**GROUP 3**

|  |  |  |
| --- | --- | --- |
| **Member** | **Role** | **Responsibilities** |
| 1. Agbajeola Bisola  INT250448 | Project Lead & Report Coordinator | - Oversaw progress and coordination  - Ensured all parts are consistent and compiled  - Wrote Executive Summary and Conclusion of report  - Managed submission deadlines |
| 2. Adaobi Ogbonna INT250073 | Environment Setup & Tooling | - Set up vulnerable web application (DVWA)  - Configured testing tools (Docker & Ngrok)  - Documented setup steps |
| 3. Balogun Fawaz Aniekan INT250050 | Penetration Tester | - Tested for and document OWASP Top 10 issues: \  - Provided screenshots, test methods, and secure SDLC practices |
| 4. Jedidiah Abel  INT250298 | Security Analyst | - Suggest secure coding practices for each vulnerability- Map all fixes to Secure SDLC and **NIST SP 800-64** phases- Write recommendations section of the report |
| 5. Amina bouchenafa INT250092 | Presenter & Slide Designer | - Designed and created the presentation slides  - Summarized key findings visually  - Presented the task |

# ****Penetration Testing of DVWA Web Application****

**Executive Summary:**

This report presents the results of a penetration test conducted on the Damn Vulnerable Web Application (DVWA). The primary objective was to identify and exploit OWASP Top 10 vulnerabilities and recommend secure coding practices based on findings. The team successfully set up a DVWA instance using Docker and made it accessible externally via Ngrok. Using tools such as Burp Suite and a Kali Linux testing environment, the team uncovered vulnerabilities including SQL Injection, Brute Force attacks, Reflected Cross-Site Scripting (XSS), Command Injection, and Unrestricted File Uploads. Each finding was analyzed with practical exploit evidence, mapped to secure SDLC phases in alignment with NIST SP 800-64, and supported with actionable recommendations.

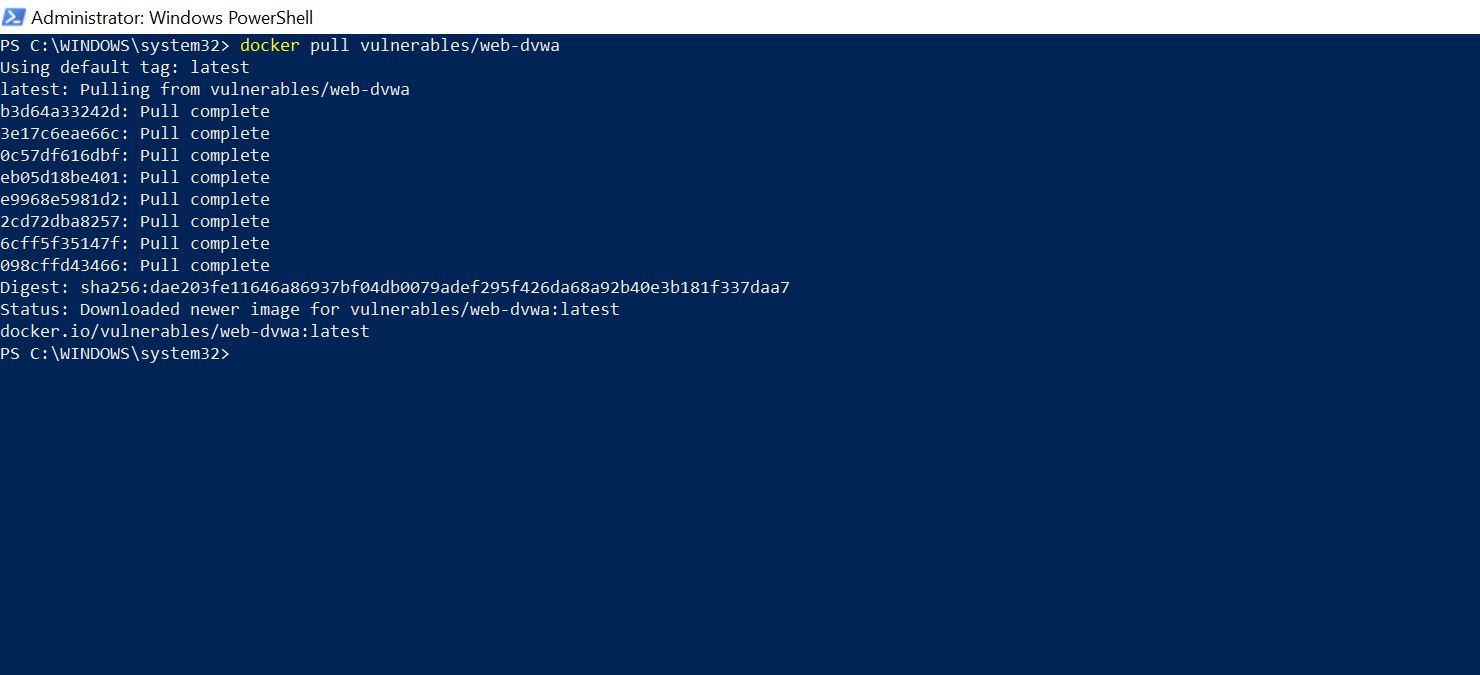
# DVWA Setup:

# Part 1: Setup Documentation

This section documents the step-by-step process taken to set up the Damn Vulnerable Web Application (DVWA) on a Windows host system using Docker and Ngrok for external access.

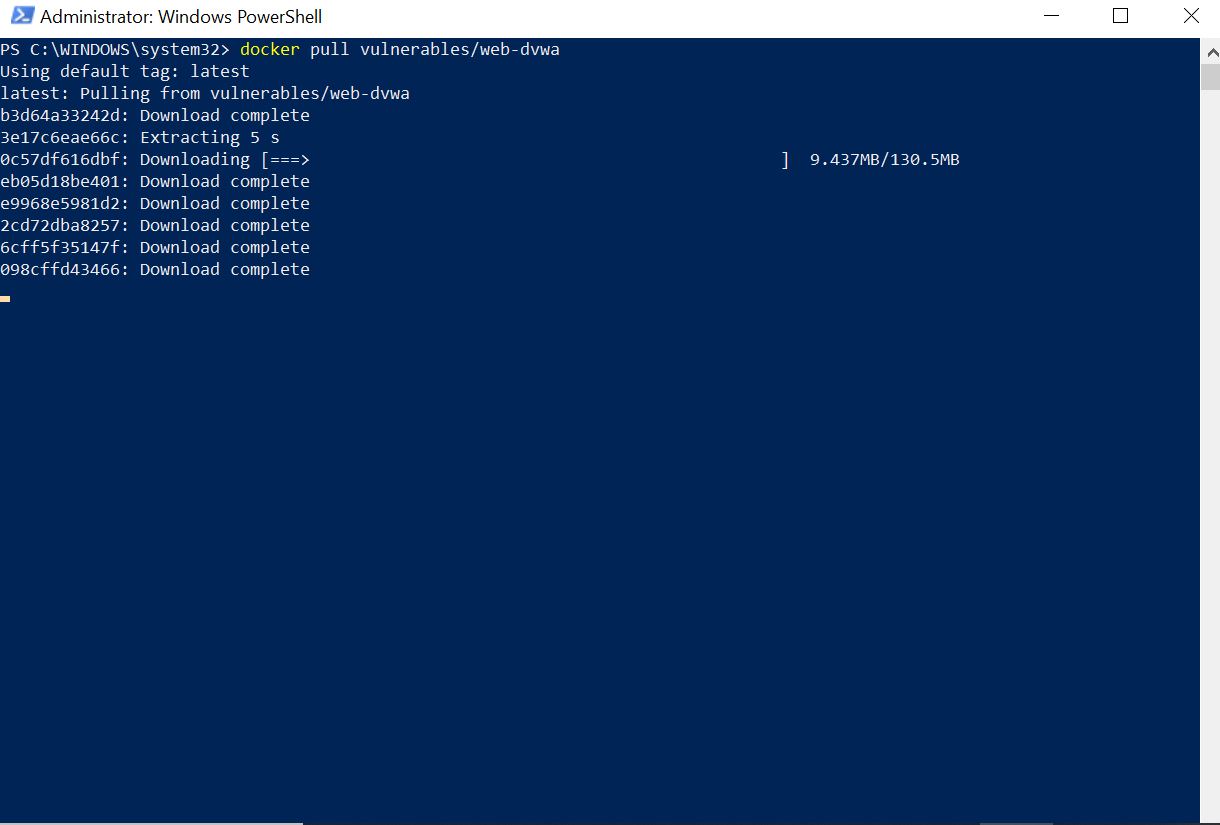
## Step 1: Install Docker Desktop

Downloaded Docker Desktop from *https://www.docker.com/products/docker-desktop* and installed it with the recommended settings (WSL2 enabled).



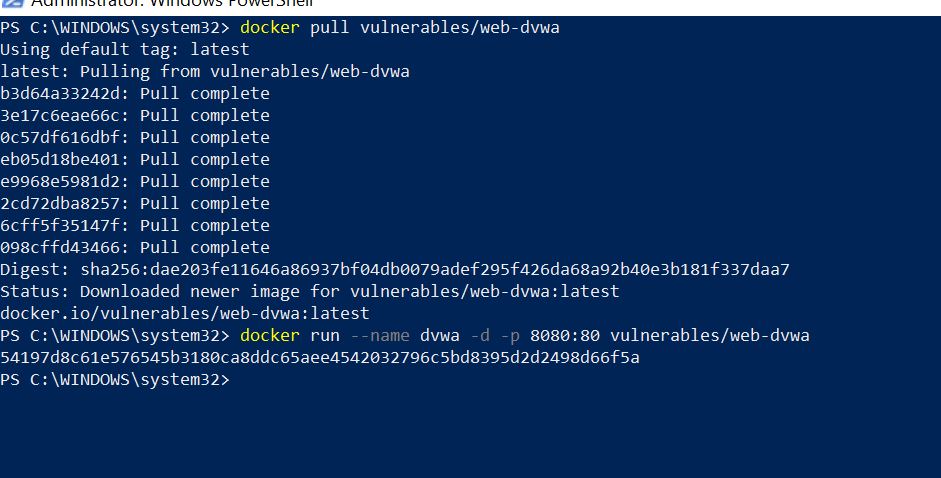
## Step 2: Pull DVWA Docker Image

Opened PowerShell and ran the following command to download the DVWA image:  
 “ *docker pull vulnerables/web-dvwa”*



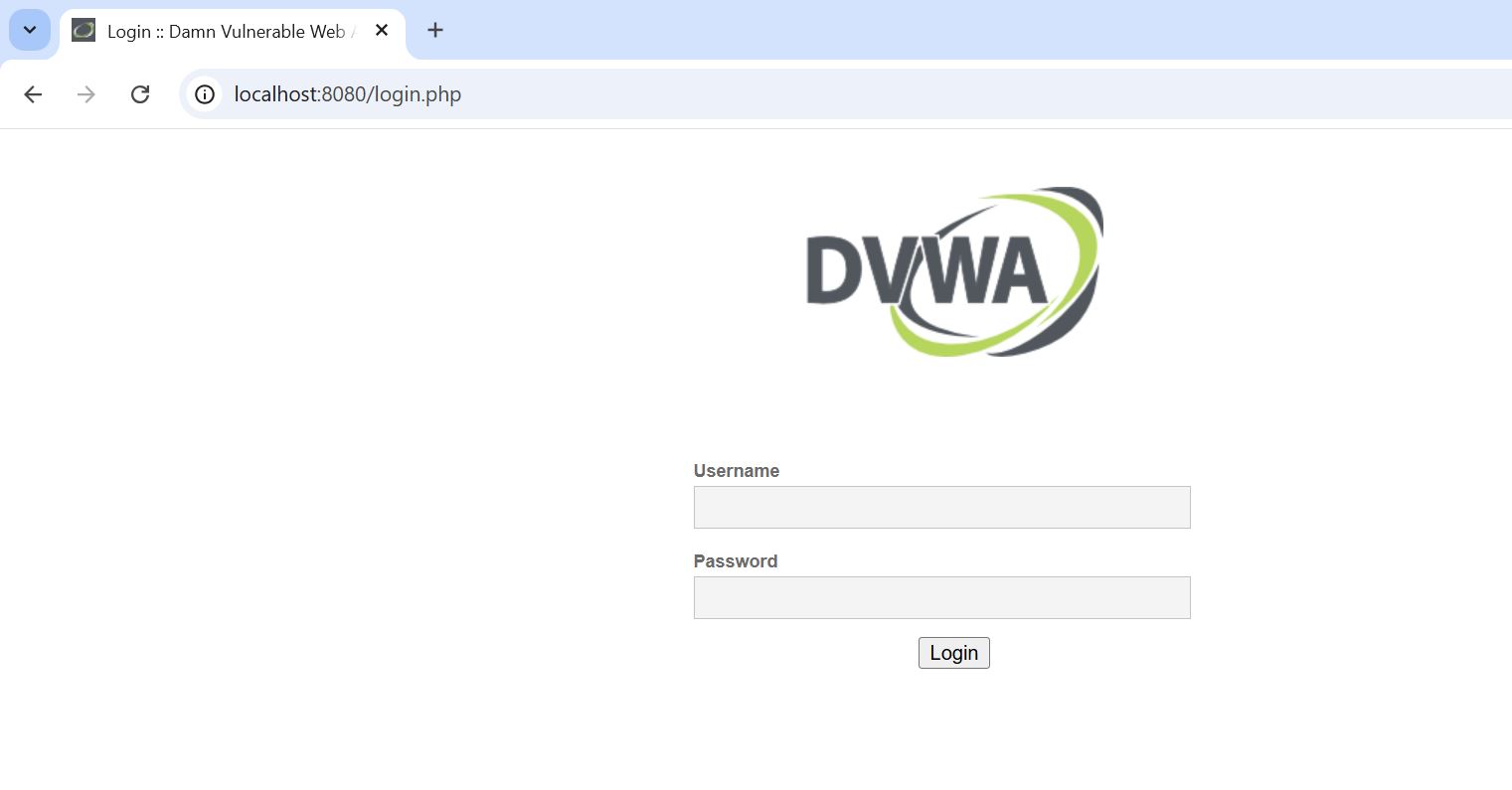
## Step 3: Run DVWA Container

Used the following command to run the DVWA container (depending on available port):  
 *“ docker run -d --name dvwa -p 80:80 vulnerables/web-dvwa”*



## Step 4: Access DVWA Locally

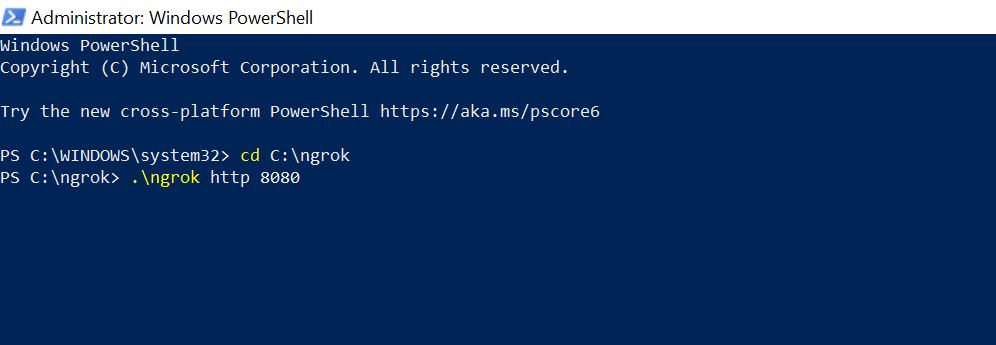
Opened a browser and navigated to *http://localhost or http://localhost:8080*.

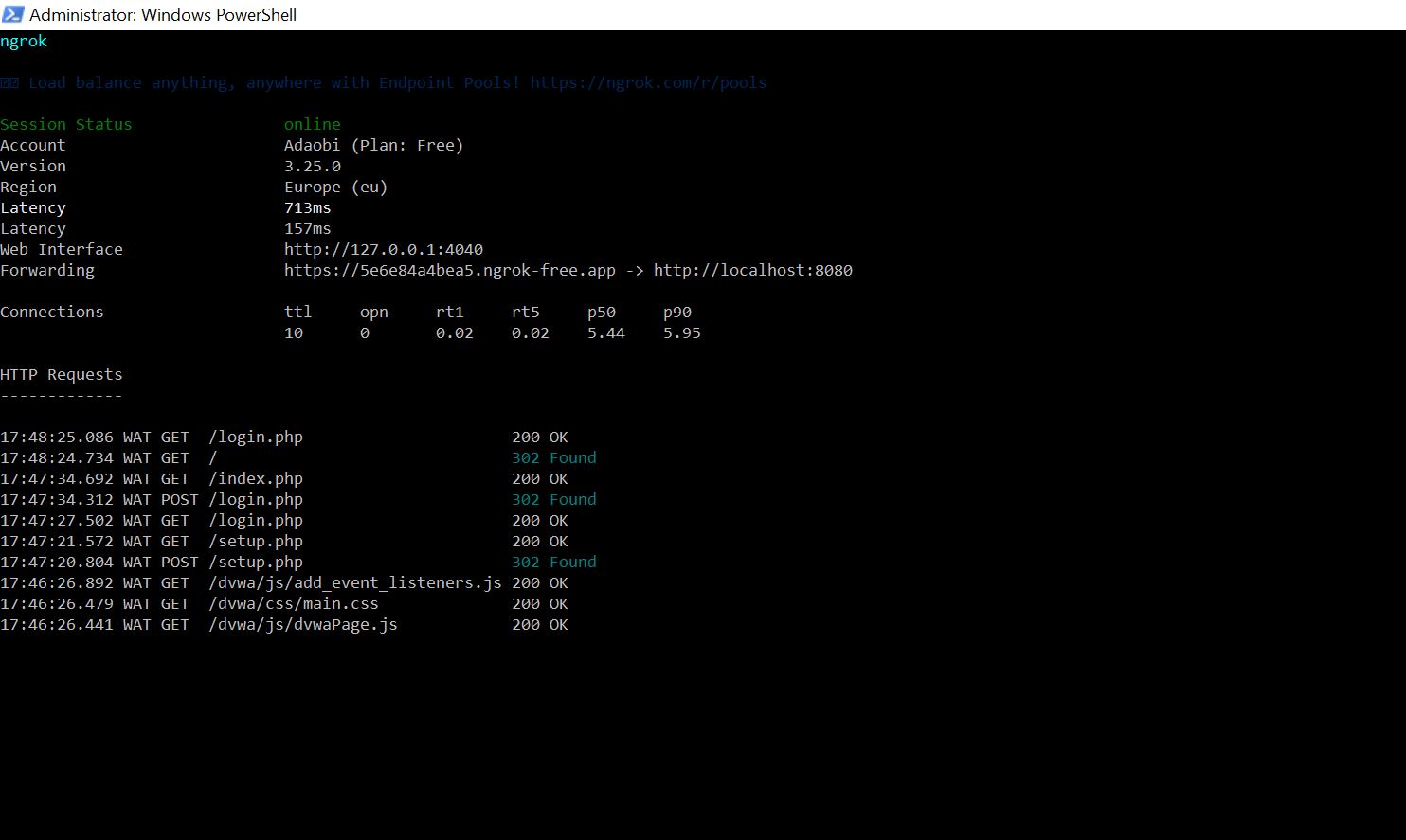


## Step 5: Install & Configure Ngrok

Navigated to /setup.php and clicked "Create / Reset Database".  
Logged in using:  
 Username: admin  
 Password: password  
Set DVWA Security Level to: Low

Downloaded Ngrok from https://ngrok.com/. Extracted ngrok.exe to C:\ngrok.  
Ran the following to connect Ngrok to my account:  
 *“ .\ngrok config add-authtoken <MY\_AUTH\_TOKEN>*”  
Then started the tunnel with:  
 .\ngrok http 80  
Copied the public forwarding link from the terminal and shared with the team so that the pentester can start working on the webapp.





# Web Application Pentesting– DVWA Testing

# 1. Testing Environment

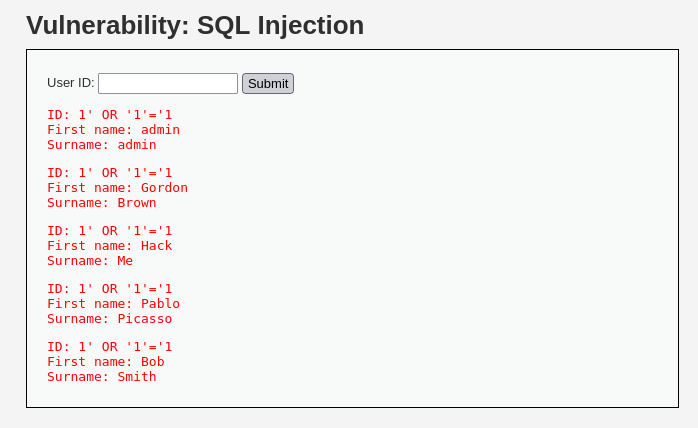
- Operating System: Kali Linux  
- Application: DVWA (Damn Vulnerable Web Application)  
- Tools Used: Burp Suite, Firefox  
- DVWA Security Level: Low

# 2. Vulnerability Assessments

## 2.1 SQL Injection (OWASP A03:2021)

Using the payload `1' OR '1'='1`, SQL Injection was successfully executed in the User ID field. This exposed sensitive user information by manipulating the backend query.

Evidence of SQL Injection:



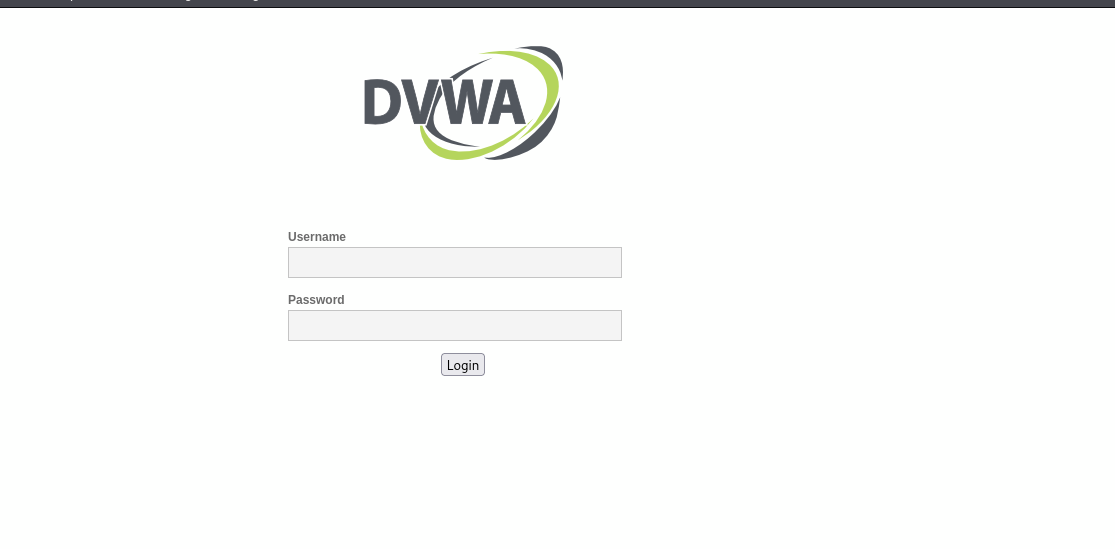
Source code showing vulnerability:



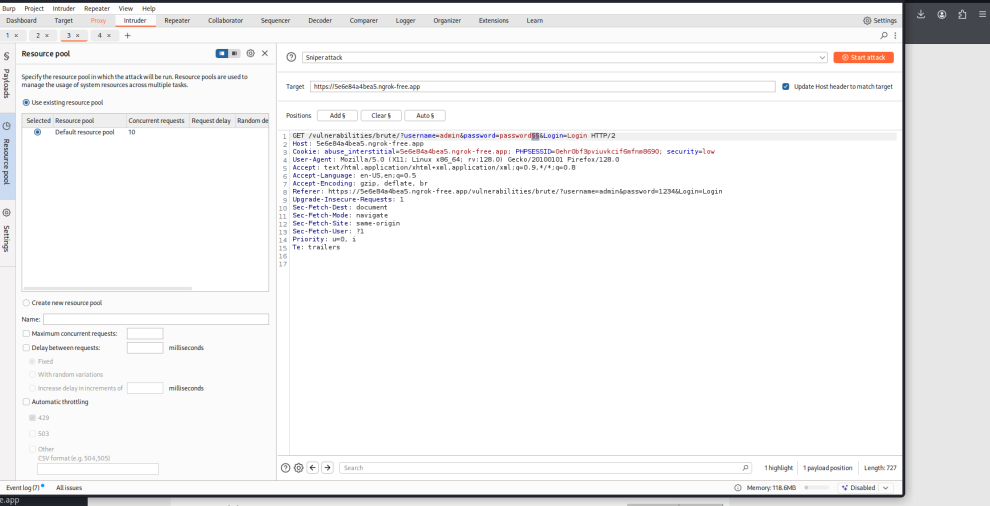
## 2.2 Brute Force (OWASP A07:2021)

A brute force attack was conducted using Burp Suite's Intruder module on the login form. By testing common passwords, access to the admin account was successfully achieved.

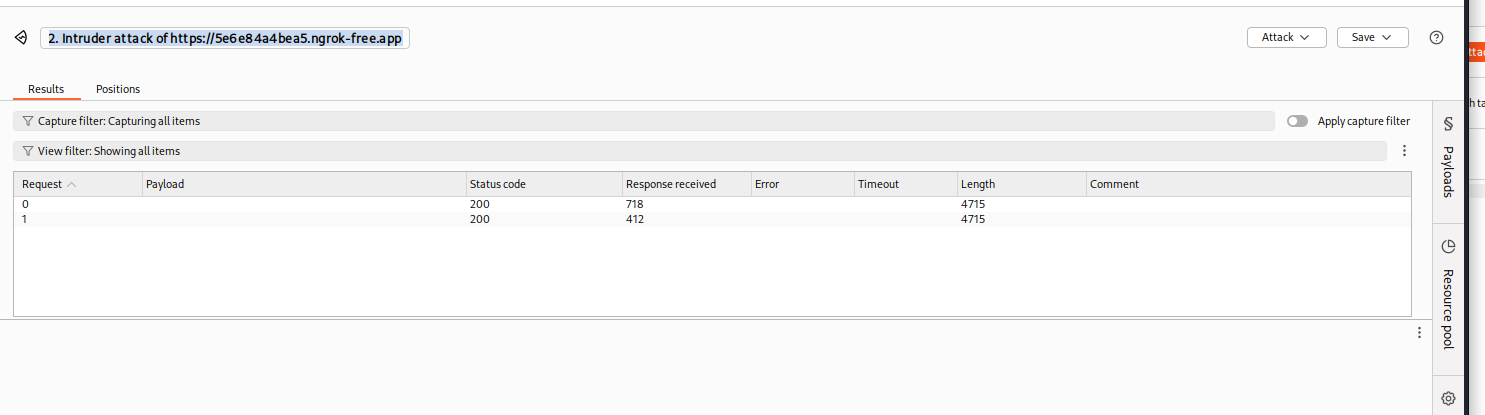
Login form captured:



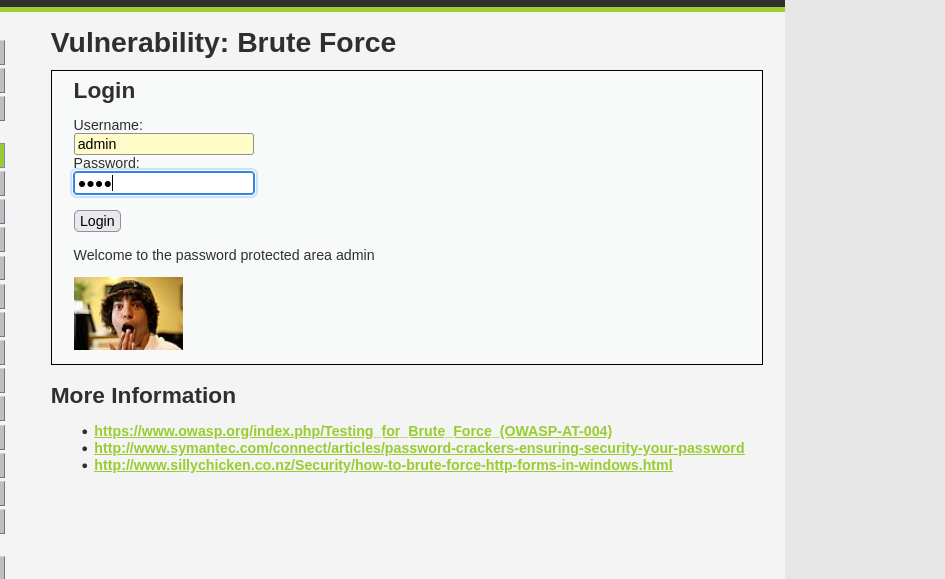
Burp Suite Intruder setup:



Intruder attack results:



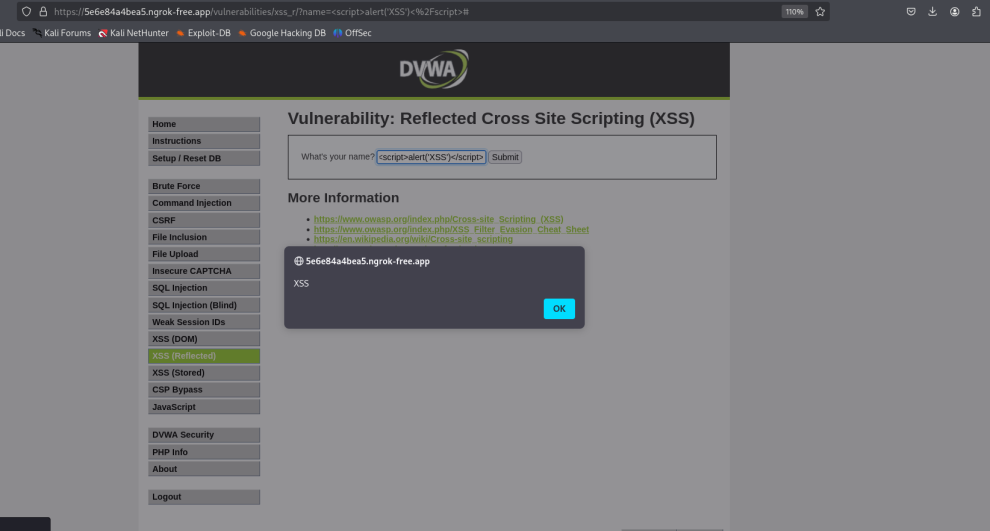
Successful login confirmation:



## 2.3 Cross-Site Scripting (XSS) – Reflected (OWASP A03:2021)

A simple `<script>alert('XSS')</script>` payload triggered a reflected XSS vulnerability in the input field, demonstrating that unescaped input was executed by the browser.

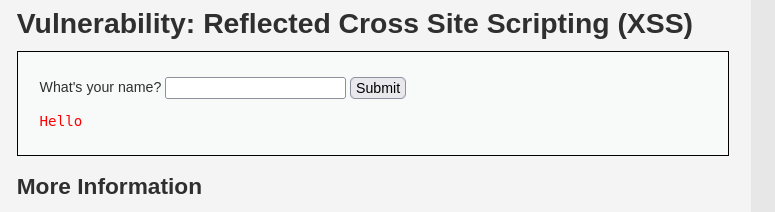
XSS payload executed:



Source code (no output encoding):



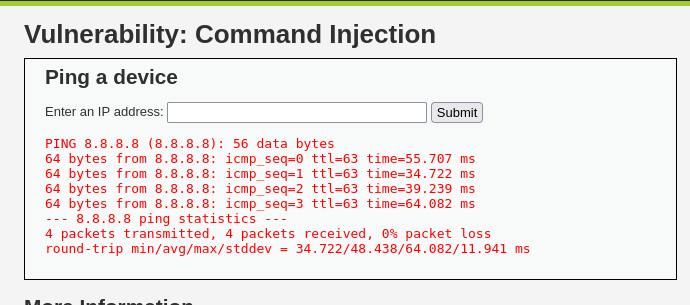
Rendered HTML response:



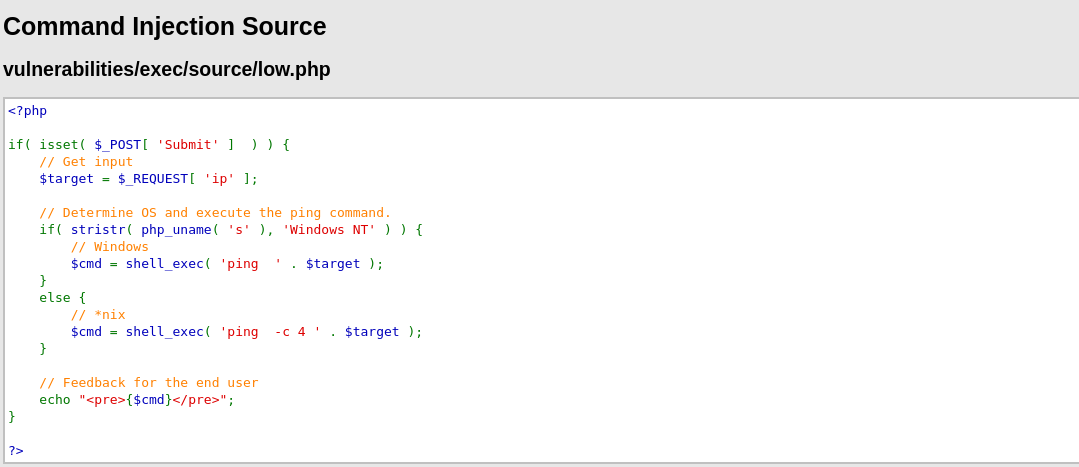
## 2.4 Command Injection (OWASP A01:2021)

The 'Ping a device' feature was vulnerable to command injection due to direct input execution via shell commands. The input was not sanitized, enabling potential arbitrary command execution.

Ping output shown (vulnerable):



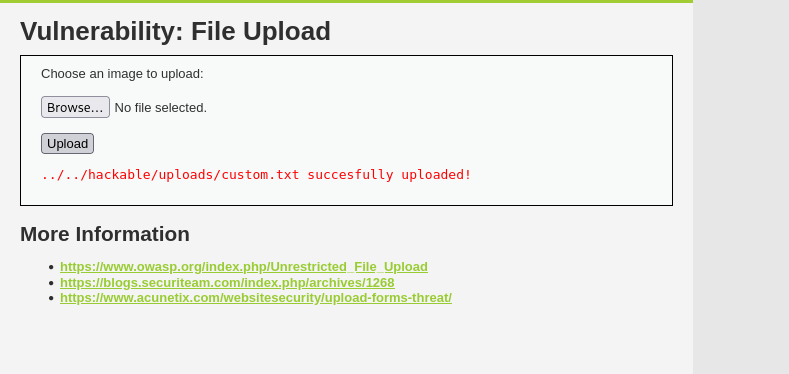
Source code with shell\_exec():



## 2.5 Unrestricted File Upload (OWASP A08:2021)

DVWA allowed upload of files such as `.txt` without validation. In a real system, this could enable remote code execution by uploading malicious scripts.

Successful file upload message:



Source code with no validation:



# Secure Coding Practices & SDLC Mapping

**2.1 SQL Injection**

* Secure Coding Practices:
  + Always use prepared statements/parameterized queries.
  + Implement server-side input validation (e.g., regex or allowlists).
  + Apply principle of least privilege on database accounts.
* SDLC Mapping:
  + Requirements (NIST Phase 1): Define acceptable input types and SQL handling requirements.
  + Development (NIST Phase 2): Implement parameterized queries using frameworks like PDO (PHP) or ORM tools.
  + Testing (NIST Phase 3): Conduct dynamic scans and unit tests for input handling.
  + Deployment & Operations (NIST Phase 4): Restrict Database access and enable logging of suspicious queries.

**2.2 Brute Force**

* Secure Coding Practices:
  + Add account lockout mechanisms after repeated failed attempts.
  + Use CAPTCHA to prevent automation.
  + Apply rate-limiting and Multi-Factor Authentication (MFA).
* SDLC Mapping:
  + Requirements: Identify authentication flow and set password policies.
  + Design: Integrate CAPTCHA and MFA early in design.
  + Development: Use libraries for rate-limiting and login attempt tracking.
  + Testing: Simulate brute-force attempts using automated tools like Burp Suite.
  + Operations: Monitor failed login attempts and implement alerts.

**2.3 Cross-Site Scripting (XSS)**

* Secure Coding Practices:
  + Sanitize all user input and apply output encoding based on context (HTML, JS, etc.).
  + Use security libraries that automatically escape data.
  + Set Content Security Policy (CSP) headers.
* SDLC Mapping:
  + Requirements: Define output contexts and escape rules.
  + Development: Use secure templating engines (e.g., Twig, Handlebars).
  + Testing: Use scanners (e.g., OWASP ZAP) to detect XSS.
  + Deployment: Configure CSP and HTTP security headers.

**2.4 Command Injection**

* Secure Coding Practices:
  + Whitelist acceptable inputs and strictly validate user data.
  + Avoid using shell\_exec() or similar commands—use safe APIs instead.
  + Escape any user input used in command-line contexts.
* SDLC Mapping:
  + Requirements: Document commands that need execution and alternatives.
  + Design: Favor internal APIs or service abstraction over shell commands.
  + Development: Remove direct shell execution unless unavoidable.
  + Testing: Test with payloads (; ls, &&whoami) to detect injection vectors.
  + Deployment: Harden the execution environment (e.g., no shell access for app user).

**3.5 Unrestricted File Upload**

* Secure Coding Practices:
  + Restrict accepted file types and MIME types.
  + Rename and sanitize filenames to prevent path traversal.
  + Store files outside the web root and remove execution permissions.
  + Perform server-side virus scanning.
* SDLC Mapping:
  + Requirements: Specify upload policy including max size and file types.
  + Design: Isolate upload storage from executable paths.
  + Development: Use file validation libraries or built-in server features.
  + Testing: Upload test payloads (e.g., .php, .js) and analyze behavior.
  + Deployment: Apply OS-level protections (e.g., noexec on upload directories).

# Recommendations

1. **Implement Secure Coding Standards**: All developers should follow OWASP secure coding practices. Regular code reviews and security linting should be introduced.
2. **Adopt a Secure SDLC Framework**  
   Integrate security at every phase of SDLC:
   * **Planning:** Include threat modeling in project initiation.
   * **Design:** Choose secure architectural patterns.
   * **Development:** Enforce secure coding guidelines and peer reviews.
   * **Testing:** Automate security testing and perform regular penetration tests.
   * **Deployment & Maintenance:** Apply security configurations, monitor logs, and patch regularly.
3. **Use Automated Security Tools**  
   Integrate tools like:
   * **SAST** (Static Application Security Testing) for code analysis.
   * **DAST** (Dynamic Application Security Testing) for runtime testing.
   * **Dependency Scanners** (e.g., OWASP Dependency-Check) to monitor third-party libraries.
4. **Continuous Security Training**: Conduct recurring secure coding training and security awareness programs for developers and QA staff.
5. **Compliance and Policy Enforcement**: Align with **NIST SP 800-64** and **OWASP ASVS** for process maturity and technical coverage.

**Conclusion**

The assessment confirmed that DVWA, when configured at a low security level, exposes numerous web application vulnerabilities consistent with the OWASP Top 10. Exploiting these flaws provided hands-on insight into how attackers can manipulate poorly secured applications. The report recommends implementing a secure software development life cycle (SDLC) that integrates robust coding practices, regular testing, and post-deployment monitoring. Emphasizing secure design, parameter validation, input/output encoding, and least privilege access is crucial. Organizations that fail to adopt such measures remain exposed to avoidable and potentially damaging cyber threats. This project reinforces the value of combining penetration testing with structured security frameworks like **NIST SP 800-64** to build safer applications from the ground up.