Lab 8

Chetan Sahrudhai Kimidi - ckimidi

***Abstract*—A lab exercise consisting of SQL Injection Mitigation and injection flaws such as Command Injection and Log Spoofing.**

I. Introduction

SQL Injection Mitigation deals with fixing the SQL injection flaws or vulnerabilities in an application/code. In this lab, I worked on and solved the following: SQL Injection Mitigation in WebGoat 8, and Command Injection and Log Spoofing in WebGoat 7.1.

II. SQL Injection Mitigation: Steps 1-10

Lesson 1 consisted of information about types of queries, such as parameterized, static, immutable etc. Lesson 2 elaborated about the difference between safe and injectable stored procedures. Lessons 3 and 4 showed Java code, which described the scenario of mitigation.
In Lesson 5, I needed to fill in the code snippets, which were the key to implementing the mitigation. I completed it using Lessons 3 and 4, as seen in Fig. 1. In Lesson 6, I wrote the code from scratch, which can be seen in Fig. 2. In Lesson 9, the developer fixed the injection with filtering, but I was supposed to spot the weakness after the fix. I observed that using inline comments, [1], starting and ending them together at once, can be used as a substitute for a space. I solved this as seen in Fig. 3.
In Lesson 10, the developers further deepened the strictness and also filtered out keywords, as part of the input validation this time, such as from, select etc. so as to avoid their usage in the input. But, the weakness in this approach, which I observed is, the keyword elimination can give rise to the combination of two different parts of input, which itself can be a keyword. I solved it using the same ideology and a similarly developed malicious input, as seen in Fig. 4.

How can parameterized queries mitigate SQL injection?
Query Parameterization (Prepared Statement) is a technique to protect against SQL injection attacks. It works by separating the SQL statement from the user input values. The SQL statement is compiled once and then executed multiple times with different parameter values. This prevents attackers from injecting malicious SQL code into the query.

How cannot parameterized queries mitigate SQL injection?
If the parameterized queries are not properly constructed (such as usage of any operators like concatenate etc.) or if the database is specifically vulnerable to a SQL injection attack (boolean or time-based etc.), then the flaw might still be exploited.

How can a web developer prevent SQL injection?
A web developer can prevent SQLi, by -
Using prepared statements (parameterized queries).
Validating all user input, which is checking all user input for malicious characters and rejecting any invalid input, using regular expressions and similar measures.
Other methods are to encode input before query usage, using a firewall etc.

III. Injection Flaws - WebGoat 7.1

A. Command Injection

In this challenge, there is a drop-down menu of various files, which are some of the lessons present in the WebGoat 7.1. I have to exploit the command injection in this, by searching for the flaw. First, I observed that the result/view depends upon the selection in the drop menu. So, I proposed that if I am able to inject some command into the input in the drop-down menu, and select 'View', that would be given to the system as input. The command which is currently running might be a simple cat command, since it is just displaying the selected lesson. So, I could use query chaining, using ;, and inject an additional command into the operating system. To achieve this, I initially thought of using ZAP and intercepting the input, but then, I thought of simply altering the script, using the Inspect functionality of the browser. I brought up Inspect, using F12, hovered on the drop-down menu, found out the corresponding HTML code, which had the list of inputs in the menu. I modified the first input, AccessControlMatrix, using query chaining, and then clicked on 'View'. In this way, I successfully injected another command (whoami) into the OS, as seen in Fig. 5 and Fig. 6.

B. Log Spoofing

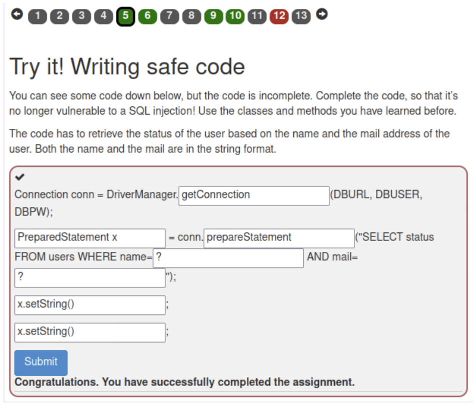
In this task, there were two fields, username and password. The challenge was to make it seem like a user called 'admin' logged in successfully. So, to make it seem like that, I could just print it, using the username field. But, the default case was the output of 'Login failed for username: '. In general, log files contain lines separated by carriage returns, for which the URL Encoding is %0d%0a. So, I appended the respective URL encoding, along with some text, which stated that admin logged in successfully. This can be seen in Fig. 7.

IV. Conclusion

In summary, I successfully completed Command Injection and Log Spoofing in WebGoat 7.1, and SQL Injection Mitigation, WebGoat 8, first 10 steps.

References

[1] Bypass Mechanisms
[2] Prepared Statements



Try it! Writing safe code

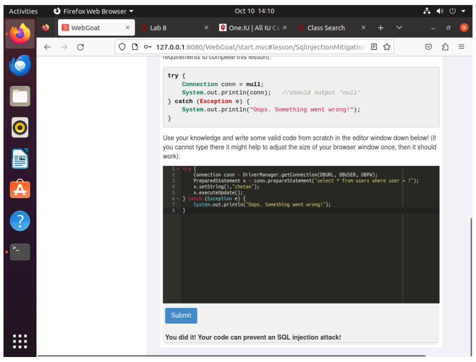
You can see some code down below, but the code is incomplete. Complete the code, so that it's no longer vulnerable to a SQL injection! Use the classes and methods you have learned before.

The code has to retrieve the status of the user based on the name and the mail address of the user. Both the name and the mail are in the string format.

Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW);
PreparedStatement x = conn.prepareStatement("SELECT status FROM users WHERE name=? AND mail=?");
x.setString();
x.setString();

Congratulations. You have successfully completed the assignment.

Fig. 1. Lesson 5



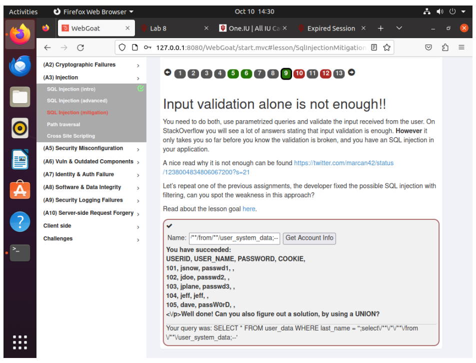
```
try {
    Connection conn = null;
    System.out.println(conn);   //should output 'null'
} catch (Exception e) {
    System.out.println("Oops. Something went wrong!");
}
```

Use your knowledge and write some valid code from scratch in the editor window down below! (if you cannot type there it might help to adjust the size of your browser window once, then it should work):

```
try {
    Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW);
    PreparedStatement x = conn.prepareStatement("select * from users where user = ?");
    x.setString(1,"chetan");
    x.executeUpdate();
} catch (Exception e) {
    System.out.println("Oops. Something went wrong!");
}
```

You did it! Your code can prevent an SQL injection attack!

Fig. 2. Lesson 6


Activities
Firefox Web Browser
Oct 10 14:30
WebGoat
Lab 8
One.IU | All IU Ca
Expired Session
127.0.0.1:8080/WebGoat/start.mvc#lesson/SqlInjectionMitigation
(A2) Cryptographic Failures
(A3) Injection
SQL Injection (intro)
SQL Injection (advanced)
SQL Injection (mitigation)
Path traversal
Cross Site Scripting
(A5) Security Misconfiguration
(A6) Vuln & Outdated Components
(A7) Identity & Auth Failure
(A8) Software & Data Integrity
(A9) Security Logging Failures
(A10) Server-side Request Forgery
Client side
Challenges
1 2 3 4 5 6 7 8 9 10 11 12 13
Get Account Info

Input validation alone is not enough!!

You need to do both, use parametrized queries and validate the input received from the user. On StackOverflow you will see a lot of answers stating that input validation is enough. **However** it only takes you so far before you know the validation is broken, and you have an SQL injection in your application.

A nice read why it is not enough can be found https://twitter.com/marcan42/status/1238004834806067200?s=21

Let's repeat one of the previous assignments, the developer fixed the possible SQL injection with filtering, can you spot the weakness in this approach?

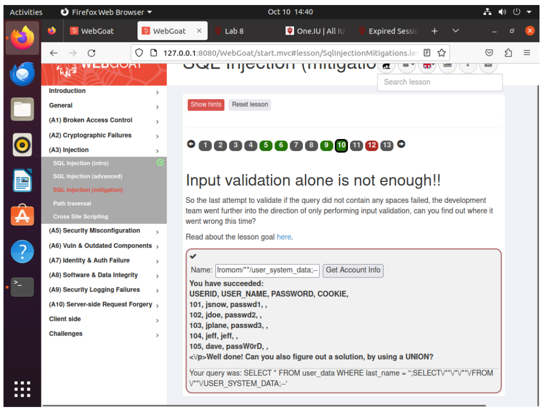
Read about the lesson goal here.

Name: /**/from/**/user_system_data;--

You have succeeded:
USERID, USER_NAME, PASSWORD, COOKIE,
101, jsnow, passwd1, ,
102, jdoe, passwd2, ,
103, jplane, passwd3, ,
104, jeff, jeff, ,
105, dave, passW0rD, ,
<\/p>Well done! Can you also figure out a solution, by using a UNION?

Your query was: SELECT * FROM user_data WHERE last_name = '';select\/**\/*\/**\/from \/**\/user_system_data;--'

Fig. 3. Lesson 9


Activities
Firefox Web Browser
Oct 10 14:40
WebGoat
WebGoat
Lab 8
One.IU | All IU
Expired Sessio
127.0.0.1:8080/WebGoat/start.mvc#lesson/SqlInjectionMitigations.les
SQL Injection (mitigatio
Search lesson
Show hints
Reset lesson
Introduction
General
(A1) Broken Access Control
(A2) Cryptographic Failures
(A3) Injection
SQL Injection (intro)
SQL Injection (advanced)
SQL Injection (mitigation)
Path traversal
Cross Site Scripting
(A5) Security Misconfiguration
(A6) Vuln & Outdated Components
(A7) Identity & Auth Failure
(A8) Software & Data Integrity
(A9) Security Logging Failures
(A10) Server-side Request Forgery
Client side
Challenges
1 2 3 4 5 6 7 8 9 10 11 12 13
Get Account Info

Input validation alone is not enough!!

So the last attempt to validate if the query did not contain any spaces failed, the development team went further into the direction of only performing input validation, can you find out where it went wrong this time?

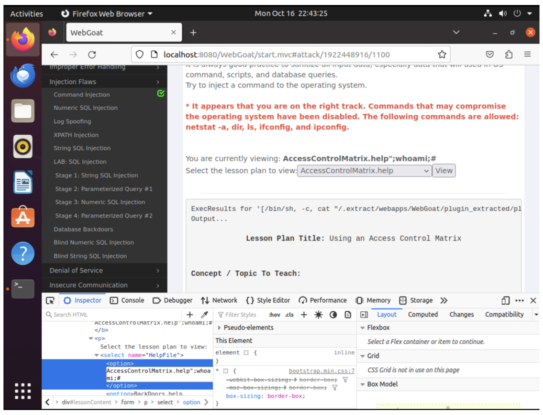
Read about the lesson goal here.

Name: fromom/**/user_system_data;--

You have succeeded:
USERID, USER_NAME, PASSWORD, COOKIE,
101, jsnow, passwd1, ,
102, jdoe, passwd2, ,
103, jplane, passwd3, ,
104, jeff, jeff, ,
105, dave, passW0rD, ,
<\/p>Well done! Can you also figure out a solution, by using a UNION?

Your query was: SELECT * FROM user_data WHERE last_name = '';SELECT\/**\/*\/**\/FROM \/**\/USER_SYSTEM_DATA;--'

Fig. 4. Lesson 10



command, scripts, and database queries.
Try to inject a command to the operating system.

* It appears that you are on the right track. Commands that may compromise the operating system have been disabled. The following commands are allowed: netstat -a, dir, ls, ifconfig, and ipconfig.

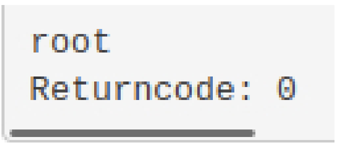
You are currently viewing: **AccessControlMatrix.help";whoami;#**
Select the lesson plan to view: AccessControlMatrix.help

```
ExecResults for '[/bin/sh, -c, cat "/.extract/webapps/WebGoat/plugin_extracted/pl
Output...

        Lesson Plan Title: Using an Access Control Matrix


Concept / Topic To Teach:
```

```
<p>
  Select the lesson plan to view:
  <select name="HelpFile">
    <option>
    AccessControlMatrix.help";whoami;#
    </option>
    <option>BackDoors.help
```

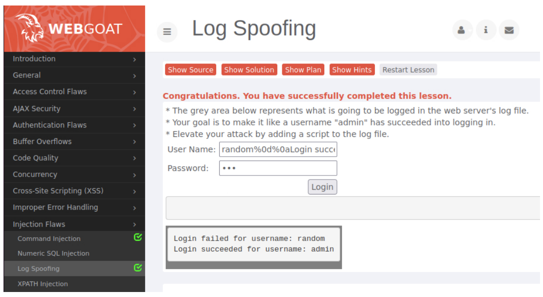
```
element {                     inline
}
* {                    bootstrap.min.css:7
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
```

Fig. 5. Command Injection



```
root
Returncode: 0
```

Fig. 6. Output of injected command - whoami



Congratulations. You have successfully completed this lesson.

* The grey area below represents what is going to be logged in the web server's log file.
* Your goal is to make it like a username "admin" has succeeded into logging in.
* Elevate your attack by adding a script to the log file.

User Name: random%0d%0aLogin succ
Password: •••

```
Login failed for username: random
Login succeeded for username: admin
```

Fig. 7. Log Spoofing