APDS7312 Attack prevention document

Attack prevention:

-- Session jacking

I created a session hijacking prevention system that uses JWT-based authentication along with several security measures to protect user accounts and sessions.

How it works:
When a user logs in, the system generates a JSON Web Token (JWT) that contains the user's name and account number, a unique session ID (jti) to prevent session fixation attacks, and the user's IP address and user agent string to detect unusual activity. The token is signed with a secret key and set to expire in 30 minutes, which limits the lifespan of any potentially compromised tokens. It's then sent back to the client as a secure, HTTP-only cookie, which helps prevent scripts from accessing it on the client side.

For authentication, all protected routes use a checkAuth middleware that verifies the JWT before granting access. The middleware reads the JWT from a secure cookie (or from the authorisation header as a fallback), decodes and verifies it using the secret key, and checks that the IP address and user agent match the ones stored in the token. If the token is missing, expired, or invalid, or if the IP or user agent doesn't match, the request is rejected with a 401 Unauthorised response. If everything checks out, the middleware attaches the decoded user information to req.user so other route handlers can use it.

This setup ensures that even if someone manages to steal a JWT, they can't use it from a different IP address or device. By combining short-lived JWTs, IP and user agent validation, and rate limiting, the system significantly reduces the chances of session hijacking or unauthorised account access.

-- MIM

The two precautions taken against Man in the middle attacks in this application reside within these two defences.

- HTTPS: Which secures the connection of the user portal
- Helmet – secures all http headers used during user and payment transactions

```
Https- Protection on the network level
import https from "https";
import fs from "fs";

const options = {
key: fs.readFileSync("keys/mongodb-key.pem"),
cert: fs.readFileSync("keys/mongodb-cert.pem")
};

https.createServer(options, app).listen(3000);
```

The above portion of code seeks to encrypt data as a means of preventing attacks from third parties. In addition to this the code also authenticates the server to ensure clients know they have not been lead to a false site connection. With these features in hand the TLS connection will break should any of the connection data be modified but outside parties.

```
Helmet- blocks attacks on an application and browser level
import helmet from "helmet";
app.use(helmet());
```

Helmet provides the network with strict security headers. This ensures that should an attacker manage to intercept the requests that they wont be able to exploit the browser. These attacks are blocked by features meant to protect the browser level such as :

Strict-Transport-Security: Which forces the browser to use HTTPS blocking SSL stripping and preventing downgrades to http.

Content-Security-Policy: Prevents problematic injection of script

X-Content-Type-Options: Prevents file-type spoofing

-- DDoS

For DDoS prevention we used a rate limiter from the express-rate-limit package which checks the amount of requests made by an IP on a window and if it exceeds a certain amount it will block that IP

```
const limiter = rateLimit({
    windowMs: 5 * 60 * 1000,
    max: 100,
    message: 'Too many attempts, please try again later'
});

const limiterSignup = rateLimit({
    windowMs: 15 * 60 * 1000,
    max: 10,
    message: 'Too many sign up attempts, please try again later'
});
```

Here we can see the rate limiting code, the one limit is a 100 in 5 minutes, and the other is 10 in 15 minutes

we also have this which makes it so that a user can not send a request that is greater than 10 kb in the json format

```
app.use(express.json({ limit: "10kb" }));
```

-- XSS Prevention

Purpose
I added a server-side XSS prevention system to sanitize and safely handle user-supplied strings (e.g., signup fields) so we don’t store or reflect HTML/script content.

How it works

* The `clean()` helper uses `sanitize-html` with no allowed tags or attributes to strip any scripts or HTML.
        * Sanitization happens before validation and database storage.
        * Regex checks still apply to ensure format validity.
        * Only safe values are stored and returned in responses.

        Quick test script

Create a file `xss-test.mjs` and run it with `node xss-test.mjs` while your server is active:

        ```js
        import fetch from 'node-fetch';

        const payloads = [
        { fullName: "Hacker <script>alert('xss')</script>", idNumber: "098765432", accountNumber: "112233445566", name: "hacker_101", password: "StrongPass123" },
        { fullName: "<img src=x onerror=alert('boom')>", idNumber: "123456789", accountNumber: "998877665544", name: "evil_user", password: "Pass1234" }
        ];

        for (const body of payloads) {
        const res = await fetch('https://localhost:3000/user/signup', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(body),
        });
        console.log(await res.json());
        }
        ```

Expected: The server either rejects invalid data or stores sanitized text (no tags/scripts).

Setup & run (quick)

1. Ensure Node 18+ is installed (recommended).
2. Init a small project and install node-fetch:

```bash
npm init -y
npm install node-fetch@3
```

3. Create `xss-test.mjs` with the script below and run the server (https local dev must be running):

```js
// xss-test.mjs
import fetch from 'node-fetch';
import https from 'https';

const agent = new https.Agent({ rejectUnauthorized: false });
const payloads = [
{ fullName: "Hacker <script>alert('xss')</script>", idNumber: "098765432", accountNumber: "112233445566", name: "hacker_101", password: "StrongPass123" },
{ fullName: "<img src=x onerror=alert('boom')>", idNumber: "123456789", accountNumber: "998877665544", name: "evil_user", password: "Pass1234" }
];

for (const body of payloads) {
const res = await fetch('https://localhost:3000/user/signup', {
method: 'POST',
headers: { 'Content-Type': 'application/json' },
body: JSON.stringify(body),
agent
});
console.log('STATUS', res.status);
const text = await res.text();
console.log('BODY', text);
}
```

4. Run the script:

```bash
node xss-test.mjs
```

Notes:

* We use `rejectUnauthorized: false` for local self-signed certs. Do **not** do this in production.
* Alternatively, use the `curl -k` examples in this README if you prefer not to run Node.

-- Clickjacking

Clickjacking is a type of attack where users are deceived into clicking hidden or disguised elements on a webpage, leading to unintended actions.
To mitigate this, the project implements Helmet’s frameguard middleware, which applies the X-Frame-Options: DENY header. This prevents the site from being embedded in external <iframe> elements, protecting users from malicious interface overlays.

```
import helmet from "helmet";
app.use(helmet.frameguard({ action: "deny" }));
```

SQL/NoSQL Injection occurs when attackers inject malicious queries through user input to gain unauthorized access or manipulate the database.
This project utilizes Mongoose ORM along with express-mongo-sanitize to defend against such attacks. User inputs are sanitized and validated before database interaction, removing potentially harmful operators like $ and ..

```
import mongoSanitize from "express-mongo-sanitize";
app.use(mongoSanitize());
```

-- SQL/NoSQL

SQL/NoSQL Injection occurs when attackers inject malicious queries through user input to gain unauthorized access or manipulate the database.
This project utilizes Mongoose ORM along with express-mongo-sanitize to defend against such attacks. User inputs are sanitized and validated before database interaction, removing potentially harmful operators like $ and ..

```
import mongoSanitize from "express-mongo-sanitize";
app.use(mongoSanitize());
```