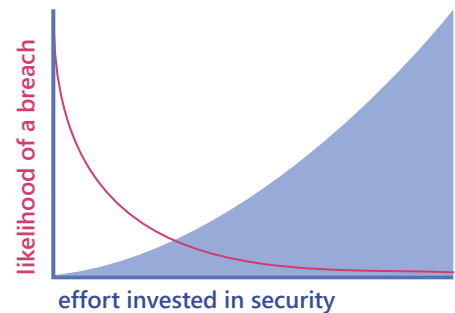


Security

Security is often overlooked when building apps. It is true that it is impossible to build software that is completely impenetrable—we've yet to invent a completely impenetrable lock (bank vaults do, after all, still get broken into). However, the probability of falling victim to a malicious attack or being exposed for a security vulnerability is inversely proportional to the effort you're willing to put in to protecting your application against any such eventuality. Although an ordinary padlock is pickable, it is still much harder to get past than a cabinet hook!

In this guide, you will learn about best practices for storing sensitive information, authentication, network security, and tools that will help you secure your app. This is not a preflight checklist—it is a catalogue of options, each of which will help further protect your app and users.



Storing Sensitive Info

Never store sensitive API keys in your app code. Anything included in your code could be accessed in plain text by anyone inspecting the app bundle. Tools like [react-native-dotenv](#) and [react-native-config](#) are great for adding environment-specific variables like API endpoints, but they should not be confused with server-side environment variables, which can often contain secrets and API keys.

If you must have an API key or a secret to access some resource from your app, the most secure way to handle this would be to build an orchestration layer between your app and the resource. This could be a serverless function (e.g. using AWS Lambda or Google Cloud Functions) which can forward the request with the required API key or secret. Secrets in server side code cannot be accessed by the API consumers the same way secrets in your app code can.

For persisted user data, choose the right type of storage based on its sensitivity. As your app is used, you'll often find the need to save data on the device, whether to support your app being used offline, cut down on network requests or save your user's access token between sessions so they wouldn't have to re-authenticate each time they use the app.

Persisted vs unpersisted — persisted data is written to the device's disk, which lets the data be read by your app across application launches without having to do another network request to fetch it or asking the user to re-enter it. But this also can make that data more vulnerable to being accessed by attackers. Unpersisted data is never written to disk—so there's no data to access!

Async Storage

Async Storage is a community-maintained module for React Native that provides an asynchronous, unencrypted, key-value store. Async Storage is not shared between apps: every app has its own sandbox environment and has no access to data from other apps.

DO USE ASYNC STORAGE WHEN...	DON'T USE ASYNC STORAGE FOR...
Persisting non-sensitive data across app runs	Token storage
Persisting Redux state	Secrets
Persisting GraphQL state	
Storing global app-wide variables	

Developer Notes

Web

Async Storage is the React Native equivalent of Local Storage from the web

Secure Storage

React Native does not come bundled with any way of storing sensitive data. However, there are pre-existing solutions for Android and iOS platforms.

iOS - Keychain Services

Keychain Services allows you to securely store small chunks of sensitive info for the user. This is an ideal place to store certificates, tokens, passwords, and any other sensitive information that doesn't belong in Async Storage.

Android - Secure Shared Preferences

Shared Preferences is the Android equivalent for a persistent key-value data store. **Data in Shared Preferences is not encrypted by default**, but Encrypted Shared Preferences wraps the Shared Preferences class for Android, and automatically encrypts keys and values.

Android - Keystore

The Android Keystore system lets you store cryptographic keys in a container to make it more difficult to extract from the device.

In order to use iOS Keychain services or Android Secure Shared Preferences, you can either write a bridge yourself or use a library which wraps them for you and provides a unified API at your own risk. Some libraries to consider:

- expo-secure-store
- react-native-encrypted-storage - uses Keychain on iOS and EncryptedSharedPreferences on Android.
- react-native-keychain
- react-native-sensitive-info - secure for iOS, but uses Android Shared Preferences for Android (which is not secure by default). There is however a branch that uses Android Keystore.
 - redux-persist-sensitive-storage - wraps react-native-sensitive-info for Redux.

Be mindful of unintentionally storing or exposing sensitive info. This could happen accidentally, for example saving sensitive form data in redux state and persisting the

whole state tree in Async Storage. Or sending user tokens and personal info to an application monitoring service such as Sentry or Crashlytics.

Authentication and Deep Linking

Mobile apps have a unique vulnerability that is non-existent in the web: **deep linking**. Deep linking is a way of sending data directly to a native application from an outside source. A deep link looks like `app://` where `app` is your app scheme and anything following the `//` could be used internally to handle the request.

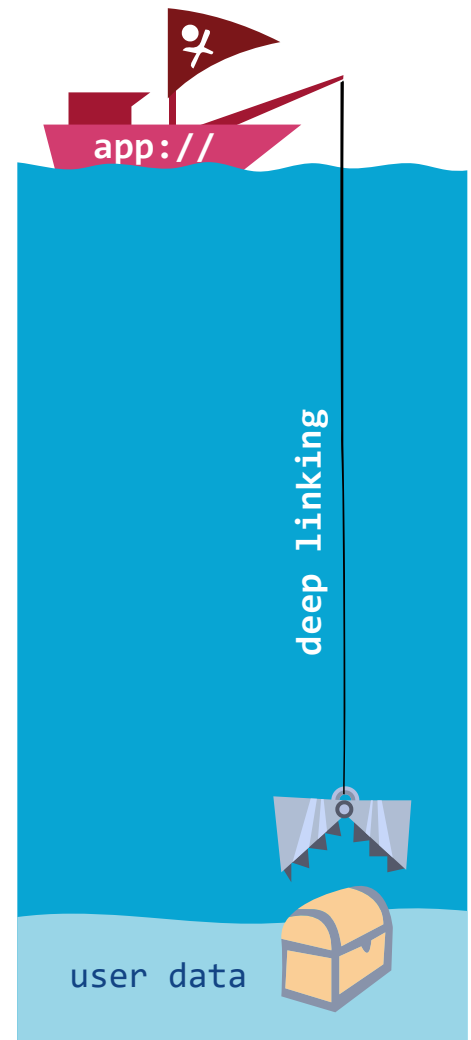
For example, if you were building an ecommerce app, you could use `app://products/1` to deep link to your app and open the product detail page for a product with id 1. You can think of these kind of like URLs on the web, but with one crucial distinction:

Deep links are not secure and you should never send any sensitive information in them.

The reason deep links are not secure is because there is no centralized method of registering URL schemes. As an application developer, you can use almost any url scheme you choose by configuring it in Xcode for iOS or adding an intent on Android.

There is nothing stopping a malicious application from hijacking your deep link by also registering to the same scheme and then obtaining access to the data your link contains. Sending something like `app://products/1` is not harmful, but sending tokens is a security concern.

When the operating system has two or more applications to choose from when opening a link, Android will show the user a Disambiguation dialog and ask them to choose which application to use to open the link. On iOS however, the operating system will make the



choice for you, so the user will be blissfully unaware. Apple has made steps to address this issue in later iOS versions (iOS 11) where they instituted a first-come-first-served principle, although this vulnerability could still be exploited in different ways which you can read more about [here](#). Using [universal links](#) will allow linking to content within your app securely in iOS.

OAuth2 and Redirects

The OAuth2 authentication protocol is incredibly popular nowadays, prided as the most complete and secure protocol around. The OpenID Connect protocol is also based on this. In OAuth2, the user is asked to authenticate via a third party. On successful completion, this third party redirects back to the requesting application with a verification code which can be exchanged for a JWT — a [JSON Web Token](#). JWT is an open standard for securely transmitting information between parties on the web.

On the web, this redirect step is secure, because URLs on the web are guaranteed to be unique. This is not true for apps because, as mentioned earlier, there is no centralized method of registering URL schemes! In order to address this security concern, an additional check must be added in the form of PKCE.

[PKCE](#), pronounced “Pixy” stands for Proof of Key Code Exchange, and is an extension to the OAuth 2 spec. This involves adding an additional layer of security which verifies that the authentication and token exchange requests come from the same client. PKCE uses the [SHA 256 Cryptographic Hash Algorithm](#). SHA 256 creates a unique “signature” for a text or file of any size, but it is:

- Always the same length regardless of the input file
- Guaranteed to always produce the same result for the same input
- One way (that is, you can’t reverse engineer it to reveal the original input)

Now you have two values:

- **code_verifier** - a large random string generated by the client
- **code_challenge** - the SHA 256 of the code_verifier

During the initial `/authorize` request, the client also sends the `code_challenge` for the `code_verifier` it keeps in memory. After the authorize request has returned correctly, the client also sends the `code_verifier` that was used to generate the `code_challenge`. The IDP will then calculate the `code_challenge`, see if it matches what was set on the very first `/authorize` request, and only send the access token if the values match.

This guarantees that only the application that triggered the initial authorization flow would be able to successfully exchange the verification code for a JWT. So even if a malicious application gets access to the verification code, it will be useless on its own. To see this in action, check out [this example](#).

A library to consider for native OAuth is [react-native-app-auth](#). `React-native-app-auth` is an SDK for communicating with OAuth2 providers. It wraps the native [AppAuth-iOS](#) and [AppAuth-Android](#) libraries and can support PKCE.

`React-native-app-auth` can support PKCE only if your Identity Provider supports it.

OAuth2 with PKCE



developer



user's device



Identity Provider



login screen

0. Registration



Registers client with Identity Provider (IDP)

The IDP will assign a **client_id** for the app (e.g. 123)

Developer registers **redirect_uri** with the IDP (e.g. myapp://auth)

1. Before auth



Generates **state** and **code_verifier**

These are both random strings that are kept in-memory on the device

2. App starts the auth request



GET /authorize



invalid credentials



/authorize URL parameters:

```
response_type = code
client_id = 123
redirect_uri = myapp://auth
scope = email
state = state
code_challenge = sha256( code_verifier )
code_challenge_method = sha256
```

3. IDP links back to the app with the auth code



DEEP LINK



Deep link looks something like this:

```
myapp://auth?code=code&state=state
```

A new **code** is generated by the IDP for each auth session

4. Token Exchange



POST /token



returns the JWT

/post data:

```
grant_type = authorization_code
code = code
redirect_uri = myapp://auth
client_id = 123
code_verifier = code_verifier
```

AUTH FLOW

PKCE STE

P

P
K
C
E

IDP checks that
 $\text{sha256}(\text{code_verifier}) = \text{code_challenge}$
before returning the JWT

Icons from Flaticon flaticon.com/authors/freepik

Network Security

Your APIs should always use SSL encryption. SSL encryption protects against the requested data being read in plain text between when it leaves the server and before it reaches the client. You'll know the endpoint is secure, because it starts with `https://` instead of `http://`.

SSL Pinning

Using https endpoints could still leave your data vulnerable to interception. With https, the client will only trust the server if it can provide a valid certificate that is signed by a trusted Certificate Authority that is pre-installed on the client. An attacker could take advantage of this by installing a malicious root CA certificate to the user's device, so the client would trust all certificates that are signed by the attacker. Thus, relying on certificates alone could still leave you vulnerable to a man-in-the-middle attack.


SSL pinning is a technique that can be used on the client side to avoid this attack. It works by embedding (or pinning) a list of trusted certificates to the client during development, so that only the requests signed with one of the trusted certificates will be accepted, and any self-signed certificates will not be.

When using SSL pinning, you should be mindful of certificate expiry. Certificates expire every 1-2 years and when one does, it'll need to be updated in the app as well as on the server. As soon as the certificate on the server has been updated, any apps with the old certificate embedded in them will cease to work.

Summary

There is no bulletproof way to handle security, but with conscious effort and diligence, it is possible to significantly reduce the likelihood of a security breach in your application. Invest in security proportional to the sensitivity of the data stored in your application, the number of users, and the damage a hacker could do when gaining access to their account. And remember: it's significantly harder to access information that was never requested in the first place.

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**