Cross-Site Request Forgery Attack

CS8.403.S22.System and Network Security

Maturi Sai Shushma (2021201012)
Malla Hemalata (2021201066)

CSRF:

Cross-Site Request Forgery (CSRF) is the vulnerability in which the attacker forges a request to the server via an authorized user of the server.

Here, when a user accesses the server and requests any action, the server will check if the user is logged in or not. If the user is authorized then if the server performs the action requested by none or solely using the user's cookies then the server is vulnerable to CSRF attack. The attacker if somehow makes the user click the malicious link, the same action request is sent to the server leading to the attack.

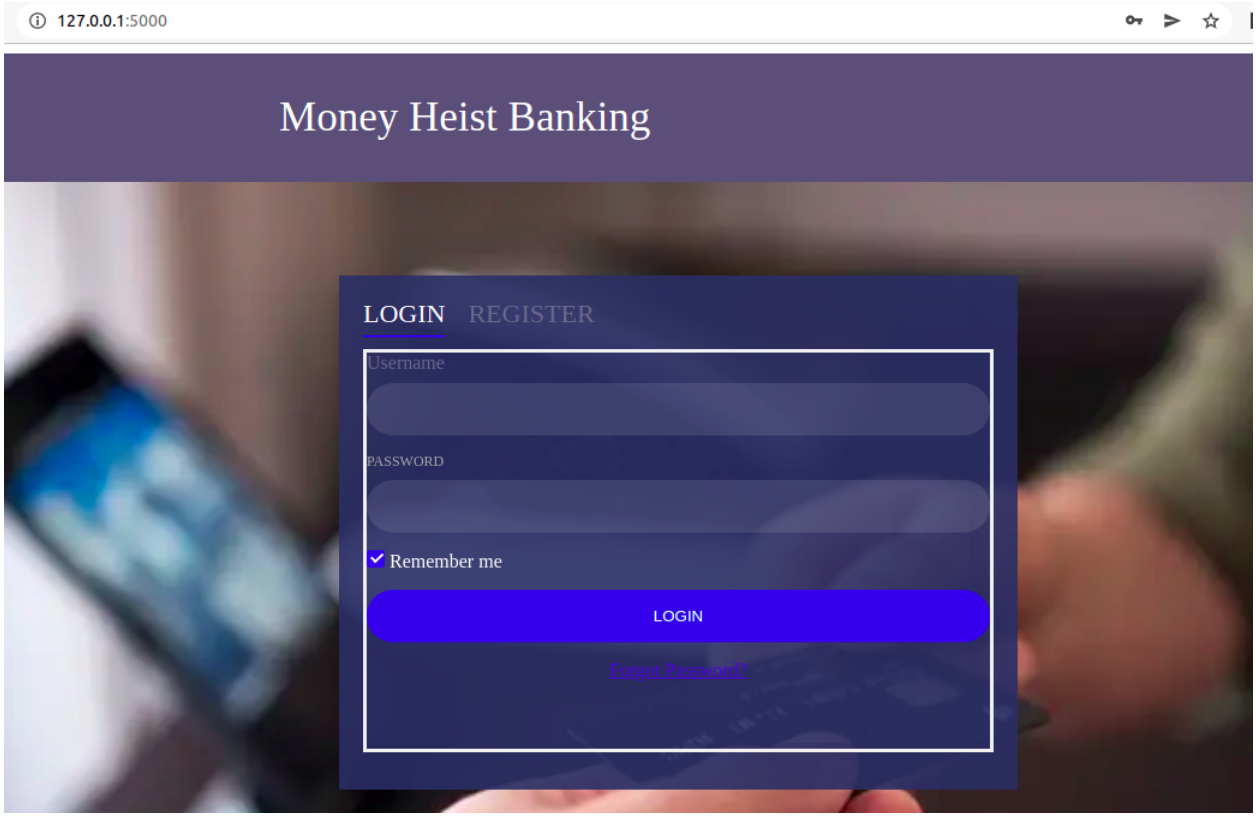
Our Goal:

Our main idea here is first we have a banking website. Users have to register and attackers also have to register in the same bank to show the amount changed after transactions. After creating the accounts, the user has to login to his account and try to send some money or just to check his balance. Now the user should not log out from his account but opens another tab and opens the link we have sent. Links can be sent using mail or any other forms. After clicking the link we have some interesting content and click the button to watch more content. If the user clicks the button the amount from the user account will be directly sent to the attacker account. Users have no idea about this. User can only know if he checks the balance again.

To implement the CSRF attack we used python, HTML, and flask. First start the python program using the common python3 CSRF.py. After that it will show the program address as shown in the below diagram. If we paste that link, a banking website will be opened, from where we have to perform all actions.

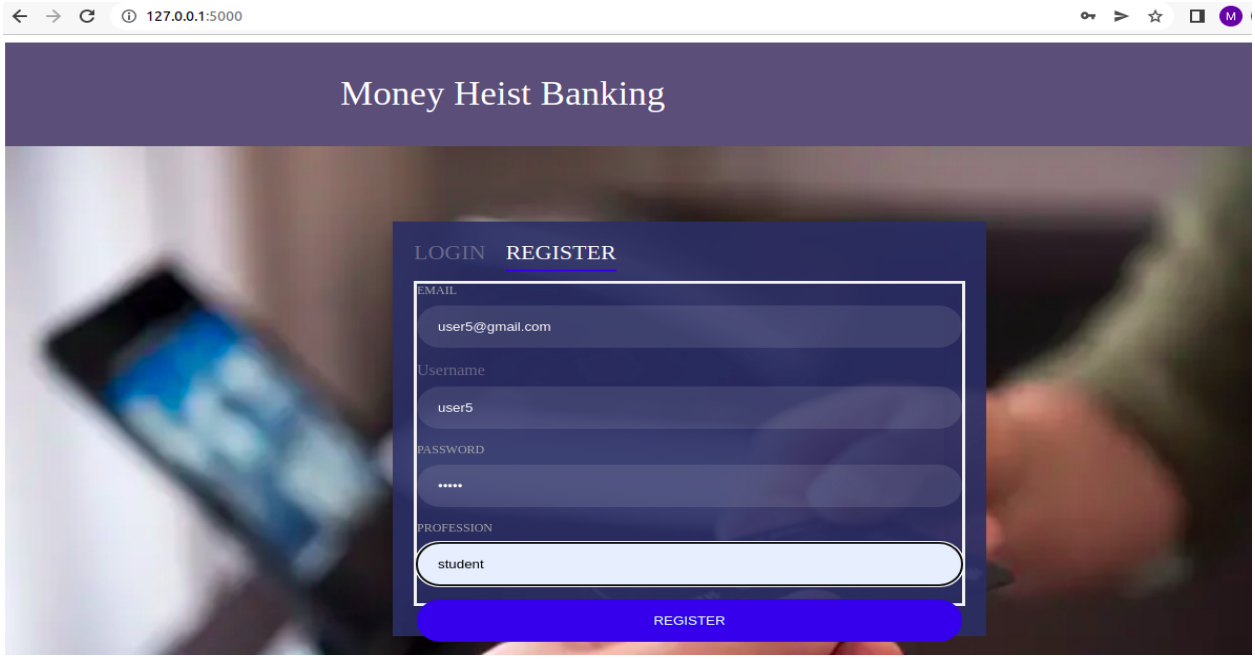
```
lata@lata-Vostro-3559:~/Pictures/final_csrf$ cd ./Attack
lata@lata-Vostro-3559:~/Pictures/final_csrf/Attack$ python3 CSRF.py
 * Serving Flask app 'CSRF' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 597-982-812
```

Website Image:

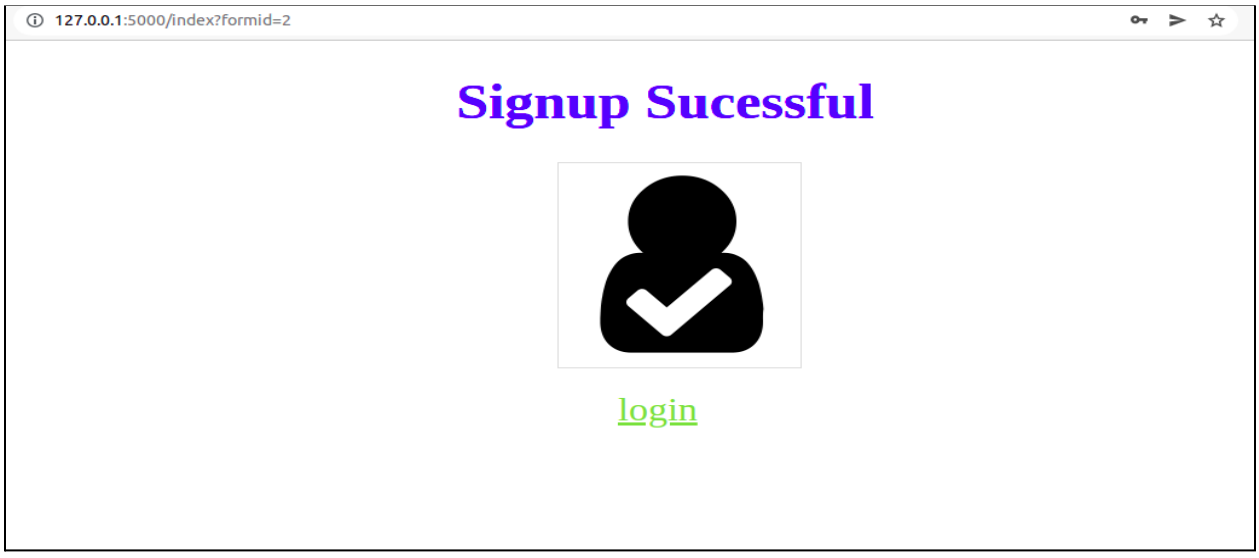


The above shown image is the image of our banking website. From here we have to register our user first into the system.

Registration:

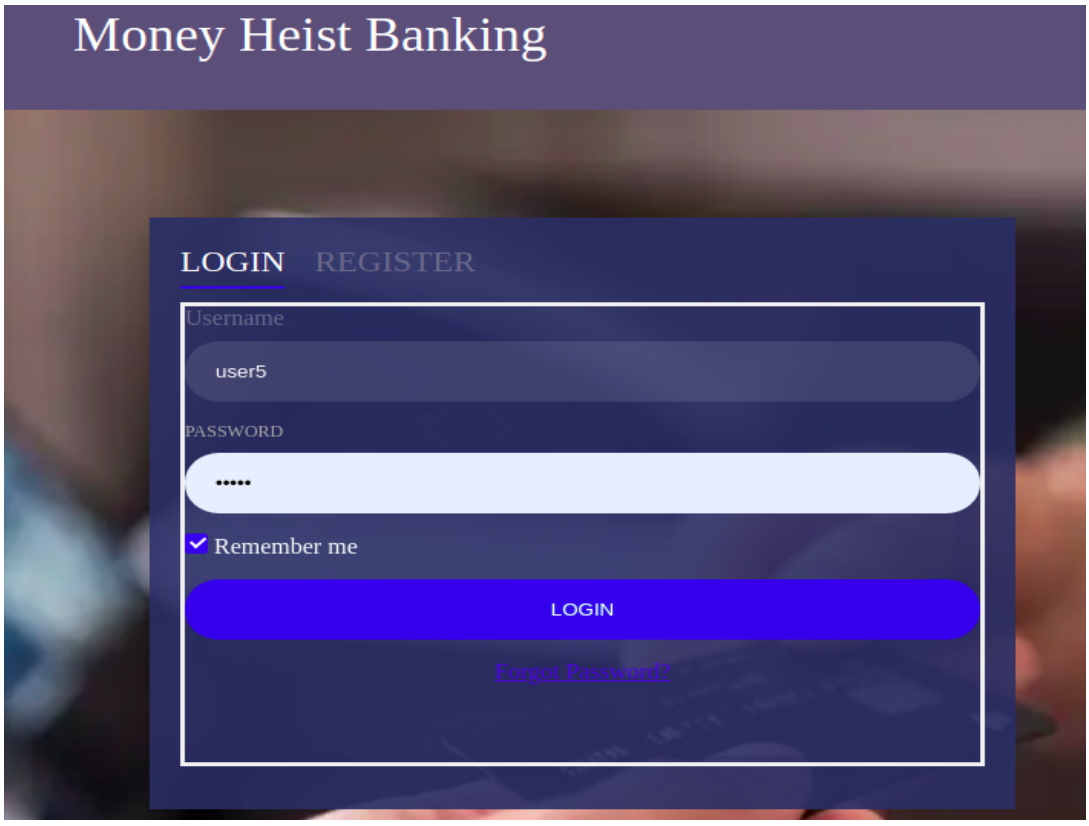


To register a new user we website is asking for Email-id, Username, Password and profession. Users have to enter the details and then have to press the register button. If the username is unique then registration will be successful. It will be redirected to the following page.

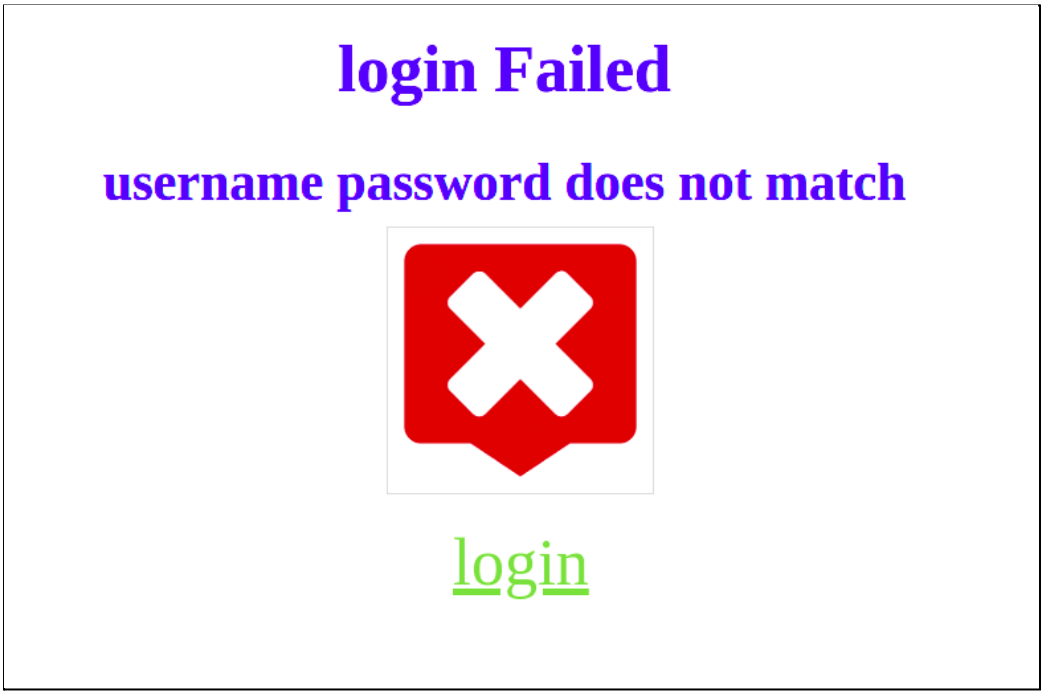


If the Username used for registration is already registered then the registration will be failed and it will show the error message as shown in the below picture.

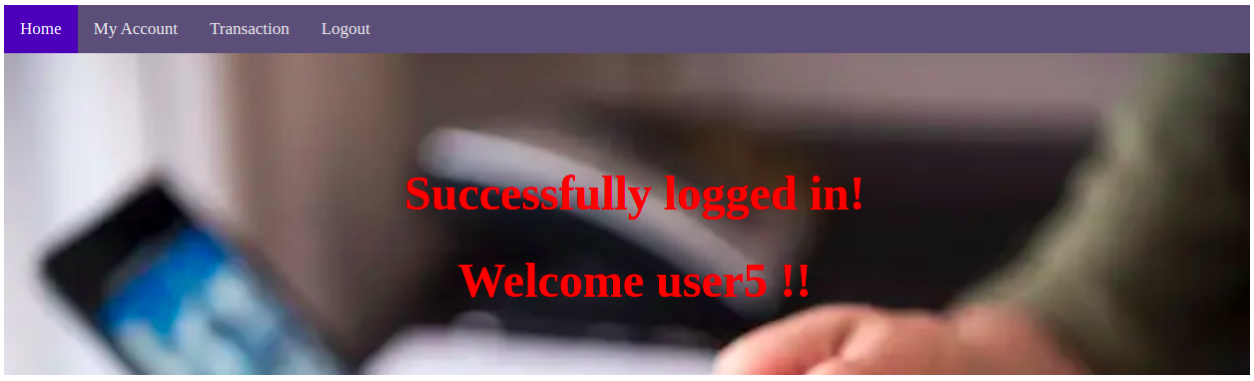
Login:



After completion of the signup page it will show the login page. If the user enters the correct details (if password and username matches) then the login will be successful. If password and username does not match then it will show an error message as shown in the below image.

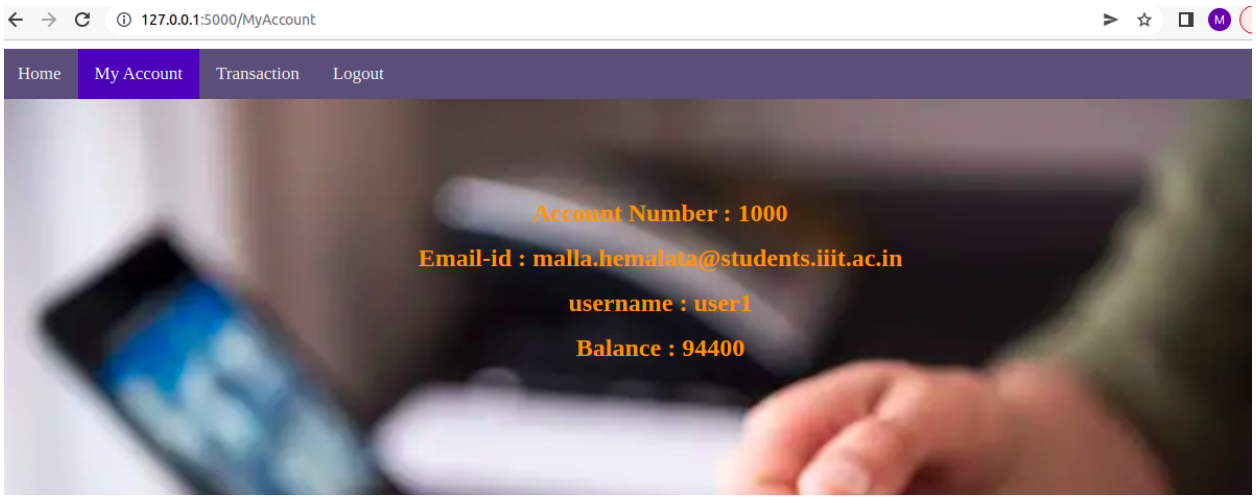


After Login:



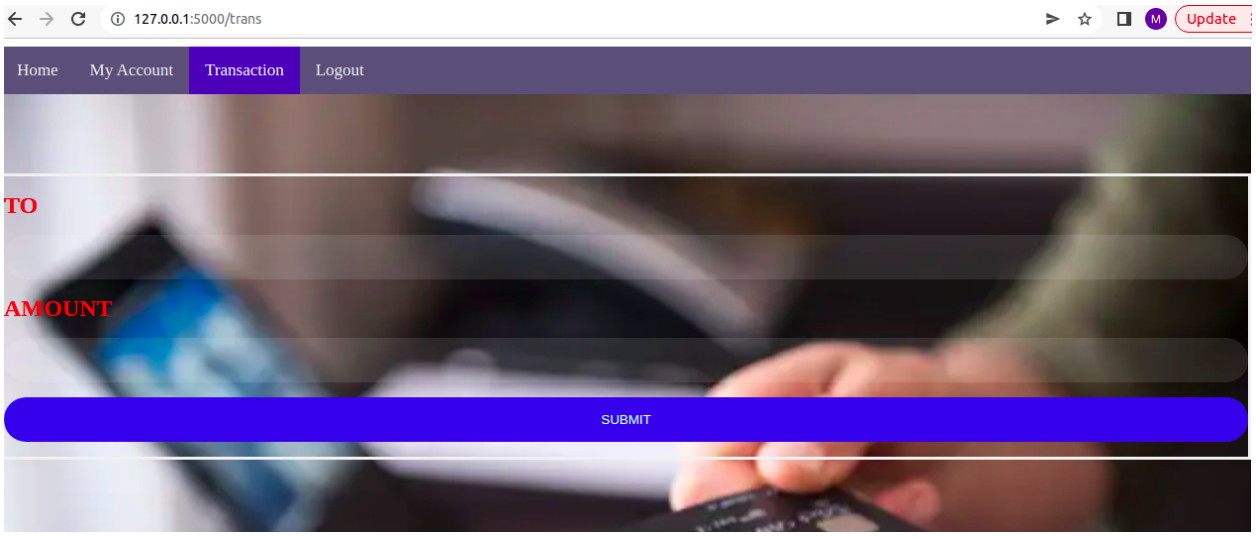
If the user provides the correct username and password then he will be directed into his account page which will be as shown in the above image. Welcome user5 in the image shows user5 is the username of the user. It contains four sections. One is home(same page) second one is My account page which gives account details. Third one is the transaction page and the fourth is the logout button.

User Account details:

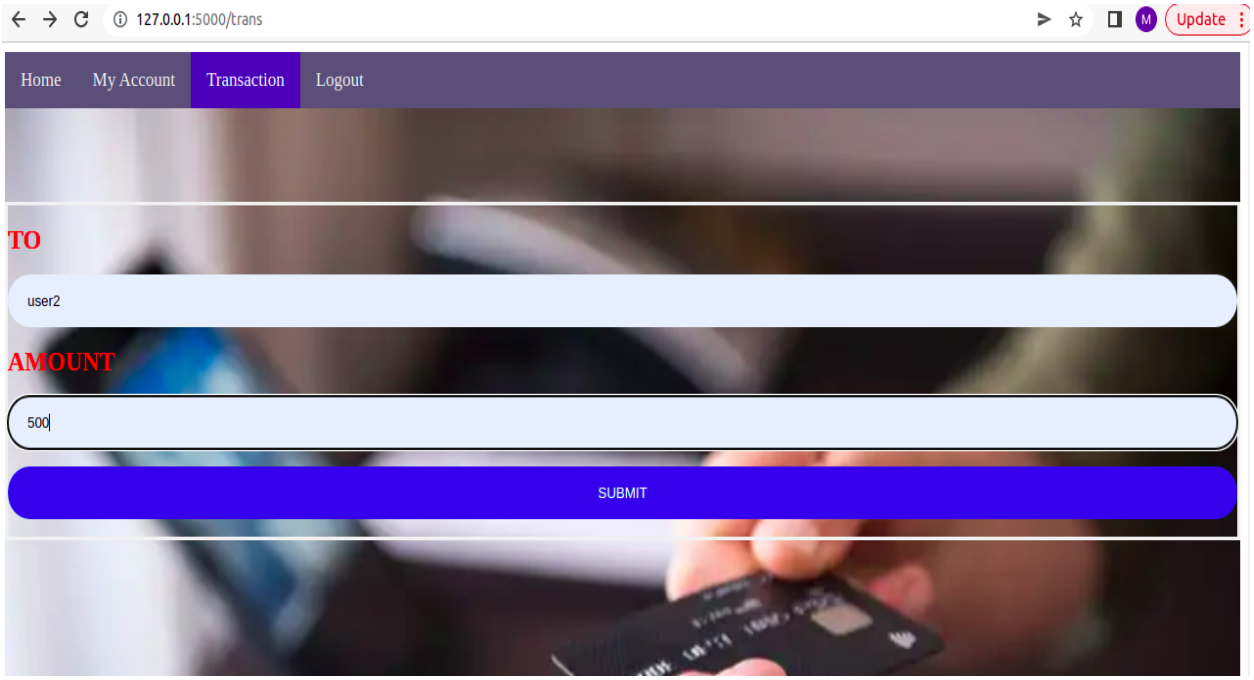


This is a screenshot of MyAccount page. It shows the account number, email-id, username and balance of the user account.

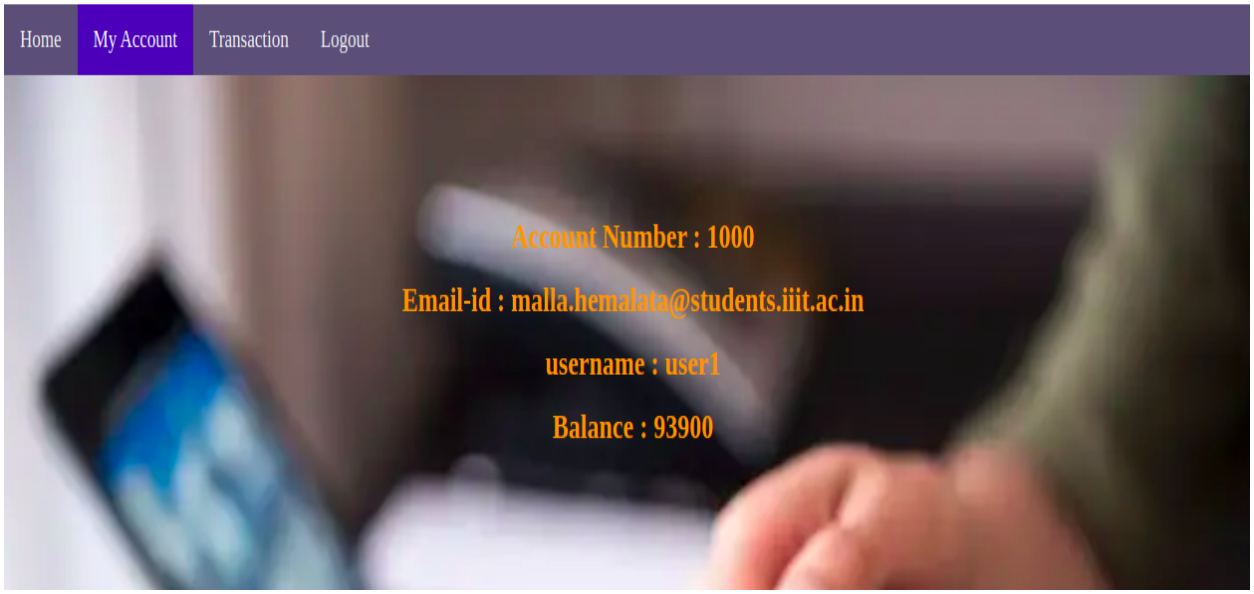
Transaction:



This is a screenshot of the transaction page where users have to enter to and amount details. Here means the user has to enter the username for which he has to send money. Amount is the money he wants to transfer from his account to another user(to).



Here user is sending money to user2 of an amount 500.

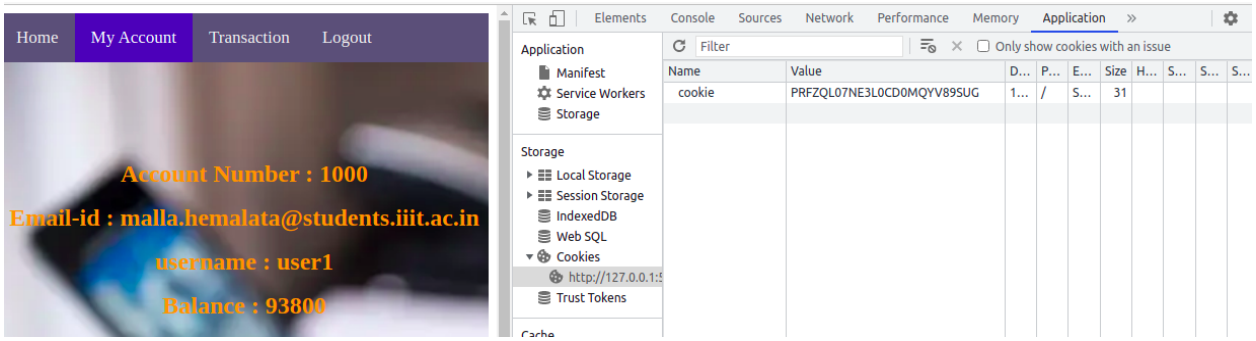


This is the balance of user1 before the transaction is 94400. After transferring 500 to user2 his balance has become 93900.

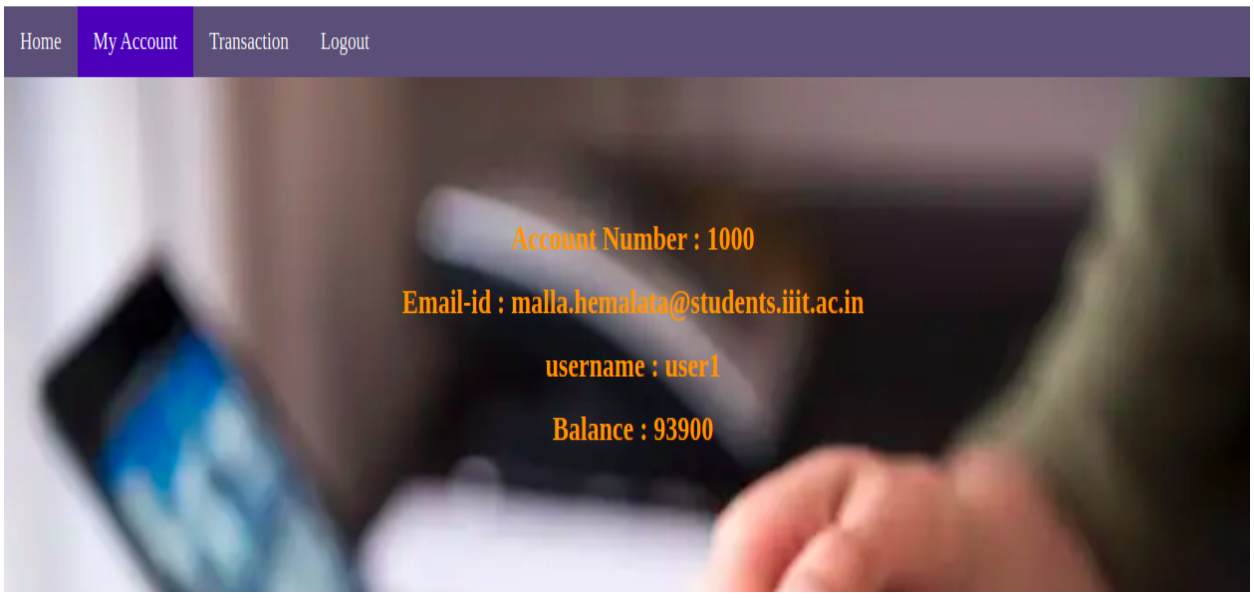
Attack:

When the session begins, the user is assigned a cookie which is stored in the user browser.

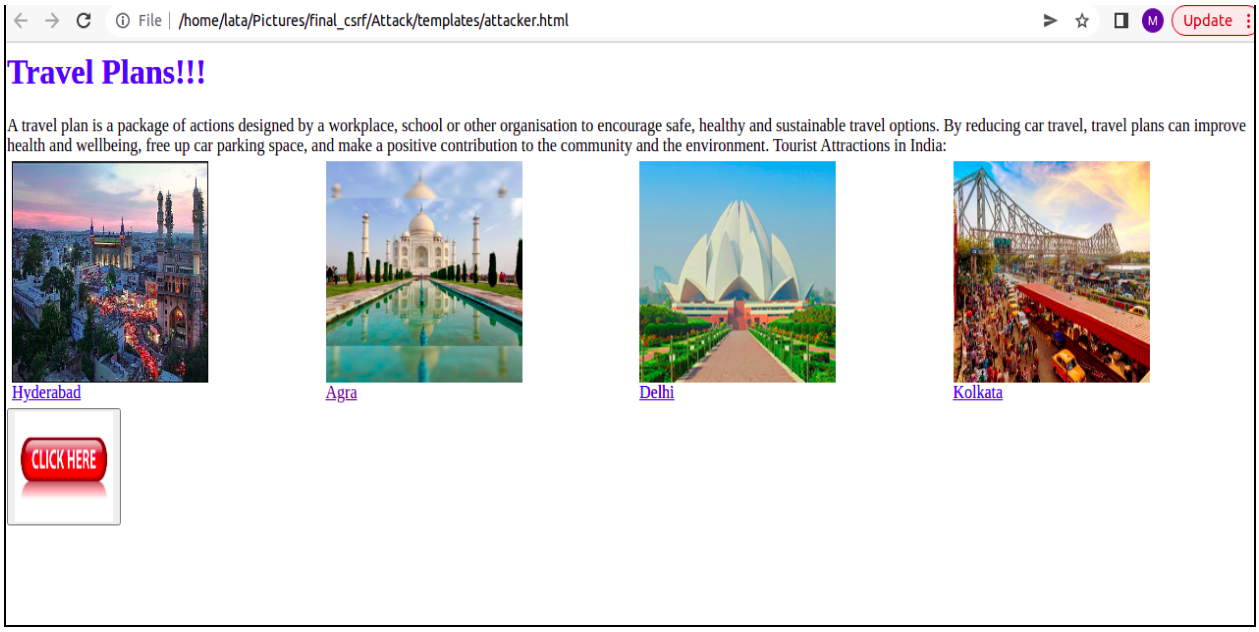
The cookies set for the session is shown below:



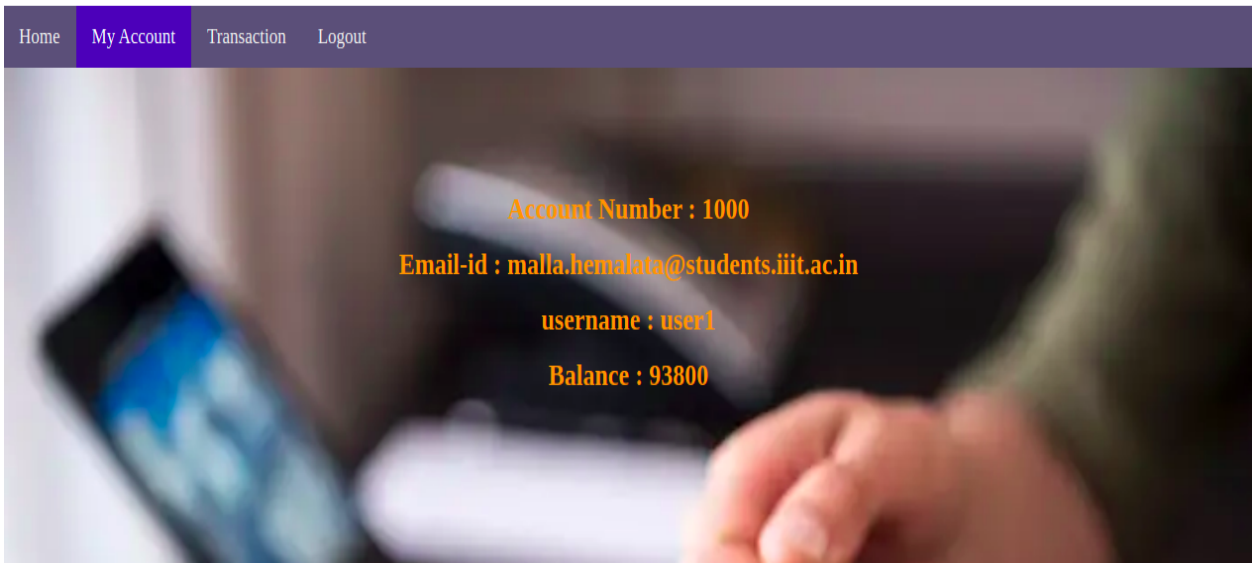
The balance of the account is shown below:



While logged in, if the user visits any other website or receives the link via email and opens the malicious page shown below:



The page looks attractive with all the traveling guidance and there's a "click here" button. If the user, out of curiosity clicks on the button, Then the user will not know any effect. But if they check their account details, the balance has been reduced by 100 Rs. It is because in the attacker.html file, the default amount to transfer is set as 100 to user2 (attacker).



The code of attacker.html is:

```
<form id="myForm" class="login-container" action="http://127.0.0.1:5000/transaction" method="POST">
    <div class="group">
        <!-- <label for="from" class="label" value="user1">From</label> -->
        <input name ="from" id="from" type="hidden" value="user1">
    </div>
    <div class="group">
        <!-- <label for="to" class="label" value="user2">To</label> -->
        <input name="to" id="to" type="hidden" value="user2">
    </div>
    <div class="group">
        <!-- <label for="amount" class="label" value="100">Amount</label> -->
        <input name ="amount" id="amount" type="hidden" value="100">
    </div>
    <div class="group">
        <input type="hidden" value={{cookie}} name="cookie" />
    </div>
    <div class="group">
        <!-- <input type='hidden' value="submit"> -->

         <button type="submit" class="button" ><img src="https://bit.ly/3BF9FD7" height ="100" width
     </div>

</form>
```

Here, as the cookie can be achieved from the user's browser, the form can be triggered directly to the transaction link as action.

Other possible attack:

Here we just implemented the money transfer from the user into the attacker account. Attacker can also change the password of the account or he can even delete the account.

Consider some social media applications like facebook. If a user opens facebook in one tab and opens an attacker link in another tab then the attacker can steal some personal information, change password or delete facebook account. So we should be careful enough while opening new websites or links.

Solution 1: Synchronizer Token (Anti-CSRF token)

This is the prevention technique called Synchronizer tokens or Anti-CSRF tokens. In this, the server will maintain a randomly generated

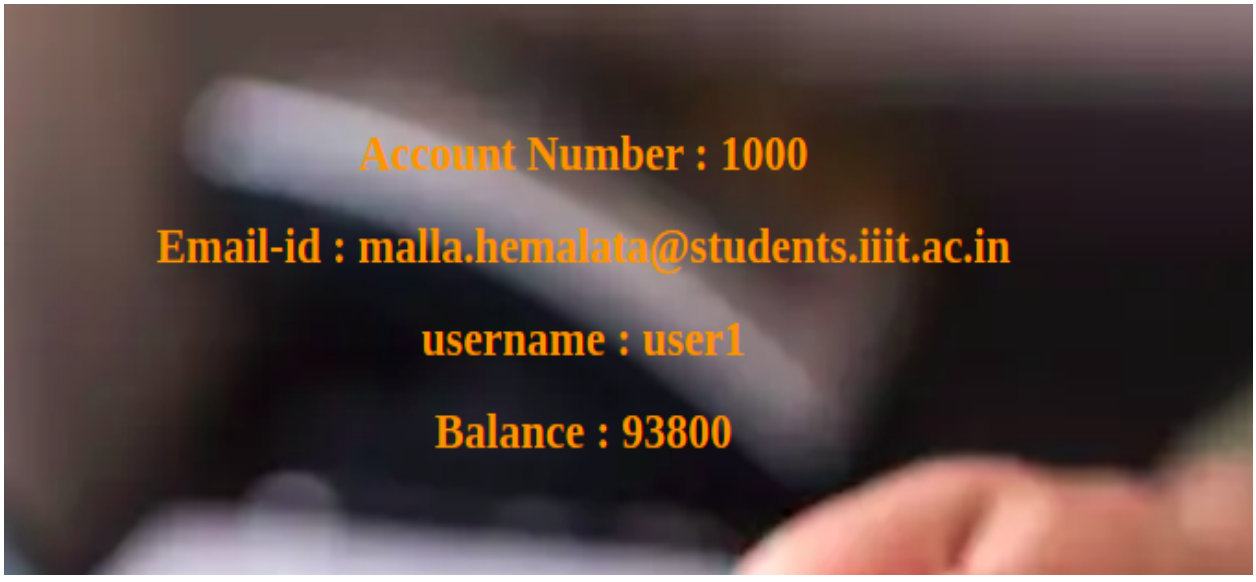
token for each user. While submitting any request the token will be submitted hidden or embedded to the server and the server will match the token. The attacker won't be able to identify the token associated with the user thus the attack is prevented.

```
lata@lata-Vostro-3559:~/Pictures/final_csrf$ cd ./Synchronizer_token
lata@lata-Vostro-3559:~/Pictures/final_csrf/Synchronizer_token$ python3 CSRF.py
 * Serving Flask app 'CSRF' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 597-982-812
```

To Run this solution code, we need to run the server from the Synchronizer_token folder.

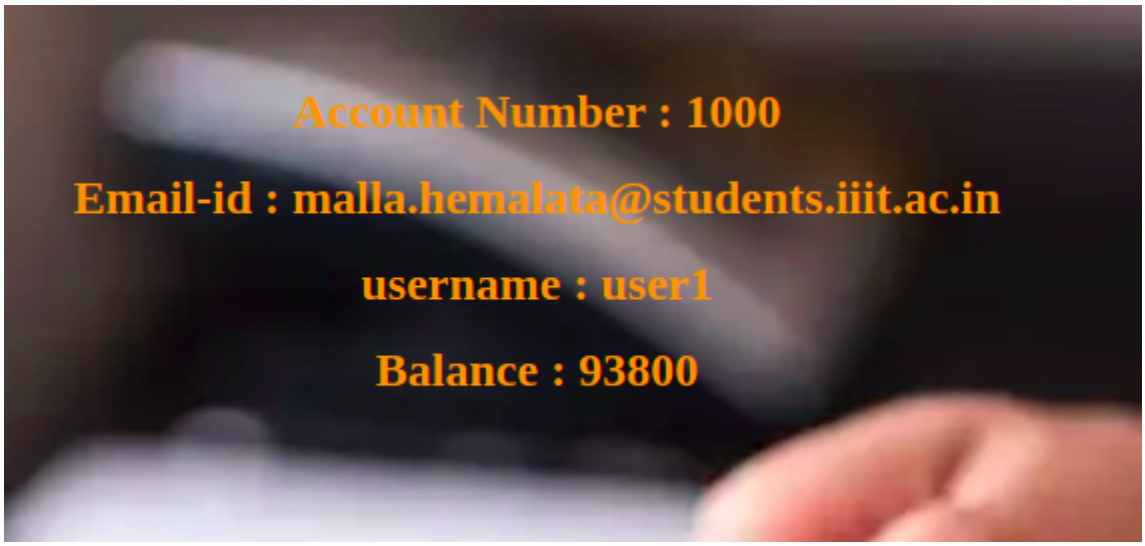
```
1007    user3@gmail.com user3   user3   student 100000  5Q0YLVEZEUYZITE6PUBJ6M29E
```

The token corresponding to the user is stored during registration.



The balance of user1 before the attack is shown in the image.

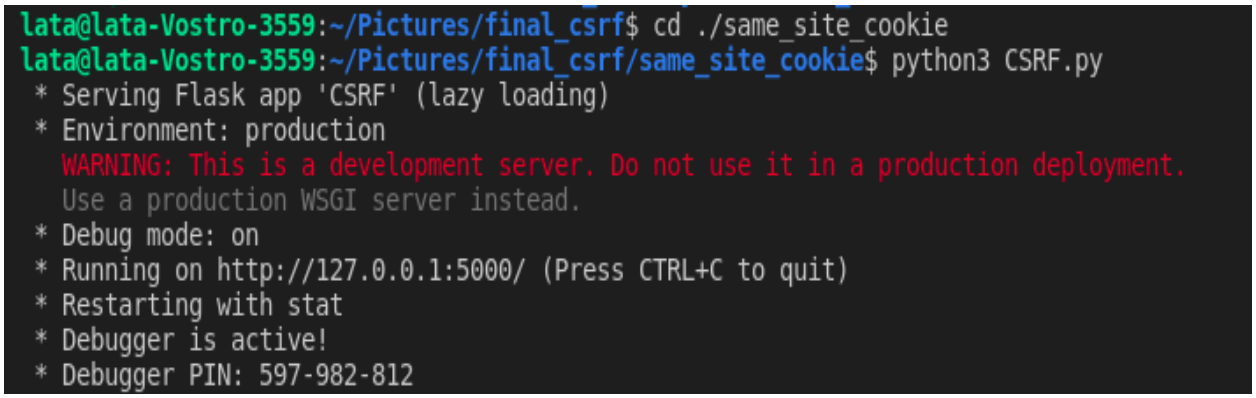
As usual the attacker placed the malicious form in the click here button. When the user clicks on the button, the transaction action is triggered. But, due to the lack of Synchronizer token, the transaction can't be completed.



The balance after the user has clicked on the "Click here" button is still the same as before. This implies that the CSRF attack has been prevented successfully.

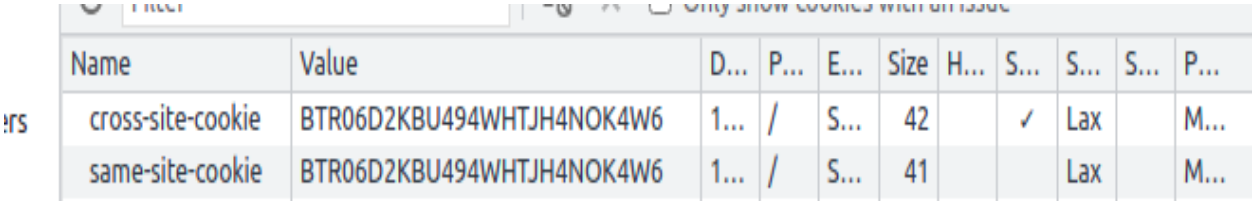
Solution 2: Same-site Cookies

The SameSite attribute can be used to control whether and how cookies are submitted in cross-site requests. By setting the attribute on session cookies, an application can prevent the default browser behavior of automatically adding cookies to requests regardless of where they originate. In this, the cookies sent to the user will be valid only for one site. If the user accesses the malicious link from another site, it won't be able to trigger the required action as the attacker can't tell whether cookies are used or same-site cookies.

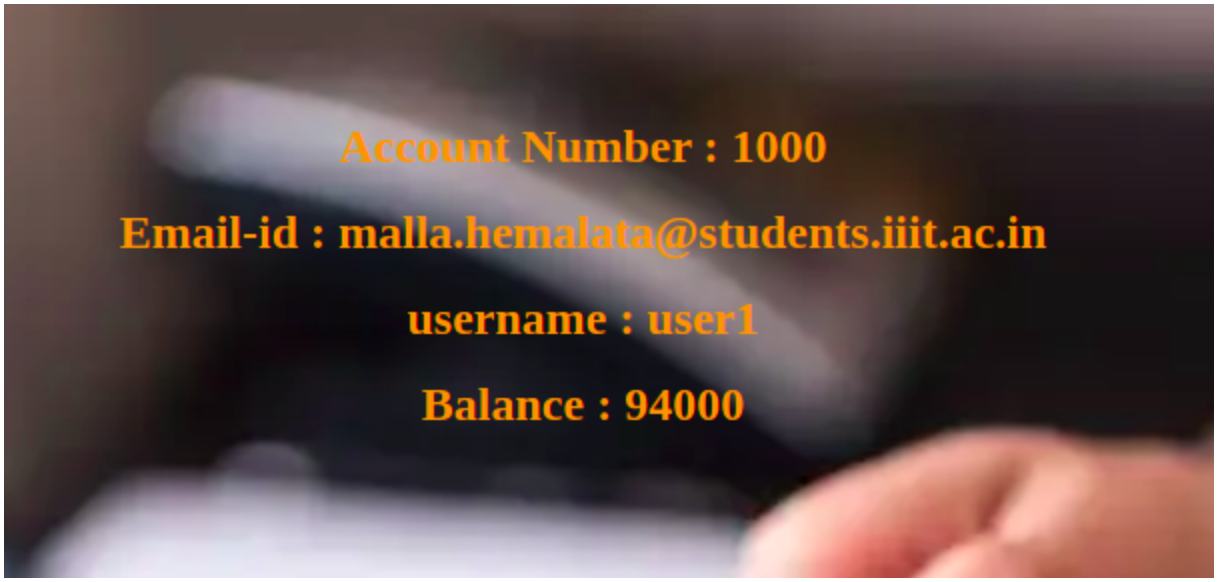


To run the same-site cookie solution, we need to run CSRF.py from the same_site_cookie folder.

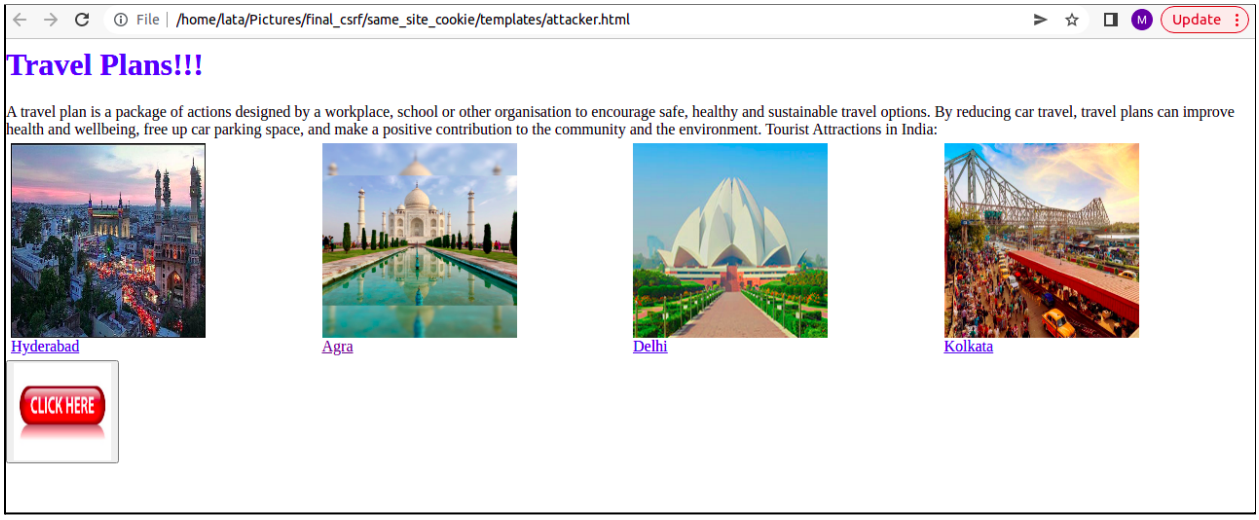
The cookies set for this session is shown below:



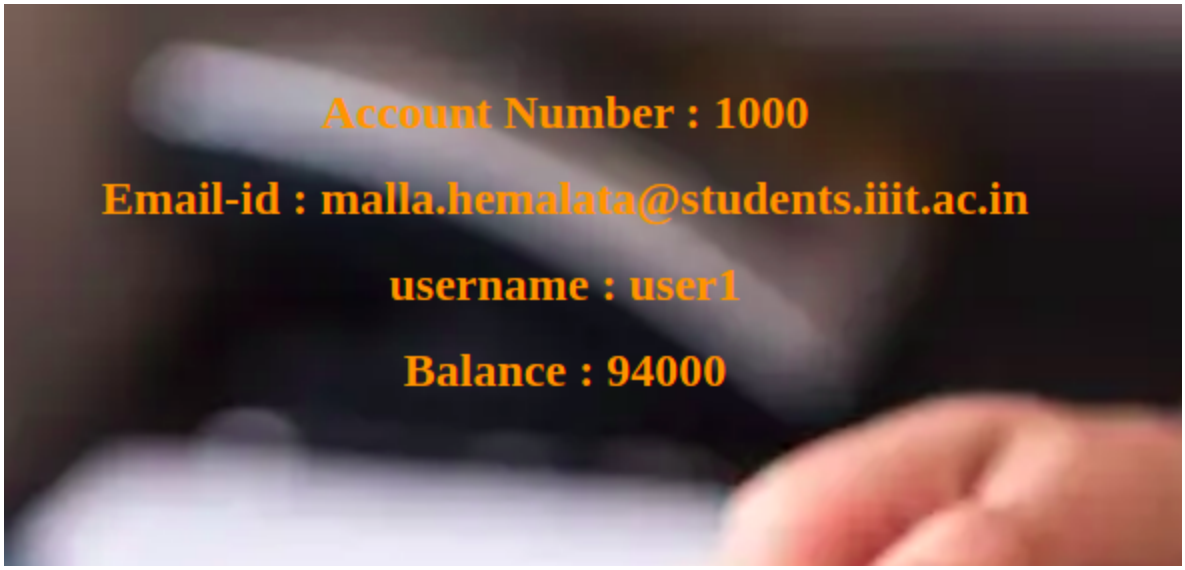
Name	Value	D...	P...	E...	Size	H...	S...	S...	S...	P...
cross-site-cookie	BTR06D2KBU494WHTJH4NOK4W6	1...	/	S...	42		✓	Lax		M...
same-site-cookie	BTR06D2KBU494WHTJH4NOK4W6	1...	/	S...	41			Lax		M...



The balance before the attack for user1 is shown in the image.



If the user clicks on the “Click here” button in the malicious page, the transaction request is triggered but we can see in the image below that the final amount is still the same as the initial amount. This is because of using the same-site cookies.



Conclusion:

A CSRF attack occurs if there is any privileged action happening in the server based on user login which the attacker is interested in. If the application relies solely on cookies or user login, then that application is most vulnerable to this type of attack.

In this project, we have implemented one such vulnerable banking website that is performing transactions solely based on cookies and implemented the CSRF attack that can be done on this website.

As a prevention to this attack, we have implemented two solutions, Synchronizer token and Same-site cookie. In these, addition of an extra authentication field and site restriction has been implemented. These are able to prevent the CSRF attack that we implemented. But CSRF attacks can be more advanced which require more effective ways to prevent them.