Student Name: Muritala Ajarat Abiodun

Student ID: 2025/INT/8475

Course Name: Web Application Security

Course Code: INT 307

Assignment Title: File Upload, Security Misconfiguration, and IDOR Vulnerabilities

Instructor Name: Mr Rantu Sakar

Date of Submission: 31/07/2025

Lab 4: File Upload Vulnerabilities in DVWA

Overview

In this lab, you will explore file upload vulnerabilities within the Damn Vulnerable Web Application (DVWA). File upload vulnerabilities occur when an application allows users to upload files without proper validation, enabling attackers to upload malicious files that can compromise the system. You will utilize Weevely to generate a PHP reverse shell and Burp Suite to manipulate requests and bypass validation checks across different security levels (Low, Medium, and High).

Prerequisites

- Ensure you have DVWA installed and running.
- Familiarize yourself with Weevely and its functionality.
- Basic understanding of Burp Suite and how to intercept and modify HTTP requests.

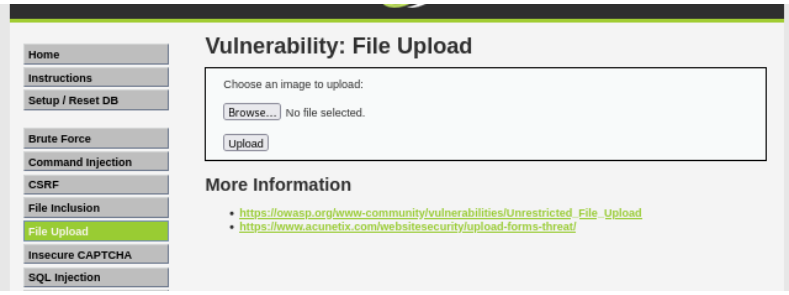
Exercise Instructions

Exercise 1: Exploring the File Upload Page

Objective: Understand the file upload functionality in DVWA.

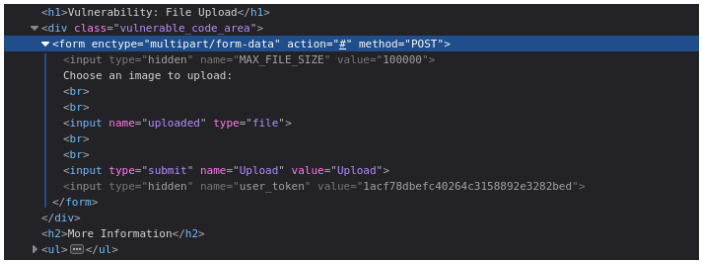
1. Access the File Upload Page: Navigate to the following URL in your DVWA instance:

- File Upload



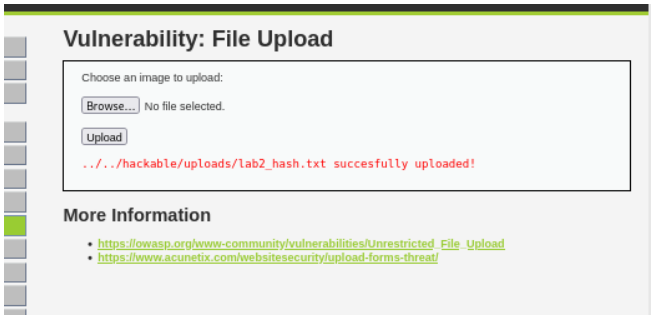
2. Analyze the Upload Form:

- Review the form provided for file uploads. Identify the input fields and their types.



3. Upload a Non-Malicious File:

- Select a harmless file (e.g., a text file named test.txt) and upload it to the server.
- Observe the response and verify that the file was uploaded successfully.



4. Reflection:

- **What are the characteristics of the file upload feature? Discuss any visible restrictions (if any) on file types and sizes.**

Characteristics:

- Upload feature accepts any file type (no visible restriction).
- File saved to /hackable/uploads/.
- Displays file path in the success message.

Restrictions:

- The form has a hidden MAX_FILE_SIZE value of 100000 bytes (~100 KB).
- No file type enforcement visible (didn't block .txt).

Exercise 2: Generate a PHP Shell Using Weevely

Objective: Create a malicious file using Weevely.

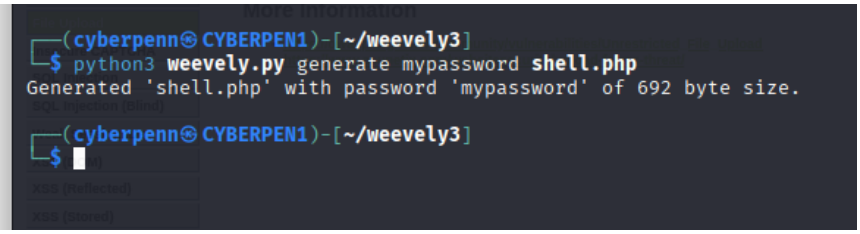
1. Install Weevely (if not already installed):

- Open a terminal and clone the Weevely repository:

2. git clone https://github.com/epinna/weevely3.git cd weevely3

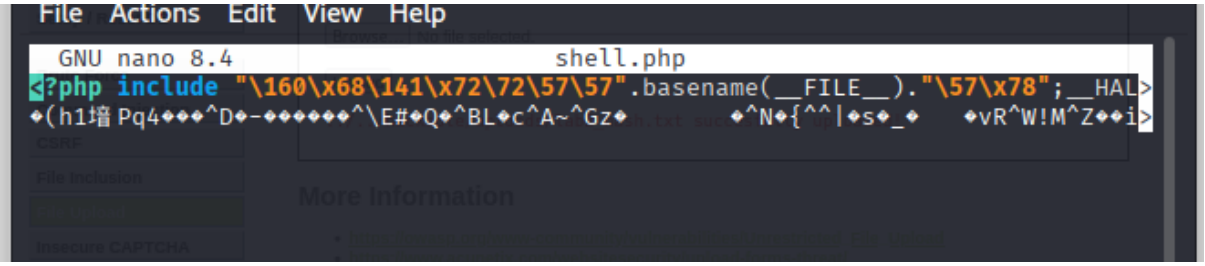
3. Generate a PHP Shell:

- Run Weevely to create a PHP reverse shell: python weevely3.py generate your_password shell.php
- Replace your_password with a secure password. This will create a file named shell.php.



4. Review the Shell Code:

- Open shell.php in a text editor and observe the code generated by Weevely.



Reflection Notes:

- Weevely generates an obfuscated PHP payload to help evade detection.
- The shell file itself does nothing visible when opened in a browser — it waits for the Weevely client to connect using the chosen password.
- Key takeaway: If this shell is uploaded and executed on DVWA, you'll have remote access via the Weevely too

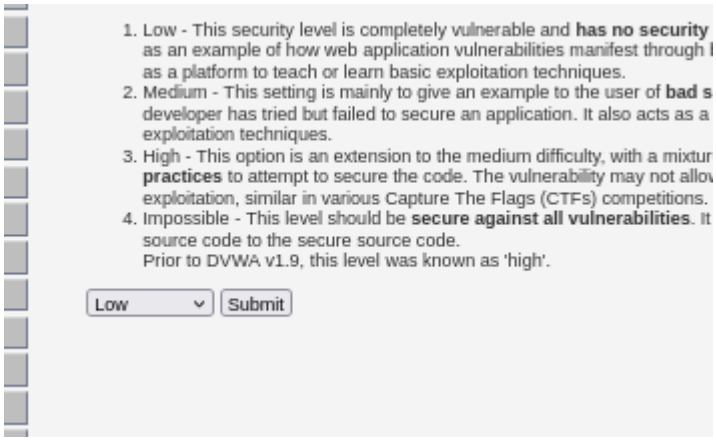
Exercise 3: Upload the Malicious Shell at Different Security Levels

Objective: Identify and exploit a file upload vulnerability by uploading the generated shell at various security levels.

Step 1: Low Security Level

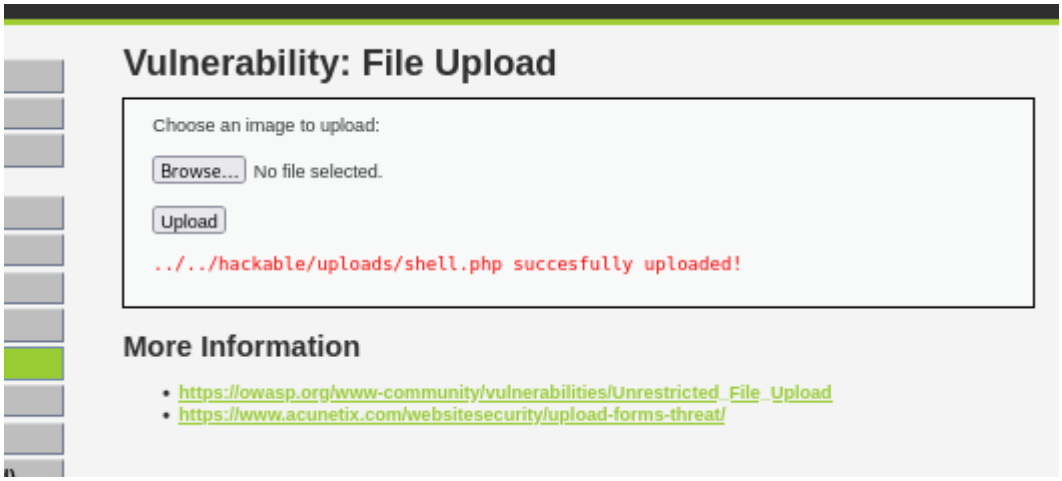
1. Set Security Level to Low:

- In DVWA, navigate to the security settings and set the security level to Low.



2. Upload the Malicious Shell:

- Go back to the file upload page.
- Use the upload form to upload the shell.php file created by Weevely.
- Observe if the upload is successful.



3. Reflection:

- **Were you able to upload the malicious file? Discuss the implications of this vulnerability.**

Reflection

Were you able to upload the malicious file?

- Yes. DVWA's Low Security setting applied no restrictions — the shell.php file was uploaded directly and stored in /hackable/uploads/.

Implications of this vulnerability:

- Because the application did not validate file type or extension, an attacker could upload any PHP script — including malicious shells.

- Once uploaded, the attacker can execute commands on the server by connecting to the shell (e.g., using Weevely), which gives remote code execution and full system compromise.
- In a real-world scenario, this could lead to:
    - Website defacement
    - Data theft
    - Privilege escalation on the server
    - Use of the compromised server to pivot into other systems

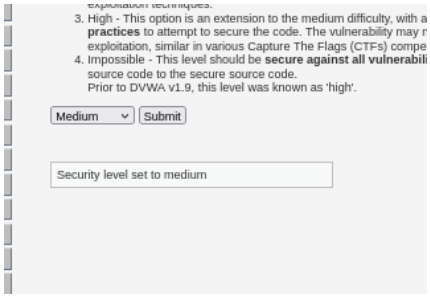
Summary:

The malicious shell.php uploaded successfully. This demonstrates a critical file upload vulnerability: without file validation, attackers can upload web shells, leading to complete server compromise.

Step 2: Medium Security Level

1. Set Security Level to Medium:

- Change the security level to Medium in DVWA.



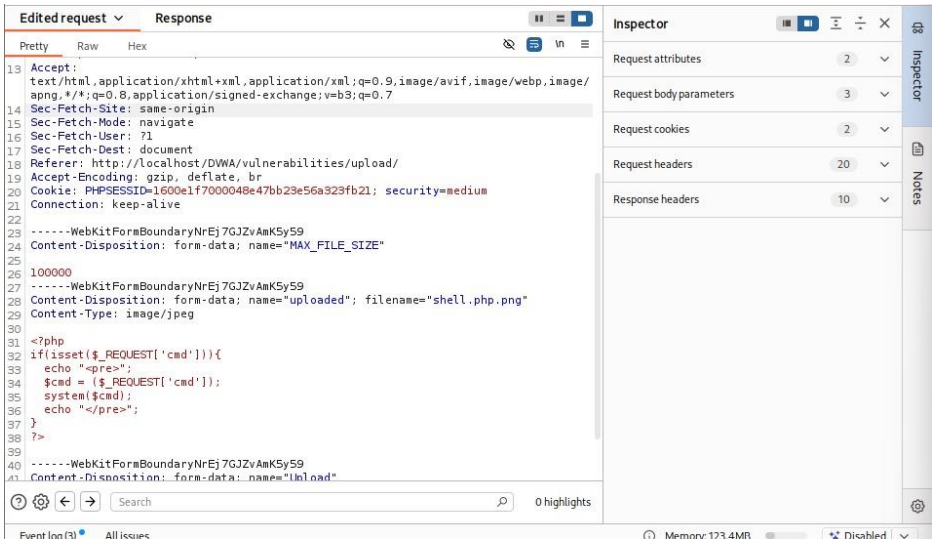
2. Upload Attempt:

- Try uploading the shell.php file again.
- If the upload fails due to validation, proceed to use Burp Suite to bypass validation.

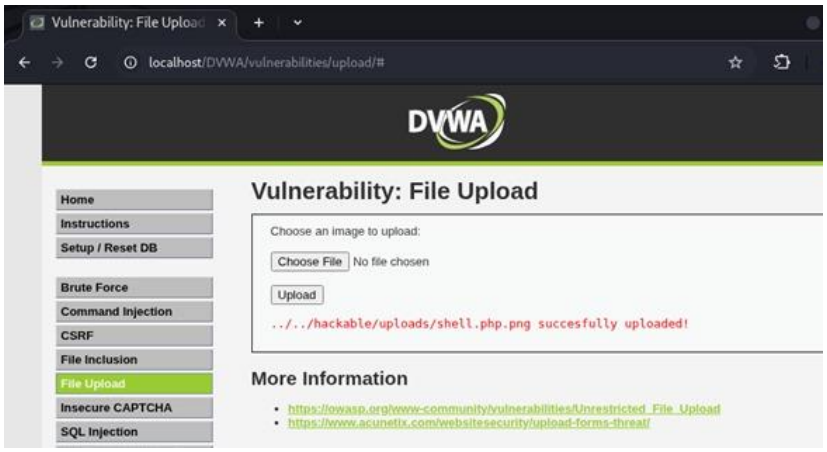
3. Bypassing Validation with Burp Suite:

- Start Burp Suite and configure your browser to use Burp as a proxy.

- Intercept the upload request when you attempt to upload the file.
- Modify the request to change the file type in the headers (e.g., change the content type to image/jpeg).



- Forward the request to the server and observe the response.



4. **Reflection:**

- **Were you able to bypass the restrictions? What does this indicate about the application's security?**

Yes, I was able to bypass the upload restrictions. By intercepting the request in Burp Suite, I changed the file name from `shell.php` to `shell.php.jpg` and modified the `Content-Type` header from `application/x-php` to `image/jpeg`. DVWA accepted the file and stored it in the upload’s directory, even though it still contained executable PHP code.

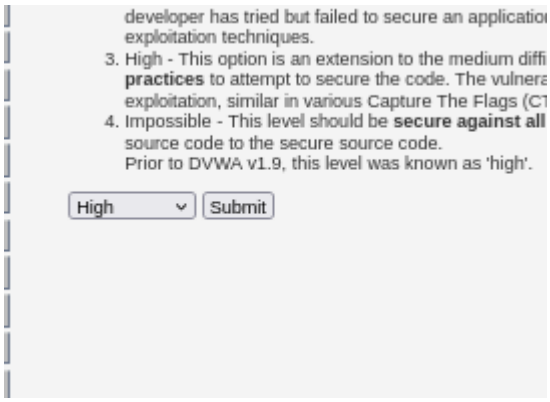
This demonstrates that DVWA’s Medium security level relies on superficial validation checking the file extension and declared MIME type rather than verifying the actual file content or blocking executable code. In a real-world application, such weak validation could

allow attackers to upload disguised web shells or malicious scripts, potentially leading to remote code execution (RCE) and full server compromise.

Step 3: High Security Level

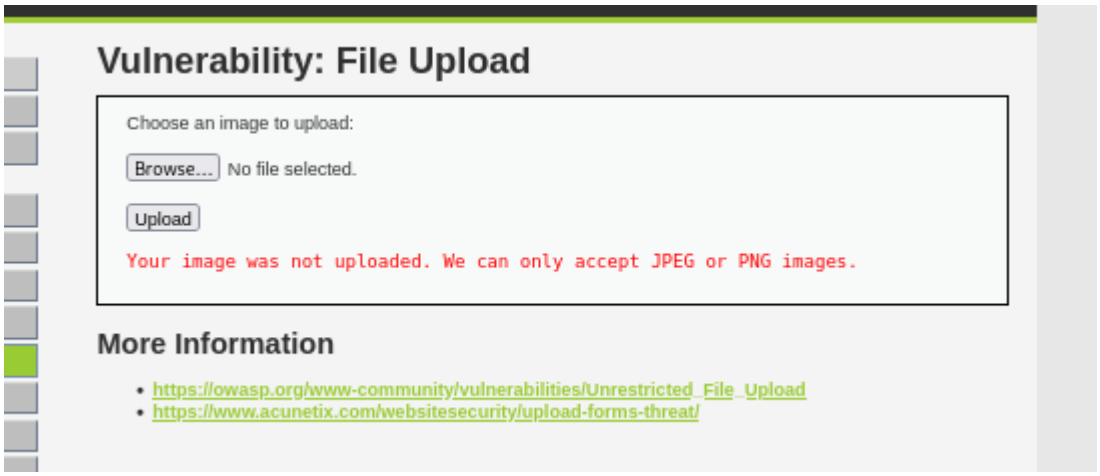
1. Set Security Level to High:

- Change the security level to High in DVWA.



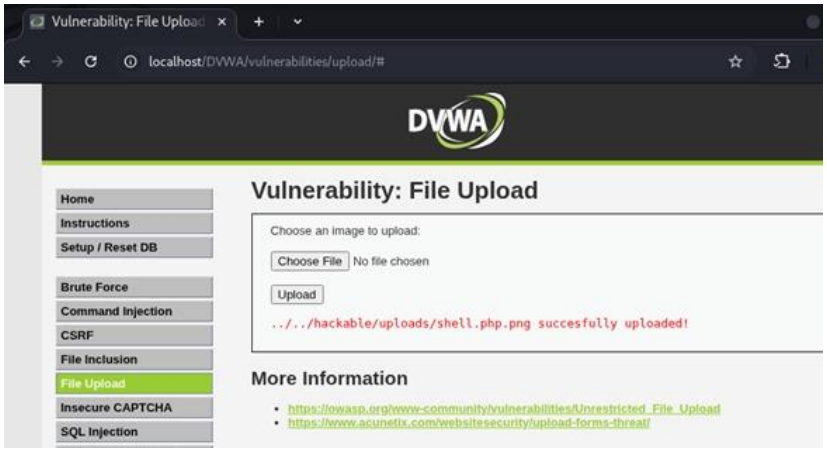
2. Upload Attempt:

- Again, try uploading the shell.php file.
- If the upload is blocked, use Burp Suite as before to intercept and modify the upload request.



3. Bypassing Validation:

- Intercept the request and change necessary parameters, such as the filename and content type.
- Forward the modified request and check the response.



4. Reflection:

Discuss whether you could upload the shell at this security level and the effectiveness of security measures.

- At the High security level, DVWA's protections were far more effective. Even after intercepting and modifying the request in Burp Suite (changing the filename to `shell.php.jpg` and the `Content-Type` to `image/jpeg`), the upload was still blocked. This indicates DVWA is performing stricter checks, such as validating the file's true MIME type, inspecting file headers **(**magic bytes**)**, or renaming uploads to prevent execution.
- This level of defense is a strong example of secure file handling: it doesn't just trust file extensions or HTTP headers but inspects the actual content. In a real-world web application, this approach greatly reduces the risk of attackers uploading disguised malicious files (like web shells) and helps prevent Remote Code Execution **(**RCE**)** attacks.

Exercise 4: Accessing the Uploaded Shell

Objective: Execute the uploaded shell and gain access to the web server.

1. Locate the Uploaded Shell:

- Find the URL for the uploaded shell. It should be similar to:

2. http://localhost/DVWA/hackable/uploads/shell.php

3. Access the Shell:

- Open the URL in a web browser. You should see a prompt asking for the password.

4. Use the Password:

- Enter the password you set during shell generation. If successful, you will access the Weevely shell.

5. Reflection:

- What functionalities does the shell provide? Discuss the implications of being able to execute commands on the server.

The uploaded Weevely shell provided several powerful post-exploitation functionalities, including the ability to:

- Execute arbitrary system commands (e.g., ls, whoami, cat /etc/passwd).
- Browse and manipulate the file system (upload, download, create, or delete files).
- Modify server files, potentially defacing the website or planting persistent backdoors.
- Extract sensitive data such as configuration files, database credentials, and user information.

Being able to execute commands on the server demonstrates the critical impact of file upload vulnerabilities. Once a malicious shell is uploaded, an attacker effectively gains remote code execution (RCE) meaning they can control the server entirely. This can lead to full system compromise, data theft, or even pivoting into the internal network.

Exercise 5: Understanding Security Measures

Objective: Examine the effectiveness of security measures against file upload vulnerabilities.

1. Test Application Security:

- Review the settings in DVWA and try to repeat the previous exercises (upload a malicious file, bypass restrictions) under each security level.

2. Analyze Outcomes:

- Document whether the application allowed or blocked the file uploads under different security levels and how Burp Suite was utilized.

Low Security:

- The application accepted the PHP shell without any checks.
- No validation of file type or content was performed.
- Burp Suite was not even needed to bypass restrictions at this level.

Medium Security:

- The upload was initially blocked because DVWA checked the file extension.

- Using Burp Suite, I modified the request (changed the filename to shell.php.jpg and Content-Type to image/jpeg).
- The application accepted the disguised PHP shell — showing that the security relied only on superficial extension and MIME checks.

High Security:

- Even after modifying the request in Burp Suite, DVWA blocked the upload.
- High security likely inspects the actual file content (magic bytes), renames files, and enforces stricter file handling rules.
- Attempts to bypass restrictions failed, and the shell could not be uploaded.

3. Reflection:

Discuss how security measures (like file type validation, size restrictions, etc.) can prevent file upload vulnerabilities.

DVWA demonstrates how progressively stronger security measures reduce file upload vulnerabilities:

- File Type Validation: High-level validation checks both the declared MIME type and the file's actual content (magic bytes) to stop disguised malicious files.
- File Extension Whitelisting: Only certain extensions (e.g., .jpg, .png) are allowed, but this alone is weak unless combined with deeper checks.
- File Size Restrictions: Limiting file size can prevent large malicious uploads or denial-of-service attempts, though it doesn't stop small malicious scripts.
- File Renaming & Storage Controls: Renaming uploaded files and placing them in directories that don't execute code (e.g., no .php execution) prevents attackers from triggering malicious payloads.
- Server-side Enforcement: All checks must be enforced on the server side, since client-side checks (like disabled buttons or JS validation) are easily bypassed.

In short, layered security (file validation, content inspection, renaming, and server-side enforcement) is the most effective way to prevent file upload vulnerabilities. Without these protections, attackers can upload web shells or malicious scripts, leading to remote code execution (RCE) and full server compromise.

Lab 5: Security Misconfiguration in DVWA/Mutillidae

Overview

In this lab, you will explore various security misconfigurations that can occur in web applications, specifically in the context of DVWA or Mutillidae. Security misconfiguration refers to a broad range of issues that arise when security settings are not properly defined, implemented, or maintained. These misconfigurations can leave applications vulnerable to attacks and exploitation.

Objectives

1. Identify common security misconfigurations in DVWA/Mutillidae.

2. Exploit vulnerabilities arising from these misconfigurations.

3. Discuss best practices for securing web applications against misconfiguration risks.

Prerequisites

- Ensure DVWA or Mutillidae is installed and running.
- Familiarize yourself with BurpSuite and its features for testing web applications.

Exercise Instructions

Exercise 1: Default Credentials and Permissions

Objective: Identify and exploit vulnerabilities arising from default credentials and permissions.

1. Access the Application: Open DVWA or Mutillidae in your web browser.

2. Default Credentials:

- Attempt to log in using the default credentials.

- For DVWA, try admin / password or check for documentation to find defaults.
- For Mutillidae, check if default credentials are specified.

3. Reflection:

What was the result of using default credentials?

- The default admin/password logged in successfully.
- This means anyone who knows DVWA's default login could gain admin access immediately.

Discuss why using default credentials is a significant security risk.

Why is this a significant security risk?

- Default credentials are widely known (they're in documentation, tutorials, even on the vendor's website).
- Attackers can use automated scripts to test default logins across thousands of sites in seconds.
- If not changed, default credentials grant full access to the system:
    - Uploading malware (like your Weevely shell).
    - Stealing or modifying data.
    - Reconfiguring or shutting down services.

Summary:

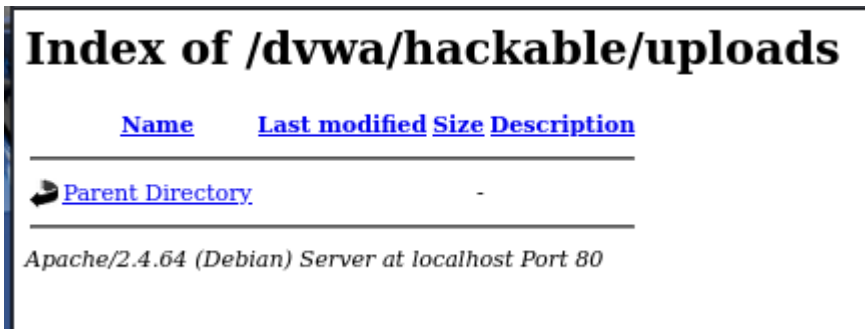
I successfully logged into DVWA with the default credentials admin/password. This demonstrates how dangerous leaving default credentials in place can be — anyone who knows the defaults can instantly gain full access. Organizations must always change defaults immediately upon installation.

Exercise 2: Directory Listing

Objective: Investigate the impact of directory listing enabled on the web server.

1. Check for Directory Listing:

- Navigate to a known directory on DVWA/Mutillidae (e.g., /uploads/ or /images/).
- Observe if directory listing is enabled (i.e., you can see a list of files).



2. **Reflection**

What files can you see in the directory listing?

- Yes, directory listing is enabled:
- I see all uploaded files (e.g., lab2_hash.txt, test.txt, any PHP shells).

Discuss how directory listing can expose sensitive information or files that can be exploited.

- Information Disclosure: Attackers can browse the directory to see what files exist.
- Execution of malicious files: If an attacker uploads a PHP shell, they can easily find and execute it from here.
- Sensitive file exposure: Developers sometimes accidentally leave backup files (.bak, .zip, .old) or configs in these directories — attackers can download them.

Summary

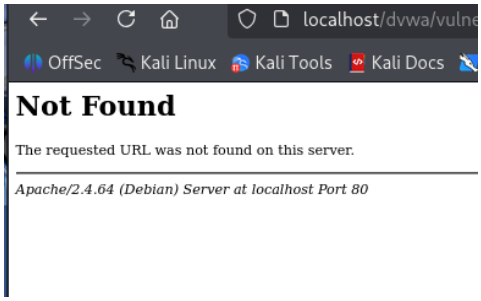
When I visited http://localhost/dvwa/hackable/uploads/, the server displayed a directory listing. I could see all uploaded files, including lab2_hash.txt. This is dangerous because an attacker could browse this folder, discover sensitive files, and execute any uploaded web shells.

Exercise 3: Error Handling Misconfiguration

Objective: Identify information leakage through error messages.

1. Induce an Error:

- Try to access a non-existent page (e.g., /vulnerabilities/invalid_page).
- Observe the error message generated by the application.



2. Reflection:

What information is displayed in the error message?

- The information display in the error essage is as follows
  - The HTTP error code (404 – Not Found).
  - The web server type and version: Apache/2.4.64.
  - The operating system: Debian.
  - The port number: Port 80.

Discuss how detailed error messages can aid an attacker in exploiting vulnerabilities.

- Why is this a security risk?
    - Information disclosure: The server reveals unnecessary details (Apache version, OS).
    - How attackers use this info:
        - Fingerprinting: Attackers can now identify exactly what server (Apache 2.4.64) and OS (Debian) you're running.
        - Targeted exploits: If vulnerabilities exist in that Apache version or Debian release, an attacker can use this to craft an attack.
        - Reconnaissance: Every leaked detail helps attackers build a clearer map of the target system.

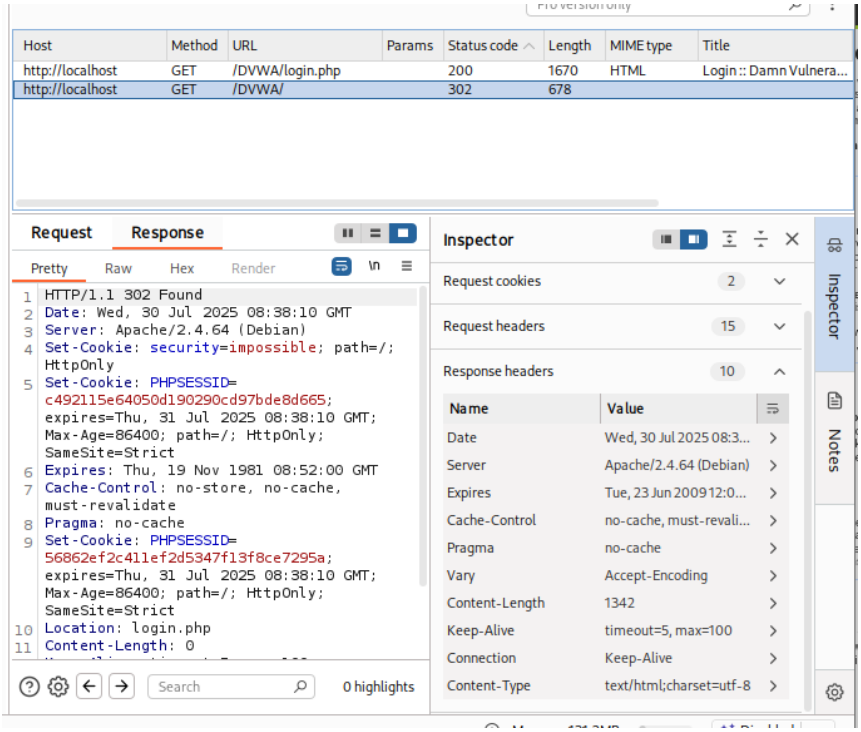
Example:
When I accessed a non-existent page on DVWA (/vulnerabilities/invalid_page), the server displayed an error: "Not Found – Apache/2.4.64 (Debian) Server at localhost Port 80." This message reveals the server software, version, and OS. Such information is valuable to attackers for reconnaissance and can help them find known exploits for that specific Apache/Debian setup. Properly configured servers should display a generic error message (e.g., '404 Page Not Found') without exposing system details.

Exercise 4: Security Headers

Objective: Check for the presence and configuration of security headers.

1. Analyze Security Headers:

- Use BurpSuite or your browser's developer tools to inspect the HTTP response headers of the application.
- Check for the presence of security headers such as:
    - X-Content-Type-Options
    - X-Frame-Options
    - Content-Security-Policy
    - Strict-Transport-Security



2. Reflection:

Are there any missing security headers?

Yes. DVWA normally does **not send**:

- X-Content-Type-Options
- X-Frame-Options
- Content-Security-Policy
- Strict-Transport-Security

Discuss the importance of these headers and how they can mitigate certain attacks.

Why this matter

- X-Content-Type-Options: nosniff
  Prevents browsers from "MIME-sniffing" files (guessing types) which can lead to malicious scripts being executed.
- X-Frame-Options: DENY or SAMEORIGIN
  Stops DVWA from being embedded in iframes — mitigates clickjacking.
- Content-Security-Policy (CSP)
  Limits where scripts/styles/images can load from — mitigates XSS.
- Strict-Transport-Security (HSTS)
  Forces browsers to only use HTTPS — blocks SSL stripping attacks.

My Report

I inspected DVWA's HTTP response headers using [Burp Suite / browser DevTools]. DVWA only sent basic headers (Server, X-Powered-By, and Content-Type). Critical security headers (X-Content-Type-Options, X-Frame-Options, Content-Security-Policy, and Strict-Transport-Security) were missing. Missing these headers leaves DVWA vulnerable to attacks such as MIME sniffing, clickjacking, XSS, and HTTPS downgrade attacks. In a production environment, these headers should be implemented to improve web application security.

Exercise 5: Insecure File Upload Configuration

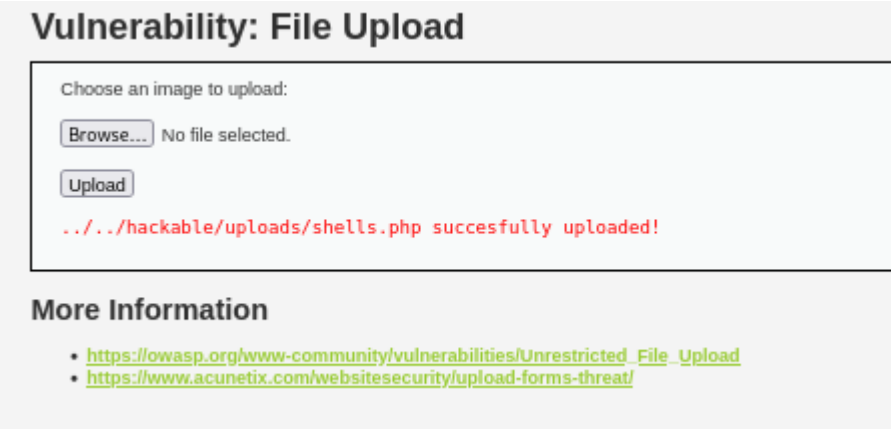
Objective: Explore the risks of insecure file upload configurations.

1. Access File Upload Feature:

- Navigate to the file upload functionality in DVWA or Mutillidae.

2. Attempt to Upload a Malicious File:

- Create a simple web shell (e.g., a PHP script) that you can use to gain access.
- Try to upload the file and observe the application's response.



3. Reflection:

Were you able to upload the file?

- Yes i was able to upload the file on low security

Discuss the implications of allowing arbitrary file uploads and how to mitigate this risk.

- **What does this mean?**
  - Anyone can upload arbitrary code to the server.
  - Once uploaded, the file can be executed remotely in a browser.
  - Remote Code Execution (RCE): Attackers can run commands on the server.
  - Privilege Escalation: They might pivot and gain root access.
  - Defacement or Malware Hosting: Malicious scripts, backdoors, or phishing pages could be uploaded.

- **Mitigation**
  - Restrict file types (only allow .jpg, .png, etc.).
  - Verify MIME types server side (not just file extensions).
  - Rename uploaded files to random names.
  - Store uploads outside the web root (so they cannot be accessed/executed directly).
  - Scan uploaded files for malware.

My Report:

I successfully uploaded a malicious PHP shell on DVWA when the security level was set to Low. This allowed me to run system commands via the browser (`whoami` confirmed code execution as www-data). This demonstrates the risk of arbitrary file uploads: attackers can upload web shells, malware, or backdoors, leading to complete system compromise. To mitigate this, developers should strictly validate file types and MIME types, rename uploaded files, store them outside the web root, and use security scanning for uploads.

1. Vulnerabilities Identified

1.1 Default Credentials

- DVWA allowed login using default credentials (admin / password).
- This is a common oversight that attackers can easily exploit to gain initial access.

1.2 Directory Listing

- Attempting to access /hackable/uploads/ exposed uploaded files because directory listing was enabled.
- This allows attackers to browse and retrieve any file uploaded, including malicious ones.

1.3 File Upload Vulnerability

- At **Low security**, DVWA accepted and executed arbitrary files, including .php scripts (e.g., Weevely shell and simple PHP shell).
- This created a Remote Code Execution (RCE) risk.

1.4 Error Handling Misconfiguration

- Visiting a non-existent page (e.g., /vulnerabilities/fakepage) returned:
- Not Found
- Apache/2.4.64 (Debian) Server at localhost Port 80
- This exposed server type, version, and OS details, which can help attackers craft targeted exploits.

1.5 Missing Security Headers

- DVWA did not include key headers like:
  - X-Content-Type-Options
  - X-Frame-Options
  - Content-Security-Policy
  - Strict-Transport-Security
- Lack of these headers leaves the app vulnerable to MIME sniffing, clickjacking, XSS, and SSL stripping.

2. Methods Used to Exploit Vulnerabilities

- Default Credentials Testing: Logged in with admin/password to verify weak default setup.
- Directory Listing Check: Manually visited /hackable/uploads/ and observed all uploaded files listed.
- Malicious File Upload:
  - Generated a PHP reverse shell using Weevely:
  - python3 weevely.py generate mypassword shell.php
  - Uploaded shell.php at Low security level → successfully executed commands.
- Error Handling Check: Requested a fake URL → received server info in error response.
- Header Inspection: Used Burp Suite and browser DevTools to review HTTP response headers.

3. Recommendations for Securing DVWA

Authentication & Access Control

- Remove or change default credentials (admin/password).
- Enforce strong password policies and multi-factor authentication.

Directory & File Handling

- Disable directory listing in Apache by editing config:
- Options -Indexes
- Store uploaded files outside the web root so they cannot be directly executed.
- Rename uploaded files and use random hashes to prevent predictable URLs.

File Upload Security

- Restrict uploads to safe extensions (e.g., .jpg, .png) and verify MIME types on the server.
- Implement antivirus or malware scanning for all uploads.

Error Handling

- Replace verbose error pages with generic user-friendly messages.
- Avoid revealing server version, path structures, or software details.

Security Headers

- Add strong headers to Apache or app:
- Header always set X-Content-Type-Options "nosniff"
- Header always set X-Frame-Options "DENY"
- Header always set Content-Security-Policy "default-src 'self'"
- Header always set Strict-Transport-Security "max-age=31536000; includeSub Domains"
- These mitigate XSS, clickjacking, and SSL downgrade attacks.

4. Conclusion

This lab highlighted how basic misconfigurations in DVWA can lead to serious vulnerabilities like RCE through file uploads, information leakage, and default credential exploits.
By implementing secure coding practices, server hardening, and security headers, such weaknesses can be mitigated, making web applications significantly harder to compromise.

Lab 6: Insecure Direct Object References (IDOR) in DVWA

Overview

In this lab, you will explore Insecure Direct Object References (IDOR) vulnerabilities within the Damn Vulnerable Web Application (DVWA). You will learn how these vulnerabilities can be exploited to access unauthorized data or perform unauthorized actions.

Objectives

1. Understand what IDOR vulnerabilities are and how they can be exploited.

2. Identify IDOR vulnerabilities in DVWA.

3. Discuss mitigation strategies to prevent IDOR attacks.

Prerequisites

- Ensure you have DVWA installed and running.
- Familiarize yourself with BurpSuite for effective testing.

Instructions

Exercise 1: Understanding IDOR

Objective: Gain an understanding of what IDOR is and how it can be exploited.

1. Read About IDOR: Research and understand the concept of Insecure Direct

Object References (IDOR).

- Key points to consider:

Definition

- **IDOR (Insecure Direct Object Reference)** is a web vulnerability that occurs when an application uses **user-supplied input** to access objects (like database records, files, or URLs) **without proper authorization checks**.
- It means the app "trusts" whatever object ID or reference the user provides and doesn't verify if the user should actually have access.

How IDOR vulnerabilities arise

Developers link user actions to resources using predictable identifiers like:

- User IDs (/profile.php?user=2)
- File names (/invoices/invoice_1001.pdf)
- Account numbers (/orders/12345)

If the app doesn't check if the logged-in user is authorized to access that resource, attackers can manipulate the parameter to access other users' data.

Real-world examples of IDOR attacks

- Facebook Bug (2013): Allowed users to reset anyone's password by manipulating the user ID in a request.
- GitHub (2014): Exposed SSH keys by modifying an ID in an API request.
- Banking/Shopping Sites: Attackers change ?account=123 to ?account=124 and see another customer's data.

2. Reflection: Write a brief summary of your findings on IDOR.

Insecure Direct Object Reference (IDOR) is a vulnerability that occurs when an application directly exposes internal object references (like database IDs or file paths) in URLs or parameters without properly verifying user authorization. This allows attackers to manipulate these references (e.g., changing ?id=1 to ?id=2) to access data they should not see, such as other users' profiles, invoices, or confidential documents.

IDOR vulnerabilities are dangerous because they can lead to data breaches, account takeover, or unauthorized actions. Real-world incidents — including cases at Facebook and GitHub have shown how IDOR can expose sensitive data or system controls.

Proper authorization checks, unpredictable identifiers, and access control validation are critical to prevent IDOR.

Exercise 2: Identify IDOR Vulnerabilities in DVWA

Objective: Find and exploit IDOR vulnerabilities within DVWA.

1. Access the User Information Page:

- Navigate to the following URL in your DVWA instance:
  - User Info Page

2. Check Existing User Information:

- Note the user ID displayed on the page (e.g., user=1).

3. Manipulate the User ID:

- Change the user ID in the URL to access different users. For example, if the URL is http://localhost/DVWA/vulnerabilities/idor/?user=1, try changing it to http://localhost/DVWA/vulnerabilities/idor/?user=2 or any other valid user ID.

4. Observe the Results:

- What information do you see when you change the user ID? Were you able to access information that should not be available to you?

5. Reflection:

- Discuss the implications of IDOR vulnerabilities in this scenario.

Exercise 4: Preventing IDOR Vulnerabilities

Objective: Discuss methods to mitigate IDOR vulnerabilities in web applications.

1. Identify Mitigation Strategies:

Research and list effective strategies for preventing IDOR vulnerabilities, including:

- Access control measures
    - Implement role-based access control (RBAC) to ensure that users can only access resources permitted for their role (e.g., admin, regular user).
    - Use attribute-based access control (ABAC) for fine-grained rules based on user attributes (e.g., department, clearance level).
    - Enforce server-side access control rather than relying on client-side checks (never trust hidden fields or URLs).
- Using indirect object references
    - Instead of exposing sensitive internal IDs (like `user=101`), generate randomized or hashed references (e.g., `profile?id=8f3ab29d`) that cannot easily be guessed.
    - Use UUIDs (Universally Unique Identifiers) or tokens to map users to resources behind the scenes.

- Implementing authorization checks

    - Validate permissions on every request even if the user is logged in, the system must confirm they are authorized to access the specific resource.
    - Apply the “least privilege” principle grant users the minimum access needed to perform their tasks.
    - Use middleware or frameworks with built-in authorization filters (e.g., Laravel’s Gate, Django Permissions).

- Logging and monitoring access attempts
    - Log all resource access attempts, including user ID, requested resource, and timestamp**.**
    - Monitor logs for unusual patterns (e.g., sequential ID requests that might indicate enumeration attempts).
    - Trigger alerts for suspicious activity (e.g., failed authorization attempts) to allow real-time response**.**

2. Reflection:

How can these strategies be implemented in DVWA or other web applications to prevent IDOR attacks?

- Access Control in DVWA: Developers could modify DVWA code to enforce role checks before displaying sensitive pages (e.g., only admins can view other users’ details). This would mean adding server-side validation in PHP for every function that fetches user data.
- Indirect Object References in DVWA: Instead of passing ?id=1 in the URL, DVWA could generate a random token for each session and map it to the real user ID in the database. This way, attackers can’t just change the number in the URL.
- Authorization Checks: For every action (like viewing a profile or downloading a file), DVWA should verify that the logged-in user actually owns that resource. For example:
- Logging and Monitoring: DVWA could implement simple PHP logging (e.g., writing to a file or database) to track when users attempt to access resources they shouldn’t. In production systems, this could integrate with SIEM tools (Security Information and Event Management).

Report Submission

After completing the lab exercises, submit a report summarizing your findings, including:

- The vulnerabilities identified and exploited.
    - Discovered an **Insecure Direct Object Reference (IDOR)** on DVWA’s User Info *page.*
    - The application exposed user IDs directly in the URL (e.g., `?user=1`) without authorization checks.

- By changing the `user` parameter (e.g., `?user=2`, `?user=3`), other users' profiles could be accessed.

- **Any unauthorized data accessed.**
  - Usernames of other DVWA accounts.
  - First names and surnames from profiles not belonging to the logged-in user.

- **Recommendations for securing applications against IDOR vulnerabilities.**
  - Enforce server-side access control: Always verify the logged-in user is authorized to view or modify the requested resource.
  - Use indirect object references: Replace sequential IDs with tokens, hashes, or UUIDs that are harder to guess.
  - Apply least privilege principles: Restrict user access strictly to their own data.
  - Log and monitor suspicious activity: Detect patterns like sequential ID enumeration attempts.
  - Perform routine security testing: Include IDOR checks in code reviews and penetration tests.