**Security Review of a Node.js Express Application**

I’ve been working with a simple Node.js and Express application, which handles user registration and login via a REST API. While the functionality works, I noticed a few areas where security could be improved. Let’s dive into the code, identify the vulnerabilities, and discuss some improvements to make the app more secure.

**The Code (Before Review)**

const express = require('express');

const bodyParser = require('body-parser');

const mysql = require('mysql');

const crypto = require('crypto');

const app = express();

app.use(bodyParser.urlencoded({ extended: true }));

const db = mysql.createConnection({

host: 'localhost',

user: 'root',

password: '',

database: 'user\_db'

});

db.connect();

[app.post](http://app.post/)('/register', (req, res) => {

const { username, password } = req.body;

// Using SHA1 for password hashing (insecure)

const hashedPassword = crypto.createHash('sha1').update(password).digest('hex');

const query = `INSERT INTO users (username, password) VALUES ('${username}', '${hashedPassword}')`;

db.query(query, (err, result) => {

if (err) {

return res.status(500).json({ message: 'Database error' });

}

res.status(201).json({ message: 'User registered successfully' });

});

});

[app.post](http://app.post/)('/login', (req, res) => {

const { username, password } = req.body;

const query = `SELECT password FROM users WHERE username = '${username}'`;

db.query(query, (err, result) => {

if (err || result.length === 0) {

return res.status(401).json({ message: 'Invalid credentials' });

}

// Comparing plain password with hashed password

const hashedPassword = crypto.createHash('sha1').update(password).digest('hex');

if (hashedPassword === result[0].password) {

res.status(200).json({ message: 'Login successful' });

} else {

res.status(401).json({ message: 'Invalid credentials' });

}

});

});

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

**Security Issues I Found**

* **SQL Injection Risk**:
* The SQL queries in the code are directly using user input, which makes the app vulnerable to **SQL injection** attacks. An attacker could easily manipulate the input to run malicious queries on the database.

**Fix**: Instead of string interpolation, use **parameterized queries**. This prevents the user input from being treated as executable code.  
  
 **Updated Code**:  
  
 const query = 'INSERT INTO users (username, password) VALUES (?, ?)';

db.query(query, [username, hashedPassword], (err, result) => {

...

});

Similarly, for the login query:  
  
 const query = 'SELECT password FROM users WHERE username = ?';

db.query(query, [username], (err, result) => {

...

});

* **Weak Password Hashing (SHA1)**:
* SHA1 is **no longer considered secure**. It can easily be cracked with modern computational power. Storing passwords this way is risky.

**Fix**: Use a stronger, slower hashing algorithm like **bcrypt** or **argon2**. These algorithms are designed to be computationally expensive, which makes it much harder for attackers to brute-force passwords.  
  
 **Updated Code**: First, install bcrypt:  
  
 npm install bcrypt

Then, update the password hashing code:  
  
 const bcrypt = require('bcrypt');

// In the register route

bcrypt.hash(password, 10, (err, hashedPassword) => {

if (err) {

return res.status(500).json({ message: 'Error hashing password' });

}

const query = 'INSERT INTO users (username, password) VALUES (?, ?)';

db.query(query, [username, hashedPassword], (err, result) => {

...

});

});

// In the login route

bcrypt.compare(password, result[0].password, (err, isMatch) => {

if (isMatch) {

res.status(200).json({ message: 'Login successful' });

} else {

res.status(401).json({ message: 'Invalid credentials' });

}

});

* **Plaintext Password in Database**:
* Storing a password as a simple hashed string without a **salt** (random data added before hashing) makes the hash vulnerable to attacks like **rainbow table attacks**.
* **Fix**: With **bcrypt** or similar algorithms, salting is done automatically. This addresses the vulnerability and strengthens the security of stored passwords.
* **Exposing Sensitive Data in Error Messages**:
* The error handling in the app doesn’t avoid revealing too much information. For example, a generic Database error message might leak internal details about the database structure.

**Fix**: Be more cautious with error messages and avoid exposing too much detail. You can log detailed errors on the server, but return more user-friendly messages to the client.  
  
 **Updated Code**:  
  
 if (err) {

console.error(err); // Log error internally

return res.status(500).json({ message: 'Something went wrong. Please try again later.' });

}

* **No Session Management**:
* There’s no session or token management in this app. If a user logs in, there’s no way to keep them logged in for future requests. You’d need something like **JWT (JSON Web Tokens)** or **sessions**.

**Fix**: Use **JWT** to securely manage user sessions. After a successful login, return a token to the client that they can send with each subsequent request.  
  
 **Updated Code**: First, install jsonwebtoken:  
  
 npm install jsonwebtoken

Then, update the login route to return a JWT:  
  
 const jwt = require('jsonwebtoken');

const token = jwt.sign({ username }, 'your\_jwt\_secret', { expiresIn: '1h' });

res.status(200).json({ message: 'Login successful', token });

* **Insecure HTTP Headers**:
* The app doesn’t set any secure HTTP headers to protect against common attacks like **clickjacking** or **XSS** (cross-site scripting).

**Fix**: Use the **helmet** library to automatically set a variety of security headers in your app.  
  
 **Updated Code**:  
  
 npm install helmet

Then, use it in your app:  
  
 const helmet = require('helmet');

app.use(helmet());

* **No Rate Limiting**:
* The app doesn’t have **rate limiting**, which makes it vulnerable to brute-force attacks, especially on the login route.

**Fix**: Implement rate limiting using a library like **express-rate-limit** to limit the number of login attempts from the same IP address.  
  
 **Updated Code**: First, install express-rate-limit:  
  
 npm install express-rate-limit

Then, apply it to the login route:  
  
 const rateLimit = require('express-rate-limit');

const limiter = rateLimit({

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

max: 5, // limit each IP to 5 requests per window

message: 'Too many requests, please try again later'

});

app.use('/login', limiter);

**Final Thoughts**

By addressing these vulnerabilities, the app will be much more secure. Switching to bcrypt for password hashing, using prepared statements for database queries, and introducing secure session management with JWT are key improvements. Also, implementing rate limiting and securing HTTP headers will help protect against common attacks.

Remember, security is an ongoing process, so I’ll be keeping an eye on new threats and best practices. Regularly reviewing your code and using tools like static code analyzers and penetration testing will help keep things secure in the long run.

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
|  |