Of course. Let's document **Practical 13**. This is a critical one that focuses on the security of your application. Your project already has excellent implementations of all the required security features, so this will be a matter of documenting them correctly.

Here is a detailed guide on what to write in your practical file for Practical 13.

Practical 13: Implement Validation, Sanitization, and Password Hashing

Aim: To implement robust security measures in the ByteBazaar application by enforcing server-side validation, sanitizing user inputs, using password hashing, and preventing SQL injections.

1. Key Questions & Analysis

1.1. Is password_hash() used correctly?

Yes. The application uses PHP's modern and secure password_hash() function to protect user passwords.

- **Hashing on Registration:** In the api/register.php script, when a new user is created, their plain-text password is never stored. Instead, it is converted into a strong, salted hash using password_hash($password, PASSWORD_DEFAULT). This ensures that even if the database were compromised, the actual passwords would not be exposed.
- **Verification on Login:** In the api/login.php script, the submitted password is not hashed and compared. Instead, it uses password_verify($password, $user['password']) to securely compare the plain-text input against the stored hash. This is the correct and secure way to verify a hashed password.

Example of Password Hashing from api/register.php:

PHP

```
// From api/register.php
```

```php
// Hash password for security
$hashed_password = password_hash($password, PASSWORD_DEFAULT);

// Insert user using a prepared statement
$stmt = $conn->prepare("INSERT INTO users (first_name, last_name, email, password) VALUES (?, ?, ?, ?)");
$stmt->bind_param("ssss", $first_name, $last_name, $email, $hashed_password);

if ($stmt->execute()) {
    // Success
}
```

1.2. Are form inputs validated on both ends?

Yes. The project follows the best practice of "never trust the client" by implementing validation on both the frontend (client-side) and backend (server-side).

- **Client-Side:** JavaScript in files like js/register.js and js/login.js provides immediate, real-time feedback to the user, checking for empty fields, email format, and password strength before the form is even submitted.
- **Server-Side:** The PHP scripts (api/register.php, api/update_profile.php) re-validate all incoming data. This is a crucial security layer that protects the application even if the frontend JavaScript is bypassed. For example, api/register.php checks for empty fields, validates the email format with filter_var, and enforces password rules, rejecting any invalid data before it reaches the database.

1.3. Are SQL injections prevented?

Yes. The application is protected against SQL injection attacks by exclusively using **prepared statements** for all database queries that involve user-supplied data.

- **Implementation:** Instead of directly inserting variables into SQL strings, the code uses mysqli::prepare(), mysqli_stmt::bind_param(), and mysqli_stmt::execute(). This method separates the SQL query logic from the data.
- **Security Benefit:** By treating user input strictly as data, the database engine does not execute any malicious SQL code that might be injected into form fields. This is demonstrated in all backend scripts that interact with the database, including api/register.php, api/login.php, api/get_product.php, and api/update_profile.php.

Example of Prepared Statement from api/login.php:

PHP

```
// From api/login.php
// The '?' are placeholders. The user's email is bound as a parameter, not concatenated.
$stmt = $conn->prepare("SELECT u_id, password, ... FROM users WHERE email = ?");
$stmt->bind_param("s", $email);
$stmt->execute();
$result = $stmt->get_result();
```

2. Add a CAPTCHA field (Supplementary Problem)

(This is an advanced feature. While not currently implemented in your project, here is how you would document it if you were to add it.)

To further enhance security against automated bots, a CAPTCHA could be integrated. A service like **Google reCAPTCHA** is the industry standard.

- **Frontend:** The reCAPTCHA widget would be added to the register.html form.
- **Backend:** In register.php, before processing the registration, a request would be sent to the Google reCAPTCHA API along with the user's response token to verify that the submission was made by a human. If the verification fails, the registration process is halted.

3. Secure Form Submission Test (Post-Laboratory Work)

To demonstrate the security features, you can perform the following tests:

Test Case 1: Password Hashing Verification
[Insert a screenshot of your users table in phpMyAdmin. Highlight the password column to show the long, hashed strings, proving that plain-text passwords are not being stored.]
Test Case 2: Server-Side Validation Test

1. Temporarily disable JavaScript in your browser settings.
2. Try to submit the registration form with invalid data (e.g., an invalid email or mismatching passwords).
3. *[Insert a screenshot of the Network tab in your browser's developer tools showing the response from api/register.php. The response should be a JSON object with success: false and an appropriate error message, proving that the server rejected the invalid data.]*

Test Case 3: SQL Injection Prevention (Conceptual)
Explain how prepared statements prevent SQL injection. For example, if a user tried to enter ' OR '1'='1 as their email, the bind_param function ensures this is treated as a literal string, not as executable SQL code. The database would simply search for a user with that exact (and likely non-existent) email address, rather than returning all users.