of an unexpected error, the application requests the user to submit a bug report. The bug report button launches a native email application and initializes the email with the company's support email address and message body.

For the submission of bug reports, it is not recommended to send screen shots or photographs since the image may only contain a subset of the necessary information. Each platform handles errors in a different way. Exceptions and NSErrorprovide information about the error.
In addition, information that is more detailed can be obtained as shown as follows:

- On Android:
  The following sample shows how an Android code snippet logs all the required details of an exception and put its original cause at the beginning of the logs:

```
try {
  // Creating a token is likely to fail for a variety reasons
  // and to do so with nested exceptions.
  capTokenManager.createToken(tokenName,
                  provisioningConfig,
                  fingerprintTokenPolicy);
} catch (Throwable e) {
    // Note: errorLog is your application's error logging mechanism
    Throwable cause = Toolbox.getFirstCause(e.getCause());
    if (cause != null) {
        errorLog.println("Error cause: " + cause.getMessage());
        cause.printStackTrace(errorLog);
    }
    errorLog.println("Error: " + e.getMessage());
    e.printStackTrace(errorLog);

    // Display error
}
 // Use a helper to get source of an exception
class Toolbox {
    static public Throwable getFirstCause(Throwable cause)
    {
        Throwable first = null;

        while (cause != null && cause != first) {
          first = cause;
          cause = cause.getCause();
        }
        return first;
    }
}
```

- On iOS:
  The following sample shows how an iOS code snippet logs all the necessary details of an error and put its original underlying error at the beginning of the logs:

```
// EMTokenManagerDelegate
-(void)tokenManager:(id<EMTokenManager>)tokenManager didFailWithError:(NSError
*)error
{
    // Note: NSLog is your application’s error logging mechanism
    NSError *cause = [Toolbox getFirstUnderlyingErrorWithError:error];
    if (cause != nil)
    {
        NSLog(@"Error cause: %@ (%d): %@", [cause domain], [cause code],
        [cause localizedDescription]);
        NSLog(@"userInfo = %@", [cause userInfo]);
    }
    NSLog(@"Error: %@ (%d): %@", [error domain], [error code], [error
localizedDescription]);
    NSLog(@"userInfo = %@", [error userInfo]);

    // Display error
    ...
}


// Use a helper to get source of an error
@implementation Toolbox : NSObject


+(NSError*)getFirstUnderlyingErrorWithError:(NSError *)error
{
    NSError* first = nil;
    while(nil != error) {
        first = error;
        error = [[error userInfo] objectForKey:NSUnderlyingErrorKey];
    }
    return first;
}
```

Backup & Restore

Backup

Apps that target Android 6.0 (API level 23) or later automatically participate in Auto Backup. Backups occur automatically when the user has enabled backup on the device in Settings > Backup & Reset.

Enabling and Disabling Backup

In your app manifest file, set the boolean value for the attribute `android:allowBackup` to enable or disable backup. The syntax is as follows:

Table 26 Enable or Disable Backup

```
<manifest ... >
    ...
    <application android:allowBackup="true" ... >
        ...
    </application>
</manifest>
```

Where the default value is set to `true`, but you can set this value to `false` to disable the backup option.

Files that are Backed Up

- Shared preferences files.
- Files that are saved to your app's internal storage, and accessed by `getFilesDir()` or `getDir(String, int)`.
- Files in the directory returned by `getDatabasePath(String)`, which also includes files created with the `SQLiteOpenHelper` class.
- Files on external storage in the directory returned by `getExternalFilesDir(String)`.

Note:

Auto Backup excludes files in directories returned by `getCacheDir()`, `getCodeCacheDir()`, or `getNoBackupFilesDir()`. The files saved in these locations are either only needed temporarily, or are intentionally excluded from backup operations.

Including and Excluding Files for Backup

By default, the system backs up almost all app data. Application developers have a control to set the rules on what to back up and what to exclude from the backup.

Perform the following steps to include and exclude the files to/from the backup,

1. In `AndroidManifest.xml`, add the `android:fullBackupContent` attribute to the `<application>` element. This attribute points to an XML file that contains the backup rules.

   Table 27 android:fullBackupContent

   ```
   <application ...
       android:fullBackupContent="@xml/my_backup_rules">
   </application>
   ```

2. Create an XML file called `my_backup_rules.xml` in the `res/xml/` directory. Inside the file, add rules with the`<include>` and `<exclude>` elements. The XML syntax for the configuration file is shown below

   Table 28 Rules

   ```
   <full-backup-content>
       <include domain=["file" | "database" | "sharedpref" | "external" | "root"]
   path="string" requiredFlags=["clientSideEncryption" | "deviceToDeviceTransfer"]
   />
       <exclude domain=["file" | "database" | "sharedpref" | "external" | "root"]
   path="string" />
   </full-backup-content>
   ```

   Each element must include the `domain` and `path` attributes, where:

   - `domain` refers to only these values ("file" | "database" | "sharedpref" | "external" | "root")
   - `path` specifies the file or folder.

   Note:

   - This attribute does not support wildcard or regex syntax.
   - You can use . to reference the current directory, however, you cannot reference the parent directory `..` for security reasons.
   - If you specify a directory, then the rule applies to all files in the directory and recursive sub-directories.

3. The following sample backs up all the shared preferences except `my_preferences.xml.`

   Table 29 Sample Rules

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <full-backup-content>
       <include domain="sharedpref" path="."/>
       <exclude domain="sharedpref" path="my_preferences.xml"/>
   </full-backup-content>
   ```

Restore

When your application data is backed up (system schedule the backup automatically) and the user enables **Automatic restore** in **Settings > Backup & Reset,** then the data can be restored. A restore happens when the user uninstalls the application and reinstalls the same application again, and the application data will be restored automatically in the specific data folder.

Recommendations

Use case #1: No Backup

If your application does not back up the application data, you can set the `android:allowBackup="false"` as shown in Enabling and Disabling Backup section.

Use case #2: Backup

1. Back up only your application generated files, not the EZIO SDK generated files.
2. Perform the steps as shown in Including and Excluding Files for Backup section.
3. If you specify an `<include>` element, the system no longer includes any files by default and backs up only the files specified. To include multiple files, use multiple `<include>` elements.

Note:

- Do not use `<include domain="<any domain values>" path="."/>`, this will include all the files in the directory. Specify the exact name for the path.
- This recommendation is also applicable to Android version < 23.

Migration for Android Q Upgrade

Google has deployed the Android Q Beta, and there are specific changes on data privacy enforcement that forbid collection of device information for third party applications.

With the new Android Q, some APIs' usage is restricted in getting the IMEI and device fingerprint source service details. As Ezio Mobile SDK is dependent on device information, it is important to migrate to the new changes which are independent of device information.

For more information on Android Q privacy changes, refer to *https://developer.android.com/preview/privacy/data-identifiers#device-ids*

Warning:

To ensure that the active end users are not affected by the Android Q privacy changes, the following must be done:

- The app needs to be updated with Ezio Mobile SDK V4.9.2 to migrate the data related to Ezio features.
- Call the migration APIs in the app start up to make sure that all data is migrated.
- The EZIO Mobile modules: OTP, OOB, and Secure Storage have to be migrated individually.
- The application has to call these migration APIs before the device is upgraded to Android Q, since the migration requires the device information.

If the migration is not done before the device is upgraded to Android Q, the Ezio Mobile SDK features will not work.

Migration Steps

There are two new APIs for each module.

1. The first API `isMigrationNeededForAndroidQ` is used to check if the migration is needed. If this method returns a `true` value, the end user has to give the permission for READ_PHONE_STATE.
2. If the `isMigrationNeeded` API returns a `false` value, the READ_PHONE_STATE permission will not be required. Hence, permission is not required for the new user.Once the permissions are given, the `migrateForAndroidQ` API need to be called to trigger the data migration for Android Q.

Warning:

- A password manager login is required before calling `isMigrationNeededForAndroidQ` and `migrateForAndroidQ` API.
- The migration API must be called before performing any operation. For example, before retrieving a token on OTP Module or fetching an OOB message.

- The same custom fingerprint data is required for OTP migration.
- Ensure that the following data is available for migration:
  - On OTP Module, the token name must exist. That is, the token name in `TokenManager.getTokenNames` is used to get the available token names
  - On OOB, the same OOB application identifier must be used. This value is the same value used during the creation of OobManager: `OobModule.createOobManager`

Code Snippets

The following code snippets are used for Android Q migration.

- OTP Module

```
passwordManager.login(password);
OtpModule otpModule = OtpModule.create();
// Validate tokenName is in the set from TokenManager.getTokenNames
// Perform migration for All token
if(otpModule.isMigrationNeededForAndroidQ(tokenName)) {
    //Ask for READ_PHONE_STATE permission

    //Once the user permissions are granted
    // customData can be null if this token is created without customData
    otpModule.migrateForAndroidQ(tokenName, customData);
}
```

- OOB Module

```
// deviceFingerprintSource -- the value used on previous version
OobConfiguration oobConf = new OobConfiguration.Builder()
                        .setDeviceFingerprintSource(deviceFingerprintSource)
                        .build();
IdpCore.configure(deviceSourceBinding, oobConf);
OobModule oobModule = OobModule.create();
//applicationId -- the value used previously on OobModule.createOobManager
```

```
if(oobModule.isMigrationNeededForAndroidQ(applicationId)) {
    //Ask for READ_PHONE_STATE permission


    //Once the user permissions are granted
    oobModule.migrateForAndroidQ(applicationId);
}
```

- Secure Storage Module

```
SecureStorageModule secureStorageModule = SecureStorageModule.create();

//storageIdentifier -- the value used previously on 
SecureStorageManager.getPropertyStorage
//deviceFingerprintSource -- the value used on previous version
if (secureStorageModule.isMigrationNeededForAndroidQ(storageIdentifier, 
deviceFingerprintSource)) {
    //Ask for READ_PHONE_STATE permission


    //Once the user permissions are granted
    secureStorageModule.migrateForAndroidQ(storageIdentifier, 
deviceFingerprintSource);
}
```

Validation

To test the migration code, use the following steps:

1. Create an application with device binding enabled using Ezio Mobile SDK V4.9.1 or earlier.
2. Install the application on Android Pie or on earlier version.
3. Run the application with Token Creation, OOB Registration, and/or Secure Storage Write data.
4. Update the application with migration code using Ezio Mobile SDK V4.9.2 or later.
5. Run the application to execute migration code.
6. Re-run the application and a `false` value will be returned from the API call, `isMigrationNeededForAndroidQ`.
7. Upgrade the operating system to Android Q and validate existing data can still be used.

Frequently Asked Questions

This chapter contains the commonly asked questions about Ezio Mobile SDK.

- Can you explain possible values and configuration of IPB?
  IPB comes from the CAP specification from MasterCard in the token compression step. The IPB applies to the assembled token data which consists of:

  ```
  PSN        ||   CID     ||    ATC     ||    AC      ||   IAD
  <-1 byte->  <-1 byte->  <-2 bytes->  <-8 bytes->  <-8 bytes->
  ```

  The default IPB value of the mobile SDK is:
  `00 00 00 FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
  This gives an effective length of 24 bits, resulting in a CAP token of up to 8 digits (refer to *Chip Authentication Program - Functional Architecture 2007* (MasterCard)). The following table lists the maximum token length, depending on the effective IPB length (the number of 1-bits in the IPB). An actual token may be shorter, depending on its IPB value.

  Table 30 Length of IPB and its Maximum Token Length

Effective IPB Length	Maximum Token Length
17 – 19	6
20 – 23	7
24 – 26	8
27 – 29	9
30 – 33	10
34 – 36	11
37 – 39	12
40 – 43	13
44 – 46	14
47 – 49	15
50 – 53	16

  Other CAP configurations may have an impact on the effective IPB length, for example, IAF bit 7 (2nd bit from the left) will decide if the 1st byte of IPB (the PSN) is used (b7 = 1) or omitted (b7 = 0).

- Does the SDK support client certificates for mutual authentication in the provisioning sequence?
  No, the SDK does not support this feature for provisioning. Therefore, the server's TLS software must be configured to accept clients without certificates.

- Why am I getting an exception when I generate OTP after updating my iOS app from App Store to my current version?
  Normally, updating the app on App Store works without a problem. However, the testing of update works

outside the App Store environment is a bit tricky and usually fails when generating OTP from a new version although it has been provisioned previously in the older version. This is because a fingerprint data key component known as identifierForVendor changes when updating an app installed from App Store to a newer version via other means such as Xcode or iTunes. This does not indicate a real failure of the client but one that requires a special procedure to test this use case, refer to the following steps to counter this update failure:
Procedure:

a. Build/resign your previous app version to update (App A) with an ad hoc provisioning profile.
b. Build your updated version (App B) with the same provisioning profile.
c. Use iTunes to import App A to iTunes Library by double clicking or a simple drag. Install your app on a device by selecting the connected device and navigating to **Apps** tab. Click **Install**, and sync.
d. Perform the provisioning, generate OTP and execute your tests.
e. Build and install Previous App Version.
f. Next import App B to iTunes library. This will replace the previous library.
g. Click **Update** and sync.
h. Perform the OTP generation.

- How is the EPS public key transferred to the mobile application?
  As it is the role of the mobile application to set the EPS public key. It means that the mobile application can decide to hard-code it if they want.
- Is there any sample challenge identifying different templates for Dynamic Signature?
  The template provides a list of primitives to be used in the computation of Dynamic Signature which can be retrieved from the challenge in the unconnected mode by the SDK. Certain combinations of the templates and sample challenges are listed as follows:

Table 31 Templates and Sample Challenges

Template ID	Challenge
Template 0	00007
Template 1	00024
Template 2	00048
Template 3	00069
Template 4	00082
Template 5	00105
Template 6	00122
Template 7	00146
Template 8	00167
Template 9	00180
Template 10	00200
Template 11	00228
Template 12	00244
Template 13	00263
Template 14	00285
Template 15	00302
Template 16	00325
Template 17	00341
Template 18	00360
Template 19	00387
Template 20	00409
Template 21	00421
Template 22	00445
Template 23	00466

- Why is the SDK not working after the restoration?
  There are two possible reasons that explain why the OTP verification no longer works after an application restoration. First, the application used fingerprint mechanism to seal the credentials with the device (see Device Fingerprint) restoring application data does not work on a different device. It is suggested to perform a new provisioning in this case. Moreover, if the application data are restored on the same device assuming that the fingerprint data has not changed, there is still a chance that the OTP cannot be verified if the token used is an event-based type. The reason for this is that a backup/restore may de-synchronize the counter. In this case, there is no way to re-synchronize it as accepting the previous OTPs is considered as a major security issue.

- For iOS, why am I getting "duplicate symbols" linker error when I use Reachability or `KeychainItemWrapper` class distributed by Apple?
  One of the components of Ezio Mobile SDK uses the popular utility classes `Reachability` and `KeychainItemWrapper` which result in duplicate symbols. As a temporary workaround, you can perform any of the following methods:

- Rename your `Reachability/KeychainItemWrapper` classes. Apple grants you to use, modify or reproduce in source or binary form with or without modification. For more information on the license, refer to:
  - https://developer.apple.com/library/ios/samplecode/Reachability
  - https://developer.apple.com/library/ios/samplecode/GenericKeychain
- Remove the implementation files (`.m`) of these classes.

Terminology

This section contains abbreviations and terms found in this document or related documents.

Abbreviations

EPS	Enrollment Provisioning Server
ESN	Electronic Serial Number
HOTP	HMAC based One Time Password
HSM	Hardware Security Module
IMEI	International Mobile Equipment Identifier
IMSI	International Mobile Subscriber Identifier
OATH	Open Authentication
OOB	Out Of Band
OCRA	OATH Challenge-Response Algorithm
OTP	One Time Password
PM	Password Manager
PIN	Personal Identification Number
PTC	PIN Try Counter
RC	Registration Code
SM	Security Module (see also HSM and SSM)
SMS	Short Message Service
SSM	Software Security Module
TLS	Transport Layer Security
TOTP	Time-based One Time Password
URL	Universal Resource Locator
VIC	Verify Issuer Code
VICATC	VIC Application Transaction Counter
VICTC	VIC Try Counter
DSKPP	Dynamic Symmetric Key Provisioning Protocol

Glossary of Terms

Provisioning Access Domain is a SIM Toolkit/UICC Toolkit parameter that specifies the identities or access rights (CHV & ADM) granted to an application to access GSM/UICC files and perform actions on these files. For example, if the Access Domain value is "FF" (No Access to the File System), attempts to access a file cause an exception.

Token The Ezio Mobile SDK's representation of a user's credentials.

User The mobile device user.