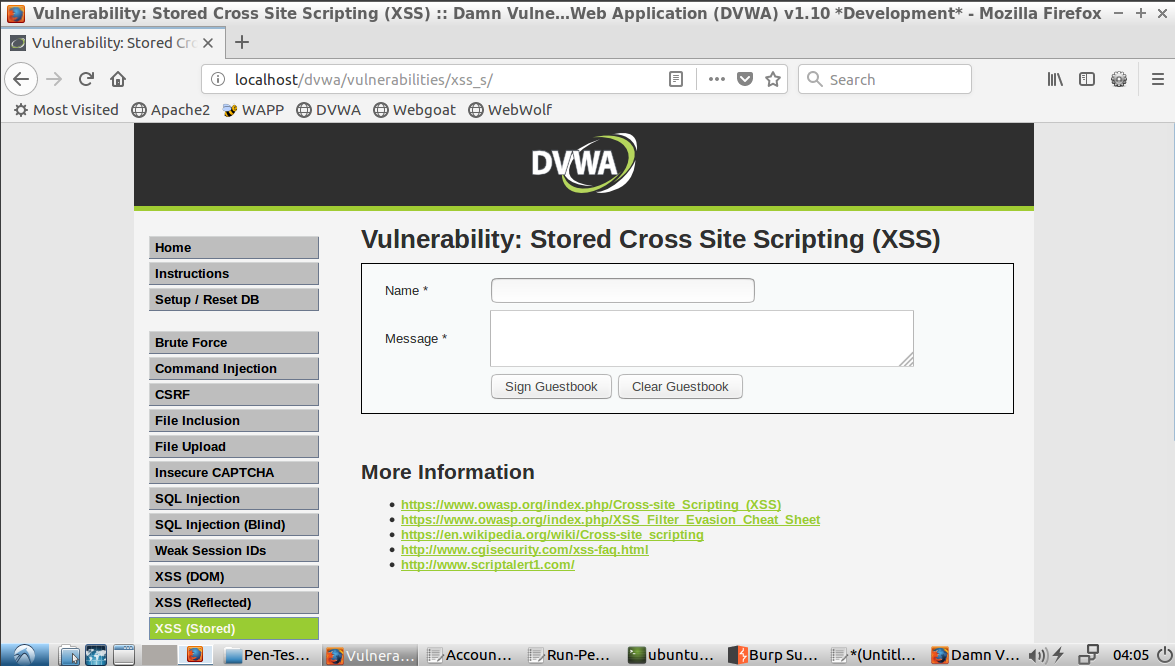
# Group Project 4

(Group 1)

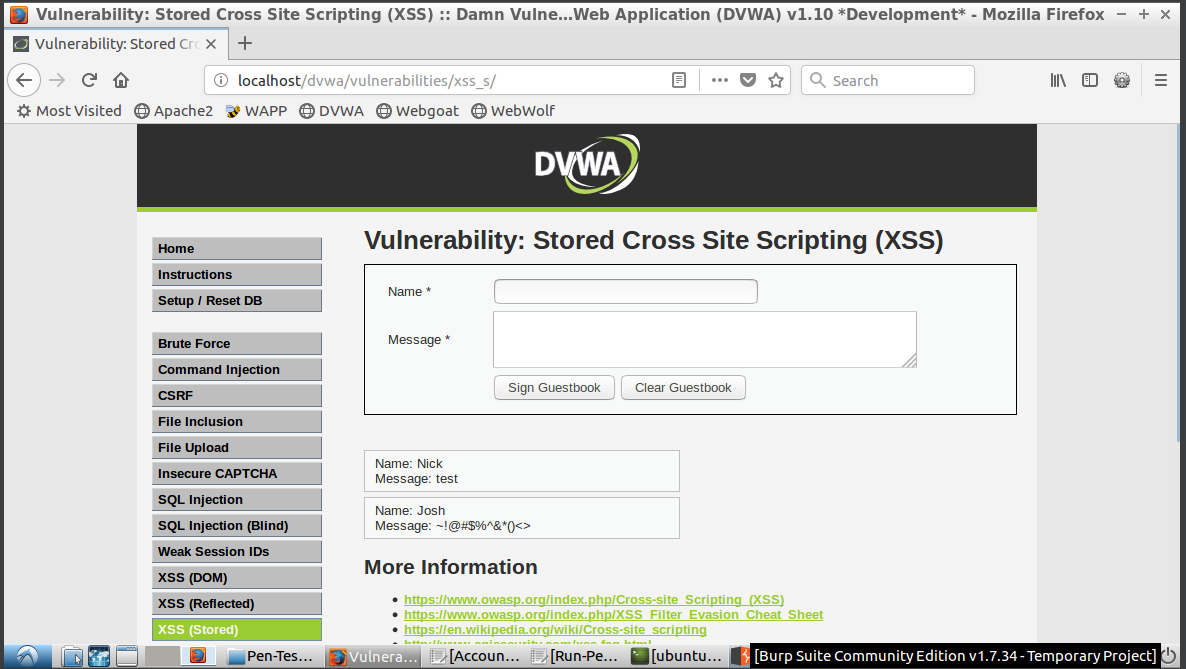
Mario Amaro   
Sushruti Bansod   
Joshua Fishman   
Nick Forleo

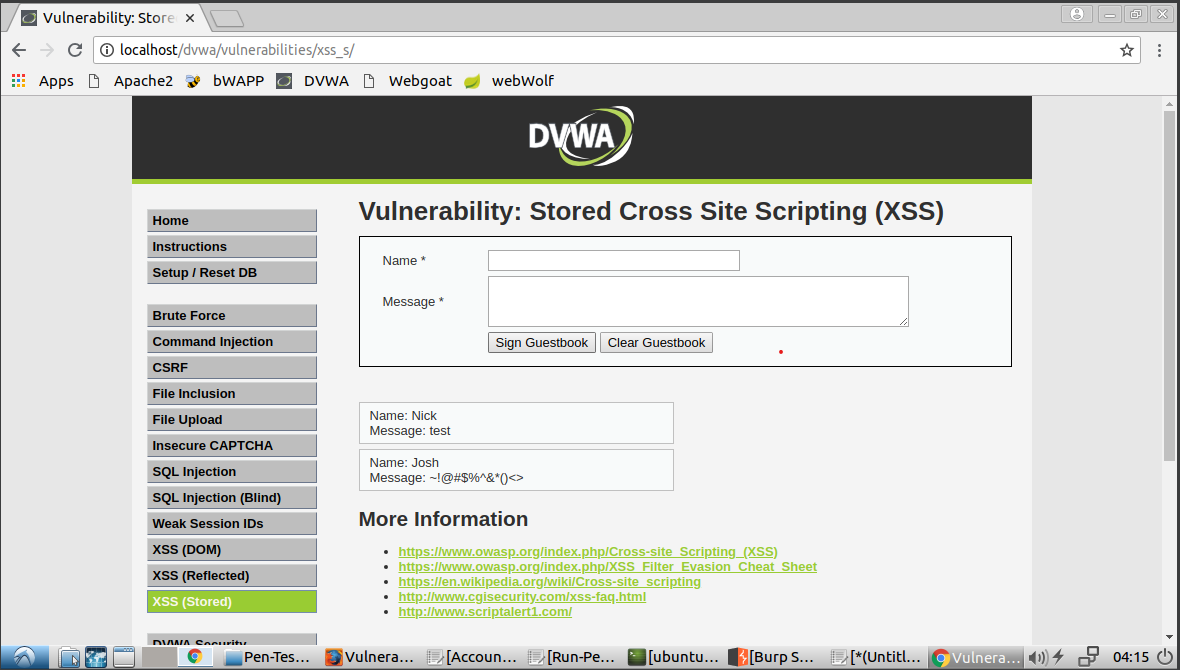
**Task 1:** Show a hacking scenario using XSS technique

To uncover an XSS vulnerability, we looked through DVWA. We were able to uncover this vulnerability without the use of a testing toolkit. We did, however, use the toolkit to verify the requests and the data being transferred. This is where we found a form that accepts user input and prints it back out on the web page. Figure 1.1 shows what that page looks like before adding any data.

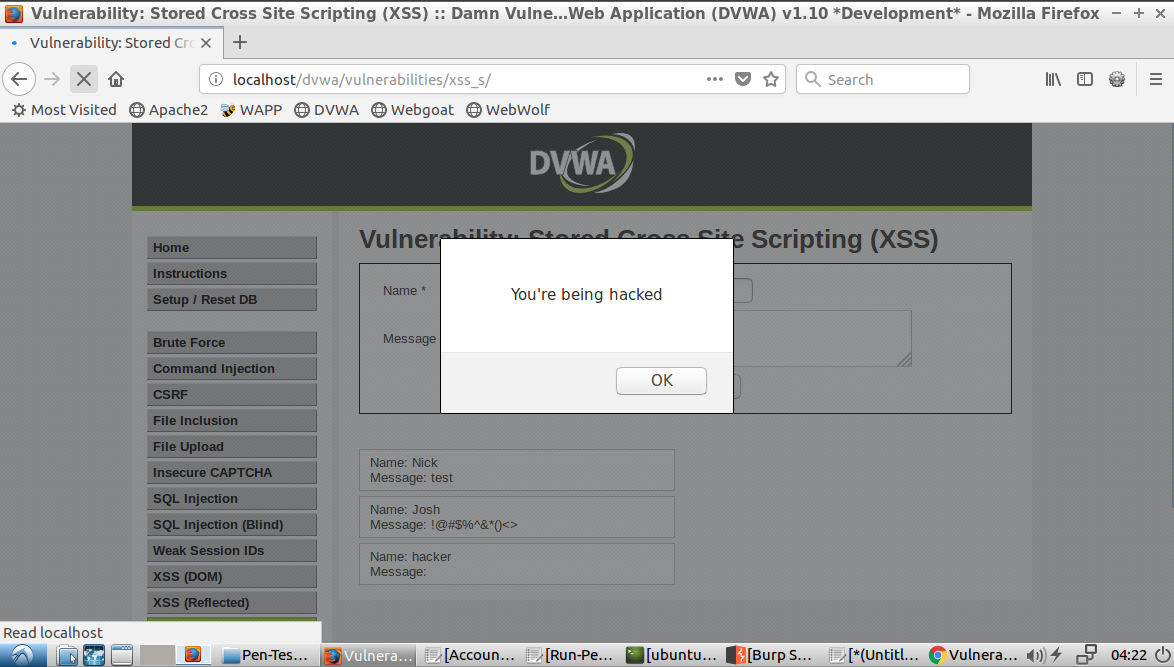
*Figure 1.1 - Form to enter data*

After inputting some dummy data, we discovered two things. The first being that special characters are not stripped out. This means that the user input does not have basic input sanitization. We show this in Figure 1.2. The second discovery was that the data persists through a reload and a new browser. This tells us that the data is being stored in a database and that other users have the same data returned from the database, which is shown in Figure 1.3.

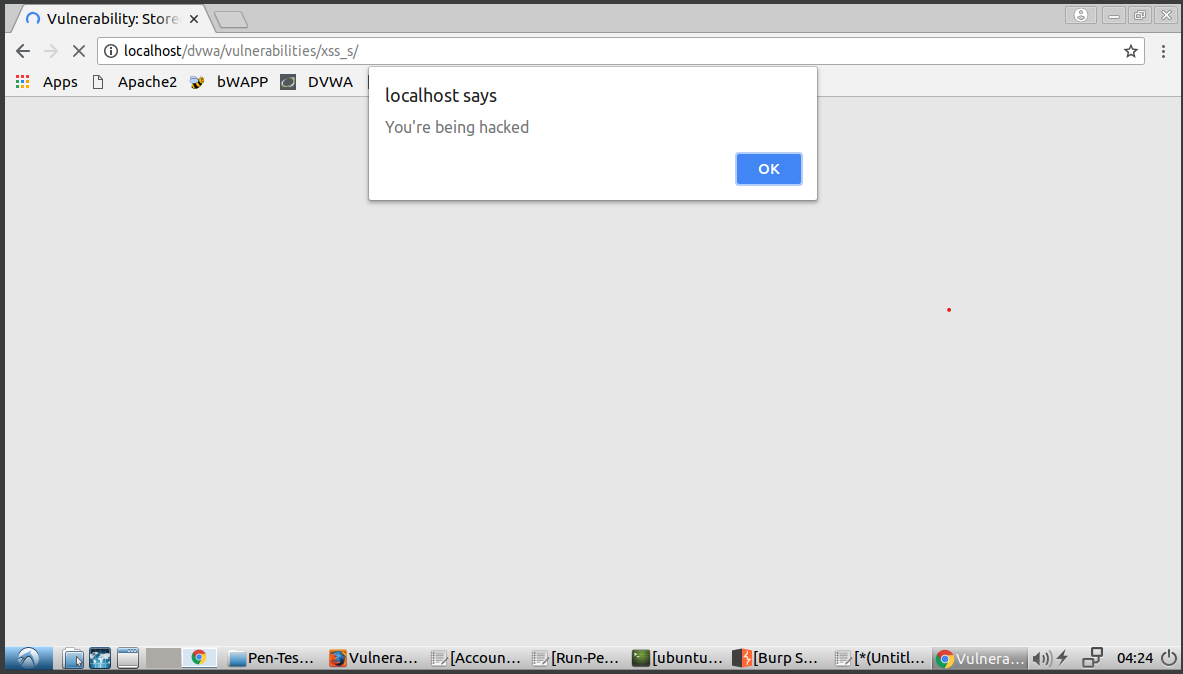
*Figure 1.2 - Special characters are rendered*

*Figure 1.3 - The data input from Mozilla Firefox is shown in a new browser (Chrome) and session*

Now that we have discovered this, we can craft a simple stored XSS attack. This is a stored XSS attack because we can save our malicious input onto the server which can then affect other users. We demonstrated this by inserting `<script>alert(“You’re being hacked”)</script>` in the message field. Once this is saved and the browser retrieves the content from the database, the browser tries to render it, causing our code to be executed. Figure 1.4 shows the results of the script when initially input (in Firefox) while Figure 1.5 shows how the separate browser (Chrome)/session is affected by the attack.



*Figure 1.4 - Original input in Firefox showing results of script saved to database*



*Figure 1.5 - Second browser (Chrome) on reload showing the script being retrieved from the database*

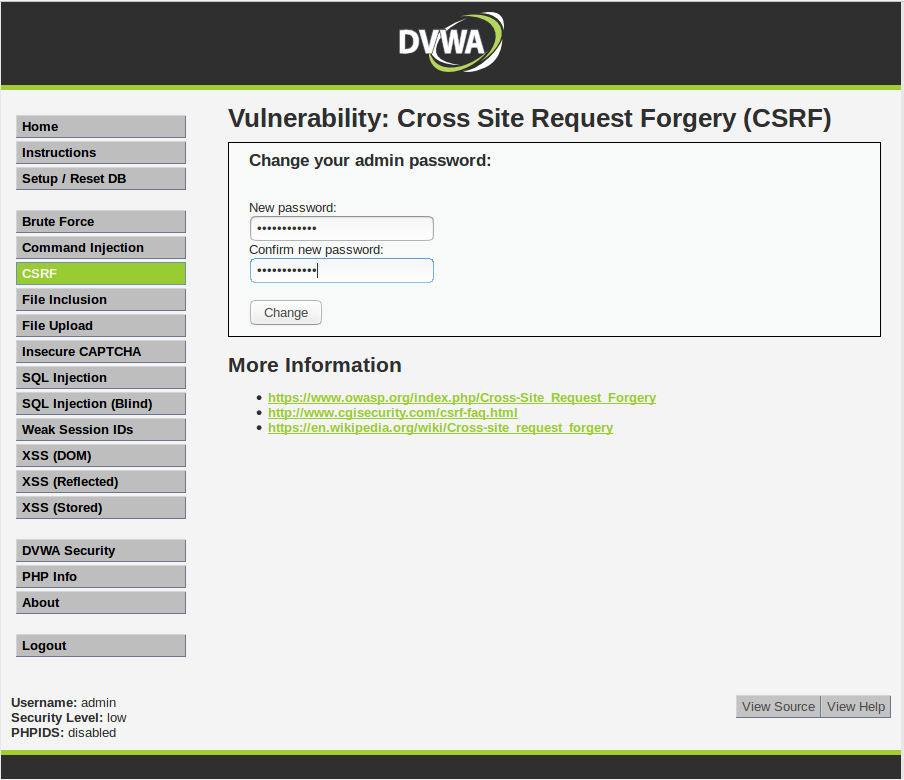
This shows that this attack can range anywhere from a mild inconvenience to a dangerous security breach depending on the complexity and/or intention of the bad actor taking advantage of this vulnerability.

**Task 2:** Show a hacking scenario using CSRF technique.

Using ZAP and DVWA, we were able to simulate a CSRF attack on a user. This attack could change the user’s password to a value known by the attacker and the attacker would be able to log into the application as the attacked user.

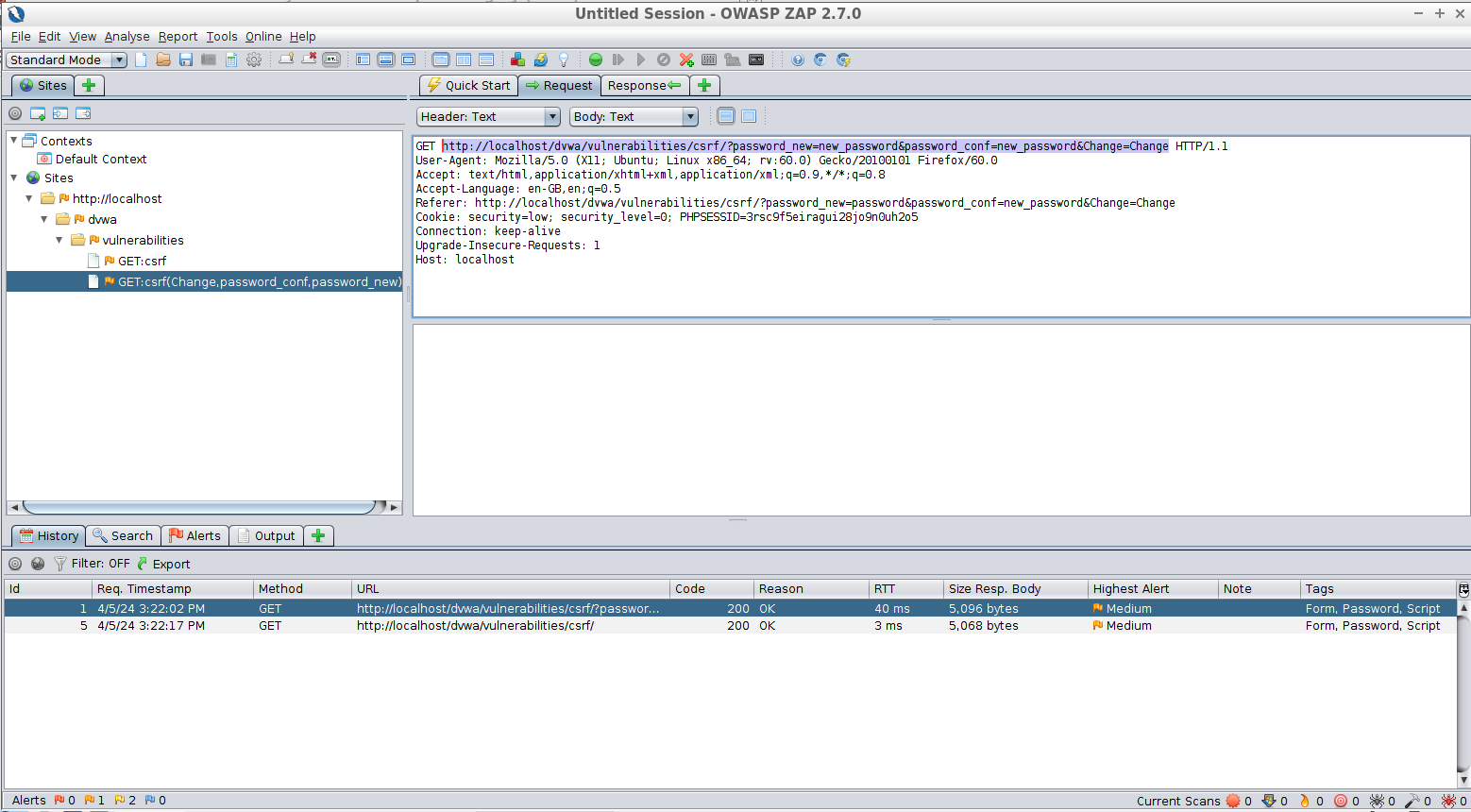
**Uncovering the CSRF vulnerability**

Using the CSRF page, a user can change their password by inputting a new password and then confirming that new password. We used the password “new\_password” in this case.



*Figure 1.6 - Searching for CSRF vulnerability by testing a password change request in DVWA.*

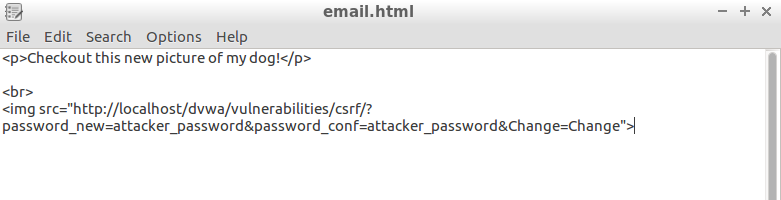
This HTTP conversation between the client and the server was captured via ZAP. It appears to be a GET request. The URL of this request was noted.



*Figure 1.7 - Analyzing the change password GET request for DVWA and taking note of the URL.*

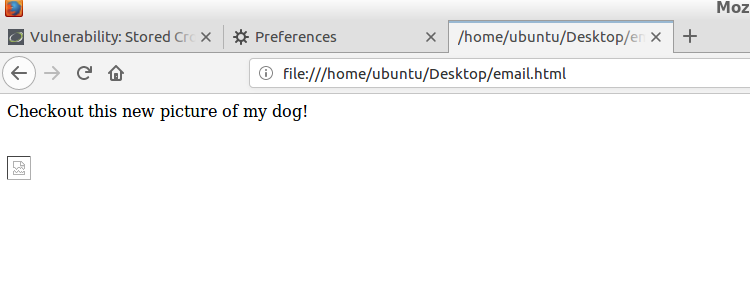
**Compromising the web application**

The URL was changed so that the new password arguments in the URL will be “attacker\_password” so that we (the attackers) can log in with that password once the attack is complete. This new URL was put in the src attribute of an image in an email to a user we could attack.



*Figure 1.8 - Crafting an email with an image referencing the malicious URL we created.*

This email can now be sent to the user whose password we want to change. To them it will look like this:



*Figure 1.9 - Malicious email from the perspective of the attacked user.*

The email provider (gmail/outlook/etc.) will attempt to download the image by calling the GET request that we put as the image source. The GET request won’t return an image. If the user is logged in while viewing this email, this request will reset their password without their knowledge, and we (the attackers) can login with their username and the new password “attacker\_password”.

**Task 3:** Based on the found vulnerable issues, what should you do to prevent attackers from hacking your web application?

**Task 3.1:**

The scenario from task 1 exposes a Stored XSS vulnerability. To mitigate such attacks, key changes need to be made to the application.

First, implement robust input validation and sanitization. Don't accept any data blindly. Instead, use techniques like regular expressions or limit the characters users can enter. Even if the data seems valid, sanitize it before storing it. This involves removing or escaping special characters that could be interpreted as malicious code. Secondly, focus on output encoding. Whenever there is display user input on the application, encode it using the appropriate method based on the context such as HTML, JavaScript, etc. This will ensure the browser interprets the data as content, not code. Finally, the use of prepared statements for database queries that involve user input. These pre-defined SQL statements separate data from the actual query, preventing malicious code injection.

By implementing these mitigations alongside additional security measures, you can significantly reduce the risk of stored XSS vulnerabilities in the application.

**Task 3.2:** Protection/prevention from a CSRF hacking based on the uncovered vulnerabilities in Task 2

To safeguard against CSRF attacks, it's crucial to adopt a multi-layered security approach. Firstly, integrate unique CSRF tokens with each form submission, ensuring they are generated server-side and validated upon submission to thwart unauthorized requests. Utilize appropriate HTTP methods such as POST for sensitive operations like password changes, as they are more resistant to CSRF attacks compared to GET requests. Additionally, implement server-side validation of the HTTP Referer header to verify requests' origins, adding an extra layer of protection against malicious requests originating from unauthorized sources.

Enhance security by setting SameSite attributes on cookies to 'Strict' or 'Lax' to prevent them from being sent in cross-site requests, mitigating the risk of CSRF attacks leveraging stolen session identifiers. Enforce session timeouts and re-authentication mechanisms for critical actions, reducing the window of opportunity for attackers to exploit compromised sessions. Employ Content Security Policy (CSP) headers to control the loading of external resources, limiting potential vectors for attacks such as injection of malicious scripts or content.

**Team peer-review table**

|  |  |
| --- | --- |
| Name | Contributing Efforts in this project (0 ~ 100%) |
| Josh Fishman | 100% |
| Sushruti Bansod | 100% |
|  |  |
|  |  |