

Penetration Testing Report

Full Name: Mantone Malikhetla

Introduction

This report document hereby describes the proceedings and results of a Black Box security assessment conducted against the **Week {2} Labs**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

1. Objective

The objective of the assessment was to uncover vulnerabilities in the **Week {2} Labs** and provide a final security assessment report comprising vulnerabilities, remediation strategy and recommendation guidelines to help mitigate the identified vulnerabilities and risks during the activity.

2. Scope

This section defines the scope and boundaries of the project.

Application Name	{Lab 1 SQL Injection}, {Lab 2 Insecure Direct Object References}
------------------	--

3. Summary

Outlined is a Black Box Application Security assessment for the **Week {2} Labs**.

Total number of Sub-labs: {16} Sub-labs

High	Medium	Low
{2}	{3}	{5}

High 2 - Number of Sub-labs with hard difficulty level

Medium 3 - Number of Sub-labs with Medium difficulty level

1. {Lab 1 SQL Injection}

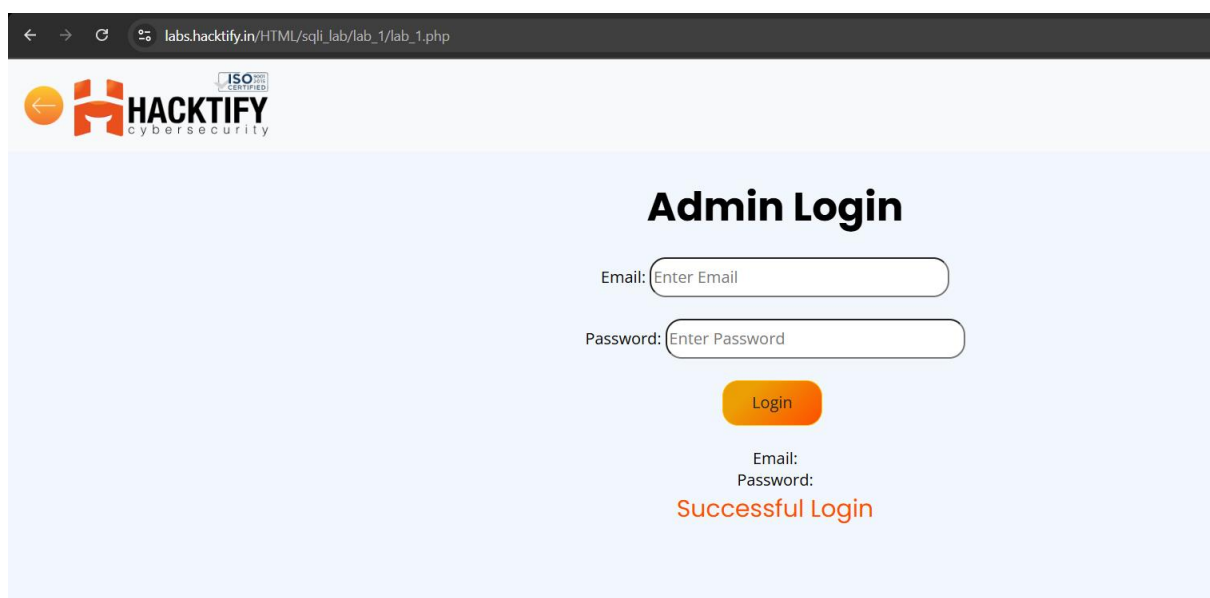
1.1. {Sub-lab-1 Strings & Errors Part 1!}

Reference	Risk Rating
{Sub-lab-1 Strings & Errors Part1!}	High
Tools Used	
Browser	
Vulnerability Description	
<p>The vulnerability occurs due to improper handling of user input in an SQL query. The input field for login credentials is vulnerable because the application does not properly sanitize user input before executing SQL queries.</p> <p>The login system likely has an underlying SQL query structured like this:</p> <pre>SELECT * FROM users WHERE email = 'user_input' AND password = 'user_password';</pre> <p>When the attacker enters ' OR '1'='1 -- in the email, since 1=1 is always true, the database returns user records, allowing unauthorized access.</p>	
How It Was Discovered	
Manual Analysis using payload ' OR '1'='1 -	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
<p>Bypass Authentication – Gain unauthorized access to admin/user accounts.</p> <p>Extract Sensitive Data – Fetch usernames, emails, and passwords stored in the database.</p> <p>Modify or Delete Data – If access privileges are high, the attacker can change or remove records.</p> <p>Compromise the Entire System – If combined with privilege escalation, the attacker may take over the database server.</p> <p>Lead to Data Breaches – Compliance violations (GDPR, HIPAA, etc.) can result in legal and financial penalties.</p>	
Suggested Countermeasures	
<p>1. Use Prepared Statements (Parameterized Queries) eg</p> <pre>\$stmt = \$pdo->prepare("SELECT * FROM users WHERE email = ? AND password = ?"); \$stmt->execute([\$email, \$password]);</pre> <p>2. Use Stored Procedures eg</p> <pre>CREATE PROCEDURE ValidateUser(IN user_email VARCHAR(100), IN user_pass VARCHAR(100)) BEGIN SELECT * FROM users WHERE email = user_email AND password = user_pass; END;</pre> <p>3. Implement Strong Input Validation eg Reject special characters, enforce length constraints</p> <p>4. Enforce length constraints eg use least privilege access for database users, separate accounts for read-only vs. write operations. restrict DROP, ALTER, and DELETE permissions.</p>	
References	
https://www.w3schools.com/sql/sql_injection.asp	

https://owasp.org/www-community/attacks/SQL_Injection
<https://www.imperva.com/learn/application-security/sql-injection-sqli/>
<https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16>
<https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



1.2. {Sub-lab-2 Strings & Errors Part 2!}

Reference	Risk Rating
{Sub-lab-2 Strings and Errors Part 2!}	High
Tools Used	
Browser	
Vulnerability Description	
The vulnerability occurs due to unsanitized user input in the URL parameter (id). The website directly uses the id parameter in an SQL query, making it vulnerable to SQL Injection. The backend likely has an SQL query like:	

```
SELECT email, password FROM users WHERE id = 'user_input';
```

Manipulation of the id parameter in the URL (?id=1) allows retrieval of sensitive information (admin email & password). This suggests that the application executes unsanitized SQL queries, exposing confidential data. Since the database contains a user with id=1, their credentials are displayed.

How It Was Discovered

Manual Analysis by appending ? id=1 OR 1=1 -- into URL

Vulnerable URLs

https://labs.hacktify.in/HTML/sql_i_lab/lab_2/lab_2.php

Consequences of not Fixing the Issue

Unauthorized Data Access – Attackers can dump user credentials, emails, and other sensitive data.

Account Takeover – If passwords are weak or not hashed properly, attackers can log in as admin.

Full Database Compromise – Using UNION queries, attackers can extract entire tables.

Data Manipulation – Attackers may update, delete, or insert new data.

System Takeover – If an attacker exploits command execution via SQLi, they may get server access.

Suggested Countermeasures

1. Use Prepared Statements (Parameterized Queries)
2. Use Input Validation & Data Type Enforcement
3. Implement Least Privilege Database Access

References

https://www.w3schools.com/sql/sql_injection.asp

https://owasp.org/www-community/attacks/SQL_Injection

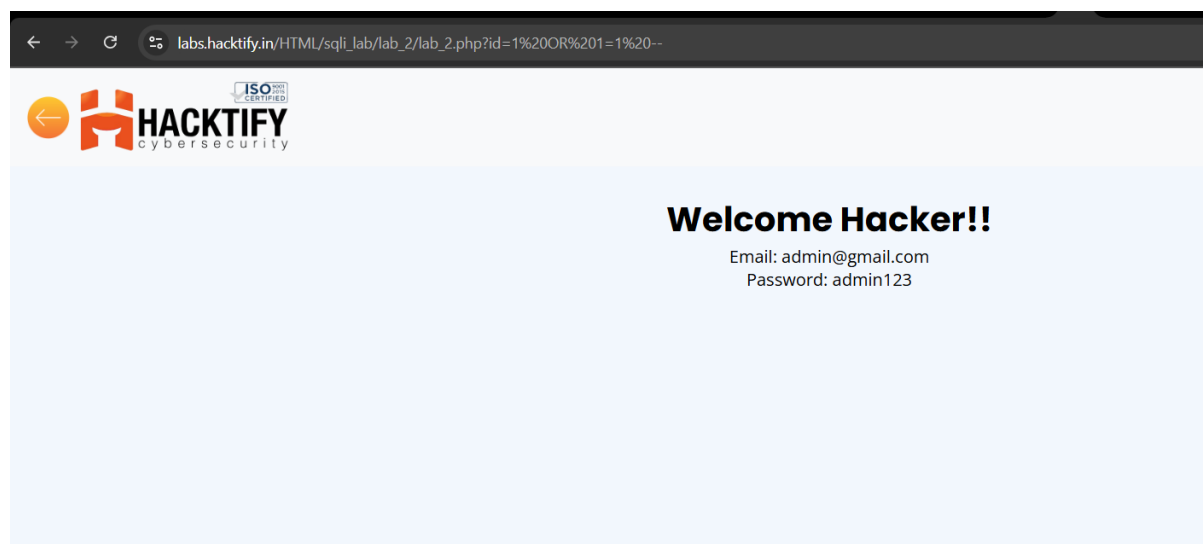
<https://www.imperva.com/learn/application-security/sql-injection-sqli/>

<https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16>

<https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab

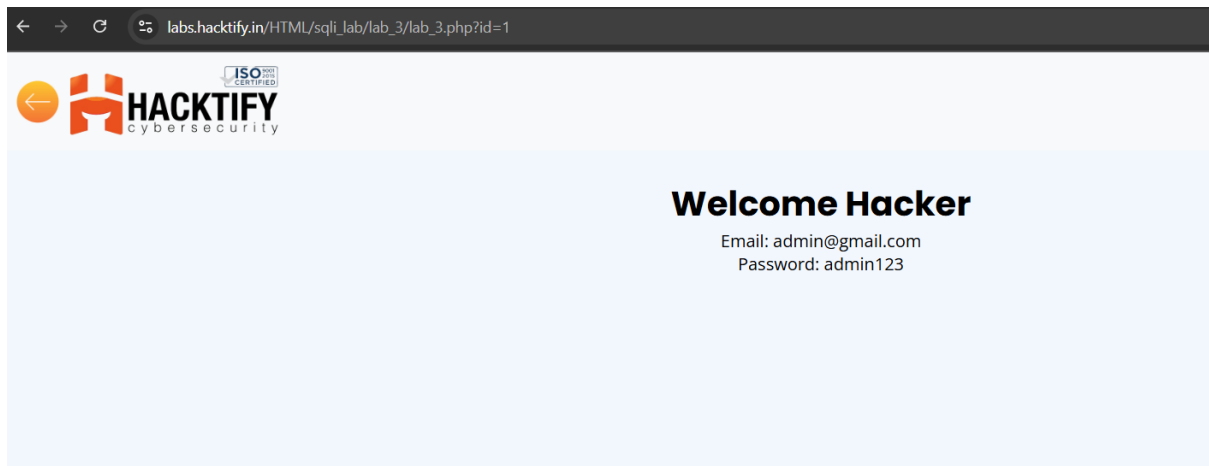


1.3. {Sub-lab-3 Strings & Errors Part 3!}

Reference	Risk Rating
{Sub-lab-3 Strings & Errors Part3!}	High
Tools Used	
Browser	
Vulnerability Description	
<p>The vulnerability occurs due to unsanitized user input in the URL parameter (id). The website directly uses the id parameter in an SQL query, making it vulnerable to SQL Injection.</p> <p>The backend likely has an SQL query like: SELECT email, password FROM users WHERE id = 'user_input';</p> <p>Manipulation of the id parameter in the URL (?id=1) allows retrieval of sensitive information (admin email & password). This suggests that the application executes unsanitized SQL queries, exposing confidential data. Since the database contains a user with id=1, their credentials are displayed.</p>	
How It Was Discovered	
Manual Analysis by appending <i>id=1</i> in URL	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sql_i_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
<p>Unauthorized Data Access – Attackers can dump user credentials, emails, and other sensitive data.</p> <p>Account Takeover – If passwords are weak or not hashed properly, attackers can log in as admin.</p> <p>Full Database Compromise – Using UNION queries, attackers can extract entire tables.</p> <p>Data Manipulation – Attackers may update, delete, or insert new data.</p> <p>System Takeover – If an attacker exploits command execution via SQLi, they may get server access.</p>	
Suggested Countermeasures	
<ol style="list-style-type: none">1. Use Prepared Statements (Parameterized Queries)2. Use Input Validation & Data Type Enforcement3. Implement Least Privilege Database Access	
References	
<p>https://www.w3schools.com/sql/sql_injection.asp</p> <p>https://owasp.org/www-community/attacks/SQL_Injection</p> <p>https://www.imperva.com/learn/application-security/sql-injection-sqli/</p> <p>https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16</p> <p>https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/</p>	

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



1.4. {Sub-lab-4 Let's Trick 'em!}

Reference	Risk Rating
{Sub-lab-4 Let's Trick 'em!}	High
Tools Used	
Browser	
Vulnerability Description	
<p>The vulnerability is an error-based SQL Injection in a login form. The application executes an unsanitized SQL query when a user submits credentials, allowing an attacker to manipulate the SQL logic using UNION-based injection.</p> <p>The vulnerable backend SQL query likely follows this format: SELECT email, password FROM users WHERE email = '\$email' AND password = '\$password';</p> <p>The UNION SELECT bypasses authentication by returning valid results. Since NULL values are placeholders, an attacker can replace them with database information. This means advanced exploitation can take place since now that we know there are 4 columns, we can extract sensitive information.</p> <p>To retrieve Database Name, the following can be used ' UNION SELECT NULL, database(), NULL, NULL --</p> <p>To retrieve Current User, the following can be used ' UNION SELECT NULL, user(), NULL, NULL --</p>	
How It Was Discovered	
Manual Analysis using payload ' UNION SELECT NULL, NULL, NULL, NULL --	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
<p>Full Database Exposure – Attackers can extract sensitive tables, user credentials, and configurations.</p> <p>Authentication Bypass – Attackers can log in as an administrator without valid credentials.</p> <p>Data Manipulation – The vulnerability may allow deleting, modifying, or inserting new records.</p>	

Server Takeover – If LOAD_FILE() or xp_cmdshell is enabled, an attacker might execute system commands.

Suggested Countermeasures

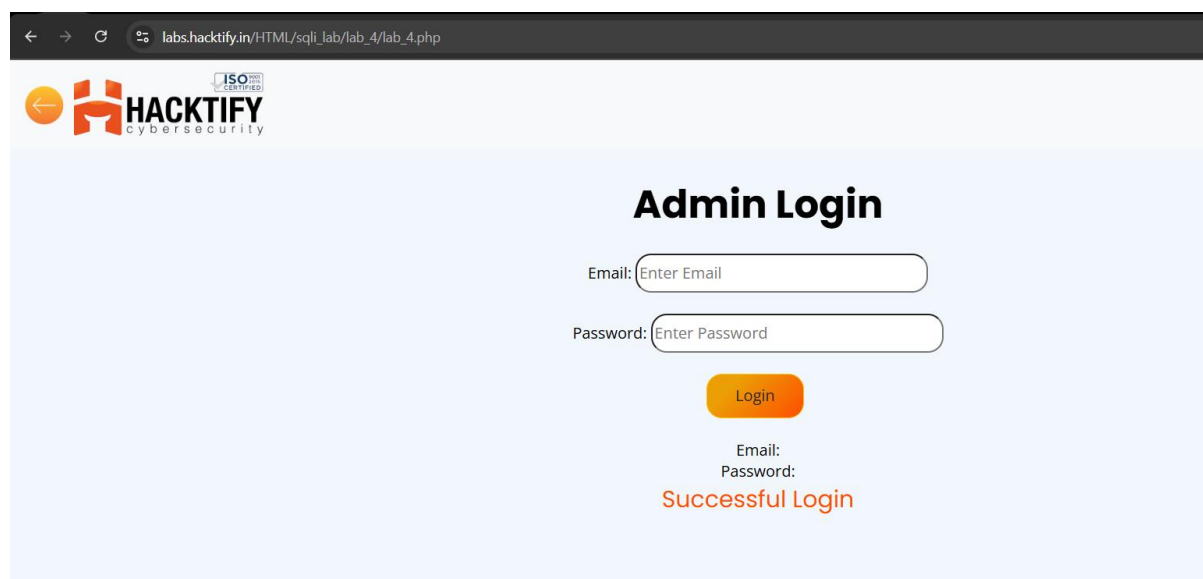
1. Use Prepared Statements (Parameterized Queries)
2. Use Input Validation & Data Type Enforcement
3. Implement Least Privilege Database Access

References

https://www.w3schools.com/sql/sql_injection.asp
https://owasp.org/www-community/attacks/SQL_Injection
<https://www.imperva.com/learn/application-security/sql-injection-sqli/>
<https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16>
<https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



1.5. {Sub-lab-5 Booleans and Blind!}}

Reference	Risk Rating
{Sub-lab-5 Booleans and Blind!}	High
Tools Used	
Browser	
Vulnerability Description	
The vulnerability is boolean-based Blind SQL Injection, a technique used when the application does not display direct SQL errors but behaves differently based on true/false conditions in the query.	

First part is to confirm that there is an SQL injection vulnerability using `?id=1' AND 1=1 --` -
 The next part is extracting the database name. The query can help determine if the database name starts with a letter greater than 'M' (N-Z), the query evaluates TRUE or if the database name starts with 'A-M', the query evaluates FALSE. The response will differ in each of these instances.

`database()` – Retrieves the current database name.
`SUBSTR(database(),1,1)` – Extracts the first character of the database name.
`ASCII(SUBSTR(database(),1,1))` – Converts that character into its ASCII value.
`> 77` – Compares the ASCII value to 77 (the letter 'M').
`--` – Comments out the rest of the SQL statement.

How It Was Discovered

Manual Analysis using payload `?id=1' AND ASCII(SUBSTR(database(),1,1)) > 77 --`

Vulnerable URLs

https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php

Consequences of not Fixing the Issue

Full Database Enumeration – The attacker can extract database names, tables, columns, and credentials.
 Authentication Bypass – If the application relies on SQL validation, the attacker can log in.
 Data Exfiltration – Attackers steal sensitive user information even when no direct errors appear.
 Slow But Effective – Blind SQL injection takes longer than error-based attacks, but works even when errors are hidden.

Suggested Countermeasures

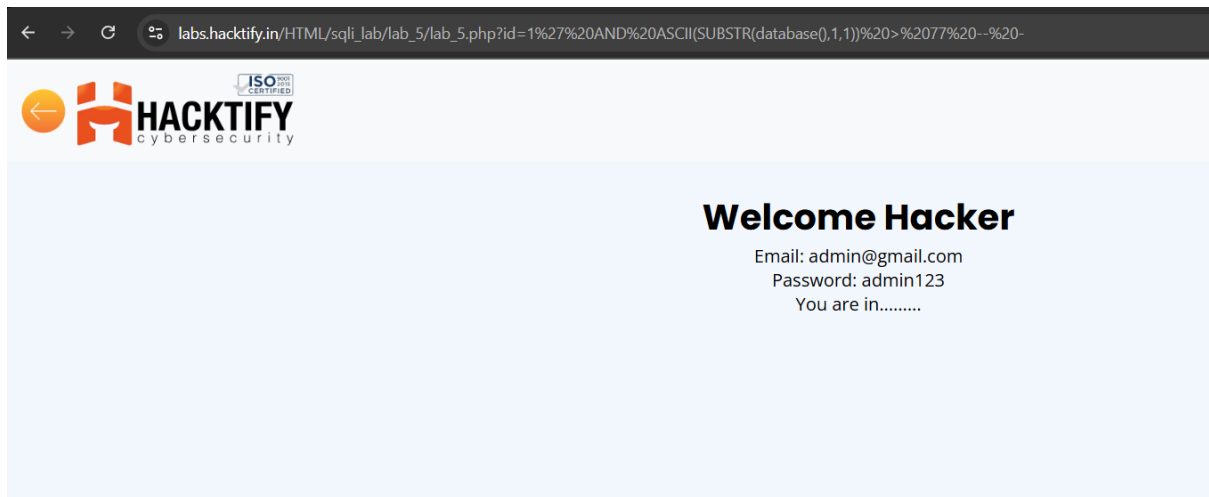
1. Use Prepared Statements (Parameterized Queries)
2. Implement Least Privilege Database Access
3. Implement Rate Limiting, since Boolean-based SQLi is slow, rate-limiting API requests delays attackers.

References

https://www.w3schools.com/sql/sql_injection.asp
https://owasp.org/www-community/attacks/SQL_Injection
<https://www.imperva.com/learn/application-security/sql-injection-sqli/>
<https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16>
<https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



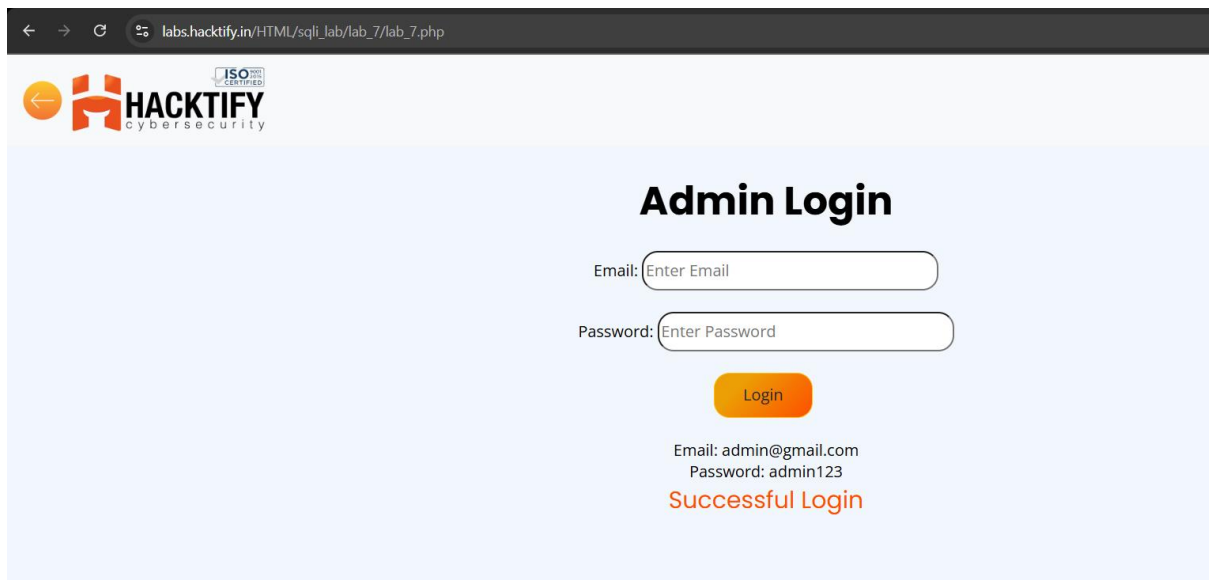
1.7. {Sub-lab-7 Errors and Post!}

Reference	Risk Rating
{Sub-lab-7 Errors and Post!}	High
Tools Used	
Browser	
Vulnerability Description	
Error-Based SQL Injection via POST requests, where direct feedback from the database helps the attacker construct payloads.	
Closes the intended SQL statement then 1=1 always evaluates to TRUE, allowing access without valid credentials. And using -- - to ignores the rest of the SQL statement to prevent syntax errors.	
How It Was Discovered	
Manual Analysis ' OR 1=1 -- -	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php	
Consequences of not Fixing the Issue	
Full Authentication Bypass – Any user, including admins, can be logged into without a password. Database Structure Exposure – Attackers can list databases, tables, and columns. User Credential Theft – If passwords are stored in plaintext, they can be stolen. Account Takeover – If hashes are used, an attacker might crack them offline.	
Suggested Countermeasures	
1. Use Prepared Statements (Parameterized Queries) 2. Use Input Validation & Data Type Enforcement 3. Implement Least Privilege Database Access	
References	
https://www.w3schools.com/sql/sql_injection.asp https://owasp.org/www-community/attacks/SQL_Injection https://www.imperva.com/learn/application-security/sql-injection-sqli/	

<https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16>
<https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



1.10. {Sub-lab-10 Oh Cookies!}

Reference	Risk Rating
{Sub-lab-10 Oh Cookies!}	High
Tools Used	
ZAP – Fuzzer	
Vulnerability Description	
Cookie manipulation by authentication where the server sets a cookie after logging in. Instead of injecting SQL via GET or POST parameters, manipulate the cookie value to exploit the vulnerability. Used OWASP ZAP to fuzz credentials, which helped discover how the application validates users.	
How It Was Discovered	
Automated Tools – Credentials found (admin:admin)	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can impersonate authenticated users. Account Takeover: If persistent cookies are used, long-term access is possible.	

Privilege Escalation: If an admin account is compromised, the attacker gains full control.

Suggested Countermeasures

1. Use Parameterized Queries – Prevent SQL injection.
2. Implement Strong Session Security – Set Secure, HttpOnly, and SameSite flags.
3. Rate-Limit Login Attempts – Prevent brute-force attacks.
4. Implement CAPTCHA – Stops automated credential fuzzing.

References

https://www.w3schools.com/sql/sql_injection.asp

https://owasp.org/www-community/attacks/SQL_Injection

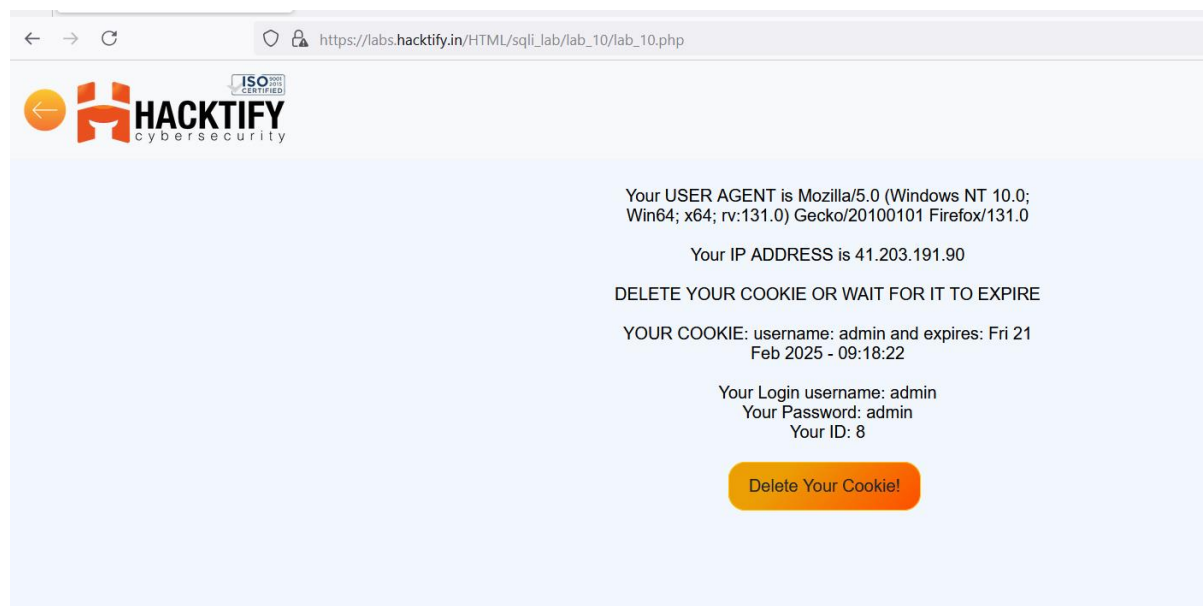
<https://www.imperva.com/learn/application-security/sql-injection-sqli/>

<https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16>

<https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



2. {Lab 2 Insecure Direct Object Reference}

2.1. {Sub-lab-1 Give me my amount!!}

Reference	Risk Rating
{Sub-lab-1 Give me my amount!!}	High
Tools Used	
Browser	
Vulnerability Description	
<p>This vulnerability allows unauthorized users to manipulate financial data without proper access controls. The login system allows access with weak or predictable credentials (admin:1234), hinting at a lack of strong authentication controls. After logging in, users can change financial values, such as balances or transaction details, which persist across sessions.</p> <p>A user can alter another user's balance (in this case admin) by manipulating request parameters. Changes remain even after logging out and back in, indicating a lack of proper authorization and verification mechanisms.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
<p>Perform unauthorized balance modifications – Attackers may increase their account balance, leading to financial fraud.</p> <p>Access other users' financial data – Leaking sensitive information like bank balances, credit limits, and transaction histories.</p> <p>Manipulate transactions – Attackers can approve, reject, or modify financial transactions without proper authorization.</p> <p>Bypass security controls – Exploiting this issue could escalate into privilege escalation or complete account takeovers.</p>	
Suggested Countermeasures	
<ol style="list-style-type: none">1. Implement Proper Access Controls - Use server-side authorization checks to verify that a user can only access their own records. Restrict access based on user roles and permissions.2. Replace direct numerical IDs (id=1438) with hashed or randomized identifiers (e.g., id=a9d8e7f6c4b2).3. Validate & Sanitize User Input - Block unauthorized ID modifications by validating user access before processing any financial changes. Apply parameterized queries and strict access control logic to prevent forced browsing.	
References	
<p>https://portswigger.net/web-security/access-control/idor</p> <p>https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html</p> <p>https://www.imperva.com/learn/application-security/insecure-direct-object-reference-idor/</p> <p>https://www.spiceworks.com/it-security/vulnerability-management/articles/insecure-direct-object-reference-idor/</p> <p>https://www.invicti.com/learn/insecure-direct-object-references-idor/</p>	

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab

The screenshot shows a web browser window with the address bar displaying `labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1438`. The page header features the Hacktify logo and an ISO 27001 CERTIFIED badge. The main content area is titled "User Profile" and contains a form with the following fields and values:

- Email:
- Credit Card:
- Transaction 1:
- Transaction 2:
- Transaction 3:

At the bottom of the form are two orange buttons: "Update" and "Log out".

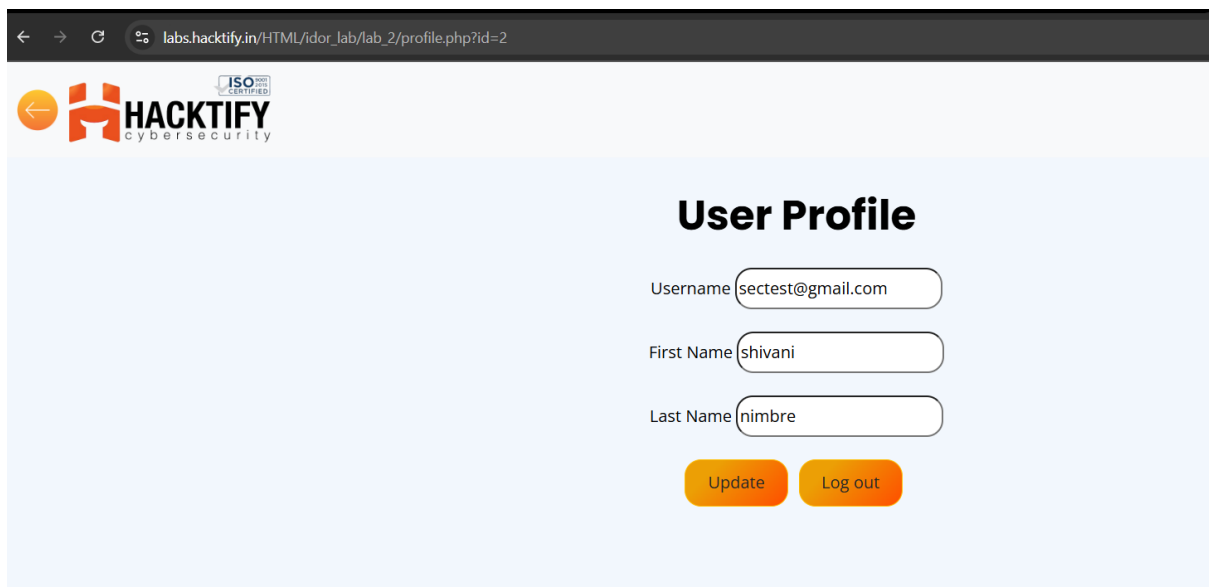
2.2. {Sub-lab-2 Stop polluting my params!}

Reference	Risk Rating
{Sub-lab-2 Stop polluting my params!}	High
Tools Used	
Browser	
Vulnerability Description	
This vulnerability allows attackers to access or modify other users' profiles by manipulating URL parameters. The application likely exposes user profile data through predictable URL parameters. Changing the id value (e.g., id=2) grants access to another user's profile, indicating a lack of authorization controls. This allows an attacker to view, edit, or delete another user's profile information without authentication.	
How It Was Discovered	
Manual Analysis – appending profile.php?id=2	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_2/lab_2.php	

Consequences of not Fixing the Issue
<p>Data Breach – Attackers can steal personal, financial, or private information from other users.</p> <p>Identity Theft – Malicious actors can modify victim profiles, impersonating them.</p> <p>Account Hijacking – If the attacker changes email settings or security questions, they can lock out legitimate users.</p> <p>Business Reputation Damage – If sensitive user information is leaked, it can lead to legal consequences.</p>
Suggested Countermeasures
<ol style="list-style-type: none"> 1. Implement Proper Access Controls 2. Enforce Role-Based Access Controls (RBAC) 3. Validate and Sanitize User Input
References
<p>https://portswigger.net/web-security/access-control/idor</p> <p>https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html</p> <p>https://www.imperva.com/learn/application-security/insecure-direct-object-reference-idor/</p> <p>https://www.spiceworks.com/it-security/vulnerability-management/articles/insecure-direct-object-reference-idor/</p> <p>https://www.invicti.com/learn/insecure-direct-object-references-idor/</p>

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



The screenshot shows a web browser window with the address bar displaying `labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=2`. The page header includes the Hacktify Cybersecurity logo and an ISO 27001 CERTIFIED badge. The main content area is titled "User Profile" and contains three input fields: "Username" with the value "sectest@gmail.com", "First Name" with the value "shivani", and "Last Name" with the value "nimbre". Below these fields are two orange buttons: "Update" and "Log out".

2.3. {Sub-lab-3 Someone changed my password!}

Reference	Risk Rating
{Sub-lab-3 Someone changed my password}	High
Tools Used	
Browser	
Vulnerability Description	
<p>This vulnerability allows unauthorized users to change passwords and even create new accounts by simply manipulating URL parameters. The web application does not properly enforce authentication and access controls when handling password reset requests. By appending ?username=admin to the URL, the application grants access to the password reset page for the "admin" account. This suggests the system relies solely on user-provided parameters (instead of session-based authentication) to determine whose password is being changed. As a result, an attacker can reset anyone's password or even create a new account by modifying the URL.</p>	
How It Was Discovered	
Manual Analysis – appending ?username=admin in URL	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
<p>Full Account Takeover – Attackers can lock out administrators or privileged users by changing their passwords.</p> <p>Creation of Rogue Admin Accounts – If new accounts can be created, attackers can maintain persistent access even after remediation.</p> <p>Complete System Compromise – If the attacker gains admin-level control, they can steal, modify, or delete all system data.</p> <p>Legal and Compliance Violations – Unauthorized access to sensitive user data can lead to GDPR, CCPA, or PCI DSS violations.</p>	
Suggested Countermeasures	
<ol style="list-style-type: none">1. Implement Strong Access Controls - authenticate the user before allowing password changes.2. Use Secure Password Reset Mechanisms - Require email confirmation or OTP-based verification before allowing password resets.3. Enforce Proper Authorization Checks - Restrict access to password reset pages.	
References	
<p>https://portswigger.net/web-security/access-control/idor</p> <p>https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html</p> <p>https://www.imperva.com/learn/application-security/insecure-direct-object-reference-idor/</p> <p>https://www.spiceworks.com/it-security/vulnerability-management/articles/insecure-direct-object-reference-idor/</p> <p>https://www.invicti.com/learn/insecure-direct-object-references-idor/</p>	

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



Change Password

Username

New Password

Confirm Password

Change

Back