WebAuthn iOS SDK

Quickstart Guide

December 2022

Table of Contents

Overview

Add strong authentication with Passkeys to your native iOS application, while providing a native experience. This describes how to use our iOS SDK to register credentials and use them to authenticate for login and transaction approval scenarios. For transaction approval flows, transactions are signed per PSD2.0 SCA.

How it works

Our WebAuthn iOS SDK implements Apple's public-private key authentication for passkeys. It allows you to add FIDO2-based biometric authentication to your iOS app, while providing your users a native experience instead of a browser-based one.

With passkeys, credentials are securely stored by the device in the iCloud keychain. These credentials must be associated with your domain, so they can be shared between your mobile app and your website (if you have one). Our SDK also cryptographically binds the credentials to the device itself, which ensures that they can only be used by the device that registered it.

Requirements

The requirements for passkey authentication include:

- iOS 15.0+ (or iOS 16.0+ for passkeys)
- Xcode 13.0+ (or Xcode 14.0+ for passkeys)
- Device with registered biometrics (e.g., FaceID or TouchID)
- Device registered with the user's Apple ID
- Device with iCloud KeyChain turned on

Sample flows

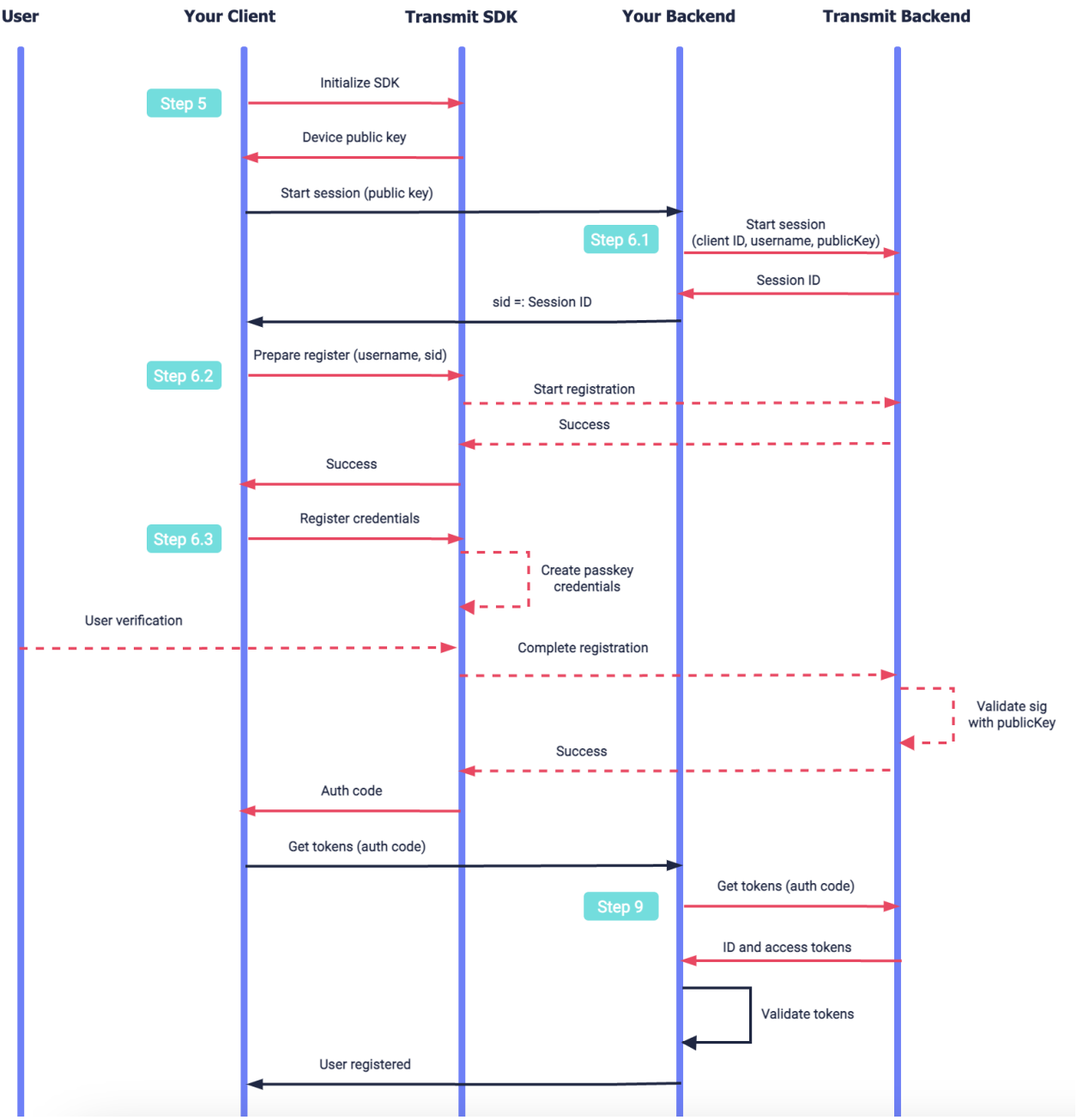
Here are samples of some flows that implement biometric authentication using our iOS SDK. There are many different ways to implement this and this is just an example. In the diagrams below, Transmit APIs are shown in pink along with the relevant integration step.

Registration

This flow represents a new user logging in from a device that supports passkeys. It assumes that you've collected and verified the username before initiating registration. The username should be a representation of the user in your system, such as their email, phone number, or another verifiable username known to the user (like account ID).

The SDK is initialized, and the device's public key is returned so it can be added to the session. An authorized session is then created to establish a secure context for registration. For a better UX, the SDK prepares for registration by fetching the challenge from the Transmit Server and storing it locally. Once the user is ready to register their device, the SDK creates credentials, which may require the user to verify (e.g., via TouchID).

Once completed, an authorization code is returned to the client to exchange for user tokens via their backend. Once tokens are validated, the user's device is registered.

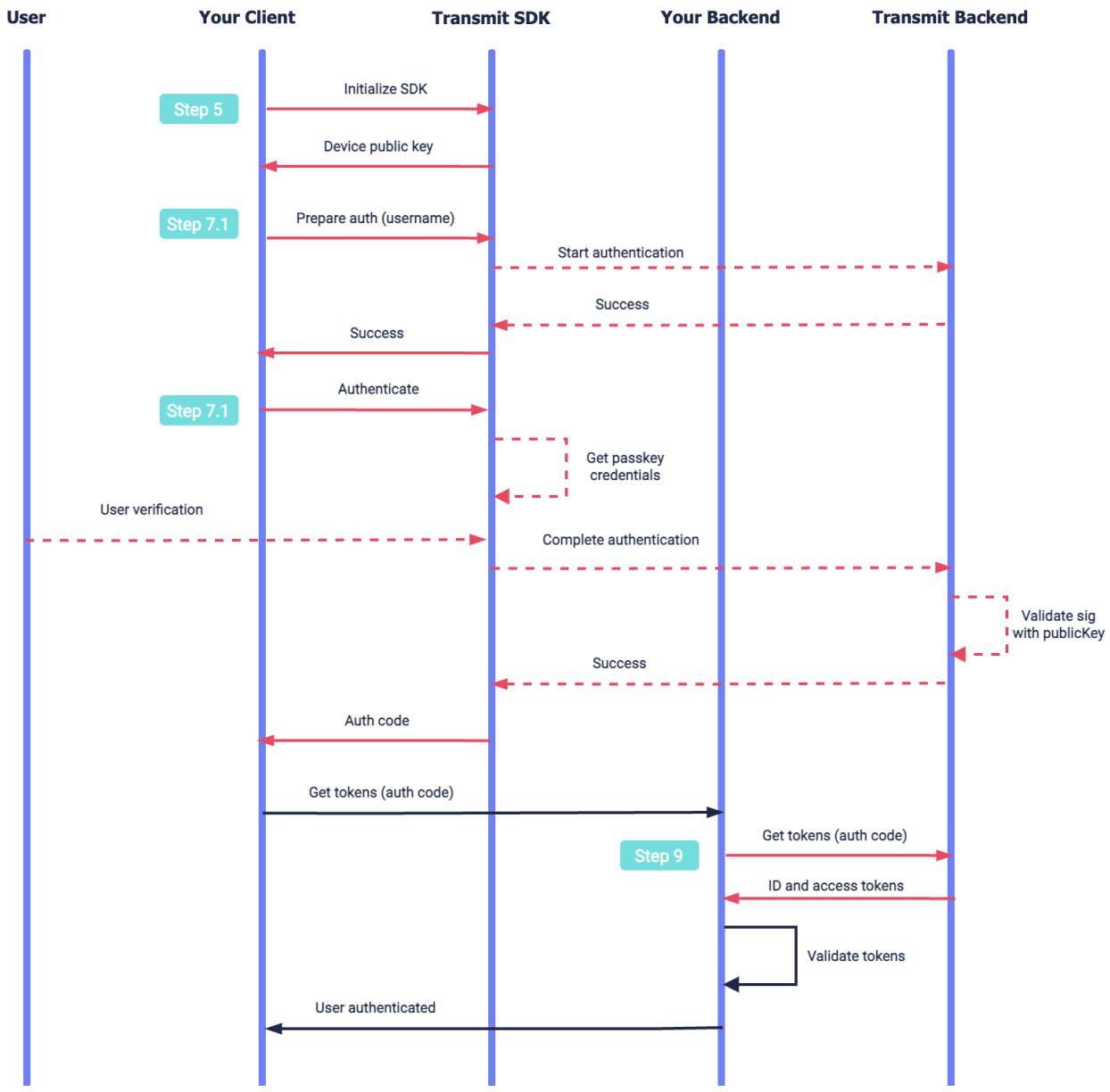


Authentication

This flow represents an existing user authenticating using a registered device. It assumes that you've collected the username before initiating authentication.

NOTE: This integration flow also applies to transaction signing, except that the authentication challenge is derived from the transaction details, and these details must be validated in the ID token.

The SDK is initialized. For a better UX, the SDK prepares for authentication by fetching the challenge from the Transmit Server and storing it locally. Once the user is ready to authenticate, the SDK retrieves credentials, which may require the user to verify (e.g., via TouchID). Once completed, an authorization code is returned to the client to exchange for user tokens. Your client backend exchanges the auth code for ID and access tokens. Once tokens are validated, the user is logged in.



Setup steps

This describes the required integration steps.

Step 1: Configure your app

To integrate with Transmit, you'll need to configure an application.
From the Applications page, create a new application or use an existing one.

From the application settings:

- For **Client type**, select **native**
- For **Redirect URI**, enter your website URL. This is a mandatory field, but it isn't used for this flow.
- Obtain your client ID and secret for API calls, which are auto-generated upon app creation.

Step 2: Configure auth method

From the Authentication page, configure the WebAuthn login method for your application (whose name is shown at the top of the page in the drop-down list):

- For **WebAuthn RP ID**, add your website's full domain (e.g., **`www.example.com`**). This is the domain that will be associated with your credentials in Step 3.

- For **WebAuthn RP Origin**, use **https://YOUR_DOMAIN**, where YOUR_DOMAIN is your website's domain that you configured as the RP ID (e.g., **`https://www.example.com`**). This is the origin that will be provided when requesting registration and authentication with passkey credentials.

Step 3: Associate your domain

In order to support passkeys, Apple requires having a domain associated with the relevant credential type (as noted here). This is done by adding the associated domain file to your website, and the appropriate entitlement in your app.

To use Apple's sample code project, follow these steps. The domain should be set to your domain, and match the WebAuthn RP ID configured in Step 2. To learn more about associated domains, click here.

Step 4: Install the SDK

The SDK can be integrated into your project manually. Download the TSWebAuthnSdk framework manually, open the new **`TSWebAuthnSdk`** folder, and drag the **`TSWebAuthnSdk.xcframework`** into the Project Frameworks directory of your application's Xcode project.

The **`TSWebAuthnSdk.xcframework`** is automatically added as a target dependency, linked framework and embedded framework in a copy files build phase which is all you need to build on the simulator and a device.

Step 5: Initialize the SDK

Configure the SDK by calling the **`initialize`** SDK method using a snippet like the one below, where YOUR_DOMAIN is the associated domain used in Step 3 and CLIENT_ID is your client ID (obtained in Step 1).

If successful, the SDK returns the device's public key in the response (in **`response.publicKey`**). This corresponds to the key that cryptographically binds the device to the credentials. If it doesn't already exist, the SDK will generate one. All subsequent authentication requests will be signed using the private key, and the public key will be used to validate the signature. If credentials aren't already registered for this device, you'll need to pass the public key to your app backend so it can be sent to Transmit in Step 6.1.

```
    let config = TSConfiguration()
    config.domain = "YOUR_DOMAIN"

    TSWebAuthnSDK.shared.initialize(serverUrl: "https://webauthn.identity.security/v1",
clientId: "CLIENT_ID", configuration: config) { response, error in
        if let error {
            print("SDK initialization failed \(String(describing: error.code))
\(String(describing: error.message))")
        } else {
            //Send the response.publicKey to the server
            print("SDK initialized")
        }
    }
```

Step 6: Register credentials

To implement a credential registration flow, you'll need to:

1. Create a secure context for registration
2. Warm up registration for a better UX
3. Perform the credential registration

1. Start an authorized session

From your backend, create an authorized authentication session for the user in order to provide the secure context required for WebAuthn registration. The user will be specified by their username.This should be a representation of the user in your system, such as their email, phone number, or another verifiable username known to the user (like account ID). It should be verified before initiating registration, for example, using Transmit authentication or verification APIs or your own verification process.

To do this, send the /auth-session/start-with-authorization request below from your backend. Use the client ID retrieved in Step 1, a verified username, a valid client access token to authorize the request (see Retrieve

access tokens), and the device's public key returned by the SDK upon initialization in Step 5. The API returns the session ID of the created session, which you'll need for subsequent API calls.

```
const basePath = 'v1';
const resp = await fetch(
  `https://webauthn.identity.security/${basePath}/auth-session/start-with-authorization`,
  {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: 'Bearer TOKEN'
    },
    body: JSON.stringify({
      client_id: 'CLIENT_ID',
      username: 'USERNAME',
      device_public_key: 'PUBLIC_KEY'
    })
  }
);

const data = await resp.json();
console.log(data);
```

2. Prepare registration

As soon as the username is known, call the **`prepareWebauthnRegistration`** SDK call as shown below. Pass the username, and the session ID returned upon session creation. This call will generate the challenge required for registration.

```
    TSWebAuthnSDK.shared.prepareWebauthnRegistration(username: "USERNAME", authSessionId:
"AUTH_SESSION_ID") { [weak self] success, error in

            if let error {
                //Handle error
            } else {
                //Complete the registration flow
            }
        }
```

3. Complete registration

When the user is ready to proceed to device registration (after calling **`prepareWebauthnRegistration`**), call the **`executeWebAuthnRegistration`** SDK method. This will prompt the user for biometrics.

If successful, it returns a callback that resolves to the authorization code (**`response.authCode`**), which you'll exchange for a token via your backend in Step 9.

```
    TSWebAuthnSDK.shared.executeWebauthnRegistration { [weak self] response, error in

        if let error {
           //Handle error
        } else {
           //User registered successfully. Use response.authCode for the token exchange
        }
    }
```

Step 7: Authenticate user

To implement an authentication flow, you'll need to:

1. Prepare authentication
2. Perform the authentication

1. Prepare authentication

When logging in users with registered devices, call the **`prepareWebauthnAuthentication`** SDK call as soon as the username is known. This will generate the challenge required for authentication. For example:

```
    TSWebAuthnSDK.shared.prepareWebauthnAuthentication(username: "USERNAME") { [weak
self] success, error in

        if let error {
            //Handle error
        } else {
            //Complete the authentication flow
        }

    }
```

2. Complete authentication

When the user is ready to proceed to authentication (after calling **`prepareWebauthnAuthentication`**), call the **`executeWebAuthnAuthentication`** SDK method using the snippet shown below. This will prompt the user for biometrics. If successful, this call returns a callback that resolves to the authorization code (**`response.authCode`**), which you'll exchange for a token via your backend in Step 9.

```
    TSWebAuthnSDK.shared.executeWebauthnAuthentication { [weak self] response, error in

        if let error {
            //Handle error
        } else {
           //User is authenticated. Use response.authCode for the token exchange
        }
    }
```

Step 8: Add transaction signing

To implement a transaction signing flow, you'll need to:

1. Warm up transaction signing
2. Complete transaction signing

1. Prepare transaction signing

In a transaction signing flow, call the **`prepareWebauthnSignTransaction`** SDK call when the user needs to approve a transaction. This will generate a challenge based on the transaction details.

The **`transactionData`** parameter contains the data that your customer should approve for a transaction signing flow. This should be the exact data that is displayed to the user, or be derived from it (e.g. a hash). It can contain up to 10 key-value pairs, and only alphanumeric characters, underscores, hyphens, and periods. The data will be returned in the ID token upon successful authentication.

For example:

```
    let transactionData = [ "payee": "Acme", "payment_method": "Acme card", "pay_amount"
: "200"]
    TSWebAuthnSDK.shared.prepareWebauthnSignTransaction(username: "USERNAME",
transactionData: transactionData) { [weak self] success, error in

            if let error {
               //Handle error
            } else {
              //Complete the transaction signing flow
            }
        }
```

2. Complete transaction signing

When the user is ready to proceed to authentication (after calling **`prepareWebauthnSignTransaction`**), call the **`executeWebAuthnSignTransaction`** SDK method. This will prompt the user for biometrics. If

successful, this call returns a callback that resolves to the authorization code (**`response.authCode`**), which you'll exchange for a token via your backend in Step 9.

```
    TSWebAuthnSDK.shared.executeWebauthnSignTransaction() { [weak self] response, error
in

            if let error {
               //Handle error
            } else {
             //User has authenticated successfully. You can use the response.authCode for
token exchange.
            }
        }
```

Step 9: Get user tokens

In response to each successful authentication or registration, an authorization code is returned to your client application (frontend). After sending this code to your backend, exchange it for an ID and access token by sending a /token request like the one below from your backend. Pass your client credentials, along with the authorization code returned by the SDK upon successfully completing authentication or registration (in **`response.authCode`**).

```
const resp = await fetch(
  `https://api.userid.security/v1/token`,
  {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      code: 'CODE',
      client_id: 'CLIENT_ID',
      client_secret: 'CLIENT_SECRET'
    })
  }
);

const data = await resp.json();
console.log(data);
```

Step 10: Validate tokens

A successful authentication returns ID and access tokens that should be validated as described here.

For transaction signing flows, the ID token will include an **`approval_data`** claim that corresponds to the **`transactionData`** passed in Step 8.1. This claim should be validated against the requested data, and what was presented to the user to approve.

Here's an example of a decoded ID token returned upon a successful transaction approval flow:

```
{
  "tid": "8AEksjdhfkuwaefOrJ2",
  "email": "user@email.com",
  "groups": [],
  "new_user": false,
  "amr": [
    "webauthn"
  ],
  "roles": [],
  "auth_time": 1671471638,
  "at_hash": "f0MaRhWO9pLgFvuJVDhJqw",
  "aud": "83474278.8kjsdfuwe2.transmit",
  "exp": 1671475240,
  "iat": 1671471640,
  "iss": "https://userid.security",
  "approval_data": {
     "payment_amount": "200",
     "payee": "Acme",
     "payment_method": "Acme card"
   }
}
```