(A01) Broken Access Control - View Basket via Session Storage [High]

Description

A Broken Access Control vulnerability has been identified in the application, allowing attackers to view another user's shopping basket by manipulating the `bid` value stored in the session storage. The vulnerability is caused by the application's failure to enforce proper access controls, enabling unauthorized access to another user's basket by simply modifying session data in the browser's developer tools.

Technical Details

- **Vulnerable Mechanism**:
  - The `bid` value (basket ID) is stored in the session storage within the browser's developer tools, allowing it to be manipulated by the user.
  - The application does not enforce proper access control checks on the `bid` stored in session storage, meaning users can modify it to access baskets that do not belong to them.
- **Steps to Exploit**:
  1. Log in to your account on the application.
  2. Open the browser's developer tools (e.g., Chrome inspect Tools).
  3. Navigate to `Application` > `Session Storage` and locate the key storing the `bid` value (e.g., `bid`).
  4. Modify the `bid` value to the ID of another user's basket (e.g., change the basket ID to someone else's).
  5. Refresh the page or navigate to the basket page to view the contents of the other user's basket.
- **Example of the `bid` Manipulation**:
  - Original `bid` in session storage: `bid: 1`
  - Modified `bid`: `3`
  - By changing the `bid`, the attacker can view and manipulate the basket of another user.

Vulnerability Details

- **Root Cause**: The application uses client-side session storage to store sensitive data like the `bid`, and there is no validation on the server side to ensure that the `bid` belongs to the authenticated user.
- **Exploitation**: By modifying the `bid` value in session storage, an attacker can bypass access control and view another user's basket, which could expose sensitive information such as products in the basket, quantities, and prices.

- **Risk**: Full access to a user's shopping basket data, which may include private or sensitive shopping preferences. This could lead to privacy violations and potentially affect user trust.

Steps to Reproduce

1. **Login**:
    - Log into your user account and view your shopping basket.
2. **Modify `bid` in Session Storage**:
    - Open the browser's developer tools (right-click on the page, click “Inspect”, and navigate to the `Application` tab).
    - Under `Session Storage`, locate the `bid` key that stores your basket ID.
    - Change the value of `bid` to the ID of another user's basket.
3. **Access Another User’s Basket**:
    - After modifying the `bid`, reload the page or navigate to the shopping basket page to view the other user's basket contents.

Impact

- **Data Exposure**: Attackers can view another user's shopping basket, which may contain sensitive product selections, pricing, or other personal information.
- **Privacy Violation**: This allows attackers to access information about other users’ shopping behavior or preferences without authorization.
- **Reputation Damage**: Unauthorized access to user data can damage the platform’s reputation and user trust, leading to potential data protection violations.

Recommendations

1. **Enforce Access Controls on the Server**:
    - Validate that the `bid` in session storage belongs to the currently authenticated user before allowing access to the shopping basket. This should be done server-side, ensuring that session data cannot be tampered with.
2. **Do Not Store Sensitive Data in Session Storage**:
    - Avoid storing sensitive information, such as `bid` or user-specific data, in session storage. Instead, rely on server-side mechanisms to associate the user with their basket.
3. **Use Tokenized Session Data**:
    - Consider using session tokens or encrypted identifiers that cannot be easily modified by the user. This adds a layer of security against tampering.
4. **Implement Proper Authorization Checks**:
    - Ensure that each user can only access their own resources. Before serving the basket data, the application should check whether the logged-in user is authorized to view the basket ID stored in their session.

(A02) Broken Authentication - Password Strength [Critical]

Description

The application is vulnerable to *Broken Authentication* due to weak password policies and the ability to guess the administrator's password without implementing proper security measures. By sending multiple requests with common or easily guessable passwords, an attacker can successfully authenticate as the administrator without requiring SQL injection or any pre-existing changes to the password. The email for the administrator account (`admin@juice-sh.op`) was discovered through one of the reviews on the platform.

Technical Details

- **Vulnerable Endpoint**:
  `POST /rest/user/login`
- **Request Headers**:
  - **Cookie**: Contains session data.
  - **Content-Type**: `application/json`.
  - **Origin**: `https://juice-shop.herokuapp.com`.
- **Payload**:

```
 {

  "email": "admin@juice-sh.op",
  "password": "§admin§"
}
```

- **Attack Method**:
  - I tried multiple common passwords for the `admin@juice-sh.op` account using Burp Suite's "Intruder" tool.
  - The password "admin123" was successfully accepted after testing around 20 common passwords.
- **Response**:
  - HTTP Status: `200 OK`, indicating a successful login with the admin credentials.

Vulnerability Details

- **Root Cause**: The application does not implement strong password policies, allowing attackers to guess weak passwords through brute-force attempts.
- **Exploitation**: Without any account lockout or rate limiting in place, attackers can easily attempt common passwords and gain unauthorized access to the admin account.

- **Risk**: Full administrative access to the application, potentially compromising sensitive user data and allowing further exploitation of the system.

Steps to Reproduce

1. **Identify the Vulnerable Login Endpoint**:
   Intercept the login request using Burp Suite and find the `/rest/user/login` endpoint.
2. **Test Common Passwords**:
   Using Burp Suite Intruder, test a list of common passwords (e.g., "admin123", "password", "123456") against the `admin@juice-sh.op` account.
3. **Observe Successful Login**:
   After testing the password "admin123", the response was `200 OK`, confirming successful login as the admin user.

Impact

- **Full System Compromise**: With access to the admin account, attackers can perform any action, including viewing, editing, or deleting user data, and modifying the application's configuration.
- **Reputation Damage**: Unauthorized access to administrative accounts undermines the platform's credibility and user trust.
- **Legal and Compliance Issues**: Unauthorized access to sensitive data can violate data protection regulations, resulting in legal consequences.

Recommendations

1. **Enforce Strong Password Policies**:
   - Require strong passwords for all user accounts, especially for admin accounts (e.g., minimum length of 12 characters, a mix of uppercase, lowercase, numbers, and special characters).
2. **Implement Rate Limiting**:
   - Limit the number of failed login attempts per account to mitigate brute-force attacks.
   - Implement an account lockout or CAPTCHA mechanism after a set number of failed attempts.
3. **Use Multi-Factor Authentication (MFA)**:
   - Enforce MFA for admin users to provide an additional layer of protection against unauthorized logins.
4. **Monitor and Log Login Attempts**:
   - Continuously monitor login attempts, especially for sensitive accounts like admin, and flag any suspicious activities.

(A03) Broken Access Control: Admin Section Access [High]

Description

Unauthorized access to the administration section of the application (`/administration`) allows attackers with valid administrator credentials to view sensitive data, such as user details and customer feedback. This vulnerability highlights weak access control mechanisms for sensitive endpoints.

Technical Details

- **Affected URL:** `https://demo.owasp-juice.shop/#/administration`
- **Observation:**
  1. Using the browser's **Inspect Element** tool, the keyword "admin" was searched in the client-side application code.
     - This search revealed hints about various admin-related paths, including `/admin` and `/administration`.
     - `/admin` did not provide any functionality or data.
  2. Upon navigating to `/administration`, the server responded with a **403 Forbidden** error for unauthenticated or regular user accounts.
  3. However, after logging in using administrator credentials (`admin@juice-sh.op` and `admin123`), the `/administration` endpoint became fully accessible.

Vulnerability Details

- **Root Cause:**
  Insufficient server-side access control validation for the `/administration` endpoint. The endpoint is accessible based solely on login status and role without robust security checks.
- **Sensitive Data Exposed:**
  - List of registered users and their email addresses.
  - Customer feedback data with associated emails.
- **Impact:**
  - **Confidentiality:** Exposes sensitive user information.
  - **Integrity:** Allows tampering with customer feedback and misuse of administrative actions.

Steps to Reproduce

1. Open the website: `https://demo.owasp-juice.shop/`.
2. Use **Inspect Element** to search for "admin." Identify possible admin-related paths like `/admin` and `/administration`.
3. Test the `/administration` path:
    - Without logging in: Results in a **403 Forbidden** error.
    - With regular user credentials: Also results in a **403 Forbidden** error.
4. Log in with administrator credentials:
    - **Email:** `admin@juice-sh.op`
    - **Password:** `admin123` (or any discovered admin password).
5. Access the `/administration` endpoint:
    - URL: `https://demo.owasp-juice.shop/#/administration`.
    - Observe the exposed sensitive data, including user email addresses and customer feedback.

Impact

- **Data Exposure:** Sensitive user and customer data is visible.
- **Privilege Misuse:** Administrative actions could be exploited to harm data integrity or user trust.

Recommendations

1. **Endpoint Validation:**
    - Implement robust server-side role verification for sensitive pages like `/administration`.
    - Restrict access to such endpoints using middleware or secure access policies.
2. **Credential and Role Security:**
    - Regularly rotate admin credentials.
    - Enforce strong password policies for admin accounts.
3. **Monitoring and Incident Response:**
    - Track and log access to sensitive endpoints.
    - Use a Web Application Firewall (WAF) to block unauthorized attempts.
4. **Fix Client-Side Exposures:**
    - Avoid revealing sensitive paths or keywords like "admin" in the client-side code.

(A04) Broken Access Control - Forged Review [high]

Description

A Broken Access Control vulnerability allows an attacker to impersonate other users by altering the `author` field in a review submission request. By exploiting this issue, attackers can post or edit reviews under the identities of other users, severely compromising the application's integrity and user trust.

Technical Details

- **Endpoint**: `PUT /rest/products/1/reviews`
- **Headers**: Includes `Authorization` and other necessary cookies for authentication.
- **Payload**:

```
{

  "message": "hahahaha",
  "author": "jim@juice-sh.op"
}
```

- The `author` field is not validated server-side to ensure it matches the authenticated user's identity.

Vulnerability Details

- **Root Cause**: The server does not enforce proper access control checks on the `author` field, allowing any user to specify another user's email address or identifier.
- **Exploit**: By modifying the `author` field in the request payload using tools like Burp Suite Repeater, the attacker can submit a review under another user's name.
- **Response**:
  - Status: `201 Created`
  - Response: `{"status":"success"}`
- This confirms the unauthorized creation of a review with a forged identity.

Steps to Reproduce

1. **Preparation**:
   - Log in to the application and intercept the `PUT /rest/products/1/reviews` request using Burp Suite.

2. **Exploit**:
    - Modify the `author` field in the request payload to any other user's email (e.g., `"jim@juice-sh.op"`).
    - Forward the modified request.
3. **Result**:
    - The server processes the request and creates the review with the specified `author`, regardless of the authenticated user's actual identity.

Impact

- **Data Integrity**: Unauthorized manipulation of user-generated content.
- **Reputation Risk**: Fraudulent reviews damage the platform's credibility.
- **Legal and Compliance Issues**: Violates user privacy and may breach data protection laws.

Recommendations

1. **Enforce Strict Access Controls**:
    - Validate the `author` field server-side to ensure it matches the authenticated user's identity from the session or token.
2. **Secure Authentication Mechanisms**:
    - Use JWT or similar tokens to associate review actions with the authenticated user.
3. **Remove User-Controlled Input**:
    - Avoid allowing users to specify sensitive fields like `author` directly in the request payload. Instead, derive this information from the authentication context.
4. **Audit Logging**:
    - Log all review actions with user identifiers and timestamps for traceability.
5. **Regular Security Testing**:
    - Conduct routine penetration testing to identify and remediate similar vulnerabilities.

(A05) Sensitive Data Exposure - Access to Confidential Documents [high]

Description

A Broken Access Control vulnerability allows unauthorized access to confidential documents by navigating to an unprotected directory. This issue arises due to improper access control mechanisms on a public-facing endpoint. By manually altering the path in a URL, an attacker can access sensitive files and download confidential documents.

Technical Details

- **Vulnerable Path**:
  - Initial Link: `https://demo.owasp-juice.shop/ftp/legal.md`
  - Altered Link: `https://demo.owasp-juice.shop/ftp`
- **Root Cause**:
  The `/ftp` directory is publicly accessible and lacks proper access controls. This allows users to list and download files stored within it without authentication or authorization.
- **Files Exposed**:
  Upon accessing the `/ftp` directory, the following files were accessible:
  - `acquisitions.md`
  - `announcement_encrypted.md`
  - `coupons_2013.md.bak`
  - `incident-support.kdbx`
  - `order_6e77-09c50f0dcdbf05b0.pdf`
  - Other files, including sensitive scripts (`encrypt.pyc`) and configuration backups.

Steps to Reproduce

1. **Locate an Unprotected Path**:
   - Visit the **About Us** page of the application.
   - Identify a hyperlink to a specific document:

     ```
     https://demo.owasp-juice.shop/ftp/legal.md
     ```

2. **Manipulate the Path**:
   - Remove the file name (`legal.md`) from the path to access the parent directory:

     ```
     https://demo.owasp-juice.shop/ftp
     ```

3. **Access and Download Files**:
   - The directory lists all accessible files without restrictions.

- Click on any file name to download it (e.g., `acquisitions.md`, `order_6e77-09c50f0dcdbf05b0.pdf`, etc.).

Impact

1. **Data Breach**:
   - Exposed sensitive documents may include user data, internal company details, or proprietary information.
2. **Regulatory Violations**:
   - Leaked data could lead to violations of data protection regulations such as GDPR or CCPA.
3. **Potential for Further Exploitation**:
   - Access to files like `acquisitions.md` or `encrypt.pyc` may enable attackers to decrypt sensitive information or identify further vulnerabilities.
4. **Reputation Damage**:
   - Public disclosure of such a vulnerability could harm the organization's reputation and lead to loss of user trust.

Recommendations

1. **Implement Directory Restrictions**:
   - Restrict public access to directories like `/ftp`. Use access control mechanisms to ensure only authorized users can access specific files.
2. **Disable Directory Listing**:
   - Configure the server to disable directory indexing, preventing users from listing the contents of directories.
3. **Validate File Paths**:
   - Use input validation to ensure users can only access intended files. Avoid exposing sensitive file paths in the application.
4. **File Permissions**:
   - Assign strict permissions to sensitive files, ensuring they cannot be accessed without proper authentication.
5. **Logging and Monitoring**:
   - Monitor access to sensitive directories and files. Implement alerts for unauthorized access attempts.
6. **Secure Backup Files**:
   - Remove unnecessary backup files (e.g., `package.json.bak`) and encrypt sensitive backups. Ensure they are not stored in publicly accessible directories.