**OWASP ZAP Security Assessment Report**

**Website Security Scan Overview**

A security scan was conducted on the website using the OWASP ZAP tool. The following vulnerabilities were detected:

**Summary of Alerts**

|  |  |
| --- | --- |
| Alerts | Total |
| High Priority Alerts | 0 |
| Medium Priority Alerts | 4 |
| Low Priority Alerts | 3 |
| Informational Priority Alerts | 3 |

**Medium Priority Alerts**

**1. Absence of Anti-CSRF Tokens**

* **URL:** http://127.0.0.1:5000/login
* **Risk Level:** Medium
* **Confidence:** Low
* **Description:** No Anti-CSRF tokens were found in an HTML submission form. This makes the application vulnerable to Cross-Site Request Forgery (CSRF) attacks, where malicious actors can force users to perform unintended actions.
* **Code Issue:**
  + <form action="https://validator.w3.org/check" method="post">
* **Recommended Fix:** Implement CSRF protection by using a CSRF token in forms.

**2. HTTP to HTTPS Insecure Transition in Form Post**

* **URL:** http://127.0.0.1:5000/login
* **Risk Level:** Medium
* **Confidence:** Medium
* **Description:** A form is posted from an HTTP page to an HTTPS destination. This can be exploited via a Man-in-the-Middle (MITM) attack, replacing or altering the form submission.
* **Recommended Fix:** Enforce HTTPS site-wide and ensure all forms are submitted securely.

**3. Missing Anti-clickjacking Header**

* **URL:** http://127.0.0.1:5000/login
* **Risk Level:** Medium
* **Confidence:** Medium
* **Description:** The response does not include X-Frame-Options or Content-Security-Policy headers to prevent Clickjacking attacks.
* **Recommended Fix:** Implement the following HTTP headers:
* @app.after\_request
* def set\_headers(response):
* response.headers["X-Frame-Options"] = "DENY"
* response.headers["Content-Security-Policy"] = "frame-ancestors 'none';"

return response

**4. Content Security Policy (CSP) Header Not Set**

* **URL:** http://127.0.0.1:5000/robots.txt
* **Risk Level:** Medium
* **Confidence:** High
* **Description:** The absence of a CSP header increases the risk of Cross-Site Scripting (XSS) and data injection attacks.
* **Recommended Fix:** Implement CSP headers with restrictive policies:

response.headers["Content-Security-Policy"] = "default-src 'self'; script-src 'self';"

**Low Priority Alerts**

**1. Cookie Without SameSite Attribute**

* **URL:** http://127.0.0.1:5000/register
* **Risk Level:** Low
* **Confidence:** Medium
* **Description:** Cookies are missing the SameSite attribute, making them susceptible to CSRF attacks.
* **Recommended Fix:**

response.set\_cookie("session", value, samesite="Strict", secure=True, httponly=True)

**2. Server Leaks Version Information via 'Server' Header**

* **URL:** http://127.0.0.1:5000/static/styles.css
* **Risk Level:** Low
* **Confidence:** High
* **Description:** The web server exposes version information, which can aid attackers in identifying vulnerabilities.
* **Recommended Fix:** Disable the Server header in Flask:
* @app.after\_request
* def remove\_server\_header(response):
* response.headers.pop("Server", None)

return response

**3. X-Content-Type-Options Header Missing**

* **URL:** http://127.0.0.1:5000/static/styles.css
* **Risk Level:** Low
* **Confidence:** Medium
* **Description:** The absence of X-Content-Type-Options: nosniff allows MIME-type sniffing, which can lead to content security risks.
* **Recommended Fix:**

response.headers["X-Content-Type-Options"] = "nosniff"

**Cross-Site Scripting (XSS) Vulnerability Check**

* **Findings:** No XSS vulnerabilities detected. User input is sanitized before rendering.
* **Testing Payload:** <script>alert('XSS')</script>
* **Application Response:** Redirects to an apology template (apology.html).

**SQL Injection Vulnerability Check**

* **Findings:** The application uses parameterized queries, mitigating SQL injection risks.
* **Example Code Review:**

rows = db.execute("SELECT \* FROM users WHERE username = ?", request.form.get("username"))

* **Recommendations:** Continue using parameterized queries and avoid string concatenation in SQL.

**Authentication and Password Security Review**

* **Findings:** Passwords are securely hashed using werkzeug.security.
* **Example Code:**
* from werkzeug.security import check\_password\_hash, generate\_password\_hash

hashPass = generate\_password\_hash(password)

* **Recommendation:** Consider implementing Multi-Factor Authentication (MFA) for enhanced security.

**MFA Implementation Steps**

1. Install necessary packages: pip install pyotp qrcode pillow
2. Generate MFA secret per user:
3. secret = pyotp.random\_base32()

db.execute("UPDATE users SET mfa\_secret = ? WHERE id = ?", secret, user\_id)

1. Validate MFA during login:
2. totp = pyotp.TOTP(user\_mfa\_secret)
3. if not totp.verify(request.form.get("mfa\_code")):

return apology("Invalid MFA Code", 403)

**Security Misconfiguration Fixes**

1. **Secure HTTP Headers:** Use Flask-Talisman to enforce security headers.
2. **HTTPS Enforcement:** Redirect HTTP requests to HTTPS.
3. **Debug Mode Disabled:** Ensure debug=True is not used in production.
4. **Keep Dependencies Updated:** Use pip list --outdated and update packages regularly.

**Conclusion and Recommendations**

**Immediate Fixes Required:**

* Implement CSRF tokens in forms.
* Enforce HTTPS for form submissions.
* Add Clickjacking protection headers.
* Set Content Security Policy (CSP) headers.

**Best Practices to Follow:**

* Maintain secure coding practices.
* Regularly scan and update dependencies.
* Implement Multi-Factor Authentication (MFA).
* Follow OWASP Top 10 security guidelines.

By implementing these recommendations, the security of the website will be significantly improved against common web vulnerabilities.