iDENprotect for iOS Developer Guide

Apply Mobile

Version 1.0 beta, May 23th, 2017

Table of Contents

This guide describes how to develop iOS applications that authenticate your users using the iDENprotect architecture.

iDENprotect applications for iOS are developed using the iDENprotect SDK. The iDENprotect SDK components use the **Objective-C programming language** and **CocoaPods dependency manager** in **macOS environment**, and familiarity with those tools is helpful when developing iDENprotect applications.

General information on iOS development can be found directly from Apple at http://developer.apple.com.

iDENprotect Architecture

iDENprotect provides strong software or hardware based authentication for the iOS platform. It combines ease of use and suitability for high-security environments. iDENprotect is designed for mobile use, where iDENprotect can be used to provide secure authentication and identity services for both managed and unmanaged mobile devices.

iDENprotect helps secure communications and transactions within the financial sector, healthcare, government, defence and other security conscious sectors and organizations. iDENprotect integrates with existing services such as PKI, RADIUS, Active Directory, as well as web authentication technologies such as OATH, SAML and OpenID.

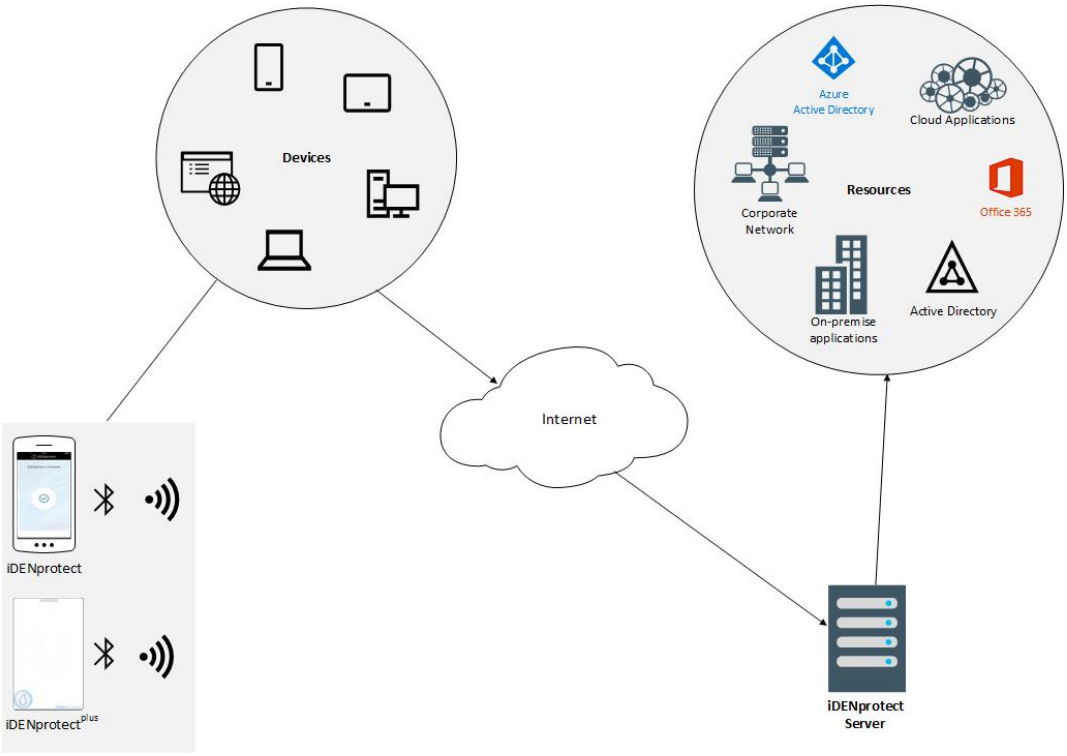

Figure 1. iDENprotect General Usage

Tokens

In iDENprotect architecture, the device-specific security credentials (Keys) are stored in a Token, which can be a programmatic software Token located on the user's mobile device, or a separate hardware module for extra security. The Tokens are authenticated by biometric identification on the iOS device, or the user's PIN code, which is set when enrolling the iDENprotect Token into use on a backend iDENprotect server.

Possession of either the Soft Token or Hard Token is effectively a 2nd factor in multi-factor authentication.

iDENprotect Tokens authenticate with the same backend iDENprotect server system they are enrolled on. iDENprotect server listens to incoming authentication requests and processes them. In order for authentication requests to be processed by the iDENprotect server, iDENprotect Tokens need to first be enrolled on the iDENprotect server.

iDENprotect Soft Token

An iDENprotect Soft Token is an authentication application that runs all sensitive operations in the iOS device's Secure Enclave. The Secure Enclave is a separate processor in iOS devices used to perform cryptographic operations securely in a tamper-resistant environment.

The iDENprotect Soft Token stores all Keys securely on the mobile device. The Keys are only used for authentication requests such as signing. The Keys can't be extracted or copied from the device.

iDENprotect^{plus} Hard Token

An iDENprotect^{plus} Hard Token is a separate hardware authentication module which provides an additional layer of security. The Hard Token is a SmartCard-style module that is combined with other biometric authentication on the user's iOS device, and doesn't require other manual input.

iDENprotect^{plus} connects to the iOS device by forming a temporary pairing over a proprietary encrypted Bluetooth Smart channel. The iOS device then relays iDENprotect traffic between iDENprotect^{plus} Token and the iDENprotect server.

When performing the authentication, iDENprotect^{plus} treats the iOS device as a proxy, and the client has no access to the data within the communication. This provides true end to end secrecy as all communications between the provisioning server and the iDENprotect device are encrypted without the user of the client having access to the data.

For more information on Bluetooth Smart security, see NIST Guide to Bluetooth Security.

Typical Operation Overview

iDENprotect devices are typically used to authenticate the owner of a iOS device running the iDENprotect application to access a service or a resource. The mobile device acts as a relay within a typical client-server model and responds to requests within the secure realm of the iOS device or iDENprotect^{plus} Hard Token before communicating the results back to the iDENprotect application.

Within this flow, the mobile device always requires the user's PIN or biometric identification to use the Token, and always "logs out" to securely close the session. For iDENprotect^{plus}, this also preserves the device's power resources.

The Token must be taken into use on a iDENprotect server in a process called Provisioning, where the device owner **Registers** the Token on the iDENprotect server, and **Enrolls** it after the device owner's identity has been confirmed. Tokens not Enrolled on a particular iDENprotect server are not able to perform authentication activities.

Enrolled Tokens can be used to authenticate the device owner in iDENprotect applications. iDENprotect applications are commonly integrated with Mobile Application Management (MAM) software such as BlackBerry Dynamics to take advantage of an existing management framework within an organization.

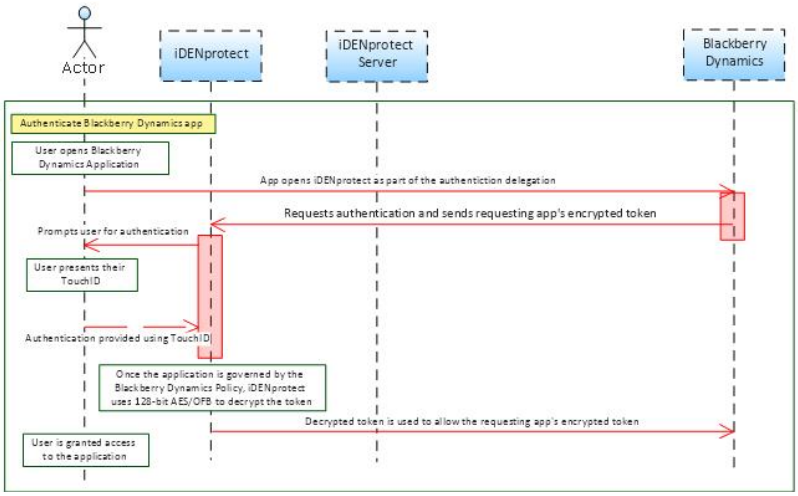

Figure 2. iDENprotect Authentication with BlackBerry Dynamics

Communications Security

iDENprotect devices are authenticated in the iDENprotect server by the user's biometric authentication or a PIN code.

Communications between iDENprotect devices and the iDENprotect server are secured by an Elliptic Curve Diffie-Hellman (ECDH) handshake based on a Patent Pending, FIPS-compatible random number generator that uses multiple environmental inputs to generate the random seed. After the handshake, data transaction proceeds over AES128-encrypted channel. This means that there is no cleartext phase of communication present during any transaction with the iDENprotect device.

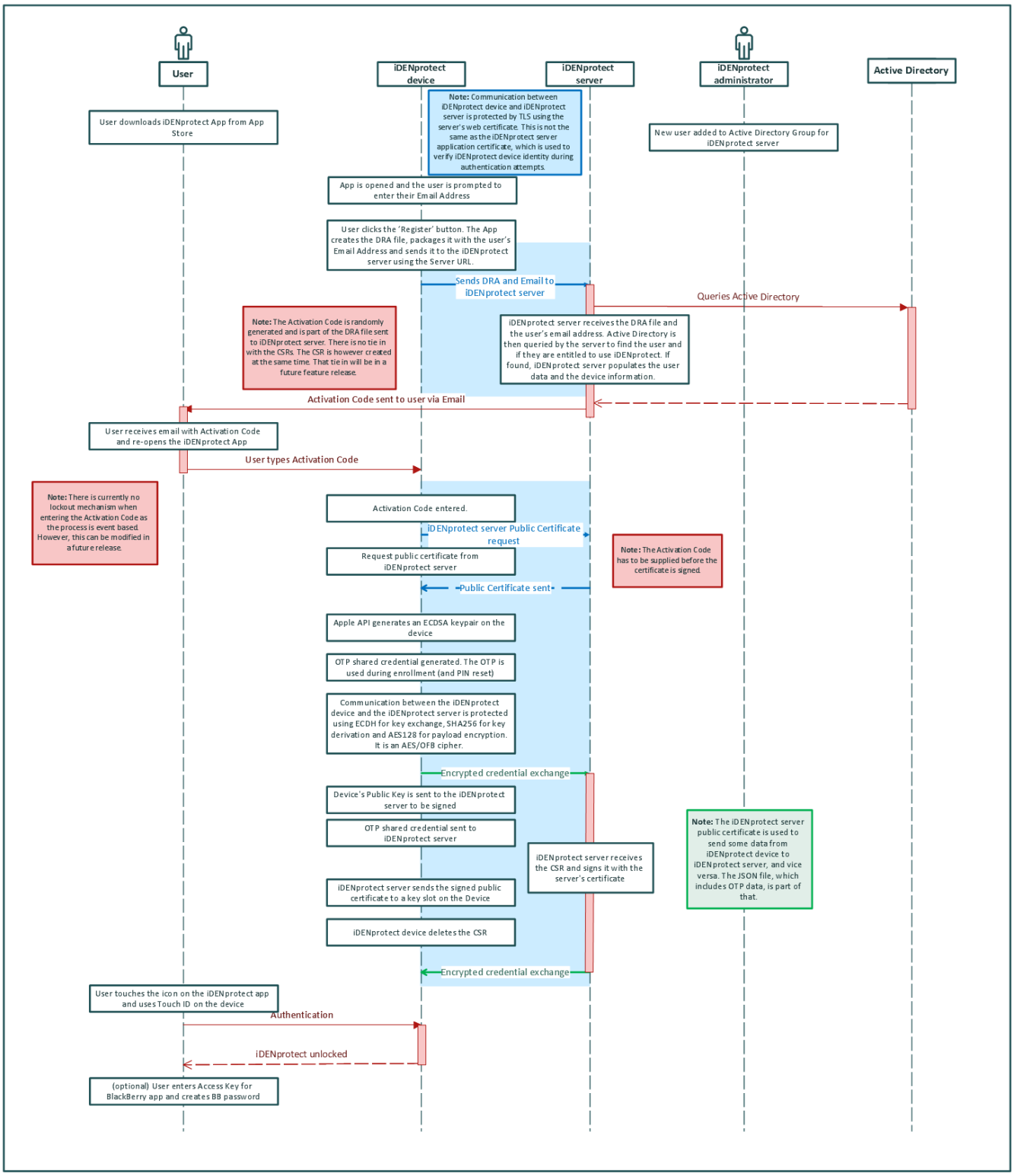

Figure 3. iDENprotect Enrollment Sequence

Both Token types use a JSON-based API to communicate with iDENprotect applications. This API and its related security processes are designed so that secret keys are never transferred. Instead, the iDENprotect Token functions as a Hardware Security Module (HSM) that uses securely stored internal private keys for signing and encryption operations. When creating Time-based One-Time Passwords (TOTPs), where shared secrets are required, iDENprotect architecture uses a secure ECDH/AES encryption scheme between the Token and the iDENprotect server to ensure secrecy.

As well as setting up iDENprotect devices, the iDENprotect server also ensures that the device has the correct configuration for that particular organization. Additionally, it syncs the time and performs the

backend part of the process flow in device PIN resets.

Furthermore, user and device credentials can be revoked from the server-side iDENprotect server by an iDENprotect Administrator, ensuring the user life cycle can be managed from one location.

Authentication Concepts

When building iDENprotect applications, the libraries within the iDENprotect SDK act as the authentication agent and interface with the iDENprotect device (Soft Token or Hard Token).

When an authentication request is made, the user is prompted to confirm their identity with Touch ID or PIN code (Soft Token), and an additional button press if using iDENprotect^{plus} to provide proof that the holder of the iDENprotect device is the owner of the private key. The authentication request follows a challenge-response model where the iDENprotect server issues a challenge for which the Token responds successfully.

The private key stored on iDENprotect device never leaves the Secure Enclave (when using Soft Token) or secure realm of the iDENprotect^{plus} (when using Hard Token).

To secure each authentication transaction, Time-based One-Time Passwords (OTPs) are generated on the iDENprotect device during enrollment and optionally during normal authentication operations.

The OTP generation has a set process within the iDENprotect device in order to meet industry accepted standards for OTP generation under RFC4226 (HMAC-based OTP algorithm) and RFC6238 (Time-based OTP algorithm). The basic OTP generation process flow can be seen in Figure 8 below, along with the services that iDENprotect can work with.

iDENprotect Keystore

iDENprotect Tokens manage cryptographic material within its robust secure realm. The keystore is based on slots within the Token, so that each key can be allocated a slot or a credential. Slots 0 to 2 are always reserved for iDENprotect itself for storing the device certificates and the iDENprotect server certificate. The corresponding private keys are stored in a protected area that replicates the certificate slots model.

The remaining slots (from number 3 onwards) are configurable through the iDENprotect server when provisioning credentials to the iDENprotect device. The permissions of each of the slots are configurable at the time of provisioning.

iDENprotect iOS SDK

The iDENprotect SDK is a collection of tools and libraries that allow you to integrate iDENprotect authentication functionality in iOS applications. The applications created with iDENprotect SDK can use either software (iDENprotect) or hardware (iDENprotect^{plus}) Tokens (or both) for authentication.

The basic architecture of iDENprotect product suite can be seen in the figure below, which illustrates the interaction between the iDENprotect device, the SDK, and ultimately the iDENprotect server and other backend authentication services for online authentication and validation.

Concepts and Terminology

In iDENprotect architecture, the product iDENprotect refers to the software authentication module and iDENprotect^{plus} to the hardware authentication module. However, in iDENprotect SDK methods and tools they're generally referred to as soft token and hard token, respectively.

In addition, many components in the iDENprotect SDK use the previous name - iDENtear - of the iDENprotect solution. In fact, the whole SDK package is still named `iDENtearSDK`. These inconsistencies will be refactored out in a future iDENprotect SDK release.

SDK Architecture and Contents

The iDENprotect SDK is based on the Functional Reactive Programming (FRP) paradigm which is based around composing and transforming streams of values. This allows the use of simplified and top down readable code that is asynchronous by nature. iDENprotect SDK uses Objective-C ReactiveCocoa framework, which is automatically installed as a module in Xcode projects when you take the SDK into use.

The iDENprotect SDK contains the following types of components within the package:

- Header files for accessing the iDENprotect Soft Token and iDENprotect^{plus} Hard Token
- Header files for communicating with the backend iDENprotect server
- Fat libraries for all iOS platforms and macOS
- Example projects

The iDENprotect tools within the iDENprotect SDK package perform a number of useful functions that can be used for a variety of different tasks when developing applications for iOS. These tools provide the capability to:

- Develop iDENprotect applications on the iOS platform
- Check the status of the iDENprotect device
- Access iDENprotect server backend services for integration and iDENprotect device provisioning

Prerequisites

iDENprotect SDK can be used on all OS X / macOS platforms with up-to-date Bluetooth Smart API. At least OSX 10.9 Mavericks and later releases are known to work.

Before starting with iDENprotect SDK, make sure that you have the following components installed on your system:

- Xcode development environment (available from Apple Developer)
- CocoaPods dependency manager. Install it with command

```
sudo gem install cocoapods
```

iDENprotect SDK is available from iDENprotect Ltd. on request. Contact us if you are interested in developing iDENprotect applications.

The applications built with iDENprotect SDK have the following requirements for their deployment targets:

- iOS 9 or newer operating system
- Touch ID biometric authentication (when using iDENprotect Soft Token authentication)

Using iDENprotect SDK

The easiest (and heavily recommended) way to start using iDENprotect SDK is by installing the Xcode development environment first, and then including the iDENprotect SDK in the Xcode projects using the CocoaPods dependency manager. It is also possible to include iDENprotect SDK as a framework in your Xcode project, but that method is prone to dependency conflicts and therefore not officially supported.

You need online connectivity to set up iDENprotect SDK.

1. In Xcode, create a new Project with the following settings:
   a. `iOS` build target
   b. `Single View Application` application template

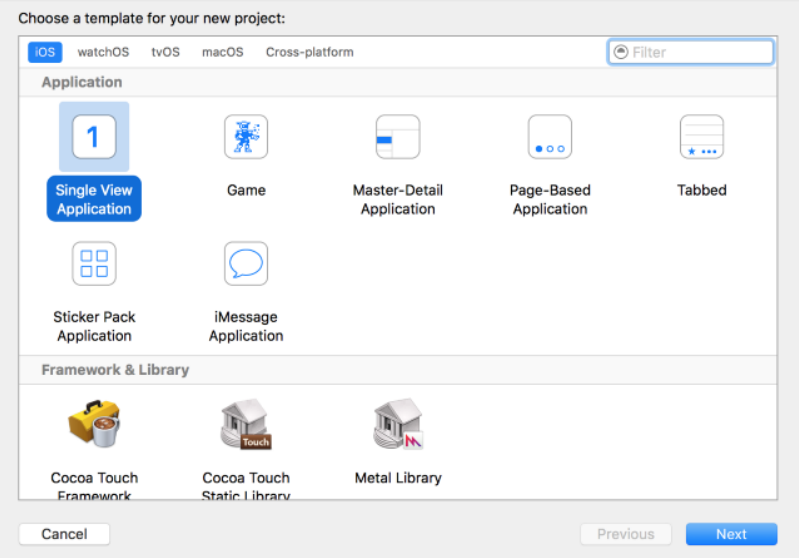

Figure 4. Creating Single View Application

2. In terminal, navigate to the project directory of your new Xcode project
3. Create a blank `Podfile` for the project:

```
pod init
```

4. Add a link to the `iDEntearSDK.podspec` specification file (located within the SDK package) in the `PodFile`:

```
pod 'iDENtearSDK', :path => '<path to iDENprotect SDK podspec
file>iDENtearSDK.podspec'
```

The .podspec file is located in the main iDENprotect SDK directory

5. Install the SDK contents in the Xcode project

```
pod install
```

6. Navigate to your Xcode project directory, and launch the `.xcworkspace` file. This starts Xcode with the correct workspace.

Tutorial - Connection Test App (Hard Token)

This tutorial demonstrates how to create your first iDENprotect application with the iDENprotect SDK. The application connects to an iDENprotect^{plus} token over Bluetooth and prints its serial number, which is a useful way to verify that you can build and run apps that are able to communicate with the iDENprotect^{plus} token. This tutorial is only intended to demonstrate some basic concepts in iDENprotect development and to verify that the iDENprotect SDK is being used correctly in your development environment. Real-world iDENprotect applications contain a lot of additional functionality not featured in this tutorial.

This tutorial application has the following features:

- Connection from development workstation to iOS device
- Bluetooth connection between iOS device and iDENprotect^{plus} token
- Read status data from the iDENprotect^{plus} token

Creating Tutorial App UI

1. Create a new Xcode project with iDENprotect SDK, as described in section [Setting up SDK]
2. Open the `.xcworkspace` workspace file
3. Create a basic UI that contains at least a Button and a Label. If you're not familiar with iOS UI development in Xcode, see Build a Basic UI in Apple Developer.

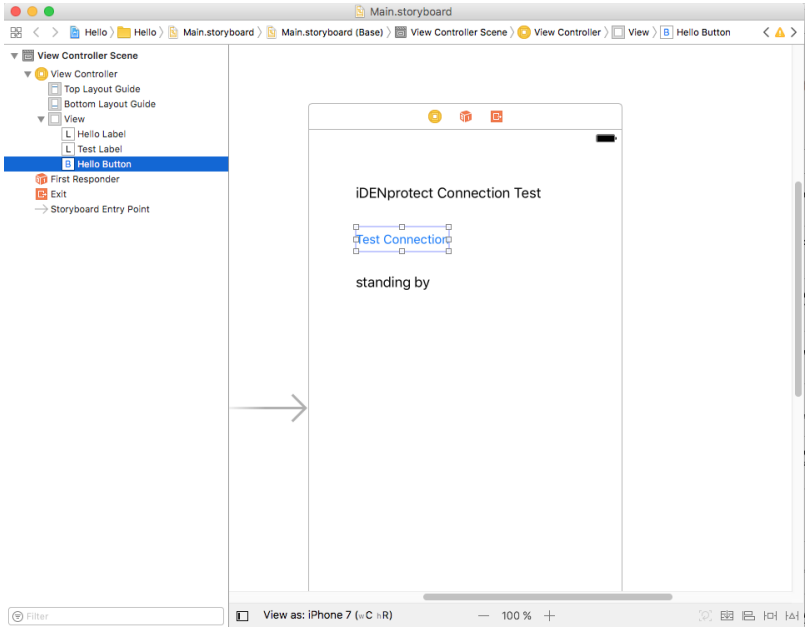

Figure 5. Tutorial App Storyboard

The example screenshot contains:

- Static application title Label *iDENprotect Connection Test* in `Hello Label` controller
- Clickable Button *Test Connection* in `Hello Button` controller
- Status text Label *standing by* in `Test Label` controller

Create at least a clickable Button and the status text Label in your application. The Button will be used to launch connection to the token, and the Label to display the token serial number.

Creating Tutorial Functionality

If you used the Single View Application template, Xcode creates an empty project structure with all necessary files to build a blank iOS application. In this tutorial, you only need to make changes in one project file - the `ViewController.m` implementation file. For a quick reference on `ViewController.m` and other Xcode project files, see Concepts and Terminology, Xcode Project Files Explained at Code with Chris.

Importing SDK Header

The `ViewController.m` file begins with a list of imported header files and other resources.

The `iDENtearSDK.h` contains additional import links to further SDK resoures, freeing you from the hassle of manually importing all other iDENprotect SDK submodules. Add the line `\#import iDENtearSDK.h` along with the rest of the import commands to import the iDENprotect SDK main header file into your project.

```
#import "ViewController.h"
#import "iDENtearSDK.h"
```

Connecting to Token and Reading Status Attributes

You can operate on the iDENprotect device - whether it's a iDENprotect soft token or iDENprotect^{plus} physical token - using the `iDENtearSDK` object. The `iDENtearSDK` object has a `setMode` attribute that defines which token type is being used. To use the hard token, set `setMode:iDENtearModeBluetooth`.

The iDENtearSDK object contains a list of status attributes (`iDENtearDeviceStatus`) as a JSON object. One of the status attributes is the `serialNumber` attribute, which we'll print in the Test Label with the following code snippet. The code snippet contains the device connection and status polling code inside a function called `connectToDevice`:

```
- (void)connectToDevice {
    [[iDENtearSDK shared] setMode:iDENtearModeSoftToken];  ①
    @weakify(self);
    [[[[iDENtearSDK shared] connect] flattenMap:^RACStream *(id value) {
        return [[iDENtearSDK shared] getStatus];  ②
    }] deliverOnMainThread] subscribeNext:^(iDENtearDeviceStatus *status) {
        @strongify(self);
        NSLog(@"status");  ③

        self.testLabel.text = status.serialNumber;  ④

    }];

}
```

① Creates the iDENprotect device object

② Returns a device status attributes JSON object

③ (optional) Prints the contents of the status object in the Xcode console log

④ Prints the device serial number in the `testLabel` Label

Mapping the Action to Button

When you drag and drop the UI elements (Button & Label) into the Workspace, Xcode automatically creates some IB (Interface Builder) macros for them. The macros are created with Objective-C -compliant naming conventions where the Hello Button is referred to as `helloButton` and the Test Label as `testLabel`.

Find the `<buttonNameClick>` macro and add a call to the previously created connection function inside it:

```
- (IBAction)helloButtonClick:(id)sender {
    [self connectToDevice];
}
```

Running the Application

To be able to run the application, you need to have an iDENprotect-compliant iOS device connected to your development workstation. When a device is connected, Xcode recognizes it automatically and you can select it as the build target in the top toolbar.

To deploy the application on the iOS device and run it, press the "Play" button in the top toolbar. This builds the application, deploys it, and launches it on the device automatically.

Developing iDENprotect Applications

Application Requirements

Each iDENprotect application requires that the the following iDENprotect elements are set up correctly:

- Authentication mode for the application. Available modes are:
  - `iDENtearModeSoftToken` for iDENprotect Soft Token authentication
  - `iDENtearModeBluetooth` for iDENprotect^{plus} Hard Token authentication
- Backend iDENprotect server connection URL with `.backendHost` parameter
- Login credentials to access the iDENprotect server. The credentials are the same as for the iDENprotect server Management Console web login.

You can set them in the application's `AppDelegate` or `ViewController` implementation files. The following example code sets all the required parameters in the `setupSDK` method, which is then called inside the `AppDelegate` method:

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
    [self setupSDK];
    return YES;
}

- (void)setupSDK {
    [[iDENtearSDK shared] setMode:iDENtearModeBluetooth]; ①
    [iDENtearSDK shared].backendHost = @"iden.example.com"; ②
    [[iDENtearSDK shared] setBackendCredentialsLogin:@"admin" password
:@"examplePASS"]; ③
}

@end
```

① Sets authentication Token type

② iDENprotect server URL

③ iDENprotect server login credentials

It can be useful to add a shorthand definition such as `SDK` for the `iDENtearSDK` object. The object will be used very often during iDENprotect development.

```
@define *SDK [iDENtearSDK shared]

[SDK setMode:iDENtearModeSoftToken];
```

When you are using iDENprotect software token mode, any user authentication attempts require that

the iOS device has Touch ID enabled. Check it with the `hasTouchIDPresentAndEnabled` function. When Touch ID is enabled, `hasTouchIDPresentAndEnabled` returns `true`:

```
[iDENtearSDK shared].device.hasTouchIDPresentAndEnabled;
```

Examples

The following examples demonstrate how you can Connect to an iDENprotect device (Soft Token or Hard Token) and take it into use on your iDENprotect server (provisioning). This section contains examples for all the steps in the provisioning process:

- Generate a DRA (Device Rights and Attributes) file and upload it to iDENprotect server
- Generate a TOTP (Time-based One-Time Password to validate the key exchange
- Enroll the device on the iDENprotect server
- Provide a means to change the device PIN Code
- (optional) remove device enrollment from the iDENprotect server

Connecting to Device

Device refers to iDENprotect Soft Token or iDENprotect^{plus} Hard Token.

Opening a connection to the iDENprotect device is usually the starting point for all iDENprotect applications. The following functions establish a connection to the device, which can then be used in authentication.

Steps

```
[[[iDENtearSDK shared] connect] flattenMap:^RACStream *(id value) {
    return [[iDENtearSDK shared] getStatus];
}];
```

Disconnecting from device

After you no longer need an active connection to the device, it's good practice to close the connection:

Steps

```
[[iDENtearSDK shared] closeConnection] subscribeNext:^(id x) {}];
```

Closing the connection is also required in order to write persistent changes to the Token. For example, changing the user's PIN code requires closing the connection before the change takes effect.

Generating and Uploading DRA Files

DRA (Device Rights and Attributes) files are device attribute packages that identify the Token. The DRA file is uploaded to iDENprotect server during device registration, and it is used to initialize the device on the iDENprotect server database.

Steps

1. Connect to Device
2. Get the DRA file from the device with the `getDRA` function, which returns an `iDENtearDeviceDRA` object. When using iDENprotect[plus], `getDRA` requires the token administrator username and PIN to access the device:

```
- (RACSignal *)getDeviceDRA {
    if ([iDENtearSDK shared].mode == iDENtearModeSoftToken) {
        return [[iDENtearSDK shared] getDRA];
    } else {
        return [[[iDENtearSDK shared] loginWithUsername:@"applyAdmin"
withPin:@"Sdff3Grsdgf24asdfGHafg43JGH"] flattenMap:^RACStream *(id value) {
            return [[iDENtearSDK shared] getDRA];
        }];
    }
}
```

3. Send the `iDENtearDeviceDRA` object contents to iDENprotect server with the `backendRegisterDevice` function, along with the user's email address:

```
- (IBAction)uploadDRA:(id)sender {
    RACSignal *registerDRA = [[self getDeviceDRA] flattenMap:^RACStream
*(iDENtearDeviceDRA *deviceDRA) {
        return [[iDENtearSDK shared] backendRegisterDevice:deviceDRA
forUserWithEmail:<my.email@address.com>];
    }];
```

4. After the DRA file has been uploaded to the iDENprotect server, the user usually has to wait until the device registration has been processed in the backed iDENprotect server, which can take some time. Best practice in this situation is to close the connection to the device:

```
[[iDENtearSDK shared] closeConnection] subscribeNext:^(id x) {}];
```

Generating and Verifying TOTP

The iDENprotect device requires use of TOTPs (Time-based One-Time Passwords) during the device enrollment process and during user-initiated PIN reset requests. Optionally, TOTPs can also be used during authentication to provide an additional layer of procedural security.

Steps

1. Connect to Device
2. Generate a TOTP with the SDK's `TOTP` function. This returns an `NSDictionary` response which contains the following parameters:
   - `otp` - the OTP
   - `time` - timestamp of current time
   - `deviceSerialNumber` - full device serial number text string
3. Send and verify the TOTP with the backend iDENprotect server using the SDK's `backendVerifyTOTP` function.

```
- (void)p_startProcess {
    @weakify(self);
    RACSignal *otpSignal = [[[iDENtearSDK shared] TOTP] flattenMap:^RACStream
*(NSDictionary *response) {  ①
        NSString *otp = [response valueForKey:@"otp"];  ②
        NSNumber *time = [response valueForKey:@"time"];
        NSString *deviceSerialNumber = [response valueForKey
:@"deviceSerialNumber"];
        [[NSOperationQueue mainQueue] addOperationWithBlock:^{
            @strongify(self);
        }];
        return [[iDENtearSDK shared] backendVerifyTOTP:otp withTime:time
withDeviceSerial:deviceSerialNumber];  ③
    }];

}
```

① `TOTP` function

② Start of parameters list

③ TOTP verification function

Enrolling Devices

Enrollment is the part of iDENprotect provisioning where a device registered on the iDENprotect server gets mapped and verified to its assigned user. After enrollment, the device (using either Soft Token or Hard Token) is fully operational for authentication on services managed by that iDENprotect server

Successful enrollment requires an Activation Code. The Activation Code is delivered to the user via email after the device has been assigned to the user either automatically (when using Active Directory) or by iDENprotect Administrator action in iDENprotect server Management Console (when using manual enrollment).

Steps

1. Connect to Device.
2. Launch the enrollment process with the `enrollWithActivationCode` function, which requires the Activation Code as a parameter.

You'll have to provide an UI element for the user to enter the Activation Code in.

```
RACSignal *enrollSignal = [[[iDENtearSDK shared]]
enrollWithActivationCode:<user's activation code>] subscribeNext:^(id x) {
 ①
    // Actions in the application UI when enrollment is done
} error:^(NSError *error) {
    // Error handling
}];
```

① User-specific Activation Code

3. For Soft Token implementation, you will have to add some more certificate processing functions, which are handled automatically for Hard Tokens. The following code snippet selects 2 slots in the Token keystore, and sends the Token certificate and a CSR (Certificate Signing Request) to the iDENprotect server to be signed. After the iDENprotect server has signed them, the device imports the signed certificate.

```
- (RACSignal *)p_secEncSignal {
    return [[[[[[[[iDENtearSDK shared] eraseCertificateInSlot:3]
flattenMap:^RACStream *(id value) {
        return [[iDENtearSDK shared] eraseCertificateInSlot:4];
    }] flattenMap:^RACStream *(id value) {
        iDENtearGeneraterateKeypairAttributes *attr =
[[iDENtearGeneraterateKeypairAttributes alloc] init];
        attr.storeSlot = 3; ①
        attr.csrSlot = 4;
        attr.generationMode = iDENtearGenerateKeypairModeSecurityEnclave; ②
        attr.certificate = [[iDENtearCertificateAttributes alloc]
initWithCommonName:@"iDENtear Secenc certificate 01" organization:@"iDENtear"
organizationUnit:@"iDENtear" country:nil];
        return [[iDENtearSDK shared] generateKeypair:attr];
    }] flattenMap:^RACStream *(id value) {
        return [[iDENtearSDK shared] getCertificateFromSlot:4];
    }] flattenMap:^RACStream *(id value) {
        iDENtearCertificateWithDevice *cert = [[iDENtearCertificateWithDevice
alloc] init];
        cert.certificate = [value valueForKeyPath:@"certificate"];
        cert.signature = [value valueForKeyPath:@"signature"];
        cert.certificateProfile = @"SECENC"; ③
        cert.endEntityProfile = @"iDENtearTokenCA"; ④
        cert.deviceSerial = <device serial number>; ⑤
        cert.tag = @"secenc";
        return [[iDENtearSDK shared]
backendAddToDeviceAndSignCertificateWithDevice:cert];
    }] flattenMap:^RACStream *(id value) {
        iDENtearCertificate *certificate = [iDENtearCertificate
certificateFromResponseString:[value valueForKey:@"certificate"]];
        return [[iDENtearSDK shared] importCertificate:certificate toSlot:3
]; ⑥
    }] flattenMap:^RACStream *(id value) {
        return [[iDENtearSDK shared] eraseCertificateInSlot:4];

    }] flattenMap:^RACStream *(id value) {
        return [[iDENtearSDK shared] listCertificates];
    }];
}
```

① This example uses hardcoded keystore and CSR slots. To select the "next available" slot, enter `0` as value here

② Sets Secure Enclave mode - required for Soft Token implementations

③ This must match the Certificate Profile name on iDENprotect server

④ This must match the End Entity Profile on iDENprotect server

⑤ Device serial number required for identification. See Tutorial - Connection Test App (Hard Token) for information on how to read the serial number

⑥ Signed certificate is imported to keystore slot 3

These settings are dependent on Certificate Profiles and End Entity Profiles in your CA (Certificate Authority) configuration. The EJBCA Configuration in document Installing iDENprotect server for RHEL 7.

Changing PIN Code

The user should have the option to change the PIN code in all iDENprotect applications. We recommend that you implement this function after successful enrollment in your applications.

Steps

1. Connect to Device

```
RACSignal *changePinSignal = [[[iDENtearSDK shared] changePin:<current PIN>
toNewPin:<new PIN>] flattenMap:^RACStream *(id value) {

    return [[iDENtearSDK shared] closeConnection];
}];

    [[changePinSignal deliverOnMainThread] subscribeNext:^(id x) {
        [self performSegueWithIdentifier:@"EnrollmentComplete" sender:nil];
    } error:^(NSError *error) {
        // error handling
        }];
```

Un-enrolling Devices

For administrative purposes, it is possible to create an application that removes a Token's enrollment from the iDENprotect server. To do this, you have to know the Administrator login and PIN for the Token.

Steps

1. Connect to Device
2. Login to device with admin privileges:

```
[[iDENtearSDK shared] loginWithUsername:@"<admin login>" withPin:@"<admin
PIN>"]
```

3. Then to reset device use:

```
[[iDENtearSDK shared] unEnroll]
```

This only removes registration information from the iDENprotect device itself. The iDENprotect Administrator has to remove the device registration from the backend iDENprotect server as well to complete the removal.

Command Line Utilities

The iDENprotect SDK package includes a downloadable set of command line utilities for managing single iDENprotect devices locally over a Bluetooth connection. Using the tools requires 64-bit macOS operating system (version 10.9 Mavericks or later).

The utilities are available as executable binaries from iDENprotect Ltd.

The following table lists available commands provided by the iDENprotect command line utilities. Running a command establishes a Bluetooth paired connection with the iDENprotect device, authenticates with the device, performs the command, and disconnects. Some commands require you to enter the device PIN.

Table 1. CLI Commands

Command	Description	Usage
`getdra`	Reads DRA (Device Registration Attributes) information from the iDENprotect device and writes them into iDENtear.dra file in the directory where the command is run.	`./getdra`
`listcerts`	Lists certificates on the iDENprotect device slots 0-8.	`./listcerts -pin <PIN>`
`erasecert`	Deletes a certificate and corresponding private key on the designated slot of the device. Slots 0-2 are normally reserved for built-in certificates.	`./erasecert -slot <3-8> -pin <PIN>`
`genkeypair`	Generates a new elliptic curve key pair on the device, with optional parameters for Common Name (CN), Organization (O), and Organizational Unit (OU). The command can also be used to create a corresponding Certificate Signing Request in another slot. The selected slot(s) must be empty.	`./genkeypair -slot <3-8> -cn <"CN"> -o <"O"> -ou <"OU"> csrslot <3-8>`
`getcert`	Reads a certificate on the device and outputs it to `stdout` in PEM format.	`./getcert -slot <0-8> -pin <PIN>`
`importcert`	Imports a PEM-formatted certificate from the local file system to the device. If a certificate is imported into a slot that already contains one, the existing certificate is overwritten. The corresponding private key is not affected.	`./importcert -slot <3-8> pin <PIN> -cert <filename>`
`resetpin`	Resets the PIN code in the iDENprotect device, and creates a new PIN code and an activation code.	`./resetpin -code <activationCode> -pin <newPIN>`
`sigverify`	Takes a file named `message.txt`, signs its SHA256 hash with the device key stored in slot 1, ands stores the corresponding certificate as test.pem and signature as `test.sig`. The signature is verified with a backend service.	`./sigverify -pin <PIN>`

Examples

The following is an example use of the command line utilities. The existing certificates are listed, and after verifying that slots 3 and 4 are empty, a new Certificate (`genkeypair`) and a new Certificate Signing Request are created in those slots. The CSR is then read and written into the file `test.csr`.

The device in the example uses PIN code 1212.

```
./listcerts -pin 1212
2016-10-08 16:34:36.570 listcerts[2229:1003] Opening connection to iDENprotect
2016-10-08 16:34:43.592 listcerts[2229:507] Handshake Completed
2016-10-08 16:34:45.017 listcerts[2229:507] serial: AMC0010201-000083
2016-10-08 16:34:45.018 listcerts[2229:507] Uptime: 1179031
2016-10-08 16:34:46.592 listcerts[2229:507]
Slot CN
--   --
0 GB
1 iDENprotect Device Certificate 01
2 GB
3 (empty)
4 (empty)
5 (empty)
6 (empty)
7 (empty)
8 CSR

./genkeypair -slot 3 -cn “My Company” -csrslot 4 -pin 1212
./getcert -slot 4 -pin 1212 > test.csr
```

After this, the CSR file can be used to create a new certificate that is signed with certificate `cert.pem` and ECDSA key `eckey.pem` included in the downloaded utilities package, and is imported into the device to replace the old certificate in slot 3.

OpenSSL version in macOS does not support ECDSA properly. You may need to use another version, for example, from MacPorts.

```
/opt/local/bin/openssl x509 -req -days 3650 -in temp.csr -CA
/Users/iDENprotect/demoCA/cert.pem -CAkey /Users/iDENprotect/demoCA/eckey.pem
-CAcreateserial -out temp.pem 2>/dev/null
./importcert -slot 3 -cert temp.pem -pin 1212
```

OpenSSL can also be used to verify the certificates and signatures. The following example stores the result digest as `test.dgst`:

```
/opt/local/bin/openssl dgst -sha256 < message.txt
/opt/local/bin/openssl x509 -pubkey -noout -in test.pem > pubkey.pem
/opt/local/bin/openssl dgst -sha256 -verify pubkey.pem -signature test.sig
test.dgst
```

Appendix A: REST API Reference

Many actions between the iDENprotect device and the iDENprotect server are handled by the iDENprotect application making calls to iDENprotect server's REST API. Usually, you don't need to concern yourself about the format of these calls since the SDK functions do the job automatically. However, this section lists the iDENprotect server REST APIs for reference.

A Swagger environment for the REST APIs is created on the iDENprotect server Web Management Console in the URL `https://<server-url>/swagger-ui.html`.

/api/device/status

Called by `getStatus` and `requestUpdateStatusToServer` functions.

Returns a NSDictionary object containing device information.

Table 2. Response Contents

Command	Description	Data Type
battery	Remaining battery charge (if using iDENprotect^{plus} Hard Token	Integer `0 ... 100`
debug		String
deviceSerialNumber	Token's serial number	String
expiry	Token's expiry date (if set)	Datetime
isActivated	Is the Token in active state	Boolean (0 = inactive, 1 = active)
isEnrolled	Has the Token been enrolled	Boolean (0 = not enrolled, 1 = enrollment completed)
sp		String
time	Current timestamp - used to synchronize Token time with iDENprotect server time for TOTP generation	Datetime
uptime	How long has the Token been in use (if using iDENprotect^{plus})	
version	Firmware version on the Token (if using iDENprotect^{plus})	String

/api/devices/register1

Called by `getDRA` function.

Generates a DRA (Device Rights and Attributes) object on the Token. Usually followed by uploading the DRA object on the iDENprotect server for validation (with the `uploadDRA` function).

Table 3. Response Contents

Command	Description	Data Type
activationCode	Activation Code to be used during enrollment	string

Command	Description	Data Type
certificate	Device public certificate for signing	string
emailAddress	User's email address during registration	string
serialNumber	Token serial number	string

Appendix B: iDENprotect Keystore

iDENprotect Soft Tokens and iDENprotect^{plus} Hard Tokens have a secure keystore with many security measures designed to protect keys and their usage, and to ensure that keys cannot be tampered with or removed from the device.

iDENprotect has an unlimited amount of slots available for key storage. iDENprotect^{plus} devices are limited to 15 slots.

Table 4. Keystore Slot Number Table

Slot Number	Key	Key Type	Key Usage	Key Generation
Slot 0	<System reserved>			
Slot 1	Device Key	ECDH / ECDSA	Encrypting and decrypting Bluetooth communications, and decrypting iDENprotect server asset transfers	Generated during initial startup by device RNG. Signed during enrollment by iDENprotect server.
Slot 2	iDENprotect server Certificate	ECDH / ECDSA		Downloaded during enrollment
Slot 3	Signing Key	ECDSA at device's secure hardware	Local signing key	Created at enrollment
Slot 4	<Spare>			
Slot 5	TOTP (Time-based One-Time Password) Key	AES-256	Used in SSO applications	Created at enrollment
Slot 6	<Spare>			
Slot 7	MDM (or MAM) Encryption Key from MDM Provider (such as Good Dynamics)	Depends on provider	Management Container encryption	Created at MDM initial setup

Appendix C: Database Schema

This section describes the iDENprotect server device database schema. If you are using iDENprotect server with the default database installed, all database operations are handled automatically. If you're using a custom database system, you have to create the following tables in the iDENprotect database for the system to work correctly.

Table 5. DEVICES Table

Description	Field	Type	Null	Key	Default	Extra
	ID	bigint(20)	NO	PRI	NULL	auto_increment
	SERIAL_NUMBER	varchar(64)	NO	UNI	NULL	
	ACTIVATION_CODE	varchar(16)	NO		NULL	
	CERTIFICATE	varchar(4096)	YES		NULL	
	DEVICE_STATE_ID	bigint(20)	NO	MUL	NULL	
	LAST_SUCCESSFUL_OTP_VALUE	bigint(20)	YES		NULL	
	LAST_SUCCESSFUL_OTP_DATETIME	timestamp	YES		NULL	
	IS_LOCKED	bit(1)	NO		b'0'	
	OTP_KEY	blob	YES		NULL	
	EXPIRY_DATE	timestamp	YES		NULL	
	LOGIN_SUCCESS_COUNT	bigint(20)	YES		NULL	
	LOGIN_FAILURE_COUNT	bigint(20)	YES		NULL	
	BATTERY_CHARGE	int(11)	YES		NULL	
	USAGE_SECONDS	bigint(20)	YES		NULL	
	PIN_RESET_FLAG	bit(1)	NO		b'0'	
	PIN_CHANGE_FLAG	bit(1)	NO		b'0'	
	OTP_FAILURE_COUNT	bigint(20)	YES		NULL	
	SW_VERSION	varchar(32)	YES		NULL	
	PRODUCT_LINE_ID	bigint(20)	NO	MUL	1	

Table 6. DEVICES_EXTRA_CERTIFICATES Table

Description	Field	Type	Null	Key	Default	Extra
	TAG	varchar(32)	NO	MUL	NULL	
	CERTIFICATE	varchar(4096)	NO		NULL	
	DEVICE_ID	bigint(20)	NO	MUL	NULL	
	ID	int(11)	NO	PRI	NULL	auto_increment

Table 7. SERVER_CREDENTIALS Table

Description	Field	Type	Null	Key	Default	Extra
	ID	bigint(20)	NO	PRI	NULL	auto_increment
	CREDENTIALS_TYPE	int(11)	NO	UNI	NULL	
	CERTIFICATE	blob	NO		NULL	
	PRIVATE_KEY	blob	NO		NULL	
	COMMON_NAME	varchar(255)	YES		NULL	
	ORGANISATIONAL_UNIT	varchar(255)	YES		NULL	
	ORGANISATION	varchar(255)	YES		NULL	
	LOCALITY	varchar(255)	YES		NULL	
	STATE	varchar(255)	YES		NULL	
	COUNTRY	varchar(255)	YES		NULL	

Appendix D: iDENprotect SDK API Reference

As noted earlier in this guide, iDENprotect interface API is based on Objective-C ReactiveCocoa (`ReactiveObjC`) signals framework.

All iDENprotect functions produce the standard `NSError` as return signal when an error occurs. In these situations, the transactional sequence is stopped.

Appendix D: iDENprotect SDK API Reference

As noted earlier in this guide, iDENprotect interface API is based on Objective-C ReactiveCocoa (ReactiveObjC) signals framework.

All iDENprotect functions produce the standard NSError as return signal when an error occurs. In these situations, the transactional sequence is stopped.

Method Documentation

activateWithCode:()

```
- (RACSignal *) activateWithCode:      (NSString *)    code
```

Activate Token for OTP use, with Activation Code generated during Token registration given as a parameter. The activation code should be distributed to the user by separated means such as email.

Parameters

code Activation code

Returns

Send next on success activation

backendAddToDeviceAndSignCertificateWithDevice:()

```
- (RACSignal *)
backendAddToDeviceAndSignCertificateWithDevice:      (iDENtearCertificateWithDevice *)    certificateWithDevice
```

Sign the given certificate request and attach it to the given device as an extra certificate.

Parameters

certificateWithDevice iDENtearCertificateWithDevice object

Returns

Send next on success

backendEnrollmentCompleteWithObject:()

```
- (RACSignal *)
backendEnrollmentCompleteWithObject:      (iDENtearEnrollEncryptedIdentifyObject *)    encryptedIdentifyObject
```

Finalize iDENtear enrollment with backend server via web service

Parameters

encryptedIdentifyObject enrollment completion object from the Token

Returns

RACSignal with enrollment completion status

backendEnrollmentIdentifyWithObject:()

```
- (RACSignal *) backendEnrollmentIdentifyWithObject:      (iDENtearEnrollIdentifyObject *)    enrollIdentifyObject
```

Continue iDENtear enrollment with backend server via web service

Parameters

enrollIdentifyObject enrollment identification object from the Token

Returns

RACSignal with enrollment initiation object

backendEnrollmentInitiate()

```
- (RACSignal *) backendEnrollmentInitiate
```

Initiate iDENtear enrollment with backend server via web service

Returns

RACSignal with enrollment initiation object

backendPINResetCompletion:withStatus:()

- (RACSignal *) backendPINResetCompletion:	(NSString *)	nonce
withStatus:	(NSString *)	status

Mark PIN Reset process as completed in the iDENprotect server

Used for statistics only

Parameters

nonce	code from the Token
status	from the Token (OK/NOK)

Returns

Send next on success

backendPINResetStart:withDeviceSerial:withSignature:()

- (RACSignal *) backendPINResetStart:	(NSString *)	nonce
withDeviceSerial:	(NSString *)	deviceSerial
withSignature:	(NSString *)	signature

Start PIN Reset process with the iDENprotect server

Parameters

nonce	code from the Token
signature	signature from the Token
deviceSerial	Token serial number

Returns

RACSignal PIN Reset continuation object

backendRegisterDevice:forUserWithEmail:()

- (RACSignal *) backendRegisterDevice:	(iDENtearDeviceDRA *)	deviceDRA
forUserWithEmail:	(NSString *)	email

Register device DRA information in iDENprotect server

Parameters

deviceDRA	iDENtearDeviceDRA object
email	e-mail address (optional)

Returns

Send next on success

backendVerifySignatureWithHash:withSignature:withDeviceSerial:certificateTag:()

- (RACSignal *) backendVerifySignatureWithHash:	(NSString *)	hash
withSignature:	(NSString *)	signature
withDeviceSerial:	(NSString *)	deviceSerial
certificateTag:	(NSString *)	certificateTag

Verify signature generated by Token with a device certificate stored in the iDENprotect server

To use this method, the Token must sign the data with device key, stored in slot 1.

Parameters

hash	base64-encoded SHA256 hash of the original data
signature	signature object from the Token
deviceSerial	Token serial number
certificateTag	tag of certificate

Returns

RACSignal true/false

backendVerifyTOTP:withTime:withDeviceSerial:()

- (RACSignal *) backendVerifyTOTP:	(NSString *)	otp
withTime:	(NSString *)	time
withDeviceSerial:	(NSString *)	deviceSerial

Verify TOTP generated by Token with a shared key stored in the iDENprotect server

Parameters

otp	TOTP code from the Token
time	time from the Token
deviceSerial	Token serial number

Returns

RACSignal true/false

beginSigningFromSlot:()

- (RACSignal *) beginSigningFromSlot:	(iDENtearSlot)	slot

Begin SHA256/ECDSA signing operation The general signing operation is started with this method. The Token continues to calculate the SHA256 hash of the data until finalizeSigning is called. The data to be signed is uploaded to the Token with signData method. The slots needs an associated private key.

Parameters

slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

Send next when device is ready to begin signing

changePin:toNewPin:()

- (RACSignal *) changePin:	(NSString *)	oldPin
toNewPin:	(NSString *)	newPin

Change the PIN code

Parameters

oldPin	Old PIN code
newPin	New PIN code

Returns

Send next on successful PIN code change

closeConnection()

- (RACSignal *) closeConnection

Close connection to the Token

Returns

Next value after connection is dropped successful

connect()

- (RACSignal *) connect

Connect to Token

Returns

Next value after connection is established

encryptData:withAESOFBKeyFromSlot:()

```
- (RACSignal *) encryptData:      (NSData *)      data
    withAESOFBKeyFromSlot:      (iDENtearSlot)    slot
```

Run binary data via Token's AES128/OFB cipher using secret key in slot.

Parameters

data	Binary data to encrypt
slot	AES OFB Key slot

Returns

Encrypted data

enrollContinueWithObject:()

```
- (RACSignal *) enrollContinueWithObject:      (iDENtearBackendVerifiedIdentifyObject *)      verifiedIdentifyObject
```

Continue enrollment with EnrollmentIdentificationResponse object from backend server

Parameters

verifiedIdentifyObject	EnrollmentIdentificationResponse object

Returns

iDENtearEnrollEncryptedIdentifyObject

enrollFinalize()

```
- (RACSignal *) enrollFinalize
```

Finalize enrolment with status (OK/NOK) from backend iDENprotect server (OK=201 received from server)

Returns

certificate Signed certificate in PEM format

enrollInitWithObject:()

```
- (RACSignal *) enrollInitWithObject:      (iDENtearBackendEnrollInitObject *)      enrollObject
```

Initiate enrollment with EnrollmentInitiationResponse object from backend server

Parameters

enrollObject	EnrollmentInitiationResponse object

Returns

iDENtearEnrollIdentifyObject

enrollWithActivationCode:()

```
- (RACSignal *) enrollWithActivationCode:      (NSString *)      activationCode
```

Helper method that simplifies the whole enroll process to one method call instead of you having to call the enrollment methods in the right order.

This method does the following:

- Activate with Activation Code
- Erease iDENprotect server certificate from Token
- Get new server certificate
- Import new server certificate
- Get server time signature
- Set server tiem signature to device
- Enrollment steps (Initiate, Continue, Finalize)

Parameters

activationCode	Activation code from user

Returns

Send next on success enroll

eraseCertificateInSlot:()

```
- (RACSignal *) eraseCertificateInSlot:      (iDENtearSlot)    slot
```

Erase certificate at slot N. Erasing requires sufficient user credentials to the slot in question.

Parameters

slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

Send next on success erease

finalizeSigning()

```
- (RACSignal *) finalizeSigning
```

Finalize SHA256/ECDSA signing operation The general signing operation is finalized with this command. The Token shall return ECDSA signed SHA256 hash of the data uploaded.

Returns

OpenSSL compatible signature payload (Base64 ASN.1).

FWUpgradeWithHash:andSignature:()

```
- (RACSignal *) FWUpgradeWithHash:      (NSString *)    hash
                          andSignature:      (NSString *)    signature
```

Prepare for firmware upgrade. Token will reset and expects connection from programmer software. Requires admin login. Note: Firmware upgrade process cannot be cancelled!

Parameters

hash	Hash of firmware
signature	Signature of firmware

Returns

Send next on success firmware upgrade

generateKeypair:()

```
- (RACSignal *) generateKeypair:      (iDENtearGeneraterateKeypairAttributes *)    attribute
```

Generate ECC keypair and certificate/certificate signing request Generated keypair uses the secp256r1 curve. The certificate attribues can optionally be input with the attr JSON string.

If the attr string is empty or incomplete, default values are used.

Parameters

attribute	iDENtearGeneraterateKeypairAttributes object

Returns

certificate and CSR if successful, and used slot number

generateSharedSecretInSlot:()

```
- (RACSignal *) generateSharedSecretInSlot:      (iDENtearSlot)    slot
```

Generate shared secret

Parameters

slot	Slot to store shared secret

Returns

Shared secret

getBackendCertificate()

```
- (RACSignal *) getBackendCertificate
```

Retrieve server public certificate via a REST API call on the server

Returns

iDENtearCertificate object

getBackendTimeSignature()

```
- (RACSignal *) getBackendTimeSignature
```

Retrieve time sync packet from backend server certificate via web service

Returns

iDENtearTimeSignature object

getBLOBFromSlot:()

```
- (RACSignal *) getBLOBFromSlot:      (iDENtearSlot)    slot
```

Get binary blob from the Token slot.

Parameters

slot Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

blob, SHA256 hash, timestamp

getCertificateFromSlot:()

```
- (RACSignal *) getCertificateFromSlot:      (iDENtearSlot)    slot
```

Get certificate at slot N.

Parameters

slot Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for slot usage)

Returns

NSString Certificate in base64-encoded PEM format.

getCertificateObjectFromSlot:()

```
- (RACSignal *) getCertificateObjectFromSlot:      (iDENtearSlot)    slot
```

Get certificate at slot N.

Parameters

slot Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

iDENtearCertificateSlot object

getDRA()

```
- (RACSignal *) getDRA
```

Generates a DRA entry.

The DRA entry of the Token contains serial number, activation code and device certificate in base64-encoded PEM format. DRA logins are only used during initial Token registration. An enrolled Token refuses this call.

Returns

iDENtearDeviceDRA object

getNonce()

```
- (RACSignal *) getNonce
```

Get session nonce from the Token for further operations

Returns

Nonce to use in other operations

getStatus()

```
- (RACSignal *) getStatus
```

Read status parameters from the Token

Returns

iDENtearDeviceStatus object

getTime()

```
- (RACSignal *) getTime
```

Get timestamp from the Token

Returns

(unsigned long long) Time in microseconds

getTypeFromSlot:()

```
- (RACSignal *) getTypeFromSlot:    (iDENtearSlot)    slot
```

Get slot type (key pair, CSR, OTP key...) at slot #.

Parameters

slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for slot usage)

Returns

iDENtearSlotType type in NSValue. nil if error.

HOTP()

```
- (RACSignal *) HOTP
```

Get next OTP number

Returns

next OTP number in sequence, based on current user's credentials.

importCertificate:toSlot:()

```
- (RACSignal *) importCertificate:    (iDENtearCertificate *)    certificate
                          toSlot:    (iDENtearSlot)             slot
```

Import certificate to slot N. The use case for import functionality is external signing for keypairs generated within the iDENprotect Token. The imported certificate is verified to contain same attributes as generated certificate, plus additional external signature. Import also requires corresponding private key in corresponding slot.

Parameters

certificate	iDENtearCertificate object
slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

Send next on success importing cert

isConnected()

```
- (BOOL) isConnected
```

Flag for active connection

Returns

Boolean indicator whether the application is connected to Token

listCertificates()

```
- (RACSignal *) listCertificates
```

List all certificates.

Returns

NSArray of iDENtearCertificateSlot objects

loginWithUsername:withPin:()

```
- (RACSignal *) loginWithUsername:    (NSString *)  username
                          withPin:    (NSString *)  pin
```

Login to the Token If username is correct, the login always succeeds initially. Invalid PIN code has causes the device to generate false OTP codes. As a result, further operations will fail.

Parameters

username	Use default user if username=""
pin	PIN code

Returns

Send next on success login

pingDevice()

```
- (RACSignal *) pingDevice
```

Ping Token to keep it alive. Only useful for Hard Token applications over Bluetooth connection.

Returns

Send next on success ping

resetDeviceToFactoryState()

```
- (RACSignal *) resetDeviceToFactoryState
```

Reset Token to factory fresh state. Requires admin login.

Returns

Send next on success reseting device

resetPin:withActivationCode:()

```
- (RACSignal *) resetPin:          (NSString *)  newPin
          withActivationCode:     (NSString *)  activationCode
```

Reset the PIN code Requires next HOTP-based Activation Code from the backend iDENprotect server.

Parameters

newPin	New PIN code
activationCode	Next activation code, from the backend

Returns

Send next on success set new PIN

setBackendCredentialsLogin:password:()

```
- (void) setBackendCredentialsLogin:    (NSString *)  login
                           password:    (NSString *)  password
```

Set credentials that are required for some backend iDENprotect server requests

Parameters

login	Admin login to iSPA server
password	Admin password to iSPA server

setMode:()

```
- (void) setMode:    (iDENtearMode)  mode
```

Set mode of peripheral connection in SDK Default mode is iDENtearModeNotSelected - no mode selected

Parameters

mode	iDENtearMode enum

setTimeWithTimeSignature:()

```
- (RACSignal *) setTimeWithTimeSignature:      (iDENtearTimeSignature *)  timeSignature
```

Set time (32 bit Unix time). Requires proper signature in time message, which can be generated by the backend iDENprotect server. Token syncs it internal clock based on the three presented times.

Parameters

timeSignature	iDENtearTimeSignature object

Returns

Send next on success set time

shared()

```
+ (instancetype) shared
```

Singleton for the Token connection.

Returns

Singleton of Token object

signData:()

```
- (RACSignal *) signData:      (NSString *)  data
```

Upload data packet for hashing/signing Uploaded data is hashed on-line. Maximum data packet size is 512 bytes in binary. Note: Long operations will drain the battery if using Hard Token authentication!

Parameters

data	Base64-encoded data packet. Maximum size 512 bytes in binary

Returns

Send next on successful signing

storeBLOB:inSlot:()

```
- (RACSignal *) storeBLOB:      (NSData *)       BLOB
                    inSlot:     (iDENtearSlot)   slot
```

Store binary blob to the Token slot. Max size 512 bytes. Note: Method allows also update of existing blob with new data without erasing.

Parameters

BLOB	NSData with max size 512 bytes
slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

Send next on success store BLOB

storeCertificate:inSlot:()

```
- (RACSignal *) storeCertificate:      (iDENtearCertificate *)  certificate
                    inSlot:            (iDENtearSlot)           slot
```

Store certificate to Token slot N. Certificate must be in base64-encoded PEM format and the slot must be free.

Parameters

certificate	iDENtearCertificate object
slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

Send next on success store cert in slot

storePrivateKey:inSlot:()

- (RACSignal *) storePrivateKey:	(NSString *)	privateKey
inSlot:	(iDENtearSlot)	slot

Store ECC private key to Token slot N. Private key must be in base64-encoded PEM format (OpenSSL compatible). Note the this method utilizes cleartext private key!

Parameters

privateKey	Private key
slot	Slot in the Token (see iDENprotect Keystore in iDENprotect iOS Developer Guide for information on slot usage)

Returns

Send next on success store private key

TOTP()

- (RACSignal *) TOTP

Get current TOTP number

This method requires user login with correct credentials.

Example return content:

{
deviceSerialNumber = "AMC0010201-000201";
otp = 50841463;
time = 1404169790000; //time in micros
}

Returns

NSDictionary of current TOTP number in sequence, based on current user's credentials

unEnroll()

- (RACSignal *) unEnroll

Reset enrollment (requires admin login)

Returns

Send next on success unenroll

Property Documentation

backendHost

- (NSString*) backendHost readwritenonatomicstrong

Backend iDENprotect server URL or IP address.

currentLocPlacemark

- (CLPlacemark*) currentLocPlacemark readwriteatomic

Current location placemark. See https://developer.apple.com/reference/corelocation/clplacemark

device

- (iDENtearCurrentDevice*) device readnonatomicassign

Information about current Token

deviceDisconnectSignal

- (RACSignal*) deviceDisconnectSignal readnonatomicassign

Notification about Token disconnect

deviceName

- (NSString *) deviceName readnonatomicassign

Name of connected device

errorSignal

- (RACSignal *) errorSignal readnonatomicassign

Any error signal(s) received by the iDENprotect SDK. Intended for analytics use.

mode

- (iDENtearMode) mode readnonatomicassign

Selected Token connection mode (read only): iDENtearModeBluetooth or iDENtearModeSoftToken

peripheralStatus

- (iDENtearDeviceStatus*) peripheralStatus readnonatomicassign

Reads Token status. This can be outdated, so it's recommended to first call getStatus method.