MINI-PROJECT REPORT ON

“SQL INJECTION VULERABILITY DEMONSTRATION”

Submitted By

Mr. Thorve Avishkar Shrikrushna

Roll No: 63

Exam No: T400840394

Class: TE AIDS

UNDER THE GUIDANCE OF

Prof. Jadhav. S. P.

Department of Artificial Intelligence and Data Science Engineering

Jaihind College of Engineering, Kuran

A/p-Kuran, Tal-Junnar, Dist-Pune-410511, State Maharashtra, India

2024-2025

Department of Artificial Intelligence and Data Science Engineering

Jaihind College of Engineering, Kuran

A/p-Kuran, Tal-Junnar, Dist-Pune-410511, State Maharashtra, India

2024-2025

CERTIFICATE

This is to certify that the Internship Report Entitled

"SQL INJECTION VULERABILITY DEMONSTRATION"

SUBMITTED BY

Thorve Avishkar Shrikrushna

Is a Bonafide work carried out by student under the supervision of **Prof. Jadhav S. P.** and it is submitted towards the partial fulfilment of the requirement of third year of Engineering in Artificial intelligence and data science under the Savitribai Phule Pune University during the academic year 2024-2025.

Prof. S. P. Jadhav	**Prof. S. K. Said**
Subject Teacher	HOD AI&DS
JCOE, Kuran.	JCOE, Kuran.

ACKNOWLEDGEMENTS

With deep sense of gratitude we would like to thank all the people who have lit our path with their kind guidance. We are very grateful to these intellectuals who did their best to help during our project work. It is our proud privilege to express a deep sense of gratitude to Principal of Jaihind College Of Engineering, Kuran **Dr. D. J. Garkal** for his comments and kind permission to complete this project. We remain indebted to H.O.D. Artificial Intelligence And Data Science Engineering Department **Prof. S. K. Said,** for his timely suggestion and valuable guidance. The special gratitude goes to **Prof. Jadhav. S. P.** excellent and precious guidance in completion of this work .We thanks to all the colleagues for their appreciable help for our working project. With various industry owners or lab technicians to help, it has been our endeavor throughout our work to cover the entire project work. We are also thankful to our parents who provided their wishful support for our project completion successfully .And lastly we thank our all friends and the people who are directly or indirectly related to our project work.

Yours faithfully,

Mr. Thorve Avishkar Shrikrushna.

ABSTRACT

This project focuses on demonstrating the exploitation of an SQL injection vulnerability in a web application to retrieve and display all product details from the backend database, including sensitive and unreleased items. SQL injection is one of the most common and critical security vulnerabilities, allowing attackers to manipulate queries executed by the database. The project begins with identifying a vulnerable input parameter in the product category filter of a website. By injecting malicious SQL code (' OR 1=1--), the application bypasses its intended functionality, exposing all product details stored in the database.

The report outlines the methodology used to perform the attack, including crafting the payload, executing it, and analyzing the results. The impact of this vulnerability is assessed, highlighting risks such as unauthorized data access, potential data breaches, and further exploitation possibilities. Additionally, preventive measures are proposed to mitigate SQL injection vulnerabilities, including secure coding practices like parameterized queries, input validation, and deploying web application firewalls (WAFs).

This project emphasizes the importance of secure software development and ethical hacking practices to identify and address vulnerabilities proactively. By understanding how SQL injection works and implementing robust defenses, organizations can protect their applications from malicious attacks and ensure data security.

INDEX

Chapter 1

INTRODUCTION

SQL Injection (SQLi) is a critical and widely recognized security vulnerability that occurs when attackers manipulate a web application's SQL queries by injecting malicious SQL code into input fields. This exploitation allows attackers to gain unauthorized access to sensitive data, manipulate database contents, or even compromise the entire system. SQL Injection has been identified as one of the top vulnerabilities in web applications, as highlighted by the OWASP Top 10 list.

How SQL Injection Works

SQL Injection exploits improper validation or sanitization of user inputs in SQL queries. For example, if a web application directly incorporates user-provided data into a query without proper safeguards, an attacker can inject harmful code to alter the query's logic. Consider the following example:

Intended Query:

SQL:

SELECT * FROM users **WHERE** username = 'admin' AND password = 'password';

Types of SQL Injection:

SQL Injection attacks can take various forms, including:

1. **Classic SQL Injection**: Directly injecting malicious SQL statements.
2. **Error-Based SQL Injection**: Exploiting database error messages to gather information about the database schema.
3. **Union-Based SQL Injection**: Using the UNION operator to retrieve data from other database tables.
4. **Blind SQL Injection**: Inferring database information by observing application behavior without direct feedback.
5. **Time-Based Blind SQL Injection**: Using time delays to infer responses from the database. Inferring database information by observing application behavior without direct feedback.

Consequences of SQL Injection:

The impact of SQL Injection attacks can be severe, including:

- **Data Breaches**: Unauthorized access to confidential data such as usernames, passwords, and financial records.
- **Data Manipulation**: Attackers can modify, delete, or insert data into the database.
- **Privilege Escalation**: Attackers may gain administrative control over the database.
- **System Compromise**: In extreme cases, attackers can execute commands on the underlying server or establish persistent backdoors for long-term exploitation

1.1 MOTIVATION

SQL injection attacks are motivated by several factors, primarily driven by the potential for financial gain, data theft, and system compromise. One of the primary motivations is financial gain, where attackers target sensitive data such as credit card numbers, passwords, and personal identifiable information (PII) to sell on the dark web or use for identity theft. Additionally, some attackers use SQL injection to deploy ransomware, encrypting data and demanding payment for decryption keys. This financial incentive makes SQL injection a lucrative option for malicious actors. Another significant motivation is system compromise, where attackers aim to gain administrative access to databases. This allows them to modify or delete data, or even execute system-level commands, providing a persistent foothold for further attacks. By compromising a system, attackers can install backdoors or malware, enabling them to maintain access over time. This level of control can be used for corporate espionage, disrupting competitors' operations by stealing proprietary information or sabotaging their databases.

1.2 PROBLEM STATEMENT

SQL Injection is a cyber-attack that targets vulnerabilities in web applications by injecting malicious SQL code into database queries. This type of attack exploits weak input validation mechanisms, allowing attackers to manipulate SQL statements and gain unauthorized access to sensitive data or perform malicious actions. SQL Injection has been a persistent threat since its discovery in 1998, despite being relatively easy to prevent through proper coding practices. It remains one of the most common security vulnerabilities, as highlighted by numerous cybersecurity reports.

1.3 OBJECTIVE

1. To Understand SQL Injection Vulnerabilities: The first objective is to thoroughly comprehend how SQL injection attacks occur, including the types of vulnerabilities that lead to these attacks and the methods attackers use to exploit them.

2. To Demonstrate SQL Injection Exploitation: This objective involves simulating an SQL injection attack on a controlled environment to illustrate how attackers can manipulate database queries and access sensitive data.

3. To Develop Secure Coding Practices: The goal here is to establish guidelines for secure coding practices that prevent SQL injection vulnerabilities. This includes using parameterized queries, stored procedures, and input validation techniques to ensure that user inputs are sanitized and cannot be used to inject malicious SQL code.

4. To Implement Robust Security Measures: This objective focuses on deploying additional security tools and techniques to protect against SQL injection attacks. This includes using web application firewalls (WAFs), conducting regular security audits, and implementing least-privilege access controls to limit the impact of a successful attack.

5. To Raise Awareness Among Developers: The objective is to educate developers about the risks associated with SQL injection and the importance of integrating security into the development lifecycle.

6. To Evaluate and Improve Existing Security Protocols: Finally, the objective is to assess current security protocols and procedures within organizations to identify gaps and areas for improvement. This involves reviewing existing codebases, testing for vulnerabilities, and updating security policies to ensure they are aligned with best practices for preventing SQL injection attacks.

1.4 DESCRIPTION

SQL Injection (SQLi) is a critical security vulnerability that occurs when attackers exploit weaknesses in an application's SQL query execution by injecting malicious SQL code into input fields. This vulnerability arises when user inputs are not properly validated, sanitized, or parameterized, allowing attackers to manipulate database queries. SQL injection attacks can result in unauthorized access to sensitive data, data manipulation, or even full control over the database and underlying systems. Attackers typically begin by identifying vulnerable inputs in a web application, such as login forms, search bars, or URL parameters. They then craft malicious SQL queries designed to alter the intended behaviour of the application's database query. For example, an attacker might inject a payload like ' OR 1=1-- into a login form to bypass authentication. When executed, this payload modifies the query to always return true, granting unauthorized access.

The consequences of SQL injection attacks can be severe. They may lead to exposure of confidential information such as usernames, passwords, credit card details, or health records. Attackers can also modify or delete data, escalate privileges to gain administrative access, and in some cases execute operating system commands to compromise the entire server. Advanced SQL injection techniques can even render applications offline or disrupt entire network infrastructures. SQL injection attacks are categorized into various types, including in-band (direct data retrieval), error-based (leveraging error messages), blind (inferring information through application behaviour), and out-of-band (using alternative channels for data exfiltration). Despite being one of the oldest attack vectors, SQL injection remains prevalent due to its simplicity and the widespread use of relational databases in web applications.

To mitigate these risks, developers must adopt secure coding practices such as using parameterized queries and prepared statements, implementing strict input validation and sanitization, and deploying web application firewalls (WAFs). Regular security audits and penetration testing are also essential to identify and address potential vulnerabilities. By proactively securing applications against SQL injection, organizations can protect their data and systems from this persistent threat.

Chapter 2

SYSTEM REQUIREMENTS SPECIFICATIONS

2.1 SOFTWARE REQUIREMENT:

1. Web Server Software: Apache HTTP Server, Nginx, or Microsoft Internet Information Services

2. Database Management System (DBMS): MySQL, PostgreSQL, SQLite, or any other relational database system supported by the chosen web application framework

3. Web Application Framework: Django (Python), Flask (Python)

4. Programming Languages: HTML, CSS, JavaScript

5. Development Tools: Text Editor or Integrated Development Environment (IDE), Visual Studio Code, Sublime Text, PyCharm, Atom, etc.

6. Web Browser: Google Chrome, Mozilla Firefox, Microsoft Edge, or Safari.

2.2 HARDWARE REQUIREMENT:

1. Computer: Desktop, laptop, or server-grade machine capable of running the required software

2. Processor (CPU): Dual-core or higher processor for optimal performance

3. RAM: Minimum of 4 GB RAM, though 8 GB or more is recommended for smoother operation, especially when running multiple software components simultaneously

4. Storage: Sufficient disk space to install the operating system, web server software, database management system, development tools, and project files

5. Network Interface: Ethernet port or Wi-Fi adapter for network connectivity, allowing access to the internet and local network resources

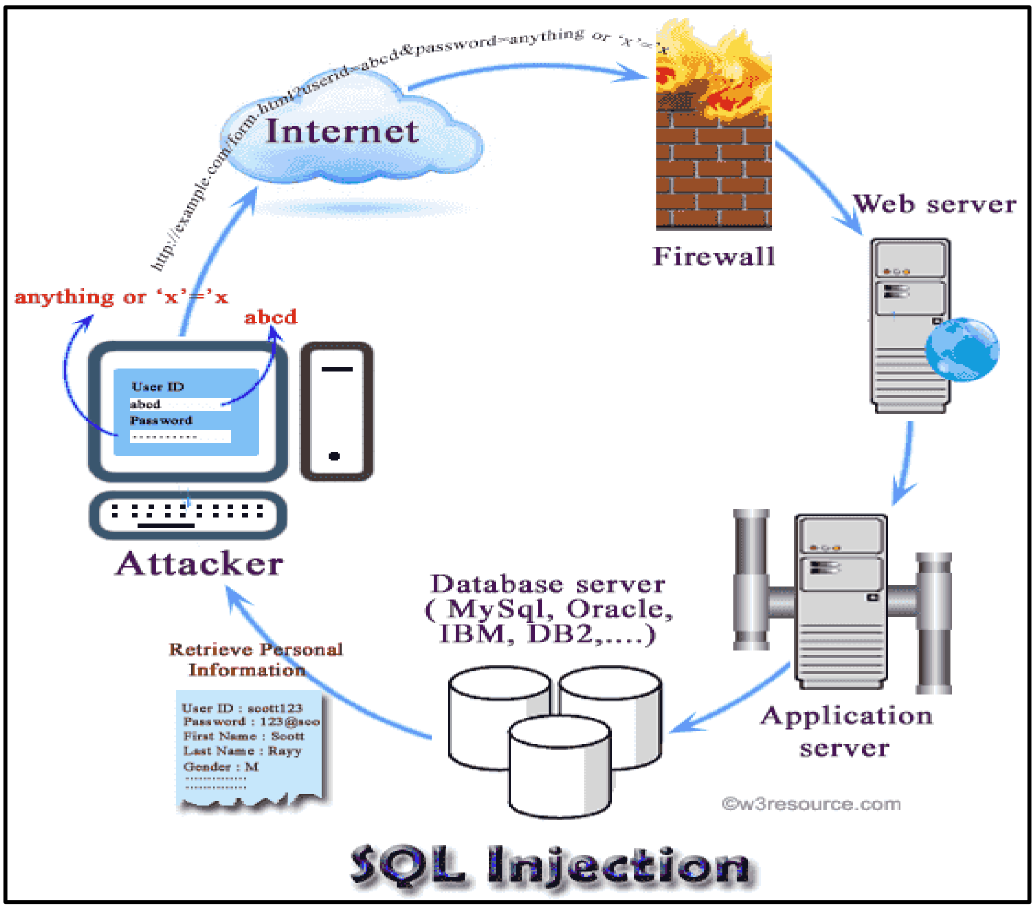
6. Display: Monitor, screen, or display device for visual output, configuration, and interaction with the development environment

Chapter 3

SYSTEM ARCHITECTURE

3.1 SQL INJECTION:

SQL injection vulnerabilities typically arise in systems where user inputs are directly concatenated into SQL queries without proper sanitization. Here's a typical architecture for demonstrating this vulnerability:



Core Components:

1. Client-Side Interface

- Web forms (login/search fields)
- HTTP request handlers (GET/POST methods)
- Basic input fields without client-side validation

2. Vulnerable Server-Side Processing:

- Dynamic SQL query construction using string concatenation.

query = "SELECT * FROM users WHERE username='" + user_input + "' AND password='" + pass_input + "'"

- No parameterization or prepared statements
- Error messages revealing database structure

3. Database Layer:

- SQL database (MySQL, PostgreSQL, etc.)
- Tables with sensitive data (users, payments, etc.)
- Privileged database user account with excessive permissions

4. Key Vulnerable Patterns:

- Concatenation of user inputs into SQL strings
- Overprivileged database accounts
- Detailed error messages returned to client
- Lack of input validation for special characters

This architecture intentionally omits security controls like parameterized queries, input sanitization, and principle of least privilege to demonstrate how SQL injection exploits occur. Real-world systems should implement prepared statements, strict input validation, and proper error handling to prevent these attacks.

Dynamic SQL query construction using string concatenation: a case, a crafted input can be given that when embedded in the response acts as a JS code block and is executed by the browser.

3.2 STORED XSS:

When the response containing the payload is stored on the server in such a way that the script gets executed on every visit without submission of payload, then it is identified as stored XSS. An example of stored XSS is XSS in the comment thread.

Chapter 4

IMPLEMENTATION

4.1 IMPLEMENTATION STEPS:

SQL injection vulnerabilities arise from improper handling of user inputs in SQL queries. By implementing parameterized queries, stored procedures, input validation, least privilege principles, and escaping special characters, developers can effectively prevent these attacks and safeguard their applications from unauthorized access or data breaches

1. Parameterized Queries (Prepared Statements):

Separate SQL logic from user inputs using placeholders, preventing input from being interpreted as executable code.

```
String query = "SELECT * FROM users WHERE username = ? AND password = ?";


PreparedStatement stmt = connection.prepareStatement(query);

stmt.setString(1, username);  // Input treated as data

stmt.setString(2, password);

ResultSet rs = stmt.executeQuery();
```

2. Input Validation and Sanitization:

Restrict inputs to allowed formats and remove harmful characters.

```
$username = filter_var($_POST['username'], FILTER_SANITIZE_STRING); // Removes
 HTML tags $email = filter_var($_POST['email'], FILTER_SANITIZE_EMAIL); // Validates
email format.
```

3. Escaping Special Characters:

Neutralize characters like ', ", and ; using database-specific functions.

```
$user = mysqli_real_escape_string($connection, $_POST['user']);

$query = "SELECT * FROM users WHERE username = '$user'";
```

4. Stored Procedures:

Use predefined SQL procedures to limit dynamic query execution.

```
CREATE PROCEDURE GetUser (IN user VARCHAR(255))

BEGIN

   SELECT * FROM users WHERE username = user;

END;
```

5. Least Privilege Access:

Restrict database account permissions:

- Grant read-only accesses for applications that don't modify data.
- Avoid using root or admin accounts for routine operations.

6. Web Application Firewalls (WAFs):

Deploy WAFs to filter malicious traffic and block injection patterns.

7. Regular Security Audits:

- Perform penetration testing to identify vulnerabilities.
- Monitor logs for unusual query patterns (e.g., bulk data access).

7.Documentation and Reporting:

By combining parameterized queries, input validation, and strict access controls, developers can effectively neutralize SQL injection risks. Regular updates and security audits further ensure long-term protection against evolving threats. By following these steps, you can effectively implement the Cross-Site Scripting (XSS) using stored attack mini project, gain practical experience in identifying and mitigating XSS vulnerabilities, and contribute to the development of more secure web applications.

4.2 SOURCE CODE

1. Database Setup:

Create a sample database with a user’s table:

```
CREATE DATABASE sql_injection_demo;

USE sql_injection_demo;

CREATE TABLE users (

    id INT AUTO_INCREMENT PRIMARY KEY,

    username VARCHAR(50) NOT NULL,

    password VARCHAR(50) NOT NULL

);

INSERT INTO users (username, password) VALUES ('admin', 'password123'), ('user', 'userpass');
```

2. Vulnerable Login Form:

This form demonstrates how SQL injection can be exploited.

- **HTML Form:**

```
<!DOCTYPE html>
<html>
<head>
    <title>SQL Injection Demo</title>
</head>
<body>
    <h1>Vulnerable Login Form</h1>
    <form method="POST" action="vulnerable_login.php">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username"><br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password"><br><br>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

- **PHP Code (vulnerable_login.php)**:

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "sql_injection_demo";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $user = $_POST['username'];
    $pass = $_POST['password'];

    // Vulnerable query
    $sql = "SELECT * FROM users WHERE username = '$user' AND password = '$pass'";
    $result = $conn->query($sql);

    if ($result->num_rows > 0) {
        echo "Login successful!";
    } else {
        echo "Invalid credentials!";
    }
}
```

```
?>
```

SQL Injection Example:

```
SELECT * FROM users WHERE username = 'admin' --' AND password = 'anything';
```

3. Secure Login Form:

- **HTML Form:**

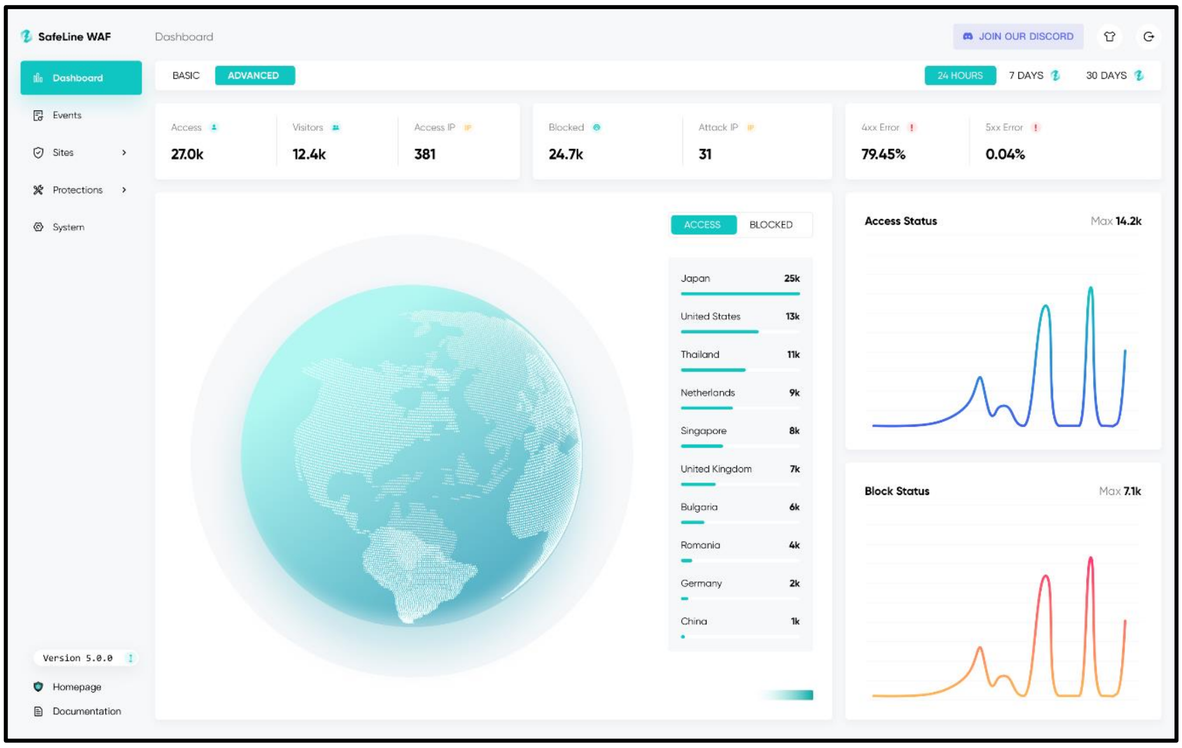
```
<!DOCTYPE html>
<html>
<head>
   <title>Secure Login Demo</title>
</head>
<body>
   <h1>Secure Login Form</h1>
   <form method="POST" action="secure_login.php">
      <label for="username">Username:</label>
      <input type="text" id="username" name="username"><br><br>
      <label for="password">Password:</label>
      <input type="password" id="password" name="password"><br><br>
      <button type="submit">Login</button>
   </form>
</body>
</html>
```

- **PHP Code (secure_login.php)**:

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "sql_injection_demo";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $user = $_POST['username'];
    $pass = $_POST['password'];

    // Secure query using prepared statements
    $stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password
= ?");
    $stmt->bind_param("ss", $user, $pass);
    $stmt->execute();
    $result = $stmt->get_result();

    if ($result->num_rows > 0) {
        echo "Login successful!";
    } else {
        echo "Invalid credentials!";
    }
}
?>
```

Chapter 5
RESULTS

1) Dashboard:



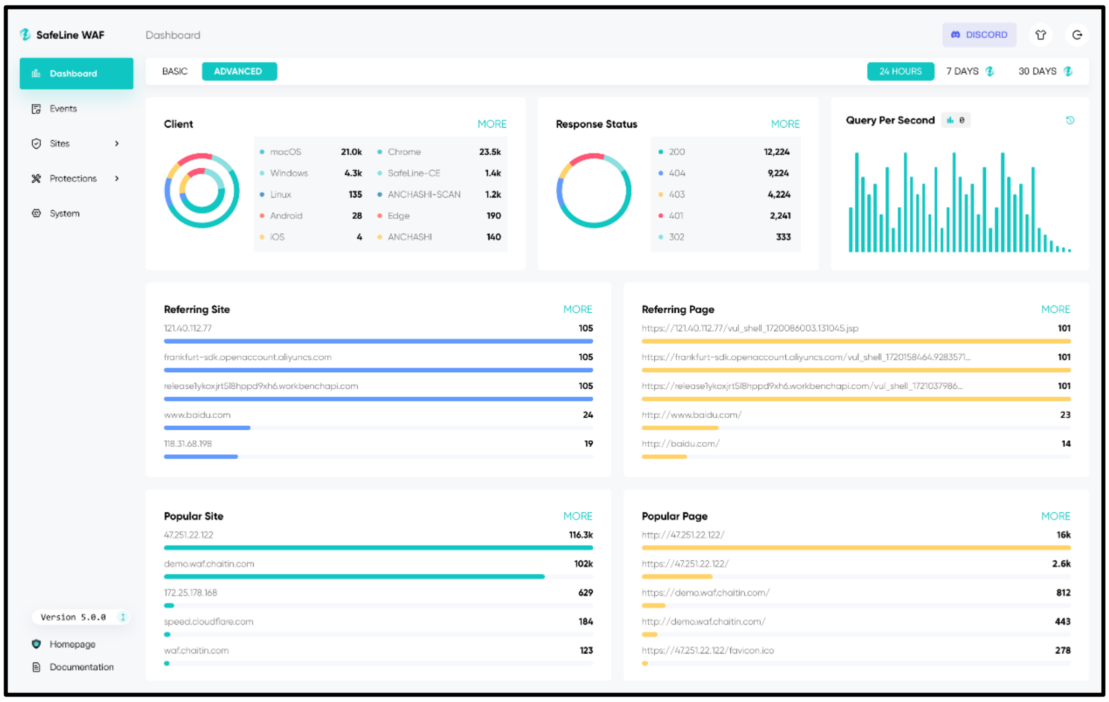
Access	Visitors	Access IP	Blocked	Attack IP	4xx Error	5xx Error
27.0k	12.4k	381	24.7k	31	79.45%	0.04%

Japan	25k
United States	13k
Thailand	11k
Netherlands	9k
Singapore	8k
United Kingdom	7k
Bulgaria	6k
Romania	4k
Germany	2k
China	1k

Access Status Max 14.2k

Block Status Max 7.1k

2) Viewer:



Client

macOS	21.0k	Chrome	23.5k
Windows	4.3k	SafeLine-CE	1.4k
Linux	135	ANCHASHI-SCAN	1.2k
Android	28	Edge	190
iOS	4	ANCHASHI	140

Response Status

200	12,224
404	9,224
403	4,224
401	2,241
302	333

Referring Site

121.40.112.77	105
frankfurt-sdk.openaccount.aliyuncs.com	105
release1ykoxjrt5l8hppd9xh6.workbenchapi.com	105
www.baidu.com	24
118.31.68.198	19

Referring Page

https://121.40.112.77/vul_shell_1720086003.131045.jsp	101
https://frankfurt-sdk.openaccount.aliyuncs.com/vul_shell_1720158464.9283571...	101
https://release1ykoxjrt5l8hppd9xh6.workbenchapi.com/vul_shell_1721037986...	101
http://www.baidu.com/	23
http://baidu.com/	14

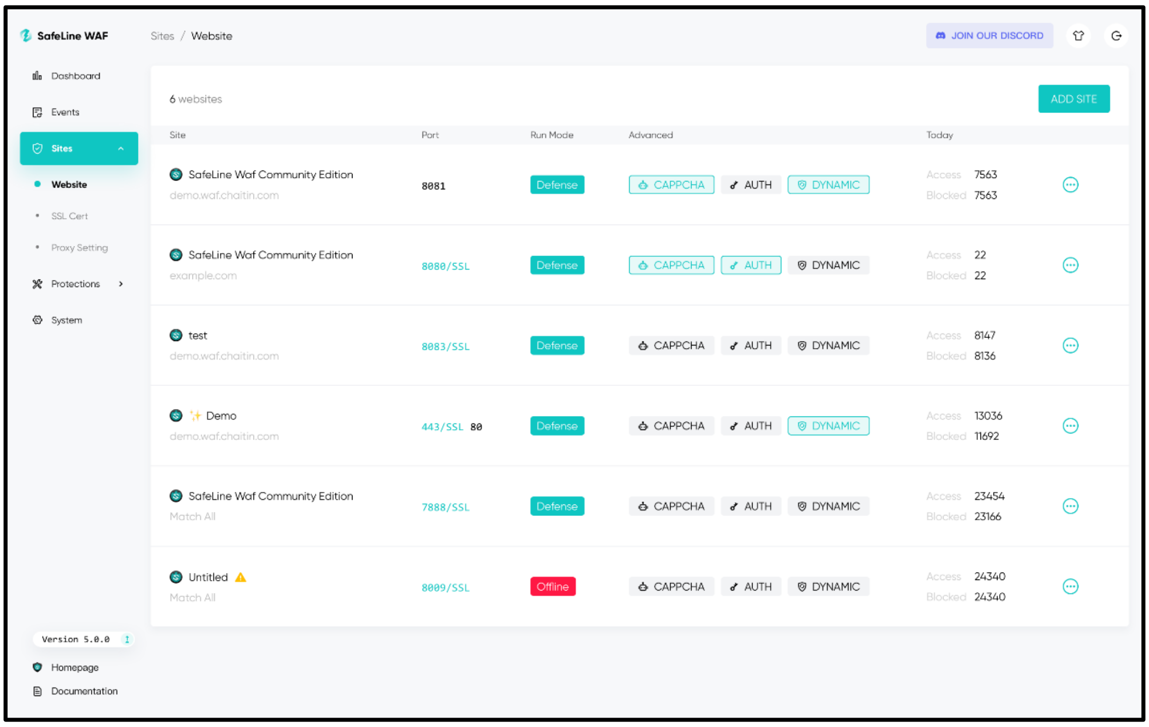
Popular Site

47.251.22.122	116.3k
demo.waf.chaitin.com	102k
172.25.178.168	629
speed.cloudflare.com	184
waf.chaitin.com	123

Popular Page

http://47.251.22.122/	16k
https://47.251.22.122/	2.6k
https://demo.waf.chaitin.com/	812
http://demo.waf.chaitin.com/	443
https://47.251.22.122/favicon.ico	278

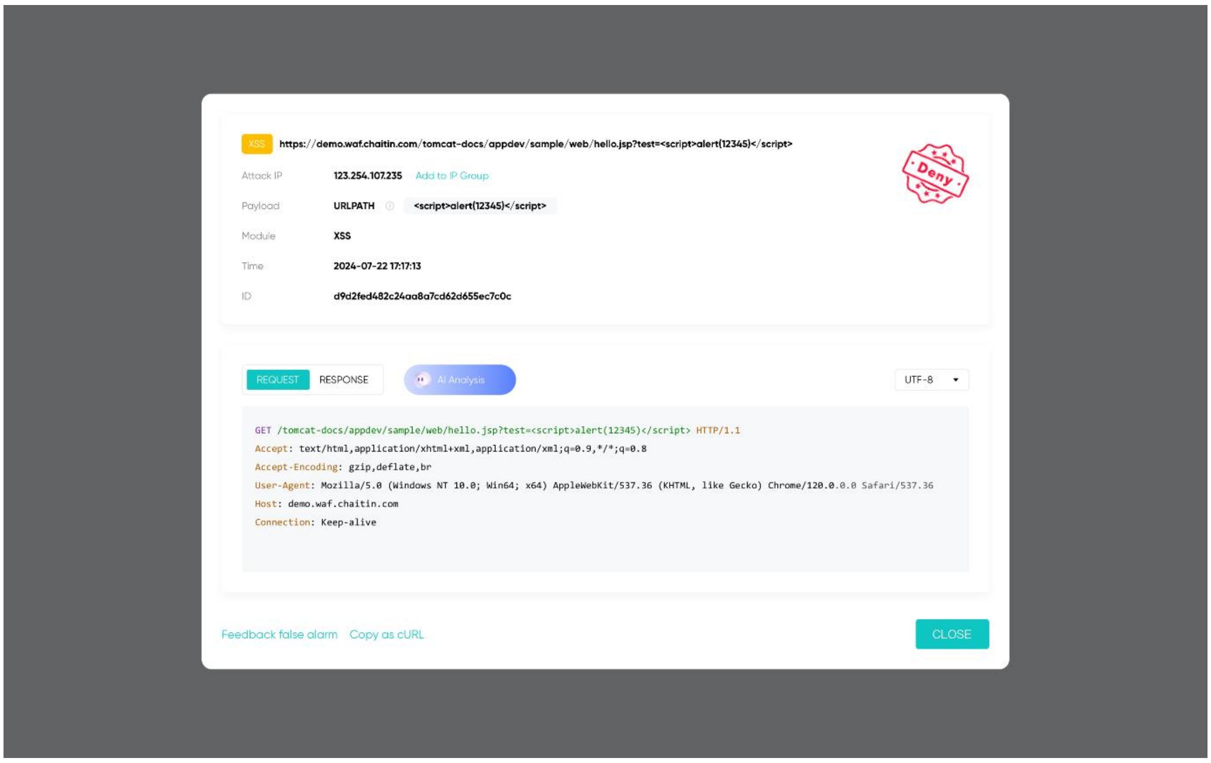
3) Website Page:



6 websites

Site	Port	Run Mode	Advanced	Today
SafeLine Waf Community Edition demo.waf.chaitin.com	8081	Defense	CAPPCHA, AUTH, DYNAMIC	Access 7563 Blocked 7563
SafeLine Waf Community Edition example.com	8080/SSL	Defense	CAPPCHA, AUTH, DYNAMIC	Access 22 Blocked 22
test demo.waf.chaitin.com	8083/SSL	Defense	CAPPCHA, AUTH, DYNAMIC	Access 8147 Blocked 8136
✨ Demo demo.waf.chaitin.com	443/SSL 80	Defense	CAPPCHA, AUTH, DYNAMIC	Access 13036 Blocked 11692
SafeLine Waf Community Edition Match All	7888/SSL	Defense	CAPPCHA, AUTH, DYNAMIC	Access 23454 Blocked 23166
Untitled ⚠ Match All	8009/SSL	Offline	CAPPCHA, AUTH, DYNAMIC	Access 24340 Blocked 24340

4) Authentication Page:



XSS https://demo.waf.chaitin.com/tomcat-docs/appdev/sample/web/hello.jsp?test=<script>alert(12345)</script>

Attack IP	123.254.107.235 Add to IP Group
Payload	URLPATH <script>alert(12345)</script>
Module	XSS
Time	2024-07-22 17:17:13
ID	d9d2fed482c24aa8a7cd62d655ec7c0c

```
GET /tomcat-docs/appdev/sample/web/hello.jsp?test=<script>alert(12345)</script> HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,br
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36
Host: demo.waf.chaitin.com
Connection: Keep-alive
```

Chapter 06

CONCLUSION

The SQL Injection vulnerability demonstration clearly highlights the significant risks posed by inadequate input validation and insecure query handling in web applications. Through practical exploitation, it was shown that attackers can manipulate vulnerable parameters to gain unauthorized access, extract sensitive data, and potentially compromise the entire database or application environment. The demonstration underscores that SQL Injection remains a critical threat, capable of leading to data breaches, data corruption, and privilege escalation if left unaddressed.

The findings reinforce the urgent need for robust security measures, including the use of prepared statements or parameterized queries, strict input validation, and the principle of least privilege for database access. Regular security assessments, developer education, and timely remediation of discovered vulnerabilities are essential to maintaining application integrity and protecting sensitive information.

Addressing SQL Injection vulnerabilities not only reduces the risk of exploitation but also strengthens the overall security posture of the application. Immediate remediation and ongoing vigilance are crucial to safeguarding systems against this persistent and well-documented threat

Chapter 7

1. FUTURE SCOPE

2. **Integration of Advanced AI Techniques:** The adoption of deep learning models, such as neural networks and ensemble methods, is expected to further enhance prediction accuracy and robustness. These models can capture more complex, nonlinear relationships and interactions among features, leading to more precise valuations.

3. **Incorporation of Diverse Data Sources:** Future models will increasingly leverage a wider array of data, including geospatial information, socio-economic indicators, satellite imagery, and unstructured data from online listings and social media. This integration will enable models to better capture market dynamics and local factors influencing property values.

4. **Real-Time and Dynamic Predictions:** The use of real-time data integration and streaming analytics will allow for dynamic market analysis, enabling stakeholders to receive up-to-date price forecasts that reflect the latest market conditions, economic events, and policy changes.

5. **Explainable and Transparent AI:** There is a growing emphasis on explainable AI methodologies, which will make machine learning models more interpretable and trustworthy for users. This is crucial for adoption in high-stakes domains like real estate, where transparency and fairness are essential.

6. **Personalized and Interactive Tools:** The development of user-friendly interfaces and applications, such as web-based dashboards and mobile apps, will empower users—including buyers, sellers, and investors—to obtain personalized price predictions and insights based on specific property characteristics and preferences.

7. **Enhanced Risk Assessment and Investment Strategies:** Predictive analytics will continue to improve risk assessment and investment planning for real estate professionals and financial institutions, helping them identify undervalued properties, forecast market trends, and optimize portfolios.

Chapter 8

REFERENCE

[1] D. Ristic, "ModSecurity Handbook: Getting Started Guide," Trustwave Holdings, Inc., 2010. [Online]. Available: https://www.modsecurity.org/. [Accessed: Apr. 25, 2024].

[2] M. Hillar, "Cross-Site Scripting Attacks: Xss Exploits and Defense," Indianapolis, IN: Wiley Publishing, 2007.

[3] J. Klein, "Cross-Site Scripting (XSS) Attacks," SANS Institute, 2009. [Online]. Available: https://www.sans.org/reading-room/whitepapers/threats/cross-site-scripting-xss-attacks-2000. [Accessed: Apr. 25, 2024].

[4] OWASP, "OWASP Top Ten," OWASP Foundation, 2021. [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed: Apr. 25, 2024].

[5] J. Stuttard and M. Pinto, "The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws," Indianapolis, IN: Wiley Publishing, 2011.