XSS Exercise

Security Testing (145322))

Francisco Manuel Colmenar Lamas
219757

Trento, October 2020

Contents

List of Figures

1. WEBGOAT EXERCISES

1.1. XSS Exercise 2

The objective of this exercise is to obtaine the cookie from the browser using two different tabs to know if they are the same.

In order to accomplish it, the command *alert(document.cookie);* is run on both tabs, in the tab from the current exercise and in a new webgoat tab.

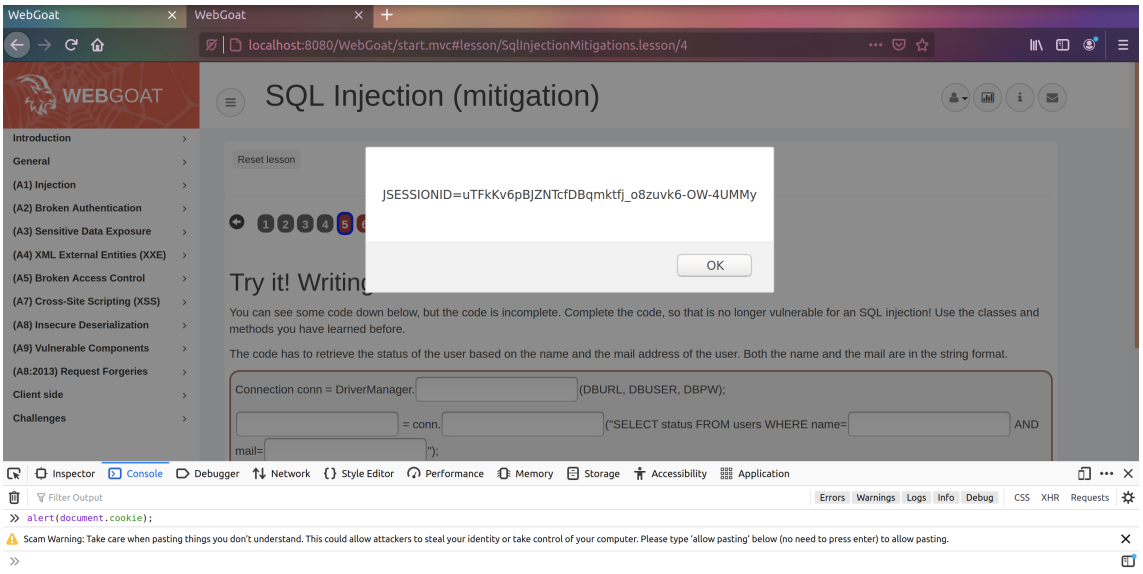

Figure 1.1: Cookie from the new webgoat tab.

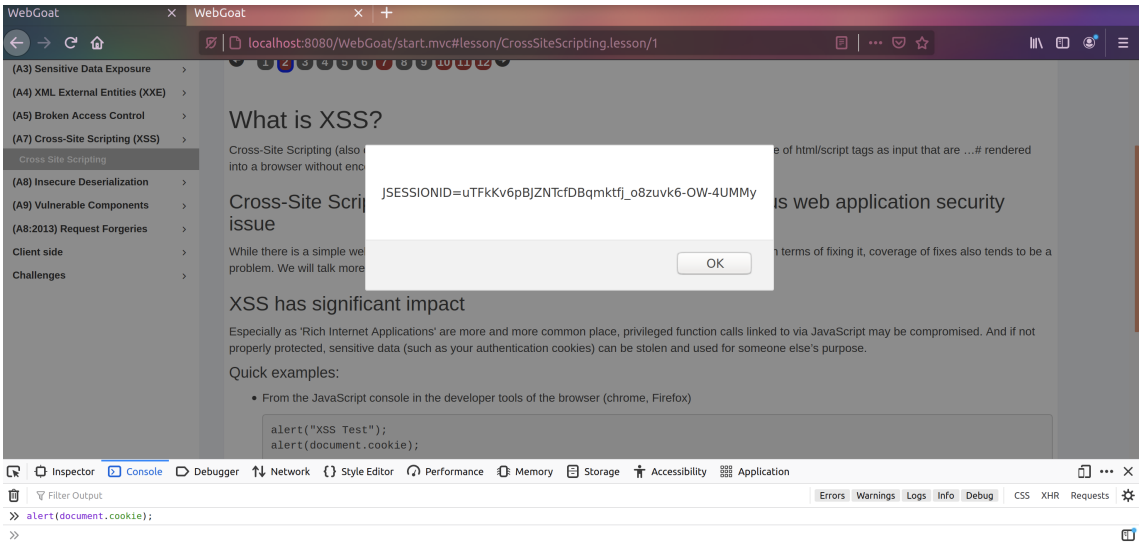

Figure 1.2: Cookie from the tab belonging to the XSS Exercise 2.

As it can be seen in the above pictures, the two cookies are the same.

1.2. XSS Exercise 7

At this exercise, it is asked to discover which is the field vulnerable to XSS. To achieve its goal, the following script is introduced at all the input boxes.

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

Finally, the field which executes the script is *"Enter your credit card number"*, therefore being the field vulnerable.

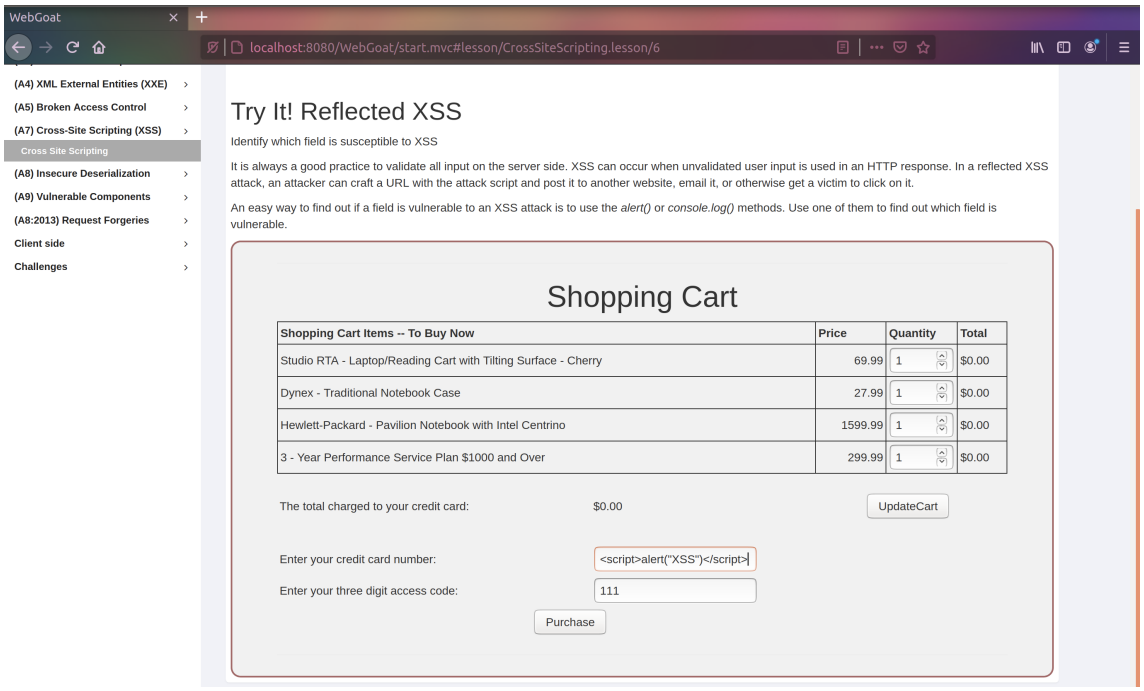

Figure 1.3: Introducing the XSS script into the vulnerable field.

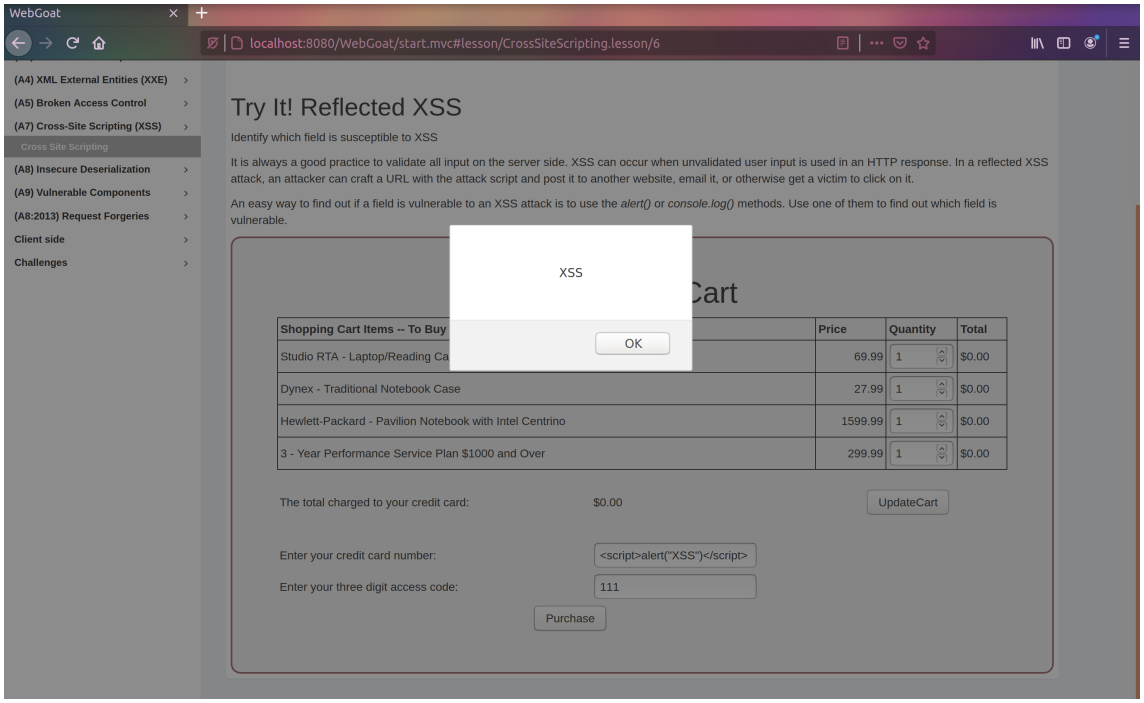

Figure 1.4: Result of the XSS.

1.3. XSS Exercise 10

The aim of this exercise is to find the hidden code inside the javascript code which allows to access the test page left from production. In order to find this piece of code, the

debugger option of the browser is opened and the word "route" is searched among the js files.

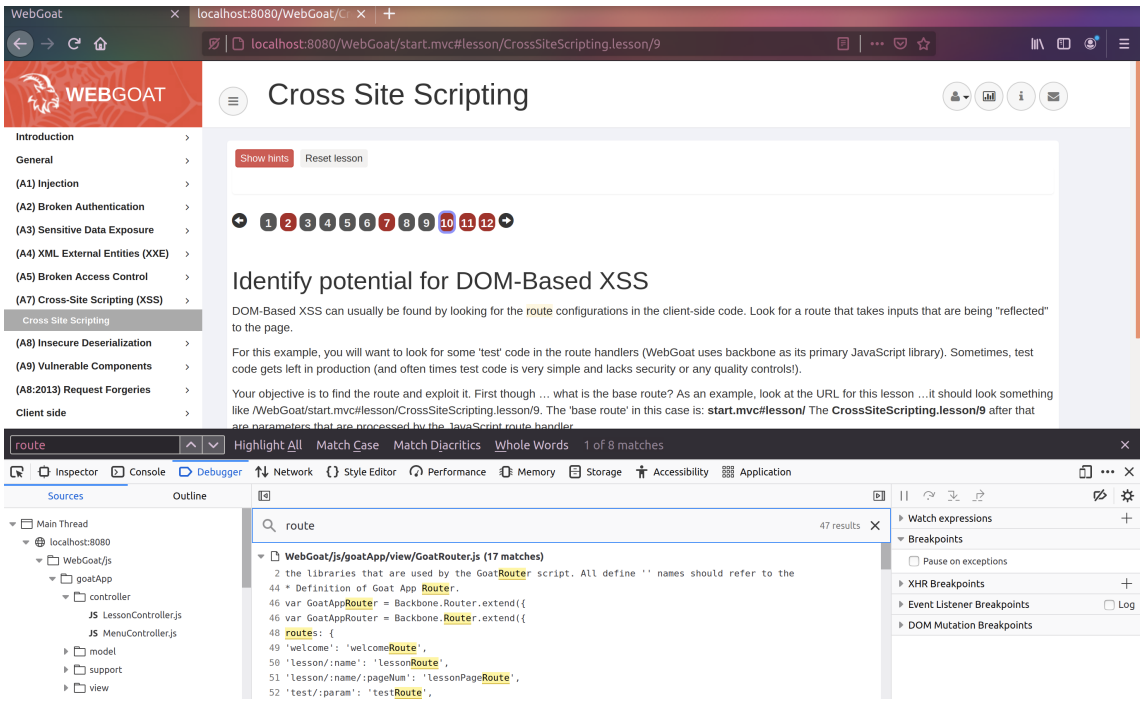

Figure 1.5: Occurrences of the search for *"route"*.

As it can be seen, the search shows a piece of code which contains different routes, among which the *testRoute* is located. Therefore, the path is constructed using this information and introduced in the submit box.

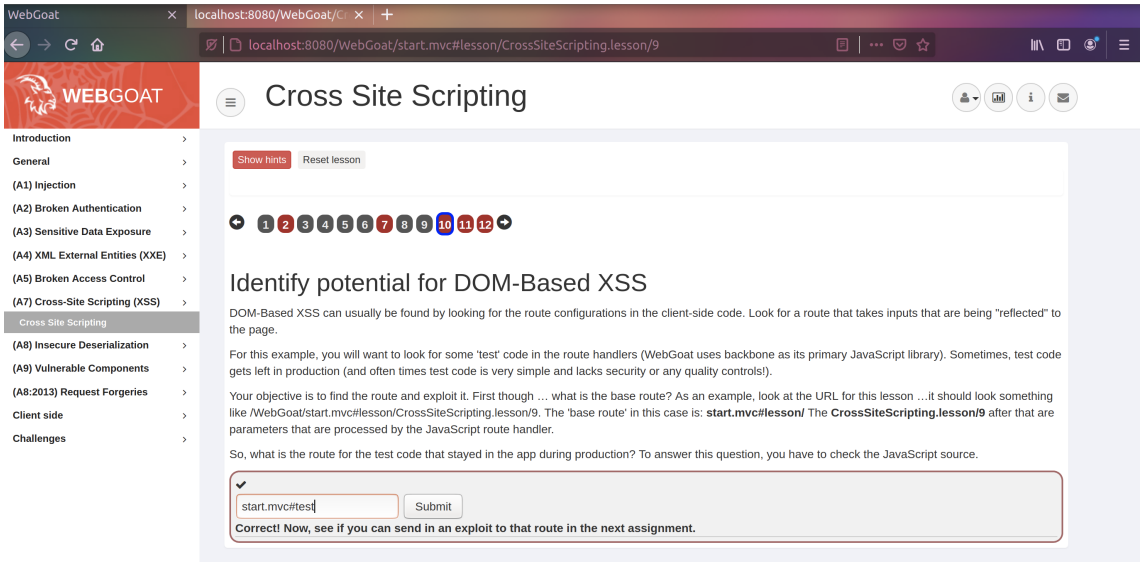

Figure 1.6: Path constructed with the code found in the js files.

1.4. XSS Exercise 11

At this exercise, the previous test route is going to be used to inject the given javascript call by the exercise statement via the URL. The javascript call to be used is the following one.

webgoat.customjs.phoneHome()

When the test page is accessed the following URL is used to try to trigger the javascript function.

http://localhost:8080/WebGoat/start.mvc#test/<script>webgoat.customjs.phoneHome()</script>

However, the desired function is not called. In order to succeed, the "/" character needs to be encoded in hexadecimal to avoid the URL processing which treat this character as a path delimiter. Therefore the URL is modified and the following one is the one used.

http://localhost:8080/WebGoat/start.mvc#test/<script>webgoat.customjs.phoneHome()<%2Fsript>

Finally, the function is correctly called and the number to be introduced as the correct answer is given.

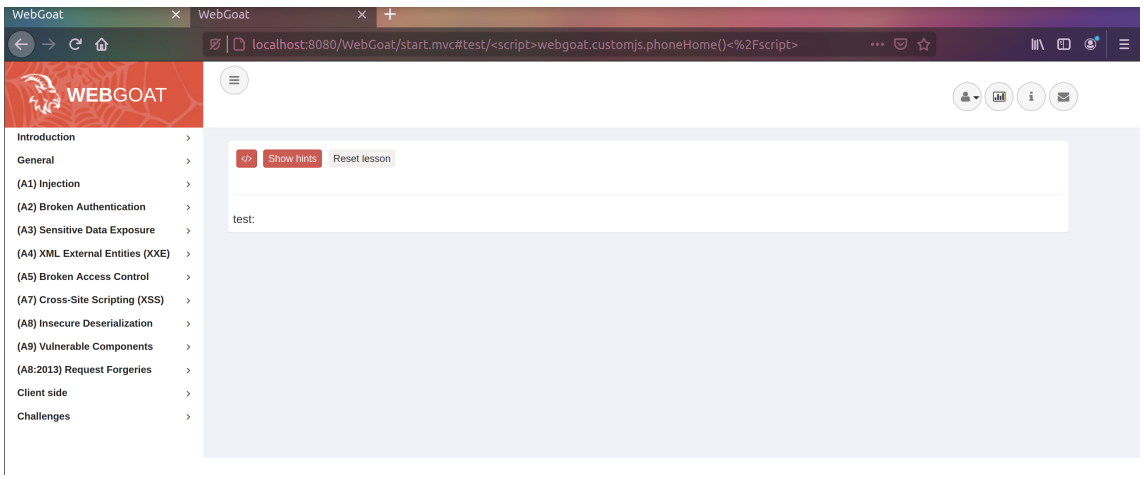

Figure 1.7: Injection of the script via URL.

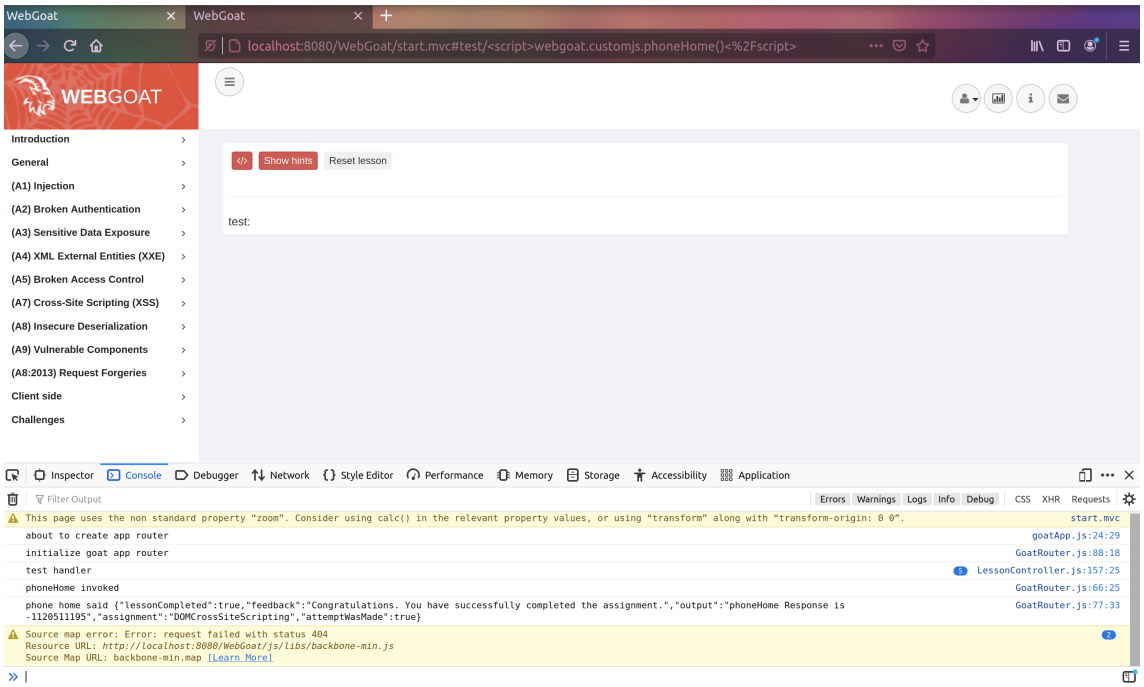

Figure 1.8: Secret number obtained from the correct call of the js function.

2. CODING EXERCISES

2.1. Exploiting the vulnerabilities

2.1.1. Exercise 1

The first exercise of this section just prints on the screen the parameter *"name"* passed through the URL.

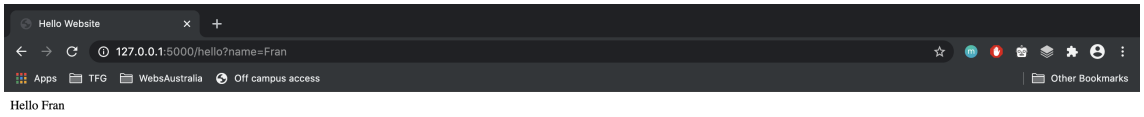

Figure 2.1: Normal behaviour of the browser belonging to Exercise 1.

To exploit the vulnerability of this page it is needed to add in the URL the script to be injected, which in this case it is just an alert with *"XSS"* inside it.

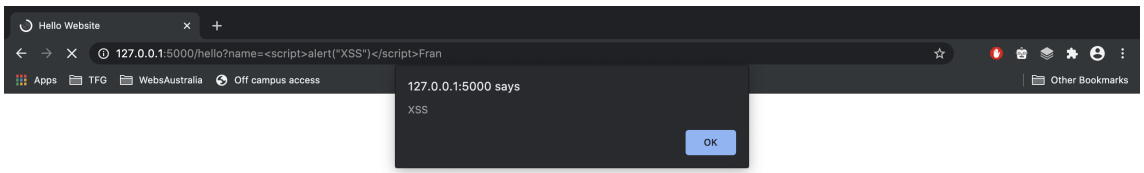

Figure 2.2: XSS injected into the URL displaying the alert.

2.1.2. Exercise 2

In this exercise, it is required to add a button with *"Click Me"* written which sends the alert with the text *"You have a Virus!"*.

In order to accomplish it, the same URL parameter as the one used in the previous exercise is going to be used. The query which is sent through the url is the following one.

hello?name="<button onclick="alert('You have a Virus!')">Click Me</button></body></html><!–

As it can be seen, the link element is closed at first. Then, the button is added with the stated values and functions. Finally, the body and the html objects are closed and the rest is commented to avoid any issue with the remaining original html code.

The result from the browser is the following screen.

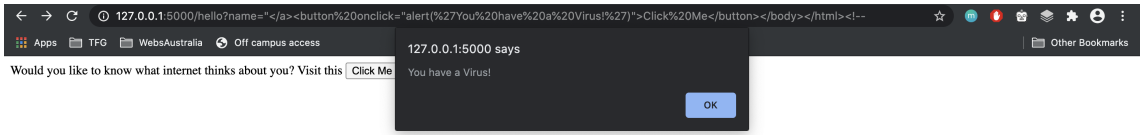

Figure 2.3: Response from the browser to the XSS injection.

2.1.3. Exercise 3

This exercise is very similar to the previous one. Instead of creating an alert with an arbitrary string in this case the cookie is going to be displayed.

The code injected can be seen in the following picture.

Figure 2.4: XSS code injected for exercise 3.

Figure 2.5: Response from the browser to the XSS script.

Figure 2.6: Cookie inserted in the given URL.

2.2. Fixing the vulnerabilities

2.2.1. Exercise 1

In order to fix the XSS vulnerability of the first exercise the function call *escape* from the python library *html* is going to be used to sanitize the input parameter introduced via the URL.

The resulting code looks similar to the following picture.

```python
from flask import Flask, request
from html import escape
app = Flask(__name__)

@app.route("/hello")
def hello():
    name = request.args.get('name')
    name = escape(name)
    content = """
    <html>
        <head><title>Hello Website</title></head>
        <body>
            Hello {}
            <script>webgoat.customjs.phoneHome()</script>
            <script>alert.("XSS")</script>
        </body>
    </html>
    """.format(name)
    return content

if __name__ == "__main__":
    app.run()
```

Figure 2.7: Fix for the XSS vulnerability from the coding exercise 1.

2.2.2. Exercise 2

In order to fix the vulnerability from the exercise 2 the approach followed above.

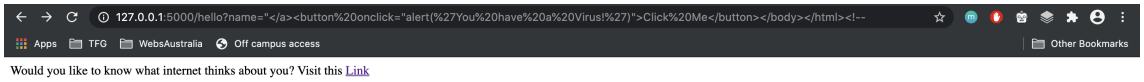

Figure 2.8: Response from the browser with the fixed code for exercise 2. (1)

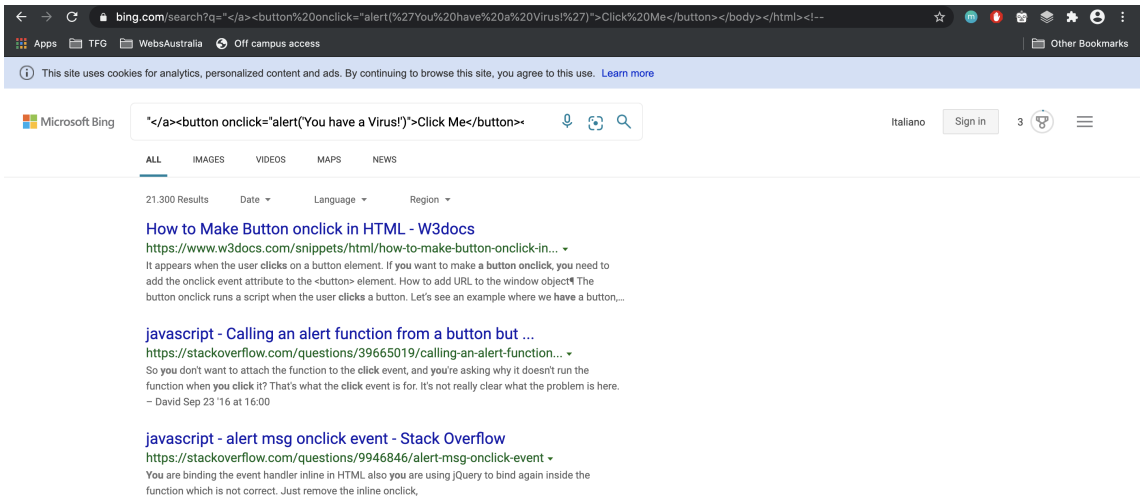

Figure 2.9: Response from the browser with the fixed code for exercise 2. (1)

As it can be seen in the above pictures, the browser does not display any alert but it treats the input parameter as a string.

2.2.3. Exercise 3

In order to fix this vulnerability the approach followed above.

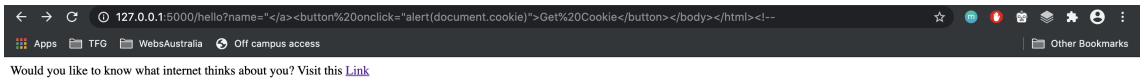

Figure 2.10: XSS code injected in the URL.

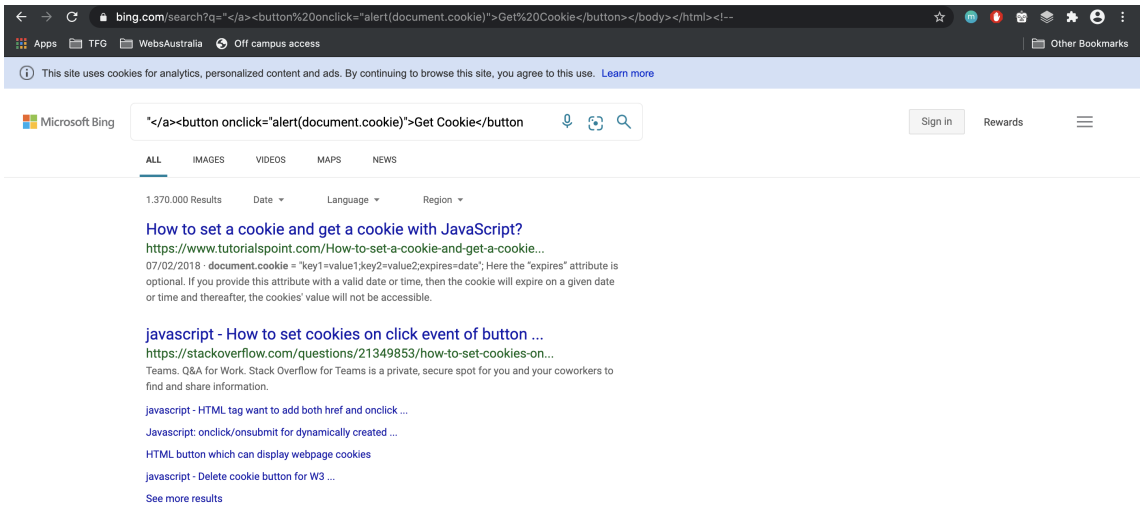

Figure 2.11: Response to the XSS code injected in the URL.

As the last picture shows, the XSS vulnerability is no longer available as it treats the parameter as a string and the code is not executed.