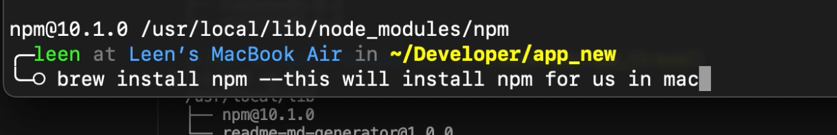
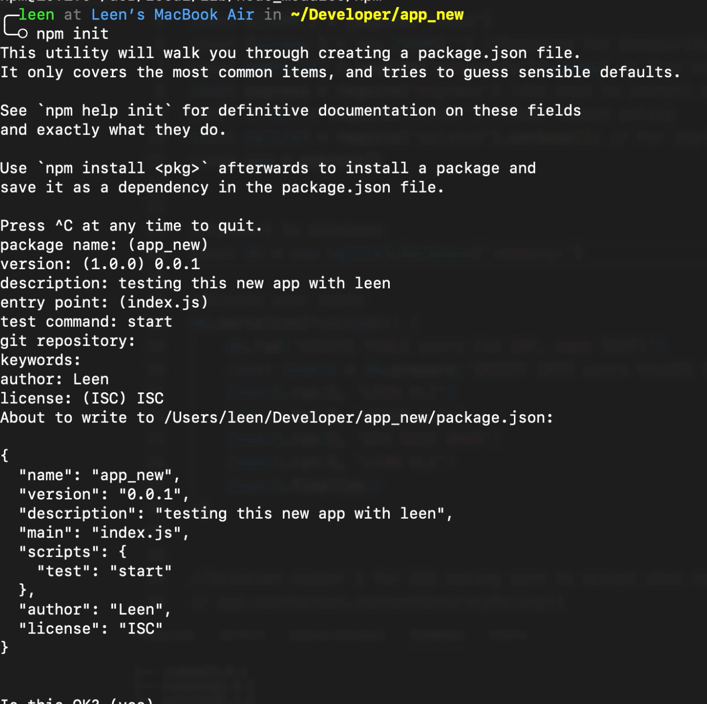
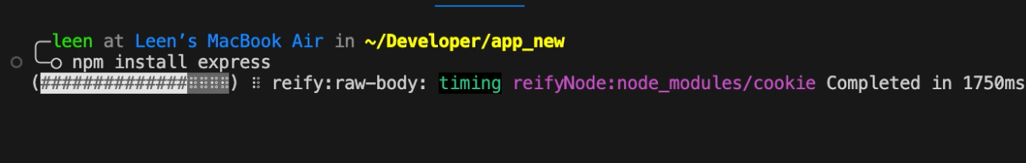
**Demonstration how the web application firewall works.**

**What we have implemented to to make this works on our website.**

* We used npm as a packet manager





* express | to use a server for demostration
* 
* helmet | to make sure the website does not execute any scripts or styles outside of the domain name. Prevent corss-site scripting.
* DOMPurify, JSDOM | Both are required to also prevent scripts entered in from users to the browser. Sanatize User Input. This is similar to the main goal to helmet.
* sqlite3 | Used to demonstrate sql injection and how to prevent sql injections.

**Solution 1 helmet (prevent scripts injection)**

Direction to use content security policy. Preventing users to execute any scripts or import any styles. It helps secure Express apps by setting HTTP response headers. Link: github.com/helmetjs/helmet

Try commenting the following code and uncommenting to see how it prevents these attacks.

...

app.use(helmet.contentSecurityPolicy({

directives: {

defaultSrc: ["'self'"],

scriptSrc: ["'self'"],

styleSrc: ["'self'"]

},

}))

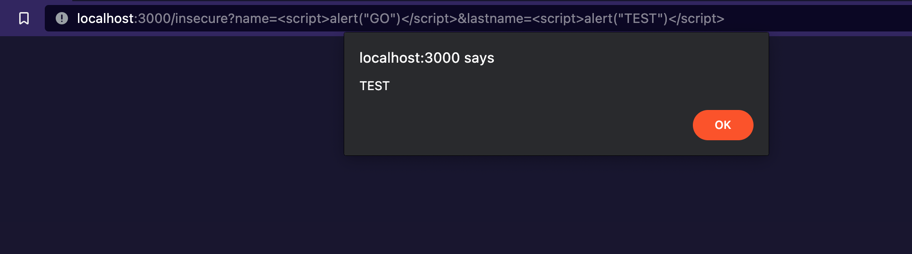
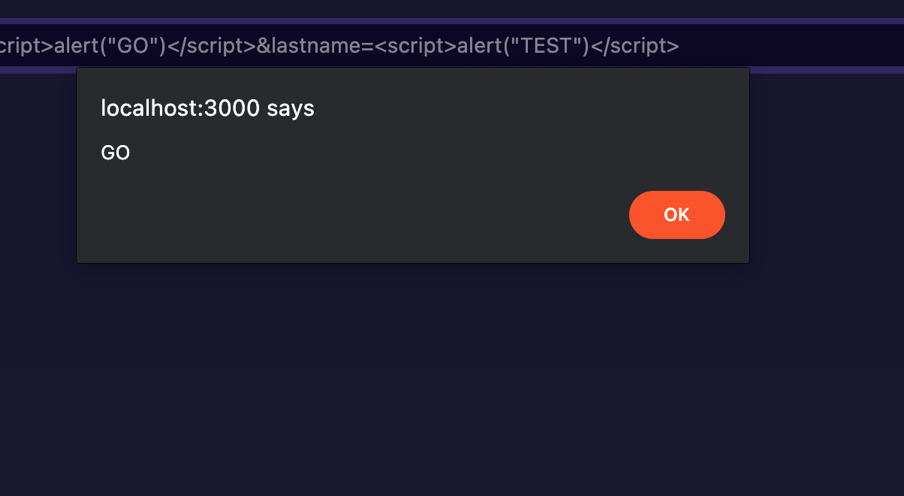
...

Test URLs

http:localhost:3000/insecure?name=leen&lastname=bajunaid

[http://localhost:3000/insecure?name=<script>alert("goo")</script>&lastname=<script>alert("hello")</script](http://localhost:3000/insecure?name=%3cscript%3ealert(%22goo%22)%3c/script%3e&lastname=%3cscript%3ealert(%22hello%22)%3c/script)>

This is if the helmet was commented.



We are using insecure paths just to show that it wouldn't matter if there was an injection, helmet will still be able to prevent it from executing.

**Solution 2 DOMPurify to sanatize input (prevent script injection)**

Requirements before using DOMPurify library. The URLs to test DOMPurify is the one that used in a different path /secure/

const {JSDOM } = require('jsdom')

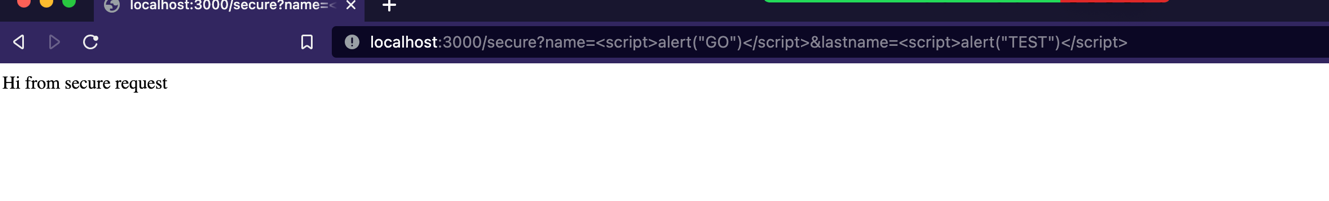
const {window } = new JSDOM('') //Required for domppurify

const DOMPurify = require('dompurify')(window) // This will sanitize user generated content

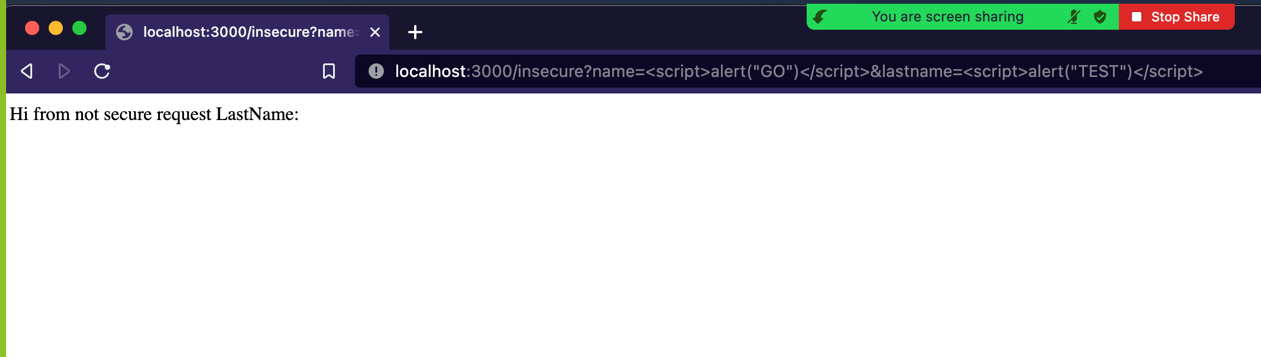
Test URLs

http:localhost:3000/insecure?name=leen&lastname=bajunaid

http://localhost:3000/secure?name=<script>alert("goo")</script>&lastname=<script>alert("hello")</script>

This URL wouldn't execute the script because its using DOMPurify.sanitize(user\_input) **this is a secure url**

**This is the insecure the scripts are not there because we used the Dumpurify.sanataize**

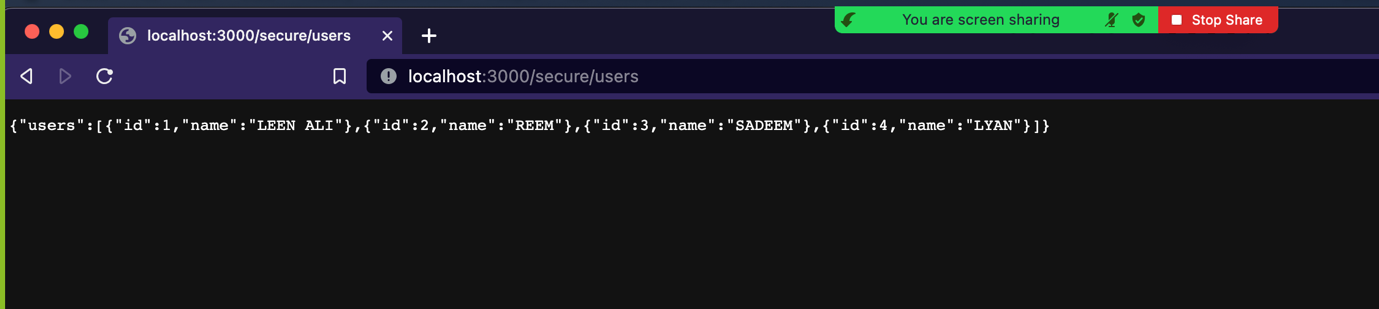
****

**SQL and SQL injection**

We need to import required libraries.

const sqlite3 = require('sqlite3').verbose(); // For implementing sql and show how to do an sql injection and how to prevent it.

We need to setup database and create our test table.

First we will show all users in the database, let us assume that we are admin when we access this link. /secure/users 

//Connect to database

const db = new sqlite3.Database(':memory:')

//Create user table

db.serialize(function() {

db.run('CREATE TABLE users (id INT, name TEXT)')

const insert = db.prepare('INSERT INTO users VALUES (?,?)')

insert.run(1, 'LEEN ALI')

insert.run(2, 'REEM')

insert.run(3, 'SADEEM')

insert.run(4, 'LYAN')

insert.finalize()

})

**NOTE:** In a secure way we have a special path that will let any user to access or display all registered users. Let us assume that only admin users can access it.

http://localhost:3000/secure/users

**Access specific user. Can be done by anyone.**

!IMPORTANT But through these urls, we shouldn't be able to access all users.

<http://localhost:3000/secure/user/2>

The following link will get user input as an ID of a specific user. i.e /secure/user/2

Here it validates the user input and make usre it doesn't contain any sql expression that may cause some sort of an injection.

A screenshot of a computer

Description automatically generated

http://localhost:3000/secure/user/1%20OR%201=1 //This url wouldn't work because of the way the function is implemented.

We can display single user using the url shown above, this will display a user with id number 2. The code that was implemented was also preventing sql injection attack. Here is a samlpe of the code.

app.get('/secure/user/:id', function(req, res) {

const userId = req.params.id

db.get('SELECT \* FROM users WHERE id = ?', [userId], function(err, row) {

if(err) {

return res.status(500).json({error: err.message})

}

if(!row) {

return res.status(404).json({error: 'User not found'})

}

res.json({user: row})

})

})

This way we make sure input data are specified and does not contain any invalid charactors.

A screenshot of a computer

Description automatically generated

However, it wouldn't care if there is an sql injection such as getting an input i.e 1 OR 1 = 1. Normaly this will work and should return all users.  
SQL VALID STATEMENT will return true: SELECT \* FROM users WHERE id = 1 OR 1 = 1 We should be worried about this kind of input.

Since we managed to implement a secure way to get user input we shouldn't worry about this, on this secure url.

**Access specific user. Can be done by anyone.**

!IMPORTANT But is weak and anyone can do an SQL injection.

http://localhost:3000/insecure/user/2 //Will display only 1 user.

http://localhost:3000/insecure/user/1%20OR%201=1 //This url will work because of the way the function is implemented. It will display all users.

WE ARE USING

1 OR 1=1

Example the following sql command should return everything on that table. SELECT \* FROM users WHERE id = 1 OR 1 = 1

Because of the way that the methods was implemented.

app.get('/insecure/user/:id', function(req, res) {

const userId = req.params.id

const sql = "SELECT \* FROM users WHERE id = " + userId

db.all(sql, function(err, rows){

if(err) {

return res.status(500).json({error: err.message})

}

res.json({users: rows})

})

})

It does not validate or check user input.

Here we will show you how the same sql injection we did on the secure url, but now we will do it on an insecure url

<http://localhost:3000/insecure/user/1%20OR%201=1>This sql inject successfully, because of how we implemented it, which disregard the input validation.

A screenshot of a computer

Description automatically generated