Week 2 Penetration Testing Report

Full Name: Atharva Jagdale
Program: HCPT
Date: 24-02-2025

Introduction

This report hereby describes the proceedings and results of a Black Box security assessment conducted against **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 **Week 2 Labs**.

Total number of Sub-labs: 16 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**

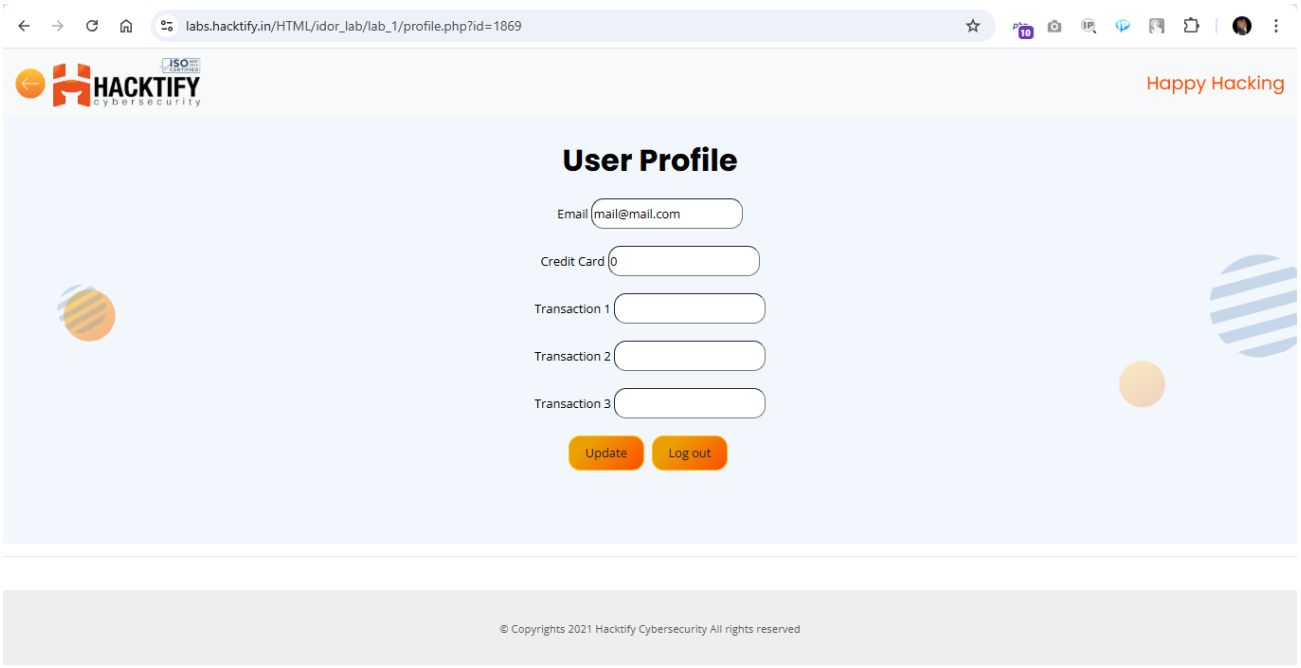
1. Insecure Direct Object References (IDOR)

1.1. Give me my amount!!

Reference	Risk Rating
Give me my amount!!	Low
Tools Used	
Google chrome	
Vulnerability Description	
An insecure direct object reference (IDOR) is an access control vulnerability where invalidated user input can be used for unauthorized access to resources or operations. It occurs when an attacker gains direct access by using user supplied input to an object that has no authorization to access. Attackers can bypass the authorization mechanism to access resources in the system directly by exploiting this vulnerability. Every resource instance can be called as an object and often represented with an ID. And if these IDs are easy enough to guess or an object can be used by an attacker to bypass access check somehow, we can talk about an IDOR at this point.	
How It Was Discovered	
I found this vulnerability by manipulating value of id parameter in the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1869	
Consequences of not Fixing the Issue	
This vulnerability violates the privacy of a User.	
Suggested Countermeasures	
1. Developers should avoid displaying private object references such as keys or file names. 2. Validation of parameters should be properly implemented. 3. Verification of all the referenced objects should be checked. 4. Tokens should be generated in such a way that they can only be mapped to the user and is not public.	
References	
1. https://portswigger.net/web-security/access-control/idor 2. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html 3. https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/	
Payload Used	
Appending “?id=1869” at last in the URL	

Proof of Concept

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

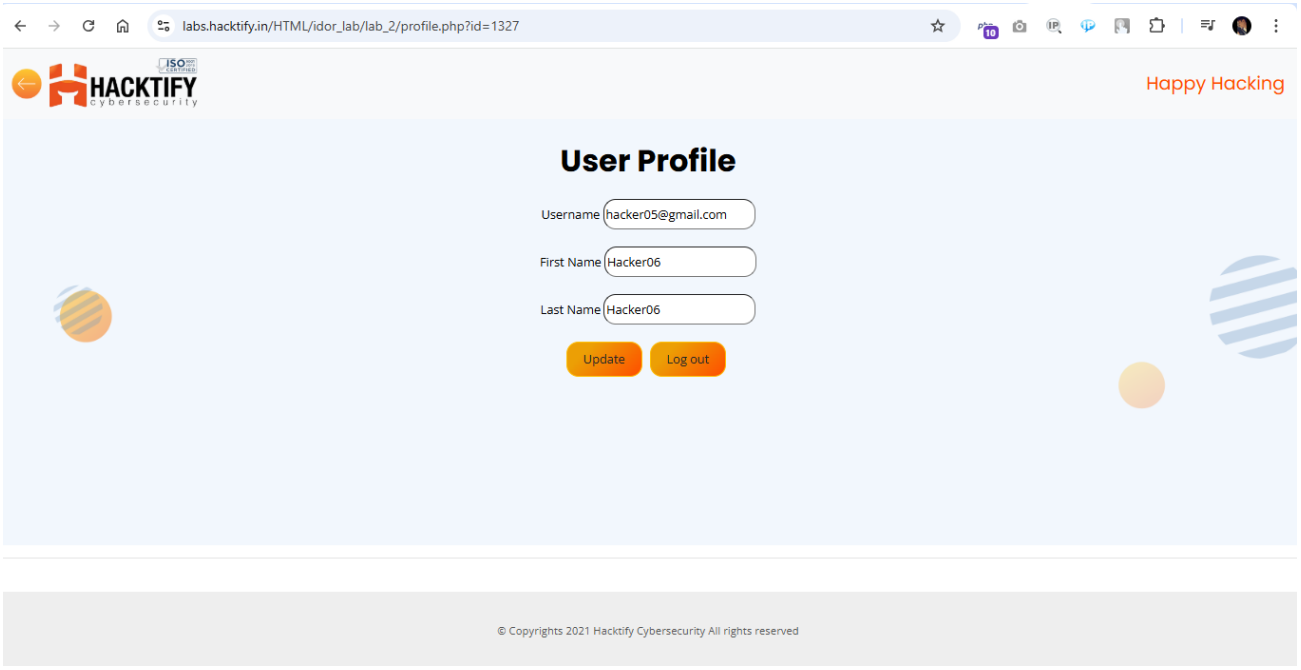


1.2. Stop polluting my params!

Reference	Risk Rating
Stop polluting my params!	Medium
Tools Used	
Google chrome	
Vulnerability Description	
An insecure direct object reference (IDOR) is an access control vulnerability where invalidated user input can be used for unauthorized access to resources or operations. It occurs when an attacker gains direct access by using user supplied input to an object that has no authorization to access. Attackers can bypass the authorization mechanism to access resources in the system directly by exploiting this vulnerability. Every resource instance can be called as an object and often represented with an ID. And if these IDs are easy enough to guess or an object can be used by an attacker to bypass access check somehow, we can talk about an IDOR at this point.	
How It Was Discovered	
I found this vulnerability by manipulating value of id parameter in the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1327	
Consequences of not Fixing the Issue	
This vulnerability violates the privacy of a User.	
Suggested Countermeasures	
1. Developers should avoid displaying private object references such as keys or file names. 2. Validation of parameters should be properly implemented. 3. Verification of all the referenced objects should be checked. 4. Tokens should be generated in such a way that they can only be mapped to the user and is not public.	
References	
1. https://portswigger.net/web-security/access-control/idor 2. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html 3. https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/	
Payload Used	
Appending “?id=1327” at last in the URL	

Proof of Concept

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

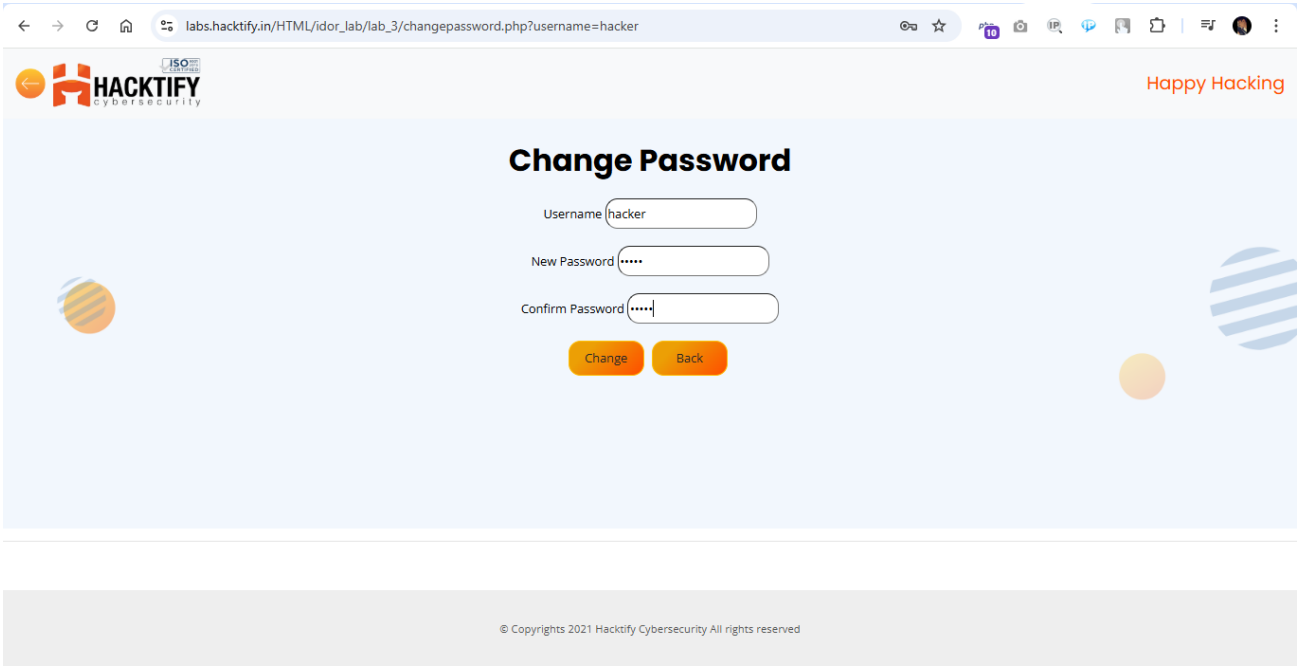


1.3. Someone changed my Password!

Reference	Risk Rating
Someone changed my Password!	High
Tools Used	
Google chrome	
Vulnerability Description	
An insecure direct object reference (IDOR) is an access control vulnerability where invalidated user input can be used for unauthorized access to resources or operations. It occurs when an attacker gains direct access by using user supplied input to an object that has no authorization to access. Attackers can bypass the authorization mechanism to access resources in the system directly by exploiting this vulnerability. Every resource instance can be called as an object and often represented with an ID. And if these IDs are easy enough to guess or an object can be used by an attacker to bypass access check somehow, we can talk about an IDOR at this point.	
How It Was Discovered	
I found this vulnerability by manipulating the value of username parameter in the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=hacker	
Consequences of not Fixing the Issue	
This vulnerability violates the privacy of a User.	
Suggested Countermeasures	
1. Developers should avoid displaying private object references such as keys or file names. 2. Validation of parameters should be properly implemented. 3. Verification of all the referenced objects should be checked. 4. Tokens should be generated in such a way that they can only be mapped to the user and is not public.	
References	
1. https://portswigger.net/web-security/access-control/idor 2. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html 3. https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/	
Payload Used	
Appending “?username=hacker” at last in the URL	

Proof of Concept

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

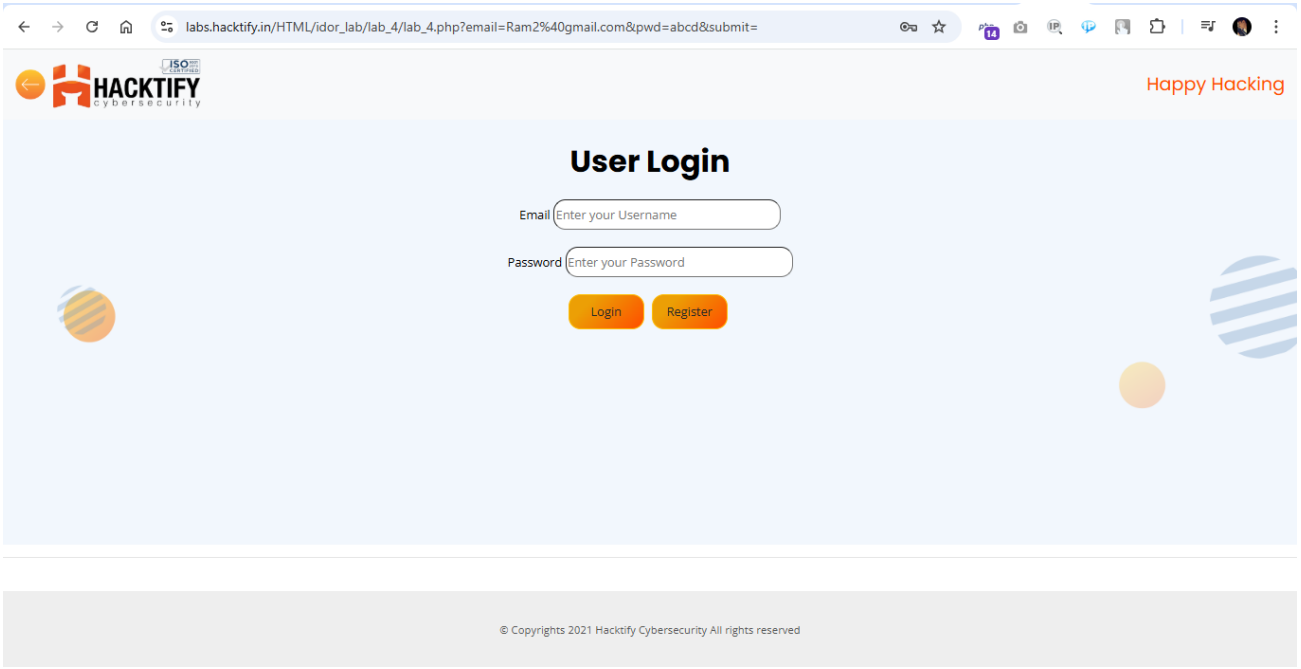


1.4. Change your methods!

Reference	Risk Rating
Change your methods!	Medium
Tools Used	
Google chrome	
Vulnerability Description	
An insecure direct object reference (IDOR) is an access control vulnerability where invalidated user input can be used for unauthorized access to resources or operations. It occurs when an attacker gains direct access by using user supplied input to an object that has no authorization to access. Attackers can bypass the authorization mechanism to access resources in the system directly by exploiting this vulnerability. Every resource instance can be called as an object and often represented with an ID. And if these IDs are easy enough to guess or an object can be used by an attacker to bypass access check somehow, we can talk about an IDOR at this point.	
How It Was Discovered	
I found this vulnerability by manipulating the value of id parameter in the URL followed by updating random details in the profile section of different user.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_4/lab_4.php?email=Ram2%40gmail.com&pwd=abcd&submit=	
Consequences of not Fixing the Issue	
This vulnerability violates the privacy of a User.	
Suggested Countermeasures	
1. Developers should avoid displaying private object references such as keys or file names. 2. Validation of parameters should be properly implemented. 3. Verification of all the referenced objects should be checked. 4. Tokens should be generated in such a way that they can only be mapped to the user and is not public.	
References	
1. https://portswigger.net/web-security/access-control/idor 2. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html 3. https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/	
Payload Used	
Appending “?id=2157” at last in the URL followed by updating random details in the profile section of different user.	

Proof of Concept

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



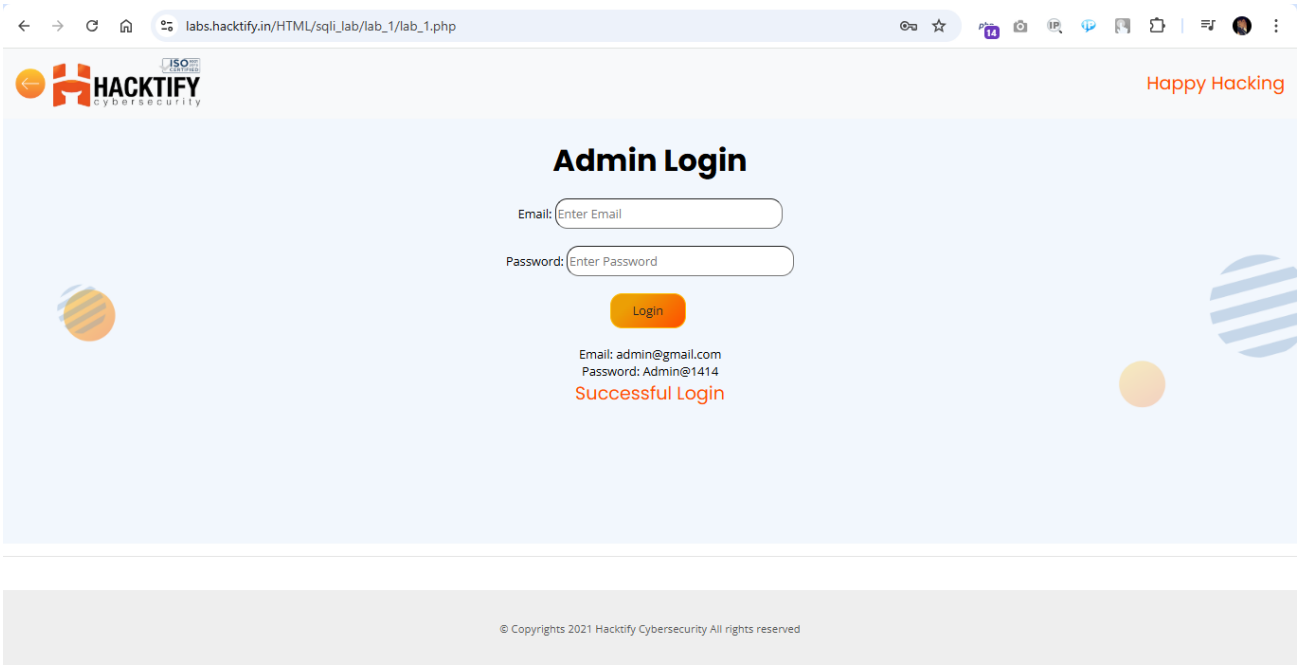
2. SQL Injection

2.1. Strings and Errors Part1!

Reference	Risk Rating
Strings and Errors Part1!	Low
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into user input fields, i.e., Email and Password.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
In both the user input fields insert below payload and click login 1" OR "1"="1	

Proof of Concept

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

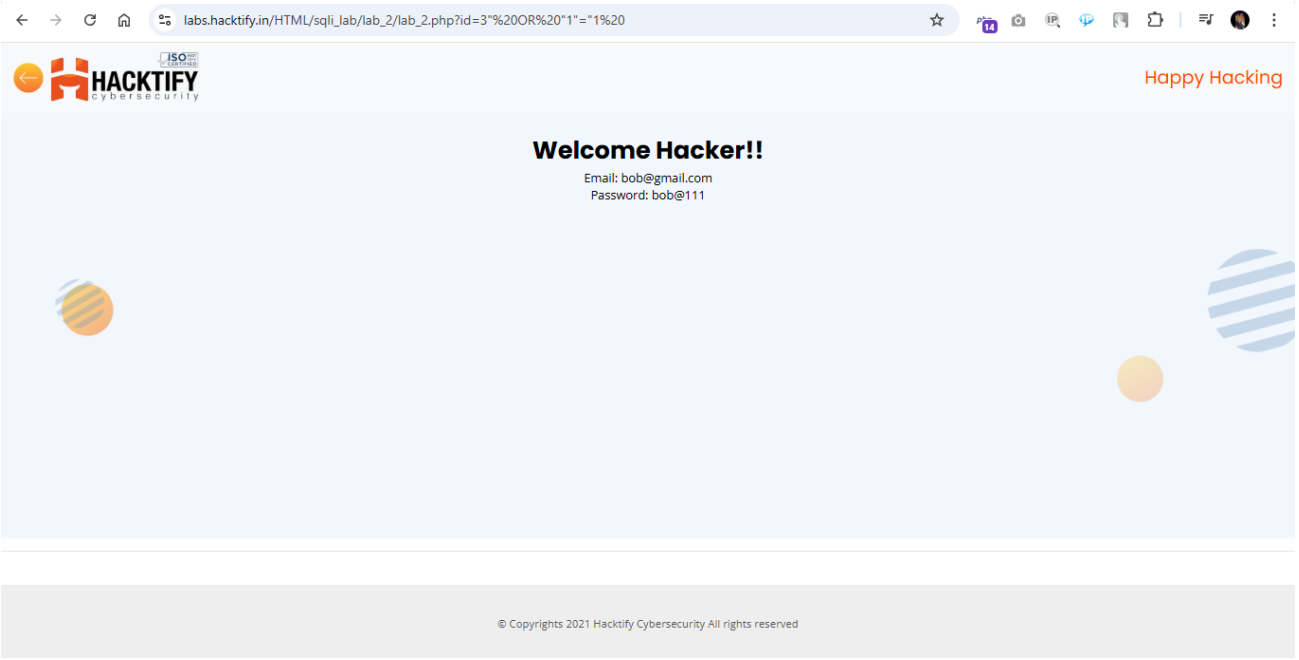


2.2. Strings and Errors Part2!

Reference	Risk Rating
Strings and Errors Part2!	Low
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=3%22%20OR%20%221%22=%221%20	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Appending below payload at last in the URL ?id=3"%20OR%20"1"="1%20	

Proof of Concept

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

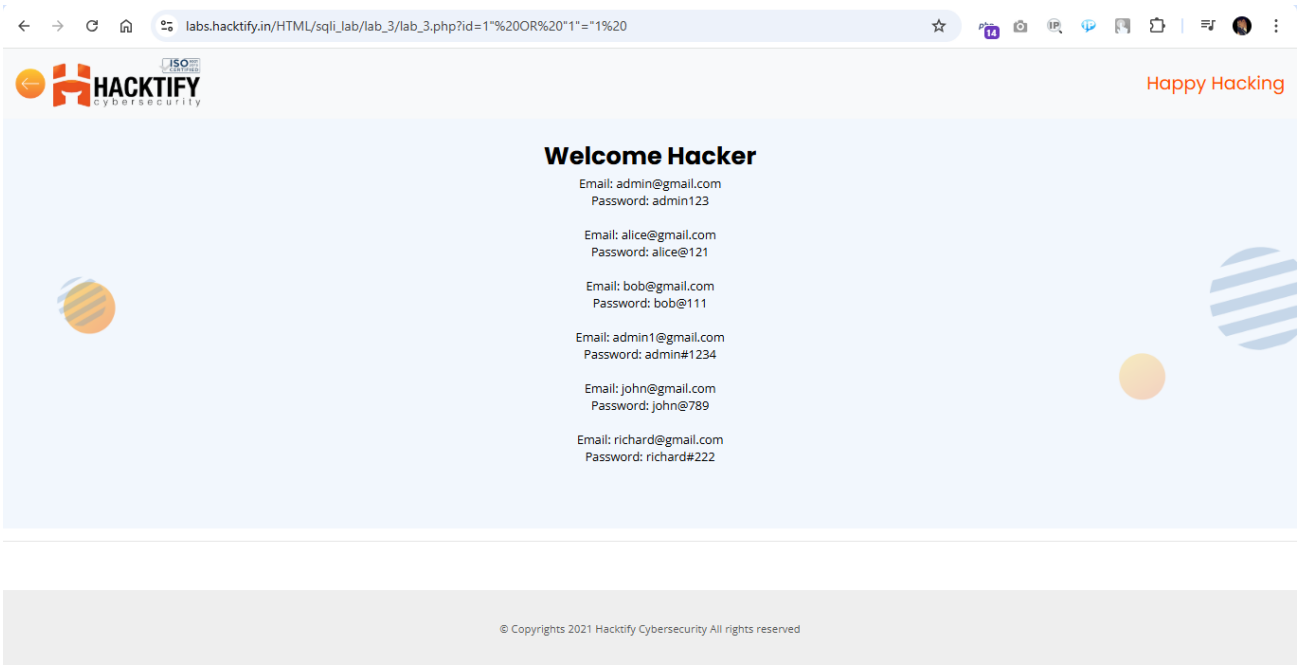


2.3. Strings and Errors Part3!

Reference	Risk Rating
Strings and Errors Part3!	Low
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1%22%20OR%20%221%22=%221%20	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Appending below payload at last in the URL ?id=3" OR "1"="1	

Proof of Concept

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

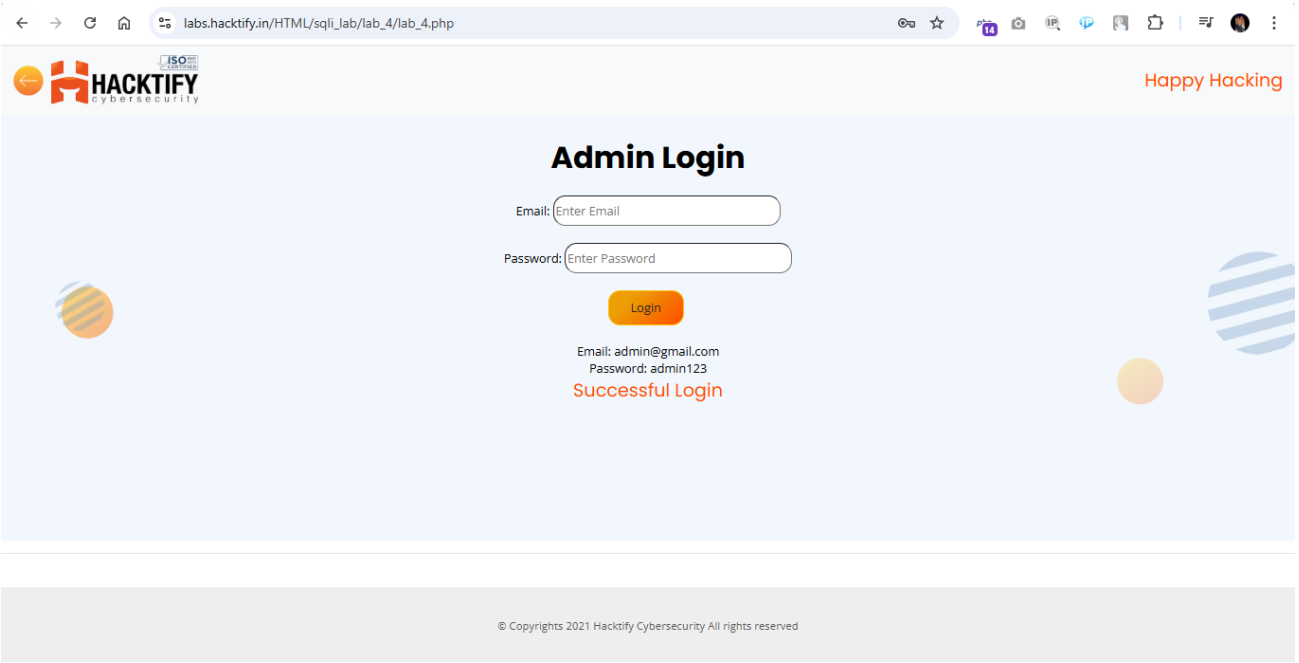


2.4. Let's Trick 'em!

Reference	Risk Rating
Let's Trick 'em!	Medium
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into user input fields, i.e., Email and Password.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
In both the user input fields insert below payload and click login 1' || '1'= '1	

Proof of Concept

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

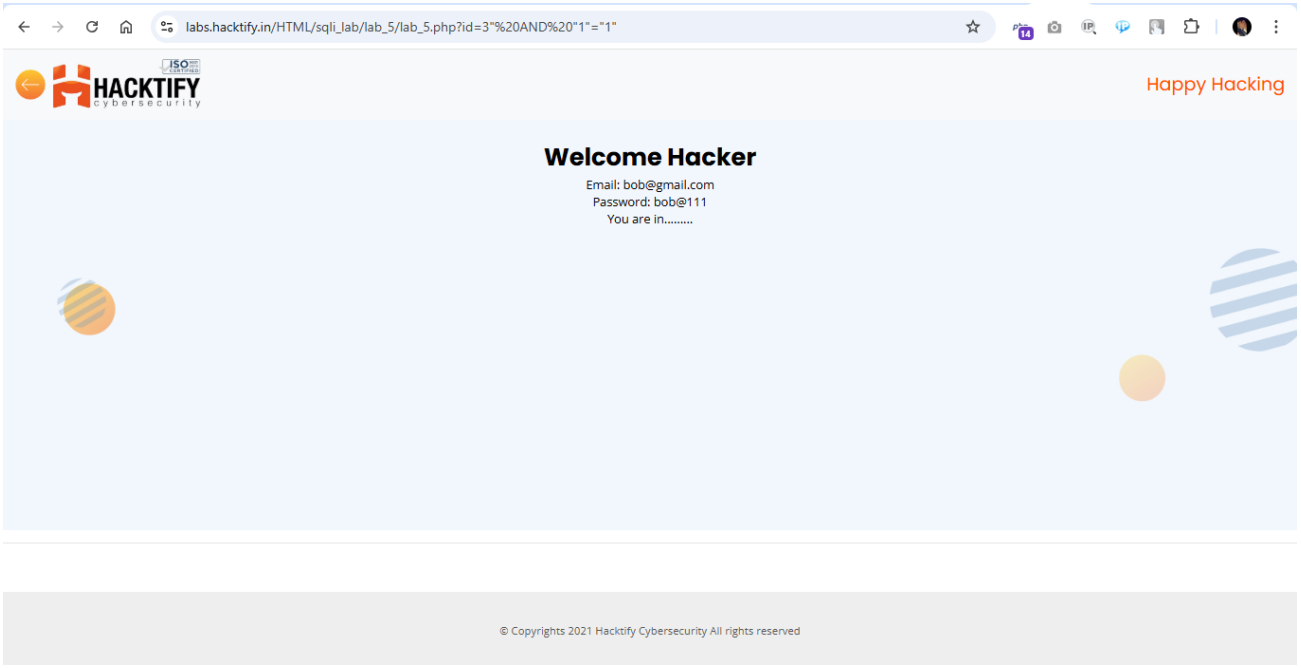


2.5. Booleans and Blind!

Reference	Risk Rating
Booleans and Blind!	High
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=3%22%20AND%20%221%22=%221%22	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Appending below payload at last in the URL ?id=3" OR "1"="1	

Proof of Concept

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

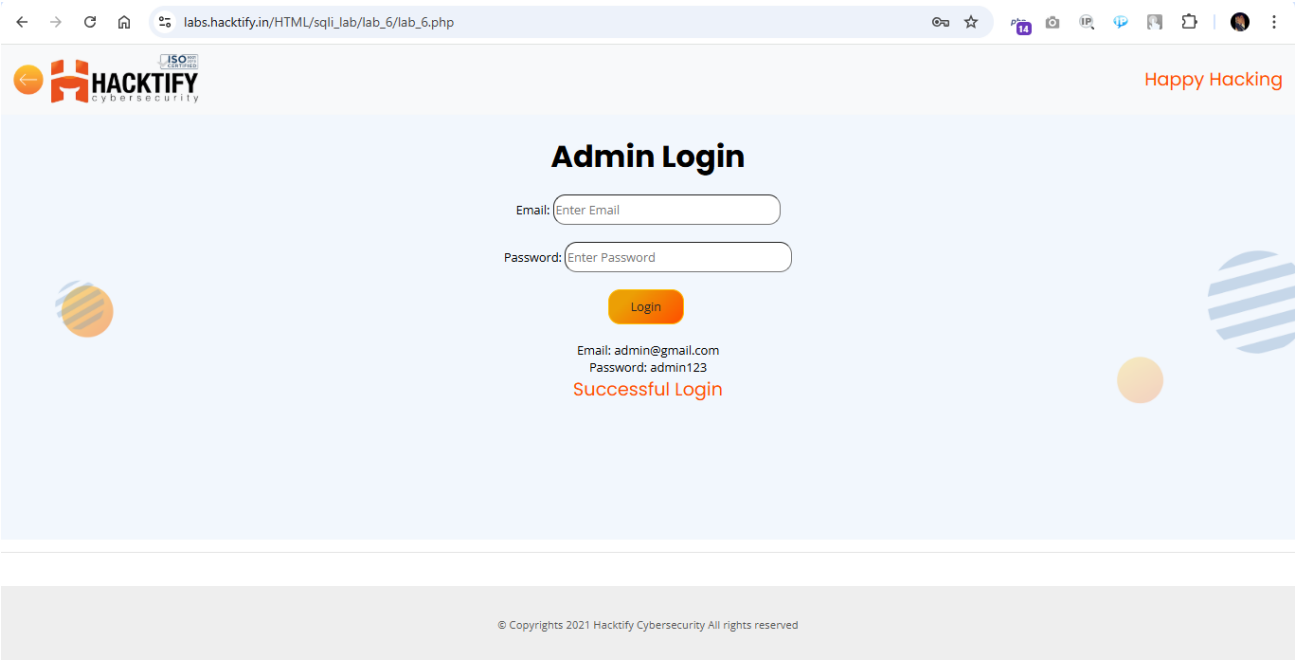


2.6. Error based: Tricked

Reference	Risk Rating
Error based: Tricked	Medium
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into user input fields, i.e., Email and Password.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_6/lab_6.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
In both the user input fields insert below payload and click login ") or ("1")=("1	

Proof of Concept

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

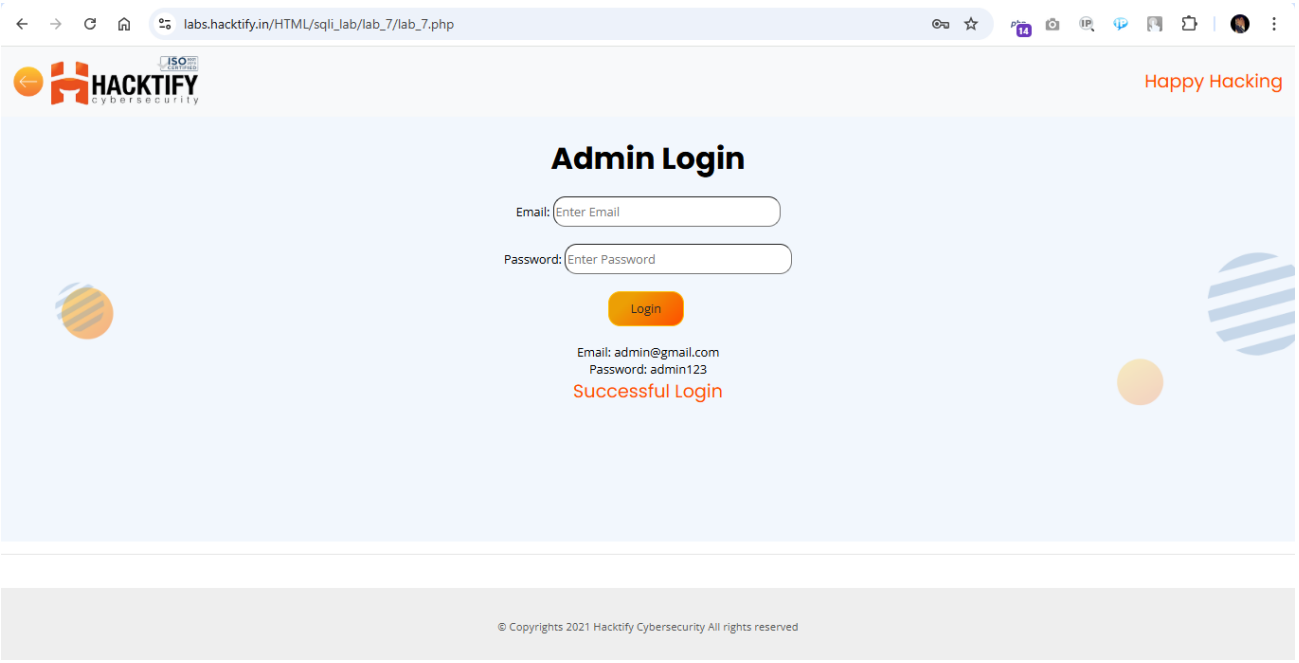


2.7. Errors and Post!

Reference	Risk Rating
Errors and Post!	Low
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into user input fields, i.e., Email and Password.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
In both the user input fields insert below payload and click login ' or '1'='1	

Proof of Concept

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

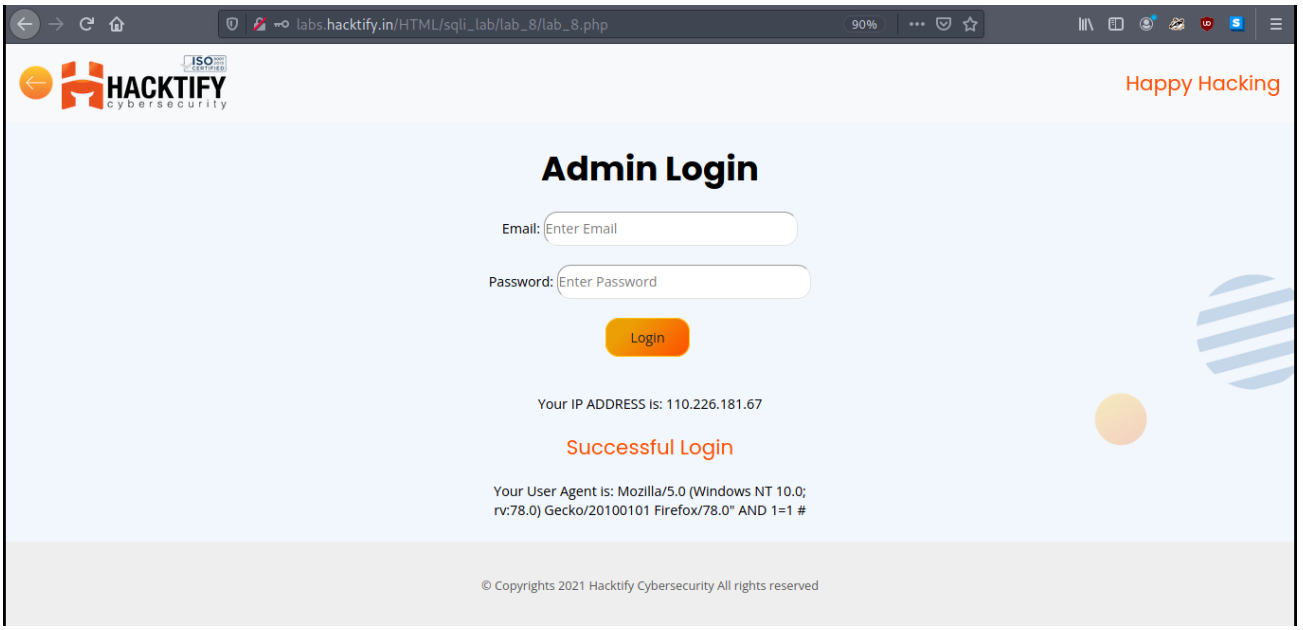


2.8. User Agents lead us!

Reference	Risk Rating
User Agents lead us!	High
Tools Used	
Google chrome and Burp suite	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into user agent field by intercepting login request into Burp Suite.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Login with Admin credentials obtained in previous labs, Intercept the traffic with Burpsuite, In the User Agent field insert the below payload " AND 1=1 #	

Proof of Concept

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

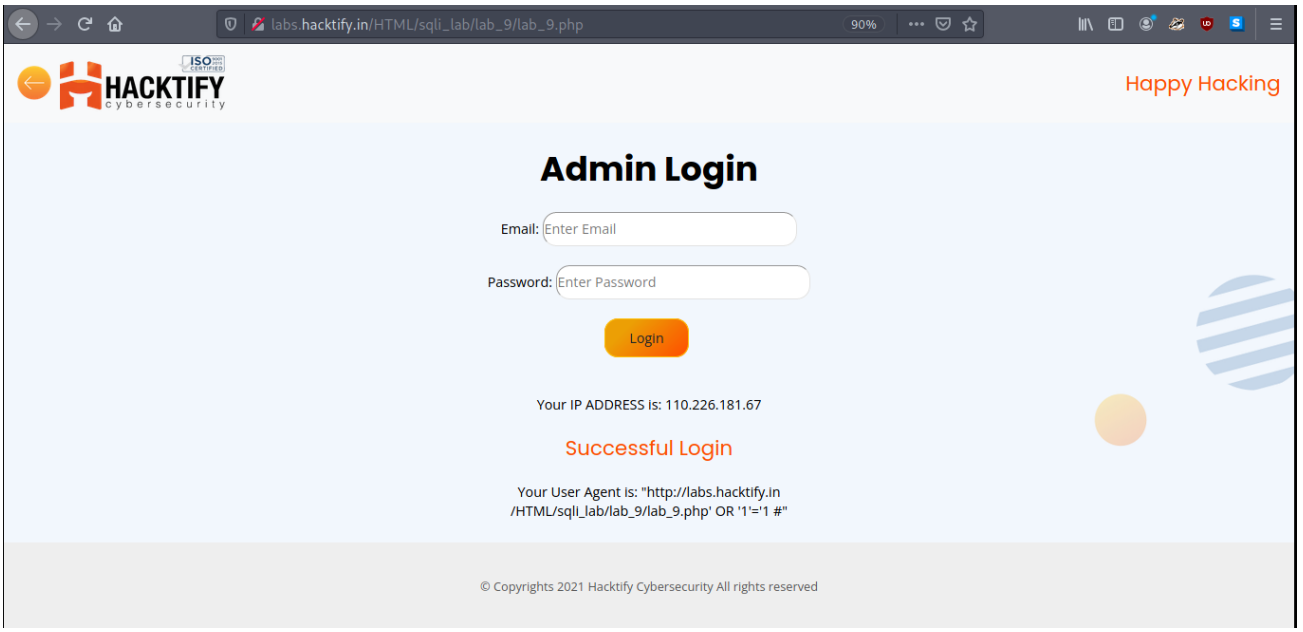


2.9. Referer lead us!

Reference	Risk Rating
Referer lead us!	Medium
Tools Used	
Google chrome and Burp suite	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into referrer field by intercepting login request into Burp Suite.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Login with Admin credentials obtained in previous labs, Intercept the traffic with Burpsuite, In the Referer field insert the below payload ' OR '1'='1 #	

Proof of Concept

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

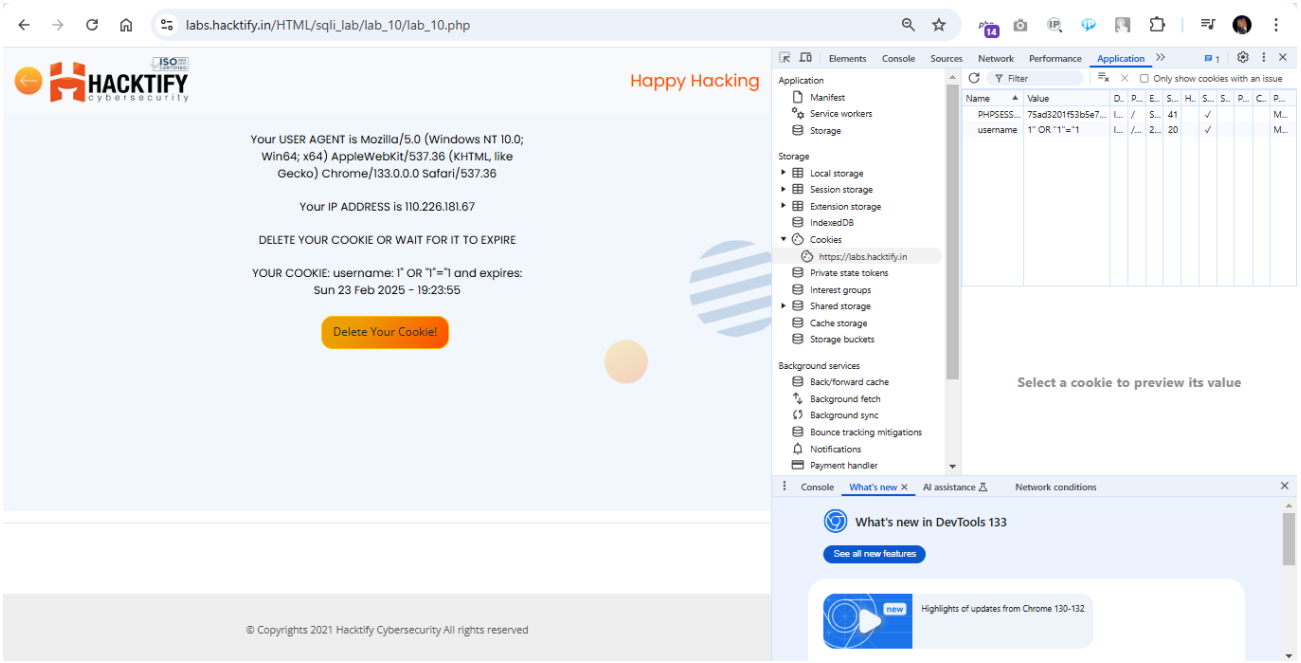


2.10. Oh Cookies!

Reference	Risk Rating
Oh Cookies!	High
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into cookie value by inspecting the cookie settings after logging into the sign in page.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Login with Admin credentials, click on Inspect and go to the Cookies section and insert the below payload into Username field 1" OR "1"="1	

Proof of Concept

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

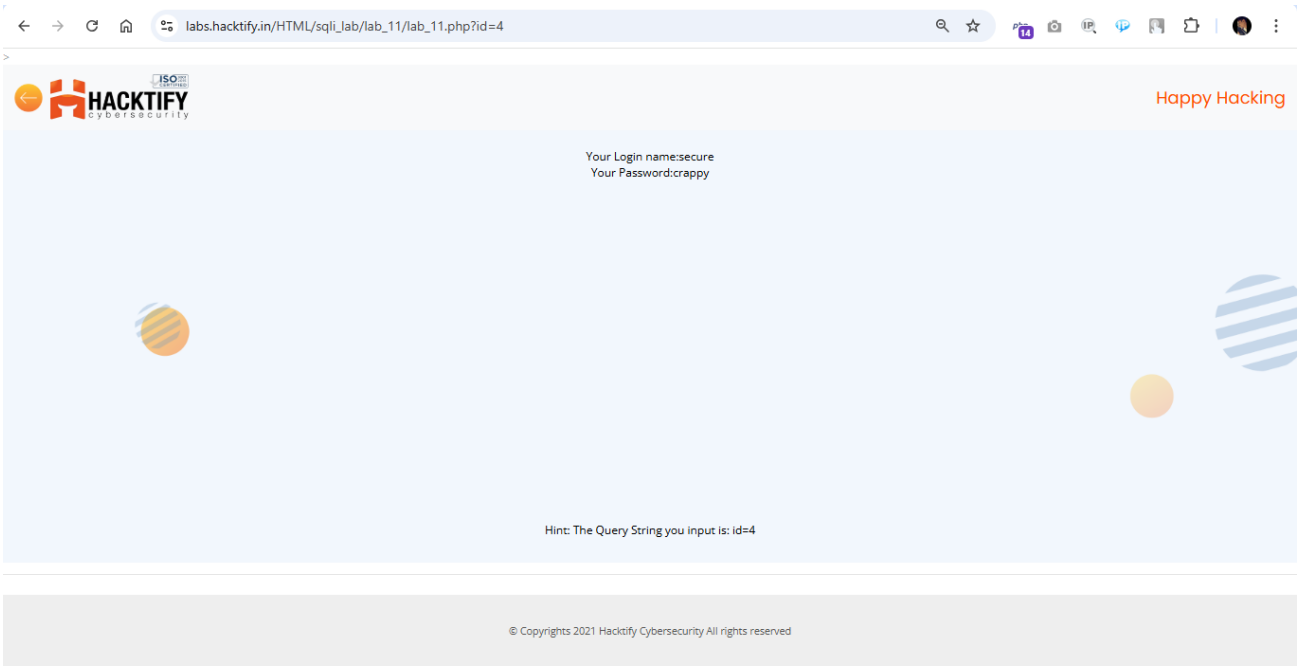


2.11. WAF's Are injected!

Reference	Risk Rating
WAF's Are injected!	High
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=4	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Appending below payload at last in the URL ?id=4	

Proof of Concept

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



2.12. WAF's Are injected Part2!

Reference	Risk Rating
WAF's Are injected Part2!	Medium
Tools Used	
Google chrome	
Vulnerability Description	
SQL Injection allows an attacker to view data that attackers are normally not able to retrieve. Data can be information about users, their credentials, personal details etc. It is the process of inserting or injecting SQL queries through input fields to an application to give the hacker the data he wants! Attackers can modify or delete this data causing persistent changes to the application's content or behavior. SQL Injection can also be escalated to compromise the underlying server (or) other back-end infrastructure, or perform a denial-of-service attack	
How It Was Discovered	
I found this vulnerability by injecting malicious SQL injection code into the URL.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=8	
Consequences of not Fixing the Issue	
Stealing credentials, access to the database, altering or modifying data, access to the network.	
Suggested Countermeasures	
1. Implement Input Validation and Sanitization 2. Using Escaping for User Input 3. Utilize Parameterized Statements (Prepared Statements) 4. Incorporate Stored Procedures 5. Conduct Continuous Scanning and Penetration Testing 6. Adopt the Least Privilege Principle 7. Deploy Web Application Firewalls (WAF)	
References	
1. https://portswigger.net/web-security/sql-injection/cheat-sheet 2. https://portswigger.net/web-security/sql-injection 3. https://owasp.org/www-community/attacks/SQL_Injection	
Payload Used	
Appending below payload at last in the URL ?id=8	

Proof of Concept

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