

Systems and Services Security

Assignment 4

Νικολόπουλος Ευάγγελος 2020030040

Φώτη Ερμιόνη 2020030081

Tasks

1. SQL Injection in the login form in order to access the account 'user'

Inside the python code of our application, in the login() method we observed that the following query

```
"SELECT * FROM users WHERE username = 'user' AND password = '{password}'"
```

is being executed upon a login attempt. Therefore we injected the following statement inside the password field on page <http://139.91.71.5:11337/>

' OR '1'='1

so that we break out of the password string and add the above statement so that the resulting query

```
SELECT * FROM users WHERE username = 'user' AND password = " OR '1'='1';
```

always evaluates to true.

2. DOM-Based XSS

Through DOM manipulation (greet.js file) which processes URL hashes without sanitizing them, we injected a “malicious” script through the search bar on the web app by inserting the following address

[http://127.0.0.1:8080/dashboard#%3Cscript%3Ealert\(%27DOM%20XSS%27\)%3C/script%3E](http://127.0.0.1:8080/dashboard#%3Cscript%3Ealert(%27DOM%20XSS%27)%3C/script%3E)

which is essentially the script alert('DOM XSS') encrypted. Once we loaded the page, an alert box showed up on our browser, which meant that our script was executed successfully.

3. Reflected XSS

In the dashboard.html file we observed that payloads are reflected back into the “noitem” block (line 42) without escaping or sanitizing, which would result in the execution of any JavaScript script that would be submitted inside the search bar. So we submitted for execution the following script:

```
<script>alert("")</script>
```

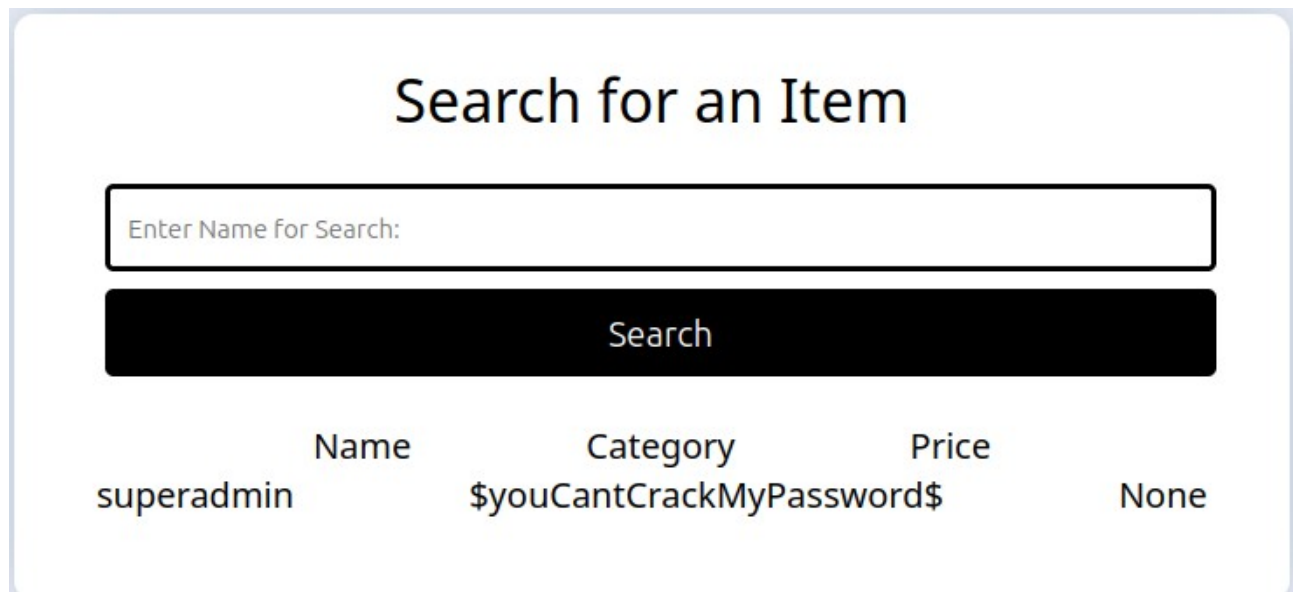
Something notable was that when we placed a message inside the alert box it would cause the web app to crash, while on the other hand an empty message would exhibit the expected behaviour.

4. SQL injection to extract the 'superadmin' password

After logging in as 'user' using the first SQL injection, we used the search function defined inside app.py. But we thought that it would be a good idea to combine the result of the item search with the results of a query on the user table:

```
' UNION SELECT username, password, NULL FROM users WHERE username='superadmin'--
```

The above injection, which we input inside the search bar of the app, “merges” the results of the item search with the results of searching for the (username,password) tuple on the users table with the username being 'superadmin' (Pretty self-explanatory right?). Below is the result we got.



The screenshot shows a web application interface with a search bar and a table of results. The search bar contains the text "Enter Name for Search:". Below the search bar is a black button labeled "Search". The table below the button has three columns: "Name", "Category", and "Price". The first row of the table shows the results of the SQL injection: "superadmin" in the Name column, "\$youCantCrackMyPassword\$" in the Category column, and "None" in the Price column.

Name	Category	Price
superadmin	\$youCantCrackMyPassword\$	None

Extra: If we'd like to know the password of the 'user' account, we would run the following query:

```
' UNION SELECT username, password, NULL FROM users WHERE username='user'--
```

5. Login with the superadmin credentials

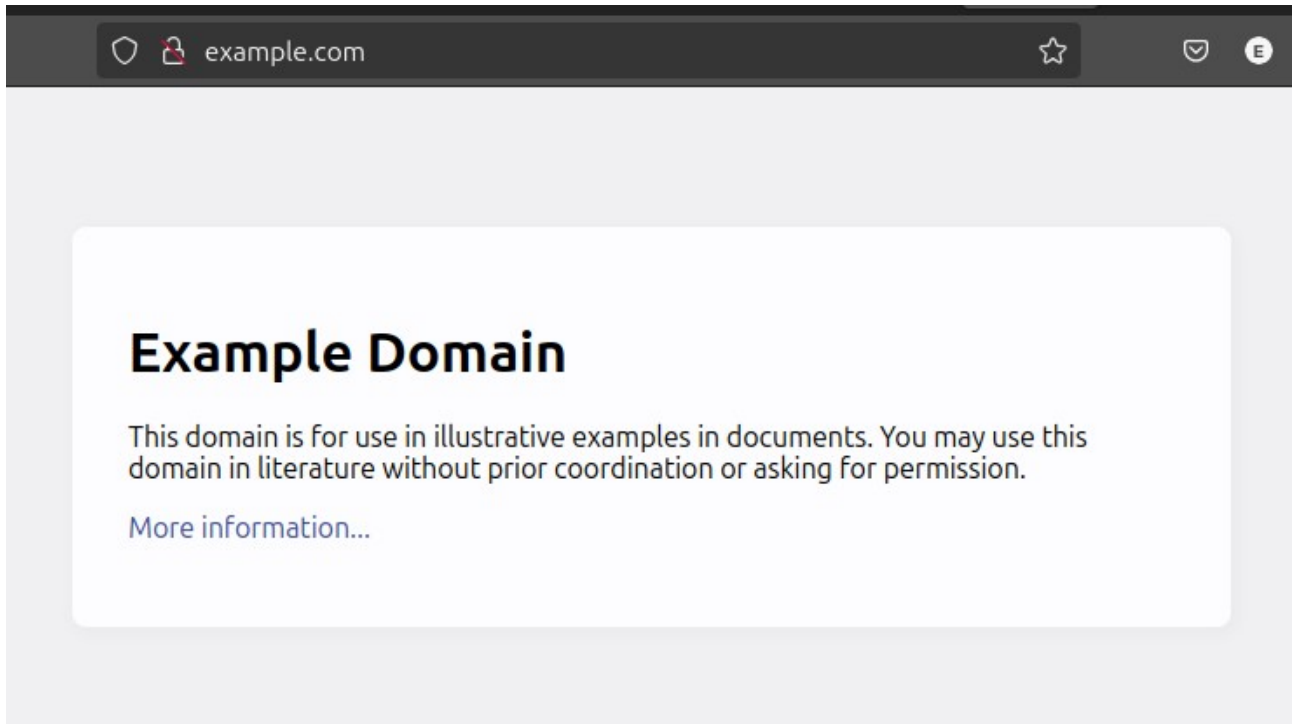
Now that we have acquired the superadmin password, we login via the following page
<http://139.91.71.5:11337/admin>

6. Redirection

The goto() method inside app.py (line 131) uses a parameter 'to' from the URL that isn't validated and enables phishing attacks and exploitation. We confirmed that by inputting in our browser's search bar the following:

<http://139.91.71.5:11337/go?to=http://example.com/>

The result was a redirection without warning



7. Local File Inclusion – Search for the flag

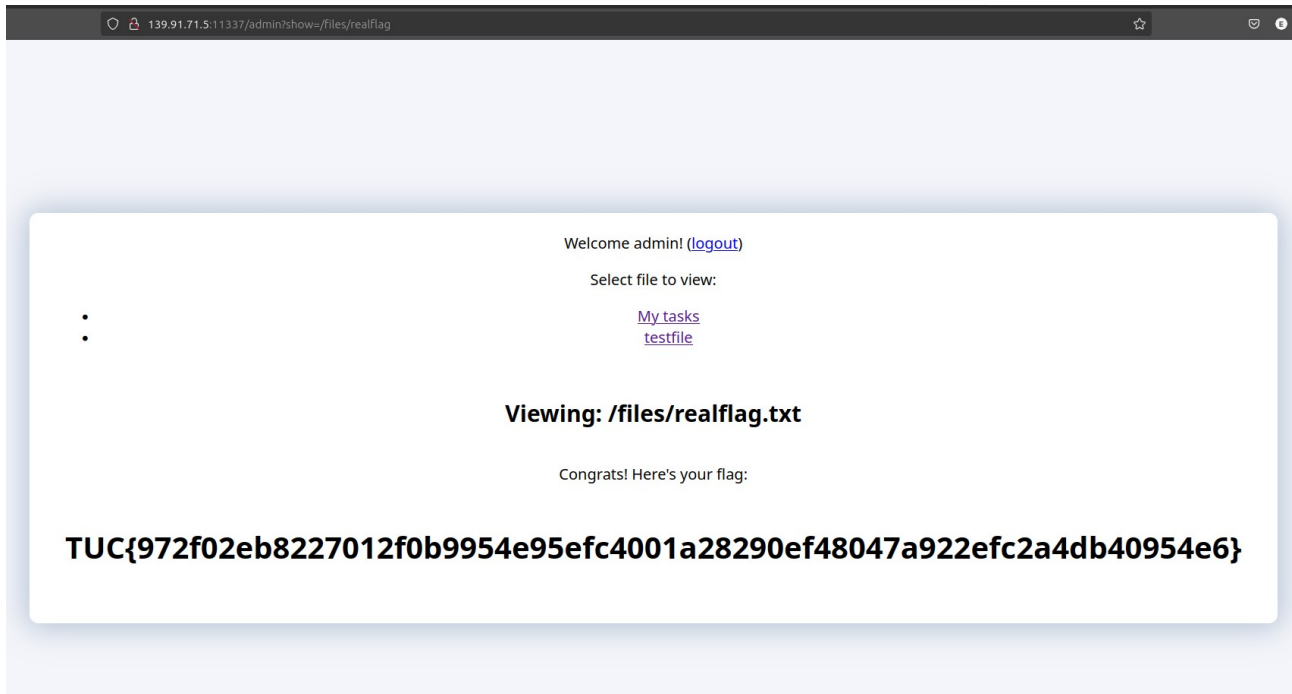
Finally, we observed that the app.py had a function called files(), which would show an index of all files in the APP_ROOT directory if we simply input the following address in our browser:

<http://139.91.71.5:11337/files>



This of course would prove to be very helpful in our CTF quest, since we now knew the name of the flag file and where it was. Using the LFI vulnerability which allows traversal and inclusion of sensitive files, we typed the following address to our browser's search bar which now indicated the location of the hidden flag file.

<http://139.91.71.5:11337/admin?show=/files/realflag>



The secret flag is the following:

TUC{972f02eb8227012f0b9954e95efc4001a28290ef48047a922efc2a4db40954e6}