Class 12 - Security and Deployment

Agenda

1. Adding encrypted password - why bcrypt is the preferred way
2. Adding rate limiter- reducing # of times a service can be accessed
3. NoSql injection
4. Adding required headers
5. Deployment

Class starts

1. There are three general guidelines for security basis on which we design our backend
    a. Zero Trust Model: "Never Trust, Always Verify
        i. 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.
        ii. Implement strong authentication mechanisms, validate and sanitize all inputs, and regularly audit logs and activities.
    b. Principle of Least Privilege: "Minimal Access for Maximum Security”
        i. 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.

c. Reduce Attack Surface: "Minimize Risk by Minimizing Exposure"
    i. 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. Reduce the attack surface minimizes the potential entry points for attackers.

Password Encryption]

1. We have seen hashing techniques in class where we pass some plain text, use a key and generate a hashed string
2. Now lets understand why it is suggested to not use commonly used passwords like password, 12345678
3. For a given input, if we know the algo, the generated hash will be always same
    a. Let say for input password -> asfdad123r2#$% (hash)
    b. For 1234567 -> asd12!@#% (hash)
    c. Now if you want to break into my system , what is the first step that you would like to try
    d. Do a brute force attack with a dictionary of most used passwords and their hashes

    e. Want to briefly callout the The Role of Salting: Modern password hashing techniques use a 'salt' - a random value added to the password before hashing. This makes the hash of common passwords unique (e.g., 'password' with a salt won't hash to 'asfdad123r2#$%' every time). However, if you're using a system that doesn't salt passwords, or if the attacker knows the salt, common passwords remain vulnerable.
    f. Generate a hash for password here - https://www.md5hashgenerator.com/
    g. Decrypt the hash here - https://10015.io/tools/md5-encrypt-decrypt#google_vignette
    h. Now try with a strong password . It wont be easy to decrypt
4. Emphasizing the problem
    a. A modern server can calculate the MD5 encryption at crazy speed
    b. 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.
    c. By spending a 1000 dollars or so you can crack millions of commonly used passwords
    d. Salts also do not help there
5. Solution

a. Bcrypt
b. It slows down the process of generating the hash
c. Instead of cracking a password every 40 seconds, I'd be cracking them every 12 years or so.
d. https://codahale.com/how-to-safely-store-a-password/

Bcrypt

1. Bcrypt is not an encryption algorithm like SHA256; rather, it is a password hashing function
2. 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.
3. 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
4. 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.
5. Hashing vs. Encryption: Encryption is a reversible process (you encrypt data to later decrypt it), while hashing is one-way (once

you hash data, you can't turn the hash back into the original data).

6. 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

Code

1. Create a file bcryptTest.js
2. Npm i bcrypt

```
const bcrypt = require('bcrypt');

const password = 'Ayush@123';

async function hashPassword(password){
 const salt = await bcrypt.genSalt(10);
 const hashedPassword = await bcrypt.hash(password, salt);
 console.log("hashedPassword", hashedPassword);
 return hashedPassword;
}

hashPassword(password);
```

3. What is salt again
    a. 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.
    b. The main purpose of a salt is to ensure that the same password does not always produce the same hash
4. Salt rounds refer to the cost factor or the number of iterations the hashing algorithm uses. It's a measure of the computational work required to hash the password.
    a. The higher the number of salt rounds, the longer it takes to compute the hash

Anatomy of the generated hash

1. Let us print the salt and time taken

```
const bcrypt = require('bcrypt');

const password = 'Ayush@123';

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;
}

hashPassword(password);
```

2. 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.

Comparing passwords

```
const bcrypt = require('bcrypt');

const password = 'Ayush@123';

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("***************")

 const isMatching = await bcrypt.compare(password,
hashedPassword);
 console.log(isMatching);
}
```

```
hashPassword(password);
```

1. The stored hash has the cost factor, the salt
2. It will hash the given input and then compare the two values

Updating the user model - hash user password

1. In the pre hook for the userSchema

```
userSchema.pre("save", async function(next){
   // check if password and confirmPassword are same
   console.log("cf", this.confirmPassword)
   console.log("password", this.password)
   if(this.password !== this.confirmPassword){
     next(new Error("Password and confirm password should
be same"))
   }
   this.confirmPassword = undefined;
   // use bcrypt to hash password
   const hashedPassword = await bcrypt.hash(this.password,
12);
   this.password = hashedPassword;
   console.log("updated",this.password, hashedPassword)
   // checking for roles
   if(this.role){
     const isValid = validRoles.includes(this.role);
     if(!isValid){
       next(new Error("User can either be admin, user or
seller"))
```

```
    }else {
      next()
    }

  }else {
    this.role = "user";
    next()
  }

})
```

2. We can comment the validate method in the schema for confirmPassword

```
confirmPassword:{
    type: String,
    // validate: {
    //   validator: function(){
    //     return this.password === this.confirmPassword
    //   },
    //   message: "Password and confirm password should be same"
    // }
  },
```

Rate limiter

1. Denial of Service

    a. DoS, is a type of cyber attack where the attacker aims to make a website or online service unavailable to its users
    b. they do this by overwhelming the service with an excessive amount of traffic or sending information that triggers a crash. Imagine it like a crowd blocking the entrance to a shop, preventing genuine customers from entering
2. This is where rate limiter comes in
3. We have a npm package called express rate limit - https://www.npmjs.com/package/express-rate-limit
4. In app.js

```
const rateLimit = require('express-rate-limit');
```

```
app.use(cors({ origin: "http://localhost:5173",
credentials: true }));

const limiter = rateLimit({
 windowMs: 15 * 60 * 1000, // 15 minutes
 limit: 100, // Limit each IP to 100 requests per `window`
(here, per 15 minutes).
 standardHeaders: 'draft-7', // draft-6: `RateLimit-*`
headers; draft-7: combined `RateLimit` header
 legacyHeaders: false, // Disable the `X-RateLimit-*`
headers.
 // store: ... , // Use an external store for consistency
across multiple server instances.
})
// app.use(cors())
app.use(limiter);
```

5. There is something also called as DDOS
    a. A Distributed Denial of Service (DDoS) attack is a specific type of Denial of Service (DoS) attack.
    b. A Distributed Denial of Service (DDoS) : In a DDoS attack, the attacker flood a target, like a website or online service, with overwhelming traffic
    c. 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.
    d. configure firewalls and routers to recognize and filter out malicious traffic.

SQL injection

1. SQL injection is a type of cyber attack where an attacker manipulates a standard SQL query to gain unauthorized access to or manipulate your database.
2. SQL injection can be prevented by properly validating and sanitizing user inputs
3. https://stackoverflow.com/questions/24843689/whats-the-meaning-of-admin-or-1-1
4. 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

5. Npm package - https://www.npmjs.com/package/express-mongo-sanitize
6. `npm i express-mongo-sanitize`

```
const mongoSanitize = require('express-mongo-sanitize');
app.use(limiter);
app.use(mongoSanitize());
```

Response Headers

1. Go to localhost:3000/api/products on browser to get the products
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. `app.use(helmet());`
8. Helmet applies a set of security headers to your application, reducing the risk of several well-known web vulnerabilities.
9. 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.
10. It sets the X-Frame-Options header to prevent your content from being embedded into other sites.

11. Helmet helps in implementing Content Security Policy, a powerful tool to mitigate XSS and other injection attacks
    a. 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.

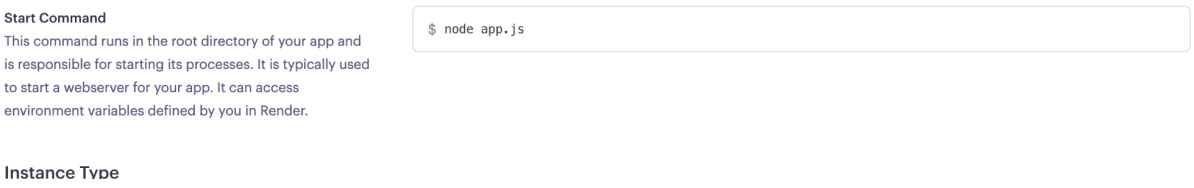
CSRF

1. The attacker can make requests as if they are you, potentially changing your settings, making purchases, or even sending messages.
2. Mitigation measures
    a. he CSRF token approach is a widely used method to prevent Cross-Site Request Forgery (CSRF) attacks.
    b. A CSRF token is a unique, secret, and unpredictable value generated by the server for each user session.
    c. The server sends this token to the client, typically as part of a form or in a header response.
    d. When the client submits a form or makes a request that should cause a state change, it must include this token.
    e. The server then validates the token to ensure that the request is legitimate and not forged.
    f. Use middleware like csurf in Express applications to generate and validate CSRF tokens automatically.

g. When a user accesses a form, the server generates a CSRF token and includes it as a hidden field in the form or sends it in a header.
h. For AJAX-based applications, send the token in the response headers or in the body, and then include it in subsequent AJAX requests.
i. res.setHeader('X-CSRF-Token', req.csrfToken());

Deployment

1. Lets deploy our backend first
2. Copy both the projects and have them separately
3. We will take them to desktop
4. Go to render - https://render.com/
5. We will use the repo on our github to deploy
6. Go to render-> create a webservice
7. Build an deploy from git repository
8. Connect your repository



9. Enter your start command as per your project
10. Enter your env variables other than port
11. Choose a free version - no cc needed

12. Our service should be deployed
13. To test enter the deployed url and the route for products
14. You can deploy latest commits on render when needed

Deploying frontend on netlifly

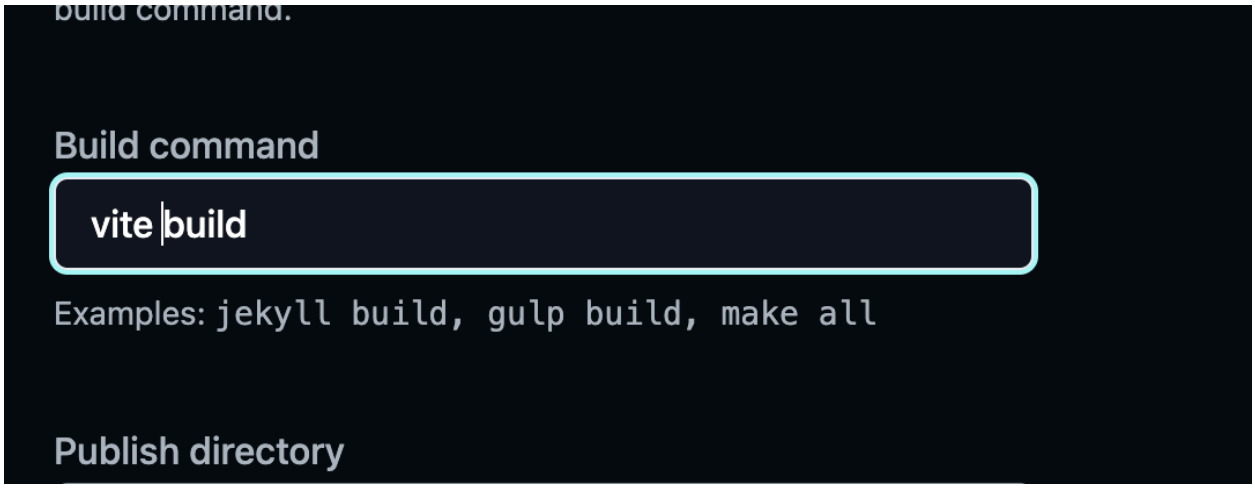
1. Making these cors changes on backend to allow requests for now

```
const corsConfig = {
 origin: true,
 credentials: true,
};
// this is allowing all the requests
app.use(cors(corsConfig));
app.options('*', cors(corsConfig));
```

2.
3. Copy the front end folder to desktop or somewhere and create a separate github repository for this
4. Create a new repo
5. Copy the commands in your vs code terminal for the front end project
6. Add all the files
7. Need to make change to the urlconfig file

```
const  BASE_URL = 'https://backend-test-cckz.onrender.com'
```

8. Copy this backend url from render
9. Import the project on netlify
10. In the build command, enter vite build (from package.json)



11. It should work.
12. Go to cart page. Should take you to login page