**Penetration Testing Report**

**Full Name: Saurav Pundir  
Program: HCS - Penetration Testing Internship Week-2**

**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** | **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: 16 Sub-labs**

**Week\_1\_Hacktify\_HCS\_Saurav**

|  |  |  |
| --- | --- | --- |
| **High** | **Medium** | **Low** |
| **5** | **6** | **5** |

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

**Medium - Number of Sub-labs with medium difficulty level**

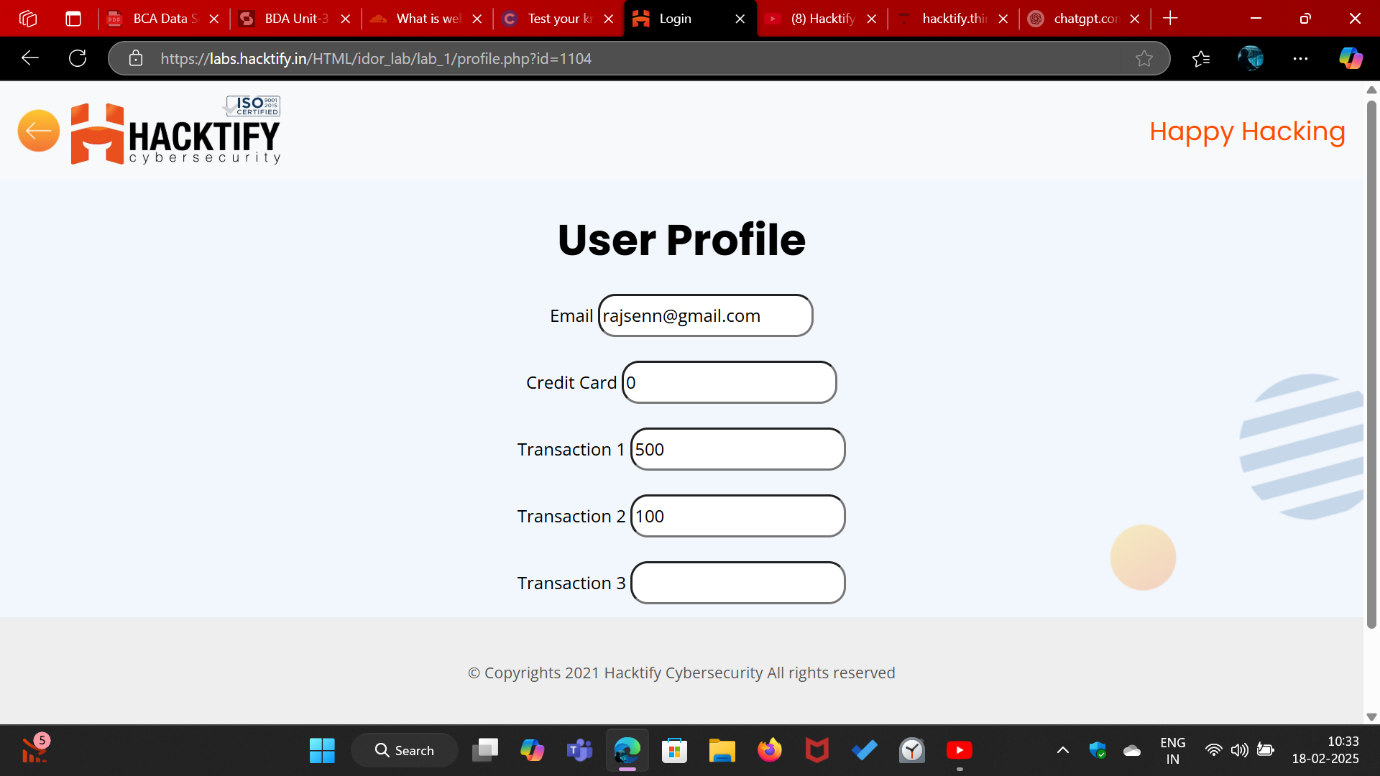
**Low - Number of Sub-labs with Easy difficulty level**

# 1. Insecure Direct Object References

# 1.1. Give me my amount!!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Give me my amount!! | **Low** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| IDOR allows unauthorized users to access or modify other users’ sensitive data by manipulating the id parameter. | |
| **How It Was Discovered** | |
| By changing the id value in the URL (e.g., ?id=1104 → ?id=1105), an attacker can view or edit another user's profile. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/idor_lab/lab_1/lab_1.php> | |
| **Consequences of not Fixing the Issue** | |
| Exposure of sensitive user data such as emails, credit card details, and transactions, unauthorized modifications to user profiles, potential financial or reputational damage. | |
| **Suggested Countermeasures** | |
| Implement proper authentication and authorization checks, use session-based user validation instead of direct ID references, apply access control mechanisms such as RBAC/ABAC, log and monitor unauthorized access attempts | |
| **References** | |
| <https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html>  [Insecure direct object references (IDOR) | Web Security Academy](https://portswigger.net/web-security/access-control/idor) | |

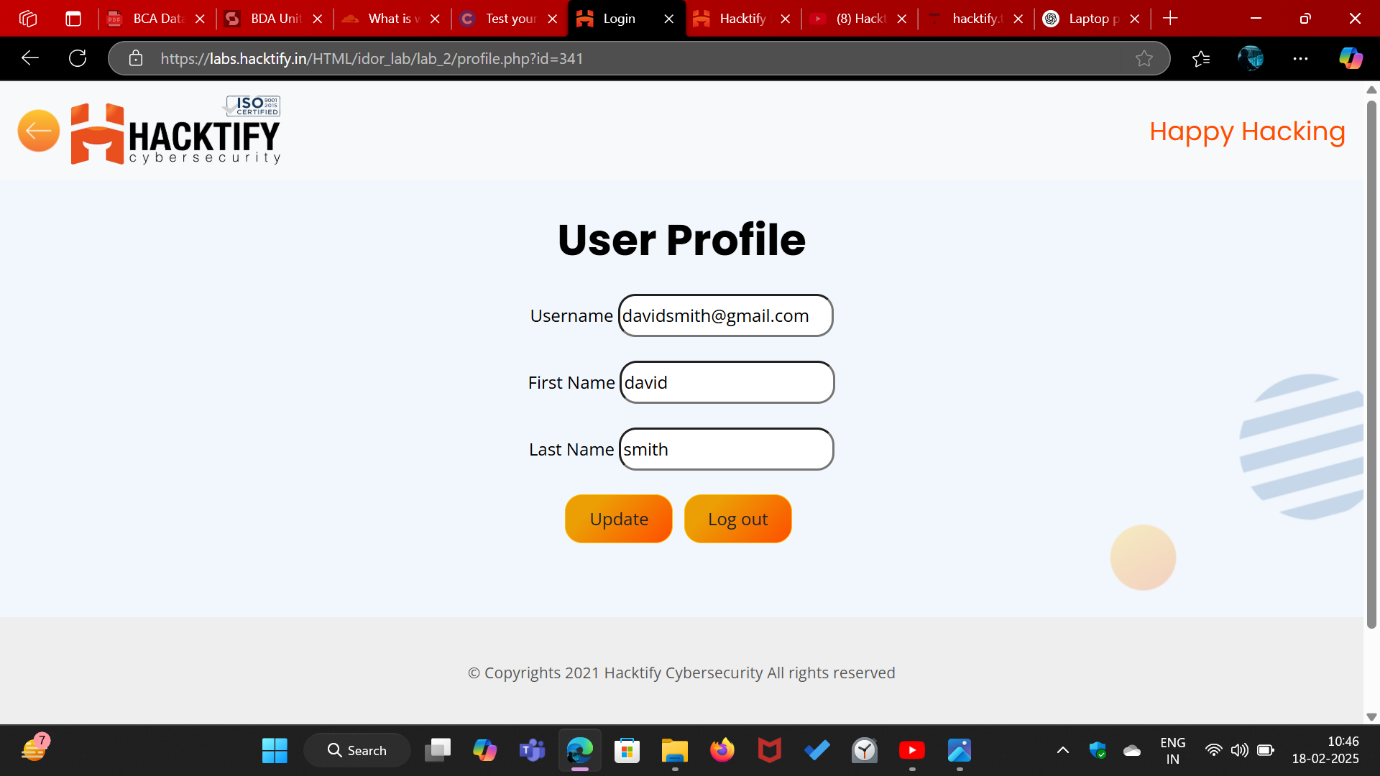
# Proof of Concept



# 1.2. Stop polluting my params!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Stop polluting my params! | **Medium** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| IDOR allows an attacker to access or modify another user’s profile by manipulating the id parameter in the URL. | |
| **How It Was Discovered** | |
| Changing the id value in the URL (e.g., ?id=341 → ?id=342) enables unauthorized access to another user's profile details. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/idor_lab/lab_2/lab_2.php> | |
| **Consequences of not Fixing the Issue** | |
| Exposure of personal user information, unauthorized modification of profile details, potential identity theft or account takeover. | |
| **Suggested Countermeasures** | |
| Implement proper authentication and authorization checks, restrict direct object references, use session-based validation, enforce access controls to prevent unauthorized profile edits. | |
| **References** | |
| <https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html>  [Insecure direct object references (IDOR) | Web Security Academy](https://portswigger.net/web-security/access-control/idor) | |

# Proof of Concept

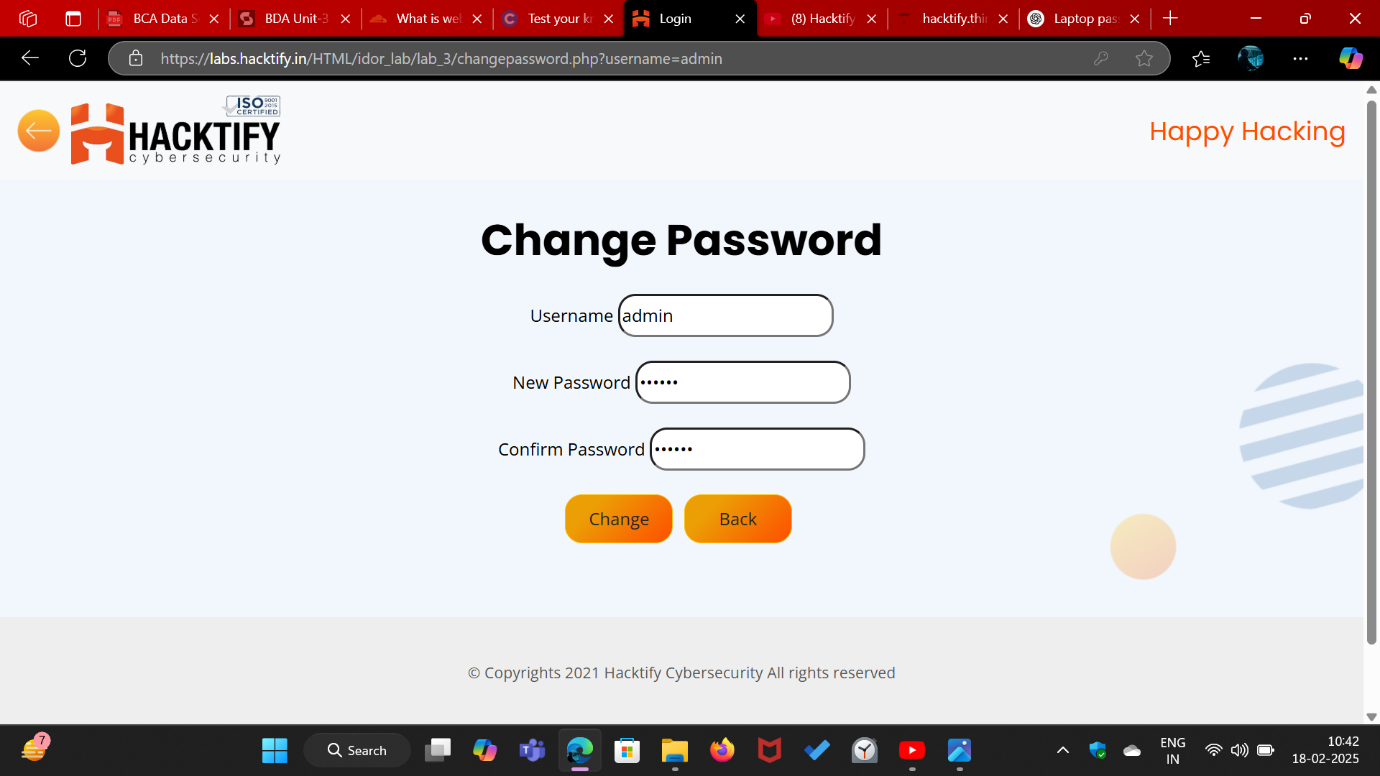


# 

# 1.3. Someone changed my Password 🙀!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Someone changed my Password! | **High** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| IDOR allows an attacker to change another user's password by modifying the username parameter in the URL | |
| **How It Was Discovered** | |
| Changing the username value in the URL (e.g., ?username=admin → ?username=victim) enables unauthorized password resets for other users. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/idor_lab/lab_3/lab_3.php> | |
| **Consequences of not Fixing the Issue** | |
| Full account takeover, unauthorized access to sensitive data, potential privilege escalation if an admin account is compromised. | |
| **Suggested Countermeasures** | |
| Enforce authentication checks before allowing password changes, restrict direct object references, implement session-based user validation, log and monitor suspicious password reset attempts. | |
| **References** | |
| <https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html>  [Insecure direct object references (IDOR) | Web Security Academy](https://portswigger.net/web-security/access-control/idor) | |

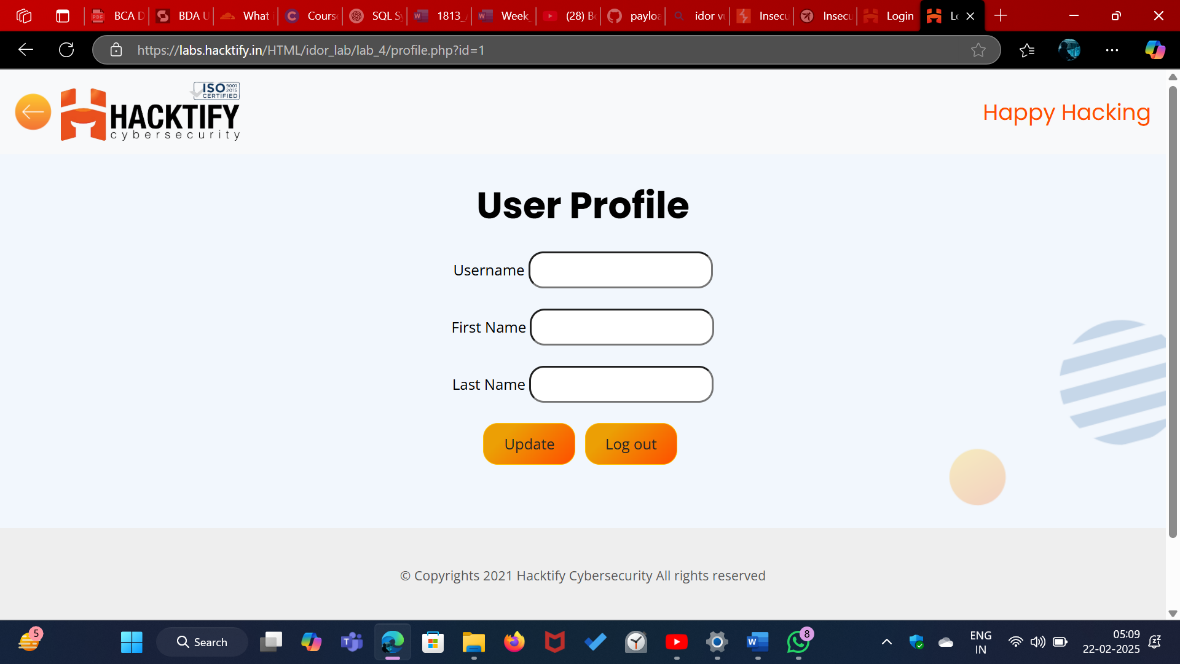
# Proof of Concept

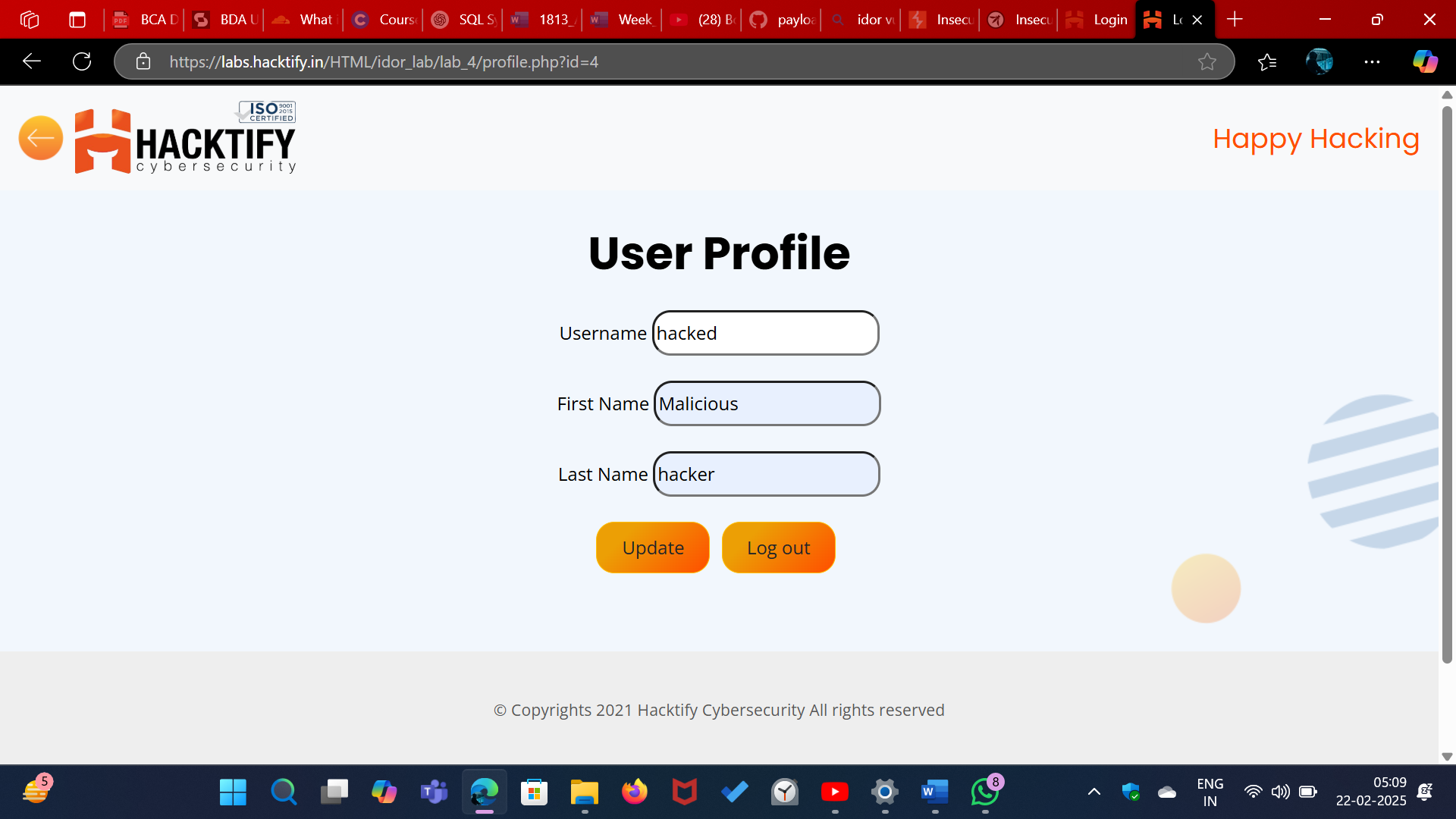


# 1.4. Change your methods!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Change your methods! | **Medium** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| IDOR allows an attacker to modify another user's profile details by changing the id parameter in the URL. | |
| **How It Was Discovered** | |
| Altering the id value (e.g., ?id=1 → ?id=4) allowed unauthorized access and modification of another user's profile information. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/idor_lab/lab_4/lab_4.php> | |
| **Consequences of not Fixing the Issue** | |
| Unauthorized changes to user data, potential identity fraud, reputational damage, and loss of user trust. | |
| **Suggested Countermeasures** | |
| Implement access control checks to ensure users can only modify their own data, use session-based authentication, restrict direct object references, log and monitor unauthorized access attempts. | |
| **References** | |
| <https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html>  [Insecure direct object references (IDOR) | Web Security Academy](https://portswigger.net/web-security/access-control/idor) | |

# Proof of Concept





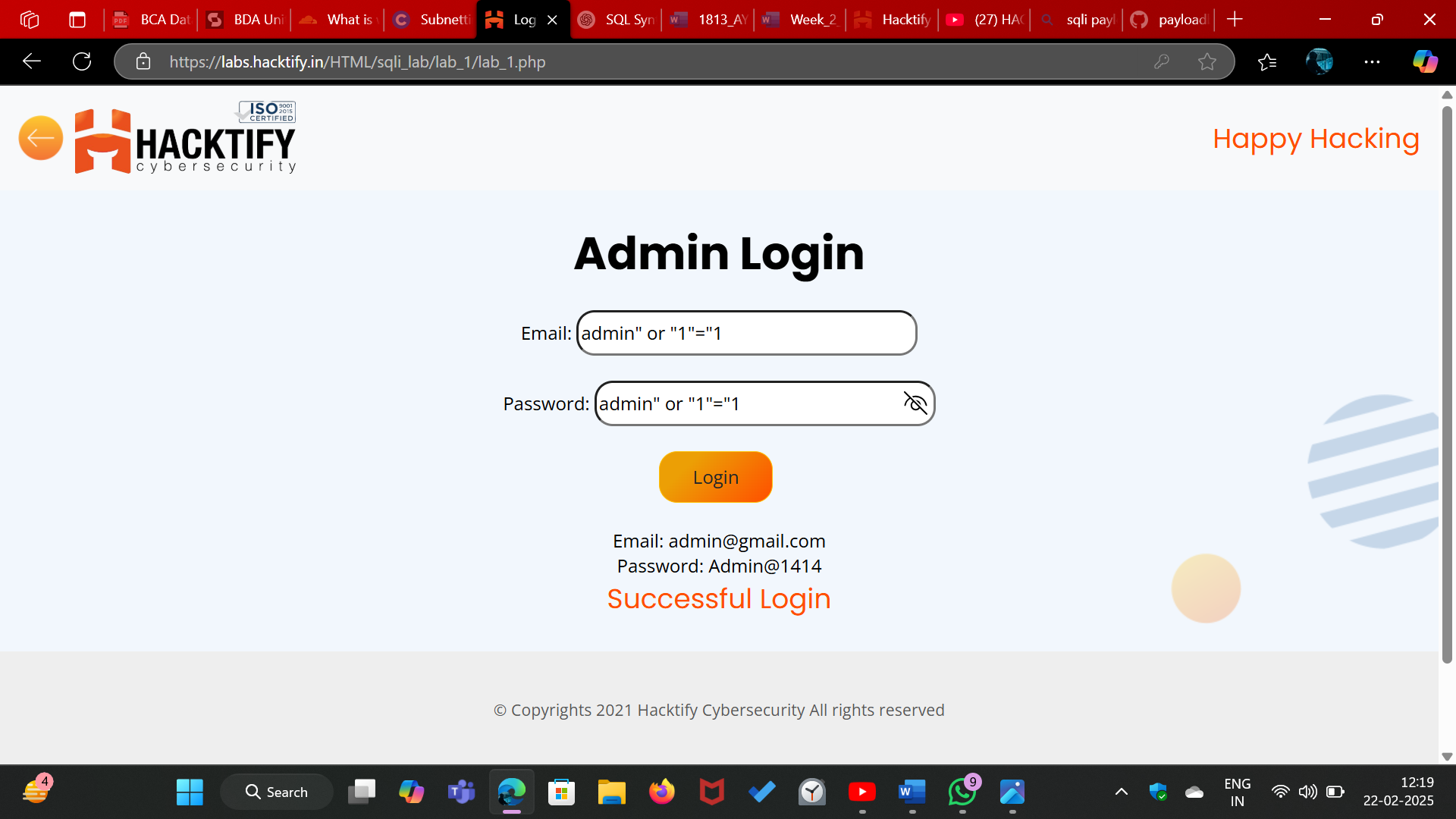
# 2. SQL Injection

# 2.1. Strings & Errors Part 1!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Strings & Errors Part 1! | **Low** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection allows an attacker to bypass authentication by injecting malicious SQL queries into input fields. | |
| **How It Was Discovered** | |
| Using ' OR '1'='1' -- in the email and password fields granted unauthorized admin access, indicating improper input validation. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php> | |
| **Consequences of not Fixing the Issue** | |
| Unauthorized access to admin accounts, data breaches, full database compromise, potential system takeover. | |
| **Suggested Countermeasures** | |
| Use prepared statements and parameterized queries, implement input validation and sanitization, restrict error messages to prevent SQL error-based attacks, enforce least privilege access for database users. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

# 

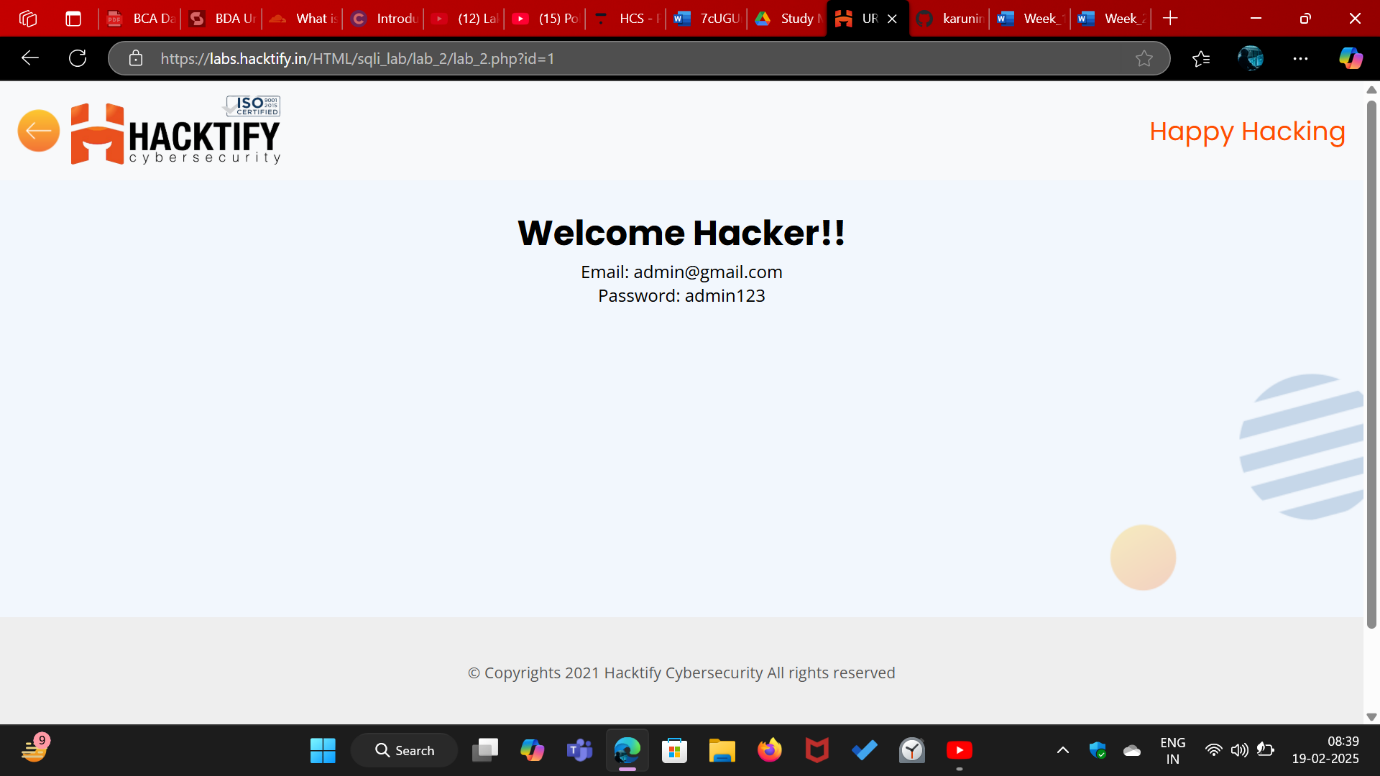
# Proof of Concept



# 2.2. Strings & Errors Part 2!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Strings & Errors Part 2! | **Low** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection allows an attacker to retrieve sensitive database information by manipulating the id parameter in the URL. | |
| **How It Was Discovered** | |
| Modifying the id parameter (?id=1) exposed admin credentials, indicating a lack of proper input validation and database protection. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php> | |
| **Consequences of not Fixing the Issue** | |
| Exposure of user credentials, unauthorized access to admin accounts, full database compromise, increased risk of data leaks and system breaches. | |
| **Suggested Countermeasures** | |
| Use prepared statements and parameterized queries, implement strict input validation and sanitization, restrict database error messages, enforce least privilege access to sensitive data. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

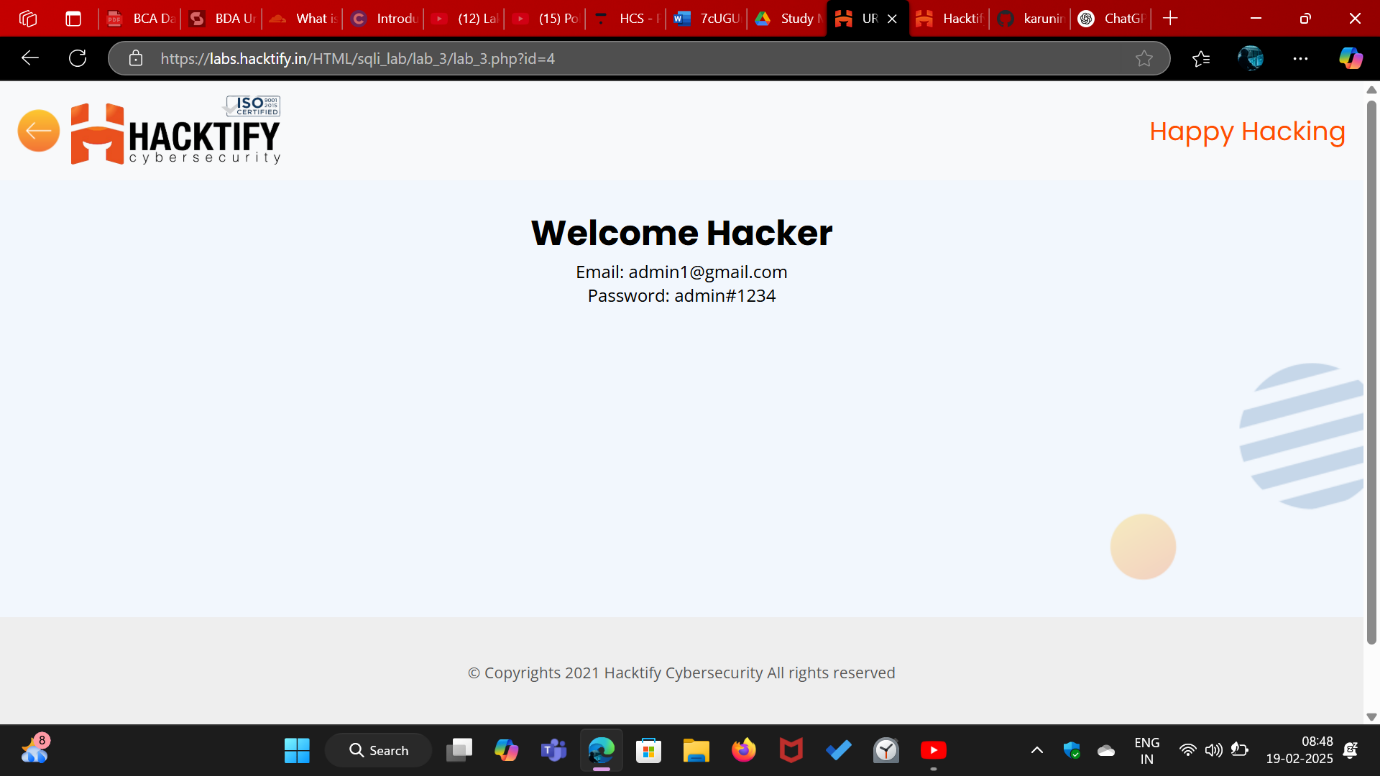
# Proof of Concept



# 2.3. Strings & Errors Part 3!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Strings & Errors Part 3! | **Low** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection allows an attacker to extract admin credentials by manipulating the id parameter in the URL. | |
| **How It Was Discovered** | |
| Changing the id parameter (?id=4) revealed stored admin credentials, confirming the database query is vulnerable to injection. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php> | |
| **Consequences of not Fixing the Issue** | |
| Exposure of sensitive credentials, unauthorized admin access, full database compromise, risk of further exploitation like privilege escalation or data theft. | |
| **Suggested Countermeasures** | |
| Use prepared statements and parameterized queries, validate and sanitize user inputs, disable detailed error messages, implement least privilege access to sensitive database records. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

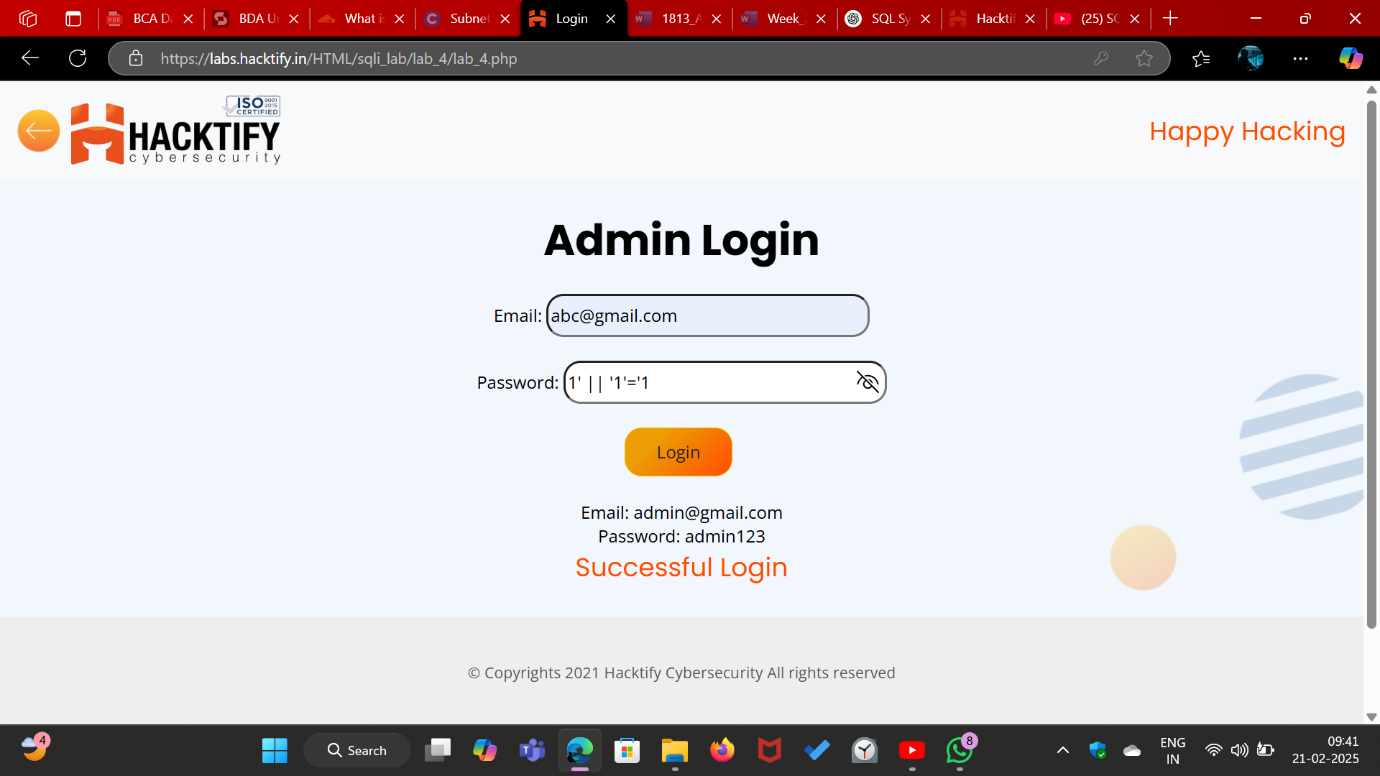
# Proof of Concept



# 2.4. Let’s Trick ‘em!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Let’s Trick ‘em! | **Medium** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection in the password field allows authentication bypass by injecting ' OR '1'='1. | |
| **How It Was Discovered** | |
| Entering a SQL payload in the password field resulted in a successful login without valid credentials. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_4/lab_4.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can gain unauthorized admin access, leading to full control over the system, data breaches, and further exploitation. | |
| **Suggested Countermeasures** | |
| Implement prepared statements, parameterized queries, and input validation, disable detailed SQL errors, and enforce strong authentication mechanisms like multi-factor authentication (MFA). | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

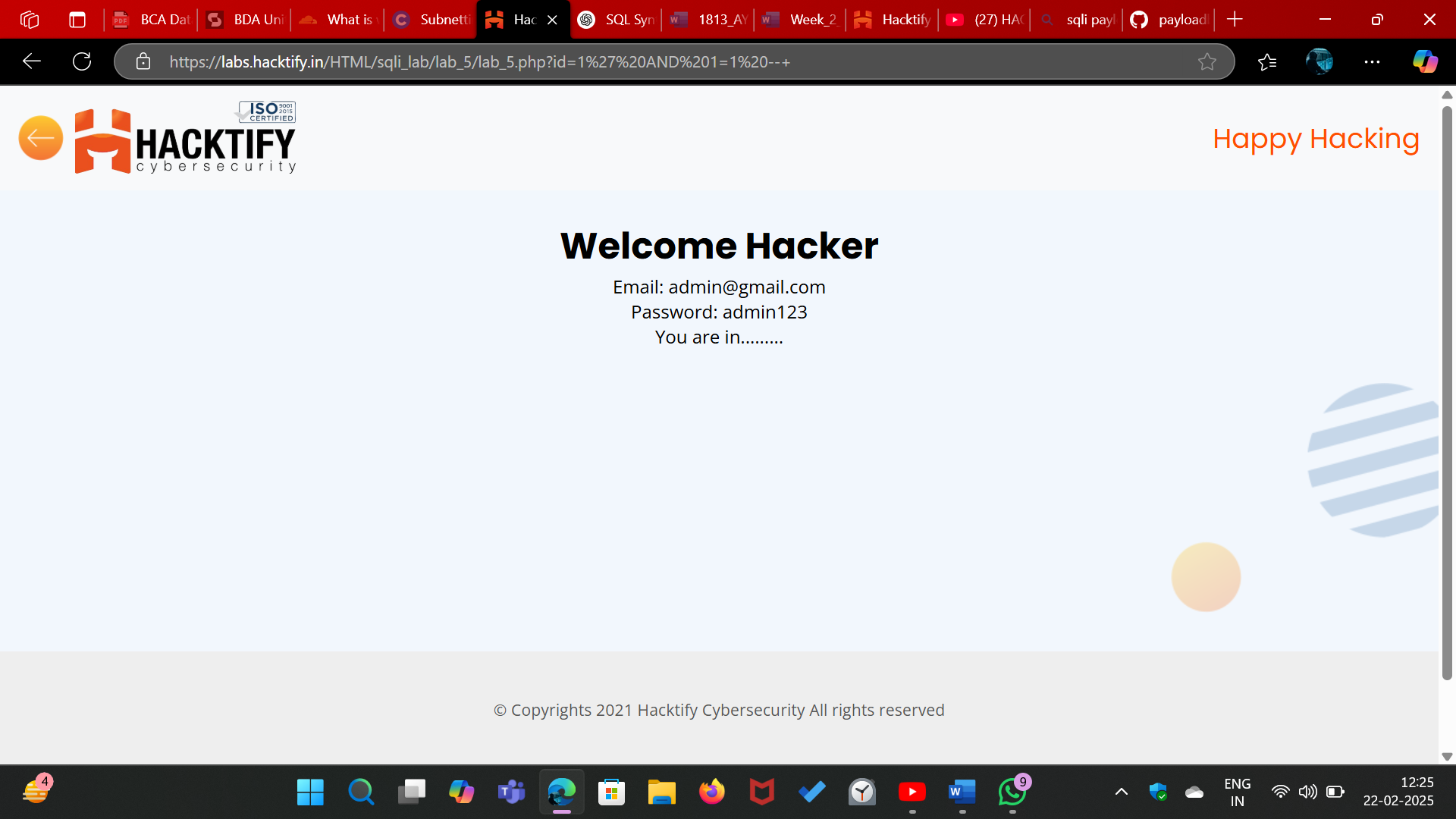
# Proof of Concept



# 2.5. Booleans and Blind!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Booleans and Blind! | **High** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection via the URL parameter (id=1%27%20AND%201=1%20--%20-) allows unauthorized data retrieval. | |
| **How It Was Discovered** | |
| Modifying the id parameter in the URL with an SQL injection payload resulted in a successful login and displayed admin credentials. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can extract sensitive information, manipulate database entries, escalate privileges, and compromise the entire system | |
| **Suggested Countermeasures** | |
| Use prepared statements and parameterized queries, validate and sanitize user inputs, restrict error messages, and implement web application firewalls (WAF). | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

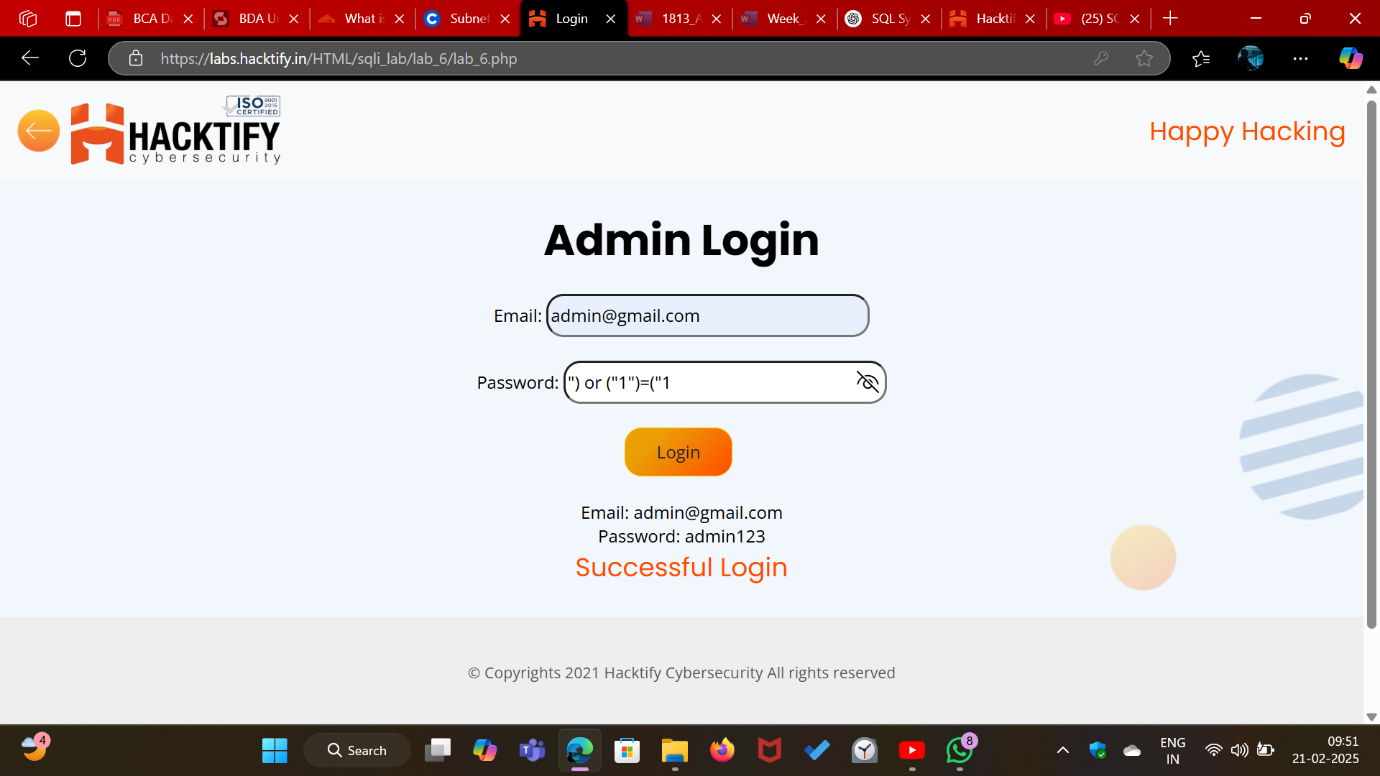
# Proof of Concept



# 2.6. Error Based : Tricked

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Error Based : Tricked | **Medium** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection via the login form allows bypassing authentication by injecting a crafted SQL statement in the password field. | |
| **How It Was Discovered** | |
| Entering ") or ("1"="1 in the password field resulted in a successful login, revealing admin credentials without requiring the correct password. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_6/lab_6.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can gain unauthorized access to admin accounts, extract sensitive data, escalate privileges, and potentially take full control of the system. | |
| **Suggested Countermeasures** | |
| Implement prepared statements with parameterized queries, enforce strict input validation, use least-privilege database accounts, and deploy a Web Application Firewall (WAF) to detect and block SQL injection attempts. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

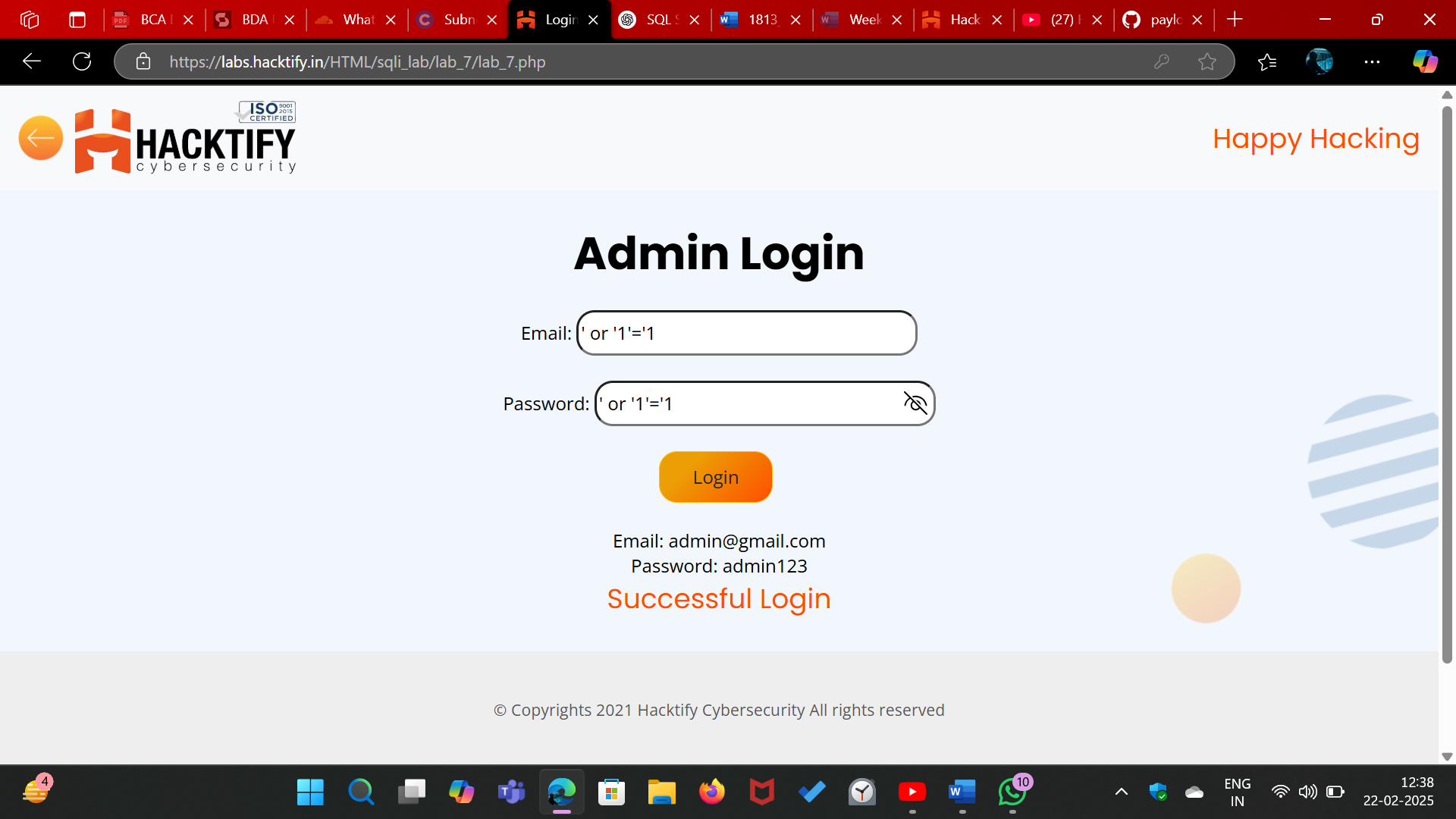
# Proof of Concept



# 2.7. Errors and Post!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Errors and Post! | **Low** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection via both the email and password fields allows authentication bypass by injecting a universally true SQL condition (' OR '1'='1). | |
| **How It Was Discovered** | |
| Entering ' OR '1'='1 in both the email and password fields resulted in a successful login, revealing admin credentials without requiring valid login details. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can gain unauthorized access to user accounts, retrieve sensitive database information, modify or delete records, and potentially escalate privileges to take full control of the system. | |
| **Suggested Countermeasures** | |
| Use prepared statements with parameterized queries, enforce strict input validation, escape special characters, disable detailed error messages, and implement a Web Application Firewall (WAF) to detect and block SQL injection attempts. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

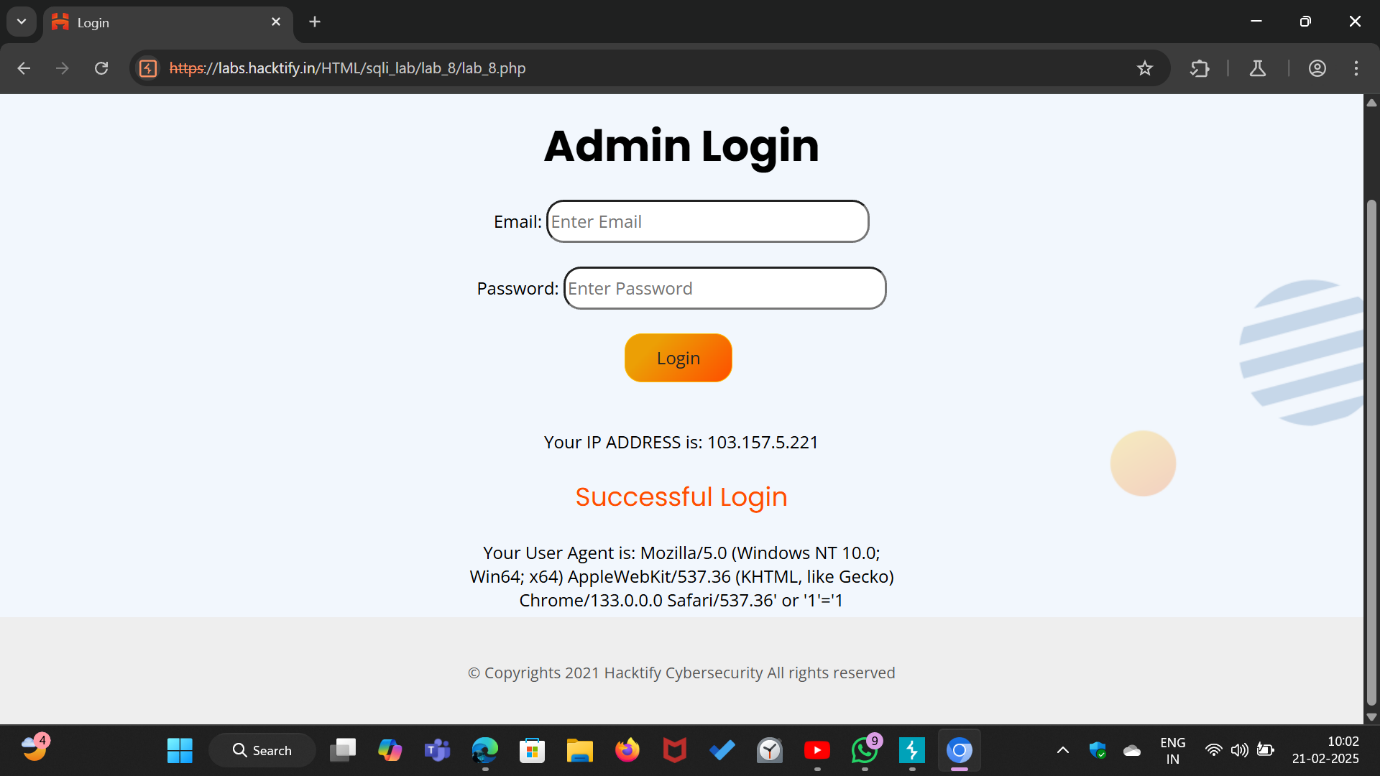
# Proof of Concept



# 2.8. User Agents lead us!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| User Agents lead us! | **High** |
| **Tools Used** | |
| Burpsuite | |
| **Vulnerability Description** | |
| SQL Injection via the User-Agent header allows authentication bypass by injecting a condition (' OR '1'='1) into the request header. | |
| **How It Was Discovered** | |
| Modifying the User-Agent field in the HTTP request to include ' OR '1'='1 resulted in a successful login, demonstrating that the input is directly used in an SQL query without proper sanitization. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can bypass authentication, execute arbitrary SQL queries, extract sensitive database information, and manipulate user session data. This can lead to account takeovers and full system compromise. | |
| **Suggested Countermeasures** | |
| Sanitize and validate all HTTP headers, use prepared statements with parameterized queries, restrict unnecessary headers in authentication logic, and implement Web Application Firewalls (WAF) to block SQL injection attempts. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

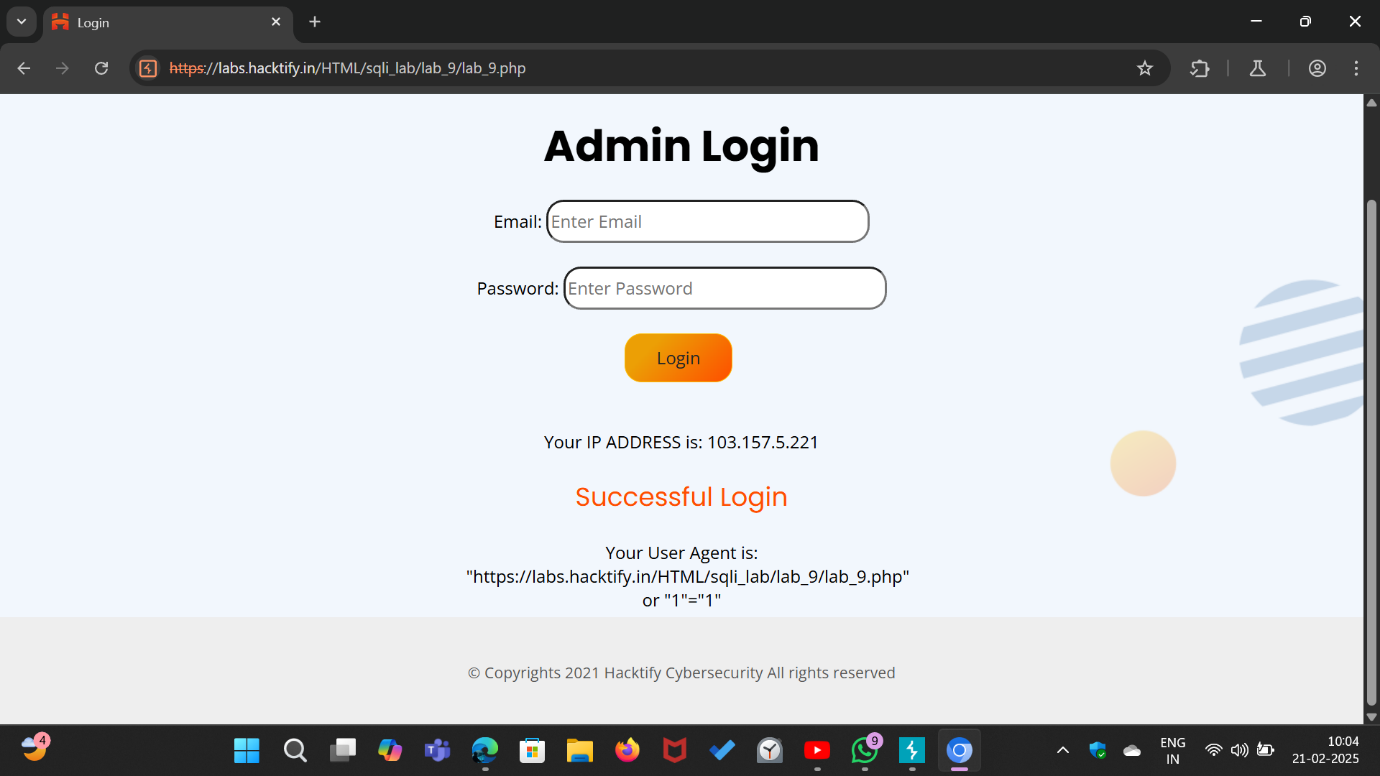
# Proof of Concept



# 2.9. Referer lead us!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Referer lead us! | **Medium** |
| **Tools Used** | |
| Bursuite | |
| **Vulnerability Description** | |
| SQL Injection via the Referer header allows authentication bypass by injecting a condition (' OR '1'='1) into the HTTP request header. | |
| **How It Was Discovered** | |
| Modifying the Referer header in the HTTP request to include ' OR '1'='1 resulted in a successful login, indicating that the application improperly handles user-supplied input from HTTP headers in SQL queries. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can bypass authentication, access unauthorized accounts, execute arbitrary SQL queries, extract sensitive data, and modify database contents. This can lead to full system compromise and data breaches. | |
| **Suggested Countermeasures** | |
| Sanitize and validate all HTTP headers, avoid using the Referer header for authentication logic, implement prepared statements with parameterized queries, and deploy Web Application Firewalls (WAF) to detect and block SQL injection attempts. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

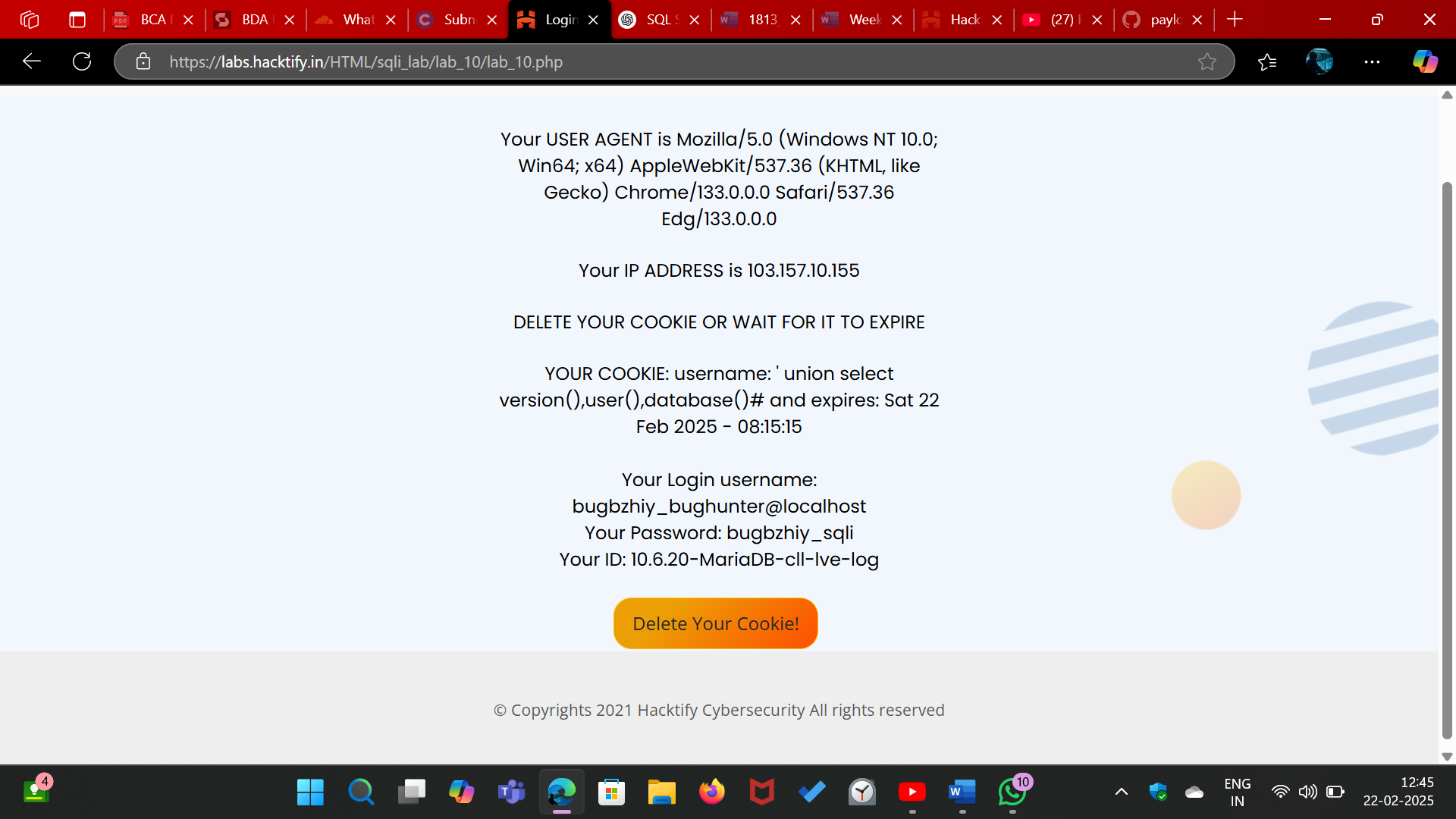
# Proof of Concept



# 2.10. Oh Cookies!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| Oh Cookies! | **High** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection via cookie manipulation allows unauthorized data retrieval by injecting a UNION SELECT statement into the username cookie parameter. | |
| **How It Was Discovered** | |
| Modifying the username cookie value to ' UNION SELECT version(), user(), database()# resulted in the disclosure of database details, including the database version, current user, and active database. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can extract sensitive database information, manipulate session authentication, escalate privileges, and potentially execute further SQL injection attacks to compromise the entire system. | |
| **Suggested Countermeasures** | |
| Use secure cookie handling by encrypting and signing cookies, validate and sanitize cookie inputs before using them in SQL queries, implement prepared statements with parameterized queries, and deploy Web Application Firewalls (WAF) to detect and block malicious cookie modifications. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

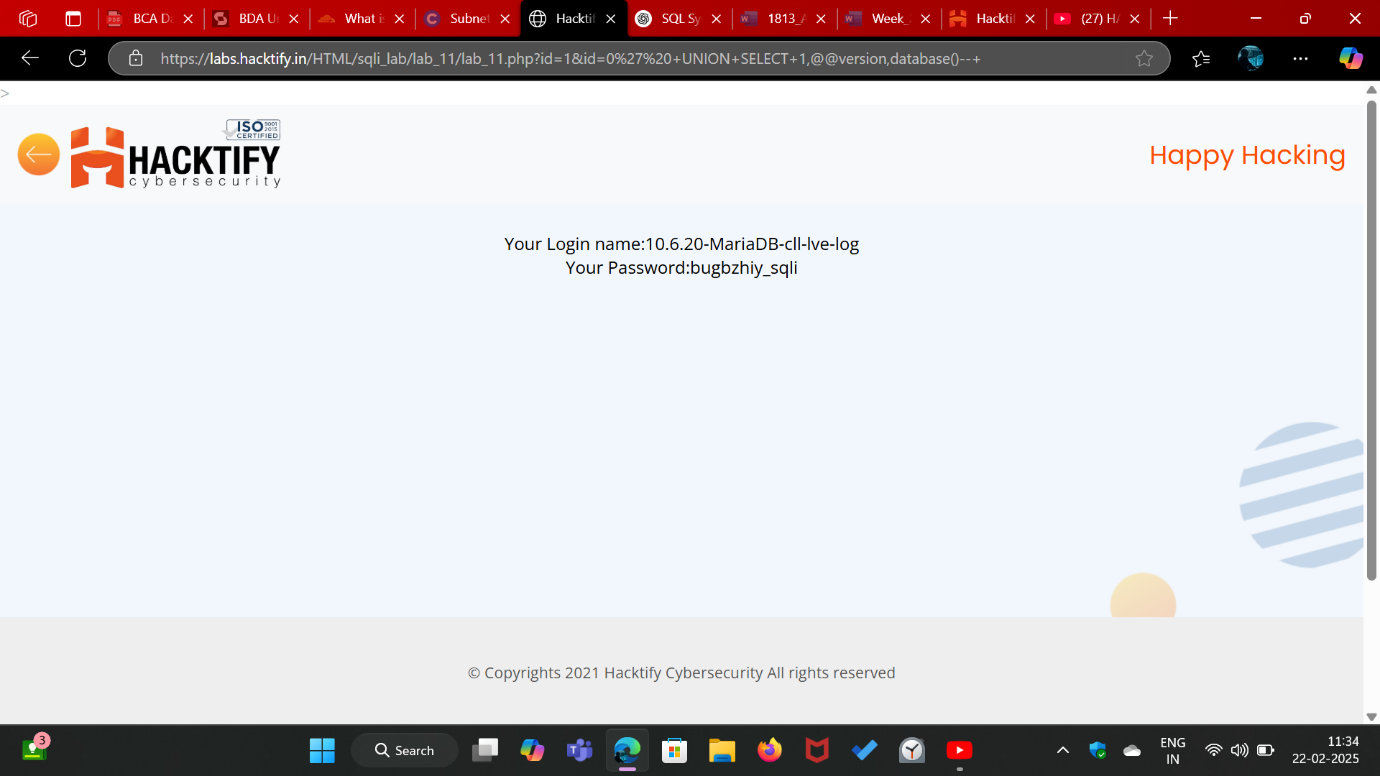
# Proof of Concept



# 2.11. WAF’s are injected!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| WAF’s are injected! | **High** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection via the id URL parameter allows unauthorized database information disclosure through a UNION SELECT query | |
| **How It Was Discovered** | |
| Modifying the id parameter in the URL to id=0&id=1’ +UNION+SELECT+1,@@version,database()--+ successfully retrieved and displayed the database version and name. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can extract database metadata, enumerate database structures, escalate privileges, and execute further SQL injection attacks, potentially leading to full database compromise. | |
| **Suggested Countermeasures** | |
| Implement prepared statements with parameterized queries, validate and sanitize user inputs, limit database permissions for web applications, and deploy Web Application Firewalls (WAF) to detect and block malicious SQL injection attempts. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

# Proof of Concept

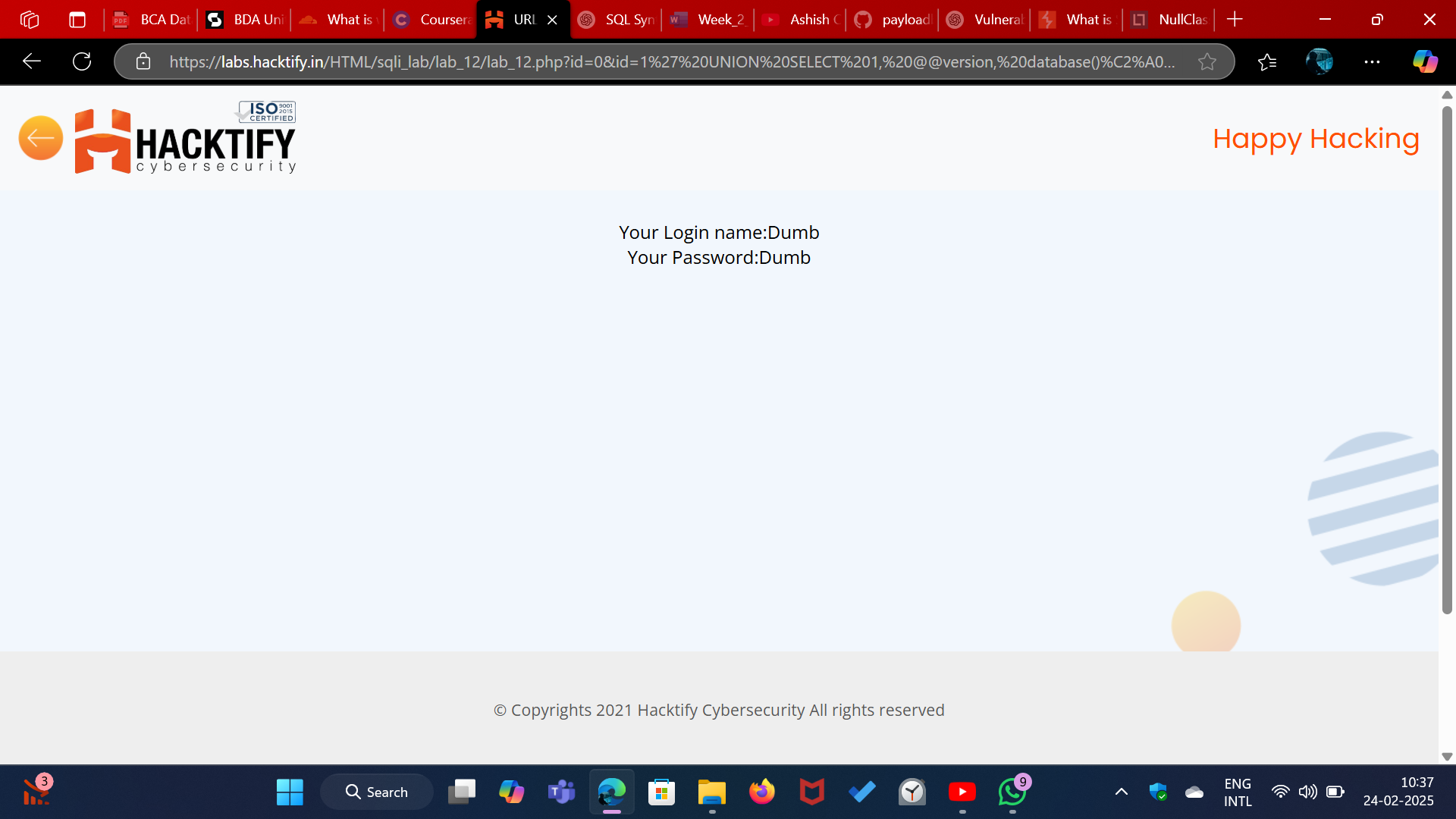


# 

# 2.12. WAF’s are injected Part2!

|  |  |
| --- | --- |
| **Reference** | **Risk Rating** |
| WAF’s are injected Part2! | **Medium** |
| **Tools Used** | |
| Manual Analysis | |
| **Vulnerability Description** | |
| SQL Injection via the id URL parameter allows unauthorized database information disclosure through a UNION SELECT query | |
| **How It Was Discovered** | |
| Modifying the id parameter in the URL to id=1&id=0’ +UNION+SELECT+1,@@version,database()--+ successfully retrieved and displayed the database version and name. | |
| **Vulnerable URLs** | |
| <https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php> | |
| **Consequences of not Fixing the Issue** | |
| Attackers can extract database metadata, enumerate database structures, escalate privileges, and execute further SQL injection attacks, potentially leading to full database compromise. | |
| **Suggested Countermeasures** | |
| Implement prepared statements with parameterized queries, validate and sanitize user inputs, limit database permissions for web applications, and deploy Web Application Firewalls (WAF) to detect and block malicious SQL injection attempts. | |
| **References** | |
| <https://github.com/payloadbox/sql-injection-payload-list>  <https://portswigger.net/web-security/sql-injection>  <https://owasp.org/www-community/attacks/SQL_Injection> | |

# Proof of Concept

****