Systems and Applications Security

Assignment 3

Gialouris Georgios | 2019030063
Emmanouilidis Ioannis | 2019030036

In this assignment we worked with a Web application that contained four distinct vulnerabilities:

- Two separate SQL injection vulnerabilities.
- A reflected cross-site scripting (XSS) vulnerability.
- A DOM-XSS vulnerability.

Our task was to identify these vulnerabilities and make use of them to either gain access as a legitimate user, or inject potentially malicious payloads, as a form of attack.

SQL injection | Logging in as user

For the first part, we were asked to bypass the initial login page using an SQL injection payload and login as ‘user’. To do so, we had to take a look at the corresponding python code that took the password entered by the user and formed the query below to search for a user with that password.

SQL query:
*SELECT * FROM users WHERE username = 'user' AND password = '{password}'*

In order to bypass the login page and therefore manage to login as a user, we had to type in any of the following strings as passwords.

' OR 1 --
' OR TRUE --
whatever' OR TRUE --
' OR 1=1 --

The above inputs basically worked as extensions to the code of the SQL query, changing it to

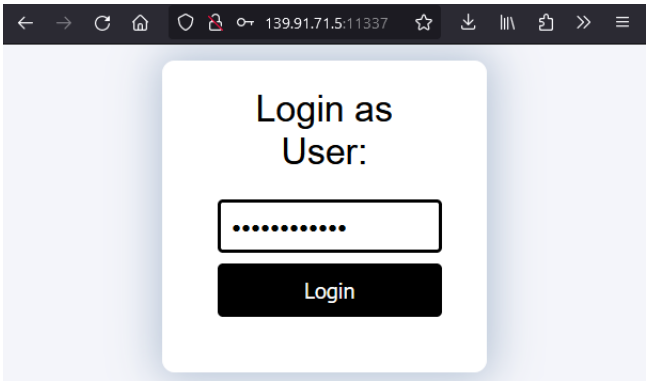
*SELECT * FROM users WHERE username = 'user' AND password = '' OR TRUE --'*

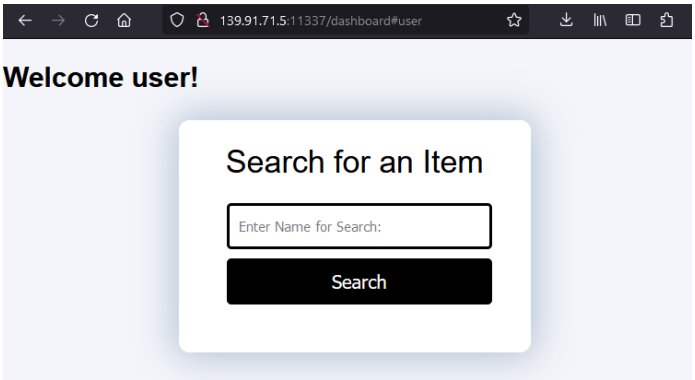
What this does is that it adds a semicolon signifying the end of a string that would normally correspond to a password. The whole ‘WHERE’ condition could be interpreted as

(username = 'user' AND password = '') OR TRUE --'

since the logical operator ‘AND’ holds higher precedence over ‘OR’. As we can see, the first statement is false but the whole statement ends up being true due to the final condition, following ‘OR’, having been set to true. Therefore, this query returns all the rows from the table ‘users’. We then comment out the last semicolon by adding ‘--', so that it gets ignored.

It's worth noting that this type of query (checking username and password directly) is not recommended due to security reasons. Storing passwords directly in the database and using them in queries makes an application vulnerable to SQL injection attacks.





DOM-XSS attack

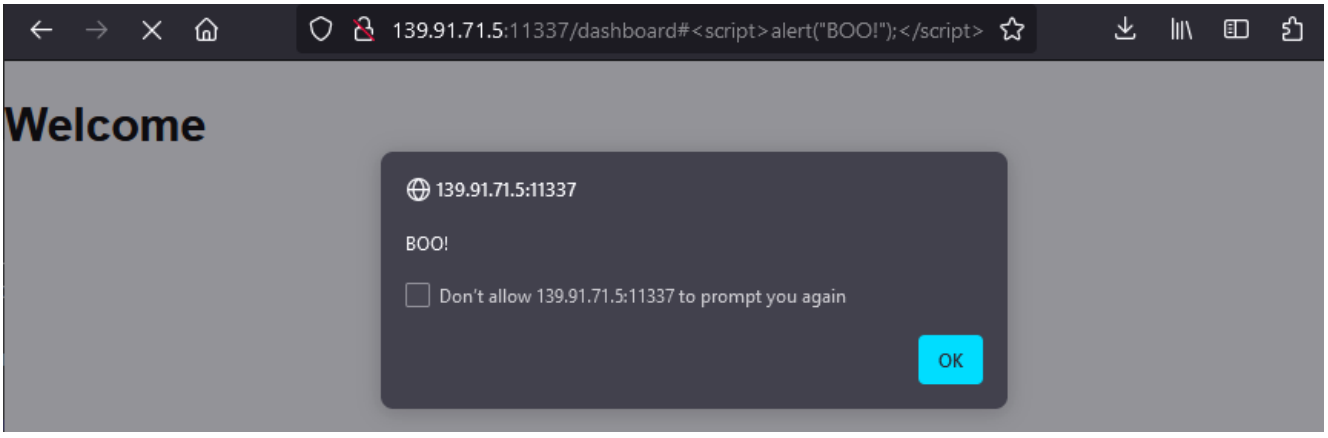
We were then asked to identify and exploit the DOM-XSS vulnerability. The JavaScript code that is used to display a greeting message to the user, assumes that the URL contains a username in the hash portion and displays a greeting using that username if present. The code takes the value from the URL hash and directly inserts it into the document. A malicious user could potentially manipulate the URL hash to inject a script that will be executed as soon as it gets attached to the html code.

The payload for a DOM-based XSS attack would therefore be some JavaScript code appended to the URL's hash portion. Below there's an example of a payload that creates an alert box:

<script>alert("BOO!");</script>

By appending this to the URL, the attacker is able to form a potentially malicious address that could be sent to a user. The user would then visit this URL and their browser would execute the JavaScript code within the hash. The URL would look something like this:

http://139.91.71.5:11337/dashboard#<script>alert("BOO!");</script>



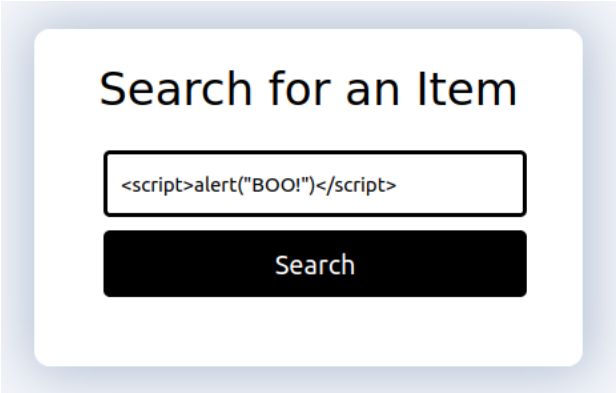
Reflected XSS attack

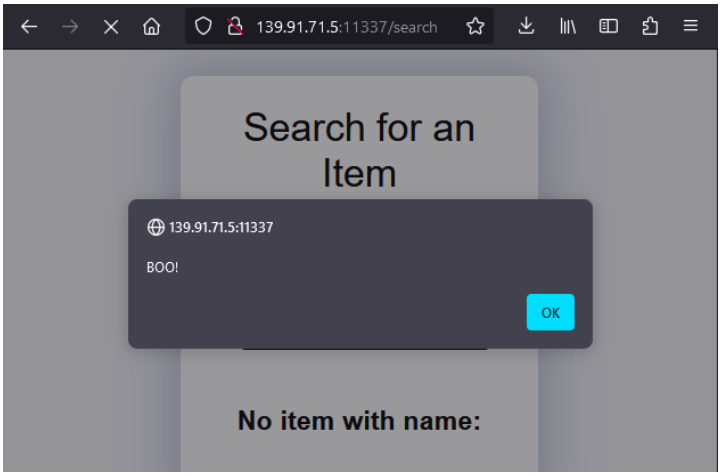
The web application also came with a reflected cross-site scripting (XSS) vulnerability which arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way. In the given application, after the user is logged in, they are asked to search for an item by typing into a form. If the application does not find the requested item, it proceeds to reflect the given input in the website's results by passing it to the HTML code, within heading tags. A malicious user could potentially exploit the above functionality, by typing in a script that gets executed by the browser as it evaluates the corresponding HTML code.

The payload for a Reflected XSS attack would therefore be some JavaScript code provided as input to the search form. Below there's an example of a payload that creates an alert box:

<script>alert("BOO!")</script>

The application ensures that the given input does not contain semicolons, to prevent stacked queries. This also prevents adding multiple JavaScript commands, separated by semicolons.





SQL injection | Acquiring admin's password

In the last part of the assignment, we conducted a second SQL injection, by misusing the item search functionality to retrieve data from the 'users' table and acquire the admin's password. By taking a look at the related SQL query:

SELECT name, category, price FROM items WHERE name = '{name}'

we figured that in order to get the application to display the password, we had to type in one of the following as input:

*' UNION SELECT * FROM users –*

' UNION SELECT username, password, NULL FROM users --

Once again, the above input worked as an extension to the query, changing it to:

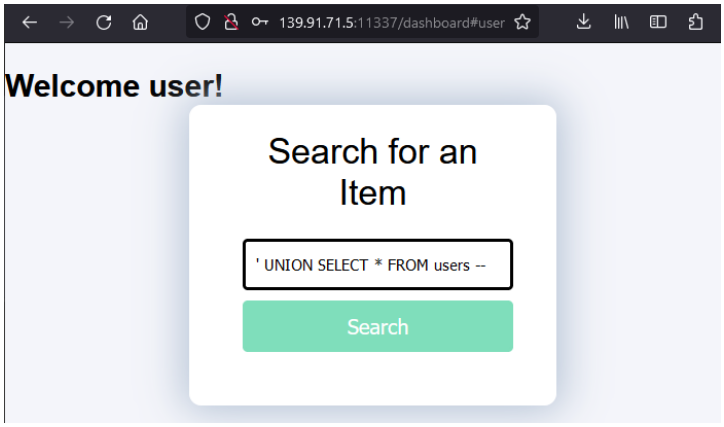
SELECT name, category, price FROM items WHERE name = ''

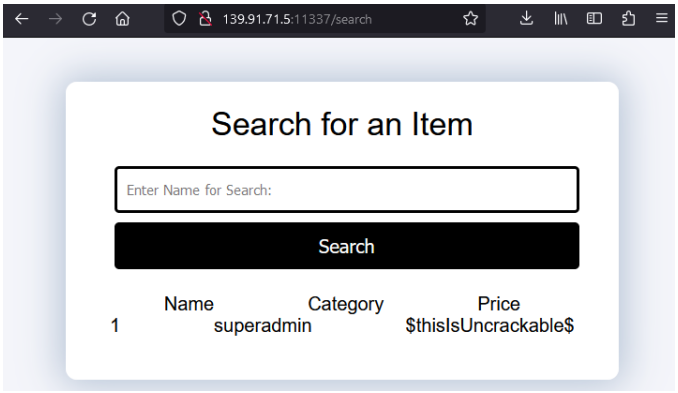
UNION

*SELECT * FROM users -- '*

Similarly, what this does is that it adds a semicolon signifying the end of a string that would normally correspond to an item's name. By combining the above two queries using the 'UNION' operator, we would normally expect our program to combine the three-columned rows of each table. However, the condition of the first query fails, so we only get the

contents of the ‘users’ table. The python code retrieves the two rows and prints the contents of just the first one, which corresponds to the admin’s logging information. We then comment out the last semicolon by adding ‘--', so that it gets ignored.





After we acquired the admin’s password ($thisIsUncrackable$), we proceeded to log in as the administrator of our application and fetched the secret flag:

TUC{1e3bdec35674e4003b24e97a739c606da29e8751fe921a6ca23f23bb67697b60}