Task 1: Web Application Security Testing

Objectives :

Conduct security testing on a sample web application to identify vulnerabilities such as:

- SQL injection
- XSS (Cross-Site Scripting)
- Authentication flaws/Bypass

Skills :

- Web application security
- ethical hacking
- penetration testing

Tools :

- **OWASP ZAP -** for automated vulnerability scanning
- **Burp Suite -** for request interception and manual testing
- **SQLMap -** for SQL Injection testing

Introduction :

A Weakness or flow in a Web application that attackers can exploit to steal data, gain access, or disrupt the system. These are Weaknesses or bugs in a web app that hackers can use to harm it. They can steal information, break the system, or stop it from working. Knowing about these problems helps developers and companies keep their apps safe from cyberattacks.

- **Target Machine**
  - Metasploitable 2 (VMware)
- **Sample Web App**
  - DVWA (Damn Vulnerable Web App)

Vulnerability Testing :

▢ SQL Injection Testing

- **Target :** DVWA → SQL Injection module
- **Tool Used :** SQLMap
- **Process :**
  - Intercepted a vulnerable GET request using Burp Suite
  - Saved it to a file dvwa-sqli.txt
  - Ran SQLMap command
- **Result :**
  - Successfully extracted database names:
    - dvwa
    - Information_schema
  - Vulnerability confirmed in parameter id
- **Severity :** High
- **Mitigation :**
  - Use parameterized queries (prepared statements)
  - Apply input validation
  - Avoid concatenating SQL strings directly with user input

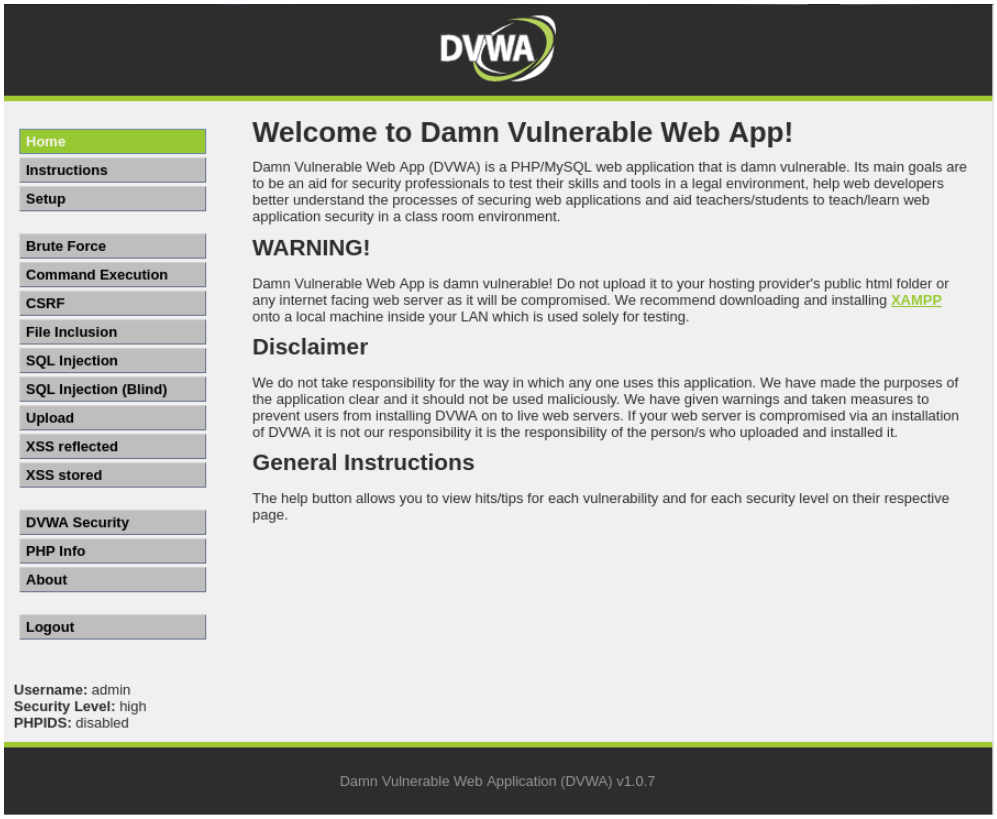
▢ Cross-Site Scripting (XSS) Testing

- **Target :** DVWA → XSS (Reflected) module
- **Tool Used :** OWASP ZAP
- **Process :**
  - Navigate DVWA via browser through ZAP proxy (127.0.0.1:8080)
  - Performed an active scan on /xss_r/
  - ZAP injected payloads like
    - <script>alert(1)</script>
- **Result :**
  - Reflected XSS detected in name parameter
  - Alert triggered successfully
- **Severity :** Medium to High
- **Mitigation :**
  - Sanitize and encode all user input
  - Use Content Security Policy (CSP)
  - Avoid rendering raw input into HTML without escaping

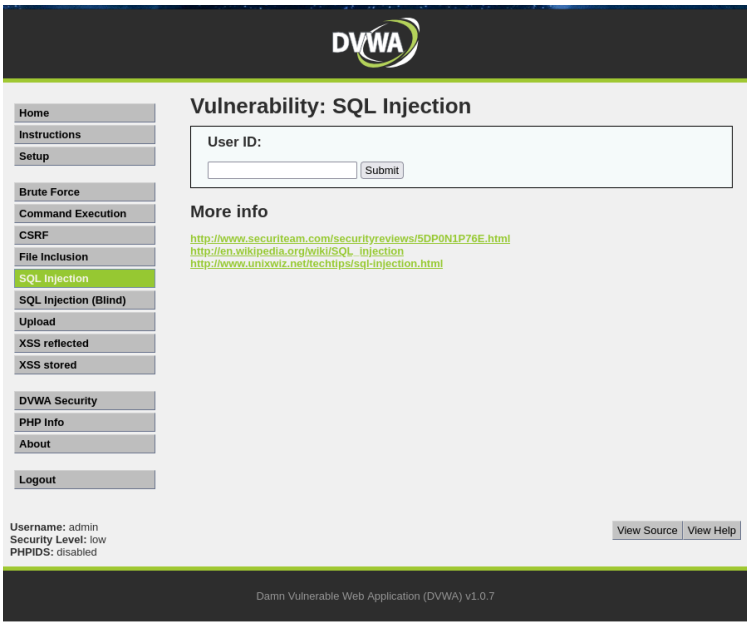
▢ Authentication Bypass Testing

- **Target :** DVWA login page
- **Tool Used :** Manual testing + Burp Suite + Hydra (optional)
- **Process :**
  - Attempted SQL-based payloads
  - Access granted without valid credentials
  - Brute force simulation using Hydra with rockyou.txt
- **Result :**
  - Authentication bypass confirmed
  - Weak login handling logic exposed
- **Severity :** High
- **Mitigation :**
  - Validate login inputs and responses
  - Implemented account lockout mechanisms
  - Use CAPTCHA for brute force protection
  - Enforce strong password policies

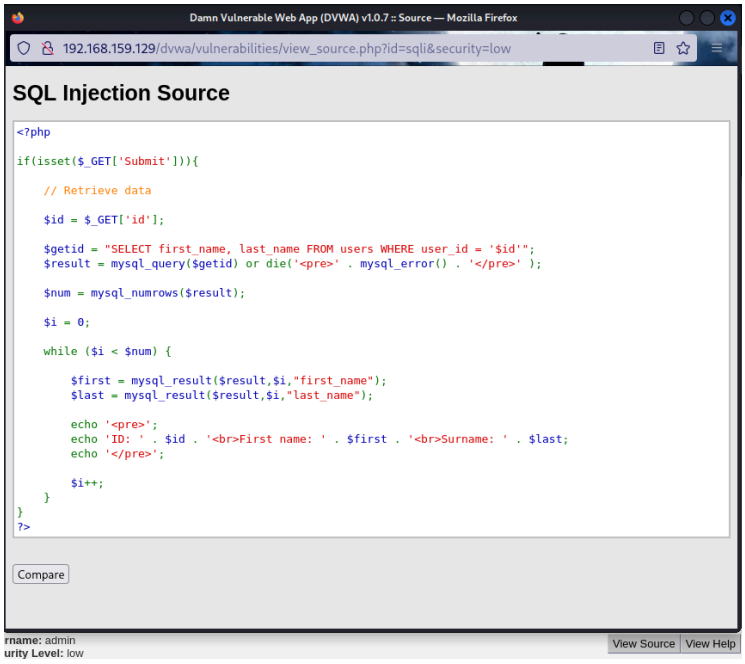
Now we Start attempting Attack on our Vulnerable Machine



Go to SQL injection section for getting access for databases which is a vulnerable which connects to Mysql server and bypassing the authenticating for extracting data from db



If we wanna see source code how db go to mysql to simply click on right side corner **View Source** then we see our source how the data sql query work there.



Here we see the sql query is

- “SELECT first_name, last_name FROM users_id = ‘id’”;

In this query we perform our SQL Injection attacks.

Here we put digit to check how the query working.

Now we use sqlmap to perform SQL Injection attacks.

- Used sqlmap -u "http://192.168.159.129/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit#" --cookies="security=low; PHPSESSID=cookie" --batch. Cookies can be easily found by inspecting page source and going to storage.

Now we Added -tables and -D dvwa to query.

Now we added -T users and --dump to see in users data deeply and all dump all database in the user file.

Here we get credentials with the help of sqlmap to perform SQL Injection attack.

Conclusion :

1. Use Prepared Statements (Parameterized Queries)

- Always use parameterized queries in PHP (PDO/MySQLi), Java (JDBC), Python (SQLite/MySQL), etc.
- This separates SQL code from user input, preventing injection.

2. Input Validation

- Validate all user inputs:
  - Ensure correct data type (e.g., integer, email).
  - Use whitelists for allowed input.
  - Reject unexpected or special characters where not needed.

3. Use ORM Libraries

- Use frameworks or Object-Relational Mapping tools (e.g., SQLAlchemy, Hibernate, Entity Framework) that automatically handle query escaping.

4. Least Privilege Access

- The database user used by your app should have minimal permissions:
  - Avoid using root or admin-level accounts.
  - Only allow the actions (SELECT/INSERT) that the app needs.

5. Avoid Dynamic SQL

- Do not build queries by concatenating user input.

6. Web Application Firewall (WAF)

- Deploy a WAF (e.g., ModSecurity) to help detect and block SQLi attempts in HTTP traffic.

7. Error Handling

- Disable detailed SQL error messages in production.
- Use generic error responses like: "Something went wrong. Please try again later."

8. Regular Security Testing

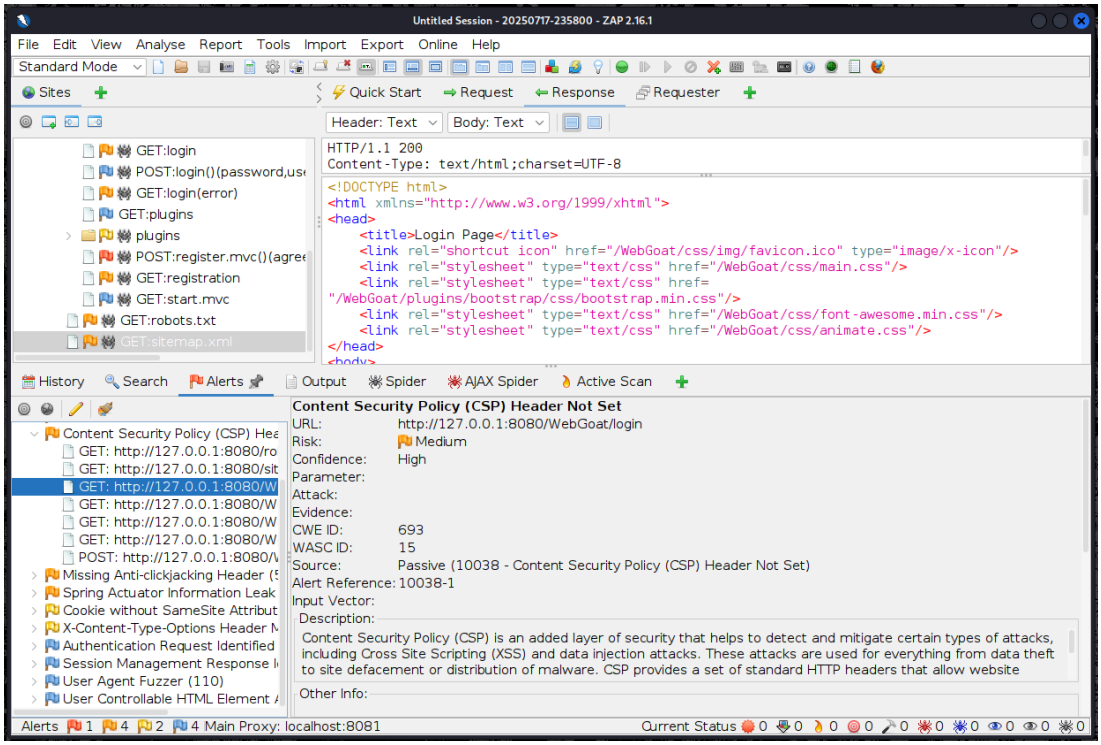
- Use tools like:
  - sqlmap
  - OWASP ZAP
  - Burp Suite
- Perform regular pen tests and code reviews.

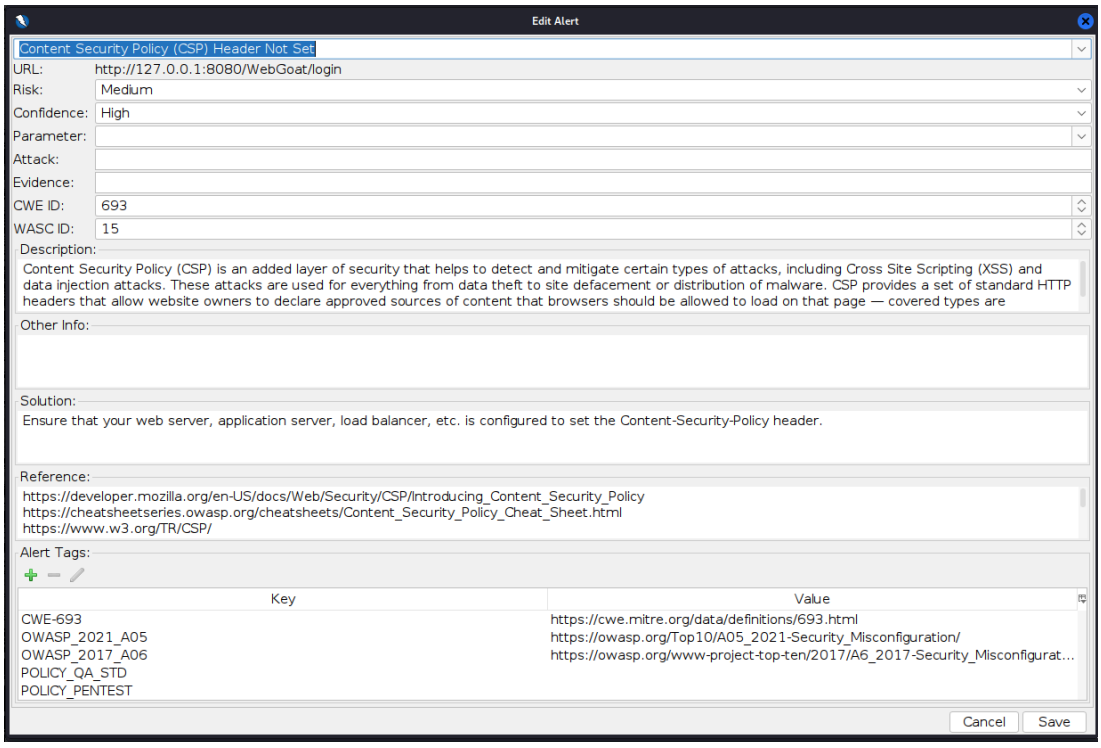
9. Keep Software Updated

- Keep your DBMS, libraries, and frameworks up to date with the latest security patches.

XSS : (Cross Site Scripting)

Open the OWASP ZAP and copy the URL of the WebGoat login page and paste in the Automated Scan URL header option and start the scan.





Content Security Policy (CSP) Header Not Set :

Content Security Policy (CSP) is missing, which limits sources from which content (scripts, images, etc.) can be loaded. Without CSP, the application is more vulnerable to attacks like **Cross-Site Scripting (XSS)**, where attackers can inject malicious scripts that get executed in the user's browser.

Consequence of XSS :

Session Hijacking

- Attackers steal session cookies to impersonate users, gaining unauthorized access to accounts.

Credential Theft

- XSS can be used to trick users into entering login details on fake pages.

Data Theft

- Sensitive information on the page can be accessed and stolen by attackers.

Malware Distribution

- Malicious code injected through XSS can redirect users to phishing sites or download malware.

Phishing and Social Engineering

- XSS can display fake interfaces, tricking users into revealing personal information.

Defacement of Websites

- Attackers can alter the apperance of web pages, damaging credibility and reputation.

Denial of Service

- XSS may overload the server, making the site inaccessible to legitimate users.

Impacts on SEO and Reputation

- Hidden spam links or malware can harm a site's SEO and get it blacklisted.

Prevention:

1. Sanitize and validate user input on both the client and server sides.
2. Use Content Security Policy (CSP) to limit where scripts can be loaded from.
3. Encode output (especially when rendering user-generated content in HTML).
4. Use secure headers such as X-Content-Type-Options, X-Frame-Options, and X-XSS-Protection.

Conclusion

- Through this task, I gained hands-on experience
- Identifying and exploring web app vulnerabilities
- Using professional tools for penetration testing
- Preparing detailed technical documentation

The project emphasizes the critical need for secure coding practices in web development. All discovered vulnerabilities have been documented with the recommendations for mitigation.

Acknowledgment

I would like to thank **Future Interns** for their guidance during this task and for providing me access to real-world tools and scenarios to enhance my cybersecurity skills.

Name :- Devansh Namdev
Linkedin:-https://www.linkedin.com/in/devansh-namdev-9765701bb

Date:- 1 Aug 2025