Exercise 9

Download exercise 9 from our website (https://course.cs.tau.ac.il/infosec16/exercises), run ex_unpack on the exercise file and a target directory, and the specified file should be extracted into the specified directory with the q1/, q2/, q3/ and q4/ directories, each of which has a server/ subdirectory, and the pack.py script.

Questions

In all the questions, you will have to run a simple web server with sudo python server in that question's directory (the servers listen on ports 80 and 443, thus the sudo). Then, you'll have to open a browser (either on the InfoSec VM itself, or on the host - it might work faster) and surf to the VM's IP address (for example, http://192.168.62.142 or https://192.168.62.142).

The website is a simple chat, with the following users:

<table>
<thead>
<tr>
<th>username</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>ComplicatedP4ssw0rd!</td>
</tr>
<tr>
<td>bob</td>
<td>@c0mpl1catedPASSWORD</td>
</tr>
<tr>
<td>mallory</td>
<td>1234</td>
</tr>
</tbody>
</table>

You should only log in as Mallory, except when testing that your hacks indeed work on Alice and Bob as expected. Once logged in, you can submit a new message, and it should be saved and posted. To log out, click the link at the beginning of the page; to reset the website to its initial state, surf to http://IP/reset.

If you have problems running the server because the port is already in use, try running sudo service apache2 stop.
1. Open the browser (I recommend Chrome) with the security settings disabled, so no SOP is enforced. On Linux, you can do so with `chromium-browser --disable-web-security`, and on Windows with "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security --user-data-dir=C:\tmp (or wherever else your Chrome is installed; also, make sure to delete the C:\tmp directory afterwards).

Don't do anything other than this exercise with the browser open in this mode!

In this question, you'll implement a simple CSRF. Skim over bottle's tutorial, to get a sense of how to write a simple web server in Python, and use the q1.py template to get you started.

The idea is this: Alice surfs to http://IP/, logs in, and views the chat at her leisure. Then, you send her a link to your own website, at http://IP:8000/ (in real life that would be a different IP/domain entirely, of course). Alice naïvely surfs to that website, while logged into the simple chat, and there your malicious JS forges a chat message submission request from Alice. With SOP turned off, just by browsing your website while logged in to the chat, a message gets posted by Alice!

Submit a script q1.py that serves a simple webpage under construction (preferably, based on the q1.py template), but in fact does a CSRF so whoever is logged into the chat posts the message "Hacked!", and describe your solution in q1.txt.

Notes
- To understand how to post messages, etc. you can view the server source code, it's quite simple.
- When you refer to the server in JS, don't hardcode your IP (ours may be different); instead, derive it dynamically, like with `window.location.hostname`.
Here’s what should happen: you run the server, surf to it with SOP disabled, and log in as Alice; then, you run your own website, surf to it as Alice, and just like that, a message from Alice should be posted!

Now open the browser in its standard mode, and see for yourself that CSRF stops working; in fact, when Alice is viewing your malicious website, you can open the developer tools, go to the console tab, and see an explicit error about it there!
So, instead, log in as Mallory, and post a comment with XSS that will cause Alice (and Alice only!) to post the message "Hacked!" once she views that comment. Add this comment's content and an explanation about it in q1.txt.
Here’s what should happen: you log in as Mallory, post a `<script>... </script>` comment, and log out; you then log in as Alice, view the comments (Mallory’s should look empty, but execute the script!), and if you now refresh the page (or log out and log in as Mallory again), you should see the JS executed successfully and posted a "Hacked!" message from Alice.
2. In this question, the server was fixed to escape user input; but luckily, we go to the same Coffeehouse as Alice, and connect to the same WiFi network, so we’re in the same LAN and can sniff her traffic! Write a script q2.py which uses scapy to sniff HTTP requests of users that are logged into the chat, and prints their username and password for each such request (you can assume there’s only one cookie for this domain in your parsing). The output format should be:

```
username: mallory
password: 1234
username: alice
password: ComplicatedP4ssw0rd!
```

Describe your solution in q2.txt, and explain what the problem with the cookie's format is. But even if we change the cookie format, would it really protect the users’ credentials? Find out and explain why not.

3. This time, the cookie was cleverly encrypted; and let’s ignore for a moment that other problem we discussed in the previous question. Would that encryption stop us from stealing the cookie and impersonating Alice?

Write a script q3.py which uses scapy to sniff HTTP requests of users that are logged into the chat, like in the previous question, and prints their cookie value for each such request (again, you can assume there’s only one cookie for this domain in your parsing). The output format should be:

```
cookie: !KuwqbG8+s0fB+7dctttt88Q==?
# Here I log out as Mallory,
cookie: !6qOjx7VS9Lj4s2nPMiebHQ==?
# And here I log in as Alice
```

Describe your solution in q3.txt, and explain how you can log in as Mallory, and then use JS to change your cookie to Alice’s stolen one, refresh the page, and post messages from Alice!
4. Finally, the server developers realized they have to use HTTPS. However, they didn't want to pay for a certificate, so they decided to use a self-signed one instead. True, when you log into the site it warns you the certificate can't be validated, but at least the traffic is encrypted and the cookies can't be easily stolen.

Write a script `q4.py` which runs a webpage similar to the login page of the chat, but on port 8000 instead of 443 (in real life, we'd use some UI redressing, like calling it Imageshack instead of imageshack).

This webpage should also use a self-signed certificate, generated with OpenSSL:

```
openssl req -new -x509 -nodes -newkey rsa:2048 -keyout server.key -out server.crt
```

after all, who knows the difference between self-signed certificates anyway? And when a username and a password are inserted, it should write them to a `users.txt` file in the previous format:

```
username: alice
password: ComplicatedP4ssw0rd!
```

It should then redirect the request to the original server, so the user doesn't suspect a thing! Describe your solution in `q4.txt`.

You can copy code from the original server (namely, you'll need the SSL part at the bottom, and an index function that returns the HTML from `views/login.tpl`). However, don't distribute your code over multiple files - it's small enough to embed entirely in `q4.py`! The only files you can have are `q4.py` itself, and the self-signed certificate you generated as `server.key` and `server.crt`.

Run `ex_pack` on the exercise directory, enter your student ID, and the specified directory should be compressed into a zip file which you are to submit. It will also run the `pack.py` script, which will make sure all your files are present and valid.