Lab 8

Venkata Diwakar Reddy Kashireddy

***Abstract*—This report outlines methods to mitigate SQL injection attacks and performed tasks on Command injection and Log spoofing. A SQL injection is a technique that attackers use to gain unauthorized access to a web application database.**

I. Introduction

In this lab, we delved into SQL mitigation techniques to counter SQL injection. We executed tasks tied to SQL injection and previously explored the subject using WebGoat 8.2.2 and WebGoat 7.1. For context, WebGoat is an intentionally vulnerable web platform by OWASP, intended for web security instruction.

II. TOOLS

• KVM (Kernel-based Virtual Machine): It is a full virtualization solution for Linux. [1]
• WebGoat: A deliberately insecure web application maintained by OWASP designed for teaching web application security concepts. [2]
• Overleaf: It is a collaborative cloud-based LaTeX editor that helps to create documents easily by providing standard formats. [3]
• GitHub: GitHub is an Internet hosting service for software development and version control using Git. [4]
• Red: – IU Research Desktop (RED) is a virtual desktop service for users with accounts on the Carbonate research supercomputer at IU. [5]
• ZAP: OWASP ZAP (Zed Attack Proxy) is an open-source web application security scanner. It is used by those new to application security and professional penetration testers. [6]
• Firefox DevTools: Firefox Developer Tools is a set of web developer tools built into Firefox. It can be accessed to inspect the web page. [7]

III. Injection Mitigation

To defend against SQL injection, it's essential to handle data in a way where it's not misinterpreted. Opt for methods that treat the data as a unified piece, linked to a column without misreading its intent. We can deploy static, parameterized, and stored queries to thwart these attacks. In our recent lesson on Parameterized Queries, I executed task 5 by crafting a secure code. This code aimed to fetch a user's status using their name and email. I began by initiating a connection and formulating a statement. The task was accomplished using terms like getConnection, PreparedStatement, prepareStatement, ?, ?, setString, and setString.

In the subsequent task, I utilized JDBC to establish a connection to a database and retrieve data. I encapsulated the connection within a try-catch for error handling and crafted a query that's safeguarded against SQL injection threats.

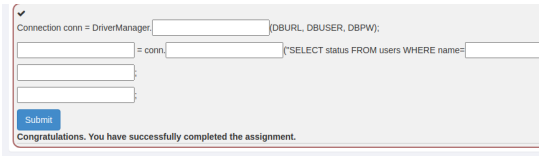

Fig. 1. Safe code using immutable queries

Following this, I executed the query. To pass the string, I employed the 'setString' method. The 'setString(int parameterIndex, String x)' function assigns the specified parameter with the provided Java String value.

Fig. 2. JDBC connectivity

IV. Input Validation

Web developers can prevent SQL injection by using parameterized queries, which serve as a robust line of defense. Parameterized queries can effectively mitigate SQL injection because they ensure that user input is always treated as data and not executable code. However, it's essential to note that parameterized queries might not be sufficient in cases where stored procedures are vulnerable to injection or when the database itself has inherent vulnerabilities. In such scenarios, a multi-layered security approach is recommended. For the subsequent task, spaces in the Name field were not allowed. Hence, I replaced spaces with comments(**). Using the comments, I formulated the query below to get the data.

```
a';/**/select/**/*/**/from/**/user_system_data;--
```

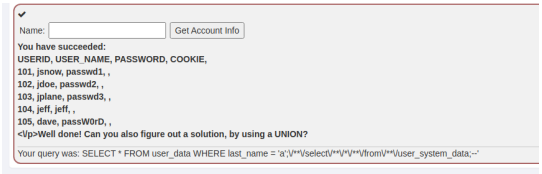

Fig. 3. SQL injection with input validation against spaces

In my subsequent task, I noticed that the form checks for spaces and specific keywords like 'SELECT' and 'FORM'.

To bypass this, I layered the words ’Select’ and ’Form’ inside similar words, resulting in ’seselectlect’ and ’frfromom’. The end input appeared as: a’;//seselectlect//*//frfromom//user system data;–.

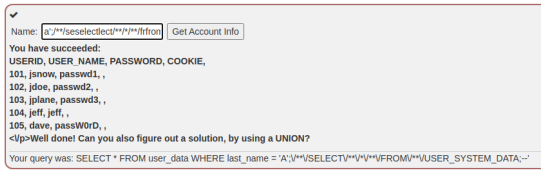

Fig. 4. SQL injection with input validation against Select and From keywords

V. Command Injection

Command injection is a type of attack where the primary objective is to execute commands on a host system via a susceptible application. Such attacks occur when an app processes insecure user data, like from forms or HTTP headers, directly into a system shell. In our lesson, the primary point of interaction is a dropdown menu. This dropdown accesses a user-selected help file. However, this alone doesn’t prevent command injection. To demonstrate this vulnerability, I modified the HTML code with developer tools, inputting ”AccessControlMatrix.help”;ls”. This action executed the ’ls’ system command within the app. As a result, the webpage displayed the ’ls’ command output.

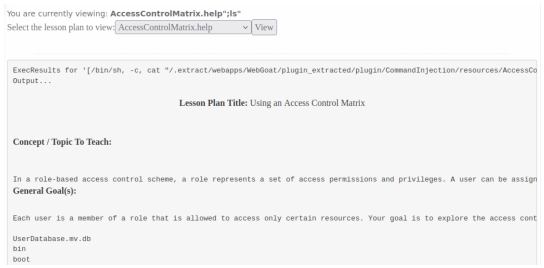

Fig. 5. Executing a system command through a drop down option.

VI. Log Spoofing

Log Spoofing involves the theft of passwords and personal details via a compromised login page. In our exercise, a designated grey area indicated the content to be logged on the web server’s file. The objective was to simulate a successful login by the ”admin” user. I input ”diwakar%0d%0aLogin succeeded for username admin” as the username. Here, %0d and %0a are URL encodings for CR (Carriage Return) and LF (Line Feed), respectively, signifying a new line. This input enabled me to execute the Log Spoofing.

In a subsequent task, the challenge was to intensify the attack by embedding a script into the log. I achieved this by inputting ”diwakar%0d%0aLogin succeeded for username admin”, which triggered an alert for Diwakar on the site.

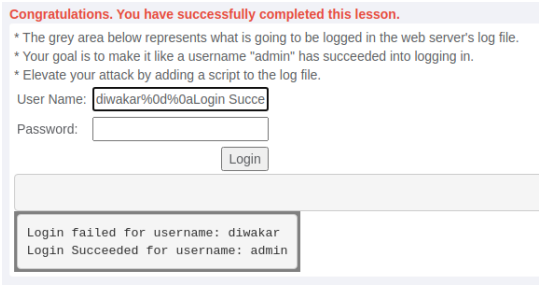

Fig. 6. Successfully performed Log Spoofing.

VII. Conclusion

SQL injection and Command injection attacks represent a serious threat to any database-driven site. While the methods behind these attacks are straightforward, their effects can range from significant disruptions to complete system compromises. Notably, Blind SQL injections, which query databases with true-false questions, present a unique challenge. However, relying solely on prepared statements or input validation isn’t foolproof. For parameter-driven sites, it’s imperative to sanitize all input, especially when used in OS commands, scripts, and database queries. With some foresight, many of these threats can be effectively mitigated.

References

[1] KVM
https://www.linux-kvm.org/page/MainPage..
[2] WebGoat
https://owasp.org/www-project-webgoat/..
[3] OverLeaf
https://www.overleaf.com/.
[4] Github
https://en.wikipedia.org/wiki/GitHub.
[5] RED
https://kb.iu.edu/d/apum.
[6] OWASP ZAP
https://www.zaproxy.org/
[7] Firefox DevTools
https://firefox-source-docs.mozilla.org/devtools-user/
[8] Information Schema Tables
https://www.mssqltips.com/sqlservertutorial/196/
information-schema-tables/.