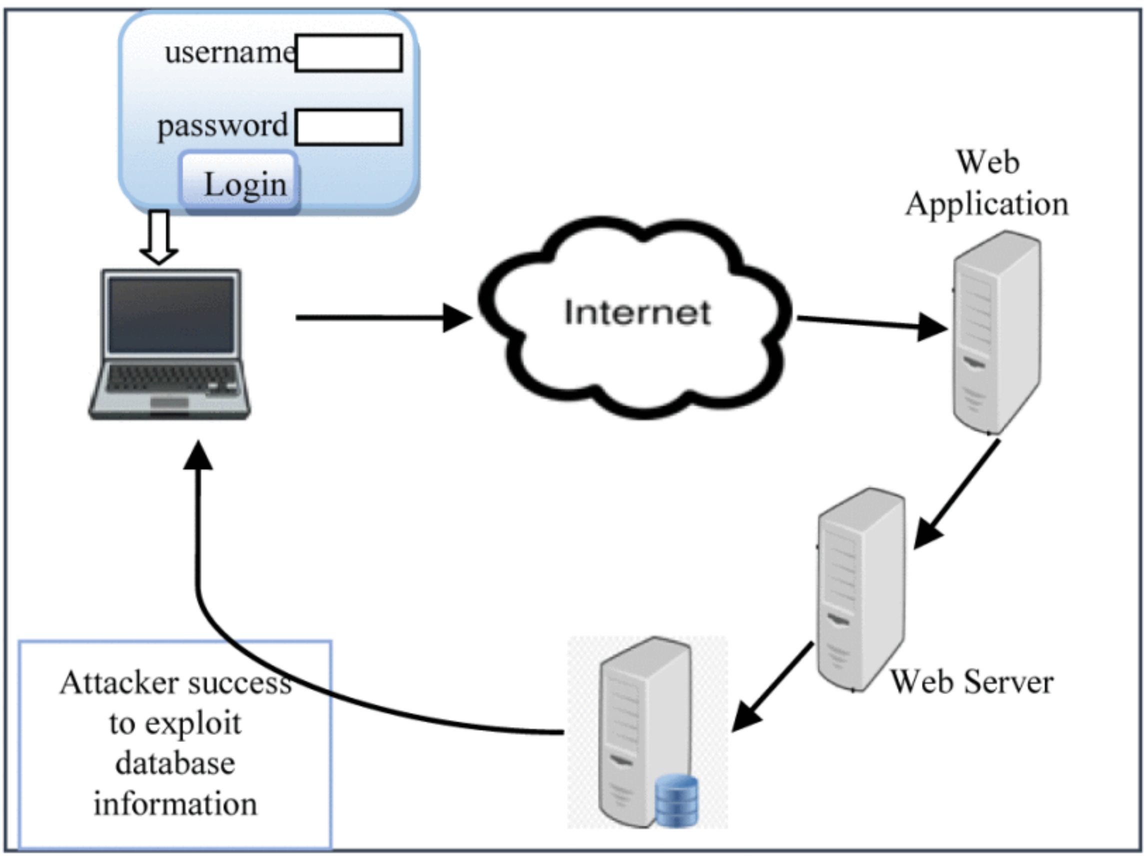
SQL Injection

Understanding the basic principles

DANIEL OKORO
UNIVERSITY OF SUNDERLAND | SUNDERLAND, UK

Introduction

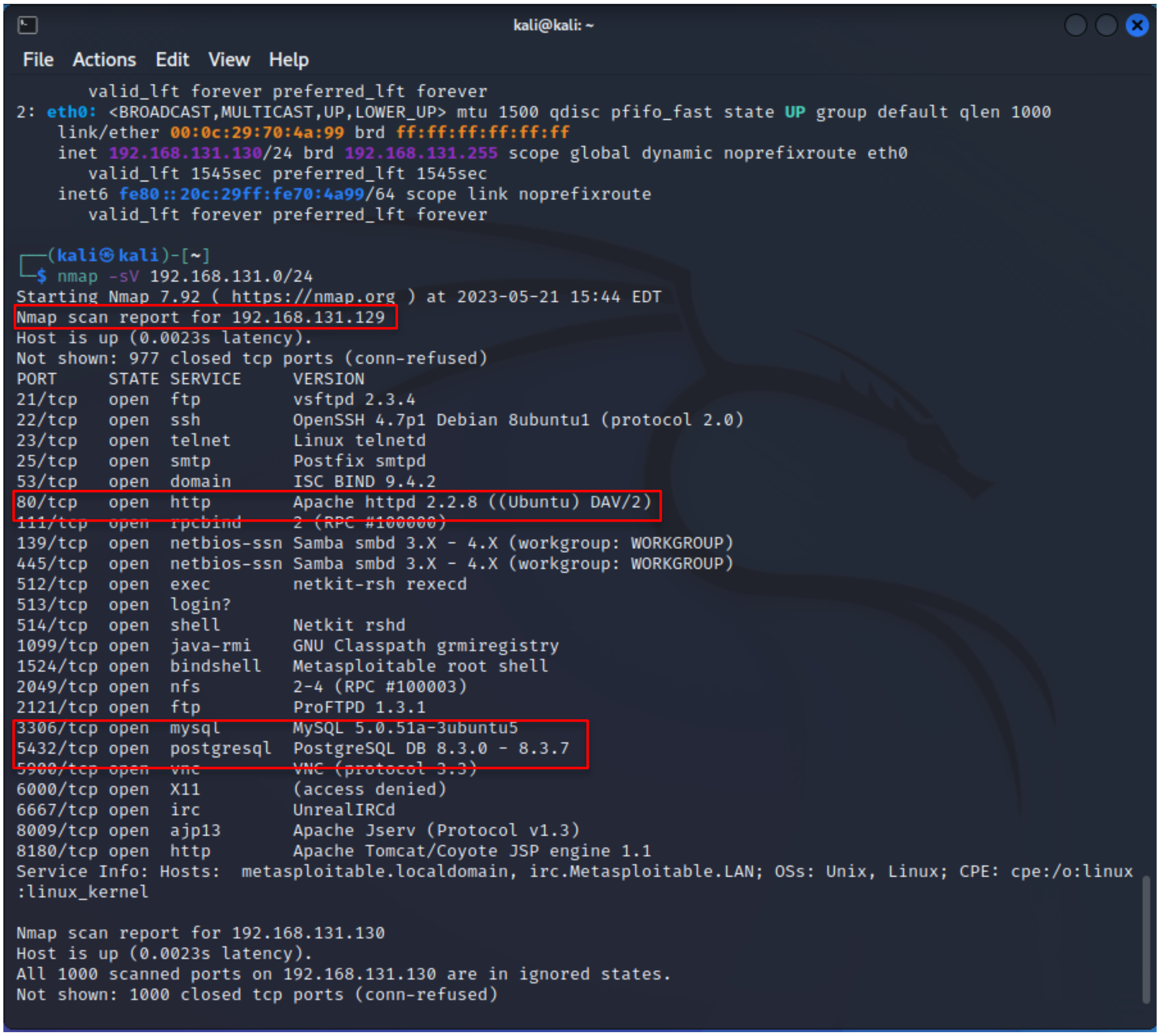
Businesses today have web applications facing the public providing seamless services to users across the internet at large exposing them to the risk of SQL Injection Attacks (SQLIA), where threat actors inject malicious SQL code through URLs or web forms to gain unauthorised access, and manipulate data, or retrieve sensitive information. Such attacks have severe consequences, including damage to user experience, organizational reputation, and potential legal penalties under data protection regulations like GDPR. SQL Injection (SQLi) is a significant threat that enables unauthorised data extraction, modification, the authentication bypass, remote command execution, and vulnerability identification E-commerce databases are particularly vulnerable, allowing attackers to extract customer data and manipulate security measures. Despite preventive measures, SQLIA remain a prominent concern, ranking third among OWASP's top ten web application vulnerabilities.



SQLi overview

SQL Injection Step-by-step Guide

Step one: Reconnaissance



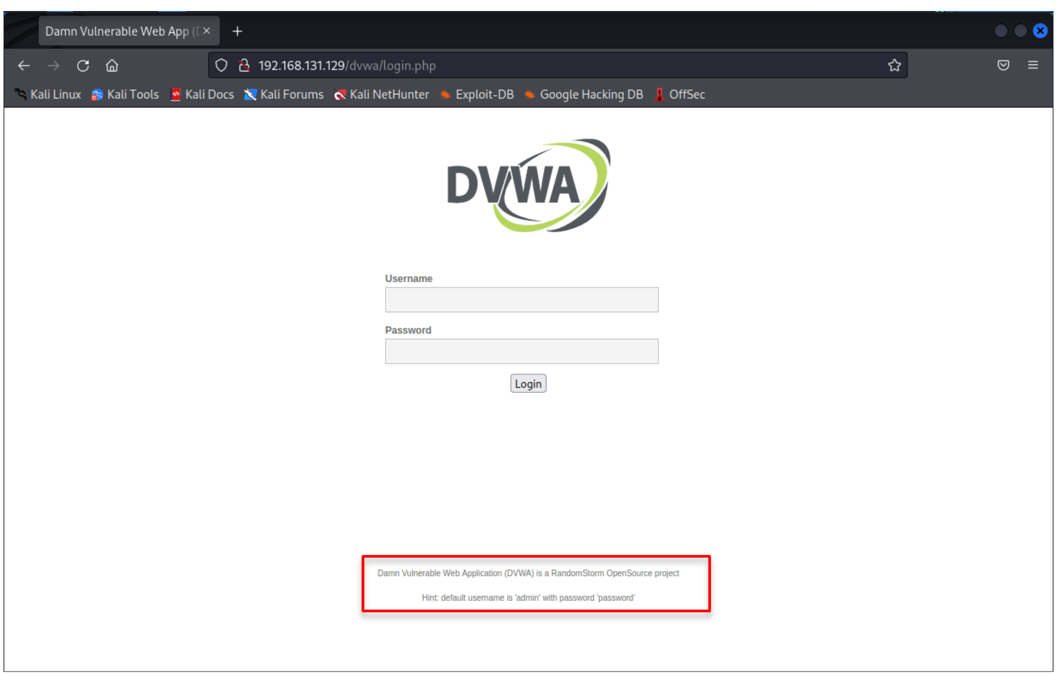
Scanning the subnet to discover host and service with nmap.

Reconnaissance is important to gather useful information about the target system, in this case *nmap* was used with the version flag. Host was discovered on 192.168.131.129 with different services running with open ports and looking closely we can spot two database services running on port 3306 (mysql) and port 5432 (postgresql) which might be the area of interest for SQLi, while vulnerabilities could quickly be checked for using tools like *SQLMap* and *msfconsole*, but let’s go check what’s is running on their web front since we find a webserver running on port 80.

Login to the Webserver

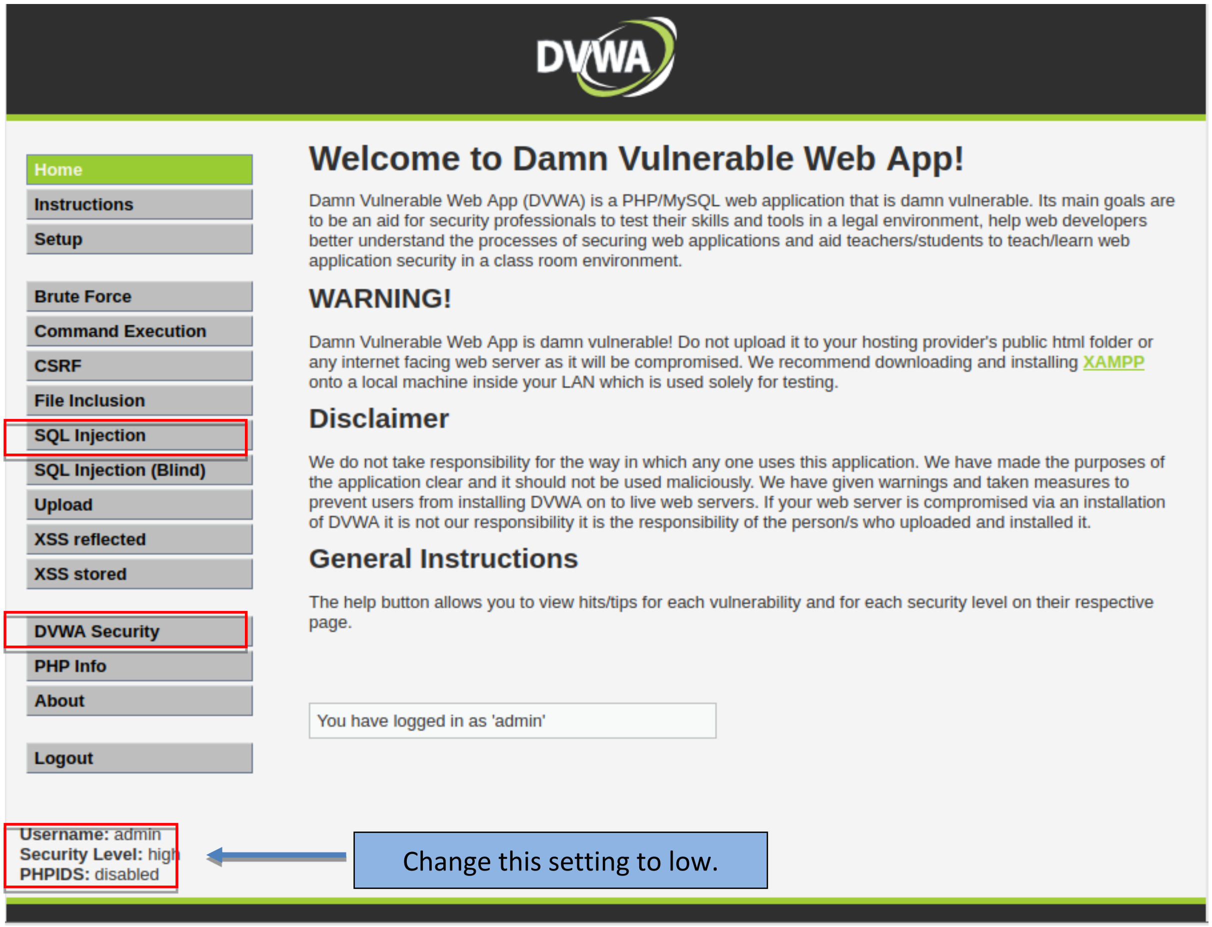
Landing page of webserver

We find the Damn Vulnerable Web Application (DVWA)



DVWA login page.

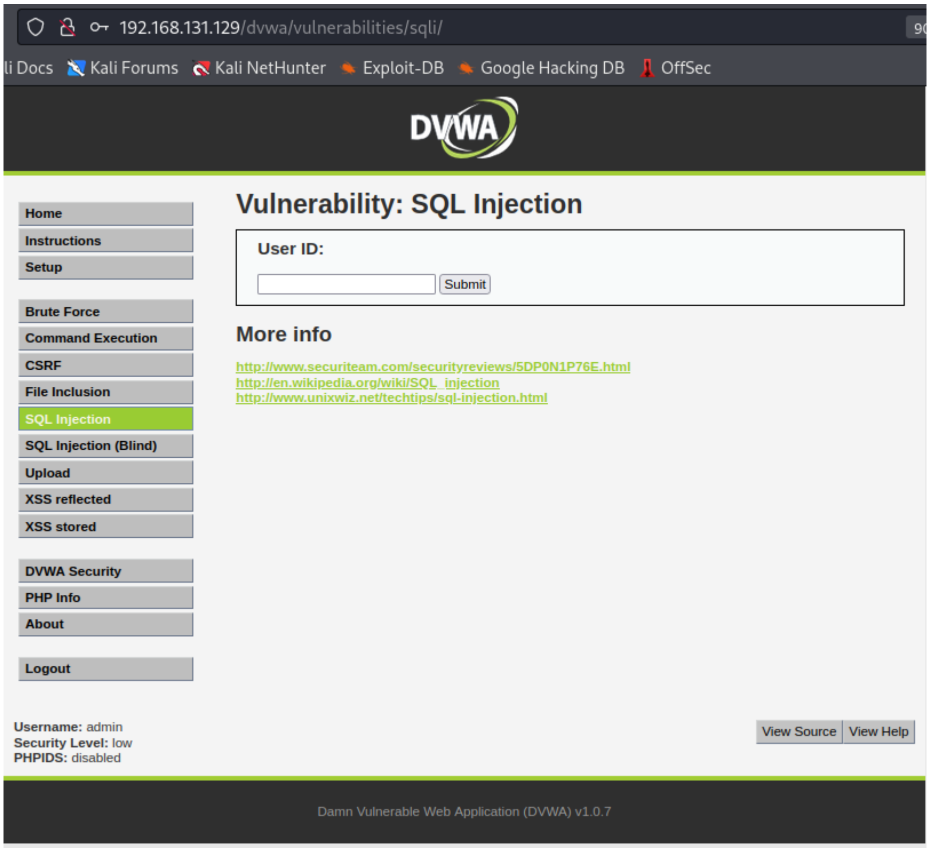
We proceed to login and commence exercise.



DVWA settings.

To explain the SQLi principle we get into DVWA security and start with low and walk up by exploiting any found vulnerabilities by testing several payloads.

Web Form



Web form.

Here we are served with a web form where a user can supply input data, which is an ‘id’ to the web application.

SQL injection

SQL Injection Source

```php
<?php

if(isset($_GET['Submit'])){

    // Retrieve data

    $id = $_GET['id'];

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre>' );

    $num = mysql_numrows($result);

    $i = 0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

Web Application source code (DVWA).

Looking at the source code, we can see the query syntax as the *$_Get* command attempts to retrieve 'id' from the URL parameter and store in a variable '$id'. SQL query to retrieve the *first_name* and *last_name* columns from a table named "users" where the *user_id* matches the provided $id value. The mysql_query() function executes the query. If the variable 'i' is less that the number of rows returned from the query, the form output the result of the query showing the *first_name* and *last_name* of each row.

Knowing how this work we would try exploiting the URL parameter and the form by injecting malicious input.

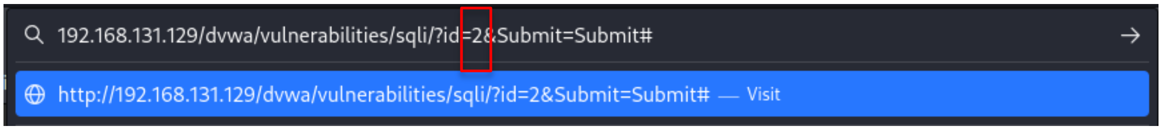
Injection test.

Inserting ‘1’ into the input form, we get an output displaying the expected data as shown on by the source code.

```
<div class="body_padded">
    <h1>Vulnerability: SQL Injection</h1>


    <div class="vulnerable_code_area">

        <h3>User ID:</h3>

        <form action="#" method="GET">
            <input type="text" name="id">
            <input type="submit" name="Submit" value="Submit">
        </form>

        <pre>ID: 1<br>First name: admin<br>Surname: admin</pre>

    </div>
```

Page inspection.

Inspecting the page shows the method used, and that user inputs are required.



Force browsing via URL.

Let’s try to manipulate the URL parameters to see if there is possible forced browsing by trying an ‘id’ = ‘2’

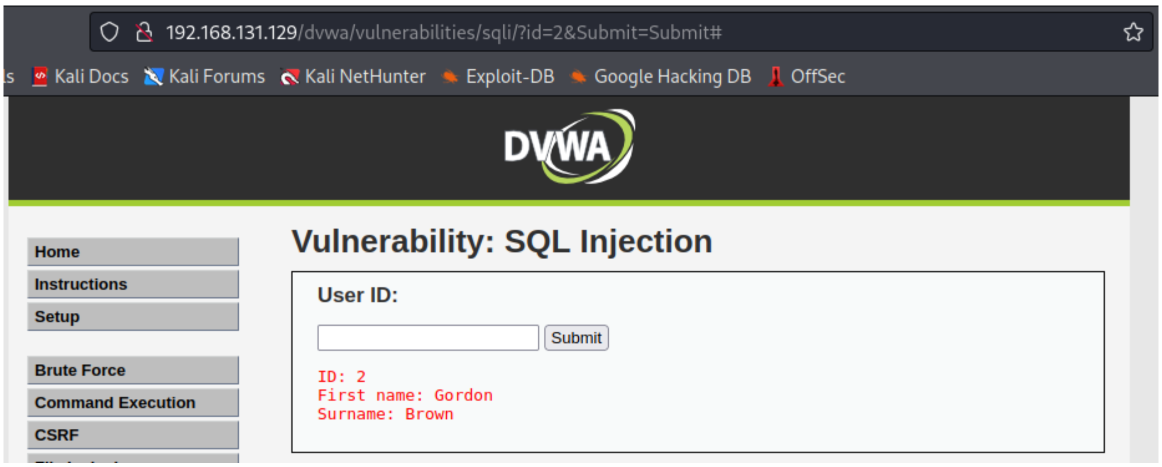


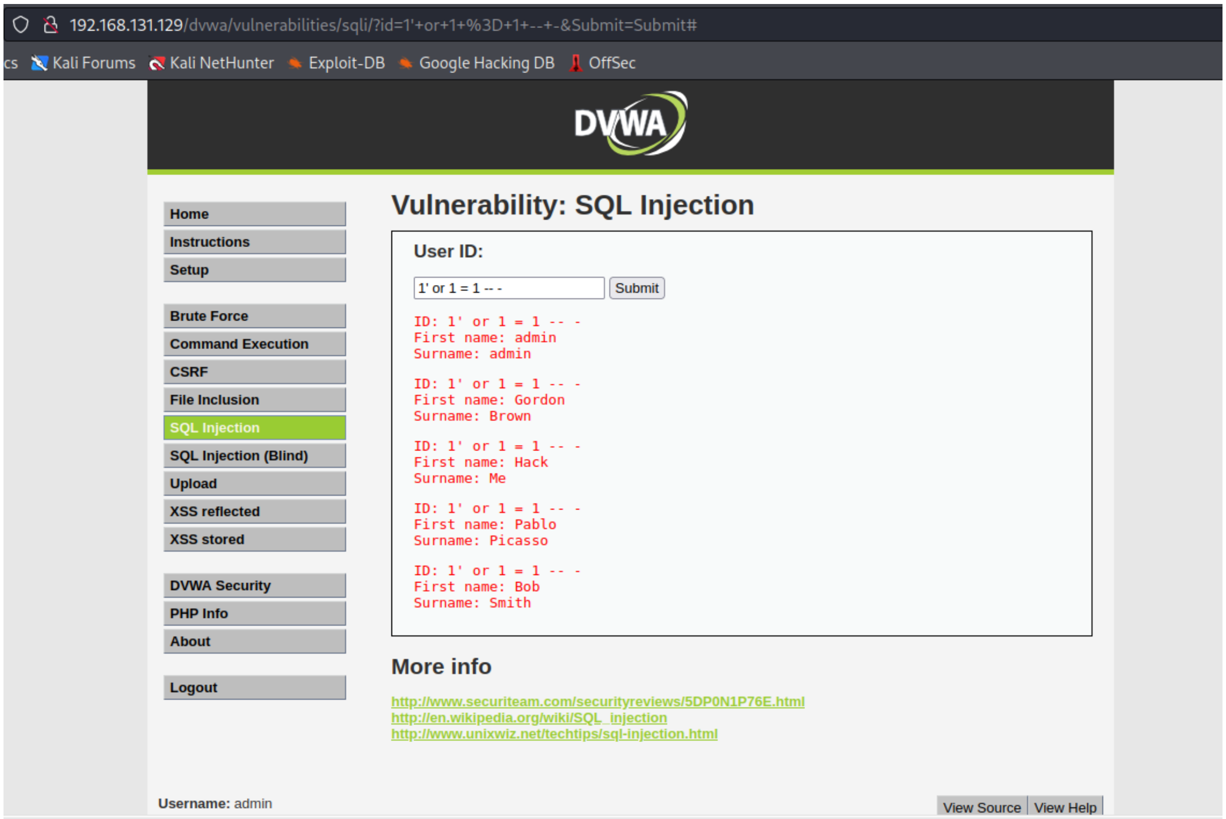
Figure 4.4.5 Injection test output.

We successfully get an output thus showing that the code is vulnerable, we delve further to get more record from the database by manipulating input.

We can tell that the query executed follows the SQL format of:

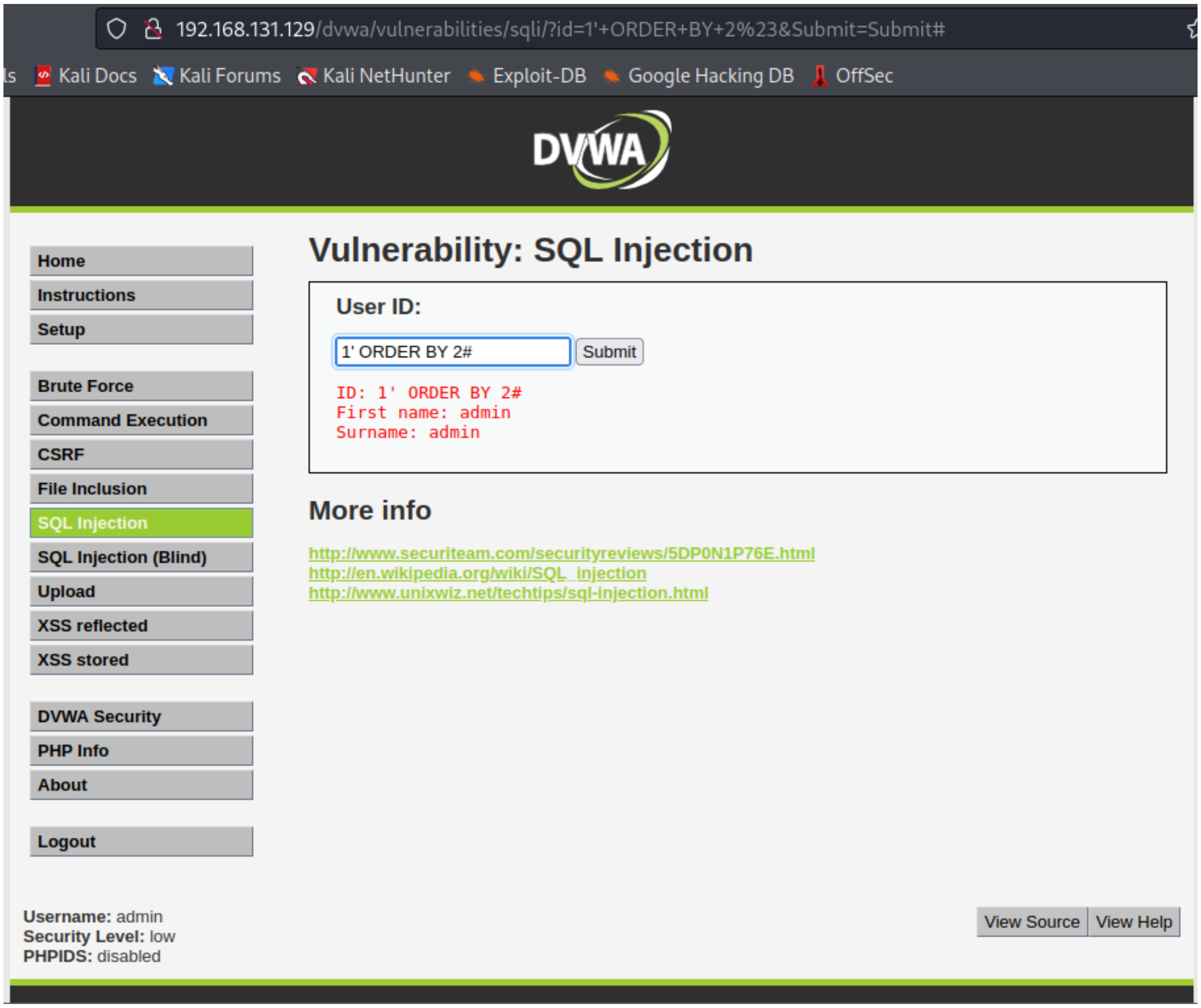
```
SELECT first_name, last_name FROM users WHERE user_id = ‘2’;
```

From the earlier PHP code on fig 4.4.1 we know that the output of First name = first_name and Surname = last_name

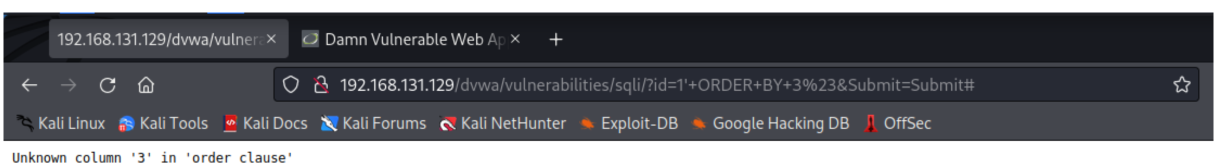


Always true injection.

Here we try an always true injection for the ‘id’ by supplying any input (in this case we use ‘1’) concatenated with “OR ‘1 = 1” with an SQL comment ‘ -- -" technique, which will return data further confirming that the web application is vulnerable to SQLi.



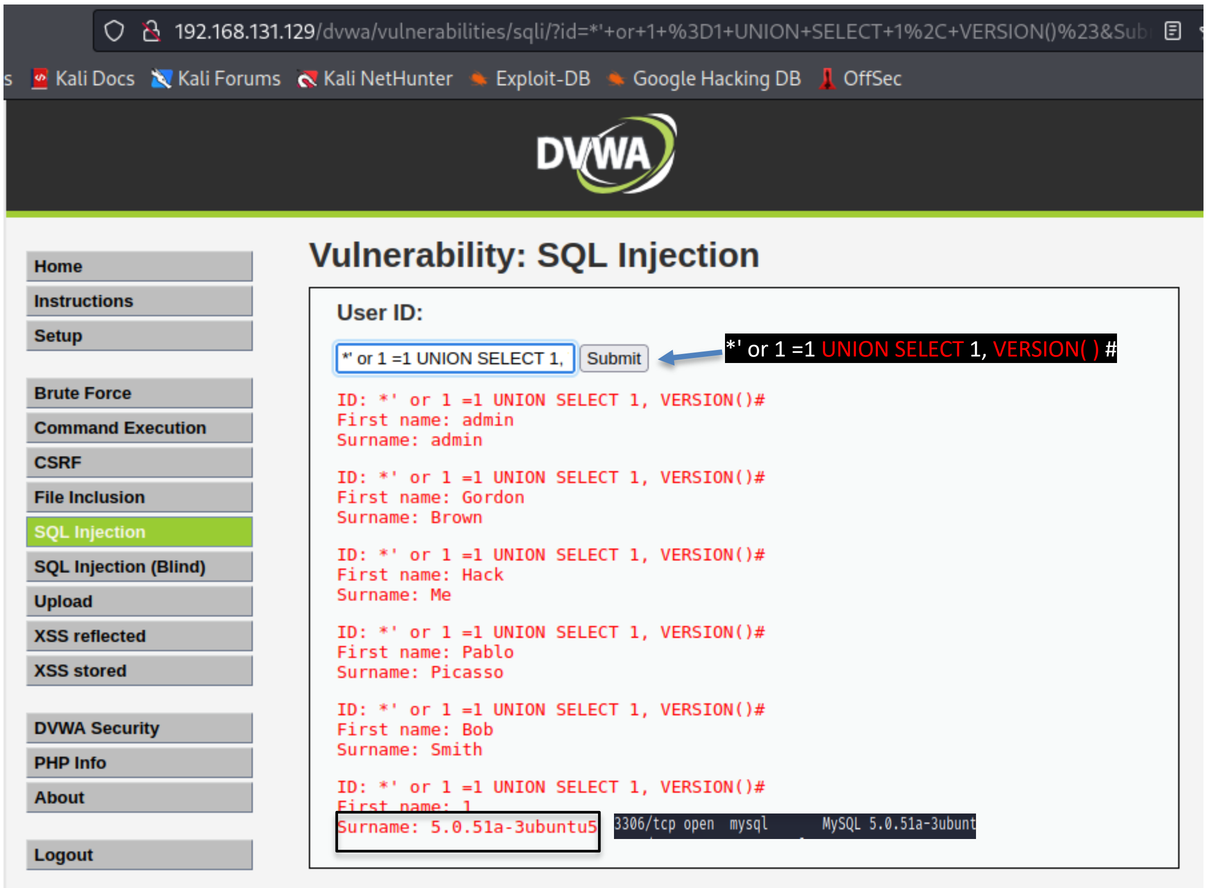
Field check



Field check error.

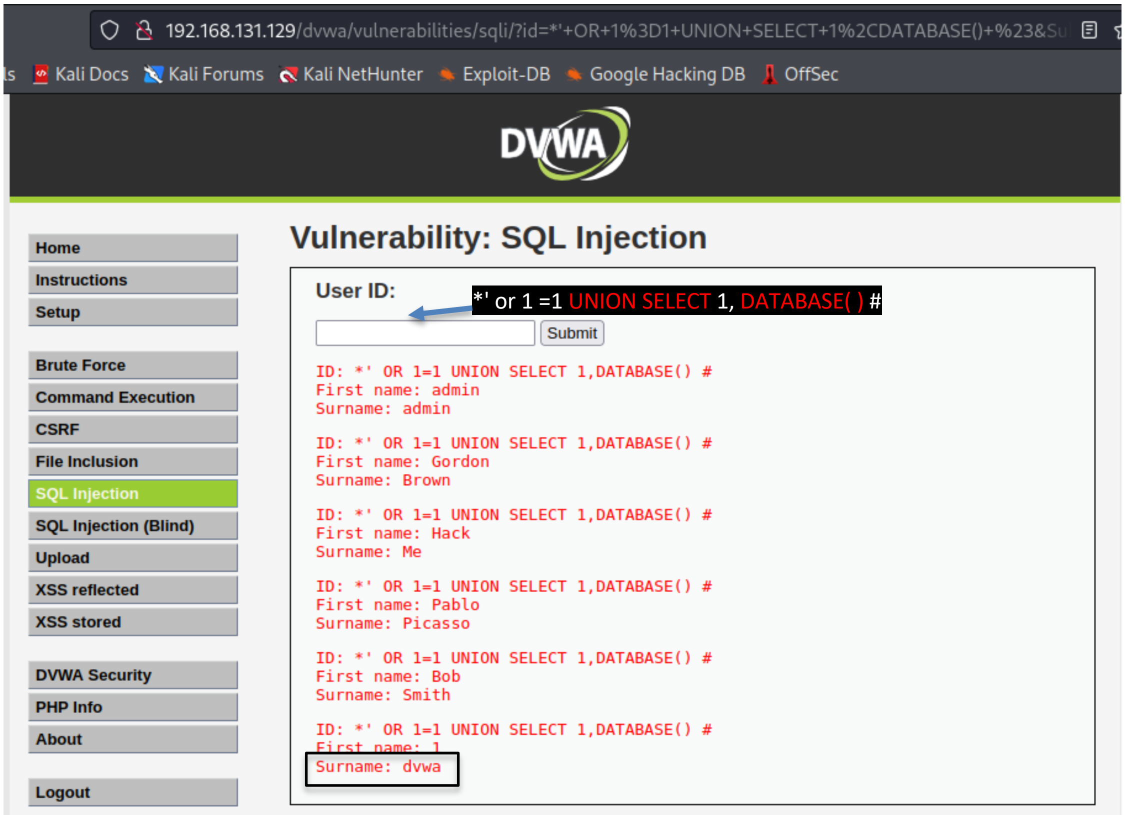
To get the number of field present in the database query we attempt to append "ORDER BY" testing for '1' and then '2' the query and we get an output, but on trying for a '3' field, we are hit with error.

This confirms that only two fields are present in each query, which will be helpful for certain "UNION SELECT" queries. A hash value '#' can also be use in place of SQL comment ' -- -'.



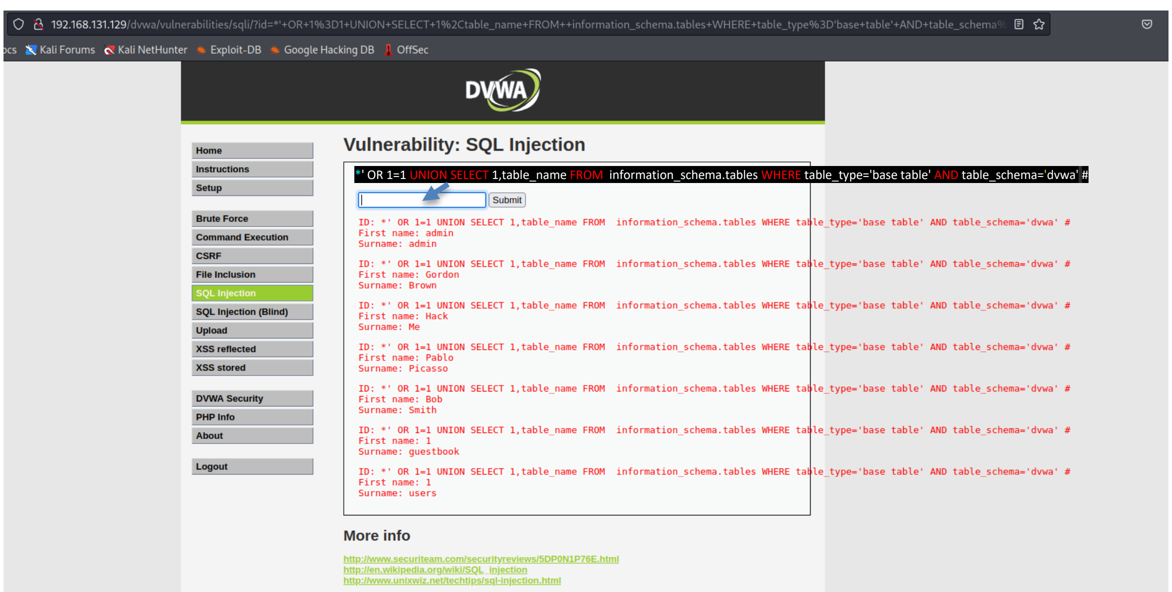
Retrieving MySQL version.

Using the "UNION SELECT" technique to obtain additional information, we can add the MySQL "VERSION" unction to our query with the assumption of retrieving the database version. If successful, we can compare it with the *nmap* output to confirm the version. This time instead of using a '1' a try using special character as an invalid input '*' and still got results which is because the True injection still works.



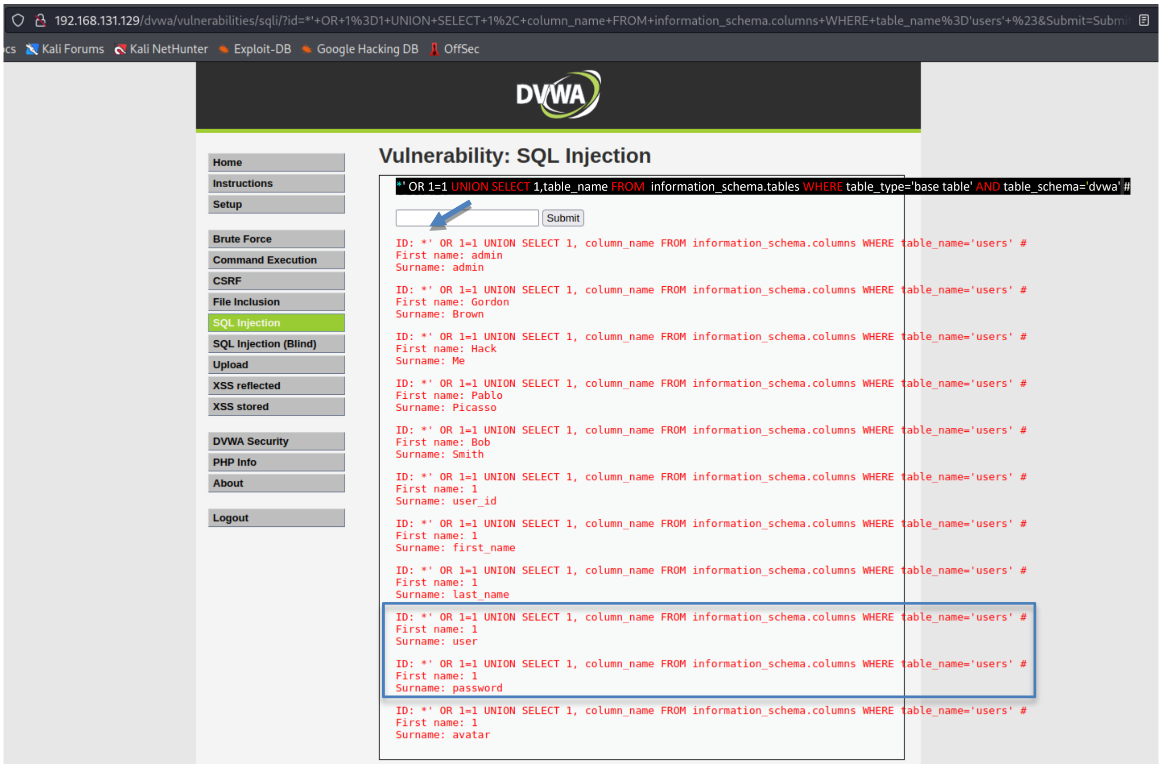
Database name retrieval.

This time, we append the "DATABASE( )" function to our query, and we successfully retrieve the database name 'dvwa'. This information will be useful for further queries and operations.



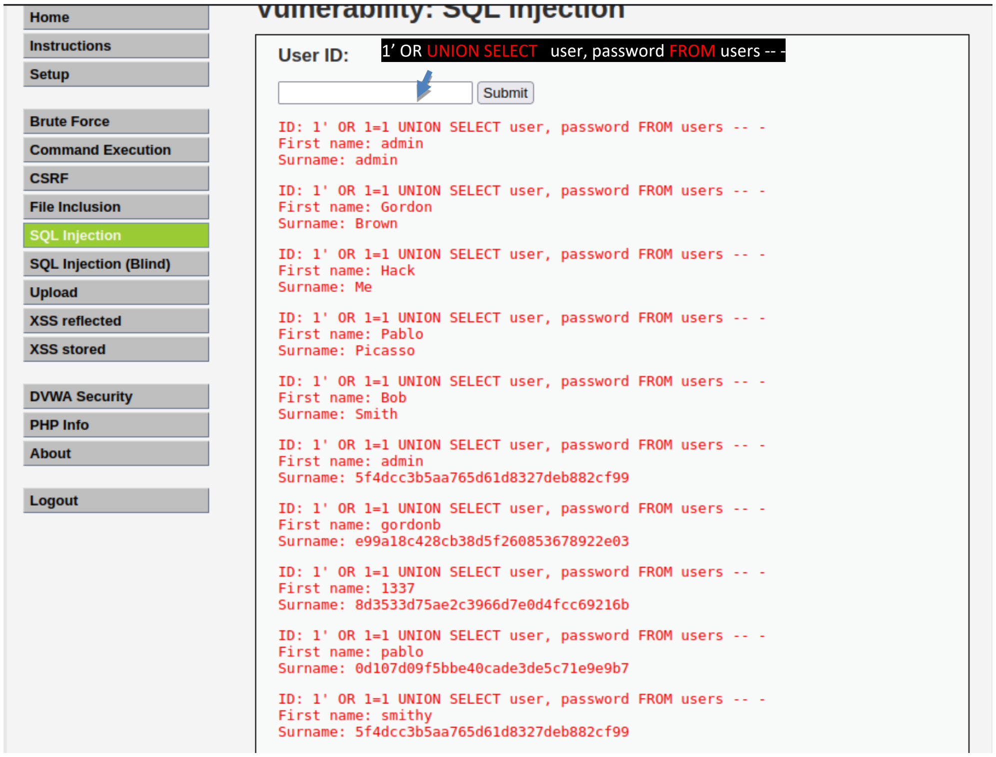
Database name retrieval.

This query filters a bunch of not useful output by inputting the targeted table 'dvwa' now we have a definite target table 'users'.



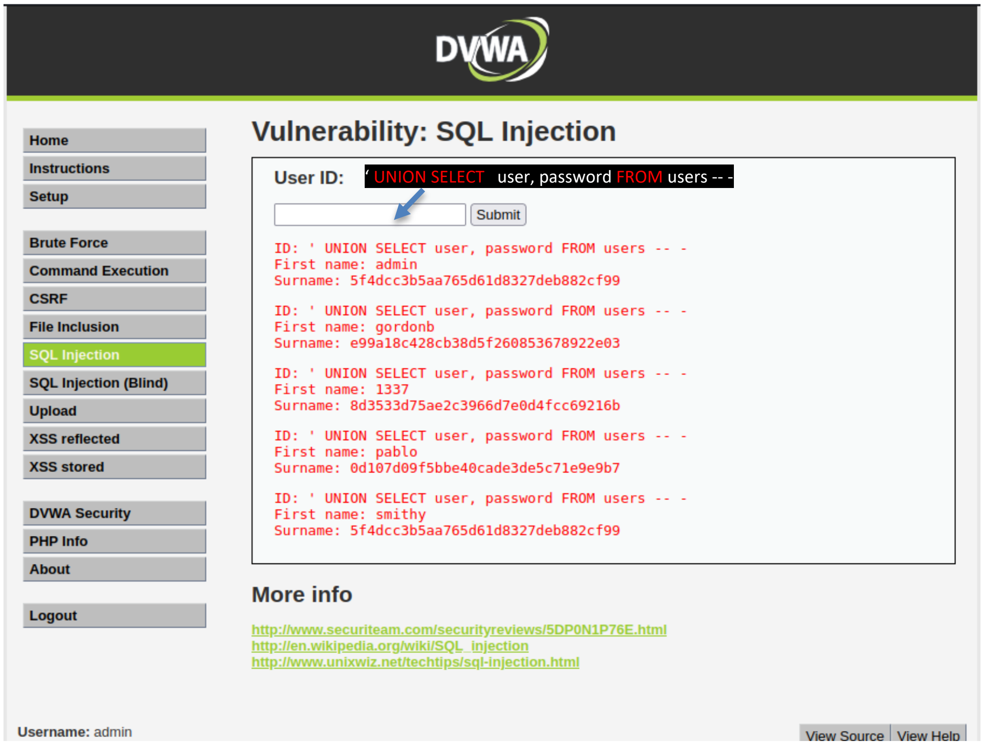
Getting columns from the table.

Here we retrieve the names of every column present in the database table users' and interestingly we have the column that somewhat signifies username 'user' and password 'password' which is of utmost interest.



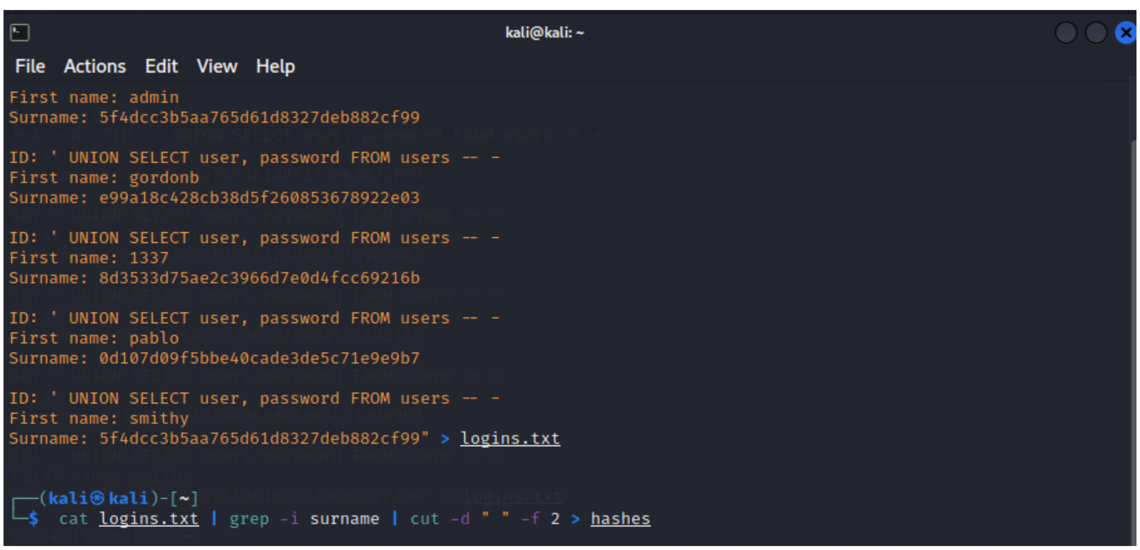
Login hash

Things begin to get interesting as with the query we have retrieved login details ‘user’ and ‘password’ which indicate further compromise.



Logins

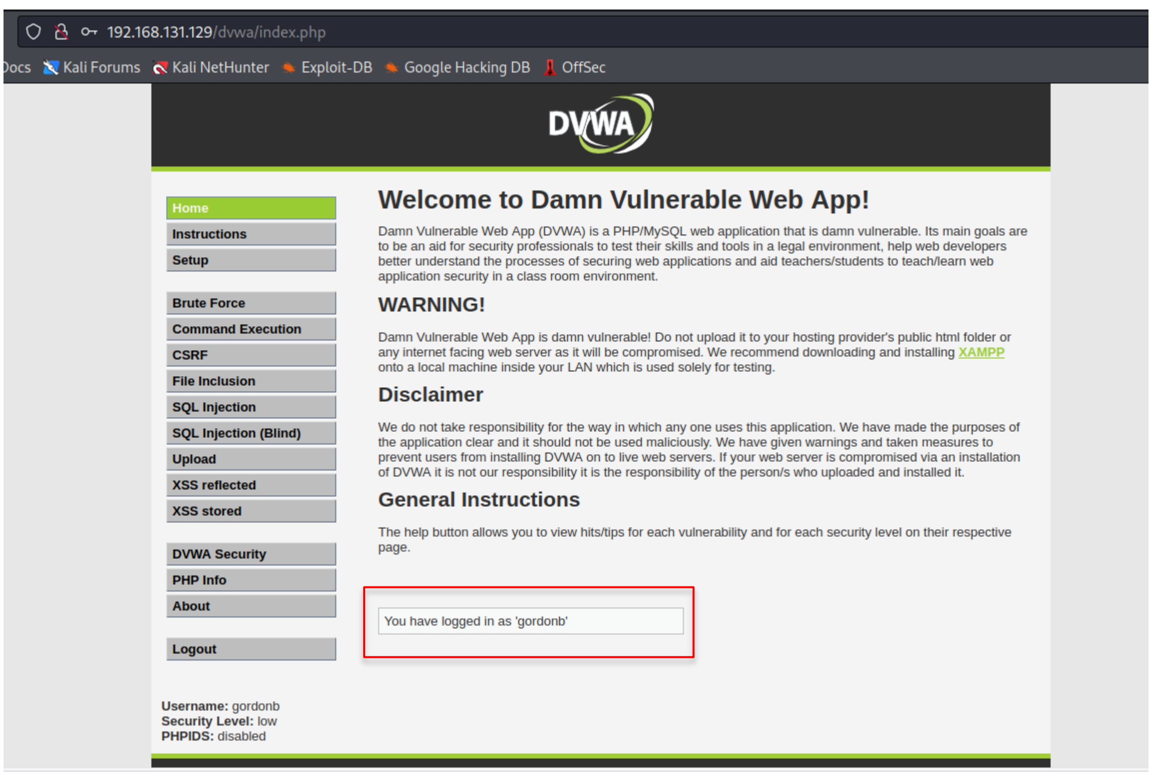
Streamlining the payload, we have just the user and hashed passwords, and it's time to crack them.



Cracking with john.

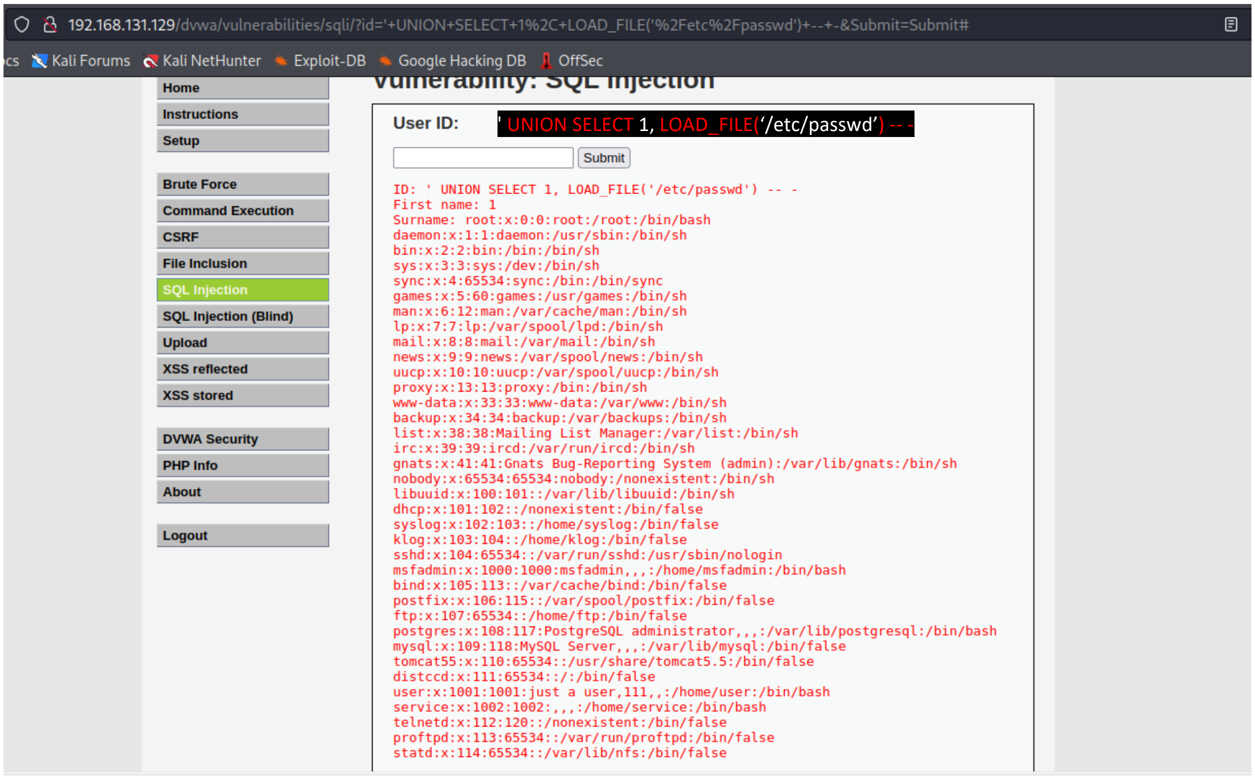
Cracking with john.

Since the password hashes for the web application have been successfully cracked, you can now attempt to log in as the user "gordonb" with the password "abc123," just as you already know for the admin being password.



Login as a user gotten from the database.

Now we have successfully logged in as user "gordonb" impersonating Gordon Brown from the data stolen from the database.



Quick test (Very vulnerable)

One quick trial to see if we have access to any system file by directory traversal, and this shows success, we could use usernames with some script engine to brute force into the system.

SQL Injection Attack Mitigation Strategies

Secure Implementation

Prepared Statements and Input Validation

Study by Ahmed and Uddin (2020) carried out SQLi against eleven organisation's web application and successfully penetrated three, the researcher's emphasis developer's requirement to specify the SQL code before passing individual parameters to parameterized queries. This approach enables the database to distinguish code and data, despite the user input supplied. Prepared statements guarantee that the intention of a query cannot be altered by an attacker, even if they attempt to insert SQL commands.

```
// This should REALLY be validated too
String custname = request.getParameter("customerName");
// Perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Prepared statement in java (TechTarget, 2022).

Parameterised queries ensure the proper distinction between queries and data, while input validation and sanitisation verify user-supplied data and filter special characters. Data encryption, and least privilege principles help mitigate the impact of successful SQLIA by limiting access to sensitive information.

```
String tableName;
switch(PARAM):
  case "Value1": tableName = "fooTable";
                 break;
  case "Value2": tableName = "barTable";
                 break;
  ...
  default      : throw new InputValidationException("unexpected value provided"
                                                   + " for table name");
```

Input Validation implementation in java

Escape Sequence

A technique that relies on escaping string and URL parameters to prevent SQLIA. URL parameters and user input columns were identified and grouped based on their query types by some researchers. The vulnerability arises from querying the database without any prior user query validation or input filtering. To address this, the study implemented handling solutions using regular expressions, prepared statements, and MySQL escape string techniques. The improved application system with these solutions was tested, confirming its security and resistance to SQLi.

```
$stmt = $pdo->prepare('SELECT * FROM employees WHERE name = :name');
$stmt->execute(array('name' => $name));
foreach ($stmt as $row) {
    // do something with $row
}
```

Escape sequence using PDO in MySQL

As a last resort when other preventive measures are not feasible, escaping user input before using it in a query is recommended. It may not fully prevent SQL injection in all cases, but by following proper escape procedures for the specific database, it helps differentiate user input from developer-written SQL code, reducing the risk of SQLIA.

Secure Coding

A secure coding approach enables developers to prevent SQLIA and enhance security in business applications. Some research primarily focused on sanitising queries by tokenising users' input, the tokenised strings are then compared with a stored dictionary, which output a possible attempt at the injection or safe query. The algorithm generated for the work was tested with input data revealing high accuracy detection with minimal false positives.

Least Privilege Principle

To enhance the security of your database, it is recommended to implement the principle of Least Privilege in conjunction with primary defenses to strengthen database security. Although SQL injection poses a significant threat, parameter manipulation can also result in unauthorised data access even by authorised application access. Minimising application privileges helps to mitigate the risk of such unauthorised access attempts. It is crucial to restrict privileges for each database account and refrain from granting admin-level access to application accounts. Granting excessive privileges for the sake of convenience can expose the system to substantial risks.

Stored Procedures

Stored procedures can provide protection against SQL injection, although they are not entirely foolproof. In most stored procedure languages, developers build SQL statements with parameters, which are automatically handled as parameterized queries. The key difference is that stored procedures store the SQL code within the database and are called from the application.

```
// This should REALLY be validated
String custname = request.getParameter("customerName");
try {
  CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}")
  cs.setString(1, custname);
  ResultSet results = cs.executeQuery();
  // … result set handling
} catch (SQLException se) {
  // … logging and error handling
}
```

Figure 5.1.5.1 Stored procedures in java (TechTarget, 2022).

Penetration Testing and Patching

To stay ahead of evolving threats and prevent the injection of malicious data into SQL databases, the importance of active penetration testing is paramount. This approach allows organizations to proactively identify vulnerabilities and mitigate potential risks. Some researchers developed VulnScan, an advanced web vulnerability scanner that utilizes a combination of techniques to bypass detection and prevention systems like WAF. The scanner generates payloads using an algorithm and outperformed the OWASP ZAP tool by identifying twice as many SQLi vulnerabilities in a test against a vulnerable web application. Identifying and addressing vulnerabilities in webservers is an effective strategy for enhancing overall security.

5.2 Machine Learning Solutions

According to Tripathy, Gohil and Halabi (2020), it can be said that intrusion detection systems operate based on predefined rules, which enables the circumvention by attackers which makes Artificial Intelligence (AI) superior as they are probabilistic utilising statistical models. AI, encompassing Machine Learning (ML) and Deep Learning (DL), has gained significant traction in cybersecurity domains. In contrast to conventional signature-based and rule-based cybersecurity strategies, AI-driven approaches offer enhanced efficiency and advanced capabilities for detecting and defending against cyber-attacks and threats

Several research works have demonstrated the use of ML for SQLIA detection and prevention, Sivasangari, Jyotsna, and Pravalika (2021) employed the AdaBoost algorithm to train a model for SQLIA detection. The dataset was categorized into weak and strong stumps, and the model was evaluated based on input validation. AdaBoost outperformed other classification algorithms, achieving a precision and accuracy of 0.97% in predicting injection attacks.

Web Application Firewall

Harefa et al. (2021) proposed a server-side WAF architecture called SEA WAF to protect against various types of SQLIA in e-commerce solutions. The SEA WAF outperformed Cloudflare and Barikode by effectively detecting and mitigating all attack types, while Cloudflare only defended against a subset of attacks.

Mukhtar and Azer (2020) conducted research demonstrating the effectiveness of deploying a WAF in preventing and mitigating SQLIA. WAFs act as intermediary reverse proxies, intercepting and filtering requests to identify and block malicious SQL injection attempts. The researchers observed a significant reduction in injectable vulnerabilities after deploying the Mod Security WAF in an Apache webserver, even during aggressive injection testing with SQLMap.