MERN 13 - Security and Deployment

1. Adding encrypted password - why bcrypt is the preferred way
2. Adding rate limiter- reducing # of times a service can be accessed
   1. An often overlooked but critical aspect of securing web applications is mitigating Denial-of-Service (DoS) and brute-force attacks. Here, we'll explore the importance of rate limiting, a technique that restricts the number of requests a user can make to our service in a given timeframe, thus preventing overuse or abuse of our APIs and protecting against such attacks."
3. NoSql injection
   1. These attacks occur when untrusted data is used to query the database, and how validating and sanitizing inputs are critical practices for preventing them, ensuring the integrity and security of our database."
4. Adding required headers
   1. discuss the importance of HTTP headers in securing web applications
5. Deployment

## title:Hashing Password with Bcrypt

1. There are three general guidelines for security basis on which we design our backend
   1. Zero Trust Model: "Never Trust, Always Verify
      1. Assume that no one (neither inside nor outside the network) is trustworthy. This means always verifying the authenticity of users, services, and systems before granting access to resources.
      2. Implement strong authentication mechanisms, validate and sanitize all inputs, and regularly audit logs and activities.
   2. Principle of Least Privilege: "Minimal Access for Maximum Security”
      1. Each user, program, or system should have the least amount of privilege necessary to perform its function. This limits the potential damage in case of a security breach.
   3. Reduce Attack Surface: "Minimize Risk by Minimizing Exposure"
      1. The attack surface refers to the total number of points (like software, services, and ports) where an unauthorized user can try to enter data or extract data from the environment. Reducing the attack surface minimizes the potential entry points for attackers.

### Registering user problem

Check register user route

The provided route handles user registration but stores the password in plain text in the database, which is insecure. If an attacker gains access to the database, they can easily read all user passwords.

To store passwords securely, we can use a hashing algorithm like bcrypt.

### Using commonly used Passwords

1. Predictability of Hashes:
   1. For a given input and a known hashing algorithm, the generated hash will always be the same.
   2. Example:
   3. For the input "password", the hash might be asfdad123r2#$%.
   4. For the input "12345678", the hash might be asd12!@#%.
2. Brute Force Attacks:
   1. If an attacker wants to break into a system, the first step they might take is to perform a brute force attack using a dictionary of the most commonly used passwords and their corresponding hashes.
3. The Role of Salting:
   1. Modern password hashing techniques use a 'salt' - a random value added to the password before hashing.
   2. This ensures that even common passwords result in unique hashes. For example, "password" with different salts won't hash to asfdad123r2#$% every time.
4. Why Salting Matters:
   1. Without salting, common passwords remain vulnerable because their hashes can be precomputed and stored in lookup tables (rainbow tables).
   2. Even if a system uses salts, if the salt is known to the attacker, they can still perform targeted attacks, but the use of unique salts for each password significantly increases the complexity and time required for such attacks.
5. Generate a hash for password here - <https://www.md5hashgenerator.com/>
6. Decrypt the hash here - <https://10015.io/tools/md5-encrypt-decrypt#google_vignette>
7. Now try with a strong password. It wont be easy to decode
8. Emphasizing the problem
   1. A modern server can calculate the MD5 encryption at crazy speed
   2. If your users have passwords which are lowercase, alphanumeric, and 6 characters long, you can try every single possible password of that size in around 40 seconds.
   3. By spending a 1000 dollars or so you can crack millions of commonly used passwords
   4. Salts also do not help there

## Solution - BCRYPT

Bcrypt is a password hashing function that incorporates a salt to protect against rainbow table attacks. It’s designed to be computationally intensive to slow down brute-force attacks.

1. It slows down the process of generating the hash
2. Instead of cracking a password every 40 seconds, I’d be cracking them every 12 years or so.
3. <https://codahale.com/how-to-safely-store-a-password/>
4. Bcrypt is not an encryption algorithm like SHA256; rather, it is a password hashing function
5. Bcrypt is specifically designed for securing passwords. It turns a plain-text password into a hash, which is a fixed-size string of characters that uniquely represents the password.
6. Bcrypt automatically handles salt generation. A salt is a random value added to the password before hashing to ensure that the same password results in different hashes
7. Work Factor: One of the key features of bcrypt is its work factor, which is a measure of how slow the hashing process is. The ability to adjust the work factor is crucial to keeping up with increasing computational power and maintaining the security of the hashes over time.
8. Hashing vs. Encryption: Encryption is a reversible process (you encrypt data to later decrypt it), while the hashing function that bcrypt uses is one-way (once you hash data, you can't turn the hash back into the original data).This is useful for storing passwords because even if the hash is exposed, the original password cannot be easily recovered.
9. So the idea is that using bcrypt will make things a little slow so we use it only for the most sensitive data - password

Owasp - <https://owasp.org/www-project-top-ten/>

Search for owasp bcrypt and open the cheatsheet for storage - <https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html>

Come down to hashing vs envryptoin part

## Steps to Securely Store Passwords using Bcrypt

1. npm install bcrypt
2. Modify the registration route to hash passwords:

const bcrypt = require("bcrypt");

if (userExists) {

return res.send({

success: false,

message: "User already exists",

});

}

// Hash the password

const saltRounds = 10; // The higher the number, the more secure but slower the hashing process

const hashedPassword = await bcrypt.hash(req.body.password, saltRounds);

const newUser = new User({

...req.body,

password: hashedPassword, // Store the hashed password

});

await newUser.save();

1. A salt in cryptography is random data that is used as an additional input to a hashing function. In the context of password hashing, a salt is a unique value that is added to the password before it is hashed.
2. The main purpose of a salt is to ensure that the same password does not always produce the same hash
3. Else attackers will start creating password and commonly used salt mapping. By randomizing the salt, this tasks becomes impossible or impractical

## Example of Hash Verification during Login:

const login = async (req, res) => {

try {

const user = await User.findOne({ email: req.body.email });

const token = jwt.sign({ userId: user.\_id }, process.env.JWT\_SECRET, {

expiresIn: "1d",

});

console.log("token", token);

if (!user) {

return res.send({

success: false,

message: "User not found. Please register",

});

}

const isMatch = await bcrypt.compare(req.body.password, user.password);

if (!isMatch) {

return res.send({

success: false,

message: "Invalid Credentials",

});

}

res.send({

success: true,

message: "Login Successful",

user: user,

data: token,

});

} catch (err) {

console.log(err);

return res.status(500).send({

success: false,

message: "Internal Server Error",

});

}

};

### Anatomy of the generated hash

async function hashPassword(password) {

console.time("time taken");

const salt = await bcrypt.genSalt(12);

console.log("salt", salt);

const hashedPassword = await bcrypt.hash(password, salt);

console.log("hashedPassword", hashedPassword);

console.timeEnd("time taken");

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

return hashedPassword;

}

const login = async (req, res) => {

try {

const user = await User.findOne({ email: req.body.email });

const token = jwt.sign({ userId: user.\_id }, process.env.JWT\_SECRET, {

expiresIn: "1d",

});

console.log("token", token);

if (!user) {

return res.send({

success: false,

message: "User not found. Please register",

});

}

const isMatch = await bcrypt.compare(req.body.password, user.password);

if (!isMatch) {

return res.send({

success: false,

message: "Invalid Credentials",

});

}

const password = "Ayush@123";

hashPassword(password);

res.send({

success: true,

message: "Login Successful",

user: user,

data: token,

});

} catch (err) {

console.log(err);

return res.status(500).send({

success: false,

message: "Internal Server Error",

});

}

};

Stored Hash Contains the Salt: When bcrypt hashes a password, it incorporates the salt into the resulting hash. This means the hash stored in your database contains both the hashed password and the salt used. The format typically includes the salt, the cost factor (or salt rounds), and the hashed password.

When a user attempts to log in, the system:

1. Retrieves the stored bcrypt hash.
2. Extracts the version, cost factor, and salt from the stored hash.
3. Hashes the provided password with the extracted salt and cost factor.
4. Compares the newly generated hash with the stored hash.

## title:Rate Limiter

1. Denial of Service
   1. DoS, is a type of cyber attack where the attacker aims to make a website or online service unavailable to its users
   2. they do this by overwhelming the service with an excessive amount of traffic or sending information that triggers a crash.
2. This is where rate limiter comes in
3. A rate limiter is a mechanism used to control the number of requests a client can make to a server within a specific time frame. This helps to prevent abuse, ensure fair usage, and protect the server from being overwhelmed by too many requests in a short period

### Implementing a Rate Limiter in an Express Application

Install express-rate-limit:

npm install express-rate-limit

This middleware helps protect your server from being overwhelmed by too many requests from the same IP address within a specified time window.

server.js

const express = require("express");

require("dotenv").config(); // To access the environment variables

const rateLimit = require("express-rate-limit");

const connectDB = require("./config/dbconfig");

const userRouter = require("./routes/userRoute");

const movieRouter = require("./routes/movieRoute");

const theatreRouter = require("./routes/theatreRoute");

const showRouter = require("./routes/showRoute");

const bookingRouter = require("./routes/bookingRoute");

const app = express();

app.use("/api/bookings/verify", express.raw({ type: "application/json" }));

app.use(express.json());

connectDB();

// Rate limiter middleware

const apiLimiter = rateLimit({

windowMs: 15 \* 60 \* 1000, // 15 minutes

max: 100, // Limit each IP to 100 requests per windowMs

message: "Too many requests from this IP, please try again after 15 minutes",

});

// Apply rate limiter to all API routes

app.use("/api/", apiLimiter);

/\*\* Routes \*/

app.use("/api/users", userRouter);

app.use("/api/movies", movieRouter);

app.use("/api/theatres", theatreRouter);

app.use("/api/shows", showRouter);

app.use("/api/bookings", bookingRouter);

app.use((req, res) => {

res.status(404).json({ message: "Route not found" });

});

app.listen(8082, () => {

console.log("Server is running on port 8082");

});

The apiLimiter variable configures the rate limiter with a time window of 15 minutes and a maximum of 100 requests per IP within that window. If the limit is exceeded, a message is sent to the client.

There is something also called as DDOS

1. A Distributed Denial of Service (DDoS) attack is a specific type of Denial of Service (DoS) attack.
2. specialized DDoS protection services from cloud providers or third-party services like Cloudflare, Akamai, or AWS Shield. These services can detect and mitigate large-scale DDoS attacks.
3. configure firewalls and routers to recognize and filter out malicious traffic.

## Helmet

1. Try loading movies in home page
2. Open network tab and check the response headers
3. x-powered -by: express
4. Now an attacker knows that our backend is on express.
5. Npm package - helmet
6. Npm i helmet
7. Helmet applies a set of security headers to your application, reducing the risk of several well-known web vulnerabilities.
8. Cross-Site Scripting (XSS) Protection: Helmet sets the X-XSS-Protection header to enable the Cross-Site Scripting (XSS) filter built into most web browsers.
9. It sets the X-Frame-Options header to prevent your content from being embedded into other sites.
10. Helmet helps in implementing Content Security Policy, a powerful tool to mitigate XSS and other injection attacks
    1. By restricting where resources can be loaded from, CSP helps protect against XSS attacks, where attackers might try to inject malicious scripts into web content.

const app = express();

app.use(helmet());

app.disable("x-powered-by"); // it will remove the x-powered-by header from the response

Little more controlled helmet configuration

app.use(

helmet.contentSecurityPolicy({

directives: {

defaultSrc: ["'self'"],

scriptSrc: ["'self'"],

styleSrc: ["'self'", "https://fonts.googleapis.com"],

imgSrc: ["'self'", "data:"],

connectSrc: ["'self'"],

fontSrc: ["'self'", "https://fonts.gstatic.com"],

objectSrc: ["'none'"],

},

})

);

**Cross-Site Scripting (XSS)**: A security vulnerability where attackers inject malicious scripts into web pages viewed by others.

**Helmet**: Middleware that helps set security-related HTTP headers to prevent various types of attacks, including XSS.

**Content Security Policy (CSP)**: Helps prevent XSS by controlling the sources of content.

**X-XSS-Protection**: Helps prevent reflected XSS attacks by enabling browser's XSS filter.

**CORS**: Helps control resource sharing and mitigate certain XSS attacks.

## title:SQL Injection

SQL Injection is a code injection technique that exploits a security vulnerability in an application's software by manipulating the SQL queries made to the database. It typically occurs when user input is improperly sanitized and then included in SQL queries, allowing attackers to execute arbitrary SQL commands.

Consider a login system where user input directly constructs a SQL query:

Imagine you have a form on your website where users can log in by entering their username and password. In an SQL injection attack, instead of typing a regular username, an attacker types in a sneaky piece of SQL code

SQL injection can be prevented by properly validating and sanitizing user inputs

<https://stackoverflow.com/questions/24843689/whats-the-meaning-of-admin-or-1-1>

<https://portswigger.net/web-security/nosql-injection#:~:text=NoSQL%20operator%20injection,-NoSQL%20databases%20often&text=Examples%20of%20MongoDB%20query%20operators,values%20specified%20in%20an%20array>

### Protecting Against SQL Injection Using express-mongo-sanitize

npm install express-mongo-sanitize

const mongoSanitize = require("express-mongo-sanitize");

// Sanitize user input to prevent MongoDB Operator Injection

app.use(mongoSanitize());

The express-mongo-sanitize package works by checking for keys in objects that begin with $ or contain .. These characters are used in MongoDB queries for operators and thus could be exploited for injection attacks. The middleware strips these characters from the input, effectively sanitizing it.

By adding app.use(mongoSanitize());, the server ensures that any malicious input attempting to manipulate MongoDB queries is sanitized, providing a layer of protection against injection attacks.

## title:Deploying The app

We will use Render to Deploy our App.

<https://dashboard.render.com/>

Step 1: Prepare Your MERN App

1. Set Up Your MERN App: Make sure your MERN app is working perfectly on your local machine. Your app should be divided into client (React) and server (Node.js, Express) directories.
2. Create a Production Build: Navigate to your React client directory and run:
3. npm run build
4. This will create a production build of your React app in a build folder.

Step 2: Set Up a Git Repository

1. Initialize a Git Repository: If you haven’t already, initialize a git repository in the root directory of your project:
2. git init
3. Commit Your Code: Add your files and make an initial commit:
4. git add .
5. git commit -m "Initial commit"
6. Push to GitHub: Create a repository on GitHub and push your code to it:
7. git remote add origin <your-github-repo-url>
8. git push -u origin main

Step 3: Set Up Render Account and New Web Service

1. Sign Up or Log In to Render: Go to Render and sign up or log in to your account.
2. Create a New Web Service: Click on the “New” button and select “Web Service.”

Step 4: Connect to Your GitHub Repository

1. Authorize GitHub: Connect Render to your GitHub account if you haven't done so already.
2. Select Repository: Choose the repository that contains your MERN app

Step 5: Configure the Service

1. Basic Settings:
2. Name: Choose a name for your service.
3. Region: Select a region close to your users.
4. Build Command: Add the build commands to set up your project. If your server code is in a folder named server, you might use:
5. cd client && npm install && npm run build && cd ../server && npm install
6. Start Command: Specify the command to start your server. For example:
7. cd server && npm start

Step 6: Environment Variables

1. Add Environment Variables: Click on “Add Environment Variable” to add any necessary environment variables such as MONGODB\_URI, JWT\_SECRET, etc.

Step 7: Deploy

1. Deploy Your App: Click “Create Web Service” to deploy your app. Render will pull the code from your GitHub repository, install dependencies, build the app, and start the server.

Step 8: Make changes to server.js to serve the build

const path = require("path");

const app = express();

const clientBuildPath = path.join(\_\_dirname, "../client/build");

console.log(clientBuildPath);

app.use(express.static(clientBuildPath));

app.get("\*", (req, res) => {

res.sendFile(path.join(clientBuildPath, "index.html"));

});

app.use(express.static(clientBuildPath)) tells the Express application to serve static files from the clientBuildPath directory.

This means any requests for static assets like JavaScript files, CSS files, images, etc., will be served from the client/build directory.

app.get("\*", (req, res) => { res.sendFile(path.join(clientBuildPath, "index.html")); }); sets up a catch-all route.

The \* wildcard matches any route that hasn’t been matched by previous route handlers.

This route handler serves the index.html file for any request that doesn't match a static file or an API route.

This is particularly important for single-page applications (SPAs) like React, where routing is handled client-side. Serving the index.html file ensures that the React app can take over routing from there.

Step 8: remove proxy from client / package.json

Update cors in server.js

const cors = require("cors");

const app = express();

// Enable CORS

// app.use(

// cors({

// origin: ["http://localhost:3000", "https://your-production-url.com"]

// methods: ["GET", "POST", "PUT", "DELETE"],

// allowedHeaders: ["Content-Type", "Authorization"],

// credentials: true, // Allow credentials such as cookies, authorization headers

// })

// );

app.use(

cors({

origin: "\*", // Allow only your frontend origin

methods: ["GET", "POST", "PUT", "DELETE"],

allowedHeaders: ["Content-Type", "Authorization"],

})

);

If getting csp:blocked error

// app.use(

// helmet({

// contentSecurityPolicy: {

// directives: {

// defaultSrc: ["'self'"],

// scriptSrc: [

// "'self'",

// "'unsafe-inline'",

// "'unsafe-eval'",

// "https://your-production-url.com", // Replace with your actual production URL

// ],

// styleSrc: ["'self'", "'unsafe-inline'", "https://fonts.googleapis.com"],

// imgSrc: ["'self'", "data:", "https://your-production-url.com"], // Replace with your actual production URL

// connectSrc: ["'self'", "https://your-production-url.com"], // Your API domain

// fontSrc: ["'self'", "https://fonts.gstatic.com"],

// objectSrc: ["'none'"],

// upgradeInsecureRequests: [],

// },

// },

// })

// );