in collaboration with

ST6005CEM Security

Final Documentation

For

Comprehensive Security Development of Booze.np

Submitted by: Nawaraj Shrestha

Coventry ID: 12980196

Module Leader: Arya Pokharel

Acknowledgment

I sincerely thank my instructor, mentors, and peers for their guidance, support, and invaluable insights throughout this project. Their expertise and encouragement were instrumental in the development of the Booze web application. I also appreciate the resources and research materials that helped improve the security implementation of this system.

Contents

Table of Figures

Table of Abbreviations

MFA	Multi-Factor Authentication
RBAC	Role-Based Access Control
XSS	Cross-Site Scripting
CSRF	Cross-Site Request Forgery
SQLi	SQL Injection
JWT	JSON Web Token
HTTPS	HyperText Transfer Protocol Secure
SSL/TLS	Secure Sockets Layer / Transport Layer Security
API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CSP	Content Security Policy
MITM	Man-in-the-Middle Attack
OWASP	Open Web Application Security Project
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
OTP	One-Time Password
HTTP	HyperText Transfer Protocol
HSTS	HTTP Strict Transport Security
NoSQL	Not Only SQL
ORM	Object-Relational Mapping
PCI-DSS	Payment Card Industry Data Security Standard
IP	Internet Protocol
CDN	Content Delivery Network

Abstract

The Booze web application is a secure and scalable online liquor ordering platform built on the MERN stack (MongoDB, Express.js, React.js, Node.js). It uses modern security measures like Multi-Factor Authentication (MFA), Role-Based Access Control (RBAC), SSL/TLS encryption, and brute-force prevention to protect user data and transactions. Key security features include XSS protection, session management, and improved API security. Rigorous vulnerability assessments and penetration testing ensure that the system is resilient to cyber threats. Booze is built for efficiency, scalability, and ease of use, providing a safe and seamless user experience while adhering to industry security standards.

Introduction

Booze is a secure and efficient online ordering liquor service, aimed at providing customer with seamless and secure online purchase experience. Since web applications are usually targeted for cyberattacks in this digital world, security of web application become an important component in the development process. Booze ensures that information about users, authentication, and transactions is secure from vulnerabilities in security by making sure that security is fore-fronted. Booze seeks to establish a safe setting for both the administrators and the customers using all the measures required for implementation and adhering to modern best practices in cybersecurity (*OWASP Top Ten | OWASP Foundation*, n.d.).

Booze lays a very sound foundation for a security framework while leveraging the concepts of the CIA Triad (Confidentiality, Integrity, and Availability). It ensures that private information is kept safe, intact, and available only to the person or people authorized. To this effect, brute-force prevention techniques reduce unwanted attempts to access information where RBAC enforces strict restrictions on what a user can and cannot do within the system, while multi-factor authentication works in verifying login credentials. It encrypts all communications via SSL/TLS, thus ensuring private and impenetrable data transfers.

Along with the development, Booze followed the important secure development guidelines provided by international cybersecurity laws such as Payment Card Industry Data Security Standard (PCI-DSS) and General Data Protection Regulation (GDPR). Activity logging is used for tracking all the processes, and industry standards compliance assured the secure session management and validation of the input values to stop the SQL injection/XSS attacks (*Cybersecurity Framework | NIST*, 2025). Besides providing a functioning site, booze demonstrates how modern web applications can integrate security in their design to defeat cyberattacks and increase users' confidence.

Software Details

The Booze web application is built by using MERN stack (MongoDB, Express.js, React.js, and Node.js) which is the reason for an effective and scalable environment in modern web applications. The React.js framework drives the dynamism of the frontend, thereby making it more responsive and easier to interact with. The Express.js and Node.js-driven backend makes for seamless and secure interaction between the client and the database to handle the requests made by a user in real time (*HTTPS | Node.js v23.7.0 Documentation*, n.d.). JavaScript on the frontend and backend will make development easier and make the code maintainable.

Booze uses MongoDB for data management, which is a NoSQL database that stores structured and semi-structured data in a flexible and scalable way Team (n.d.). Mongoose is an ODM library that enforces data validation and security constraints to block any unauthorized changes. Passwords are securely encrypted through bcrypt hashing. Query validation helps avoid NoSQL injection-type attacks. Efficient management of the database regarding user accounts, product listing, orders, and reviews for peak performance of the application even when scaling up. This is ensured through the use of RBAC to prevent the release of sensitive information from unauthorized database access.

Booze is deployed by following the best security practices that has a fully encrypted environment. This forces HTTPS with SSL/TLS certificates generated by mkcert, secures communication between client and server, protects unauthorized access and brute-force attacks on the backend by CORS policies, rate limiting, and secure session management. API keys, authentication secrets are kept safe via environment variables or .env files. This ordered deployment ensures security, scalability, and accessibility, maintaining a system that provides user experience continuity and protects user information.

Design and Implementation

The development of the Booze web application is based on a client-server architecture, considering the environment for user interaction-transaction handling-structured, scalable, and secure. On the other hand, there is a React.js-based frontend that offers an easy interface for looking through products, maintaining a cart, and ordering. On user interaction with the application, RESTful APIs will send the request to the Node.js backend with Express.js. The backend processes these requests, checks authentication and validation, and talks to MongoDB for securely fetching or storing data (Bezkoder, 2023).

Security features are on when the processes of requests are implemented. JWT enforces authentication and authorization so that only an authenticated user has access to resources that are forbidden for others. Multi-Factor Authentication provides a second factor while logging into the application. Input validation and sanitization have been used in order to defend against XSS and SQL injection for data integrity. Communication between client and server-frontend and backend-is implemented via HTTPS, using SSL/TLS encryption so that no critical user data intercepts. This architecture is selected due to its scalability which will easily expand with the increase in traffic, its security, where there is enforced access control and encryption, and also its maintainability, which comes with the small modular components, making updates and improvements pretty simple. Booze application which provide a secure, efficient, and robust way of ordering liquor online by following secure development practices.

Security by Designs

The Booze web application follows the Security by Design principle that sees to it that security is incorporated in each phase of development and not as an afterthought. This involves the detection of potential threats early and the adoption of proactive measures to secure the system. Threat modeling was employed in the process to determine risks from unauthorized access, data leakage, and API misuse, thereby enabling the use of preemptive countermeasures.

A number of security measures were implemented to avoid the top ten OWASP vulnerabilities. SQL injection vulnerabilities were avoided by employing MongoDB with Mongoose ORM to ensure parameterized queries that barred direct database manipulation. Cross-Site Scripting (XSS) was handled through input sanitization using sanitize-html, which stopped malicious scripts from being executed. Multi-Factor Authentication (MFA), strict password policies, and JWT-based secure session management put an end to broken authentication vulnerabilities (*OWASP Top Ten | OWASP Foundation*, n.d.).. Insecure Deserialization attacks were thwarted by restricting untrusted data parsing and implementing strict validation procedures for all incoming requests.

The architecture follows key security principles, including the Principle of Least Privilege, which states that users and administrators should only have the rights they need to carry out their job. Defense in Depth includes HTTPS encryption, brute-force protection, and access control. Secure Defaults were enforced, such as the strong password requirement, role-based access control (RBAC), and rate limiting and authentication policies to secure API endpoints. Booze boasts a secure and threat-proof infrastructure through the inclusion of security at all levels (*Secure by Design | CISA*, n.d.).

Key Features and interaction

Booze web application is intended to provide a secure and seamless online liquor shopping experience while also incorporating necessary security measures to safeguard user data and transactions. It provides for secure authentication and authorization, giving access to the core feature of the application to only registered and verified users. MFA enhances login security by including an additional layer of verification through OTPs at the time of login, hence reducing the chances of unauthorized access. The system employs Role-Based Access Control to limit administrative functionality so that only those authorized users can manage products, orders, and user accounts. Users are allowed to buy liquor products, add items to cart, and place orders in a secure manner AppsRhino (2024). API requests from the frontend to the backend are encrypted with HTTPS, which does not allow data interception. The payment process follows all guidelines regarding secure transaction handling, thus making it compliant with industry standards. Activity logging allows the administrator to track user activity and order history.

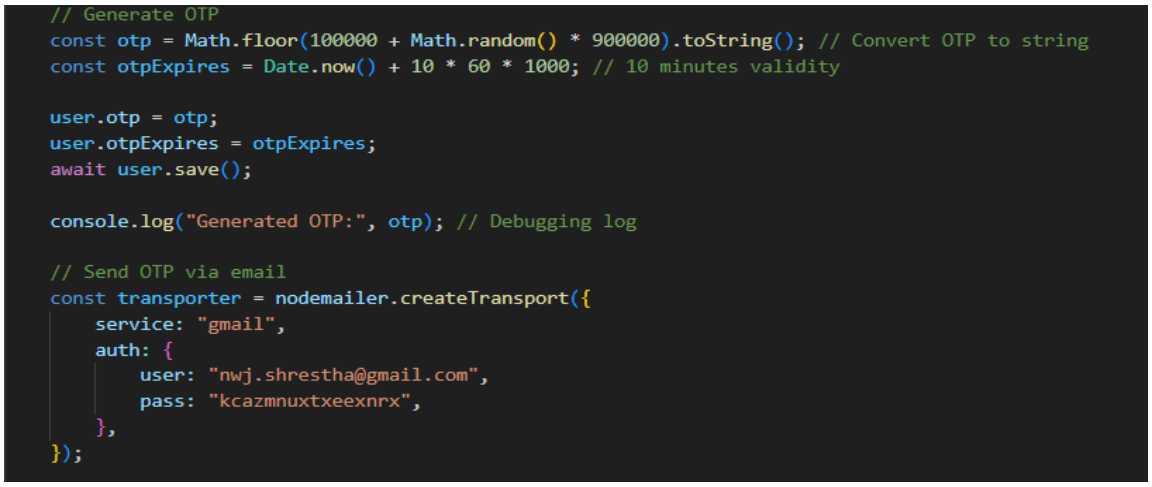
To protect against cyber threats, Booze integrates rate-limiting, brute-force prevention, and session management to ensure that system performance and security are not affected. The user-friendly interface with a secure infrastructure provides customers with a reliable and resilient online liquor shopping experience, while sensitive data is kept far away from unauthorized access.

Detailed Security Implementation

The Booze web application was developed with built-in security in order to maintain the integrity of users' information, transactions, and the system against cyber threats. Booze security adheres to best practices within the industry, such as multi-layer security mechanisms that evade unauthorized access, breaches, and vulnerabilities. Security details of the application are given below:

Multi-Factor Authentication (MFA)

To improve account security, Booze uses OTP-based Multi-Factor Authentication (MFA). Upon logging in, users are emailed a one-time password (OTP) that they must enter to access their accounts. This prevents unauthorized access, even if the login credentials are compromised. The OTP system is implemented with Nodemailer, which ensures secure email delivery (Stack, 2025).



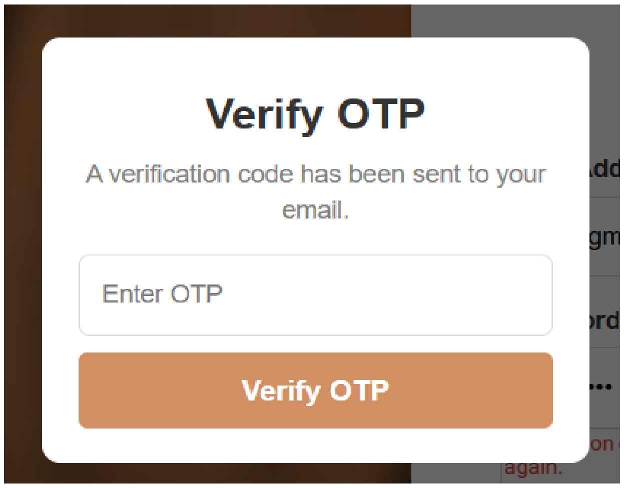
```
// Generate OTP
const otp = Math.floor(100000 + Math.random() * 900000).toString(); // Convert OTP to string
const otpExpires = Date.now() + 10 * 60 * 1000; // 10 minutes validity

user.otp = otp;
user.otpExpires = otpExpires;
await user.save();

console.log("Generated OTP:", otp); // Debugging log

// Send OTP via email
const transporter = nodemailer.createTransport({
    service: "gmail",
    auth: {
        user: "nwj.shrestha@gmail.com",
        pass: "kcazmnuxtxeexnrx",
    },
});
```

Figure 1: OTP generation and email sending logic.



Verify OTP

A verification code has been sent to your email.

Figure 2: OTP verification form

Role Based Access Control

Access control is enforced through Role-Based Access Control (RBAC), which determines which functionalities users and administrators have access to. Regular users can browse products, manage orders, and submit reviews, whereas administrators have more power to manage inventory, track user activity, and process orders. Unauthorized users attempting to access admin functions are automatically blocked by backend middleware authentication. RBAC reduces the risk of privilege escalation attacks and unauthorized administrative access by ensuring that users only have the permissions they need McQuillan (2024).

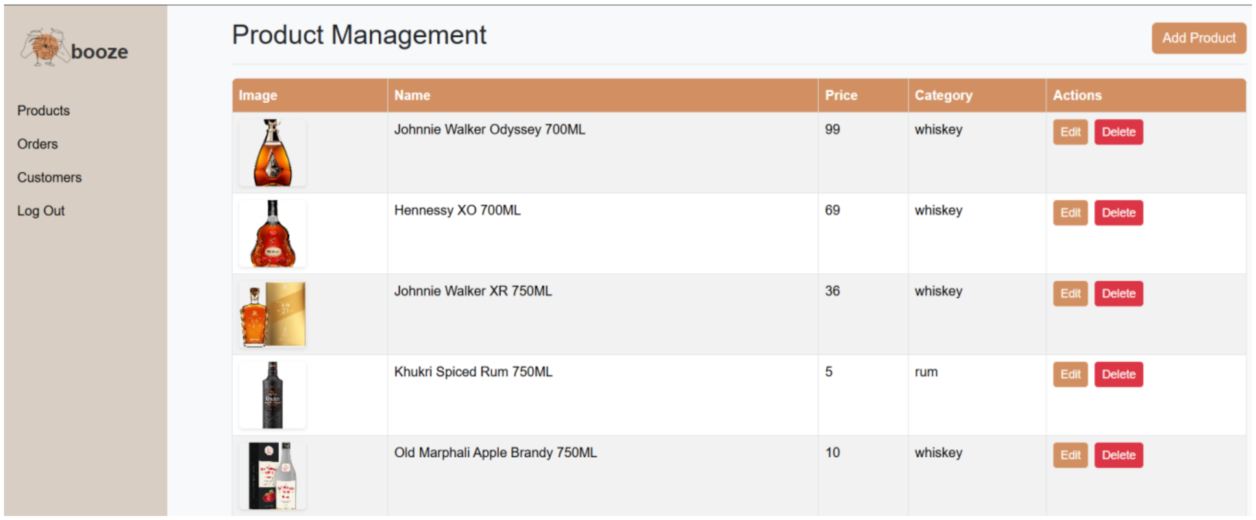
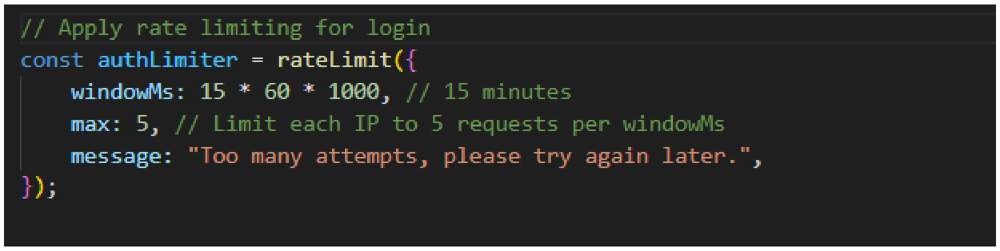


Image	Name	Price	Category	Actions
	Johnnie Walker Odyssey 700ML	99	whiskey	Edit Delete
	Hennessy XO 700ML	69	whiskey	Edit Delete
	Johnnie Walker XR 750ML	36	whiskey	Edit Delete
	Khukri Spiced Rum 750ML	5	rum	Edit Delete
	Old Marphali Apple Brandy 750ML	10	whiskey	Edit Delete

Figure 3: Admin Dashboard.

Brute-Force Prevention

To protect against automated login attempts, Booze implements brute-force prevention mechanisms. The authentication system limits failed login attempts to five per 15 minutes, and further attempts are blocked once the limit is reached. This prevents attackers from systematically guessing passwords and gaining unauthorized access. The system was built with express-rate-limit, which dynamically enforces request restrictions based on IP addresses. Implementing account lockout policies ensures that malicious login attempts are identified and mitigated in real time *(Securing APIs: Express Rate Limit and Slow Down | MDN Blog, 2024).*



```
// Apply rate limiting for login
const authLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 5, // Limit each IP to 5 requests per windowMs
    message: "Too many attempts, please try again later.",
});
```

Figure 4: Rate limiting for login.

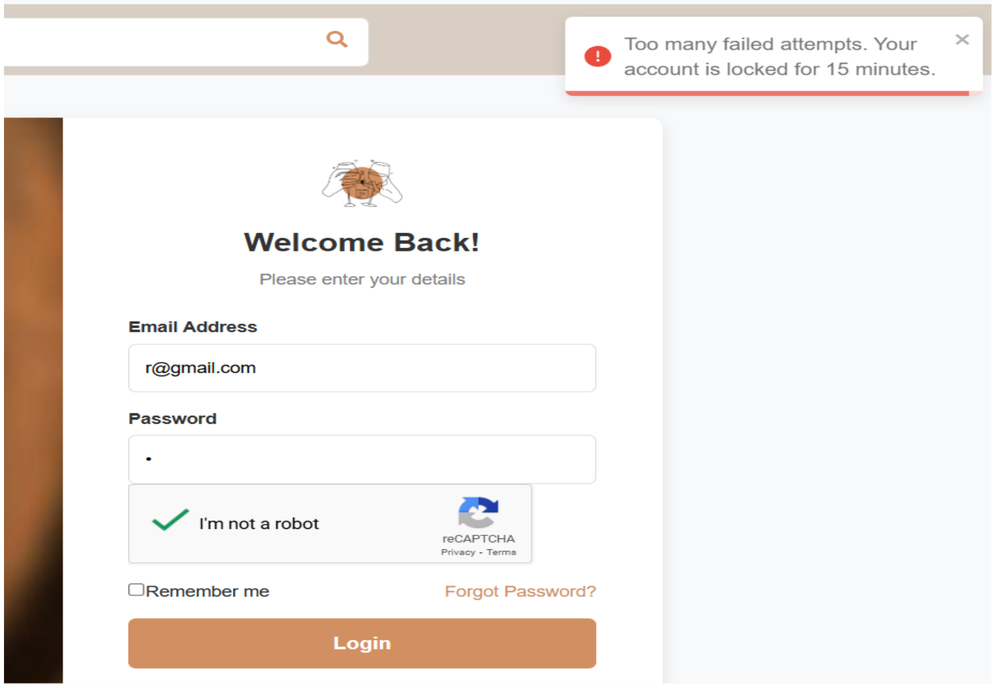
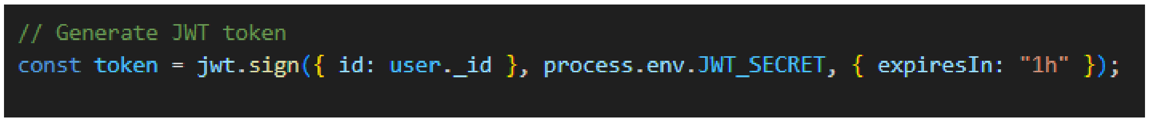

Too many failed attempts. Your account is locked for 15 minutes.

Figure 5: error message displayed to users after exceeding login attempts.

Session Management

Session management is implemented using JWT authentication to ensure that user sessions are secure and tamper-proof. JWT tokens are stored in HTTP-only cookies, which prevents XSS attacks from stealing session information. Additionally, session expiration policies ensure that tokens expire after one hour, lowering the risk of session hijacking. When a user logs out, their session token is revoked, ensuring that previous credentials cannot be used SuperTokens (n.d.). Secure session management protects against unauthorized access while providing consistent user experience.

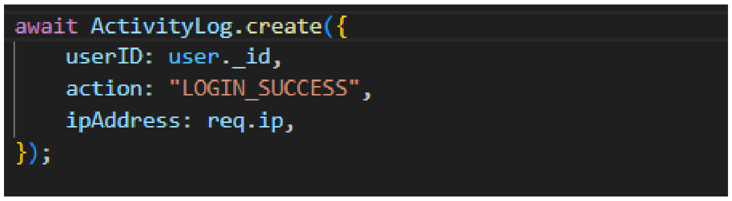


```
// Generate JWT token
const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: "1h" });
```

Figure 6: JWT token generation and expiration settings.

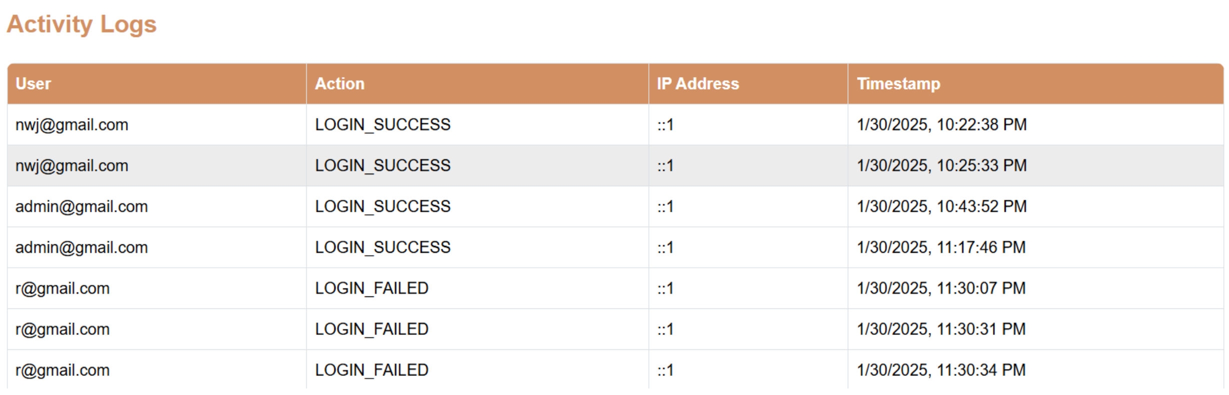
Activity Logging

A robust activity logging system is put in place to track user interactions and detect suspicious behavior. Every login attempt, failed login, order placement, and administrative modification is recorded, allowing administrators to detect potential security risks. Logs are saved in the database and can be accessed through the administrative dashboard. This audit trail allows for real-time threat detection, accountability, and system integrity (Coralogix, 2024). If a security incident occurs, logs can help identify the root cause and take corrective action.



```
await ActivityLog.create({
    userID: user._id,
    action: "LOGIN_SUCCESS",
    ipAddress: req.ip,
});
```

Figure 7: Recording user actions into the database.



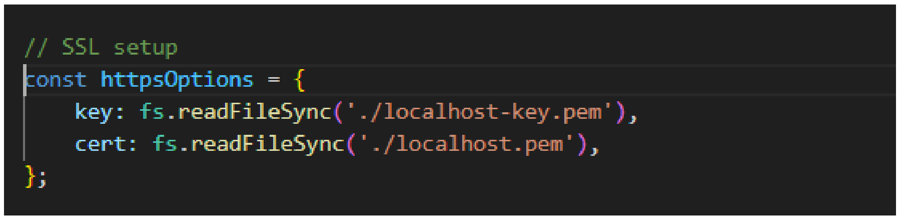
Activity Logs

User	Action	IP Address	Timestamp
nwj@gmail.com	LOGIN_SUCCESS	::1	1/30/2025, 10:22:38 PM
nwj@gmail.com	LOGIN_SUCCESS	::1	1/30/2025, 10:25:33 PM
admin@gmail.com	LOGIN_SUCCESS	::1	1/30/2025, 10:43:52 PM
admin@gmail.com	LOGIN_SUCCESS	::1	1/30/2025, 11:17:46 PM
r@gmail.com	LOGIN_FAILED	::1	1/30/2025, 11:30:07 PM
r@gmail.com	LOGIN_FAILED	::1	1/30/2025, 11:30:31 PM
r@gmail.com	LOGIN_FAILED	::1	1/30/2025, 11:30:34 PM

Figure 8: admin panel displaying logged user activities.

SSL Implementation

To protect against data interception and man-in-the-middle (MITM) attacks, all communications between the frontend and backend are encrypted with SSL/TLS certificates. The certificates were generated with mkcert, which ensures that API requests are securely transmitted over HTTPS. SSL encryption protects sensitive user data such as login credentials, order details, and payment transactions. To protect against potential security breaches, the backend enforces HTTPS-only connections and rejects unsecured HTTP requests (Rodríguez, 2024).



```
// SSL setup
const httpsOptions = {
    key: fs.readFileSync('./localhost-key.pem'),
    cert: fs.readFileSync('./localhost.pem'),
};
```

Figure 9: SSL/TLS certificate integration in the backend server.

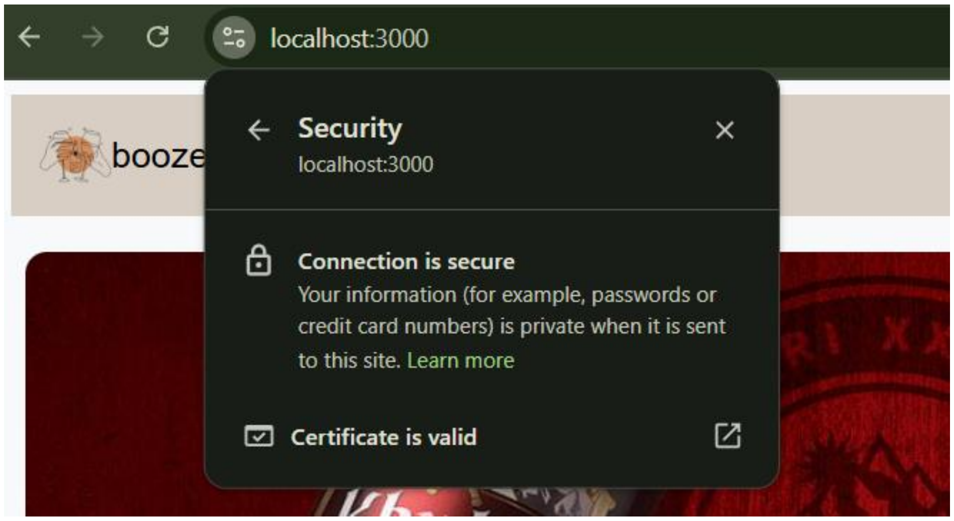
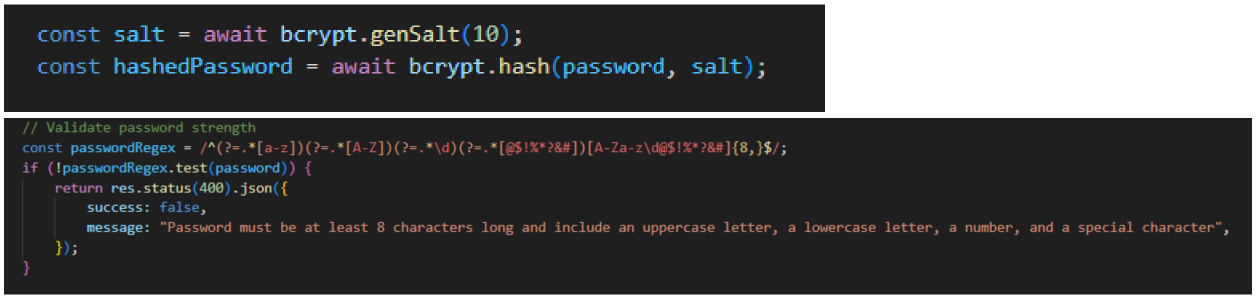

localhost:3000
booze
Security
localhost:3000
Connection is secure
Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more
Certificate is valid

Figure 10: browser address bar displaying a secure HTTPS connection.

Password Security and Encryption

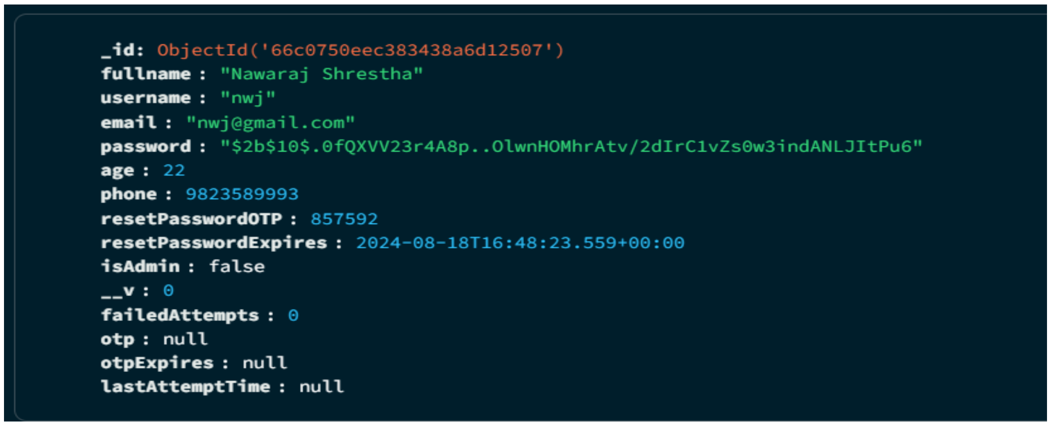
User passwords are protected with bcrypt hashing and salt, ensuring that stored credentials are secure and unreadable even if the database is compromised. Furthermore, a strong password policy is enforced, requiring users to create complex passwords that include uppercase and lowercase letters, numbers, and special symbols Selzer (n.d.).



```
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);
```

```
// Validate password strength
const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&#])[A-Za-z\d@$!%*?&#]{8,}$/;
if (!passwordRegex.test(password)) {
    return res.status(400).json({
        success: false,
        message: "Password must be at least 8 characters long and include an uppercase letter, a lowercase letter, a number, and a special character",
    });
}
```

Figure 11: bcrypt password hashing and verification in user authentication.

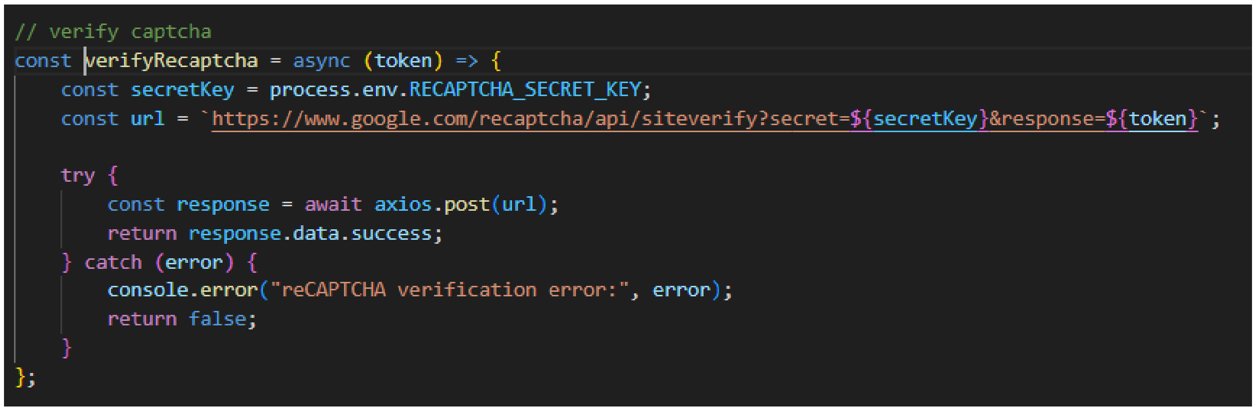


```
_id: ObjectId('66c0750eec383438a6d12507')
fullname : "Nawaraj Shrestha"
username : "nwj"
email : "nwj@gmail.com"
password : "$2b$10$.0fQXVV23r4A8p..OlwnHOMhrAtv/2dIrC1vZs0w3indANLJItPu6"
age : 22
phone : 9823589993
resetPasswordOTP : 857592
resetPasswordExpires : 2024-08-18T16:48:23.559+00:00
isAdmin : false
__v : 0
failedAttempts : 0
otp : null
otpExpires : null
lastAttemptTime : null
```

Figure 12: hashed password saved in the database

reCAPTCHA Implementation

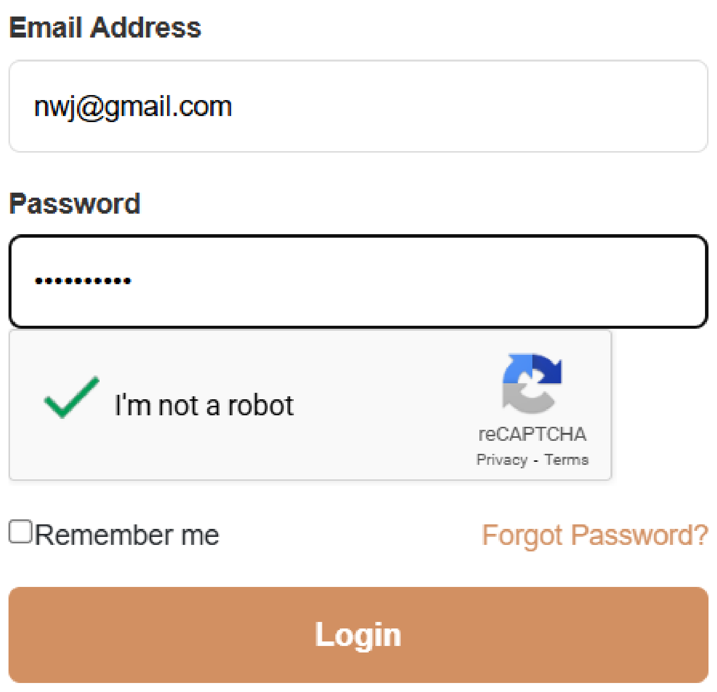
To prevent automated bot attacks and ensure that only legitimate users interact with the system, including Google reCAPTCHA v2. This security measure ensures that form submissions are made by humans, preventing spam registrations and credential stuffing attempts. If a reCAPTCHA token fails validation, the request is denied, preventing bot-driven attacks (*Create reCAPTCHA Keys for Websites*, n.d.).



```
// verify captcha
const verifyRecaptcha = async (token) => {
    const secretKey = process.env.RECAPTCHA_SECRET_KEY;
    const url = `https://www.google.com/recaptcha/api/siteverify?secret=${secretKey}&response=${token}`;

    try {
        const response = await axios.post(url);
        return response.data.success;
    } catch (error) {
        console.error("reCAPTCHA verification error:", error);
        return false;
    }
};
```

Figure 13: reCAPTCHA verification



Email Address

nwj@gmail.com

Password

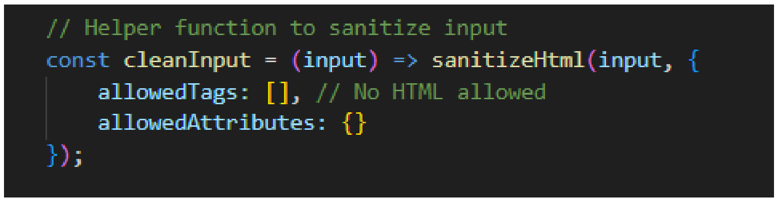
••••••••••

Figure 14: reCAPTCHA in login form.

Cross-Site Scripting (XSS) Protection

To prevent XSS attacks, user inputs are sanitized with sanitize-html, which ensures that malicious scripts cannot be injected. Attackers frequently use form fields to execute

JavaScript that can steal user sessions or manipulate content. By sanitizing inputs before storing them in the database, Booze eliminates the risk of stored XSS, ensuring that user-provided data is secure and cannot be used for session hijacking or phishing attacks (*Cross Site Scripting Prevention - OWASP Cheat Sheet Series*, n.d.).

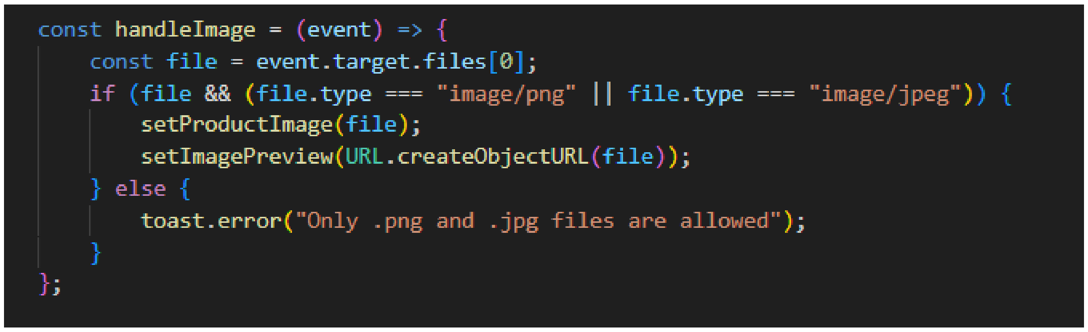


```
// Helper function to sanitize input
const cleanInput = (input) => sanitizeHtml(input, {
    allowedTags: [], // No HTML allowed
    allowedAttributes: {}
});
```

Figure 15: sanitize input

Image Upload Validation (PNG & JPEG Only accepted)

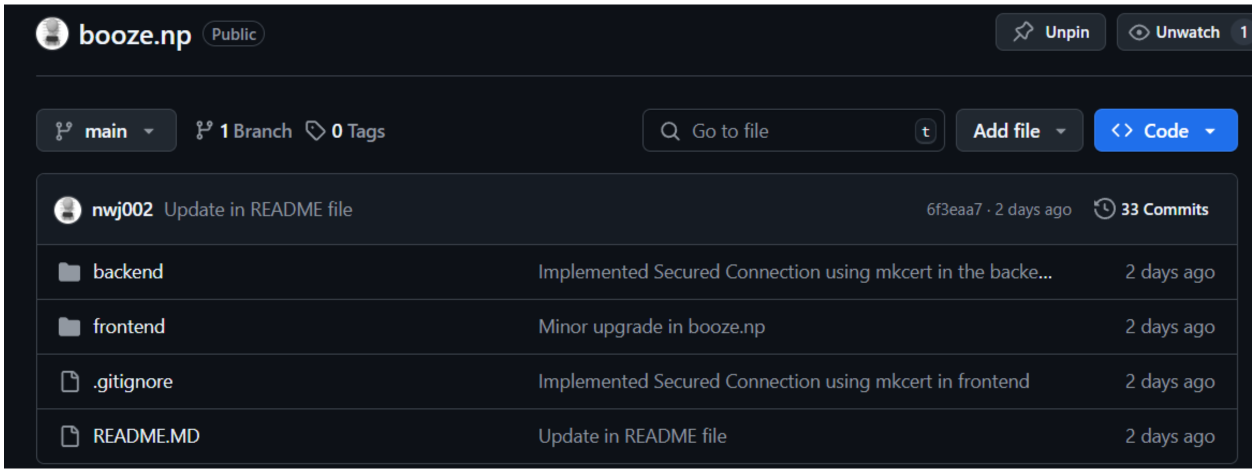
To prevent the upload of malicious files, Booze uses strict file type validation, allowing only PNG and JPEG image formats. This prevents attackers from uploading executable files disguised as images, potentially compromising the server. When you submit an image, the system checks the MIME type and extension before accepting it GeeksforGeeks (2025).



```
const handleImage = (event) => {
    const file = event.target.files[0];
    if (file && (file.type === "image/png" || file.type === "image/jpeg")) {
        setProductImage(file);
        setImagePreview(URL.createObjectURL(file));
    } else {
        toast.error("Only .png and .jpg files are allowed");
    }
};
```

Figure 16: Image Upload validation.

GitHub Commit



booze.np Public

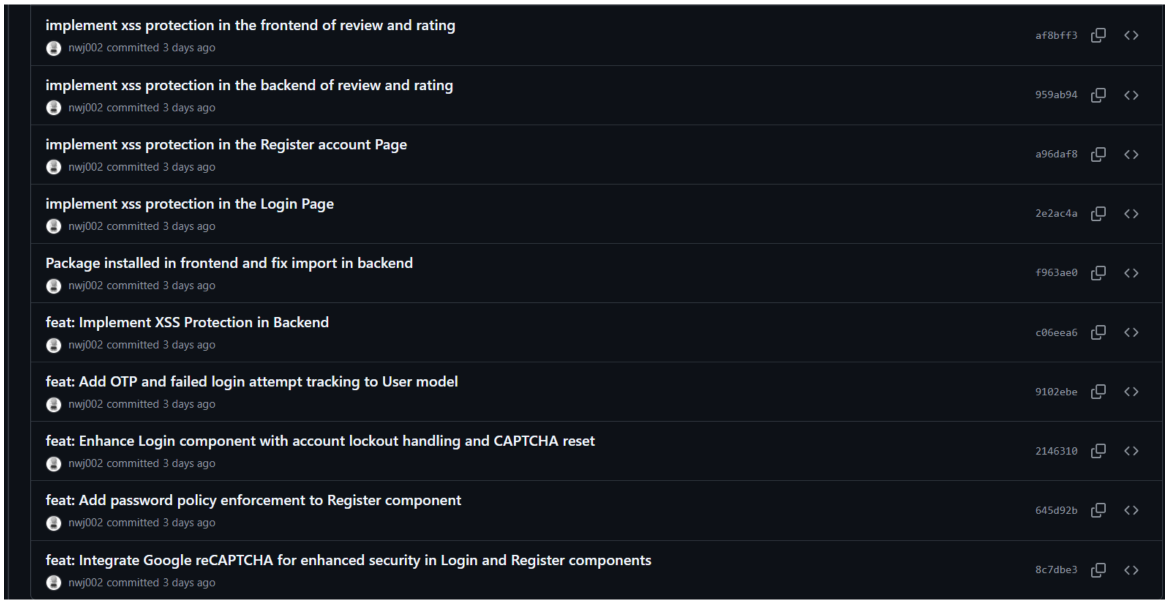
Unpin · Unwatch 1

main · 1 Branch · 0 Tags · Go to file · Add file · Code

nwj002 Update in README file · 6f3eaa7 · 2 days ago · 33 Commits

backend	Implemented Secured Connection using mkcert in the backe...	2 days ago
frontend	Minor upgrade in booze.np	2 days ago
.gitignore	Implemented Secured Connection using mkcert in frontend	2 days ago
README.MD	Update in README file	2 days ago

Figure 17: GitHub Repository



```
implement xss protection in the frontend of review and rating
nwj002 committed 3 days ago                                         af8bff3
implement xss protection in the backend of review and rating
nwj002 committed 3 days ago                                         959ab94
implement xss protection in the Register account Page
nwj002 committed 3 days ago                                         a96daf8
implement xss protection in the Login Page
nwj002 committed 3 days ago                                         2e2ac4a
Package installed in frontend and fix import in backend
nwj002 committed 3 days ago                                         f963ae0
feat: Implement XSS Protection in Backend
nwj002 committed 3 days ago                                         c06eea6
feat: Add OTP and failed login attempt tracking to User model
nwj002 committed 3 days ago                                         9102ebe
feat: Enhance Login component with account lockout handling and CAPTCHA reset
nwj002 committed 3 days ago                                         2146310
feat: Add password policy enforcement to Register component
nwj002 committed 3 days ago                                         645d92b
feat: Integrate Google reCAPTCHA for enhanced security in Login and Register components
nwj002 committed 3 days ago                                         8c7dbe3
```

Figure 18: Commit logs

GitHub Link: https://github.com/nwj002/booze.np.git

Proof of Concept

YouTube Link: https://youtu.be/PPCW0dmMBVs

Conclusion

The Booze web application effectively integrates modern security measures to provide a secure, dependable, and user-friendly online liquor ordering experience. The system effectively protects user data from cyber threats by utilizing Multi-Factor Authentication (MFA), Role-Based Access Control (RBAC), SSL encryption, XSS protection, brute-force prevention, and secure session management. The system's resilience to attacks was validated through rigorous security testing, vulnerability assessments, and penetration testing. Booze adheres to Security by Design principles, ensuring compliance with industry best practices while providing a secure, scalable, and efficient platform. Future enhancements could focus on real-time security monitoring and advanced fraud detection to ensure continuous improvement.

References

OWASP Top Ten | OWASP Foundation. (n.d.). https://owasp.org/www-project-top-ten/

Cybersecurity Framework | NIST. (2025, January 14). NIST. https://www.nist.gov/cyberframework

Team, M. D. (n.d.). Security. MongoDB Manual v8.0. https://www.mongodb.com/docs/manual/security/

HTTPS | Node.js v23.7.0 Documentation. (n.d.). https://nodejs.org/api/https.html

Secure by design | CISA. (n.d.). Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/securebydesign

SuperTokens. (n.d.). What is JWT? Understand JSON Web Tokens | SuperTokens. https://supertokens.com/blog/what-is-jwt

Bezkoder. (2023, October 16). React.js + Node.js + Express + MongoDB example: MERN stack CRUD App - BezKoder. BezKoder. https://www.bezkoder.com/react-node-express-mongodb-mern-stack/

AppsRhino. (2024, April 14). Top 9 must-have features for your Alcohol Delivery App. https://www.appsrhino.com/blogs/top-9-must-have-features-for-your-alcohol-delivery-app.

Stack, S. S. (2025, January 1). How to Implement OTP Verification in Authentication System with Express.js and MongoDB. Medium. https://sandydev.medium.com/how-to-implement-otp-verification-in-authentication-system-with-express-js-and-mongodb-c4f1c1314aed

McQuillan, R. (2024, December 6). How to Implement RBAC in 8 Steps. rbac. https://budibase.com/blog/app-building/how-to-implement-rbac/

Securing APIs: Express rate limit and slow down | MDN Blog. (2024, May 28). MDN Web Docs. https://developer.mozilla.org/en-US/blog/securing-apis-express-rate-limit-and-slow-down/

Coralogix. (2024, March 27). Application logging: definition, examples, and best practices - Coralogix. https://coralogix.com/guides/application-performance-monitoring/application-logging-best-practices/

Rodríguez, A. (2024, July 13). Setting Up Traefik and mkcert for Local Development. DEV Community. https://dev.to/agusrdz/setting-up-traefik-and-mkcert-for-local-development-48j5

Selzer, M. (n.d.). Salt and Hash Passwords with bcrypt. heynode.com. https://heynode.com/blog/2020-04/salt-and-hash-passwords-bcrypt/

Create reCAPTCHA keys for websites. (n.d.). Google Cloud. https://cloud.google.com/recaptcha/docs/create-key-website

Cross Site Scripting Prevention - OWASP Cheat Sheet series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

GeeksforGeeks. (2025, January 10). File Type Validation while Uploading it using JavaScript. GeeksforGeeks. https://www.geeksforgeeks.org/file-type-validation-while-uploading-it-using-javascript/