Real World Authentication

This is hard!

- Authenticating users in a real app is challenging problem
- If you make any mistakes you risk attackers accessing your users data, or making illegitimate requests, pretending to be one of your users!
- We've talked a bit about storing passwords securely, but there's a lot more to it than that!

Remembering authentication

- We've talked about how users shouldn't need to include their username and password with every request
- Instead, once authenticated, the server provides the user with a “token” which they can send with future requests to specify their identity
- The details of this can vary by implementation, and there's many ways to build an insecure system with this approach!
- Let's discuss some options

Approach 1: Server stores “active sessions”

- The server stores info about “active sessions” possibly in a data structure like `Map< Token, SessionData >`
- When a user authentiates (ie, gives us their username + password and we salt + hash it, and compared to our saved hash), we create a new entry in the map and send the corresponding key to the client
- On future requests, the server uses the client-provided token to look up the session data in its map
- The Map might contain a userID, or other info about the identity of the user involved in this session, as well as data like their “shopping cart”

Issues

- How should the server generate tokens? Is it OK to just increment an integer?
- What happens if a user's token gets “leaked” to an attacker?
- What if our backend is actually made up of many physical servers, maybe spread across the country/world?

Solution attempt: JSON web tokens

- To solve some of these issues, JWTs have been proposed
- A JWT consists of 3 pieces:
  - A Header: specifying crypto algorithms to be used below
  - The payload: a JSON object, fields might include the issuer(maybe a website) and the subject (the user ID, for example), expiration data, etc
  - A signature, either HMAC using either a secret key so only the issuer can verify it, or a private key so anyone with the public key can verify it
- Does this actually solve the issues?
- Does it let us do anything we couldn't support before?

JWT implementation

- Base64(header).Base64(payload).Base64(signature)
- New feature: Signer and verifier don’t have to be the same party

Can't someone else do it?

- Users already trust lots of folks to verify their identity: Google, Facebook, Apple, etc
- Can we let users log in with them and then have those providers “vouch for” our users?
- You've almost certainly used systems like this, which can be built with a variety of standards

Older version: OpenID

- The goal is for a user to prove their identity to a “relying service” which is might be the backend server for your app
- Instead of the RS storing/checking credentials, it redirects the user to an “Identity Provider” like Google or Apple
- The Identity Provider has the user provide their credentials (often username/password) and if they check out, it sends “proof of identity” to the RS
- The proof of identity might look something like: Sign(IP Private Key, "{username: username, provider: “google”, name: “Whatever”, email: "fake@gmail.com”})
- The RS can check the signature to make sure it really came from the IP

Details

- In order to ask an IP to authenticate users, you typically need to register your app with the IP
- The IP will provide you some sort of key or app ID
- When a client tries to log in, typically you redirect them to some URL provided by the IP (like google.com/auth) and include your app's ID/key along with a “callback” URL that the IP should send the user to with their token

Tweaked Use Case: Share to google drive

- Imagine you want to allow users to share images from their app to their Google Drive folders
- Users don't want any app to have permission to do whatever they want with their Google account
- So having users log into google and stealing the token google returns is not acceptable!
- For this use case, the user needs to prove their identity to google, and then tell google “this app can save stuff on my drive, but can't look at my emails”
- In general, a user wants to “delegate access” to their data which is stored by some service

OAuth

- OAuth is a standard for access delegation
- In contrast to OpenID, what my app gets is NOT “proof of identity of my user, as vouched for by google” but instead “a permission slip to touch some of the data that google is storing for $user”

OAuth Parties:

- “Resource Owner” — end user who “owns” the data, even though they're probably not storing it themselves
- “Client” — the application that wants access to the data, ironically this is often a backend server for some app
- “Resource Server” — server holding the data, so, Google or Github or Facebook who have APIs for accessing user data

Common Flow

- The first time the user wants to do “Google Drive stuff” the app sends a request to google to get an “access token” which is basically a “permission slip” to access the user's data stored by Google
- The user is sent to google to authenticate and approve access to the resources the app asked for
- Google sends an “authorization code” to the app
- The app makes a request for a “token” by providing the “authorization code” and an app-specific secret that it has registered with google
- Google sends back an “access token”
- From then on, the app can make API requests to google (ie store or retrieve files from google drive) by including the access token

Issues: “Authorization code interception attack”

- This attack occurs between getting an authorization code back (“the client authenticated and said you have permission to do stuff”) and getting an access token (sending the auth code + the app's secret)
- If a malicious app manages to get the authentication code before the real app requests a token, and it knows the secret, it can request the token first and hijack it
- If we have a backend server asking for a token, it can store its secret securely and no end-users or other apps can access it, so this isn't a concern

Unsecured “Clients”

- But what about an android app running on someone's phone? We have to expect them to be able to examine anything stored on the device.
- If a single attacker finds the app's secret key (which belongs to the app, not individual users), they could try to do these sorts of attacks on any user running that app
- “Serverless” web apps where the whole app is just JS served from somewhere have the same issue, there's no way to keep a secret from the user

Solution: “Proof Key for Code Exchange”

- Instead of using a client secret to prevent this, the app uses a crypto trick to prevent an attacker from using it's auth code
- When requesting the auth code, it sends hash(someRandomValue) with its request
- When it requests the token it sends someRandomValue as well. The resource server can check that its hash matches the hash sent previously, so it knows the same app made both requests

OpenID Connect

- Basically OAuth, but the “resource” that you're being granted access to is “info about the user”
- Android has built in libraries for using OpenID connect to authenticate users

Firebase

Firebase

- Firebase is a cloud offering from Google with a bunch of services you can use
- We're using it instead of AWS/Azure mostly because it's free tier doesn't require you to provide a credit card to start using it :)
- You organize stuff in “projects” which can have apps associated with them. Basically a project is all the associated data, and different apps can access it
- It supports authentication with existing google sign ons as well as other providers
- It also will let users sign in with an email + password which is simpler to work with since it doesn't require as much OAuth config

High level overview

- Once you have a firebase project, you can add an android app on the web console
- It'll give you a JSON file to add to your project with various IDs and keys
- There's a Firebase kotlin library to add via Gradle
- In that library, there's a Firebase singleton object that has all the functionality you want

Really basic auth

- `Firebase.auth.currentUser` — what you expect, null if no user
- `Firebase.auth.createUserWithEmailAndPassword` — you can attach a callback for what to do when it completes
- `Firebase.auth.signInWithEmailAndPassword`
- The current user has `email` and `uid` fields... UID really shouldn't be used by our app (we should request an idToken), but we can use it to keep things simple

More stuff next week

- We'll look at Firestore DB and some other stuff next week