Project 2 - Cross Site Request Forgery (CSRF) and Cross Site Scripting (XSS) attacks

Network Security - Spring 2025

Due Date: February 28 by 11:55 pm

Contents

1 Introduction

In the previous project, you implemented basic aunthentication and authorization mechanisms in the DISCO web application, so that only aunthenticated users, with the required privileges are able to access the protected resources. In this project, you will learn about two common web application vulnerabilities, **Cross Site Request Forgery (CSRF)** and **Cross Site Scripting (XSS)** attacks. You will learn how, despite the presence of authentication and authorization mechanisms, an attacker can still exploit these vulnerabilities to perform malicious actions.

2 Getting Started

Before you get started, here's a few things you need to do to setup your environment.

1. Make sure that your DB instance is up and running and not paused. You can check this by logging into your Oracle Cloud account and visiting the dashboard page of the DB instance. If the instance is paused, you can start it by clicking on the "More Actions -> Start".

2. Similarly, make sure that your VM instance (used for the previous project) is up and running, with your project 1 files, and implementation there. If you have terminated the instance, please refer to the Project 1 manual to make a new VM, and push your implementation files, along with the wallet to the new VM.

3. Download the project files from LMS. You need to add a few UPDATES to your existing codebase. The updates are as follows:

   - `bill_details.html` - This is the updated bill details page. Replace the existing `bill_details.html` file in the `templates` directory with this file, on your VM. You can either copy and then paste the contents of the updated file, in the existing file, or you can use `scp` to copy the file from your local machine to the VM.
   - `bill_ret_update.py` - The file contains the updated code for the `bill_retrieval` endpoint. Update the existing function named `bill_retrieval` in the `app.py` file with the code in this file.
   - `bill_adj_update.py` - The file contains the updated code for the `bill_adjustment` endpoint. Update the existing function named `bill_adjustment` in the `app.py` file with the code in this file.
   - `util.py` - Replace the old `util.py` file in the `application` directory with this file.

4. **Note that the updated endpoints only contain the functional code, and not the authentication and authorization checks that you implemented in part 1. You must keep those checks and update the rest (simply copy paste below the checks).** If you haven't implemented the checks, it is highly recommended that you do so, as Project1 is a pre-requisite for this project.

5. Once you have updated the files, the rest of the process (i.e. of launching the application, and accessing it) remains the same as the previous project. There's no need for setup scripts.

3 Project Tasks

The project is divided into three parts. The first part is concerned with the of CSRF attacks, while the second part is concerned with the XSS attacks, and how CSRF can be used in conjunction with XSS. The third part is a reflection on the attacks, and the counter-measures to prevent such attacks.

3.1 Part 1: Cross Site Request Forgery (CSRF) Attacks

A CSRF attack involves a victim user, a trusted site, and a malicious site. The victim user holds an active session with a trusted site while visiting a malicious site. The malicious site injects an HTTP request for the trusted site into the victim user session, causing damages. In our case, the trusted site is the DISCO web application, and the malicious site is the attacker's site. The attacker's site (i.e., malicious site) can contain any element (e.g. a hidden form, an image tag, etc.) that has the capability to send a request to the DISCO application, to perform a malicious action. The victim user is an authenticated user of the DISCO application, who visits the attacker's site. The attacker's site will submit a request to the DISCO application, to perform a malicious action, on behalf of the victim user.

Note that this is all possible because of the session token that the victim user holds with the DISCO application. This token is stored as a cookie, which is automatically sent with every request to the application. This is convenient for the user, as the user does not have to login again and again, but it also opens up the possibility of CSRF attacks, as an adversary can craft a request that the user's browser will automatically send, along with the session token, without the user's knowledge.

3.1.1 Proof of concept

To demonstrate a CSRF attack, we will use a benign action, that is still considered unauthorized. In this case, we will use make an unauthorized request to the `/dashboard` endpoint.

1. Make sure that your app is up and running.
2. Login to the app as any user (e.g. `test_u1` with the password `secret`).
3. If you are able to successfully login, the session cookie should be set.
4. You have been provided with a simple HTML file, `csrf-demo.html`. The page contains an embedded `<img>` tag, whose source can be set to any arbitrary URL. When the page loads, the browser automatically sends a GET request to the URL specified in the `src` attribute, in order to load the image. However, this `src` can be a crafted malicious URL, that will submit a request to the DISCO application, to perform an unauthorized action.
5. This page will act as the attacker's site. You can choose to deploy this page on a web server, but opening it in Chrome locally works as well. Update the `src` attribute of the `<img>` tag, to point to the dashboard endpoint of your deployed DISCO application. The URL should be something like `https://<your-vm-ip>/dashboard`.
6. If the session cookie is still active, and the user is logged in, the browser will automatically send a GET request to the `dashboard` endpoint when the page is loaded.

Retry the above with the developer console open. In the network section, you will see a successful hidden request made to the `dashboard` endpoint, without the user's knowledge. You may also notice that the session cookie was included in the request headers. This is the essence of a CSRF attack.

3.1.2 CSRF attack using a GET request

Any HTML element that can make a GET request can be used to perform a CSRF attack. In the previous example, we used an `<img>` tag, but other elements like `<script>`, `<link>`, `<iframe>` etc. can also be used. Similarly, Javascript and `fetch` API can also be used to automatically perform these requests.

You are required to craft malicious pages (the supposed attacker's site) for this task. Provide the HTML code for the page that can end a user's session by triggering a request to the endpoint used by signout. The attacker's site should contain a hidden form that submits specifically a GET request to the `/signout` endpoint (or whatever endpoint you have named that triggers a signout in your implementation) of the DISCO application automatically upon loading. This would result in the end of the user's session, without the user's knowledge.

3.1.3 CSRF attack using a POST request

CSRF attacks can also be performed using POST requests. In this case, the attacker's site may contain a form that submits a POST request to the application, to perform a malicious action. The form can be hidden, and the submission can be done automatically using Javascript, upon page load, or through some other event, e.g. a button click.

You are required to craft malicious pages (the supposed attacker's site) for this task. Provide the HTML code for the page that can perform the following attacks:

- **Unauthorized bill adjustment**: The attacker's site contains a form that submits a POST request to the `/bill-adjustments` endpoint of the DISCO application, to adjust the bill amount for a specific bill. Refer to section 4 for the required data to be submitted in the form.
- **Unauthorized bill payment**: The attacker's site contains a form that submits a POST request to the `/bill-payment` endpoint of the DISCO application, to pay the bill amount for a specific bill. Refer to section 4 for the required data to be submitted in the form.
- **Information extraction**: Assume that you are only aware of the customer ID and connection ID. Then you can get internal bill IDs for any month and year, by extracting this information from the bill retrieval page. Craft POST requests meant for the `/bill-retrieval` endpoint to extract the bill details for a specific bill. From the retrieved HTML response, parse out the BillID, and print it to the console using `console.log()`.

3.2 Part 2: Cross Site Scripting (XSS) Attacks

Cross-site scripting (XSS) is another type of vulnerability commonly found in web applications. This vulnerability makes it possible for attackers to inject malicious code (e.g. JavaScript programs) into the victim's web browser. Using this malicious code, attackers can steal a victim's credentials, such as session cookies. The access control policies (i.e., the same origin policy) employed by browsers to protect those credentials can be bypassed by exploiting XSS vulnerabilities.

The primary goal of an XSS attack is to inject malicious scripts into web pages that are viewed by other users. The attacker can then steal sensitive information, such as session cookies, or set up a keylogger, etc. all done silently, without the knowledge of the victim. In this task, you will demonstrate how an attacker can exploit an XSS vulnerability in the DISCO application.

For the attack to work successfully, there must be a way to inject the malicious script into the application. Similarly, there must be a way where this script is loaded and executed. The ideal candidate for this is the bill adjustment feature, where the application expects the adjustment reason to be provided by the user, as a string. This adjustment reason is then displayed on the bill details page when the user views the bill. If the input is not sanitized properly, an attacker can inject a malicious script, which will then be executed when the user views the bill details page.

3.2.1 Proof of concept

Before we proceed with the tasks, we'll demonstrate the concept by manually inserting a script in the adjustment reason field, and then viewing the bill details page. This will help you understand how the attack works, and what the attacker can achieve.

1. Make sure your app is up and running. Log in to the app, as a user with the required privileges to adjust the bill amount. In this case, that would be the user `test_u3` with the password `secret`.
2. Navigate to the bill adjustment page, and adjust the bill amount for bill. Let's use the bill ID `797` (which is an unpaid bill in the DB) for this purpose. You can use any amount and any name for the officer. For the adjustment reason, we will insert a simple script that will alert the user.

```
<script>alert("XSS attack");</script>
```

3. Submit the form, and you should see the adjustment receipt if the adjustment was successful.
4. Sign out, and login as a customer (though that is not necessary). Navigate to the bill details page, and view the bill details for the bill ID `797`. Use the following values for the form fields:
   - **Connection ID**: `DISCO-COM-1003`
   - **Customer ID**: `LHR-COM-0003`
   - **Billing Month**: `08`
   - **Billing Year**: `2024`
5. You should see a pop-up alert, with the message `XSS attack`. This is the script that you inserted in the adjustment reason field, and it was executed when the bill details page was loaded.

Notice how we had to manually insert the script in the adjustment reason field, and then view the bill details page. However, we can circumvent this limitation by using a CSRF attack (as done in 3.1.3) to make unauthorized adjustments to the bill, and then inject the script in the adjustment reason field. This way, the script will be executed when the bill details page is viewed, without the knowledge of the user.

3.2.2 Payload insertion using CSRF

In the previous section, we demonstrated how an attacker can manually insert a malicious script into the adjustment reason field. In this task, you will explore how attackers can use Cross-Site Request Forgery (CSRF) to inject different types of malicious payloads into the application. These payloads will be executed when the bill details page is viewed.

Your goal is to craft different CSRF attack pages that can inject XSS payloads. Below are the tasks you must complete:

1. **Session cookie stealing**: The attacker's site contains a form that submits an unauthorized adjustment, and the XSS payload script has the ability to steal the session cookie, and send these to the adversary for further exploitation. Use a `netcat` listening server to capture the session cookie.
2. **Keylogger**: The attacker's site contains a form that submits an unauthorized adjustment, and the XSS payload script has the ability to log the keystrokes of the user, and send them to a remote `netcat` server for further exploitation.

Test both of these scenarios out by first logging in as an authorized user, that can make bill adjustments. Then, visit the attacker's site, and submit the form. Finally, view the bill details page, and see if the malicious script is executed, by viewing the console logs, or the `netcat` server logs.

3.3 Part 3: Defense mechanisms

In this part, you will reflect on the attacks that you have demonstrated, and implement the defense mechanisms that can be employed to prevent them.

3.3.1 Defenses against CSRF attacks

In order to prevent CSRF attacks, the application must ensure that the request is coming from a legitimate source. This can be done by implementing the following defense mechanisms.

- **CSRF tokens**: The application can generate a unique token for each session, and include this token in the form data. When the form is submitted, the application can check if the token is valid, and reject the request if it is not. This way, even if the attacker is able to craft a request, they will not be able to include the CSRF token, and the request will be rejected. Update the `bill_retrieval` and `bill_adjustment` functions in the `app.py` file to include a secret CSRF token in the form data, and validate it before processing the request.

- **SameSite cookies**: The application can set the `SameSite` attribute of the session cookie to `Strict`. This will prevent the browser from sending the session cookie with cross-site requests, and will prevent CSRF attacks. Update the session cookie in the `app.py` file to include the `SameSite` attribute.

Make sure to test the application after implementing these defense mechanisms, to ensure that the attacks that you have demonstrated are no longer possible.

3.3.2 Defenses against XSS attacks

In order to prevent XSS attacks, the application must ensure that user input is properly sanitized before being displayed. For the case of this application, you would be required to sanitize the adjustment reason field, before displaying it on the bill details page. This can be done by escaping special characters, and removing any HTML tags from the input. Note that, this would be done by making sure that the adjustment reason text, after being retrieved from the database, is sanitized before being displayed on the bill details page.

4 Submission

You are required to submit the following files:

1. `csrf-signout.html`: The malicious HTML file that performs the session ending attack, as described in 3.1.2.
2. `csrf-post-adjustment.html`: The malicious HTML file that performs the unauthorized bill adjustment attack, as described in 3.1.3.
3. `csrf-post-payment.html`: The malicious HTML file that performs the unauthorized bill payment attack, as described in 3.1.3.
4. `csrf-post-retrieval.html`: The malicious HTML file that performs the information extraction attack, as described in 3.1.3.
5. `app.py`: The updated `app.py` file, with the updated endpoints, and the defense mechanisms incorporated.

Create an archive of these files named `s_<your_rollnumber>.zip` and submit it on LMS.

The grading for the project will be done via vivas, where you will be required to demonstrate the attacks and explain the code that you have written. Make sure you understand the code that you are submitting, and are able to explain it. A grading rubric and the viva schedule will be shared later.

Appendix

Details of unpaid bills in the Database

You may need to refer to this data in order to make valid adjustments or bill payments. Please note that if your DB state is all messed up, you can simply re-run the setup script `oic_setup.sh`. The script would reset and repopulate the database with the starting values.

Bill ID	Customer ID	Connection ID	Month	Year	Amount Before DueDate	Amount After DueDate
790	`LHR-RES-0021`	`DISCO-RES-1121`	8	2024	13088.33	14397.16
791	`LHR-RES-0022`	`DISCO-RES-1122`	8	2024	42503.84	46754.22
792	`LHR-RES-0023`	`DISCO-RES-1123`	8	2024	23811.1	26192.21
793	`LHR-RES-0024`	`DISCO-RES-1124`	8	2024	59974.56	65972.02
794	`LHR-RES-0025`	`DISCO-RES-1125`	8	2024	8876.46	9764.11
795	`LHR-COM-0001`	`DISCO-COM-1001`	8	2024	160060.39	176066.43
796	`LHR-COM-0002`	`DISCO-COM-1002`	8	2024	193286.12	212614.73
797	`LHR-COM-0003`	`DISCO-COM-1003`	8	2024	246622.42	271284.66
798	`LHR-COM-0004`	`DISCO-COM-1004`	8	2024	177091.33	194800.46
799	`LHR-COM-0005`	`DISCO-COM-1005`	8	2024	267873.92	294661.31
800	`LHR-COM-0006`	`DISCO-COM-1006`	8	2024	356462.42	392108.66