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.

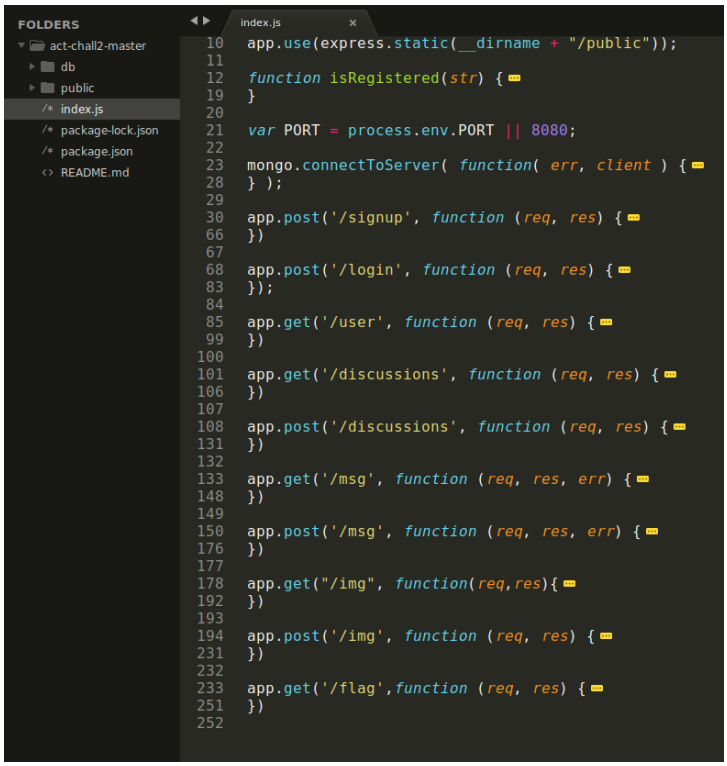

Figure 1: REST api of the website

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*.

```
app.get('/flag',function (req, res) {
  var db = mongo.getDb();
  var str = req.headers.cookie
  if (typeof str === "undefined"){
    res.send("TOP SECRET ZONE")
  }
  cookie = str.substring(14);
  str = cookie.substring(0, cookie.length - 13);
  db.collection("token").find().toArray(function(err, result) {
    if(result[0].token[0] === str){
      db.collection("flag").find().toArray(function(err, result) {
        res.send(result[0].secret)
      });
    }
    else{
      res.send("TOP SECRET ZONE")
    }
  });
})
```

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.

```
app.post('/signup', function (req, res) {
  var db = mongo.getDb();
  var query = { email: req.body.email };
  db.collection("users").find(query).toArray(function(err, result) {
    if (err) throw err;

    if(typeof result[0] === "undefined"){
      if (req.body.email.length < 5 || req.body.password.length < 8){
        res.send("Email min length 5\nPassword min length 8")
      }
      else{
        var myobj = { email: req.body.email, password: req.body.password, adminFlag: req.body.adminFlag, token: req.body.token };
        db.collection("users").insertOne(myobj, function(err, res) {
          if (err) throw err;
          console.log("1 users inserted");
        });
        if(req.body.adminFlag === "true"){
          db.collection("token").find({ token: req.body.token }).toArray(function(err, result) {
            if(typeof result[0] === "undefined"){
              console.log("He's not an admin!");
              var myquery = { email: req.body.email };
              var newvalues = { $set: {adminFlag: "false"}};
              db.collection("users").updateOne(myquery, newvalues, function(err, res) {
                if (err) throw err;
                console.log("1 users updated");
              });
            }
          });
        }
        res.send("Registered")
      }
    }
    else{
      res.send("Email already used")
    }
  });
})
```

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 (*admingFlag* on the request body setted to true), the provided token is right, if it's not, it update the value *admingFlag* 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-z^A-Z ^0-9\^_]+/g, "");
121        req.body.title = req.body.title.replace(/[^a-z^A-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.

```
app.post('/login', function (req, res) {
  var db = mongo.getDb();
  var query = { email: req.body.email, password: req.body.password};
  db.collection("users").find(query).toArray(function(err, result) {
    if (err) throw err;
    if(typeof result[0] === "undefined"){
      res.send("Wrong email and/or password")
    }
    else{
      var d = new Date();
      var n = d.getTime();
      res.cookie('access_cookie', req.body.password+req.body.email+n, {expires: new Date(Date.now() + 900000)});
      res.send("Logged in!")
    }
  });
});
```

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.

```
app.post('/img', function (req, res) {
  var db = mongo.getDb();
  var str = req.headers.cookie
  var user = isRegistered(str)
  db.collection("users").find().toArray(function(err, result) {
    result = result.find(x => encodeURIComponent(x.password+x.email) === user)
    if(typeof result === 'undefined'){
      var a = {msg: "not authorized!"}
      res.send(a)
    }
    else if(result.adminFlag !== "true"){
      var a = {msg: "not authorized!"}
      res.send(a)
    }
    else{
      var idPage = req.query.id
      req.body.src = req.body.src.replace(/[^(+)a-z^A-Z^0-9!./ \[\]\^_+]+/g, "");
      req.body.onerror = req.body.onerror.replace(/[^(+)a-z^A-Z^0-9!./ \[\]\^_+]+/g, "");
      var o_id = new mong.ObjectID(idPage);
      var myobj = {"_id": o_id}
      db.collection("topic").find(myobj).toArray(function(err, result) {
        if(typeof result[0] === "undefined"){
          console.log("This page doesn't exists!");
          var a = {msg: "The page not exists!"}
          res.send(a)
        }
        else{
          var newvalues = { $set: {img: {src:req.body.src, onerror: req.body.onerror}}};
          db.collection("topic").updateOne(myobj, newvalues, function(err, res) {
            if (err) throw err;
          });
          var a = {msg: "Image changed!"}
          res.send(a)
        }
      });
    }
  })
})
```

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.

```
$.ajax({
    url: '../../../../img?id=' + idPage,
    type: 'GET',
    dataType : 'json',
    success: (data) => {
      $("#imgbox").attr("src", data.src);
      $("#imgbox").attr("onerror", data.onerror);
    }
});
```

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 [,], (,), ! and + 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.

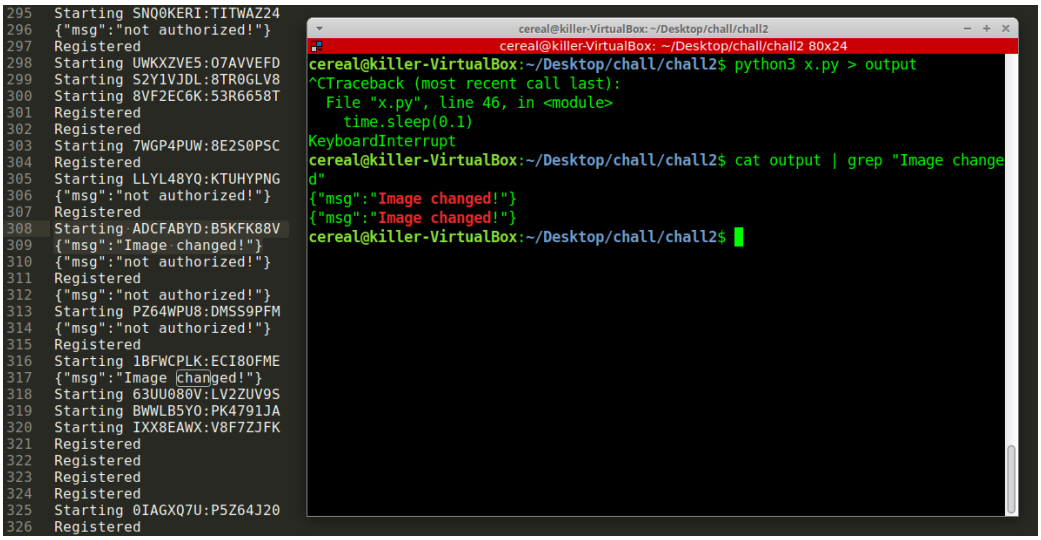

Figure 8: Run of x.py

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
/1ktfklk1?access_cookie=703373367639792442264529484 04D635166546A576E5A7234743777217
access_cookie=asdasdasd104992921580212410272

FORM/POST PARAMETERS

None

QUERYSTRING

access_cookie:
703373367639792442264529484 04D635166546A576E5A7234743777217A25432A462D4A6
14E645267556B587032733576387 82F413F4428472B4B6250655368566D59713374367739
7A244226452948404D635166546A576E5A723475377821412544 2A462D4A614E64526755
6B587032733576387 92F423F4528482B4B62501579978411999;

Figure 9: Super admin cookie

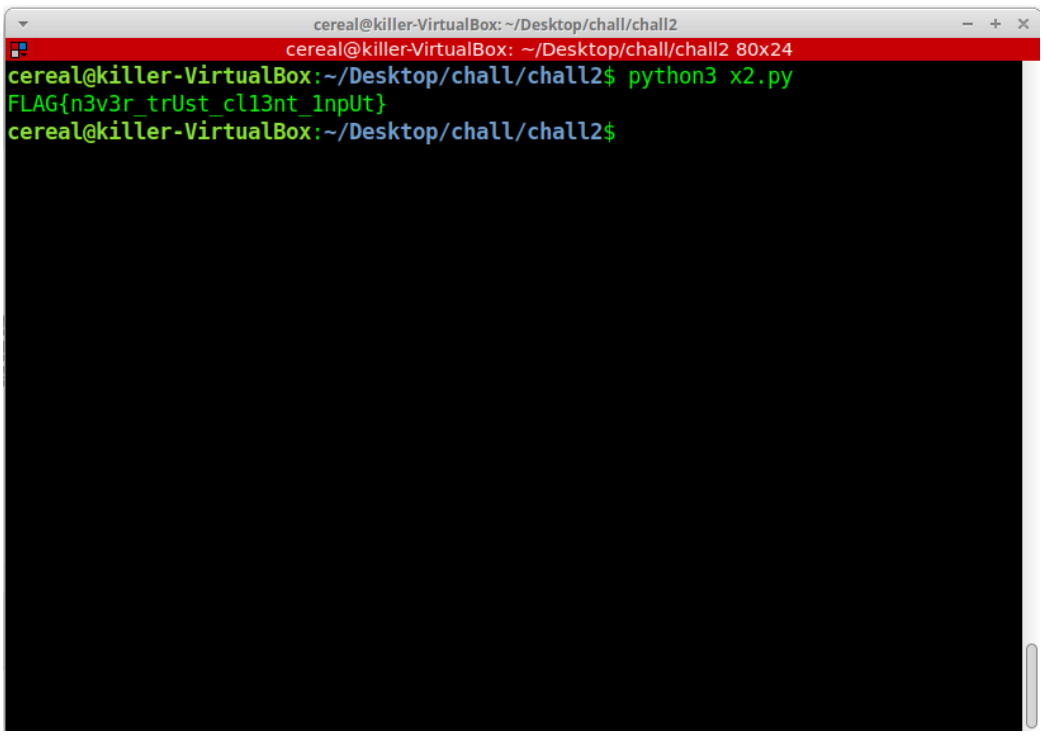

Figure 10: Run of x2.py