Title: Web Application Security

Quang, Fredrik

5.1 reDos vulnerability

Description
It is a denial of service attack that can potentially cause the server that is hosting the site to shut down. This exploit is based on the vulnerability that regex express implementing runs on the Nondeterministic Finite Automaton which has a finite state machine where for each pair of state and input symbols there can be several possible next states. This results in the algorithm attempting to try all possible paths one by one until the maximal match is found (when using ranges and other special characters, such as * and +, for 0 or more and 1 or more, respectively). For example regexes such as ^(a+)+$. This is an example of a regex that if the user has input such as `aaaaaaaaaaaaaaaaA`. It will result in 65536 paths and the number of paths doubled with each added in the match.
Vulnerability in the code

Exploited methods
In the code there are two locations where this type of regex can occur, it is the search in the main forum and the register/change password. Both of these have the same source of vulnerability but each case has a different exploitation path.
For the search case, it does not perform any filtering on the input and just feeds it immediately to the regex constructor, essentially allowing the user to craft their own regex. From that an evil regex can be injected. For example, `(a+)+` into the search to attempt to use it to match with any data that has aaaaaaaaaaaaaaaaaaaaaaa!. (Note, this is only theoretical as MongoDB will result in the data unable to be revealed in the search as well as the match won't occur so server shut down will not occur..)

A vulnerability that can be implemented is the change password and register user. Both of these take the username and password and make the username as the new regex to match it with the password as a way to check that the password doesn't have a username in it. This creates a similar environment where you can inject the evil regex and password. After that you attempt to change the password to something that will cause the server to hang again.

Mitigation
1. Update the package dependency

As the current version ua-parser-js in the server dependency is out of date it is found to have the reDos vulnerability in its regex function. I assume this is because they used the flaw Nondeterministic Finite Automaton algorithm to help them or that their built-in regex. From that update these decency is shown to lower the potential of harm from it.

2. Alter the validatePassword and searchForm

This is a new version that rather than using match, they alter it to use includes. This is a simple patch that still retains the functionality while still allowing username to check. This includes a binary that automatically checks if it is included and ends the function rather than a recursive loop for each of the letters.

```
function validatePassword(req, res, next) {

  let pass = req.body.newPassword;
  let name = req.user.username;
  // validate password
  if (pass.includes(name)) {
    req.flash('error', 'Do not include name in password.')
    return res.redirect('back')
  }
```

3. Limited Input string

There can be a filter system for the regex implementation that limit the number of characters to a certain value like 50. This is to bound the input for match to reduce the number of paths that are backtrack. A location where it can be implemented is link to validatepassword.

5.2 Side Channel Attack

Description: Side Channel attack is based on the obtaining of extra information that can be gathered based on the built in function and algorithm in the application. In this case it is a storage side channel which is a case where the attacker can learn about different username and parking plate without access to the database.

What information can be used as a side channel?

Register:

In the register, it will reveal whether the username or the parking plate had already existed in the database. You can create a user such as ‘test’ and then log out and create another user with the same username ‘test’ it will reveal that this name was already in the data which reveals information that the attacker can use for further attack. This can also be used for the parking plate.

After you log in the website, the user can attempt to park their car using their parking plate but it is also able to park with other people's plates as well. From this you can attempt to park other people's plates without control of the user account as well as learning all the available parking plates that exist in the base as it will only park the plate that exit as well as reveal if this plate had been booked by another account.

How can the side channel be exploited? Provide an implementation of how to exploit each of the found side channels. We provide some ideas for how to use Python for scraping and querying forumerly in the web scraping folder.

The process can be automated using request API and beautifulsoup so that you do not have to manually enter.

- Code to check the username and plate

```
def post_request():
   response = requests.post("http://localhost:3000/signup", data={
   'username': 'ABC',
   'plate': 'ABC123',
   'password': '123',
   'password2': '123'
})
   soup = BeautifulSoup(response.content, "html.parser")

   return soup.find('div', class_='alert-danger')
```
This code is simple: use the API to sign up and use § the data to input it in the forum without the need to use the browser itself.

- Code to check available park

```
def post_parking(plate,time,local_id):
   response = a.post("http://localhost:3000/park", data={
       'plate': plate,
       'time': time,
       'location': local_id

},cookies={"connect.sid":"s:JxExmGLCfBduEsO33aiZelF9ZCQegmZk.g/0X9x6940ewa6jqXUFFXKedn7XZRIkeUnVXyqOIC8s"}
})
   soup = BeautifulSoup(response.content, "html.parser")
   return soup.find('div', class_='alert-danger')
```

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. Note that the communication between forumerly and third-party services cannot be changed.

1. **License Plate Verification During Registration**: Forumerly should verify that the license plate provided during registration matches the registrant's information. This can be done by cross-referencing with official databases or requiring additional documentation.
2. **Consistent Error Messages**: To thwart side-channel attacks, Forumerly should ensure that error messages displayed during registration attempts are consistent regardless of whether the license plate is valid or not. This prevents attackers from gaining information about the validity of specific license plates.
3. **Randomized Error Messags**: Introducing randomized error messages adds an extra layer of complexity for potential attackers. By displaying different error messages for the same scenario, it becomes more difficult for them to discern patterns or exploit vulnerabilities.

4. **Rate Limiting**: Enforcing rate limiting on registration attempts helps mitigate brute-force attacks. By restricting the number of registration attempts within a specific timeframe, Forumerly can deter malicious actors attempting to gain unauthorized access.
5. **CAPTCHA Challenges**: Implementing CAPTCHA challenges during registration adds another obstacle for automated attacks. This ensures that registration attempts are made by genuine users rather than bots or scripts.
6. **Registration Attempt Delays**: Introducing delays between registration attempts further discourages attackers. By implementing a waiting period after failed attempts, Forumerly can hinder automated attacks while not inconveniencing legitimate users significantly.

5.3 XSS

Exploit

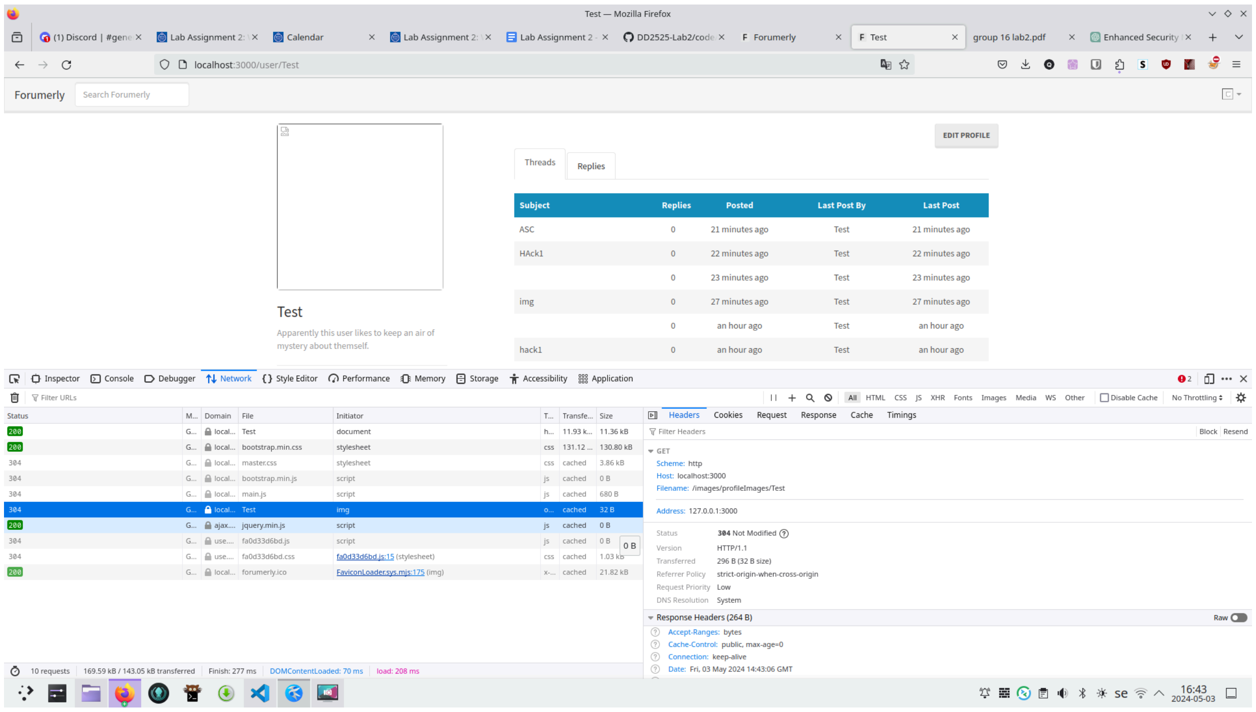
1. Inject script

The XSS vulnerability was exploited by uploading a malicious JavaScript file disguised as an image (.png.js) to the "images" section of the profile page. You can easily navigate the profile and edit profile and the new image is a drag and drop. The misleading file extension bypassed the server's basic file validation checks. Inside the file you can then set up a command such as.

```
alert("Hello World");
```

2. Code execution

As the image itself can not be executed with a button or such it will ont function as intended. From that it needs a script to be able to run it. It can be done by create a new thread in any of the category. In here you can defined it name as it allow it generate a button to activate the command. In the body you can write to escape the string and write a script tag that link to the profile image. You can find them in the network for the profile page.



Test — Mozilla Firefox

localhost:3000/user/Test

Forumerly

Test

Apparently this user likes to keep an air of mystery about themself.

Subject	Replies	Posted	Last Post By	Last Post
ASC	0	21 minutes ago	Test	21 minutes ago
HAck1	0	22 minutes ago	Test	22 minutes ago
	0	23 minutes ago	Test	23 minutes ago
img	0	27 minutes ago	Test	27 minutes ago
	0	an hour ago	Test	an hour ago
hack1	0	an hour ago	Test	an hour ago

Status	M...	Domain	File	Initiator	T...	Transfe...	Size
200	G...	local...	Test	document	h...	11.93 k...	11.36 kB
200	G...	local...	bootstrap.min.css	stylesheet	css	131.12 ...	130.80 kB
304	G...	local...	master.css	stylesheet	css	cached	3.86 kB
304	G...	local...	bootstrap.min.js	script	js	cached	0 B
304	G...	local...	main.js	script	js	cached	680 B
304	G...	local...	Test	img	o...	cached	32 B
200	G...	ajax...	jquery.min.js	script	js	cached	0 B
304	G...	use...	fa0d33d6bd.js	script	js	cached	0 B
304	G...	use...	fa0d33d6bd.css	fa0d33d6bd.js:15 (stylesheet)	css	cached	1.03 kB
200	G...	local...	forumerly.ico	FaviconLoader.sys.mjs:175 (img)	x-...	cached	21.82 kB

10 requests | 169.59 kB / 143.05 kB transferred | Finish: 277 ms | DOMContentLoaded: 70 ms | load: 208 ms

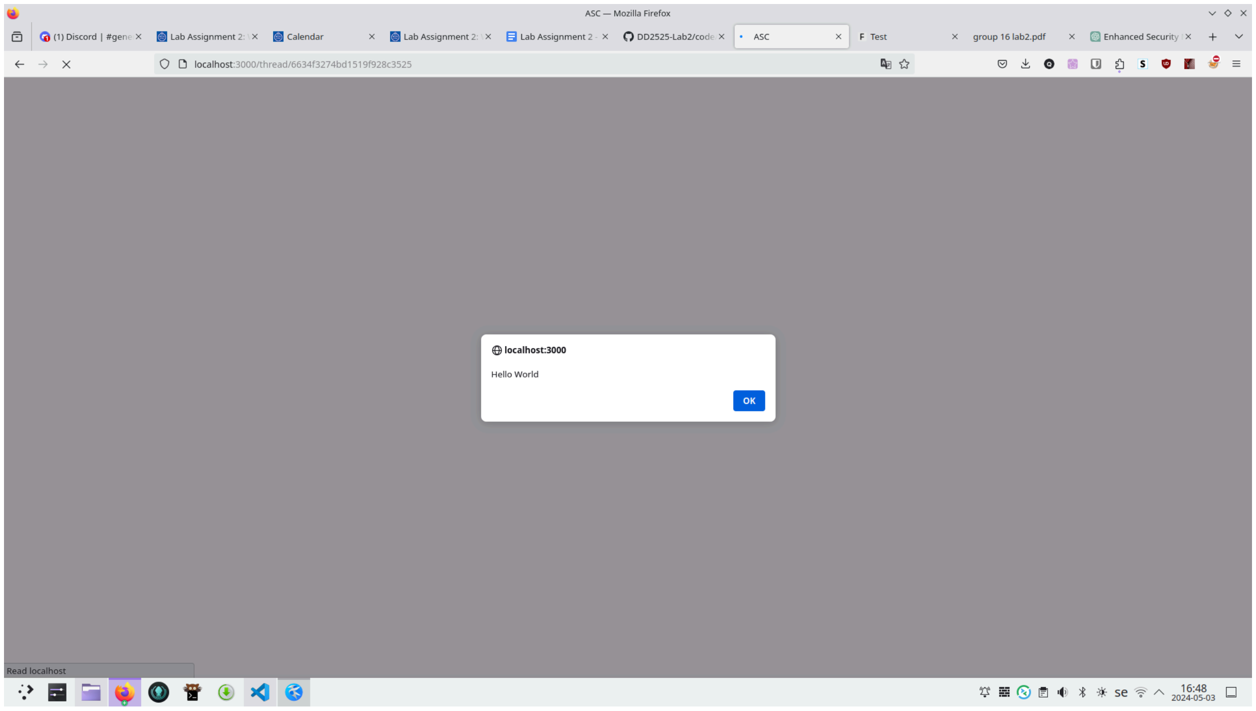
```
GET
Scheme: http
Host: localhost:3000
Filename: /images/profileImages/Test

Address: 127.0.0.1:3000

Status            304 Not Modified
Version           HTTP/1.1
Transferred       296 B (32 B size)
Referrer Policy   strict-origin-when-cross-origin
Request Priority  Low
DNS Resolution    System

Response Headers (264 B)
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Connection: keep-alive
Date: Fri, 03 May 2024 14:43:06 GMT
```

In this case it is /images/profileImages/Test, this results in our post to be <s<scriptcript src="/images/profileImages/Test"></script>. The use of <s<scriptcript is done to keep the code intact after the server attempts to sanitize the input. When posted it will run the script and the alarm will occur. Of course, different script can be implemented by simply changing the image with something else which would still retain it effect.



ASC — Mozilla Firefox

localhost:3000/thread/6634f3274bd1519f928c3525

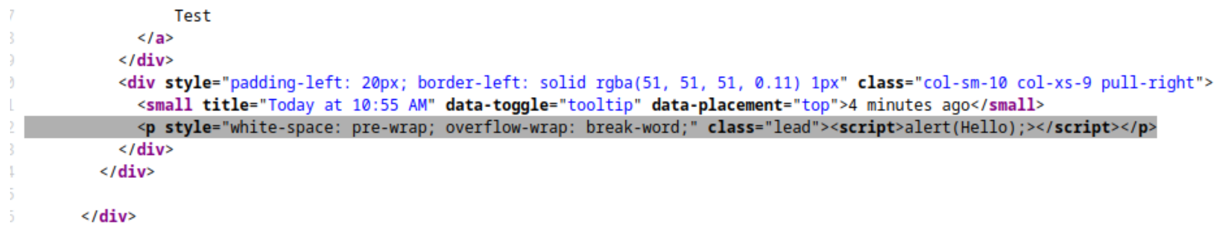
localhost:3000

Hello World

OK

Read localhost

Discussion:
The Content Security Policy (CSP) implemented aimed to limit script sources to the same domain . From that by attempt to inject a typical XSS will not work as the script doesn't originate from the server itself.



```
                Test
              </a>
            </div>
            <div style="padding-left: 20px; border-left: solid rgba(51, 51, 51, 0.11) 1px" class="col-sm-10 col-xs-9 pull-right">
              <small title="Today at 10:55 AM" data-toggle="tooltip" data-placement="top">4 minutes ago</small>
              <p style="white-space: pre-wrap; overflow-wrap: break-word;" class="lead"><script>alert(Hello);></script></p>
            </div>
          </div>

        </div>
```

Output error:
Content-Security-Policy: The page’s settings blocked an inline script (script-src-elem) from being executed because it violates the following directive: “script-src 'self' http://use.fontawesome.com http://ajax.googleapis.com http://cdnjs.cloudflare.com”

From my it need to inject the image code into the server to allow it to execute.

Further noted:
A different method can also be done using white list website by implemented those to redirect them to fulfill certain command similar to without the need of upload the image. Havent implemented them for the lack of time but i do know you can fine them here
https://github.com/zigoo0/JSONBee/blob/master/jsonp.txt
ng-app"ng-csp ng-click=$event.view.alert(1337)><script src=//ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.js></script>
This is the code that can be implemented in the body of a new thread to execute as it is white list in the CSP. The issue is that I do need to sanity it so it can be run properly but all the bottom one seem to be able to by pass CSP

Mitigation:

1. **Strict CSP Policies**: Review and enhance the CSP policies to restrict script sources not just to 'self' but also to trusted domains or specific sources where scripts are legitimately hosted.
2. **Sanitization of User Input**: Implement thorough input validation and sanitization procedures to filter out or neutralize potentially malicious script tags or code injected by users.
3. **File Type Verification**: Strengthen server-side validation to verify the file type and content of uploaded files. Ensure that only permitted file types (e.g., image files) are accepted in the designated directories.
4. **Renaming or Restructuring Directories**: Consider renaming or restructuring directories to prevent attackers from hosting malicious scripts in directories accessible by the application.

5.4 RCE

By manually inspecting the code we found a suspect function that dynamically updates some target object from a source object called *merge()* in the user.js file. The object (or objects) are some number of users that should be added if they do not exist, or updated if they do exist, in JSON format. The functionality is available through the profile for administrator accounts, but as the API endpoint is unprotected it is available to anyone who is aware of it, which is completely unreasonable.

The vulnerability can be exploited by passing some object that has a property that uses the *__proto__* or *constructor* property to assign a value to an appropriate key on the root *Object*. The property is preferably set to some key that is not available in the target object as the merge function then creates a new object through the object literal which has the root object as immediate ancestor in the prototype chain (which is not the case for objects on existing keys).

In our case the exploit is performed by setting the *userAutoCreateTemplate* property on the root Object to some string value of the function we want to execute. In the passport.js file we identified the *eval()* function being used (which is just about never a good thing in production environments). We see how the *wrapperFunction* variable holds a string representation of an immediately invoked function expression that returns the value of *options.userAutoCreateTemplate*. The property is originally undefined, but setting it on the root object, as previously described, ensures traversal up the prototype chain until the property is found assigned to some object (or the root object is reached). Clearly, this allows remote code execution. MongoDB does not allow you to update the __proto__ or constructor property without throwing an error so the application crashes, but one may still make a login request for some non-existing user to trigger it. In essence, even if one does not perform remote code execution, a denial-of-service situation occurs as the web client becomes unusable.

From here one can open a reverse shell, if desirable.

This is called prototype pollution. The assigning of the payload is often called an injection sink, and the gadget utilized to perform the attack is often called an attack sink.

The vulnerability can be mitigated by immediately returning from the callback function in the forEach loop iterating over the keys of the source object (taken as input) in the *merge()* function, if the key is of the dangerous kind, e.g., __proto__, constructor, or prototype. One can also imagine other properties that, while not as critical, would be highly undesirable to alter, such as the *toString()* function, which may cause unexpected application behavior.