### \*\*Broken Access Control\*\*

1. \*\*Issue\*\*: The supply of the user ID in the URL allows anyone to access any profile without checks.

```javascript

app.get('/profile/:userId', (req, res) => {

User.findById(req.params.userId, (err, user) => {

if (err) return res.status(500).send(err);

res.json(user);

});

});

```

\*\*Fixed\*\*: Authenticated requests ensure users can only access their own data unless they're admins.

```javascript

app.get('/profile/:userId', (req, res) => {

if (req.user.id !== req.params.userId && !req.user.isAdmin) {

return res.status(403).json({ error: 'Access denied' });

}

User.findById(req.params.userId, (err, user) => {

if (err) return res.status(500).send(err);

res.json(user);

});

});

```

2. \*\*Issue\*\*: Similar problem with user ID directly from the URL exposing private information.

```python

@app.route('/account/<user\_id>')

def get\_account(user\_id):

user = db.query(User).filter\_by(id=user\_id).first()

return jsonify(user.to\_dict())

```

\*\*Fixed\*\*: Implemented checks to ensure users can only access their own data.

```python

@app.route('/account/<user\_id>')

def get\_account(user\_id):

if str(current\_user.id) != user\_id and not current\_user.is\_admin:

return jsonify({"error": "Access denied"}), 403

user = db.query(User).filter\_by(id=user\_id).first()

return jsonify(user.to\_dict())

```

---

### \*\*Cryptographic Failures\*\*

3. \*\*Issue\*\*: MD5 is broken, unsalted, and too fast for secure password hashing.

```java

public String hashPassword(String password) throws NoSuchAlgorithmException {

MessageDigest md = MessageDigest.getInstance("MD5");

md.update(password.getBytes());

byte[] digest = md.digest();

return DatatypeConverter.printHexBinary(digest);

}

```

\*\*Fixed\*\*: Used BCrypt to hash passwords with a salt to enhance security.

```java

import org.mindrot.jbcrypt.BCrypt;

public String hashPassword(String password) {

String salt = BCrypt.gensalt(12); // work factor = 12

return BCrypt.hashpw(password, salt);

}

public boolean verify(String candidate, String storedHash) {

return BCrypt.checkpw(candidate, storedHash);

}

```

4. \*\*Issue\*\*: SHA-1 is broken and fast, exposing passwords to brute force attacks.

```python

import hashlib

def hash\_password(password):

return hashlib.sha1(password.encode()).hexdigest()

```

\*\*Fixed\*\*: Used bcrypt for better password handling.

```python

import bcrypt

def hash\_password(password: str) -> bytes:

return bcrypt.hashpw(password.encode(), bcrypt.gensalt())

def verify\_password(password: str, hashed: bytes) -> bool:

return bcrypt.checkpw(password.encode(), hashed)

```

---

### \*\*Injection\*\*

5. \*\*Issue\*\*: Direct concatenation of user input leads to SQL injection.

```java

String username = request.getParameter("username");

String query = "SELECT \* FROM users WHERE username = '" + username + "'";

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(query);

```

\*\*Fixed\*\*: Switched to parameterized queries to prevent SQL injection attacks.

```java

String sql = "SELECT \* FROM users WHERE username = ?";

PreparedStatement ps = connection.prepareStatement(sql);

ps.setString(1, username);

ResultSet rs = ps.executeQuery();

```

Feel free to customize the formatting and structure further if you need!

6. Send the key problems associated with how drivers can theoretically be exploited by an attacker. Bypassing injection operations to JSON could lead to security vulnerabilities or data disclosure. To address this, we verified and reinforced the requirements from users to ensure that input is sanitized and safe patterns are matched.

```javascript

app.get('/user', (req, res) => {

// Directly trusting query parameters can lead to NoSQL injection

db.collection('users').findOne({ username: req.query.username }, (err, user) => {

if (err) throw err;

res.json(user);

});

});

```

\*\*Fixed:\*\*

```javascript

const username = req.query.username;

if (typeof username !== 'string' || !/^[\w.@-]{3,32}$/.test(username)) {

return res.status(400).send('Invalid username');

}

db.collection('users').findOne({ username: username }, ...);

db.collection('users').findOne({ username: { $eq: username } }, ...);

```

---

\*\*Insecure Design\*\*

7. In this scenario, there is no authorization or verification for input. Anyone who knows an email address can reset anyone's password. To prevent this, we implemented a system where users can request a single-use password reset code. This code is hashed and sent via email. Once the user clicks the link to reset their password, they can create a new password that the server verifies. After a successful reset, the code is deleted.

```python

@app.route('/reset-password', methods=['POST'])

def reset\_password():

email = request.form['email']

new\_password = request.form['new\_password']

user = User.query.filter\_by(email=email).first()

user.password = new\_password

db.session.commit()

return 'Password reset'

```

\*\*Fixed:\*\*

1. Create a token and send an email link:

```python

token = secrets.token\_urlsafe(32)

hashed = hash\_token(token) # Store only the hashed token

store\_reset\_token(user, hashed, expiry)

send\_email(user.email, f"https://.../reset?token={token}")

```

2. Verify the token on the reset endpoint, then:

```python

user.password = bcrypt\_hash(new\_password)

delete\_reset\_token(...)

```

---

\*\*Software and Data Integrity Failures\*\*

8. A third-party script can compromise the integrity of this code by running JavaScript in the browser. If the content delivery network (CDN) is compromised, your site is vulnerable. To mitigate this risk, consider implementing strict source verification and enforcing Subresource Integrity (SRI) to confirm that packages are valid.

```html

<script src="https://cdn.example.com/lib.js"></script>

```

\*\*Fixed:\*\*

```html

<script src="https://cdn.example.com/lib.js"

integrity="sha384-BASE64\_HASH\_HERE"

crossorigin="anonymous"></script>

```

---

\*\*Server-Side Request Forgery\*\*

9. In this code, there is unrestricted user input from the URL, allowing requests to internal APIs and host addresses, which can be easily exploited by attackers. To combat this, implement a whitelist of allowed user domains and validate URLs. Additionally, limit the IP addresses that can be accessed.

```python

url = input("Enter URL: ")

response = requests.get(url)

print(response.text)

```

\*\*Fixed:\*\*

```python

from urllib.parse import urlparse

import ipaddress

import requests

u = urlparse(url)

host = u.hostname

# Resolve host to IP and check

ip = socket.gethostbyname(host)

if ipaddress.ip\_address(ip).is\_private:

raise ValueError("Disallowed")

# Proceed with timeout and limit the response size

resp = requests.get(url, timeout=5, allow\_redirects=False)

```

---

\*\*Identification and Authentication Failures\*\*

10. One issue here is the insecure handling of passwords, as anyone can view the code and see the passwords, which makes them vulnerable. Passwords should be hashed using secure algorithms instead of being stored in plain text. It's also important to verify user access securely.

```java

if (inputPassword.equals(user.getPassword())) {

// Login success

}

```

\*\*Fixed:\*\*

Implement a proper hashing mechanism for storing passwords and ensure that the user’s input is securely verified against the hashed password.