Penetration Testing Report

Full Name: Sahil Naik
Program: HCPT

Date: 24/02/2025

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	**Lab 1 : Insecure Direct Object References,** **Lab 2 : 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
4	7	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

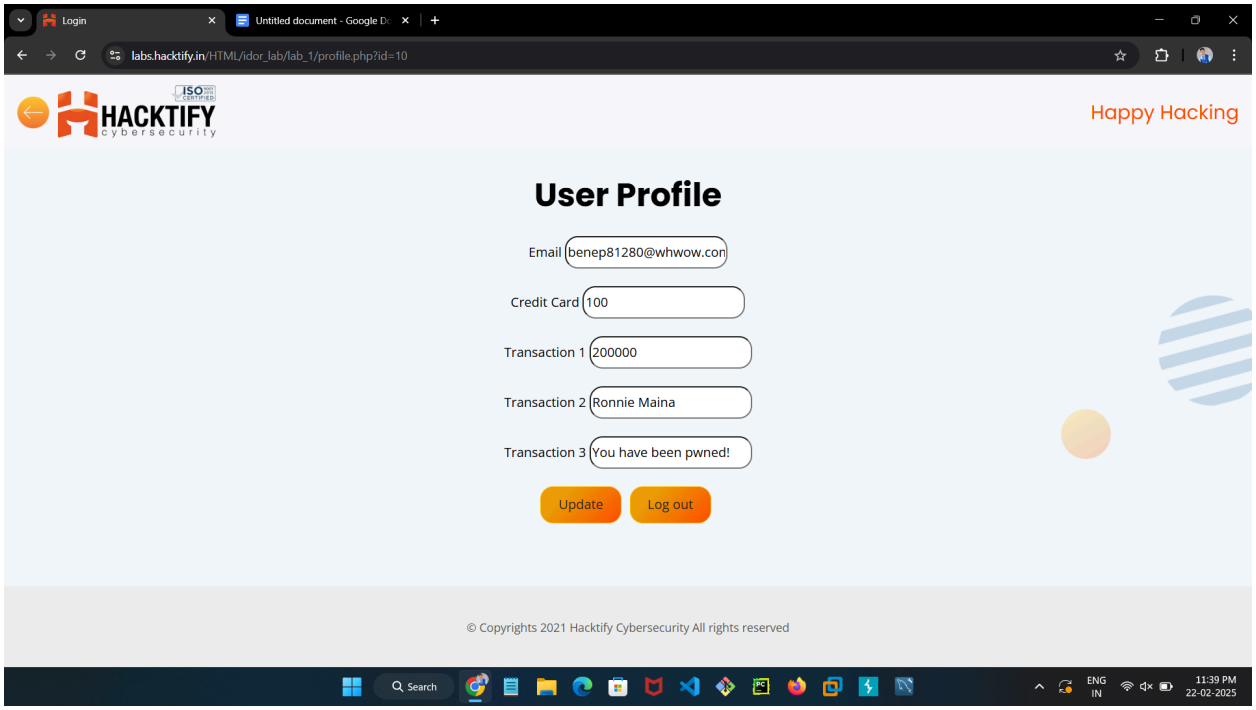
1.1. Give me my amount!!

Reference	**Risk Rating**
Give me my amount!!	**Low**
Tools Used	
Manual analysis, tampering the url id parameter	
Vulnerability Description	
IDOR is a vulnerability where an attacker accesses unauthorized resources by modifying object identifiers in requests. It occurs due to missing or improper access controls.	
How It Was Discovered / Steps to Reproduce	
1. Register one account and log in into it 2. In the url there's an id parameter, for eg id=10	

3. Change the id to 9 or 11 and gain access to user profiles of other accounts
Vulnerable URLs
https://labs.hacktify.in/HTML/idor_lab/lab_1/lab_1.php
Consequences of not Fixing the Issue
If an IDOR vulnerability is not fixed, attackers can access sensitive user data, leading to privacy breaches and data leaks. This can result in regulatory penalties, reputational damage, and potential legal consequences for the organization. Beyond data exposure, attackers may manipulate or delete user records, leading to account takeovers or service disruptions.
Suggested Countermeasures
1. Proper Authorization Checks – Implement server-side access controls to verify user permissions before serving any requested resource. 2. Use Indirect References – Replace direct object identifiers with secure alternatives like UUIDs or tokens to prevent enumeration. 3. Logging & Monitoring – Detect and respond to suspicious access patterns by logging requests and setting up alerts for anomalies.
References
https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html

Proof of Concept

Changed id from 13 to 10

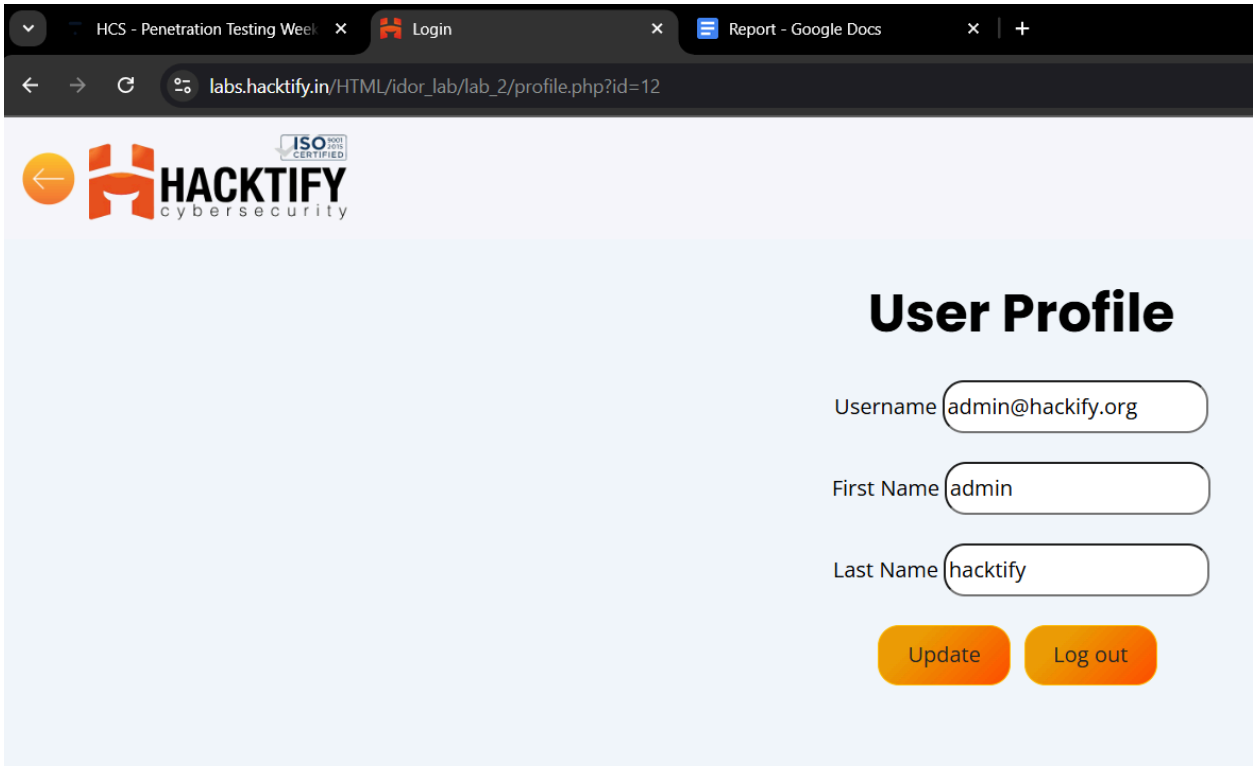


1.2. Stop polluting my params!

Reference	**Risk Rating**
Stop polluting my params!	**Medium**
Tools Used	
Manual analysis, tampering the url id parameter	
Vulnerability Description	
IDOR is a vulnerability where an attacker accesses unauthorized resources by modifying object identifiers in requests. It occurs due to missing or improper access controls.	

How It Was Discovered / Steps to Reproduce
1. Registered 2 accounts, 1 attacker 1 victim 2. Logged in to attacker and victim account with right credentials and noted the user id's 3. Logged in as attacker and on the update profile page, changed the id parameter to victims id and changed his details.
Vulnerable URLs
https://labs.hacktify.in/HTML/idor_lab/lab_2/lab_2.php
Consequences of not Fixing the Issue
1. Unauthorized Profile Modification – The attacker can alter the victim's personal details, leading to misinformation, account misuse, or identity fraud. 2. Escalation Risks – If profile updates include email or password changes, the attacker could hijack the victim's account entirely.
Suggested Countermeasures
1. Strict Access Controls – Ensure the server verifies if the logged-in user has permission to modify the requested profile. 2. Session-Based Identification – Use session tokens instead of user-controlled IDs for authentication and updates. 3. Audit Logs & Alerts – Track user activities and flag unusual ID modifications for investigation.
References
https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html

Proof of Concept



1.3. Someone changed my Password 🙀!

Reference	**Risk Rating**
Someone changed my Password 🙀!	**Medium**
Tools Used	
Manual analysis, tampering the url id parameter	
Vulnerability Description	
IDOR is a vulnerability where an attacker accesses unauthorized resources by modifying object identifiers in requests. It occurs due to missing or improper access controls.	
How It Was Discovered / Steps to Reproduce	

1. Create an account and log in.
2. Navigate to the update password page.
3. Identify the `username` parameter in the URL.
4. Modify the `username` value to another user's ID.
5. Access another user's account details.
6. Change their password successfully.

Vulnerable URLs

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

Consequences of not Fixing the Issue

1. Account Takeover – The attacker can reset passwords and lock victims out of their accounts.
2. Data & Service Abuse – Compromised accounts can be misused for fraud, impersonation, or further attacks.

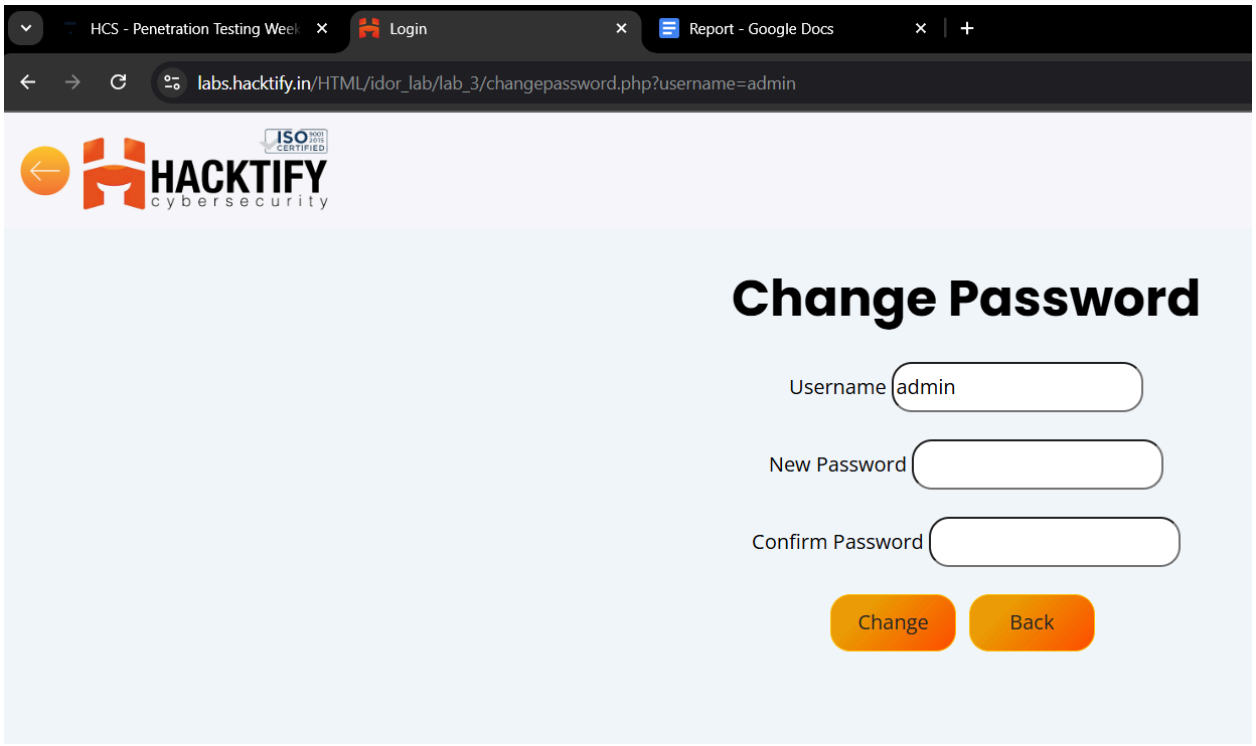
Suggested Countermeasures

1. Authorization Checks – Verify users can only update their own passwords.
2. Session-Based Identity – Use session tokens instead of user-controlled id parameters.
3. Rate Limiting & Alerts – Detect and block suspicious id modifications.

References

https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html

Proof of Concept

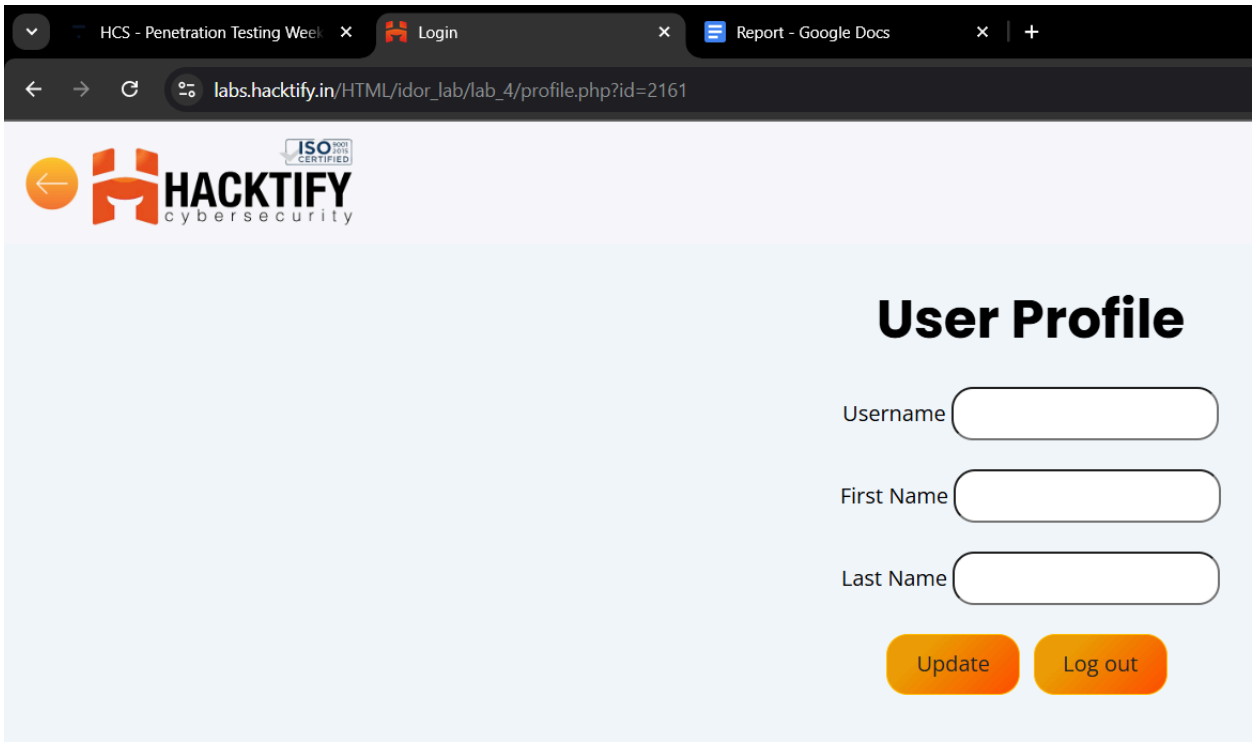


1.4. Change your methods!

Reference	**Risk Rating**
Change your methods!	**Medium**
Tools Used	
Manual analysis, tampering the url id parameter	
Vulnerability Description	
IDOR is a vulnerability where an attacker accesses unauthorized resources by modifying object identifiers in requests. It occurs due to missing or improper access controls.	
How It Was Discovered / Steps to Reproduce	

1. Registered first acc got id 2160 (attacker acc) 2. Registered second acc got id 1946 (victim acc) 3. Logged in from the acc with id 2160 and changed the id to 1946 and updated user details, then logged out and logged in using the updated details on the victim account.
Vulnerable URLs
https://labs.hacktify.in/HTML/idor_lab/lab_4/lab_4.php
Consequences of not Fixing the Issue
Full Account Takeover – The attacker gains control of the victim's account by modifying its details. Identity & Data Theft – The attacker can misuse the victim's account, access sensitive data, or perform malicious actions.
Suggested Countermeasures
1. Enforce Access Controls – Ensure users can only modify their own account details. 2. Use Session-Based Authentication - Authenticate users via session tokens, not user-controlled IDs. 3. Enable Logging & Alerts – Monitor and flag unauthorized ID modifications.
References
https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html

Proof of Concept



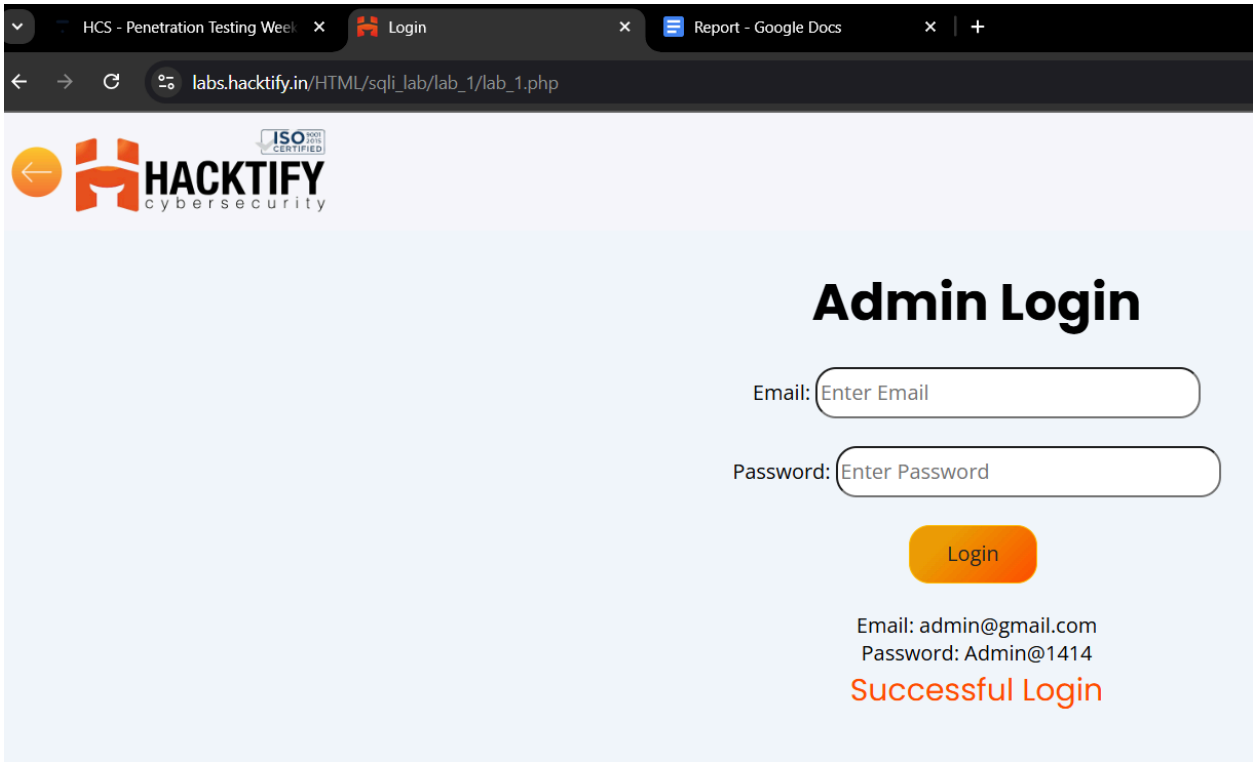
2. SQL Injection

2.1. Strings & Errors Part 1!

Reference	**Risk Rating**
Strings & Errors Part 1!	**Low**
Tools Used	
Manual analysis, using sqli payloads in the input fields	
Vulnerability Description	
Boolean-Based SQLi occurs when an attacker injects conditions that always evaluate to TRUE or False	

and observes changes in application behavior.
How It Was Discovered / Steps to Reproduce
1. Add payload in email and password field : admin" or "1"="1 2. Log in, and it gets triggered
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php
Consequences of not Fixing the Issue
Unauthorized Access – Attackers can bypass login and gain control of user accounts. Data Theft & Manipulation – Sensitive information can be stolen, altered, or deleted.
Suggested Countermeasures
1. Use Prepared Statements – Prevent SQL injection by properly parameterizing queries. 2. Input Validation & Escaping – Sanitize user inputs to block malicious characters. 3. Least Privilege Principle – Limit database permissions to minimize damage from attacks.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

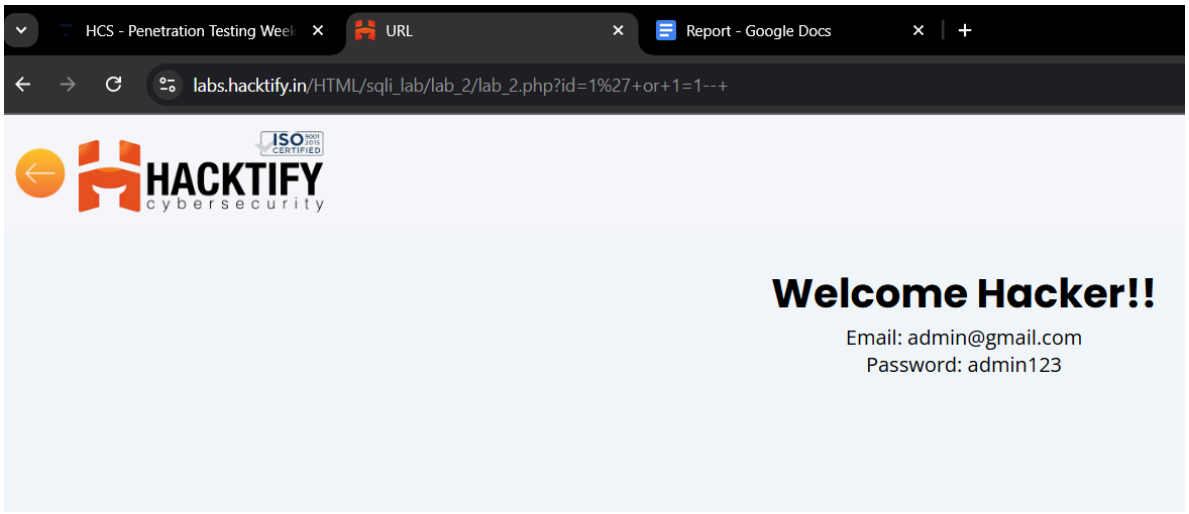


2.2. Strings & Errors Part 2!

Reference	**Risk Rating**
Strings & Errors Part 2!	**Low**
Tools Used	
Manual analysis, using sqli payloads in the input fields	
Vulnerability Description	
Boolean-Based SQLi occurs when an attacker injects conditions that always evaluate to TRUE or False and observes changes in application behavior.	
How It Was Discovered / Steps to Reproduce	
1. Add payload in the id parameter of url field : id=1" or "1"="1	

2. Triggering and showing email id of admin and other users
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php
Consequences of not Fixing the Issue
1. Unauthorized Data Access – Attackers can view sensitive records without authentication. 2. Data Manipulation – Malicious queries can modify or delete database entries. 3. Account Compromise – Login bypass can lead to full control of user accounts.
Suggested Countermeasures
1. Use Prepared Statements – Prevent SQL injection by properly parameterizing queries. 2. Input Validation & Escaping – Sanitize user inputs to block malicious characters. 3. Least Privilege Principle – Limit database permissions to minimize damage from attacks.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

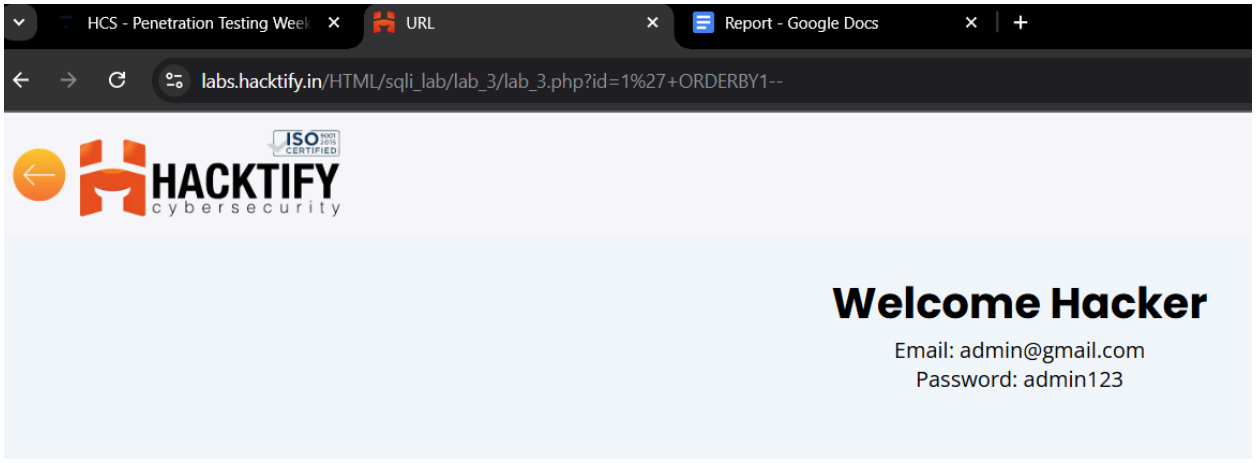


2.3. Strings & Errors Part 3!

Reference	Risk Rating
Strings & Errors Part 3!	**Low**
Tools Used	
Manual analysis, using sqli payloads in the input fields	
Vulnerability Description	
Union-Based SQLi occurs when an attacker uses the UNION SQL clause to combine results from multiple tables, extracting sensitive data. By modifying parameters like id=1 ORDER BY 1--, attackers can determine column count and inject UNION SELECT queries.	
How It Was Discovered / Steps to Reproduce	
1. Add payload in the id parameter of url field : id=1' +ORDERBY 1- - 2. Triggering and showing email id of admin and other users	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
1. Data Exposure – Attackers can retrieve sensitive information like admin credentials. 2. Account Takeover – Extracted credentials can be used to hijack accounts.	
Suggested Countermeasures	
1. Use Prepared Statements – Prevent UNION injection by parameterizing queries. 2. Limit Database Permissions – Restrict access to sensitive tables and data.	

3. Disable Error Messages – Prevent attackers from identifying table structures.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

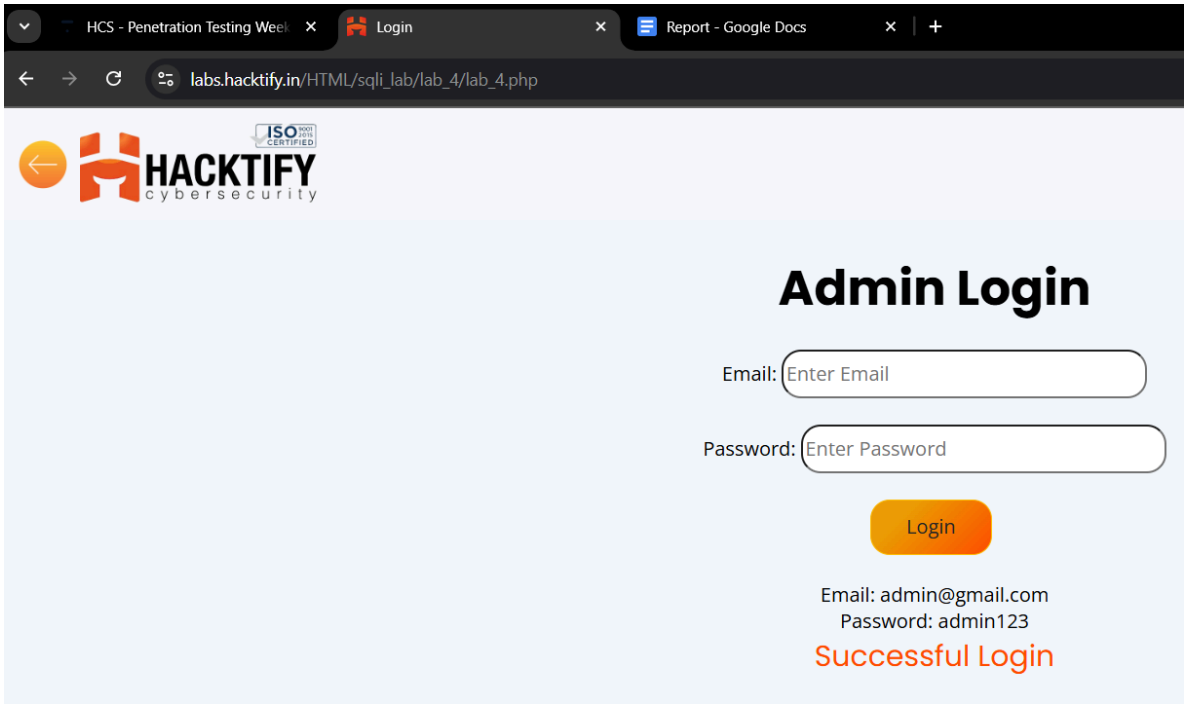


2.4. Let's Trick 'em!

Reference	**Risk Rating**
Let's Trick 'em!	**Medium**
Tools Used	
Manual analysis, using sqli payloads in the input fields	
Vulnerability Description	
Authentication Bypass SQL Injection.This attack occurs when login forms fail to properly validate user	

inputs, allowing attackers to inject SQL code to bypass authentication.
How It Was Discovered / Steps to Reproduce
1. Add payload in email and password field : '="or' 2. Successful log in as admin.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_4/index.php
Consequences of not Fixing the Issue
1. Admin Panel Access – Attackers can log in as an administrator without valid credentials. 2. Full System Compromise – Admin privileges may allow modification or deletion of data, user management, or further exploitation.
Suggested Countermeasures
1. Use Prepared Statements – Prevent input manipulation by parameterizing SQL queries. 2. Implement Strong Input Validation – Reject special characters and enforce strict input rules. 3. Use Multi-Factor Authentication (MFA) – Add an extra layer of security to prevent unauthorized logins.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

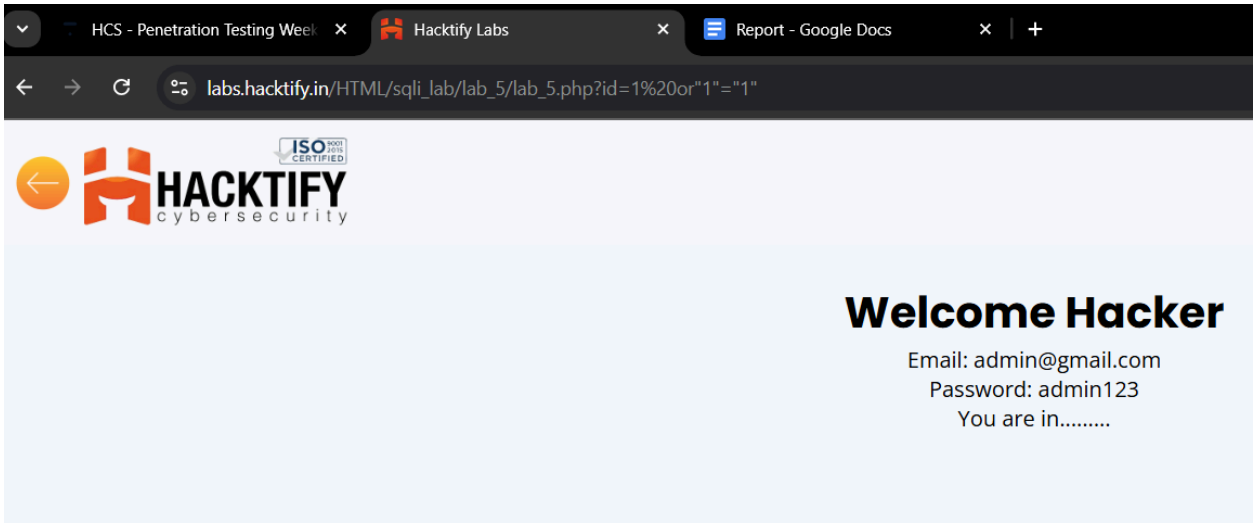


2.5.Booleans and Blind!

Reference	**Risk Rating**
Booleans and Blind!	**High**
Tools Used	
Manual analysis, using sqli payloads in the url fields	
Vulnerability Description	
Boolean-Based SQL Injection. This attack exploits improper input handling in SQL queries by injecting a condition that always evaluates to TRUE.	
How It Was Discovered / Steps to Reproduce	
1. Add payload in the id parameter of url field : id=1 or “1” = “1” 2. Triggering and showing email id of admin and other users	

Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php
Consequences of not Fixing the Issue
1. Unauthorized Data Access – Attackers can retrieve sensitive records by bypassing query restrictions. 2. Data Enumeration – Attackers can iterate through database entries, extracting confidential information.
Suggested Countermeasures
1. Use Prepared Statements – Prevent SQL injection by securely parameterizing queries. 2. Strict Input Validation – Block special characters and enforce expected input formats. 3. Limit Database Permissions – Restrict access to prevent unauthorized data extraction.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

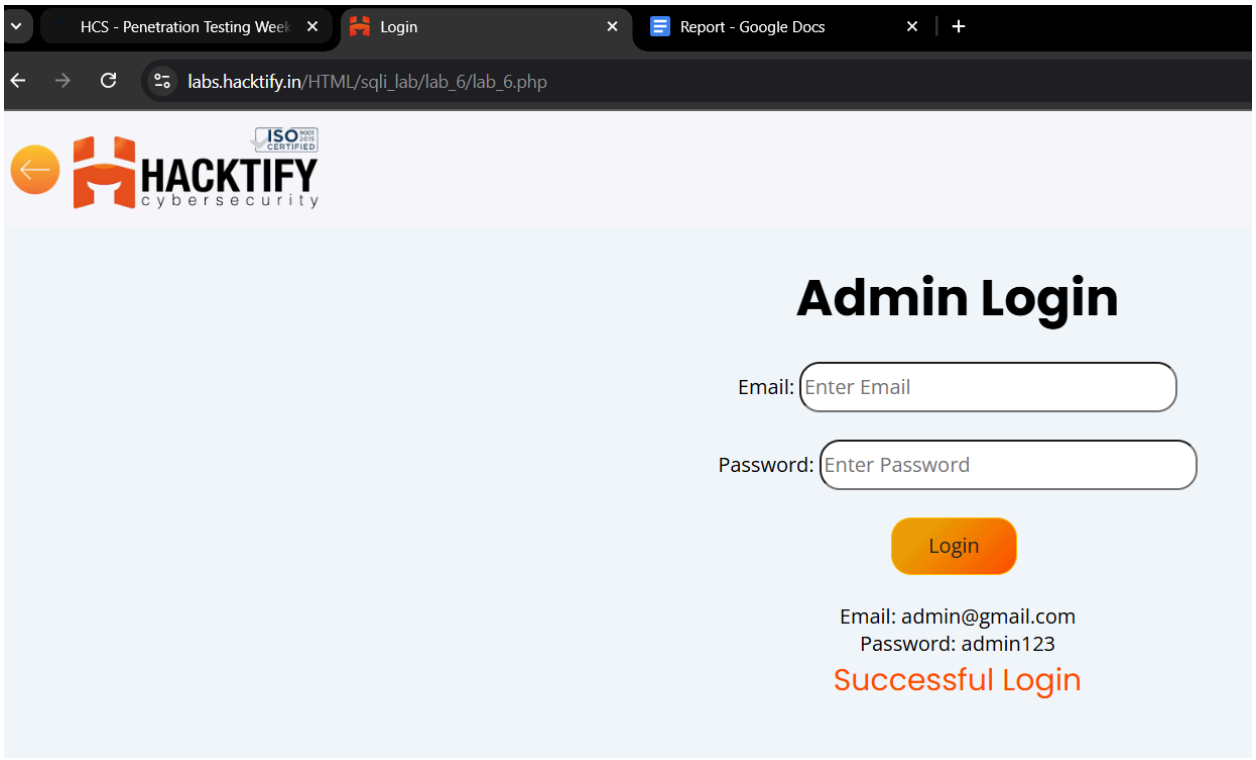


2.6. Error Based : Tricked

Reference	**Risk Rating**
Error Based : Tricked	**Medium**
Tools Used	
Manual analysis, using sqli payloads in the input fields	
Vulnerability Description	
Authentication Bypass SQL Injection.This attack occurs when login forms fail to properly validate user inputs, allowing attackers to inject SQL code to bypass authentication.	
How It Was Discovered / Steps to Reproduce	
3. Add payload in email and password field : ') or ('s' = 's and hack") or ("a"="a 4. Successful log in as admin.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_6/lab_6.php	
Consequences of not Fixing the Issue	
1. Admin Panel Access – Attackers can log in as an administrator without valid credentials. 2. Full System Compromise – Admin privileges may allow modification or deletion of data, user management, or further exploitation.	
Suggested Countermeasures	
1. Use Prepared Statements – Prevent input manipulation by parameterizing SQL queries. 2. Implement Strong Input Validation – Reject special characters and enforce strict input rules.	

3. Use Multi-Factor Authentication (MFA) – Add an extra layer of security to prevent unauthorized logins.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept



2.7. Errors and Post!

Reference	Risk Rating
Errors and Post!	**Low**
Tools Used	

Manual analysis, using sqli payloads in the input fields

Vulnerability Description

Authentication Bypass SQL Injection.This attack occurs when login forms fail to properly validate user inputs, allowing attackers to inject SQL code to bypass authentication.

How It Was Discovered / Steps to Reproduce

5. Add payload in email and password field : ' or '1'='1
6. Successful log in as admin.

Vulnerable URLs

https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php

Consequences of not Fixing the Issue

1. Admin Panel Access – Attackers can log in as an administrator without valid credentials.
2. Full System Compromise – Admin privileges may allow modification or deletion of data, user management, or further exploitation.

Suggested Countermeasures

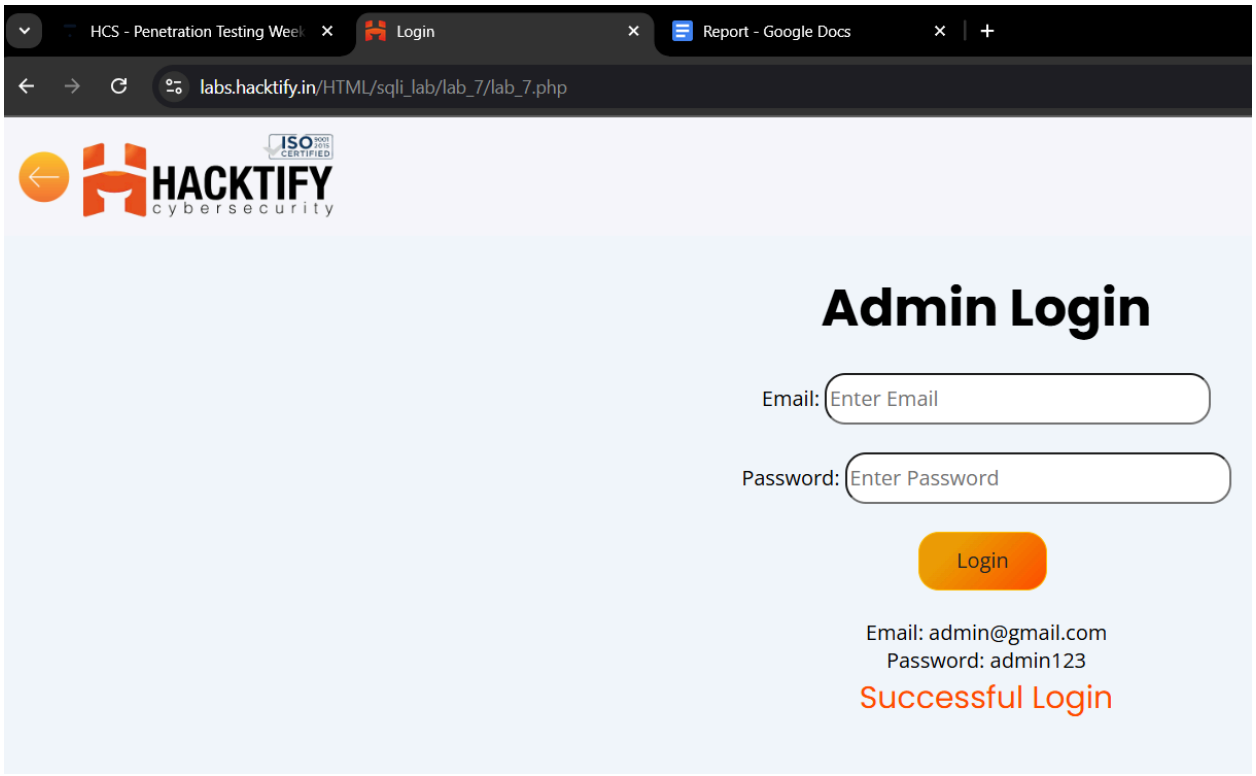
1. Use Prepared Statements – Prevent input manipulation by parameterizing SQL queries.
2. Implement Strong Input Validation – Reject special characters and enforce strict input rules.
3. Use Multi-Factor Authentication (MFA) – Add an extra layer of security to prevent unauthorized logins.

References

https://github.com/payloadbox/sql-injection-payload-list

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

Proof of Concept

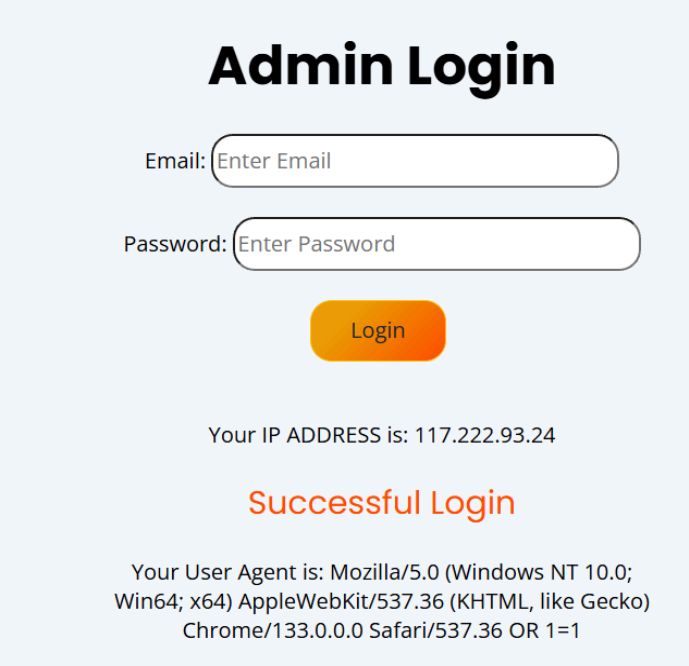


2.8. User Agents lead us!

Reference	**Risk Rating**
User Agents lead us!	**High**
Tools Used	
Burpsuite and sqli payloads	
Vulnerability Description	
SQL Injection via User-Agent Header. This vulnerability occurs when the application logs or displays the User-Agent header unsafely, allowing SQL injection. By intercepting the request with Burp Suite and	

appending a SQL payload to the User-Agent, the attacker can execute malicious queries.
How It Was Discovered / Steps to Reproduce
1. Access Admin Login Page – Visit the admin panel sign-in page where the User-Agent is displayed. 2. Intercept Request – Use Burp Suite to capture the login request. 3. Modify User-Agent Header – Append an SQLi payload (e.g., User-Agent: Mozilla/5.0" OR "1"="1). 4. Forward the Request – Send the modified request and observe SQL execution.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php
Consequences of not Fixing the Issue
1. Data Exposure – Attackers can extract sensitive admin data. 2. Privilege Escalation – Malicious queries may grant admin-level access.
Suggested Countermeasures
1. Sanitize Logged Inputs – Escape or filter user-supplied headers before processing. 2. Use Parameterized Queries – Prevent SQL execution from unsanitized input. 3. Restrict Debug Information – Avoid displaying untrusted data in the UI.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

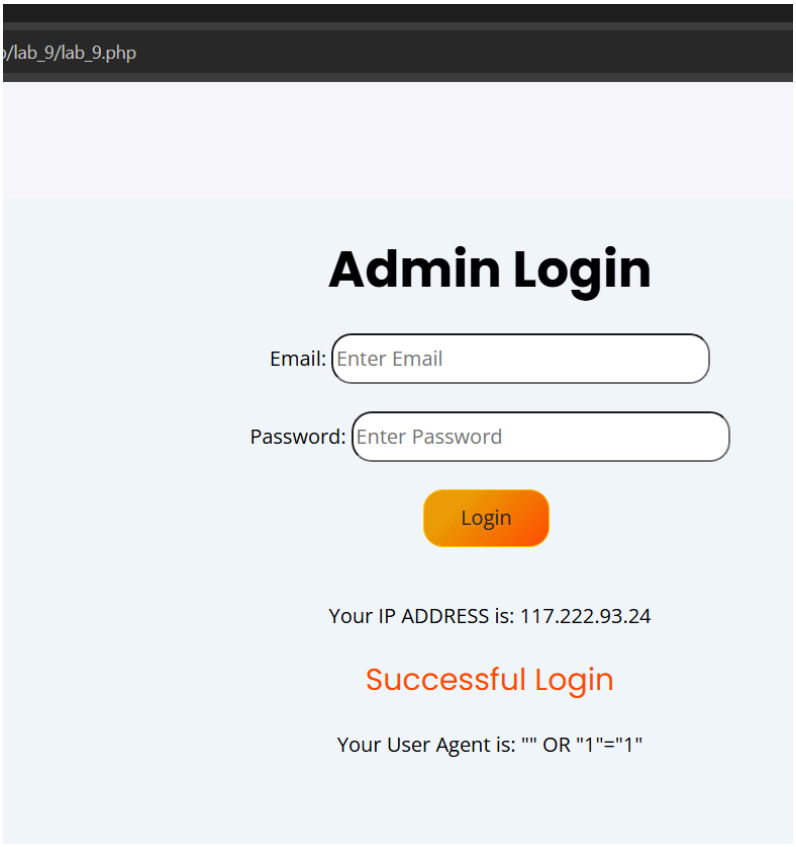


2.9. Referer lead us!

Reference	**Risk Rating**
Referer lead us!	**Medium**
Tools Used	
Burpsuite and sqli payloads	
Vulnerability Description	
SQL Injection via Referer Header. This vulnerability occurs when the application logs or displays the Referer header without sanitization, allowing SQL injection. By intercepting the request with Burp Suite and appending an SQL payload to the Referer, the attacker can execute arbitrary queries.	
How It Was Discovered / Steps to Reproduce	

1. Access Admin Login Page – Visit the admin panel where the Referer is processed or displayed. 2. Intercept Request – Use Burp Suite to capture the login request. 3. Modify Referer Header – Append an SQLi payload (e.g., Referer: http://site.com" OR "1"="1). 4. Forward the Request – Send the modified request and observe SQL execution.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php
Consequences of not Fixing the Issue
1. Data Breach – Attackers can extract sensitive database records. 2. Privilege Escalation – SQL queries may allow unauthorized admin access.
Suggested Countermeasures
1. Sanitize Logged Inputs – Properly escape or filter all user-supplied headers. 2. Use Parameterized Queries – Prevent SQL execution from injected inputs. 3. Limit Logging Exposure – Avoid displaying raw request headers in the UI.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

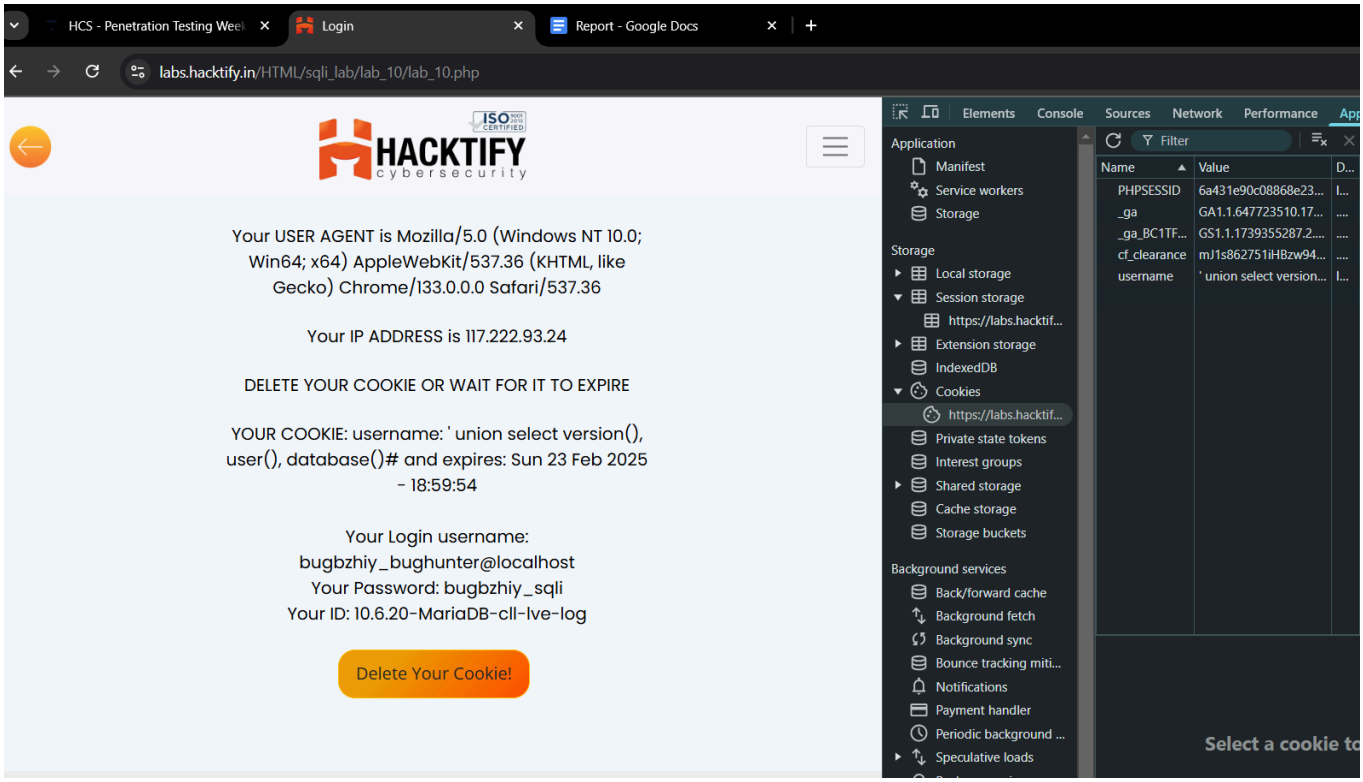


2.10. Oh Cookies!

Reference	**Risk Rating**
Oh Cookies!	**High**
Tools Used	
DevTools on Browser and sqli payloads	
Vulnerability Description	
SQL Injection via Cookie Header. This vulnerability occurs when the application displays cookie values in the UI without proper sanitization, allowing SQL injection. Using browser DevTools, an attacker can	

modify the cookie value with an SQLi payload, leading to database information disclosure.
How It Was Discovered / Steps to Reproduce
1. Inspect Cookies in DevTools – Open browser DevTools and navigate to the storage tab. 2. Modify Cookie Value – Replace it with an SQLi payload like: ' UNION SELECT version(), user(), database()# 3. Refresh the Page – Observe the database version, admin credentials, or other sensitive data displayed in the UI.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php
Consequences of not Fixing the Issue
1. Database Information Disclosure – Attackers can view version, user, and database details. 2. Privilege Escalation – Sensitive data can be used for further exploitation
Suggested Countermeasures
1. Sanitize and Validate Cookies – Never process user-controlled cookies without proper filtering. 2. Use Prepared Statements – Prevent direct execution of cookie values in SQL queries. 3. Restrict Debug Information – Avoid exposing raw cookie data in the UI.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

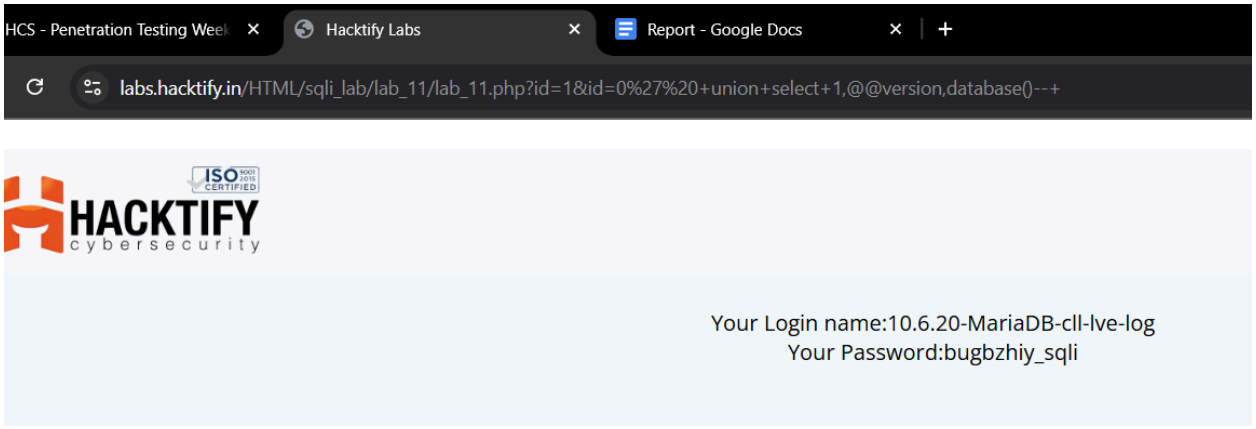


2.11.WAF's are injected!

Reference	**Risk Rating**
WAF's are injected!	**High**
Tools Used	
Manual analysis, using sqli payloads in the url fields	
Vulnerability Description	
WAF Bypassed SQL Injection via id Parameter.This attack targets a web application firewall protected site by obfuscating SQL payloads to bypass detection. By injecting SQL via the id using encoded or space-altered payloads, the attacker successfully extracts database information.	

How It Was Discovered / Steps to Reproduce
1. Identify WAF Protection – Test for basic SQLi payloads being blocked. 2. Craft WAF-Bypassing Payload – Use encoded or obfuscated payload like: id=1&id=0' +union+select+1,@@version,database()--+ 3. Send the Request – Observe the extracted database version and name in the response.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php
Consequences of not Fixing the Issue
1. Database Fingerprinting – Attackers can identify the database type and version for further exploitation. 2. Security Bypass – Successfully bypassing WAF protections weakens overall security defenses.
Suggested Countermeasures
Advanced WAF Rules – Implement strict SQL injection detection, including obfuscation techniques. Use Parameterized Queries – Ensure backend queries do not directly process user-controlled input. Monitor Logs & Anomalies – Detect and block suspicious payload patterns in real time.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept

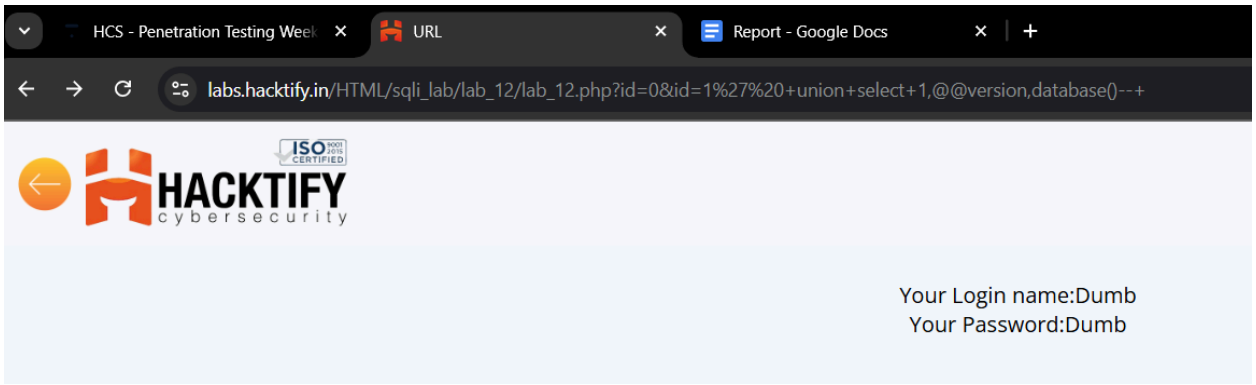


2.12.WAF's are injected Part 2!

Reference	**Risk Rating**
WAF's are injected Part 2!	**Medium**
Tools Used	
Manual analysis, using sqli payloads in the url fields	
Vulnerability Description	
WAF Bypassed SQL Injection via id Parameter.This attack targets a web application firewall protected site by obfuscating SQL payloads to bypass detection. By injecting SQL via the id using encoded or space-altered payloads, the attacker successfully extracts database information.	
How It Was Discovered / Steps to Reproduce	
1. Identify WAF Protection – Test for basic SQLi payloads being blocked. 2. Craft WAF-Bypassing Payload – Use encoded or obfuscated payload like: id=0&id=1' +union+select+1,@@version,database()--+ 3. Send the Request – Observe the extracted database version and name in the response.	

Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php
Consequences of not Fixing the Issue
1. Database Fingerprinting – Attackers can identify the database type and version for further exploitation. 2. Security Bypass – Successfully bypassing WAF protections weakens overall security defenses.
Suggested Countermeasures
Advanced WAF Rules – Implement strict SQL injection detection, including obfuscation techniques. Use Parameterized Queries – Ensure backend queries do not directly process user-controlled input. Monitor Logs & Anomalies – Detect and block suspicious payload patterns in real time.
References
https://github.com/payloadbox/sql-injection-payload-list https://owasp.org/www-community/attacks/SQL_Injection

Proof of Concept



Thank You!