WEB SECURITY

COMPUTER SECURITY I
DVGC19

22 DECEMBER 2023

AHMAD SHAMMOUT : SHMOOT001@GMAIL.COM
JOHANNA OLSSON : JOHANNAMARIA1@LIVE.SE

1. Performed Tasks

1.1 Authentication Flaws

Lesson 1 : Secure Passwords
During the initial session, we opted for a robust password selection approach, ensuring a secure configuration. Our chosen password included a combination of at least one uppercase letter, one lowercase letter, one number, and one special character. The aim was to create a password with a minimum length of 12 characters. As illustrated in Figure 1, the selected password, "dvGc19_lABB-tr3!," achieved a perfect score of 4/4, and the estimated number of guesses required to crack the password was notably high.

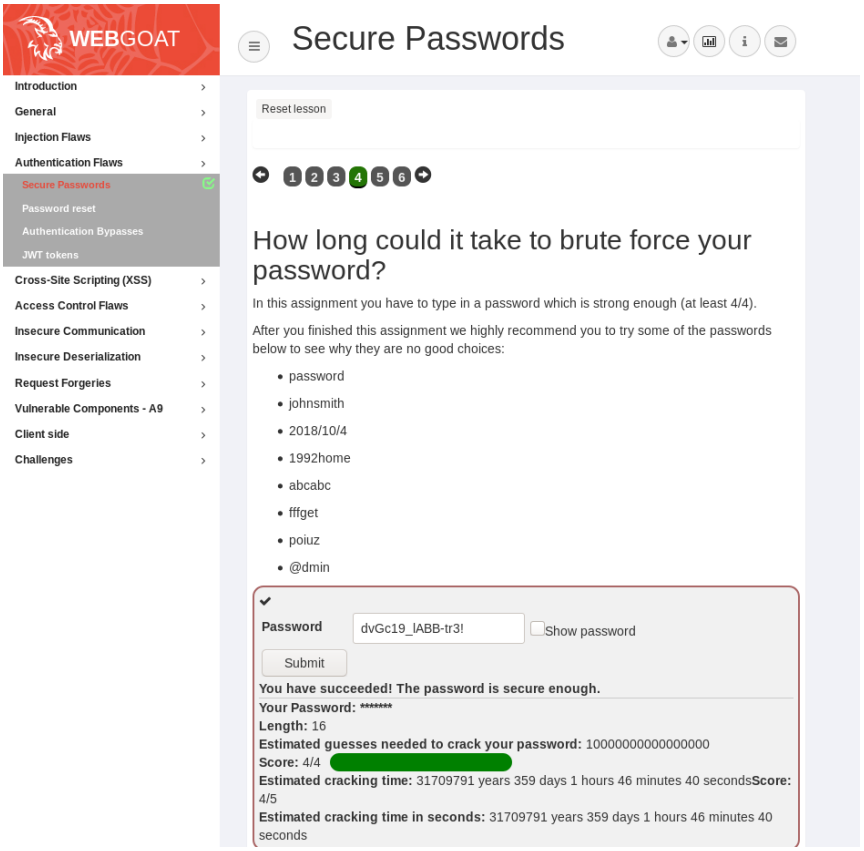

Figure 1 : Choosing a password that passes the challenge.

Lesson 2 : Authentication Bypasses

In this challenge, the user is required to reset their password. The username/email information has already been supplied. The user has opted for the alternative verification method and must respond to two security questions to authenticate and verify access to the account.

To successfully navigate the authentication mechanism, we employed the BurpSuite proxy tool to intercept the HTTP request. To get around the authentication mechanism, it is necessary to adjust the security parameters within the BurpSuite tool and subsequently forward the modified post request to the server. The figure below illustrates the two security parameters post modification.

The following illustration (Figure 2) depicts the BurpSuite intercept window, revealing all the post data acquired, which includes information related to “secQuestion0” and “secQuestion1”. What we did to pass this challenge was to change “secQuestion0” and “secQuestion1” to “secQuestionA” and “secQuestionB”.

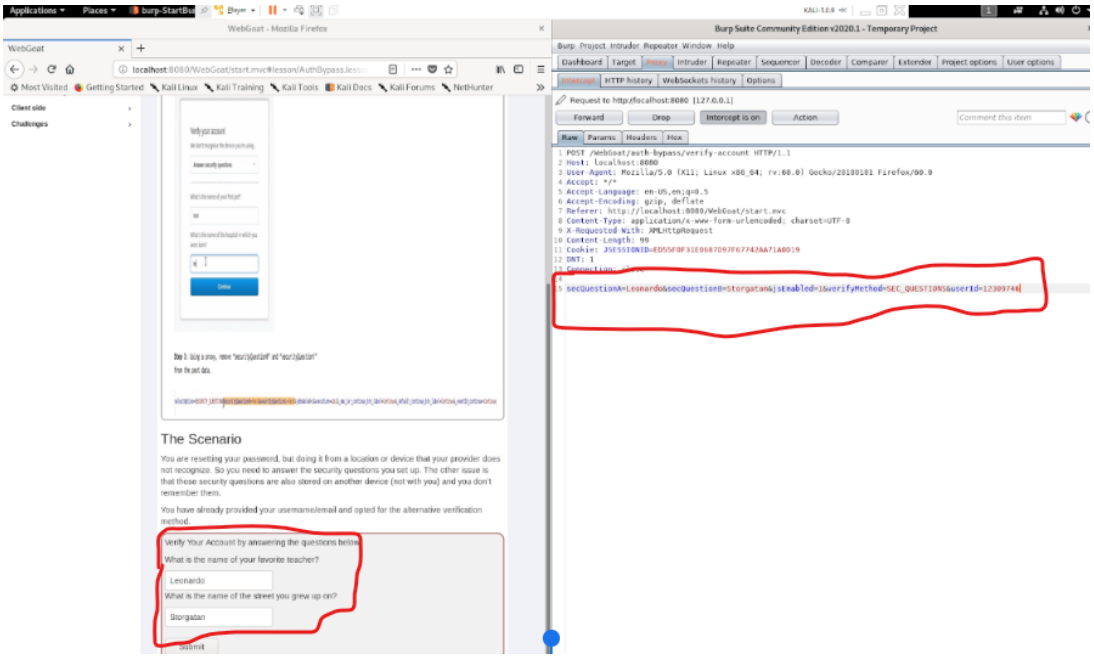

Figure 2 : Showing the edit that has been done in BurpSuite Proxy.

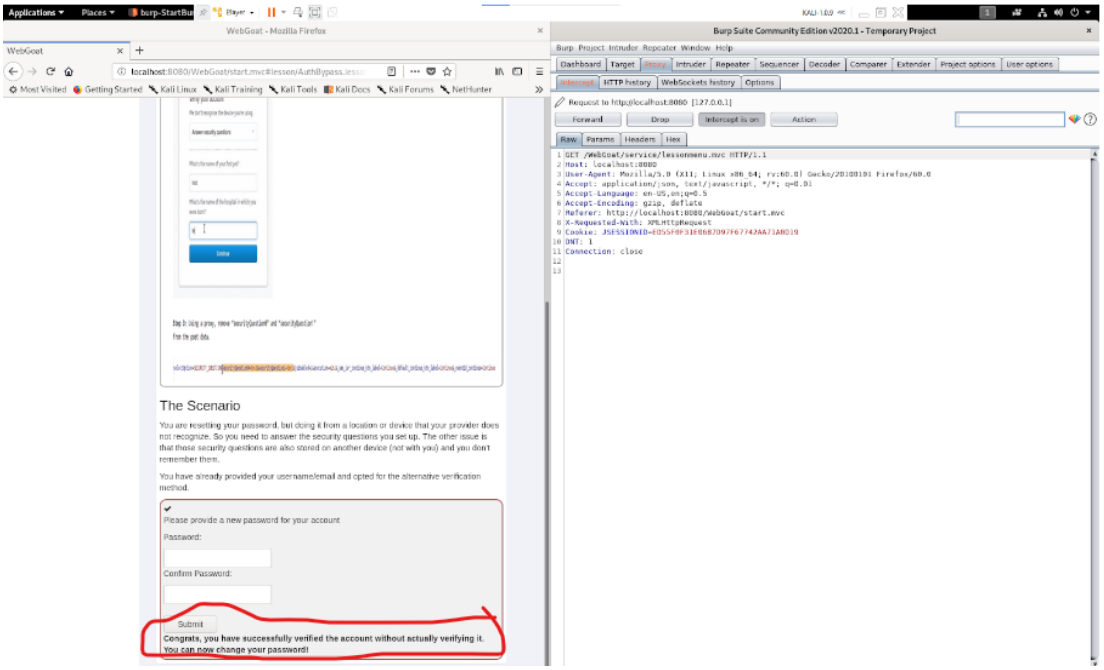

Figure 3 : Showing the challenge has been done.

1.2 Injection Flaws

1.2.1 SQL injection: introduction

Challenge 2 : What is SQL
In this challenge we tried to retrieve the department of the employee Bob Franco, to complete this task we used a Sql query to achieve the result, the using SQL query was :
SELECT department FROM Employees WHERE first_name = 'Bob'
AND last_name = 'Franco';

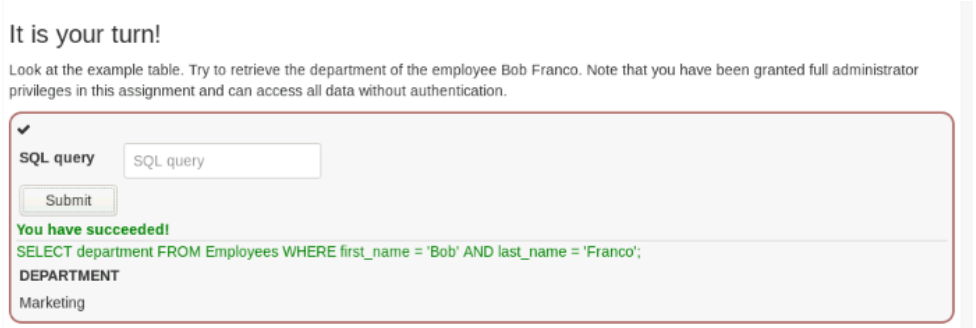

Figure 4 : Showing the result when we applied our SQL query.

Challenge 3 : Data Manipulation Language (DML)
In this challenge we had to change the department of Tobi Barnett to “Sales”, and to complete this task we had to use a new SQL query to achieve the expected results.
The SQL query that has been used:
UPDATE Employees SET department = 'Sales' WHERE first_name = 'Tobi'
AND last_name = 'Barnett';

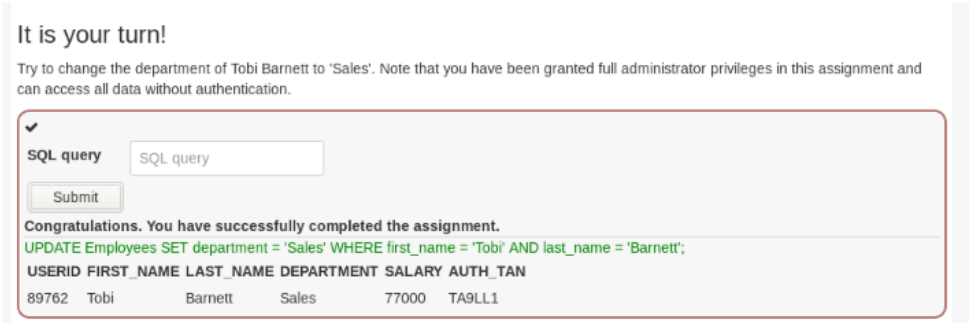

Figure 5 : Showing that we succeed with this task.

Challenge 4 : Data Definition Language (DDL)
In this task we used to modify the scheme by adding the column “phone”(varchar(20)) to the table “employees”, to complete this task we used this SQL query:
ALTER TABLE employees ADD COLUMN phone varchar(20);

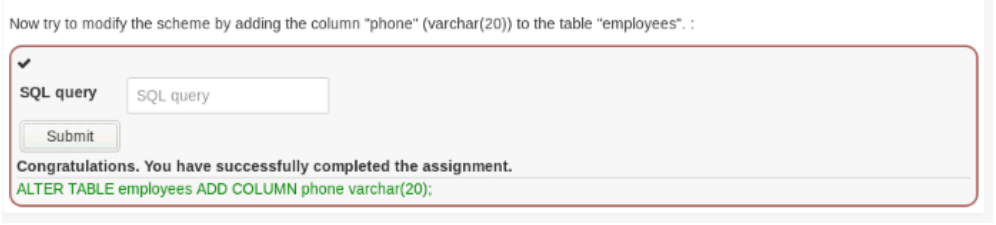

Figure 6 : Showing that we succeed with this task.

Challenge 5 : Data Control Language (DCL)
In this challenge we had to try to grant the usergroup “UnauthorizedUser” the right to alter tables.
We used this SQL query to complete the task :
GRANT ALTER TABLE
TO UnauthorizedUser

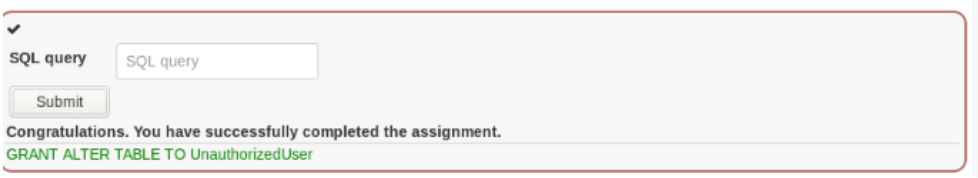

Figure 7 : Showing that we succeed with this task.

First lesson:
Challenge 11 : String SQL Injection
In this task, the objective was to exploit a string SQL injection vulnerability to access all employee data in the employees table. String SQL injection occurs when an application forms SQL queries by combining user-provided strings directly into the query. If the provided string is incorporated into the query without proper sanitization or preparation, we can manipulate the query's behavior by introducing quotation marks into the input field.

To complete this task we used this SQL query :
Employee Name : A
Authentication TAN : ‘OR’1=’1

“ ‘OR’1=’1 ” is an example of a SQL injection attempt. It aims to manipulate the logic of a SQL query's WHERE clause. The condition `'1' = '1'` is always true, allowing the injection to potentially bypass security measures and retrieve unauthorized data. SQL injection is a serious security issue, and developers should use techniques like parameterized queries to prevent such attacks.

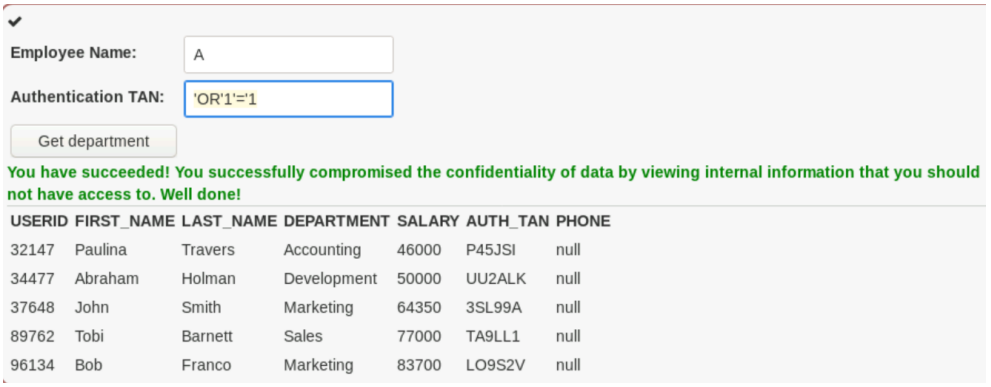

Figure 8: Showing that we succeed with this task.

Second lesson:
Challenge 12 : Query Chaining
In challenge 12, the task involved employing SQL query chaining to alter the database and update the salary of the employee John Smith. SQL query chaining is a method that enables the addition of one or more queries to the conclusion of the initial query. This technique proves advantageous in seamlessly applying additional queries without initiating a new line. SQL query chaining is achieved by appending the semicolon mark (';') at the end of the query.

To complete this task we used this SQL query :
Employee Name : A
Authentication TAN : ‘;UPDATE employees SET salary =85000 WHERE auth_tan =’3SL99A

This SQL query begins with a single quote, to close a string in the original query. The semicolon that follows is an attempt to terminate the original query. The subsequent `UPDATE` statement introduces a new command to update the `salary` column. The `SET salary = 85000` part specifies the new value for the salary. Finally, the `WHERE auth_tan = '3SL99A'` condition limits the update to the employee with the authentication token '3SL99A' which is John Smith.

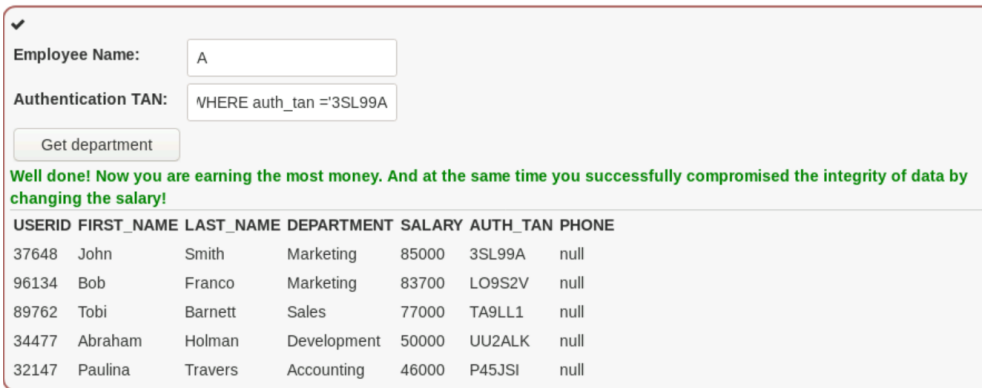

Figure 9 : Showing that we succeed with this task.

Challenge 13 : Query Chaining
In this challenge we were asked to remove all the logs so nobody notice the changes.
We used this SQL query to drop all the logs in the access log table:
%’;DROP TABLE access_log;--

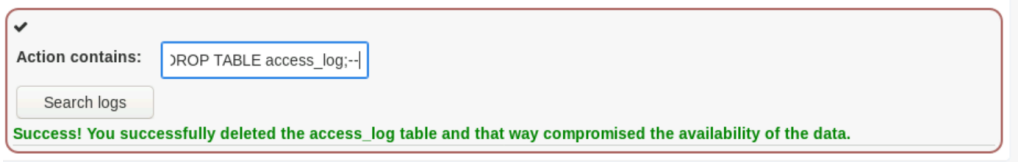

Figure 10 : Showing that we succeed with this task.

1.2.2 SQL injection: advanced First lesson

Challenge 5 : Blind SQL Injection
In this particular task, our objective was to engage with a blind SQL injection. In essence, blind SQL injection denotes a form of SQL attack wherein the assailant does not directly receive feedback from the server regarding the success or failure of a query. This adds complexity to the attack, necessitating alternative methods for the attacker to ascertain whether their queries are achieving success. For instance, the attacker might craft a query inducing a noticeable alteration in the website's behavior, such as a delay in loading a specific page, to infer the query's success.

During this exercise, we were assigned the task of discovering the password for a user named Tom. The webpage featured both a login form and a registration form, prompting us to investigate and identify which form field was susceptible to SQL injection. Following extensive testing, we determined that the vulnerability lay in the username field of the registration form. Figure 11 illustrates this vulnerability, demonstrating that altering the parameter still confirmed the existence of the user "tom" being already registered.

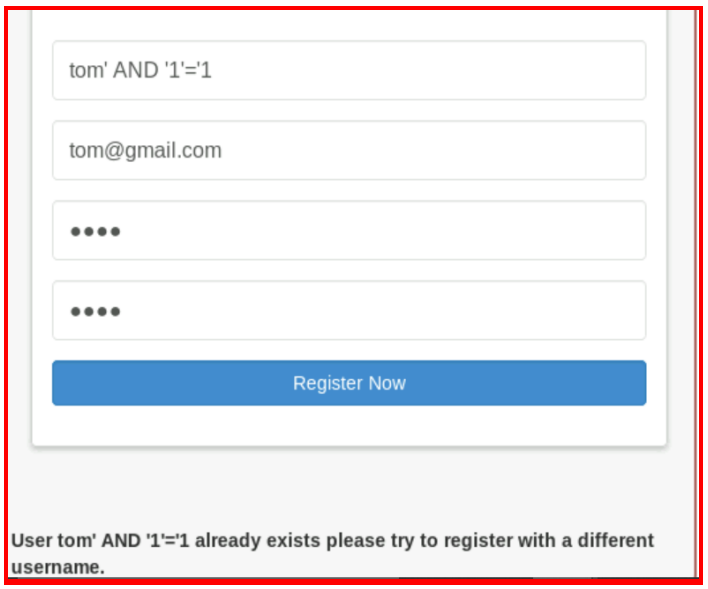

Figure 11 : showing that the user tom is already exist.

The subsequent task involves capturing a registration PUT request through the utilization of a proxy tool, such as Burp Suite. The proxy tool has been configured in alignment with the guidelines outlined in the lab sheet to ensure its proper functionality. This captured registration PUT request will be preserved and subsequently employed to streamline the execution of the SQL injection attack, leveraging the capabilities of the sqlmap tool.

Sqlmap stands as a free and open-source tool that plays a pivotal role in both detecting and exploiting SQL injection vulnerabilities. Operating primarily through a command-line interface, sqlmap facilitates the automation of processes involved in identifying and capitalizing on SQL injection vulnerabilities. Its utility extends to testing the security parameters of web applications and databases, offering a versatile toolset capable of executing various types of attacks.

In essence, this integrated approach, involving the use of Burp Suite for request capturing and sqlmap for automated SQL injection, streamlines the testing and exploitation phases, enhancing efficiency and precision in evaluating the security posture of web applications and databases.

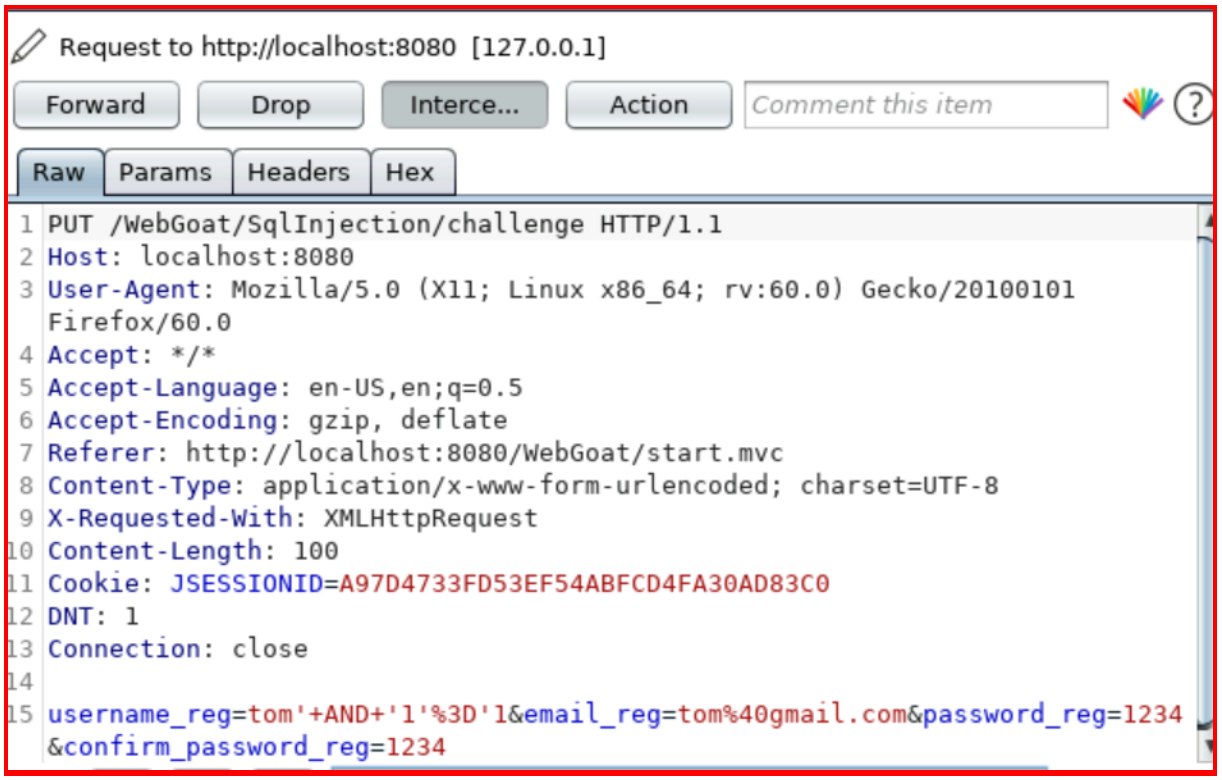

Figure 12 : Saving the login request from Burp Suite proxy.

We are now ready to initiate a scan on the targeted request using the sqlmap tool. The primary objective of the initial scan is to pinpoint any vulnerabilities existing on the server. The command employed for this purpose is as follows: “sqlmap -r login.req” where the '-r' flag designates the loading of an HTTP request from a file.

The ensuing scan results, depicted in the figure below, reveal that the PUT parameter is susceptible to a boolean-based blind attack, specifically the "username_reg" vulnerability. Subsequently, the scan successfully identifies the back-end database as "HSQLDB" Having ascertained the database, the scan prompts us to determine whether we wish to continue probing for additional vulnerabilities.

It's noteworthy that declining the previous prompts would not impact the results, as the server, in this instance, lacks additional vulnerabilities. Finally, the scan inquires if we intend to persist in testing the server for vulnerabilities beyond "username_reg," to which we respond negatively, concluding this phase of the assessment.

Figure 13: Scanning the login request using SQLMAP.

Transitioning to the subsequent scanning phase, we focus on the previously identified vulnerable test parameter, namely "username_reg".

The command employed for this stage is:
sqlmap -r login.req -p username_reg -v 1 --dbs

- The `-p` flag is utilized to furnish the test parameter, which, in this instance, is "username_reg."
- The `-v` flag specifies the verbosity level of the scan, and in this case, it is set to 1, providing a moderate level of detail without being overly verbose.
- The `--dbs` flag is invoked to systematically enumerate the databases associated with the Database Management System (DBMS) on the server.

In essence, this command orchestrates a focused SQL injection scan, centering on the susceptibility identified in the form of the "username_reg" parameter. The verbosity level is adjusted to a moderate setting (1), and the tool is directed to enumerate the databases present in the Database Management System (DBMS) on the server using the `--dbs` flag. This process aims to unveil the database landscape for further evaluation or exploitation.

Figure 14

With our knowledge of the existing databases, we progress to the subsequent phase: enumerating the tables within a selected database. To initiate this process, we must first choose a specific database. After evaluating the options, we determine that PUBLIC is the most promising choice, prompting us to commence the scan on this database.

The scan is executed using the following command:
sqlmap -r login.req -p username_reg -v 1 -D PUBLIC --tables

- The `-D` flag is employed to specify the database to be enumerated, in this case, it's set to PUBLIC.
- The `--tables` flag instructs sqlmap to systematically enumerate the tables within the specified database.

In essence, this command facilitates the targeted exploration of tables within the chosen database (PUBLIC). The verbosity level is set to 1 for a balanced level of detail, and the `--tables` flag directs sqlmap to enumerate the tables present in the designated database. This step is crucial for uncovering the structure and content within the selected database for further analysis or exploitation.

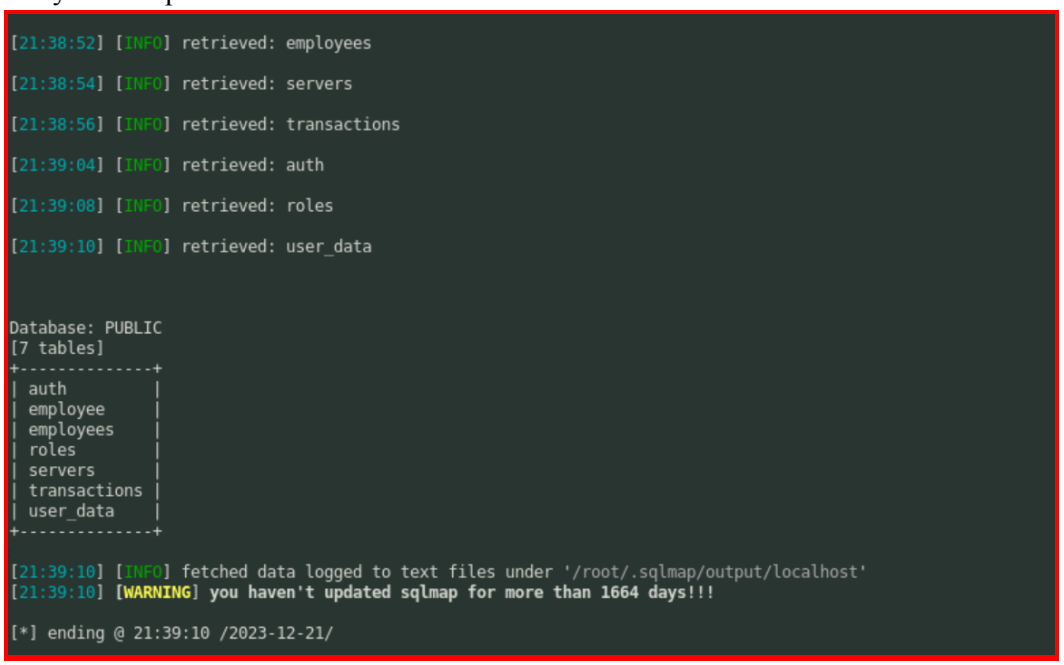

Figure 15

Concluding the process of enumerating through the selected database involves extracting data from the tables. This is achieved through the command:
sqlmap -r login.req -p username_reg -v 1 -D PUBLIC -T EMPLOYEES -C email,password --dump

- The `-T` flag is employed to designate the table of interest, which, in this case, is EMPLOYEES.
- The `-C` flag allows the selection of specific columns within the table to enumerate through. Here, the chosen columns are email and password.
- The `--dump` flag instructs sqlmap to extract and present the entries stored within the specified database tables.

This command facilitates the extraction of data from the EMPLOYEES table within the PUBLIC database. The selected columns (email and password) are enumerated, and the `--dump` flag ensures the retrieval and display of entries from the designated database tables. After scanning we see that we got the expected results, and got the password for the user.

```
[21:45:48] [INFO] fetching entries of column(s) 'email, password' for table 'EMPLOYEES' in database 'PUBLIC'
[21:45:48] [INFO] fetching number of column(s) 'email, password' entries for table 'EMPLOYEES' in database 'PUBLIC'
[21:45:48] [INFO] resumed: 5
[21:45:48] [INFO] resumed: tom@webgoat.org
[21:45:48] [INFO] resumed: thisisasecretfortomonly
[21:45:48] [INFO] resumed: tom@webgoat.org
[21:45:48] [INFO] resumed: thisisasecretfortomonly
[21:45:48] [INFO] resumed: tom@webgoat.org
[21:45:48] [INFO] resumed: thisisasecretfortomonly
[21:45:48] [INFO] resumed: tom@webgoat.org
[21:45:48] [INFO] resumed: thisisasecretfortomonly
[21:45:48] [INFO] resumed: tom@webgoat.org
[21:45:48] [INFO] resumed: thisisasecretfortomonly
Database: PUBLIC
Table: EMPLOYEES
[5 entries]
+-----------------+-------------------------+
| email           | password                |
+-----------------+-------------------------+
| tom@webgoat.org | thisisasecretfortomonly |
| tom@webgoat.org | thisisasecretfortomonly |
| tom@webgoat.org | thisisasecretfortomonly |
| tom@webgoat.org | thisisasecretfortomonly |
| tom@webgoat.org | thisisasecretfortomonly |
+-----------------+-------------------------+

[21:45:48] [INFO] table 'PUBLIC.EMPLOYEES' dumped to CSV file '/root/.sqlmap/output/localhost/dump/PUBLIC/EMPLOYEES.csv'
[21:45:48] [INFO] fetched data logged to text files under '/root/.sqlmap/output/localhost'
[21:45:48] [WARNING] you haven't updated sqlmap for more than 1664 days!!!

[*] ending @ 21:45:48 /2023-12-21/
```

Figure 16

Now we are going to test this password, and we see the figure below that it is working.

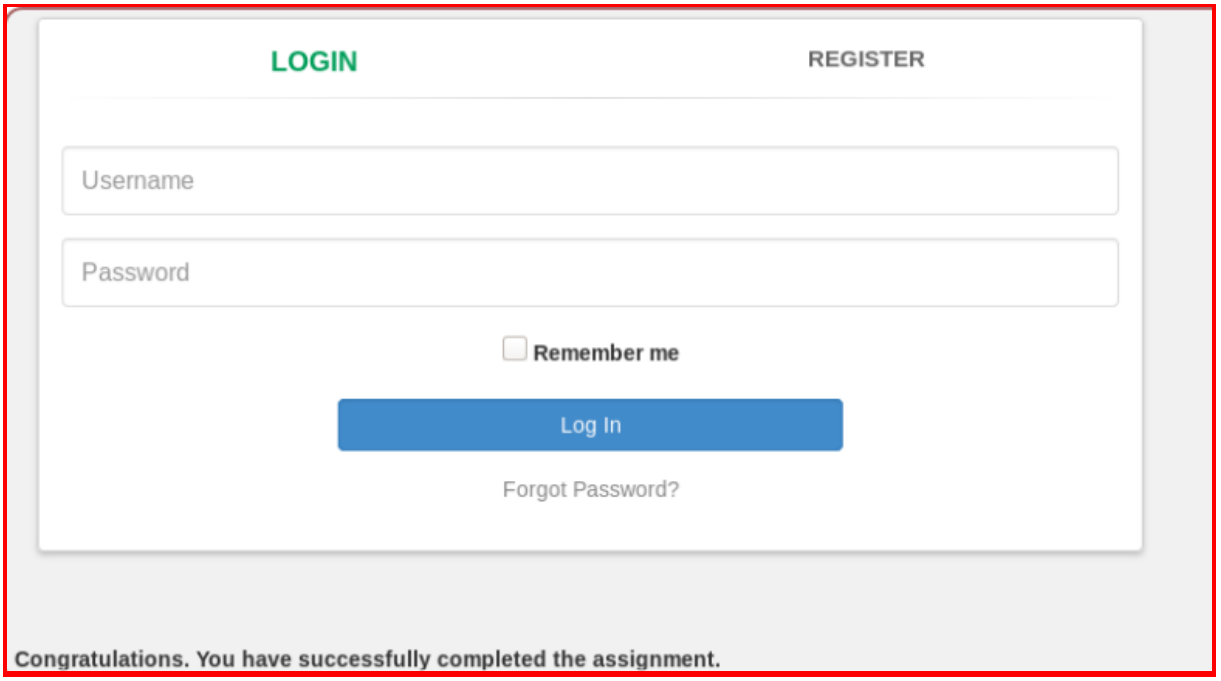

Figure 17

1.3 Cross Site Scripting

Challenge 7 : Reflected XSS

For the Cross Site Scripting tasks, the first performed challenge was challenge 7 from the lesson “Reflected XSS”. Reflected XSS is a web vulnerability where untrusted data is echoed back to the user in a web application’s response. When the user interacts with this malicious input, the server reflects and executes the embedded script within their session. This type of attack relies on tricking the user to initiate the harmful request and then potentially compromising sensitive data or enabling unauthorized access.

In this challenge we got to try and test some input fields and see if they were vulnerable to reflected XSS. For the test we passed a javascript to the input parameter of the credit card number field. This script was *<script>alert(1)</script>*, and it proceeded to present us a pop up window with the alert message “1”.

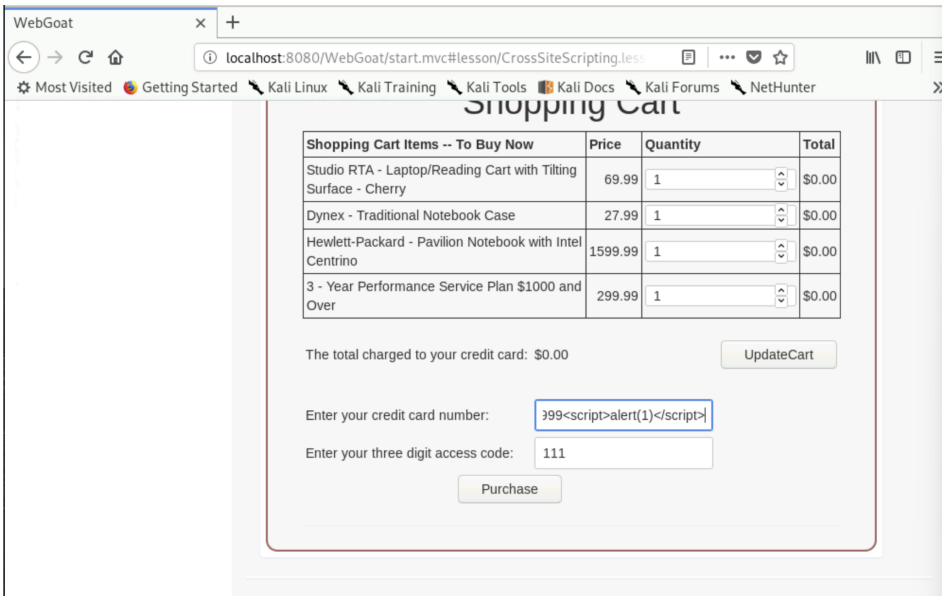

Figure 18 : Script input in the credit card number field

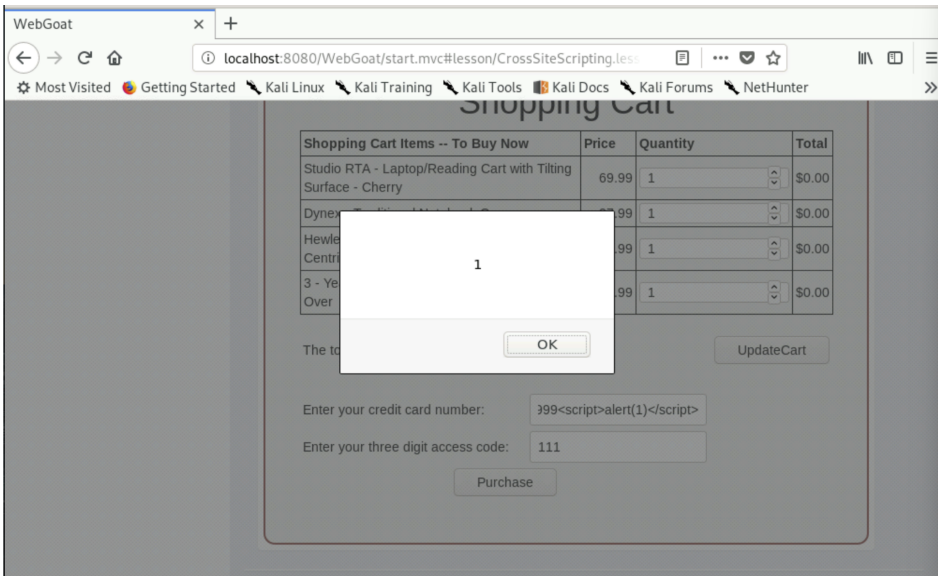

Figure 19 : Pop-up window showing the vulnerability, the result of our reflected XSS test

Challenge 3 : Stored XSS

The second task was to perform challenge 3 from the lesson “Stored XSS”. Stored XSS is a web security vulnerability where the attacker injects a malicious script into a web application’s database. The malicious script is then retrieved and executed when users access the compromised data. Different to reflected XSS, stored XSS persists beyond the current session which poses a greater threat. Stored XSS is mostly injected through input fields or user-generated content, with the goal to compromise user accounts, steal sensitive information, or perform unauthorized actions.

For this challenge we checked if the comment field was vulnerable to stored XSS. The first input in the text field to try this was *<script>alert(“Comment”)</script>*.

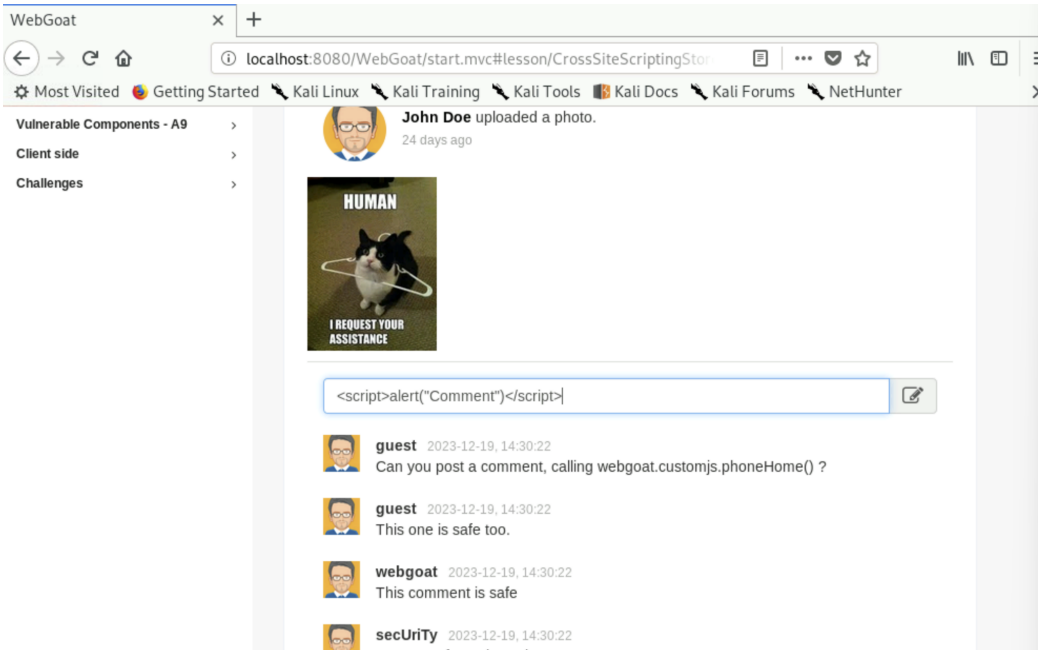

Figure 20: Input script to make a malicious comment

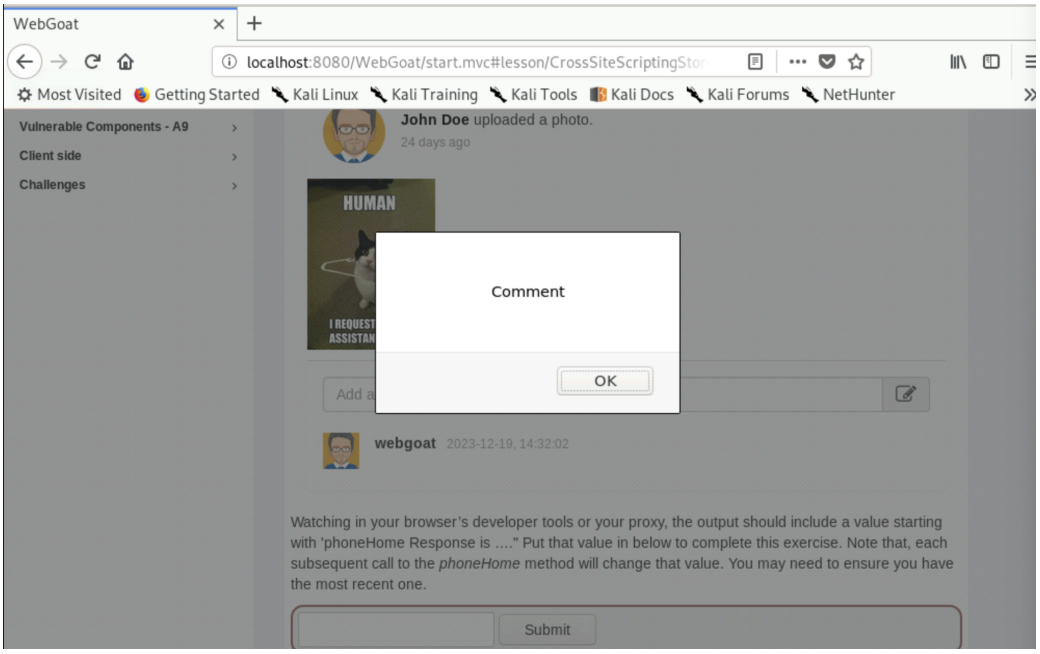

Figure 21 : Generated a pop-up window as response to the script, showing vulnerability

The result of the test was that the input field was vulnerable to stored XSS, since it generated a pop-up window with the input script, and this pop-up continued to show every time the page was reloaded as well.

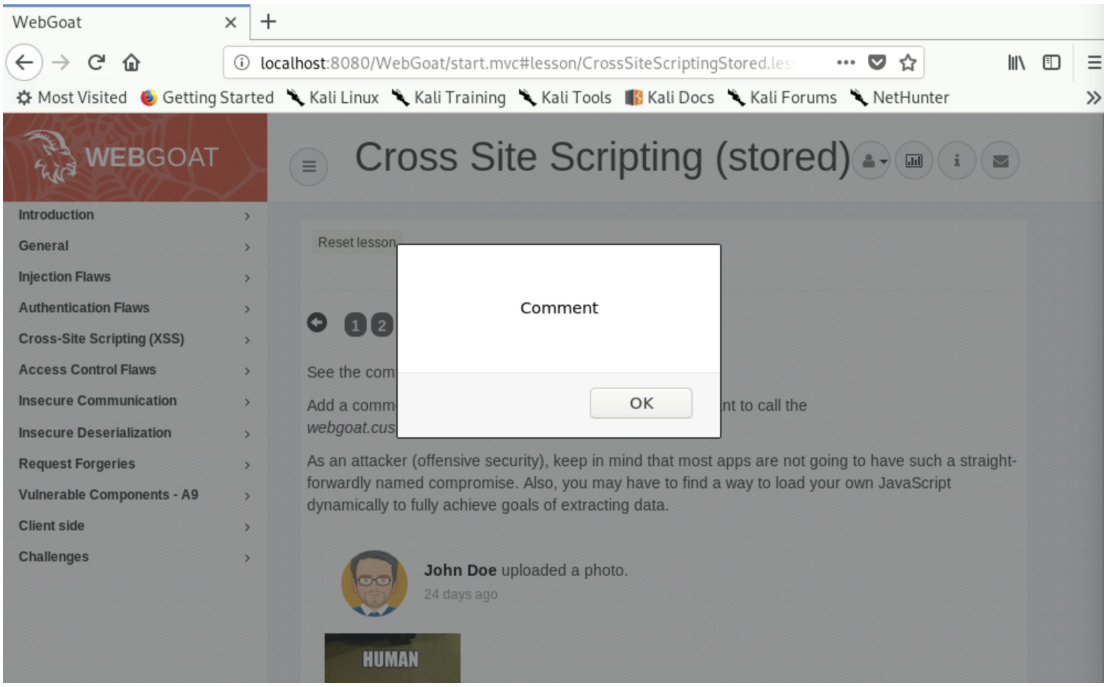

Figure 22 : After refreshing page, pop-up window is shown again

We also decided to try the proposed function *webgoat.customjs.phoneHome* to see what type of response we would receive.

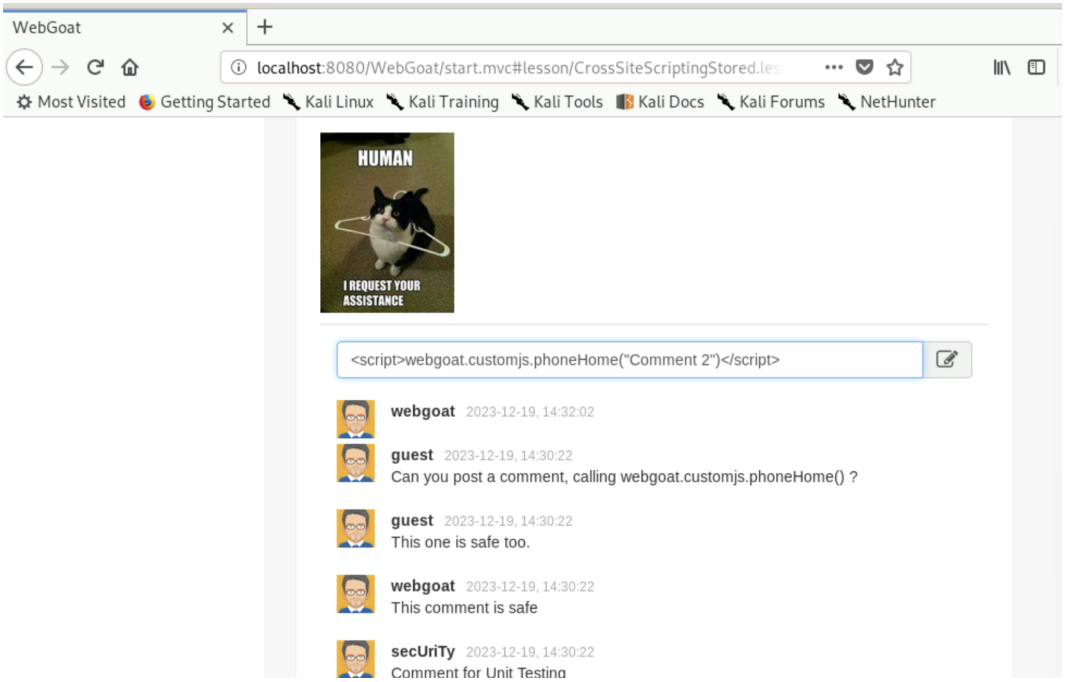

Figure 23 : The suggested function for the challenge

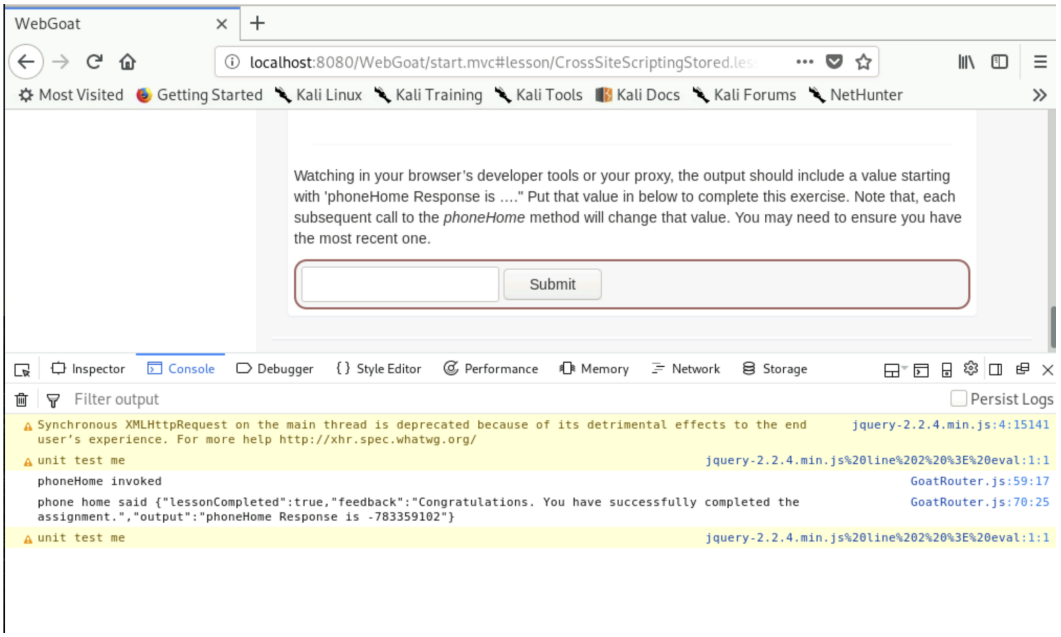

Figure 24 : Using the inspect tool to get the response, showing lesson completed

For XSS vulnerabilities, there are some counter measurements that can be taken to avoid these malicious attacks. Some examples are:

1. **Web Application Firewalls**: Used to monitor and filter HTTP traffic between a web application and the internet, helping to detect and block malicious activities such as XSS attacks.
2. **Input Validation**: Implementing a strict input validation on both the client and server sides ensures that user input adheres to expected formats. For this you would reject or sanitize any input that does not meet the specified criteria.
3. **Encoding Output**: Encoding user-generated content before rendering it in the browser converts potentially harmful characters into their HTML or URL-encoded equivalents, which prevents the browser from interpreting them as executable code.
4. **HTYP-Only Cookies**: Setting the "HttpOnly" attribute on cookies prevents them from being accessed through client-side scripts which reduces the risk of session theft through XSS attacks.

2. Discussion: Injection Vulnerabilities

Injection vulnerabilities pose significant threats to databases and web applications. For this reason it is essential to have robust countermeasures prepared for ensuring system security. The most effective way to mitigate these risks is by employing a multi-layered approach.

One important countermeasure for injection vulnerabilities is using parameterized queries and prepared statements. When separating the user inputs from the execution logic we prevent malicious code injection into SQL queries. Here, input validation on both the client and server sides also plays an important role, ensuring that only expected and sanitized data enters the system which reduces the likelihood of injection attacks.

Some complementary strategies for these vulnerabilities are escaping and encoding, these strategies add an extra layer of defense. Escaping involves placing escape characters before certain characters to neutralize any special meaning. Encoding is when you transform potentially dangerous characters into their harmless equivalents. Combined, these techniques can help safeguard against a range of injection vulnerabilities, such as Cross-Site Scripting.

3. Possible mistakes

While conducting this experiment we faced some challenges on the way, for example navigating with the BurpSuit proxy and trying to figure out how to solve some of the tasks. By researching a bit about every subject, we managed to solve all of the tasks without any major problems. Therefore we believe that no mistakes were made that affected our results in a negative way.