

PT Report

SuperDuperMarket

Executive Summary:

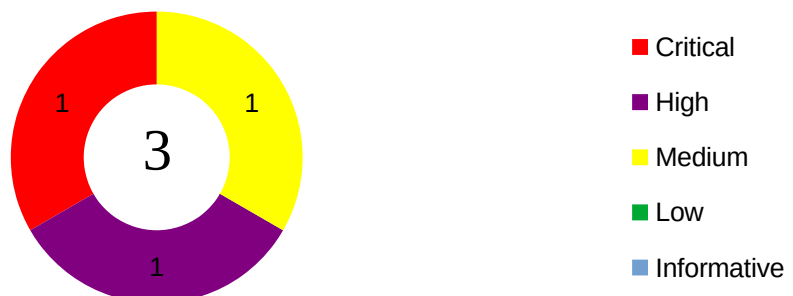
During the penetration test, I identified critical vulnerabilities in the SuperDuperMarket web application that could be exploited to compromise sensitive data and application integrity. These vulnerabilities allowed for unauthorized access to administrative files, manipulation of client-side data, and the execution of malicious scripts. Exploiting these vulnerabilities provided access to privileged information, including an admin token, and exposed critical flaws in the application's security model.

Conclusions:

My assessment revealed that the application's current security posture presents considerable risk due to the ease with which these vulnerabilities could be exploited and the potential impact of unauthorized access. The overall security level of the system is **moderate** due to the following:

- **Local File Inclusion (LFI)-like Behavior in Accessing Server Files:** This vulnerability enabled unauthorized access to sensitive server files, including `admin-api.js`, exposing administrative credentials.
- **Stored Cross-Site Scripting (XSS) in Barcode Generation:** The barcode generation mechanism allowed the injection of JavaScript, enabling the execution of unauthorized scripts and access to sensitive files.
- **Insecure Client-Side Data Manipulation in `itemsInCart` Cookie:** The application allowed manipulation of cart data via cookies, enabling unauthorized modifications to item IDs and quantities in the shopping cart.

These vulnerabilities highlight gaps in input validation, client-side security, and server access control mechanisms, which could lead to broader system exploitation if left unresolved.



Finding Details:

Vuln-001 Local File Inclusion (LFI)-like Behavior in Accessing Server Files (**Critical**)

Description:

Local File Inclusion (LFI) is a vulnerability that allows attackers to access files on the server by including them through insecure application functionality. In this case, the behavior mimics an LFI, as it enabled unauthorized access to sensitive server files. While traditional LFI involves exploiting server-side file inclusion flaws directly, this vulnerability leveraged client-side script injection to achieve similar results, exposing sensitive information such as the admin token.

Details:

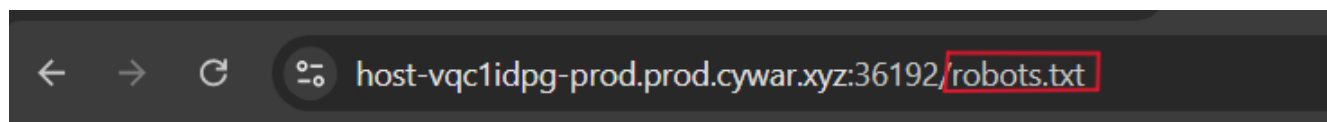
During the assessment, I discovered that the application was vulnerable to an LFI-like behavior, allowing unauthorized access to sensitive server files. The vulnerability was identified when the robots.txt file disclosed the existence of /admin-api.js, a sensitive administrative script. Based on this information, I hypothesized its location as /srv/node/admin-api.js and validated this path using a malicious payload executed via the Stored XSS vulnerability.

The injected payload directed the browser to load the file, exposing its contents, including the admin token. This token is a highly privileged credential, enabling full administrative access to the system.

This vulnerability is classified as **Critical** because it exposes sensitive administrative credentials, which can lead to full system compromise. An attacker leveraging this vulnerability could escalate privileges, manipulate critical application settings, and access other sensitive resources. Combined with the ease of discovery and exploitation, this poses a severe threat to the application's integrity and security.

Evidence:

This vulnerability was I identified when I start to enumerate the website. I checked the robots.txt and discovered admin-api.js

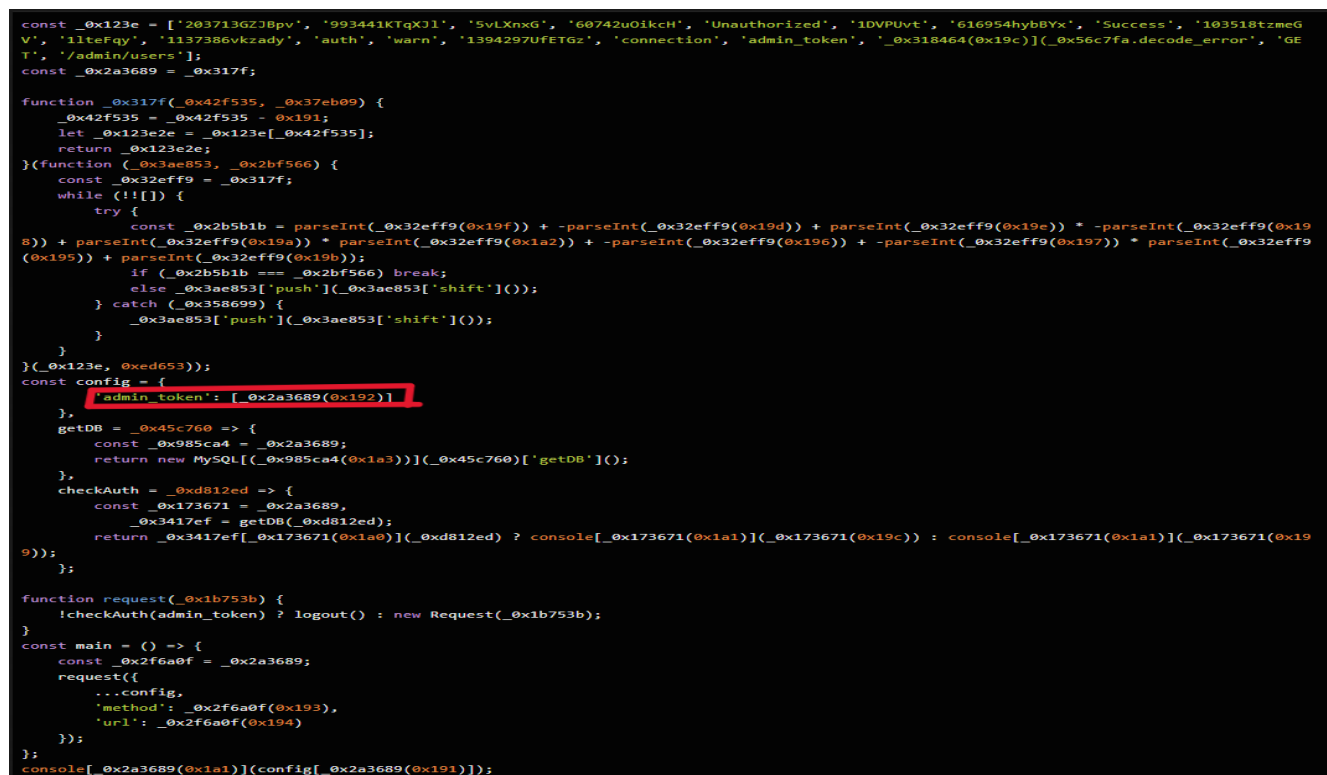


User-agent: *
Disallow: /admin-api.js

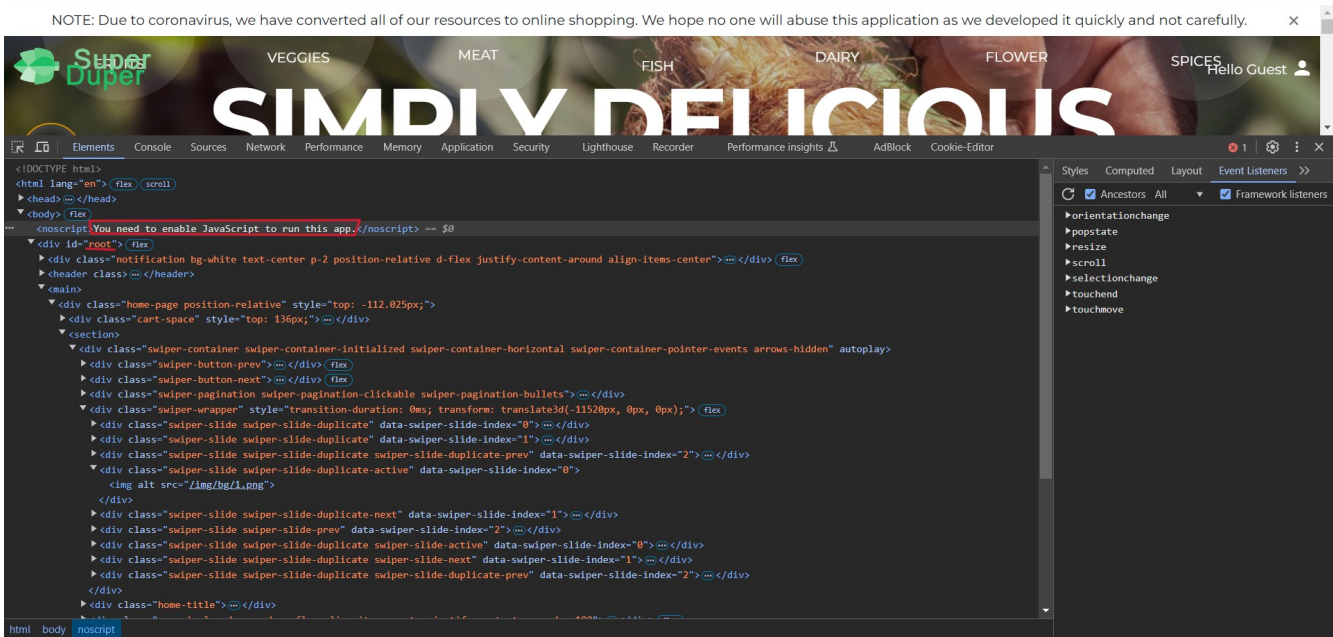
Then I checked the file and what is in it



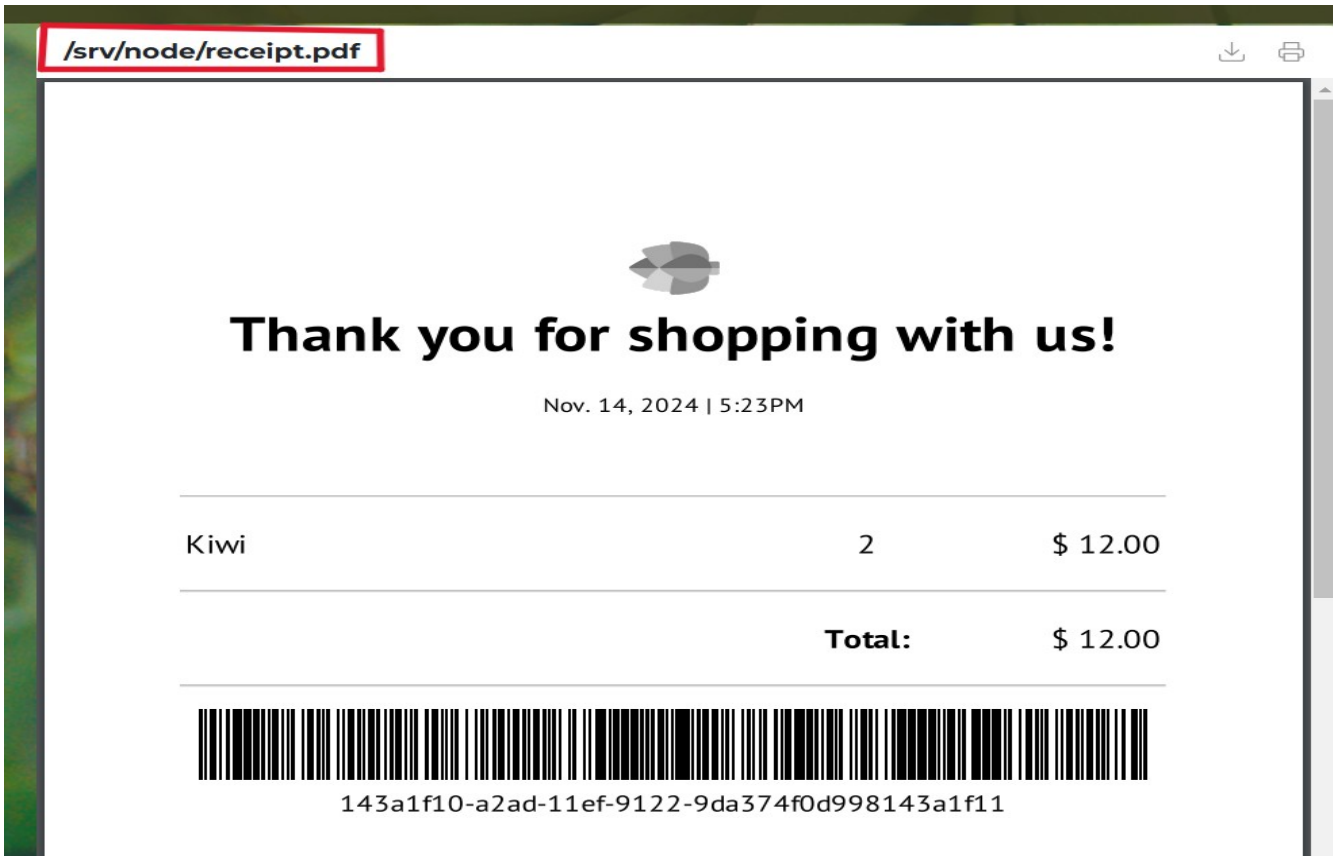
I found a script that was obfuscated and I did deobfuscated and then I discovered admin token as _0x2a3689(0x192), meaning is holding the admin token



I also viewed the page source in the website and discovered that I need to enable JavaScript to run this app



Then I searched the path of admin-api.js, I hypothesized that the file was located in /srv/node, based on its relevance to the Node.js framework. And also because the receipt I get was in the same path



I discovered that the payload could access server files through the file:/// protocol. This suggested a misconfiguration allowing local file access. And I searched for commands to open the script and I found a number of commands that I can use:

1. Using `window.location.href`

```
html
<script>window.location.href = 'file:///srv/node/admin-api.js';</script>
```

2. Using `self.location`

```
html
<script>self.location = 'file:///srv/node/admin-api.js';</script>
```

3. Using `location.replace`

```
html
<script>location.replace('file:///srv/node/admin-api.js');</script>
```

4. Using `window.open` **to Open in a New Tab**

- This will open the file in a new browser tab instead of redirecting.

```
html
<script>window.open('file:///srv/node/admin-api.js', '_blank');</script>
```

5. Using `location.assign`

```
html
<script>location.assign('file:///srv/node/admin-api.js');</script>
```

Message ChatGPT

I injected a payload that redirected the browser to file:///srv/node/admin-api.js. The browser successfully loaded the file's content, which contained sensitive information. (Using the XSS vulnerability. The details in the next vulnerability)

```
/srv/node/receipt.pdf
// [Yesterday 6:58 PM]
const config = {
  admin_token: 'dcb129fb258a43a525efaebb3dc7512d'
}

const getDB = (token) => {
  return new MySQL.Connection(token).getDB()
}

const checkAuth = (token) => {
  const db = getDB(token)

  if (db.auth(token)) {
    return console.warn("Success")
  } else {
    return console.warn("Unauthorized")
  }
}

function request(config) {
  if (!checkAuth(admin_token)) {
    logout()
  } else {
    new Request(config)
  }
}

const main = () => {
  request({ ...config, method: "GET", url: "/admin/users" })
}

console.warn(config.admin_token)
```

Remediation Options:

- **Restrict File Access:** Configure the server to prevent unauthorized access to sensitive files. Use access control mechanisms to ensure that only authorized processes or users can access files like admin-api.js.
- **Validate and Sanitize Input:** Ensure all inputs, including file paths or user-provided data, are properly validated and sanitized. Do not allow direct user input to determine file paths or access server-side resources.
- **Disable File URL Protocols:** If not required, disable the use of file:/// protocols for accessing server files via the browser. Configure the server to reject requests attempting to access files through this protocol.
- **Implement Role-Based Access Control (RBAC):** Enforce strict permissions for administrative files. Use role-based access control to limit access to files like admin-api.js to authorized users only.
- **Configure robots.txt Securely:** Avoid listing sensitive or administrative files in robots.txt. Instead, secure these files through proper authentication and authorization mechanisms.
- **Monitor and Log File Access:** Enable logging for all file access requests and monitor for unusual activity. This helps detect unauthorized access attempts and provides an audit trail for forensic analysis.
- **Use Content Security Policies (CSP):** Deploy a robust CSP to prevent the execution of unauthorized scripts or file inclusions on the client side.

Appendices:

- [Local File Inclusion \(LFI\)](#)
- [File Protocol](#)

Vuln-002 Stored Cross-Site Scripting (XSS) in Barcode Generation (High)

Description:

Stored Cross-Site Scripting (XSS) occurs when an application stores malicious input provided by an attacker and then renders it in a user's browser without proper validation or sanitization. In this case, the vulnerability was identified in the barcode generation functionality, which allowed JavaScript to be injected into the barcode's SVG content. When the barcode was rendered, the malicious script executed in the user's browser, potentially enabling unauthorized actions such as accessing sensitive files or stealing session data.

Details:

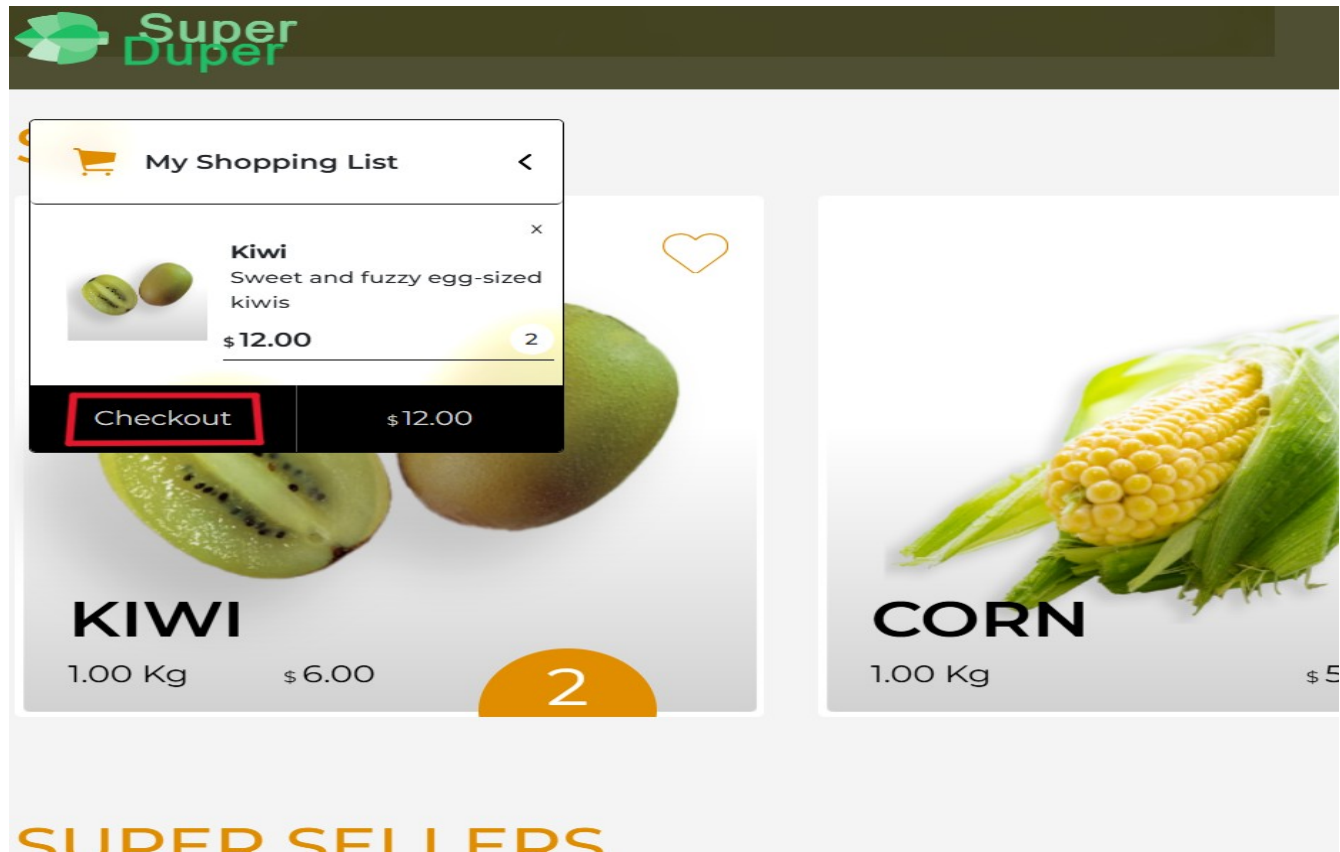
During the assessment, I discovered that the barcode generation mechanism embedded user inputs directly into the SVG barcode without sanitizing them. This allowed me to inject JavaScript code that executed when the barcode was rendered. Using Burp Suite, I intercepted the POST request sent to `/api/checkout`, which included the cart details and the barcode's SVG content. I modified the SVG content to include the JavaScript payload. The payload instructed the browser to load the file `/srv/node/admin-api.js` when the barcode was rendered. Upon completing the transaction, I viewed the barcode in the browser, which triggered the execution of the malicious JavaScript. This confirmed that the application was vulnerable to Stored XSS, as the injected script executed within the user's browser environment. The vulnerability was facilitated by:

- Lack of input validation and sanitization in the barcode generation process.
- Failure to encode or escape user-provided inputs rendered in the SVG file.

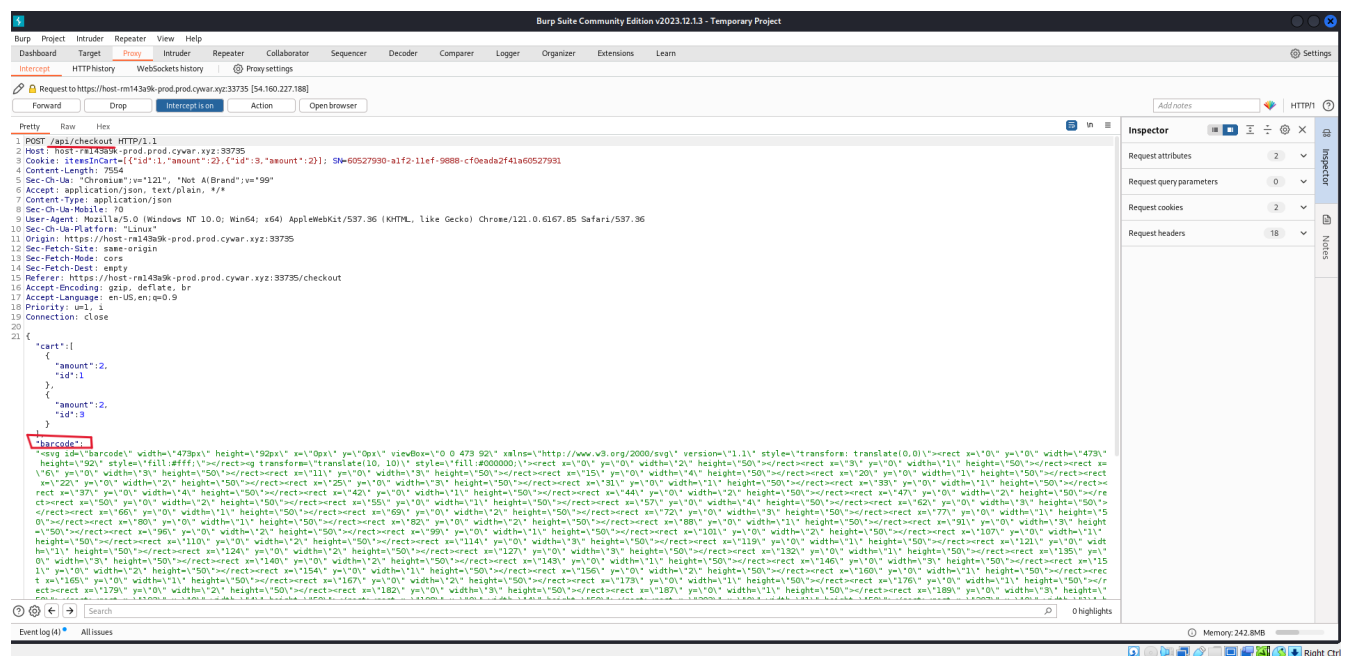
This allowed me to use the XSS vulnerability as a vector for executing arbitrary JavaScript code.

Evidence:

After I enumerate the website, I add item to the shopping list and press checkout



Then I send the request to the burp suite to intercepted when the barcode is being generated

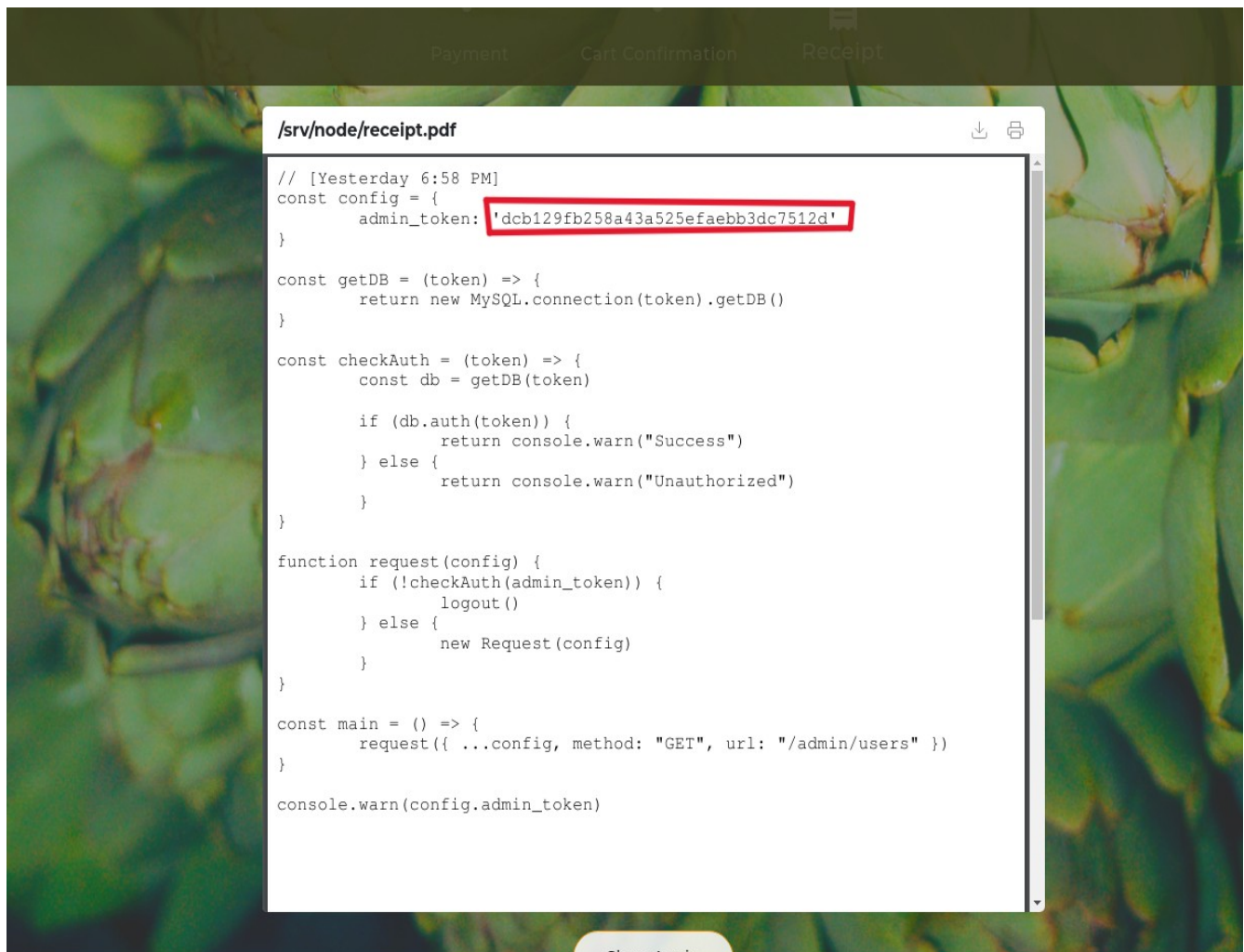


After I catch the barcode in burp, I notice the SVG structure, that allow me to injected JavaScript and create the XSS attack



Then I modify the intercepted SVG to included JavaScript inside <script> tag and add the script within a <text> or <rect> tag. (<script>self.location = <file:///srv/node/admin-api.js></script>)

Then I send the request and generated the barcode in the browser. When the barcode loads, the JavaScript executes automatically. And then I get the admin token (and the flag)



Remediation Options:

- **Sanitize and Validate Inputs:** Enforce strict input validation rules on all user-provided data. Allow only expected values (e.g., alphanumeric characters) and reject any input containing special characters (<, >, ", &) that could be used for script injection. Sanitize all inputs to remove potentially harmful characters or encode them to render them harmless.
- **Implement Output Encoding:** Encode all data before rendering it in the barcode SVG. For example, encode characters like <, > to their HTML-encoded equivalents (<, >) to ensure they are displayed as text rather than executed as code.
- **Deploy Content Security Policy (CSP):** Use a strong Content Security Policy to restrict where scripts can be executed from. Configure the CSP to block inline scripts and allow only trusted script sources.
- **Use Secure Libraries for SVG Generation:** Generate SVG barcodes using trusted libraries that enforce proper input handling and encoding. Avoid manually embedding user-provided data into SVG files.
- **Implement Server-Side Filtering:** Filter and sanitize inputs on the server side, in addition to any client-side checks, to ensure that malicious payloads cannot bypass client-side controls.

Appendices:

- [Cross-Site Scripting \(XSS\)](#)
- [Scalable Vector Graphics \(SVG\)](#)

Vuln-003 Insecure Client-Side Data Manipulation in itemsInCart Cookie (Medium)

Description:

Insecure Client-Side Data Manipulation occurs when an application stores sensitive data on the client-side, allowing users to manipulate it directly without proper server-side validation. In this case, the itemsInCart cookie stores the shopping cart's contents as JSON data. By modifying the cookie values, I was able to change the item IDs and quantities in the shopping cart, bypassing server-side restrictions. This vulnerability impacts the integrity of the shopping cart data and could potentially lead to unauthorized actions, such as price manipulation or purchasing items not intended for sale.

Details:

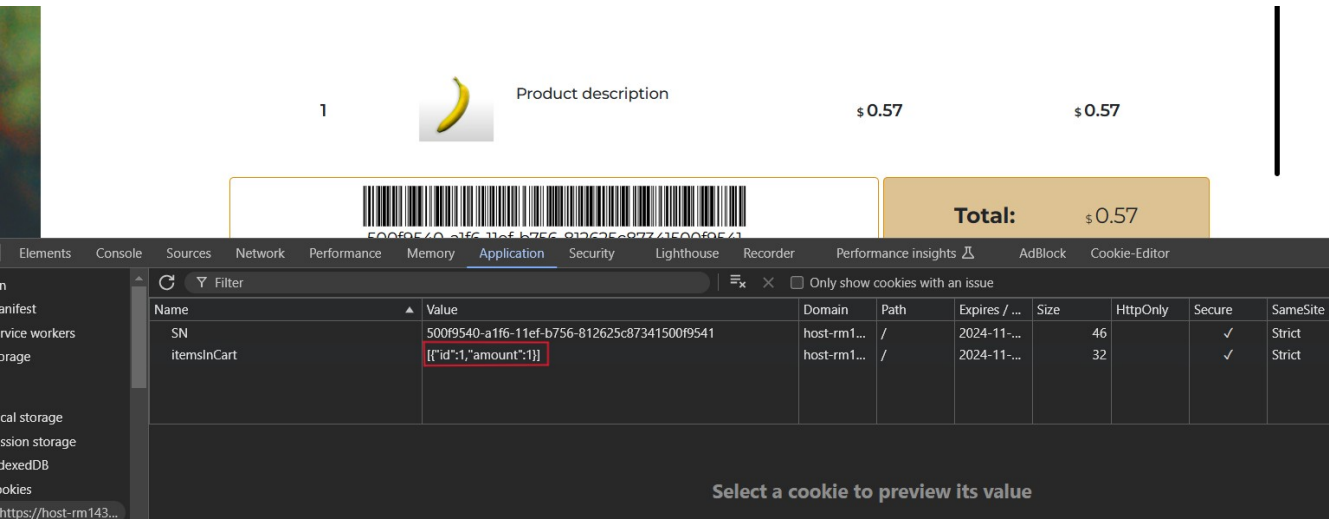
During the assessment, I identified that the itemsInCart cookie contained the cart's contents in a JSON format. (For example: [{"id":3, "amount":2}, {"id":1, "amount":1}]). Using the browser's developer tools or a proxy tool like Burp Suite, I modified the cookie values directly. Specifically: **Changed the item ID (id)** to another product ID that wasn't in the cart. **Increased the quantity (amount)** to an arbitrary number. After making these changes, I refreshed the page, and the application reflected the updated cart details, confirming that the server did not validate the integrity of the cookie's data. This lack of validation allows attackers to:

- Add unauthorized items to their cart by changing id values.
- Manipulate item quantities to potentially abuse bulk discounts or cause inventory discrepancies.
- Access restricted items by providing the ID of an item not normally available to the user.

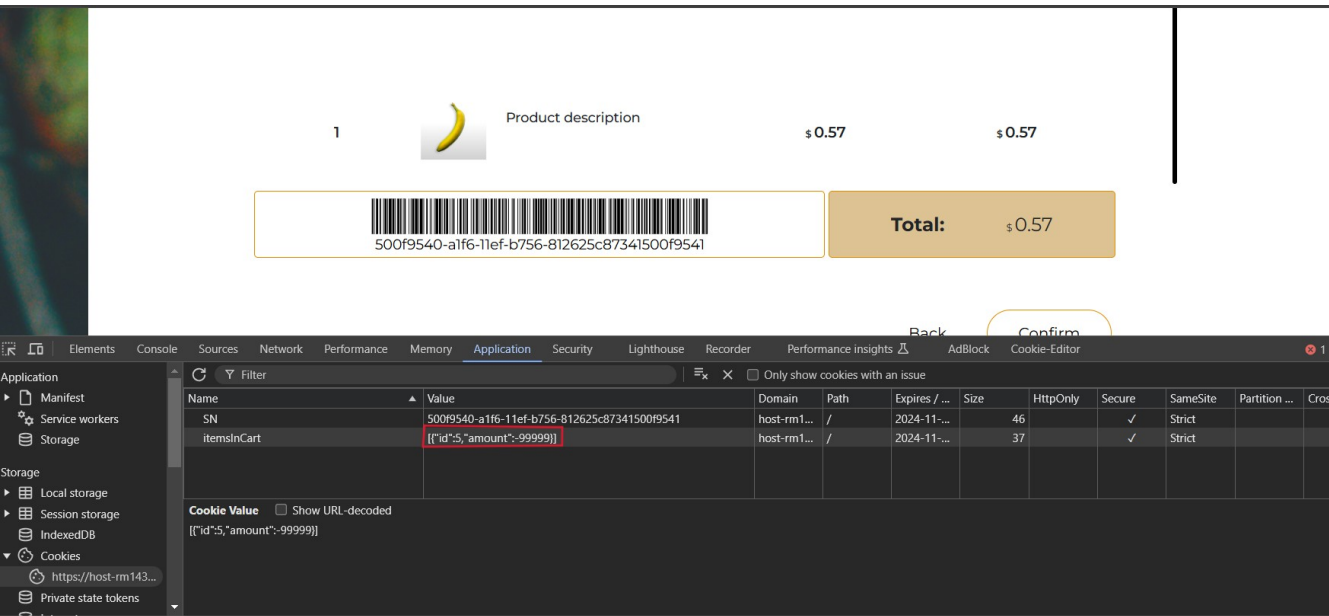
The vulnerability exists because the server trusts the client-side cookie values without verifying them against a secure backend record. This demonstrates insufficient server-side validation and poor design practices regarding sensitive data handling.

Evidence:

I choose item from the shopping list and I open the developer tools to go to application and cookies




Then I modify the id and the amount values in the cookie to manipulate the items



After I modify the id and the amount, you can see I got tomato instead banana and the amount of -99999 instead of 1


-99999



Product description

\$2.50

\$-249997.50


500f9540-a1f6-11ef-b756-812625c87341500f9541

Total: \$-249997.50

Back

Confirm


Application

Filter

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly	Secure	SameSite	Partition ...	Cross Site
SN	500f9540-a1f6-11ef-b756-812625c87341500f9541	host-rm1...	/	2024-11-...	46		✓	Strict		
itemsInCart	[{"id":5,"amount":-99999}]	host-rm1...	/	2024-11-...	37		✓	Strict		

Select a cookie to preview its value

Then I get a new receipt with a different item and different price



Thank you for shopping with us!

Nov. 13, 2024 | 7:42PM


Tomato

-99999

\$-249997.05

Total:

\$-249997.05


255fe3d0-a1f7-11ef-840c-673142532579255fe3d1

Remediation Options:

- **Store Cart Data on the Server:** Move all cart data to the server and use a session identifier stored in a secure, HTTP-only cookie to link users to their cart. This ensures the client cannot directly manipulate cart contents.
- **Validate All Cart Data Server-Side:** Verify item IDs, quantities, and other cart details against trusted server-side records before processing any actions. Reject any invalid or unauthorized changes sent from the client.
- **Sign or Encrypt Cookies** (If Server Storage is Not Feasible): If storing cart data in cookies is unavoidable, sign the cookies with a secure cryptographic hash to ensure their integrity, or encrypt them to protect confidentiality. Any tampered cookies should be rejected by the server.
- **Monitor for Suspicious Activity:** Enable logging and monitoring for unusual cart behavior, such as adding invalid item IDs or setting extreme quantities. This helps identify potential abuse early.

Appendices:

- [JavaScript Cookies](#)
- [Cookies in Chrome DevTools](#)