11. Security & Real World HTTP Servers

Github Repository | Vimeo Video Recording

Topics to cover

- [x] 1. Storing passwords as hashes
- [x] 2. Encrypted cookies
- [x] 3. HTTP/HTTPS
- [x] 4. RESTful APIs

1. Storing passwords as hashes

We never want to store passwords as plain text. In a good, secure, system it should be difficult (if not impossible) for even the administrators or developers to read the passwords created by users.

Passwords should always be *hashed*. This is when we take a string and we pass it into a function that performs a transformation on it, returning a different string called *hash*.

This is intended for **one way processes**: a hashed value cannot be easily retrieved. We make hashes harder to crack by adding a *salt* to the original string:

`(plaintext + salt) => hashing algorithm => hash (more secure password)`

This helps to protect our data from certain attacks, such as the Rainbow Table attacks.

bcrypt

One of the easiest (and safest) ways to hash passwords yourself is to make use of a robust, battle-tested, existing package like bcrypt.js.

Have a look at what lives inside of bcrypt:

```
const bcrypt = require('bcryptjs');
console.log(bcrypt); // You'll see a variety of methods that this library offers us.
```

How do we use bcrypt? Let's look at the steps in the documentation:

```
const bcrypt = require('bcryptjs');  // Grab the library.
const salt = bcrypt.genSaltSync(10); // Generate a salt.
console.log('salt', salt); // Curious about what a salt looks like?

// Hash a plaintext password.
const plaintextPass = 'myp4ss';
const hash = bcrypt.hashSync(plaintextPass, salt);
console.log('hash', hash); // Curious about what a hash looks like?

const testPass1 = 'abc123'; // Shouldn't match our above plaintext
password.
const testPass2 = 'myp4ss'; // Should match our above plaintext password!

// Check if an entered password matches.
console.log('Does testPass1 match \'hash\'?',
bcrypt.compareSync(testPass1, hash)); // false
console.log('Does testPass2 match \'hash\'?',
bcrypt.compareSync(testPass2, hash)); // true
```

Curious which companies have made the mistake of not taking steps like these to protect your password? Visit Plain-Text Offenders, plenty have been caught! Luckily, many have reformed their approach to better serve their userbase.

2. Encrypted cookies

Storing cookies as plain text cookies can become a huge security concern. **Cookies can be manipulated by the user** by something as simple as accessing the developer tools and writing something different as the cookie value.

It is a better practice to store *encrypted* cookies. In **encryption**, similar to hashing, the string is transformed by a function. The key difference is that this is intended as a **two-way process: encrypted strings can be decrypted by the intended recipient**.

`plaintext => encryption algorithm => encrypted text (sent to browser)`

`encrypted text => decryption algorithm => plaintext (received on server)`

cookie-session

There is an npm package called cookie-session that can help us with this process, offering a syntax that is quite simple:

```
const express = require('express');
const cookieSession = require('cookie-session');

const app = express();
const PORT = 8080;

app.use(cookieSession({
  // Mandatory properties.
  name: 'session', // Name of the cookie (shows in the browser.)
  keys: ['secrets', 'can be', 'rotated'] // Used for encrypting values.

  // (Optional) consider adding additional options.
  maxAge: 24 * 60 * 60 * 1000 // Expire the identifying cookie / session
in 24 hours.
}));

app.get('/', (req, res) => {
  req.session.testValue = 'ABCD1234!'; // Set a value for this user's
session (does NOT appear in the browser.)
  res.end(req.session.testValue); // Access a value from this user's
session.
});

app.get('/destroy-session', (req, res) => {
  req.session = null; // Delete current session.
  res.redirect('/');
});

app.listen(PORT);
```

Note that in a case like keeping track of a user, any and all values associated with their sign-in and account are usually considered values that we want to protect from prying eyes. We would **not** assign these to *cookies in the browser*. Instead, we would assign these to a **server's session storage**, which is not visible to the browser.

A "session" in this sense only uses an encrypted and unique **session ID** cookie in the web browser to keep track of who we are representing. Beyond this, **all session key-value pairs are kept soley on the server-side**. This means the user will **not** be able to edit these values, but we can still use them to customize and inform a user's experience on our web application.

3. HTTP/HTTPS

HTTP is a plain-text protocol. All packets and files are sent rather plainly across the network. The concern here, is especially on untrusted networks (like a café's wi-fi), there may be third parties able to read your requests, and the server's responses. That might not matter if you're simply viewing a restaurant menu, but as soon as more sensitive data like your e-mails, sign-in pages, medical info, or private chat logs, you likely want a bit more privacy!

This is where HTTPS comes to the rescue! **HTTPS uses the Transport Layer Security (TLS) to encrypt communication between client and server**. This encryption works by using **asymmetric cryptography which uses a public key and private key**. * The public key is available to anyone who wants it and is used to encrypt the communication. * The private key is known only to the receiver and is used to decrypt the communication.

4. RESTful APIs

RESTful APIs are **APIs that follow the REST convention** for naming or organizing the routes we build for a web application or API.

Routes naming convention

In a RESTful API, **each piece of entity/data we want to access and modify is called a resource**. To allow users to access our resources we define a set of routes following the REST convention. There should be different verbs other than (`GET` / `POST`) for each unique route we define in CRUD

```
|-----------|--------|---------------------|----------------------------|
--------------|
| CREATE    | POST   | /resources          | Add New Resource           |
APPLY EFFECT |
| READ      | GET    | /resources          | Show All data of Resource  |
DISPLAY INFO |
| READ      | GET    | /resources/:id      | Display Resource           |
DISPLAY INFO |
| UPDATE    | PUT    | /resources/:id      | Replace Resource           |
APPLY EFFECT |
| UPDATE    | PATCH  | /resources/:id      | Partial Update to Resource |
APPLY EFFECT |
| DELETE    | DELETE | /resources/:id      | Delete the Resource        |
APPLY EFFECT |
```

These new verbs are really `POST` in special dressing. HTTP only *really* supports `GET` and `POST`, so under the hood we have these other methods spoofed for our convenience. What do these verbs mean?

- `GET` Request a Resource
- `POST` Create a Resource
- `PUT` Update / Replace all of a Resource
- `PATCH` Update / Replace Part of a Resource
- `DELETE` Remove a Resource

Using these verbs in combination with careful naming of your paths to intuitively communicate to other developers what each route is for. If you are careful and consistent with this convention, other developers will know exactly what each route is for without the need for a comment or additional context. In fact, you'll find many developers, once they know the names of resources, can guess most any routes that might exist in a particular project.

Forms method overriding in Express.js

The only methods / verbs actually supported by HTTP are `GET` and `POST`. If this is the case, how do we even get a form to submit with a `PUT`, `PATCH`, or `DELETE` method? This is called **method overriding.**

One package that can help is method-override. It offers a few ways for us to capture these conventional verbs in form submissions. It requires a little set-up:

```
const express = require('express');
const methodOverride = require('method-override');

const app = express();
const PORT = 8080;

app.use(express.urlencoded({extended: true}));
app.use(methodOverride('_method'));

app.put('/put-test', (req, res) => {
  res.end('PUT test received!');
});

app.listen(PORT);
```

By doing this simple configuration, now we can override the HTTP Method we want to use in our forms by using code like the following:

Extra resources

- Hashing vs. Encryption vs. Encoding vs. Obfuscation
- How Does Encryption Work?
- Client Session vs Server Session
- What is HTTPS?
- Asymmetric Cryptography
- RESTful Resource Naming
- Method Override Package

View on Compass | Leave Feedback