Cross-Site Scripting (XSS) Attack

Asif Azad
Bangladesh University of Engineering and Technology

February 15, 2024

Contents

1 Introduction

Cross-Site Scripting (XSS) vulnerabilities are security flaws in web applications that allow attackers to inject malicious scripts into web pages viewed by other users. These scripts can steal user data, such as session cookies, and perform actions on behalf of the victims, compromising the integrity and confidentiality of user interactions. Understanding and mitigating XSS is crucial for web security, as it protects users from identity theft, data breaches, and unauthorized access to sensitive information. By securing web applications against XSS attacks, developers uphold user trust and compliance with data protection regulations, ensuring a safer web ecosystem.

2 Environment Setup

To ensure a controlled and reproducible environment for exploring Cross-Site Scripting (XSS) vulnerabilities, a virtualized setup was used. The setup process involved the following steps:

1. A virtual machine (VM) was created using **VirtualBox**, running **Ubuntu 20.04**. The VM was configured with **1 CPU** and **2GB of RAM**, adhering to the recommended specifications for running lightweight web applications and docker containers without significant performance issues.

2. The setup of the VM followed the guidelines provided by the SEED Labs project. Detailed instructions can be found at `https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md`.

3. After logging into the Seed Ubuntu VM, the following entry was added to the `/etc/hosts` file to simulate the domain name of the vulnerable web application used in the lab exercises. This step required root privileges:

   ```
   10.9.0.5 www.seed-server.com
   ```

4. The lab setup files were downloaded to the VM using the `wget` command. These files include docker containers configured to simulate the web application susceptible to XSS attacks. The following commands were executed to set up the environment:

   ```
   wget https://seedsecuritylabs.org/Labs_20.04/Files/Web_XSS_Elgg/La
   unzip Labsetup.zip
   cd Labsetup
   docker-compose build
   docker-compose up
   ```

This environment setup was critical for performing the XSS vulnerability exploitation and testing mitigation strategies in a safe, isolated manner.

3 Task 1: Becoming the Victim's Friend

3.1 Objective

In this task, you need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser without the intervention of the attacker. The objective of the attack is to add Samy as a friend to the victim (Alice).

3.2 Methodology

1. From Samy's profile, I sent a friend request to Charile to observe the GET request sent to the back-end.

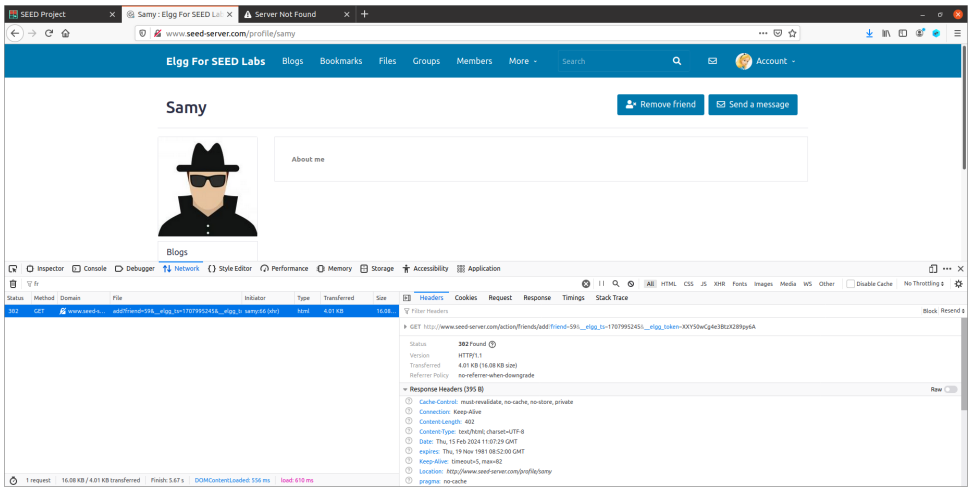

Figure 1: Screenshot GET request for friend request

2. I wrote the js code in script tag to clone the same GET request and saved it in description in Samy's profile.

3.3 Before Attack

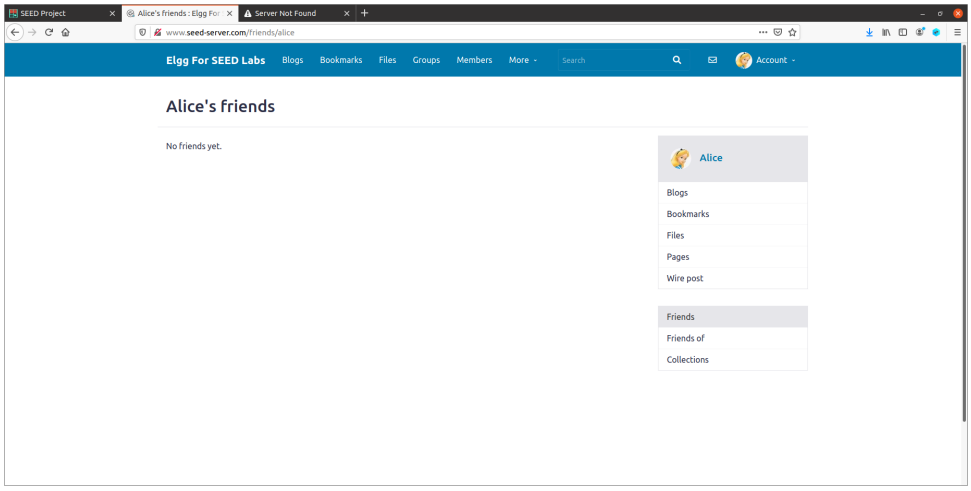

Figure 2: Screenshot of Alice's friend-list before visiting Samy's Profile

3.4 After Attack

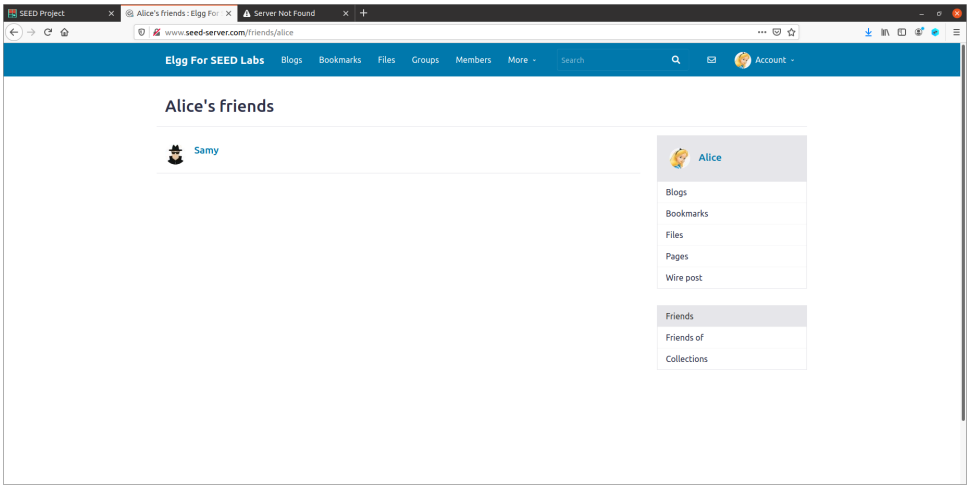

Figure 3: Screenshot of Alice's friend-list after visiting Samy's Profile

4 Task 2: Modifying the Victim's Profile

4.1 Objective

The objective of this task is to modify the victim's profile when the victim visits Samy's profile.

4.2 Methodology

1. From Samy's profile, I edited Samy's profile information to observe the POST request sent to the back-end.

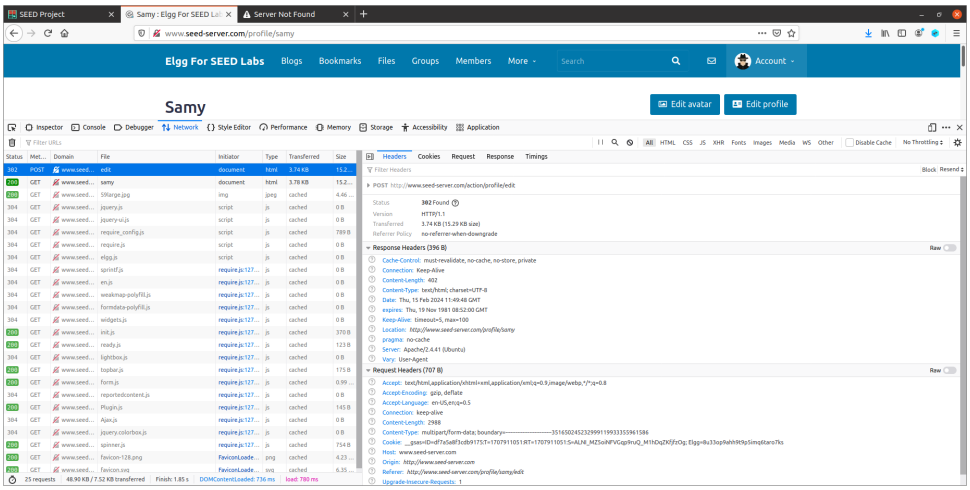

Figure 4: Screenshot of POST request of updating profile information

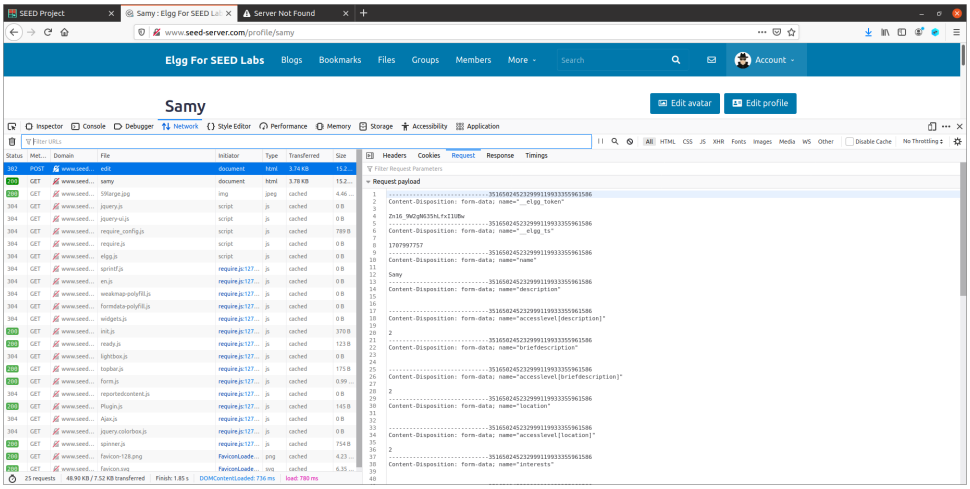

Figure 5: Screenshot of POST request body of updating profile information

2. I wrote the js code in script tag to clone the same POST request and saved it in description in Samy's profile.

4.3 Before Attack

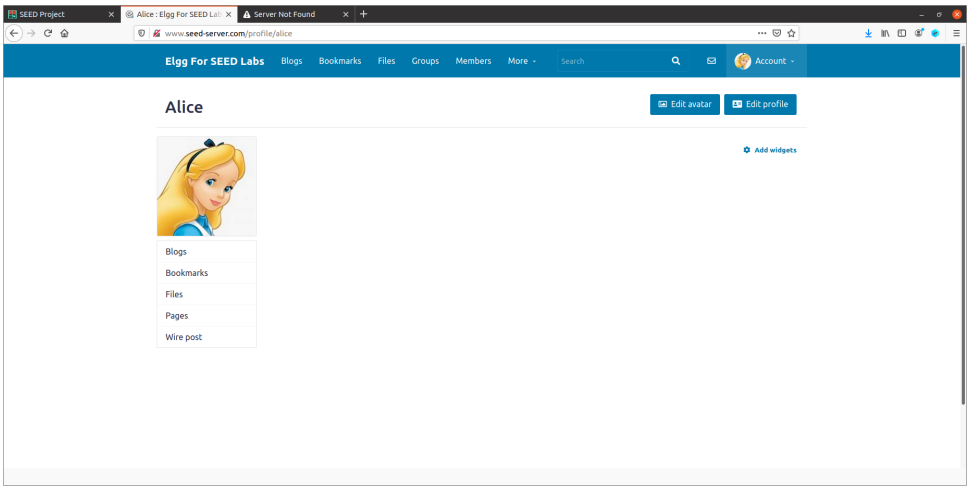

Figure 6: Screenshot of Alice's profile information before visiting Samy's profile

4.4 After Attack

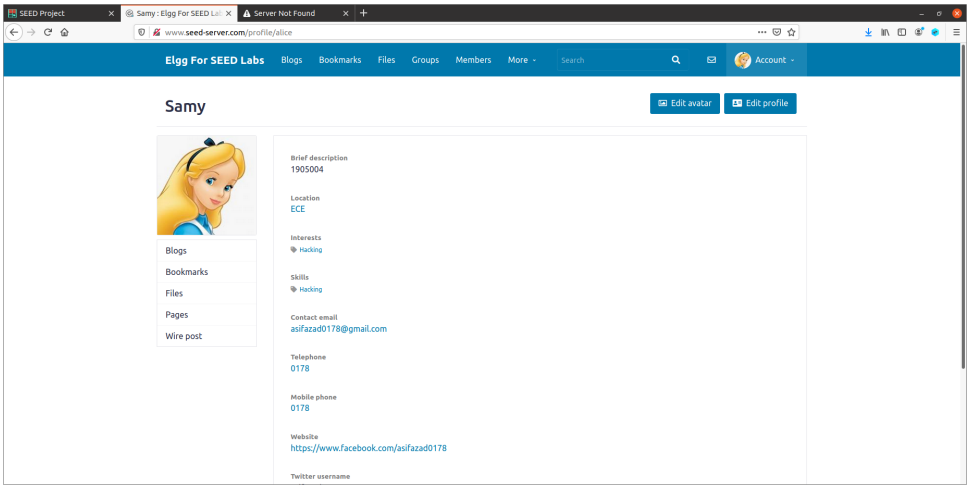

Figure 7: Screenshot of Alice's profile information before visiting Samy's profile

5 Task 3: Posting on the Wire on Behalf of the Victim

5.1 Objective

The objective of this task is to post a customized wire on victim's profile when the victim visits Samy's profile.

5.2 Methodology

1. From Samy's wire, I posted a wire to observe the POST request sent to the backend.

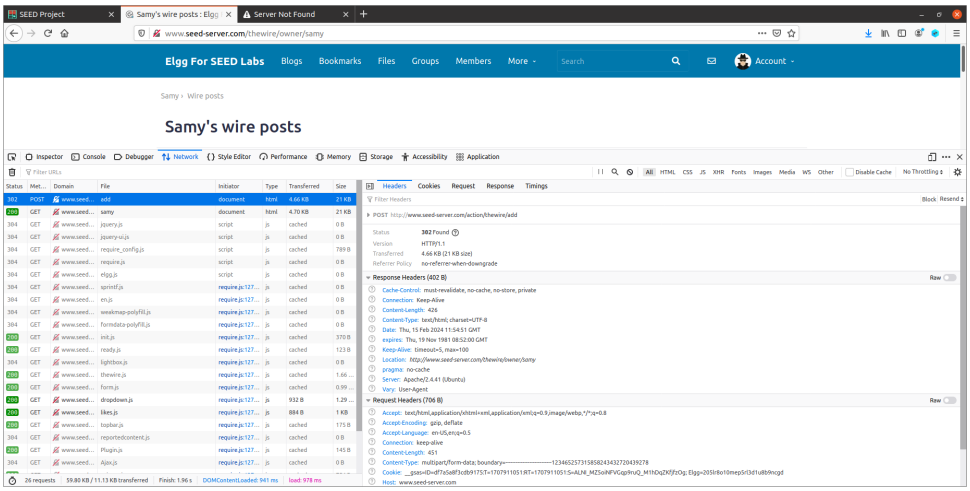

Figure 8: Screenshot of POST request of posting a wire

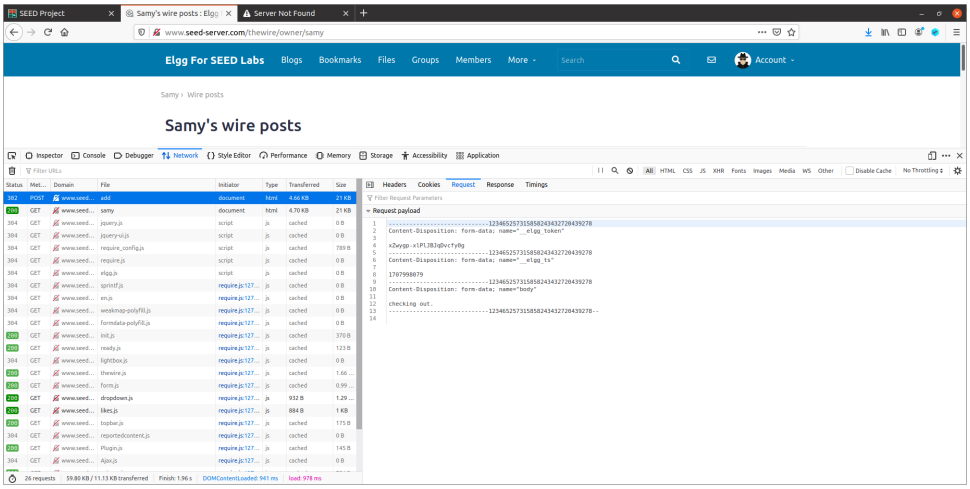

Figure 9: Screenshot of POST request body of posting a wire

2. I wrote the js code in script tag to clone the same POST request and saved it in description in Samy's profile. The *body* field was provided according to the assignment specification.

5.3 Before Attack

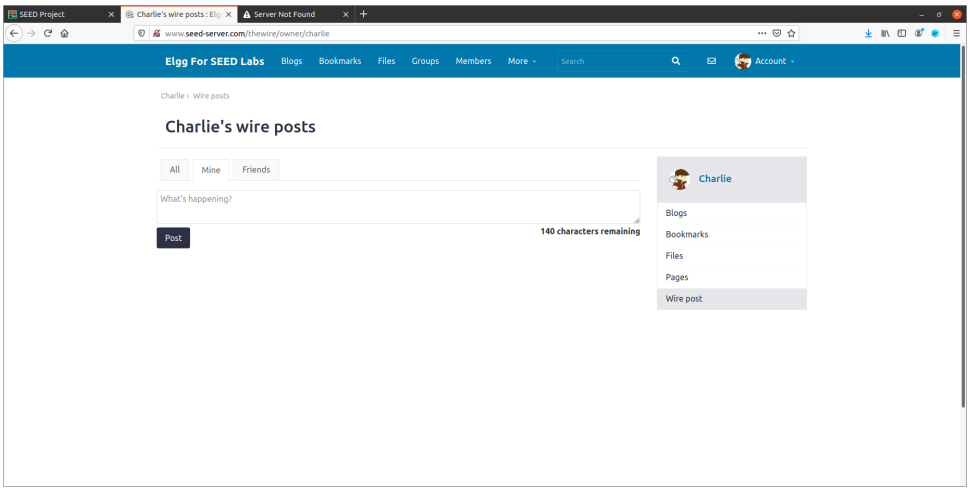

Figure 10: Screenshot of Charlie's wire posts before visiting Samy's profile

5.4 After Attack

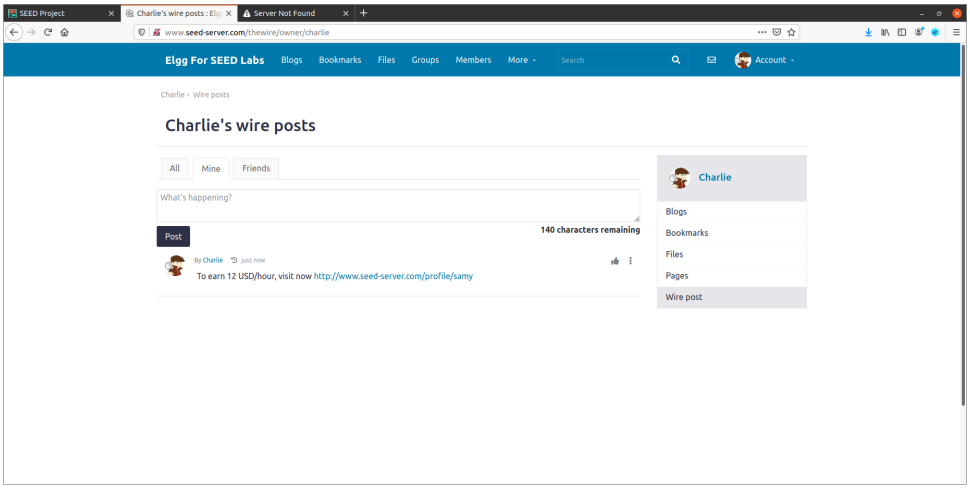

Figure 11: Screenshot of Charlie's wire posts after visiting Samy's profile

6 Task 4: Designing a Self-Propagating Worm

6.1 Objective

To become a real worm, the malicious JavaScript program should be able to propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, but the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles.

6.2 Methodology

1. For this attack, I just incorporated previous 3 attackts in one single script.

6.3 Before Attack

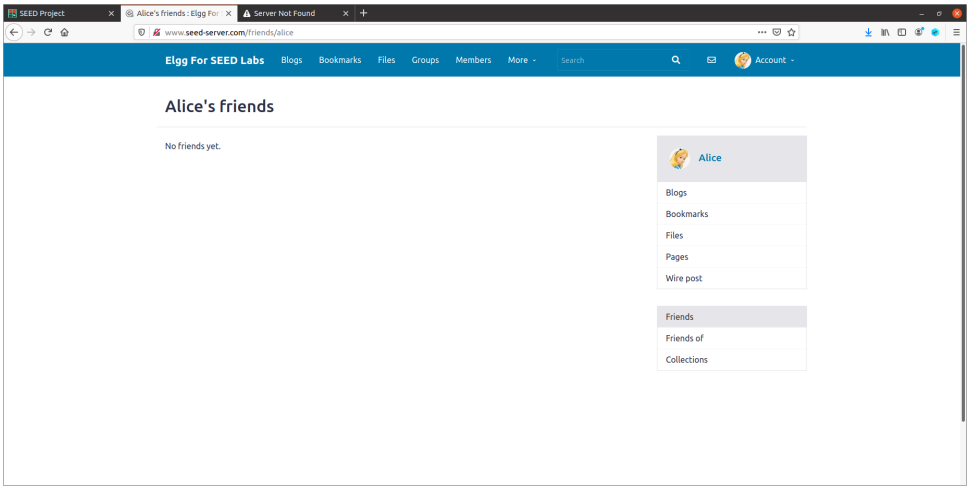

Figure 12: Screenshot of Alice's friend-list before visiting Samy's profile

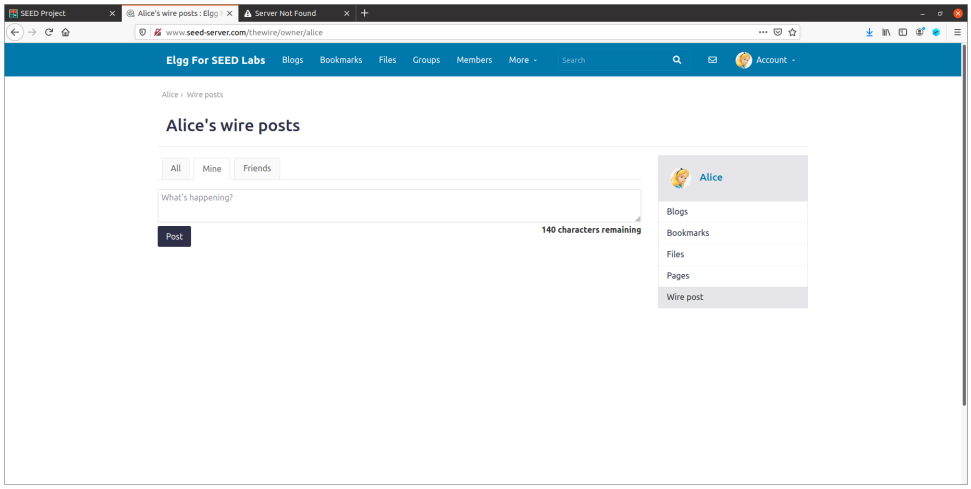

Figure 13: Screenshot of Alice's wire posts before visiting Samy's profile

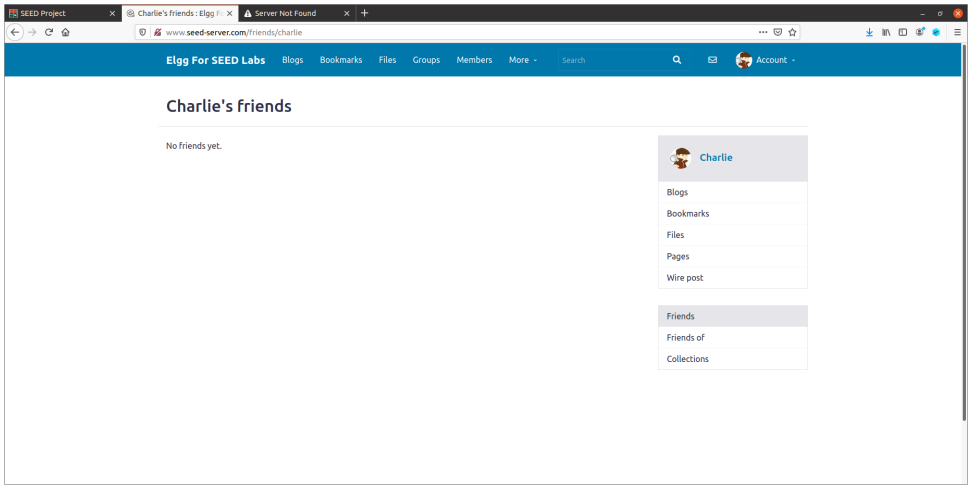

Figure 14: Screenshot of Charlie's friend-list before visiting Alice's profile

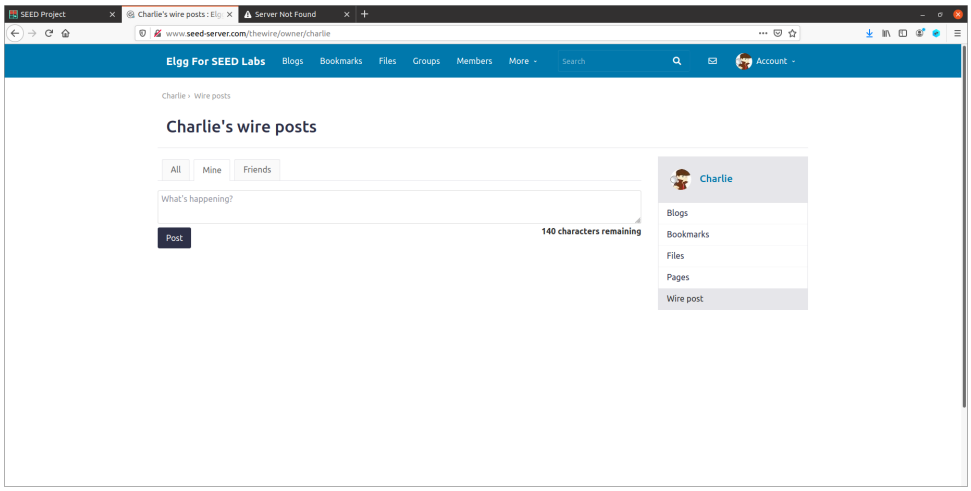

Figure 15: Screenshot of Charlie's wire posts before visiting Alice's profile

6.4 After Attack

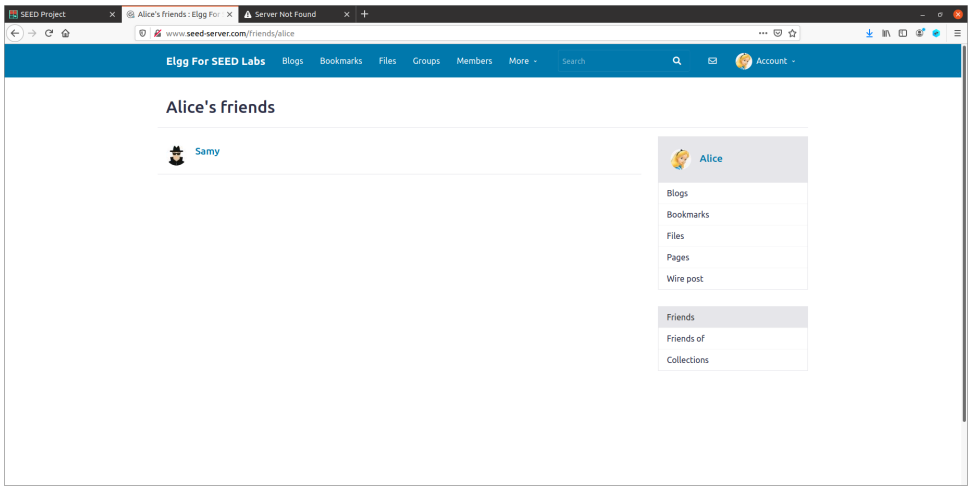

Figure 16: Screenshot of Alice's friend-list after visiting Samy's profile

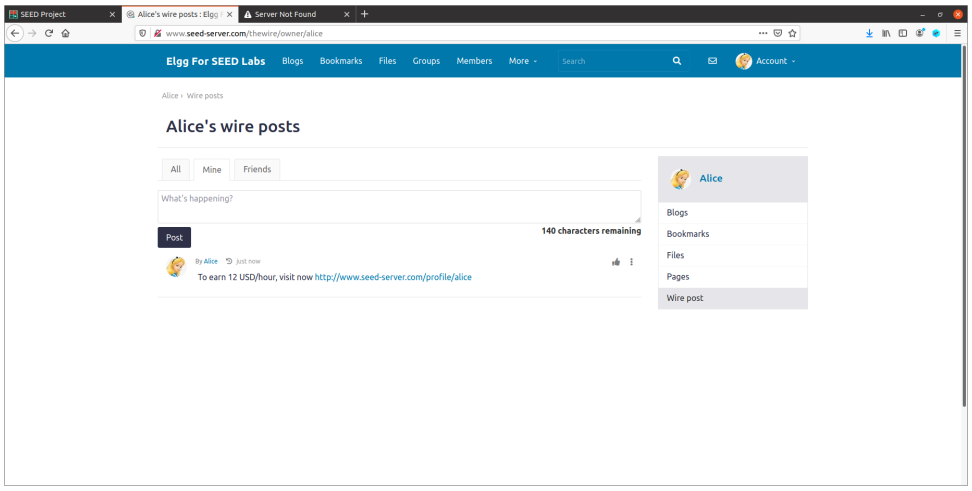

Figure 17: Screenshot of Alice's wire posts after visiting Samy's profile

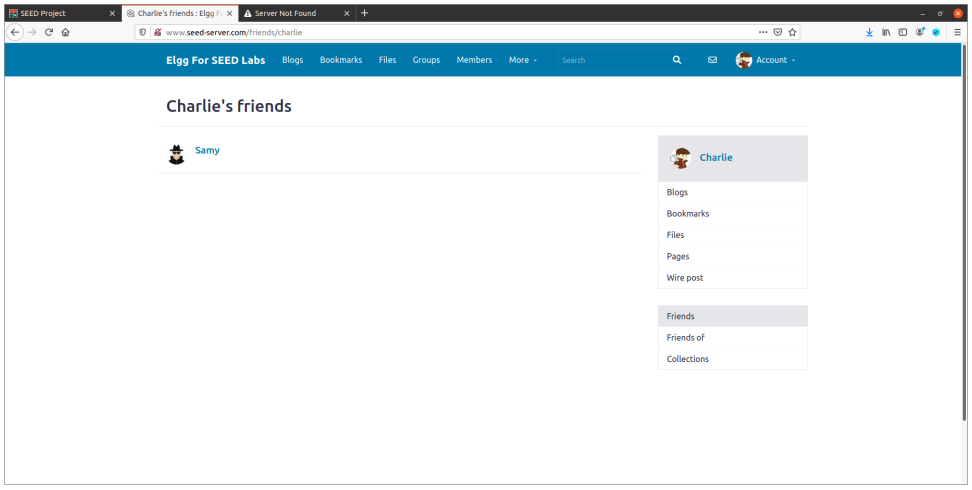

Figure 18: Screenshot of Charlie's friend-list after visiting Alice's profile

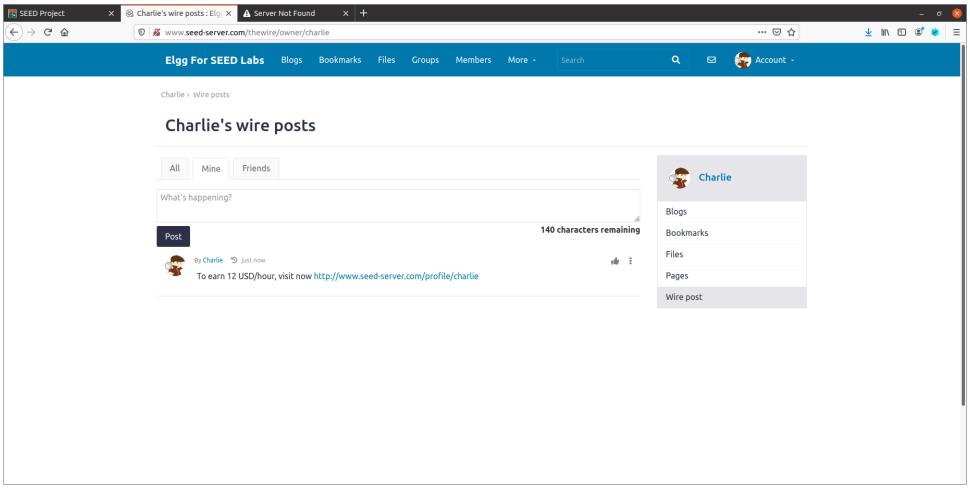

Figure 19: Screenshot of Charlie's wire posts after visiting Alice's profile

7 Mitigation Strategies

To mitigate Cross-Site Scripting (XSS) vulnerabilities, web developers must employ a comprehensive security strategy that includes both proactive and reactive measures. Key strategies include:

- **Input Validation:** Ensure that all input received from users is validated for type, length, format, and range. Use whitelisting wherever possible, allowing only known good input.
- **Output Encoding:** Encode data before it is output to the browser, converting potentially dangerous characters into their HTML entity equivalents. This prevents the browser from executing the data as code.
- **Use of Content Security Policy (CSP):** Implement CSP headers to restrict the sources from which scripts can be loaded. This can prevent the execution of unauthorized scripts.
- **Sanitize Data:** Use libraries and frameworks that automatically sanitize data, removing or encoding characters that can trigger XSS.
- **Regular Security Audits:** Conduct regular security audits and penetration testing to identify and remediate potential vulnerabilities.

By incorporating these strategies into the development and maintenance processes, developers can significantly reduce the risk of XSS attacks and enhance the security of their web applications.