2019

Secure Prog & Scripting

ASSIGNMENT 4

DYLAN FRENCH CARROLL, 20080672

Table of Contents

1.a

HTTP POST: This method is used to transfer data from client to server. It carries request parameters in the message body which will make it more secure than GET method.

HTTP GET: GET is also a way to transfer data from client to server, but it carries the request parameters in URL string instead which isn't as secure as POST.

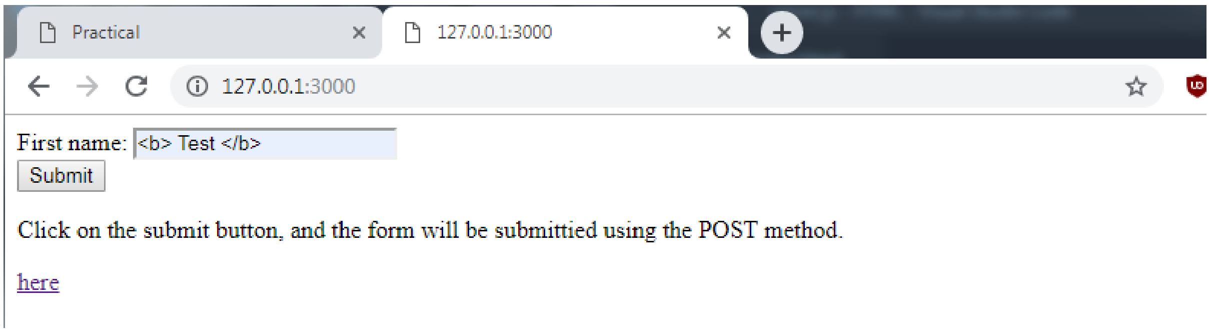
HTTP PUT: This method creates a new resource or replaces a representation of target resource with the request payload.

HTTP DELETE: This method deletes the resource specified.

HTTP PATCH: This method allows for you to apply partial modifications to a resource specified.

1.b,c,d

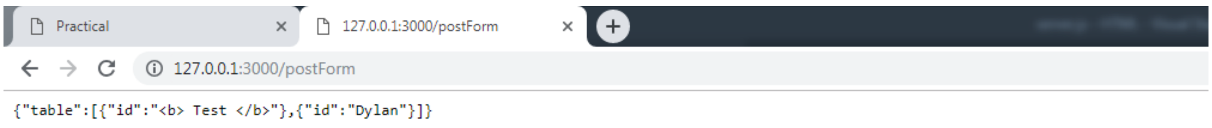
I figure it would be easier if I included the full script for this part instead of breaking it into different parts. Here is the following script including my GET and POST requests. This script is also vulnerable to XSS and I changed some text in entered by using the bold tags <b></b> which edited the text when I returned it from the JSON database. This is my server.js file on the next page. It will be in the folder *"Part 1.zip"* also if needed. Here is my form and my input:



First name: <b> Test </b>

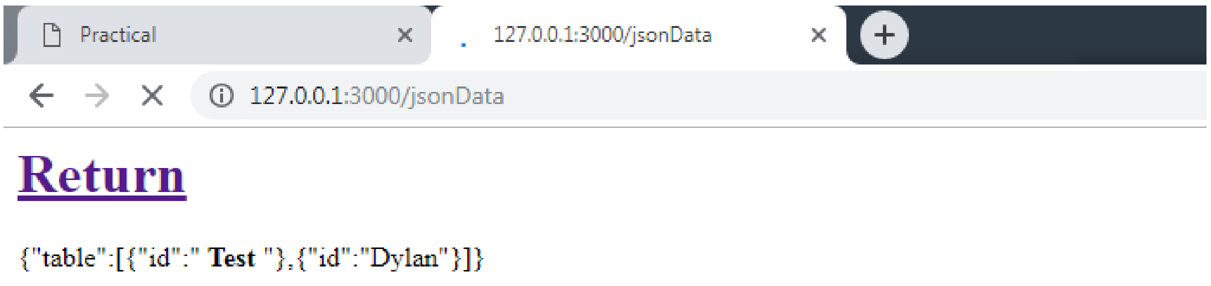
Click on the submit button, and the form will be submittied using the POST method.

It will bring you to this page, after that return to the main page with the form on it and click *"here"*



```
{"table":[{"id":"<b> Test </b>"},{"id":"Dylan"}]}
```

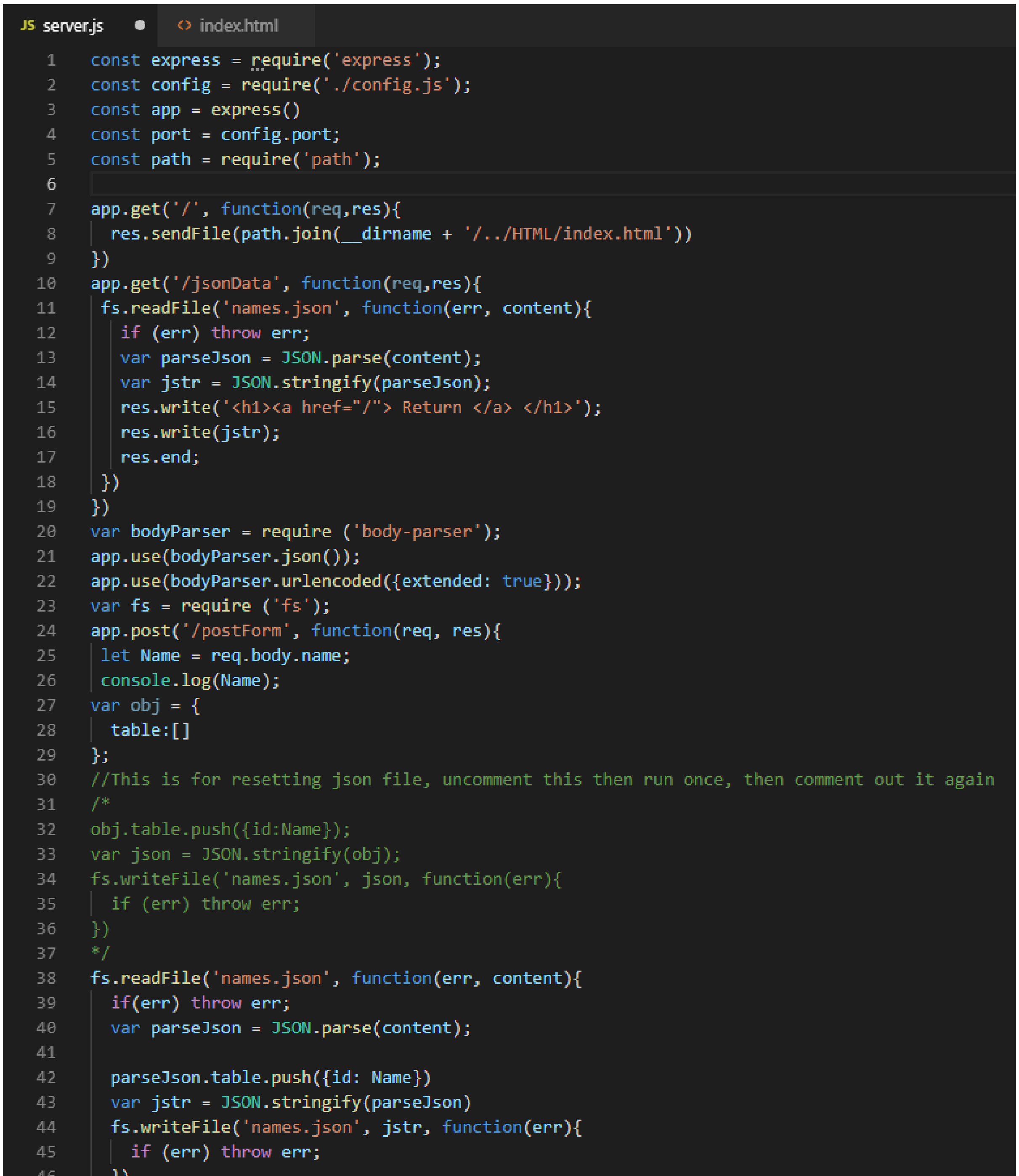
Which will bring you to a page that reads the json data and shows the XSS manipulation



Return

{"table":[{"id":" **Test** "},{"id":"Dylan"}]}

Here it shows the **Test** in bold.



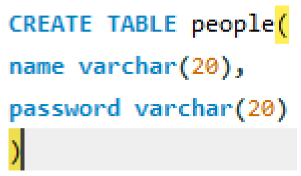
```
const express = require('express');
const config = require('./config.js');
const app = express()
const port = config.port;
const path = require('path');

app.get('/', function(req,res){
  res.sendFile(path.join(__dirname + '/../HTML/index.html'))
})
app.get('/jsonData', function(req,res){
 fs.readFile('names.json', function(err, content){
   if (err) throw err;
   var parseJson = JSON.parse(content);
   var jstr = JSON.stringify(parseJson);
   res.write('<h1><a href="/"> Return </a> </h1>');
   res.write(jstr);
   res.end;
 })
})
var bodyParser = require ('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
var fs = require ('fs');
app.post('/postForm', function(req, res){
 let Name = req.body.name;
 console.log(Name);
var obj = {
  table:[]
};
//This is for resetting json file, uncomment this then run once, then comment out it again
/*
obj.table.push({id:Name});
var json = JSON.stringify(obj);
fs.writeFile('names.json', json, function(err){
  if (err) throw err;
})
*/
fs.readFile('names.json', function(err, content){
  if(err) throw err;
  var parseJson = JSON.parse(content);

  parseJson.table.push({id: Name})
  var jstr = JSON.stringify(parseJson)
  fs.writeFile('names.json', jstr, function(err){
    if (err) throw err;
```

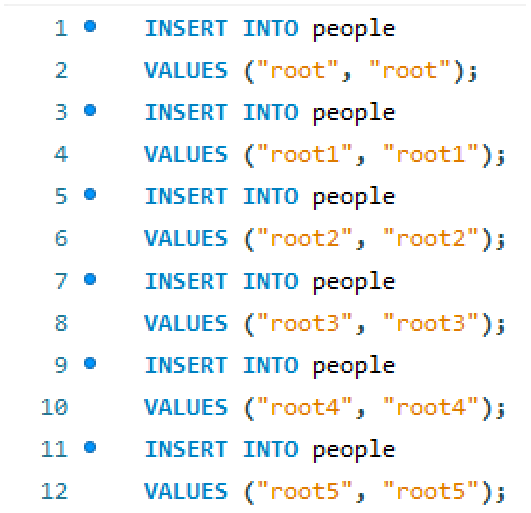
2.

My database/table:



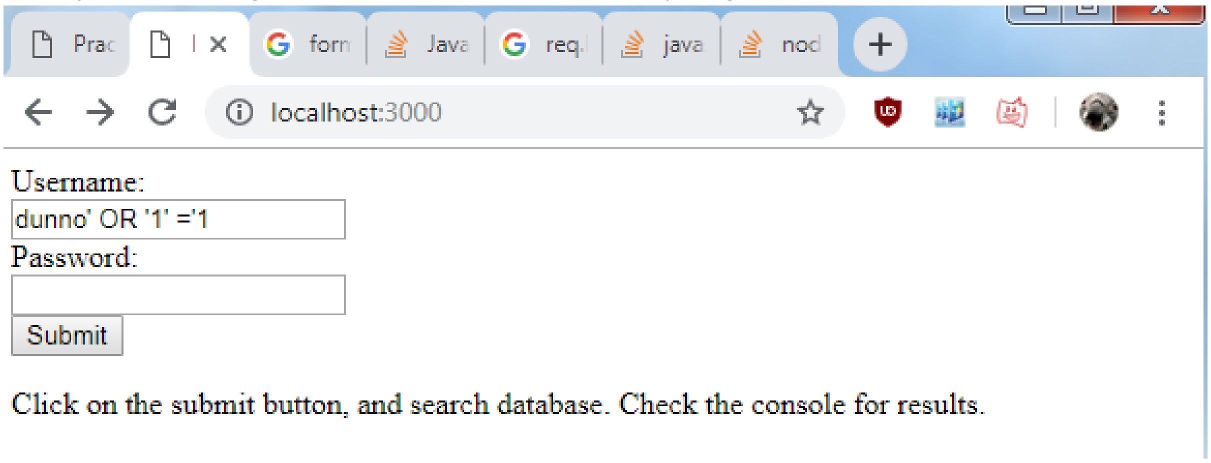
```
CREATE TABLE people(
name varchar(20),
password varchar(20)
)
```

My query to fill the database



```
INSERT INTO people
VALUES ("root", "root");
INSERT INTO people
VALUES ("root1", "root1");
INSERT INTO people
VALUES ("root2", "root2");
INSERT INTO people
VALUES ("root3", "root3");
INSERT INTO people
VALUES ("root4", "root4");
INSERT INTO people
VALUES ("root5", "root5");
```

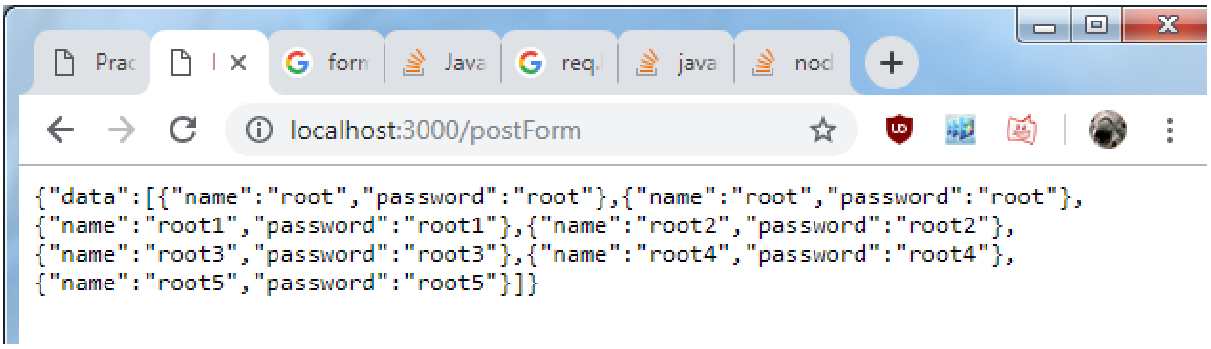
I had some problems with this, as I could not determine if the username was not in the database as the "results" variable returned as "[]" when there were no usernames matching the input. Instead I displayed the results to the web page to show whether or not they were there. I did however get the SQL injection working as shown below. It returned everything in the table.



Username:
dunno' OR '1' ='1
Password:

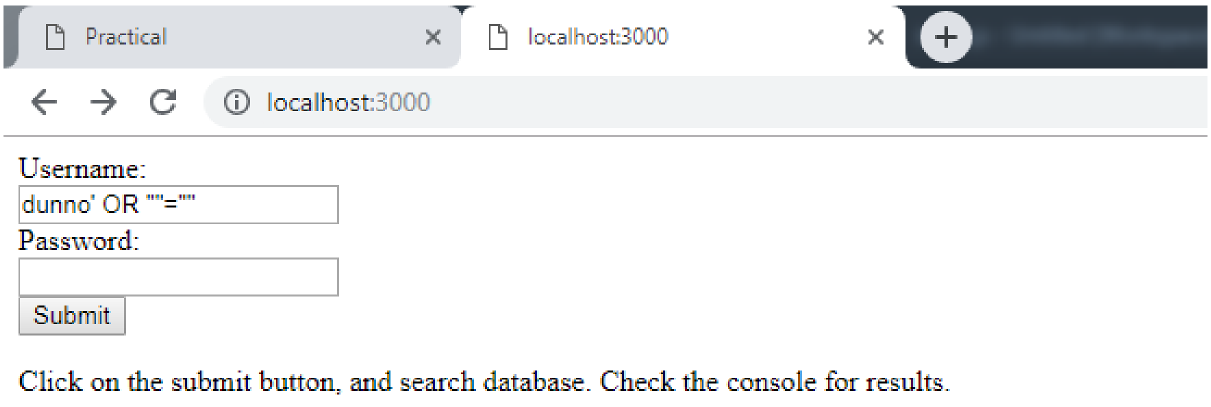
Submit

Click on the submit button, and search database. Check the console for results.



```
{"data":[{"name":"root","password":"root"},{"name":"root","password":"root"},
{"name":"root1","password":"root1"},{"name":"root2","password":"root2"},
{"name":"root3","password":"root3"},{"name":"root4","password":"root4"},
{"name":"root5","password":"root5"}]}
```

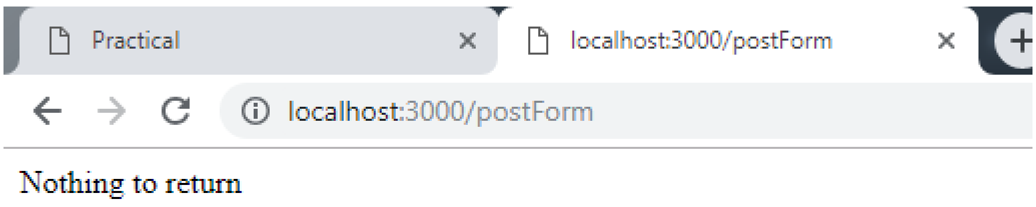
From continuing on with this assignment, if I gave an SQL injection which syntax was wrong, it threw an error and gave me this which it was programmed to do.



Username:
dunno' OR ""=""
Password:

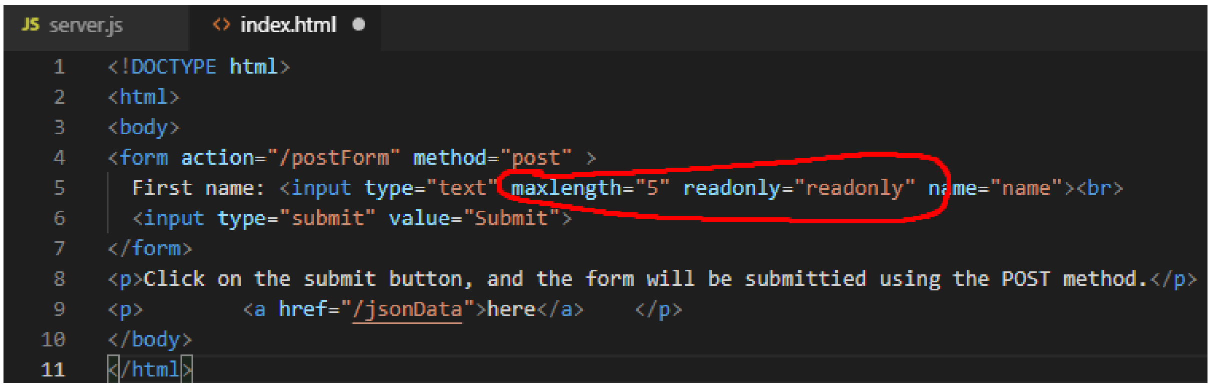
Submit

Click on the submit button, and search database. Check the console for results.



Nothing to return

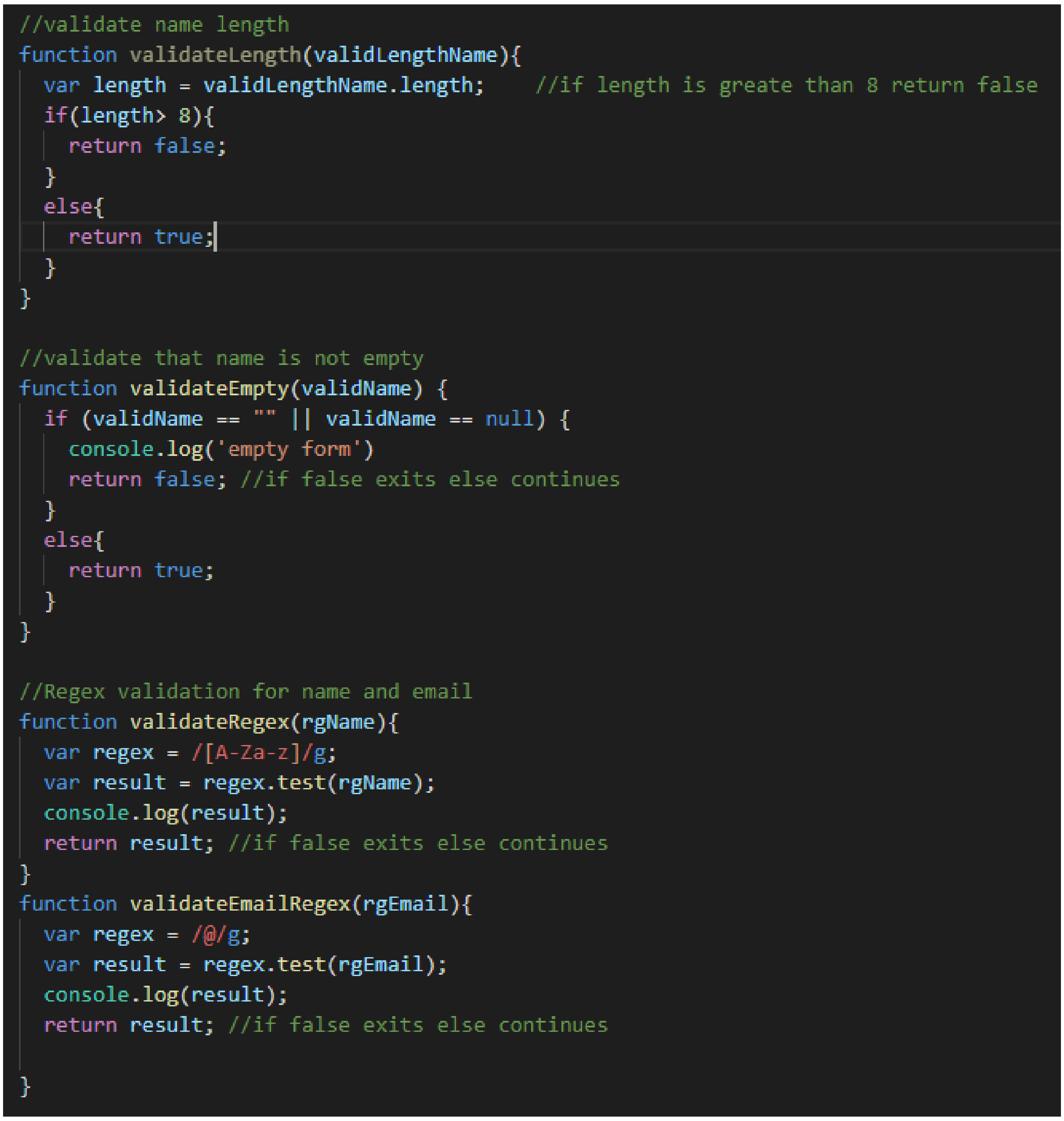
3.



```html
<!DOCTYPE html>
<html>
<body>
<form action="/postForm" method="post" >
  First name: <input type="text" maxlength="5" readonly="readonly" name="name"><br>
  <input type="submit" value="Submit">
</form>
<p>Click on the submit button, and the form will be submittied using the POST method.</p>
<p>        <a href="/jsonData">here</a>    </p>
</body>
</html>
```

Here I input the readonly=”readonly”(I used this as the W3C standard has changed and adopted this), and maxlength=”5” into the HTML form. These offer little security because the HTML is downloaded by the client and therefore the client has full control over the HTML as it is on their device. It is simple to use the inspect element tool on Chrome to remove these restrictions and send whatever data you like to the server if there is no validation server side.

4.



```javascript
//validate name length
function validateLength(validLengthName){
  var length = validLengthName.length;    //if length is greate than 8 return false
  if(length> 8){
    return false;
  }
  else{
    return true;
  }
}

//validate that name is not empty
function validateEmpty(validName) {
  if (validName == "" || validName == null) {
    console.log('empty form')
    return false; //if false exits else continues
  }
  else{
    return true;
  }
}

//Regex validation for name and email
function validateRegex(rgName){
  var regex = /[A-Za-z]/g;
  var result = regex.test(rgName);
  console.log(result);
  return result; //if false exits else continues
}
function validateEmailRegex(rgEmail){
  var regex = /@/g;
  var result = regex.test(rgEmail);
  console.log(result);
  return result; //if false exits else continues

}
```

Here I’ve added some functions for validating input from my form. The first one is very basic and makes sure that the length of their first name is no greater than 8. If it is greater than 8 the function will stop before it passes the data into the table array. The next function makes sure that the input is not empty. The 3rd and 4th are two separate regular expression functions to make sure that for names, only alphabetic characters are entered and that the email input must contain a “@” symbol. Files uploaded in “Part 4.zip”

6

A10: 2017 - Insufficient Logging and Monitoring

“Exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.” - OWASP

In my application, ideally whenever the web application did any kind of action I would want it to write to a log file. In my web app, I have some console.log methods to confirm that pieces of code are working and being used. Instead of using this, ideally I would find a library online from a trusted source that is known to have no vulnerabilities. After installing the package, creating a logger variable and then simply log it to a file using logger.log or whatever the syntax may be. These attacks usually take place over a time(days, weeks, months) and not all at once. Ideally to go unnoticed I would be taking small bits of data or doing stuff that wouldn’t look too irregular or set off alarms over the course of a week or a month and eventually build up my data.

A8:2017- Insecure Deserialization

“Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks” – OWASP

It would be relatively simple to add serialisation and deserialization into my application and if I used unsecure libraries or used untrusted data then some problems could occur. Usually the programmer would be the one at fault for deserialization attacks due to bad code. Use of unsecure libraries to serialise and deserialize are one of the biggest faults. To stop this type of attack, ideally using only external libraries that are trusted will help. Only deserializing signed data is also a big part of preventing an attack of this type. When the object is serialised, they should be signed and if the object being deserialized doesn’t have the same signature it should choose not to deserialize it.