Securing An API In A Client Side Application

Written by Ng Wen Jie Dalton Prescott

" Security is not something that can be added to software as an afterthought; just as a shed made out of cardboard cannot be made secure by adding a padlock to the door, an insecure tool or application may require extensive redesign to secure it. " - Apple Secure Coding Guide

---

Disclaimer

From a security perspective, a tried and tested mechanism is often safer than rolling your own. Implementation errors is the most common way that a good security concept gets broken. This is best described by Apple, "Don't reinvent the wheel. When securing your software and its data, you should always take advantage of built-in security features rather than writing your own if at all possible." From a usability perspective oAuth is great and easy to use; which is a great benefit to your end users. However, to the best of my knowledge, oAuth too suffers from a similar issue to firebase. This paper is not written by a professional, and all information here should be taken with precaution and common sense. While I trust the information here to be correct to the best of my ability, it is recommended you double-check before you risk the security of an innocent user.

Abstract

As a mobile developer, it became evident to me that with since many mobile applications employ a client side application approach compared to the traditional server side application approach. With a server side application, nobody else will be able to see your code (under ideal circumstances), however in a client side app, anyone can easily reverse engineer your code leading to a slew of problems. Especially as our app was built to use firebase, this problem arose. In this paper, I discuss the risks and strategies to mitigate these effectively, specifically referencing the iOS platform and Firebase.

1.0 Introduction

Let us first look at the difference between a *client side based application model* vs a *server side based application model.*

Server side Application Model - i.e. a html Webpage
The client tells the server every action it does and the server responds with what to show. Client side does no handling of the ui, simply logs the users actions, maybe a session id, sends it to the server, server decides what to do, what screen to show the user.

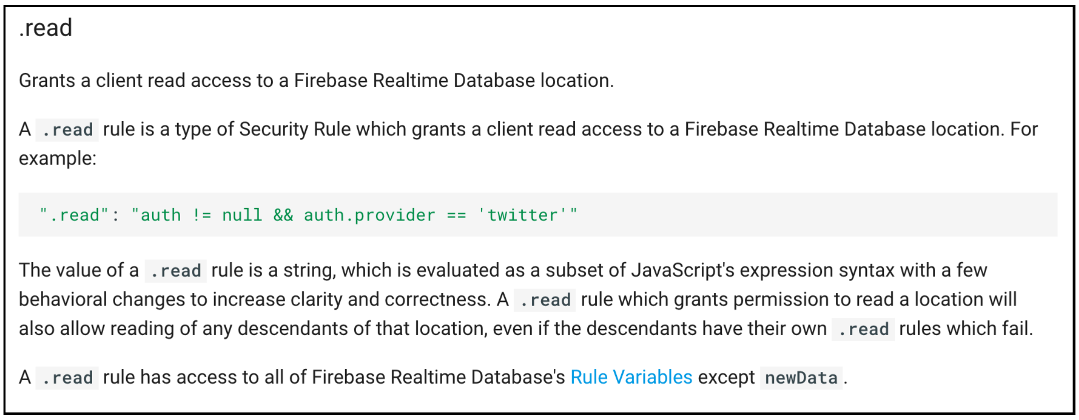
VS

Client side Application Model - i.e. an iPhone Application running on firebase
The client handles the ui and decides what screen to show the user, if this data is not stored in a cache, the client requests the data from a database, along with some form of authentication id to identify the user. There is only basic server side code, most of the processing is handled on the client itself.

As you can see, a client side application model is extremely dangerous as the user has access to most of the code. Now normally, I would suggest rethinking the approach. However many applications already run off services like Firebase and it would be unrealistic and expensive to port over to a different approach.

2.0 Firebase's Approach

Upon successful login, the application gets a `token` that can be used again for logging in when the app is closed and reopened using `auth()`. However in order to do this, a token needs to be saved on the client side. Examining the source code of Google's sample app for usage of firebase which implements authentication and session handling, the `token` is saved in the `localStorage` of the user's device.



.read

Grants a client read access to a Firebase Realtime Database location.

A `.read` rule is a type of Security Rule which grants a client read access to a Firebase Realtime Database location. For example:

```
".read": "auth != null && auth.provider == 'twitter'"
```

The value of a `.read` rule is a string, which is evaluated as a subset of JavaScript's expression syntax with a few behavioral changes to increase clarity and correctness. A `.read` rule which grants permission to read a location will also allow reading of any descendants of that location, even if the descendants have their own `.read` rules which fail.

A `.read` rule has access to all of Firebase Realtime Database's Rule Variables except `newData`.

Some of the security rules firebase offers. (Screenshot from Firebase Docs)

3.0 Risks

When examining my approach to utilise firebase I realised that this was a inadequate of working. Overall, my biggest concern was that that the secret key was stored in the client itself. This mean that a hacker could access the decompiled code/binary, revealing the secret key (*Which was merely obsfucated*). We didn't have much server-side logic so it was difficult to enforce it traditionally. A lack of server-side logic was due to some limitations on firebases' end, however we decided to find a way to work with these limitations. (*Firebase does have security rules but we found that even with them it wasn't enough.*)

In fact while we only focus on securing the secret api key in this paper, it is worthwhile to note that it was also discovered that firebase api stored session ids on the device without securely encrypting them, and merely obsfucating them. As we know, security through obscurity is not quantifiable so this surprised me. Considering the dangers of spoofing a device's session id, this could potentially allow a hacker to impersonate a user. Fortunately they do not contain the user's password or other sensitive data.

4.0 Android

Android too suffers from this problem. For instance, to compromise the Client Credentials for Twitter's Client on Android (*which uses oAuth*), an attacker can simply disassemble the classes.dex with Android dissembler tool, dexdump:

```
" dexdump -d classes.dex
```

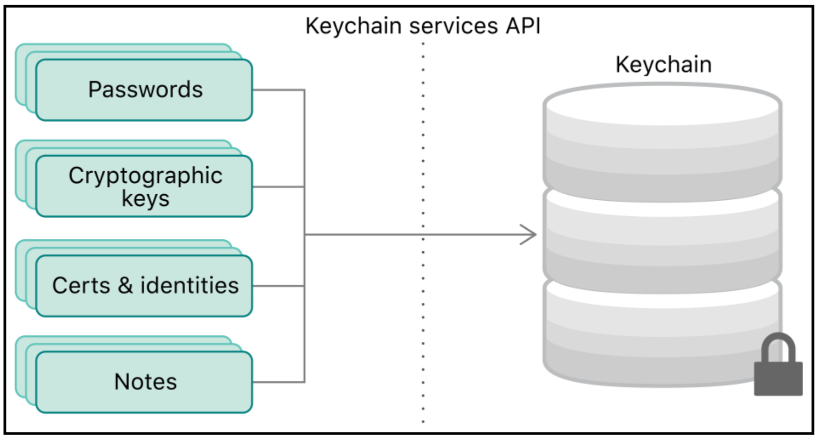
This shows how easy it is to obtain "*secret*" api keys. Even when obsfucated, the keys can easily be found if a hacker is determined.

In summary, a hacker can use the compromised Client Credentials to impersonate a valid Client and launch a spoofing attack, where the hacker can submit server requests on behalf of the victims and gain access to sensitive data.

5.0 *Approach*

5.1 iCloud Keychain

Firstly, we need a secure place to store the secret key. Apple has a wonderful place for us to store this secret api key called the *Apple iCloud Keychain Services.* It solves the problem by *"giving your app a mechanism to store small bits of user data in an encrypted database called a keychain."* iCloud Keychain also uses device based encryption which means even Apple can't decrypt it. However this is only a solution for iOS, as Apple doesn't make iCloud Keychain available outside its own operating systems.



Securing the users data in a keychain (Diagram from Apple)

Apples keychain uses AES-256 encryption, so it fits the bill for security.

However do note that if the application is installed on a device which has been jailbroken, it is apparently possible for a hacker to get your keys from the keychain. Check out : https://github.com/ptoomey3/Keychain-Dumper *for an example.*

5.2 Device-specific identifier

Secondly, we can require the client to bind both their Secret key with a device-specific identifier at run time.

```
var identifierForVendor: UUID?
```

Generates an alphanumeric string that uniquely identifies a device to the app’s vendor.

However this will change if the app is reinstalled, so we should generate a new secret key for any fresh installs of the app. We can also bind other information on such as the current timestamp which is known by the server and by the client without communication. You can force users to periodically change the token but do consider usability.

6.0 Mitigating the worst case scenario

Ultimately, there is no 100% secure way to hide a secret in a piece of code the hacker has control over. Obsfucation doesn't hurt, but don't let it give you a false sense of security. It definitely won't stop a determined hacker but it's enough to keep the average script buddy out. Focus on implementing a system to detect when that happens and recovering as fast as possible.
I highly recommend developing a sound strategy on mitigating the effects when your key is leaked rather than spending all your effort simply protecting it. Here are some of the mitigations I would suggest :

i. Generating a unique key for each user.
ii. Intermediary server to store sensitive data. (Be warned of mitm attacks.)
iii. Emergency backup code that checks with the server if it can download an override secret. In doing this, if the secrets are leaked, and your sysadmin notices, you are able to swiftly react by changing the secrets used for your app, and simultaneously shutting the rouge users. This override system should work across multiple updates so that you can fix the system without forcing users to update the application.

7.0 *Conclusion*

While I have tried to summarise all the important steps, the actual implementation is actually a lot more difficult. I would suggest using a tried and tested library like OAuth. However I definitely think the most important section is *6.0.* I hope this short paper has provided you with better clarity on securing your apps and outlining all the grim details of how insecure your app really is.

References

Firebase Documentation
https://firebase.google.com/docs/reference/security/database/

Firebase Example Source Code
https://github.com/firebase/firefeed/blob/master/www/js/firefeed.js

OAuth Extensions and Code Sample
http://oauth.net/code/

OAuth Vulnerabilities
https://dhavalkapil.com/blogs/Attacking-the-OAuth-Protocol/

Twitter API References
http://dev.twitter.com/pages/oauth_faq

Apple Keychain Services
https://developer.apple.com/documentation/security/keychain_services

Apple Security Interface Framework
https://developer.apple.com/documentation/security/certificate_key_and_trust_services

Unique Identifier on iOS
https://developer.apple.com/documentation/uikit/uidevice#//apple_ref/doc/uid/TP40006902-CH3-SW49

Keychain Vulnerabilities
http://resources.infosecinstitute.com/ios-application-security-part-12-dumping-keychain-data/#gref

Keychain Dumper
https://github.com/ptoomey3/Keychain-Dumper

Aes-256 Encryption
https://www.boxcryptor.com/en/encryption/