Authentication and users

Michael Chang

Spring 2020

Plan for today

Authenticating to APIs

- API tokens/keys, headers

Managing users and passwords

- Storing passwords, maintaining logged-in status

Authentication providers

- E.g. Google (but others work similarly)

Quick security tips

Definitions

Authentication ("authn")

Verify the user's identity

Authorization ("authz")

Determine if the user is allowed to do the thing

Authenticating to an API

- **So far: our APIs are anonymous**
  - No identity => no authorization
- **Most APIs require some authentication**
  - Even if data is public: rate limiting, handling misuse
- **API key (aka "Bearer token", "OAuth token")**
  - Opaque string sent to API in each request
  - May contain info that API can interpret
  - May be completely random; look up in DB

Sending API keys

Query string

- `api.giphy.com/v1/gifs/search?api_key=...`
- `www.alphavantage.co/query?function=TIME_SERIES&apikey=...`

HTTP header

- Commonly: "Authorization" header
  - Yes, this is not the correct name
- Example

```
GET https://api.imgur.com/3/image/...
Authorization: Client-ID ...
```

- No one agrees on what word to use here
  - Bearer, Client-ID, token, ...

Handling user login

- **Leading advice: don't write it yourself**
  - Security challenges, very bad if you get it wrong
- **But that misses the point**
  - Not storing password != not storing personal data
  - Should understand the concepts behind libraries
    - Many libraries out there, some do it wrong too...
- **Before you handle real users and real data**
  - Do your research, understand the threat model
  - Read up on best practices for your use case
    - E.g. email addresses, user content, payment info

Example: login form

Takeaways

- Store salted, hashed passwords in database
- Use JWTs as API keys to set expiration

Salted, hashed passwords

Salting and hashing passwords

```
let hash = crypto.createHash("sha256");
let salt = crypto.randomBytes(8);
hash.update(salt);
hash.update(password);
let storedPassword = hash.digest("base64");
```

Salted, hashed passwords

Salting and hashing passwords

```
let hash = crypto.createHash("sha256");
let salt = crypto.randomBytes(8);
hash.update(salt);
hash.update(password);
let storedPassword = hash.digest("base64");
```

`let hash = crypto.createHash("sha256");`

- SHA256 is a "collision resistant" hash function
  - Security community believes you won't find hash collisions in any reasonable time (millions of years)
  - Can use longer hashes if you want, but security based on weakest link

Salted, hashed passwords

Salting and hashing passwords

```
let hash = crypto.createHash("sha256");
let salt = crypto.randomBytes(8);
hash.update(salt);
hash.update(password);
let storedPassword = hash.digest("base64");
```

`let salt = crypto.randomBytes(8);`

- Use different salt for different user
- Otherwise, same password across users => same hash

Salted, hashed passwords

Salting and hashing passwords

```
  let hash = crypto.createHash("sha256");
  let salt = crypto.randomBytes(8);
  hash.update(salt);
  hash.update(password);
  let storedPassword = hash.digest("base64");
```

```
let storedPassword = hash.digest("base64");
```

- Get a string from the salt (later) + password
- For same salt + password, string is always the same
  - Different if salt or password different
- Store this in DB, compare it when user enters password

API keys with JWTs

JSON web token

- String that encodes a JS object (JSON)
- Signed with a secret key
  - Can be "verified"; only someone with the key can create a JWT that will pass verification
- Can include expiration date and other properties

Warning

- Data is not encrypted
- Can read the data (payload) without the secret

API keys with JWTs

```
const jwt = require("jsonwebtoken");
const SECRET = "my secret string";

let obj = { email: ..., name: ... };
let token = jwt.sign(obj, SECRET, {
  expiresIn: "1h"
});

try {
  let obj = jwt.verify(token, SECRET);
} catch (e) { /* Problem verifying JWT */ }
```

Plan for today

Authenticating to APIs

API tokens/keys, headers

Managing users and passwords

Storing passwords, maintaining logged-in status

Authentication providers

E.g. Google (but others work similarly)

Quick security tips

Third-party authentication

Companies provide APIs and libraries to use their accounts

- E.g. "Sign in with Google", "Connect with Facebook"
- Based on OAuth and OpenID standards
  - But they all provide their own libraries and want you to use them

Advantages

- Don't have to store passwords in DB
- Don't have to handle email validation
- Provides verifiable tokens (possibly JWTs) you can use

Example: Google sign in

- Here's the documentation
  - Has steps for creating a client ID
- There's a bunch of boilerplate
- You can just use the `GoogleSignin` module in the lecture code

More advanced: OAuth

Interface for accessing APIs on behalf of users

- E.g. an app that can update your Google calendar

Overview (using Google as example)

- You ask user to sign into Google
- Google asks user to allow your app to act on their behalf
- Google returns an "authorization code" to your app
- Your app uses that code (along with the "client secret") to get an "access token"
- Your app sends requests to Google using that access token

Quick security tips

CORS (and the cors npm module)

- `api.use(cors({ origin: ... }));`
- Restrict access to your API to certain sites
- E.g. prevents attacker from tricking user into taking actions on your site

Cross-site scripting (XSS)

- Don't inject untrusted HTML/JS into your page
  - E.g. using innerHTML, or loading untrusted scripts

Cross-site request forgery (CSRF)

- E.g. POSTing malicious action directly to your site
- A bit less of a problem with REST APIs