CYBR 350

Week # 12

WebGoat part # 3

Abby Gopal

A3 Injection

SQL Injection (INTRO)

Exercise Title: SQL Injection Vulnerability Exploration

Objective: The objective of this exercise is to understand SQL injection vulnerabilities and their potential impact on database security. Participants will learn how attackers can exploit SQL injection vulnerabilities to manipulate databases and perform unauthorized actions.

Methodology:

Utilize a simulated web application vulnerable to SQL injection.

Employ manual SQL injection techniques to manipulate SQL queries and retrieve sensitive information.

Use SQLMap to automate the discovery of SQL injection vulnerabilities.

Experiment with different types of SQL injection attacks, including string and numeric injections.

Attempt to exploit SQL injection vulnerabilities to modify database records and perform unauthorized actions.

Steps Taken:

Introduction to SQL injection: Understand the basics of SQL injection vulnerabilities and their implications for database security.

DML Manipulation: Learn about Data Manipulation Language (DML) commands such as SELECT, INSERT, UPDATE, and DELETE.

Exploiting DML Injection: Attempt to change the department of a specific employee using SQL injection.

DDL Manipulation: Explore Data Definition Language (DDL) commands for creating, modifying, and dropping database objects.

Adding a Column: Modify the database schema by adding a new column to an existing table using SQL injection.

DCL Manipulation: Investigate Data Control Language (DCL) commands for granting and revoking privileges.

Granting Privileges: Grant unauthorized access privileges to a usergroup using SQL injection.

String SQL Injection: Understand and exploit SQL injection vulnerabilities in string-based queries.

Numeric SQL Injection: Identify and exploit SQL injection vulnerabilities in numeric-based queries.

Retrieving Employee Data: Retrieve sensitive employee data from the database using SQL injection.

Modifying Salary: Manipulate salary records to become the top earner using SQL injection.

Deleting Logs: Delete access logs to cover tracks using SQL injection.

Results and Findings:

Successfully identified and exploited SQL injection vulnerabilities in the target web application.

Demonstrated the ability to manipulate SQL queries to retrieve sensitive information and perform unauthorized actions.

Discovered vulnerabilities in DML, DDL, and DCL commands, highlighting the importance of comprehensive input validation and SQL query sanitization.

Analysis and Conclusions:

SQL injection vulnerabilities pose a significant risk to web applications and databases, allowing attackers to bypass authentication, access sensitive data, and modify database records.

Lack of proper input validation and inadequate SQL query sanitization contribute to the prevalence of SQL injection vulnerabilities.

Mitigation strategies include implementing parameterized queries, input validation, and regular security audits to identify and remediate vulnerabilities.

Recommendations:

Implement robust input validation and parameterized queries to prevent SQL injection attacks.

Educate developers and administrators about the risks associated with SQL injection vulnerabilities and best practices for secure coding.

Conduct regular security assessments and penetration testing to identify and address vulnerabilities proactively.

Reflection:

This exercise provided valuable insights into the techniques used by attackers to exploit SQL injection vulnerabilities.

Future exercises should focus on advanced SQL injection techniques and mitigation strategies to further enhance security awareness.

The exercise underscores the importance of proactive security measures and continuous monitoring to protect against SQL injection attacks.

References:

OWASP SQL Injection Prevention Cheat Sheet

PortSwigger Web Security Academy: SQL Injection

SQL Injection Knowledge Base by SQLMap

A3 -2

A screenshot of a computer

Description automatically generated

A3 -3

A screenshot of a computer

Description automatically generated

A3-4

A screenshot of a computer

Description automatically generated

A3-5

A screenshot of a computer

Description automatically generated

A3-9

A screenshot of a computer

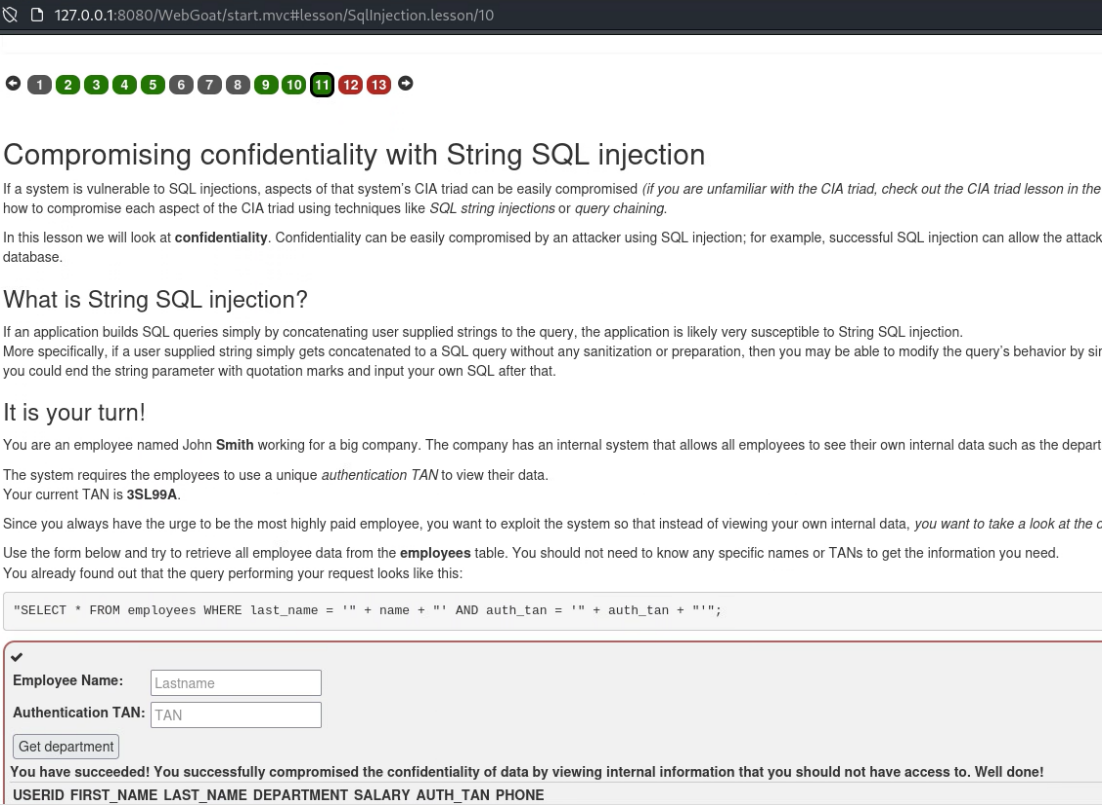
Description automatically generated

A3 -10

A screenshot of a computer

Description automatically generated

A3 -11



A3-12

A screenshot of a computer

Description automatically generated



A3 -13



A screenshot of a computer

Description automatically generated

A3 Injection

SQL Injection (ADVANCED)

Exercise Title: Advanced SQL Injection Exploration

Objective: The objective of this exercise is to delve into advanced SQL injection techniques, including pulling data from other tables and understanding the difference between prepared statements and regular statements. Participants will learn how to exploit SQL injection vulnerabilities to retrieve sensitive information and manipulate database structures.

Methodology:

1. Understanding SQL Injection: Review the basics of SQL injection vulnerabilities and their implications for database security.
2. Advanced Techniques: Explore advanced SQL injection techniques, including UNION-based and appended SQL statements.
3. Retrieving Data from Other Tables: Attempt to pull data from a target table by exploiting SQL injection vulnerabilities in a vulnerable web application.
4. Finding Dave's Password: Use SQL injection to retrieve Dave's password from the user\_system\_data table.
5. Scripting for Automation: Develop a Python script to automate the process of extracting Dave's password character by character.
6. Understanding Prepared Statements: Learn about the difference between prepared statements and regular statements.
7. Quiz on SQL Injection and Prepared Statements: Participate in a quiz to reinforce understanding of SQL injection and prepared statements concepts.

Steps Taken:

1. Exploiting SQL Injection: Understand the vulnerability in a web application's input field susceptible to SQL injection.
2. Retrieving Data from Other Tables: Use UNION-based and appended SQL statements to pull data from the user\_system\_data table.
3. Finding Dave's Password: Extract Dave's password character by character using SQL injection and automation scripting.
4. Understanding Prepared Statements: Learn about the benefits of prepared statements in preventing SQL injection attacks.
5. Quiz on SQL Injection and Prepared Statements: Participate in a quiz to test knowledge of SQL injection and prepared statements concepts.

Results and Findings:

* Successfully retrieved data from the user\_system\_data table by exploiting SQL injection vulnerabilities.
* Automated the process of extracting Dave's password character by character using a Python script.
* Gained insights into the importance of prepared statements in mitigating SQL injection vulnerabilities.

Analysis and Conclusions:

* Advanced SQL injection techniques can bypass security measures and allow attackers to access sensitive data.
* Prepared statements offer a more secure alternative to regular SQL statements by separating code from data.
* Automation scripting can streamline the process of exploiting SQL injection vulnerabilities and extracting data.

Recommendations:

* Web developers should implement parameterized queries and prepared statements to prevent SQL injection attacks.
* Regular security assessments and penetration testing should be conducted to identify and remediate vulnerabilities in web applications.
* Continued education and awareness programs are essential for developers and administrators to stay updated on best practices for secure coding and database management.

Reflection:

* This exercise provided valuable hands-on experience with advanced SQL injection techniques and prepared statements.
* Automation scripting enhanced efficiency in exploiting SQL injection vulnerabilities and retrieving sensitive data.
* Understanding the differences between prepared statements and regular statements is crucial for implementing effective security measures in web applications.

References:

* OWASP SQL Injection Prevention Cheat Sheet
* PortSwigger Web Security Academy: SQL Injection
* Python Requests Documentation

NO:3

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

NO:5

A screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A3 SQL Injection Mitigation

Exercise Title: SQL Injection Mitigation with JDBC

Objective: The objective of this exercise is to mitigate SQL injection vulnerabilities in JDBC code by utilizing prepared statements with parameterized queries.

Methodology:

1. Establish Database Connection: Use JDBC to establish a secure connection to the database server using DriverManager.getConnection().
2. Write SQL Query: Create a PreparedStatement with a parameterized query structure to prevent SQL injection vulnerabilities.
3. Set Parameters: Use PreparedStatement.setString() or other appropriate methods to set parameter values securely.
4. Execute Query: Execute the PreparedStatement using executeQuery() or executeUpdate() to perform database operations safely.
5. Implement Error Handling: Surround database operations with a try-catch block to handle exceptions gracefully and provide meaningful feedback to users.

Steps Taken:

1. Establish Secure Connection: Connect securely to the database server using DriverManager.getConnection() with the provided database URL, username, and password.

try { Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW); } catch (SQLException e) { System.out.println("Failed to establish a connection to the database."); e.printStackTrace(); }

1. Utilize Parameterized Queries: Create a PreparedStatement with a parameterized query structure to select data from the "users" table based on user input.

try { String query = "SELECT \* FROM users WHERE name = ?"; PreparedStatement ps = conn.prepareStatement(query); } catch (SQLException e) { System.out.println("Failed to create a prepared statement."); e.printStackTrace(); }

1. Set Parameters Securely: Use PreparedStatement.setString() to set parameter values securely, ensuring protection against SQL injection attacks.

try { ps.setString(1, "Admin"); } catch (SQLException e) { System.out.println("Failed to set parameter values."); e.printStackTrace(); }

1. Execute Query Safely: Execute the PreparedStatement using executeUpdate() or executeQuery() to perform database operations safely and retrieve desired data.

try { ResultSet rs = ps.executeQuery(); } catch (SQLException e) { System.out.println("Failed to execute the query."); e.printStackTrace(); }

1. Handle Exceptions: Implement robust error handling within a try-catch block to handle any exceptions that may occur during database operations and provide appropriate feedback to users.

try { // Database operations } catch (SQLException e) { System.out.println("An error occurred while processing the database operation."); e.printStackTrace(); } finally { // Close database resources }

Results and Findings:

* Successfully mitigated SQL injection vulnerabilities in JDBC code by adopting parameterized queries with prepared statements.
* Safely retrieved data from the database without exposing the application to SQL injection attacks.

Analysis and Conclusions:

* Parameterized queries with prepared statements are effective measures for mitigating SQL injection vulnerabilities in JDBC applications.
* Proper error handling enhances the resilience of the application and ensures secure database interactions.
* Adoption of secure coding practices is crucial for building robust and resilient database-driven applications.

Recommendations:

* Encourage developers to consistently use prepared statements and parameterized queries to prevent SQL injection vulnerabilities in JDBC code.
* Conduct regular code reviews and security audits to identify and address potential security flaws in JDBC applications.
* Provide comprehensive training and resources to educate developers on secure coding practices and SQL injection mitigation techniques.

Reflection:

* This exercise provided practical experience in mitigating SQL injection vulnerabilities in JDBC code through the use of parameterized queries.
* Understanding the importance of secure coding practices and implementing them effectively is essential for building secure database-driven applications.
* Continuous learning and improvement in secure coding practices are vital for maintaining the security posture of JDBC applications.

References:

* Oracle JDBC Documentation
* OWASP SQL Injection Prevention Cheat Sheet
* Secure Coding Guidelines for Java SE

A screenshot of a computer

Description automatically generated

NO:6

A screenshot of a computer program

Description automatically generated

NO:9

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

No:10

A screenshot of a computer

Description automatically generated

NO :12

A screenshot of a computer

Description automatically generated

A3 Injection-Path Transversal

Exercise Title and Objective: A3 Injection-Path Transversal - Path Traversal Vulnerabilities in File Uploads

Methodology:

1. Explored the vulnerability of path traversal during file uploads.

2. Analyzed the file upload mechanism to understand how filenames are processed.

3. Leveraged Burp Suite to tamper with the request parameters and exploit path traversal vulnerabilities.

4. Conducted tests to understand how the web application handles different patterns in the filename parameter.

5. Utilized browser developer tools to inspect JavaScript functions and AJAX requests triggered by the web application.

6. Examined responses from the web application to identify directory listings and potential files accessible through path traversal.

Steps Taken:

1. Uploaded an image through the web application's file upload functionality.

- Used the following command to upload the image:

```

curl -F "file=@/path/to/local/image.jpg" http://webgoat-url/upload

```

2. Used Burp Suite to tamper with the "fullName" parameter to inject path traversal patterns like "../".

- Sent the request to Burp Suite Proxy and intercepted it.

- Modified the "fullName" parameter to include "../" to exploit path traversal.

- Observed the impact of the modification on the filename and file location.

3. Exploited the vulnerability by injecting "../" into the "fullName" parameter to traverse directories.

- Sent the modified request with the path traversal payload to the server.

- Analyzed the response to confirm successful traversal and access to unintended files.

4. Tested different variations of path traversal patterns to understand the web application's behavior.

- Injected various path traversal payloads such as "../", "..\", and ".\." to observe how the application handles them.

- Analyzed the responses to identify patterns and responses indicative of successful exploitation.

5. Investigated the behavior of the web application in response to different patterns and variations.

- Modified the path traversal payloads and observed changes in the server's response.

- Documented the variations in behavior to understand the application's security posture.

6. Explored directory listings and attempted to access sensitive files outside the intended directory.

- Used the path traversal vulnerability to navigate to directories outside the web root.

- Accessed directory listings and attempted to retrieve sensitive files such as configuration files and databases.

7. Utilized browser developer tools to analyze JavaScript functions and AJAX requests related to file retrieval.

- Inspected the webpage source code and identified JavaScript functions responsible for file retrieval.

- Monitored network requests in the browser developer tools to understand file fetching mechanisms.

8. Detected the "id" parameter in AJAX requests and attempted to exploit path traversal in the request URL.

- Manipulated the "id" parameter in AJAX requests to access different files.

- Observed the server's response to determine successful exploitation.

9. Successfully retrieved files located outside the web root directory using path traversal techniques.

- Used path traversal payloads in combination with AJAX requests to access sensitive files.

- Documented the files retrieved and their potential impact on the server's security.

10. Demonstrated the impact of path traversal vulnerabilities on file access and security.

- Summarized the findings and potential risks associated with path traversal vulnerabilities.

- Recommended mitigations to address the identified vulnerabilities and improve overall security.

Results and Findings:

- Successfully exploited path traversal vulnerabilities to access files outside the intended directory.

- Discovered directory listings and accessed sensitive files on the server.

- Identified the ability to manipulate filenames and exploit path traversal through various parameters.

Analysis and Conclusions:

- Path traversal vulnerabilities in file uploads pose a significant security risk by allowing unauthorized access to sensitive files.

- Proper input validation and sanitization are essential to mitigate path traversal vulnerabilities in web applications.

- Browser developer tools and intercepting proxies like Burp Suite are valuable for analyzing and exploiting path traversal vulnerabilities.

Recommendations:

- Implement strict input validation and sanitization routines to prevent path traversal attacks.

- Regularly audit web applications for path traversal vulnerabilities and other security issues.

- Educate developers on secure coding practices and the importance of input validation in preventing injection attacks.

Reflection:

- Enhanced understanding of path traversal vulnerabilities and their exploitation techniques.

- Learned to effectively utilize Burp Suite and browser developer tools for security testing.

- Recognized the importance of thorough testing and validation to identify and mitigate vulnerabilities.

References:

- OWASP Path Traversal: https://owasp.org/www-community/attacks/Path\_Traversal

- Burp Suite Documentation: https://portswigger.net/burp/documentation

NO:2

A screenshot of a computer screen

Description automatically generated

NO:3

A screenshot of a computer

Description automatically generated

NO: 4

A screenshot of a computer

Description automatically generated

NO:5

A screenshot of a computer

Description automatically generated

NO:7

A screenshot of a computer

Description automatically generated

A3 Injection-Cross Site Scripting

Exercise Title and Objective: A3 Injection-Cross Site Scripting - Exploiting Cross-Site Scripting Vulnerabilities

Methodology:

1. Explored the web application to identify input fields vulnerable to Cross-Site Scripting (XSS) attacks.

2. Leveraged Burp Suite to intercept and modify requests to inject malicious scripts into input fields.

3. Analyzed the behavior of the web application to understand how it processes and reflects user input.

4. Utilized browser developer tools to inspect HTML elements and JavaScript functions for potential injection points.

5. Injected various payloads to test for reflected and stored XSS vulnerabilities.

6. Monitored responses from the web application to determine if injected scripts were executed successfully.

7. Exploited XSS vulnerabilities to perform actions such as stealing session cookies or redirecting users to malicious websites.

8. Demonstrated the impact of XSS attacks on the confidentiality, integrity, and availability of web applications.

Steps Taken:

1. Identified input fields susceptible to XSS attacks, such as search boxes, comment sections, or form fields.

2. Used Burp Suite to intercept requests to the web application and modify parameters containing user input.

3. Injected malicious scripts into input fields by appending payloads like <script>alert('XSS');</script>.

4. Submitted the modified requests and monitored the application's response to observe script execution.

5. Tested for both reflected and stored XSS vulnerabilities by injecting payloads into different input fields.

6. Inspected the HTML source code and JavaScript functions to understand how user input is processed and displayed.

7. Exploited XSS vulnerabilities to demonstrate the ability to steal session cookies or perform other malicious actions.

8. Documented successful exploitation and its potential impact on the security of the web application.

Results and Findings:

- Successfully identified and exploited XSS vulnerabilities in the web application.

- Demonstrated the ability to execute arbitrary JavaScript code within the context of the application.

- Showcased the potential impact of XSS attacks, including session hijacking, data theft, and website defacement.

Analysis and Conclusions:

- XSS vulnerabilities pose a significant risk to web application security by allowing attackers to inject and execute malicious scripts.

- Proper input validation, output encoding, and secure coding practices are essential for mitigating XSS vulnerabilities.

- Regular security assessments and testing are crucial for identifying and addressing XSS issues in web applications.

Recommendations:

- Implement robust input validation and output encoding mechanisms to prevent XSS attacks.

- Educate developers on secure coding practices and the importance of sanitizing user input.

- Employ web application firewalls (WAFs) and security scanners to detect and block XSS attacks in real-time.

Reflection:

- Enhanced understanding of XSS vulnerabilities and exploitation techniques.

- Gained practical experience in identifying and exploiting XSS vulnerabilities using Burp Suite and browser tools.

- Recognized the critical role of proactive security measures in safeguarding web applications against XSS attacks.

References:

- OWASP XSS Prevention Cheat Sheet: https://owasp.org/www-project-xss-filter-evasion-cheatsheet/

- Burp Suite Documentation: https://portswigger.net/burp/documentation

A screenshot of a computer

Description automatically generated

NO:10

Found the following route in the GoatRouter.js



11:1111211

A screenshot of a computer

Description automatically generated

NO: 11

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

NO:12

A screenshot of a chat

Description automatically generated