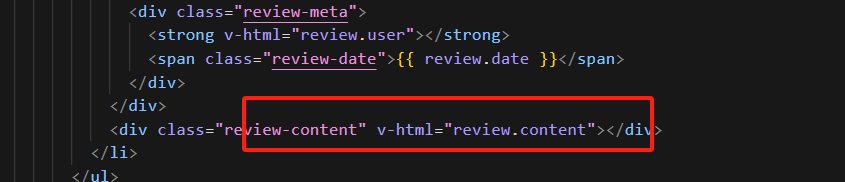
## Introduction

In modern web development, Vue.js, with its efficient component-based architecture and virtual DOM technology, has significantly enhanced the flexibility and maintainability of front-end development. However, due to the rendering characteristics of JavaScript and the component-based design of Vue, Vue applications are also susceptible to XSS (Cross-Site Scripting) attacks when handling user input. This article will demonstrate common types of XSS attacks in Vue applications, including reflected XSS and stored XSS, through the development of a simple e-commerce system, and will delve into the causes of these vulnerabilities and their unique manifestations in the Vue environment.

Vue.js provides developers with an efficient way to build user interfaces through its template syntax and data binding mechanisms. By default, Vue automatically escapes HTML in templates, meaning that any content inserted into the template will be safely escaped to prevent XSS attacks. For example, when binding data using {{ userInput }}, even if userInput contains HTML tags, it will be displayed as plain text after escaping.

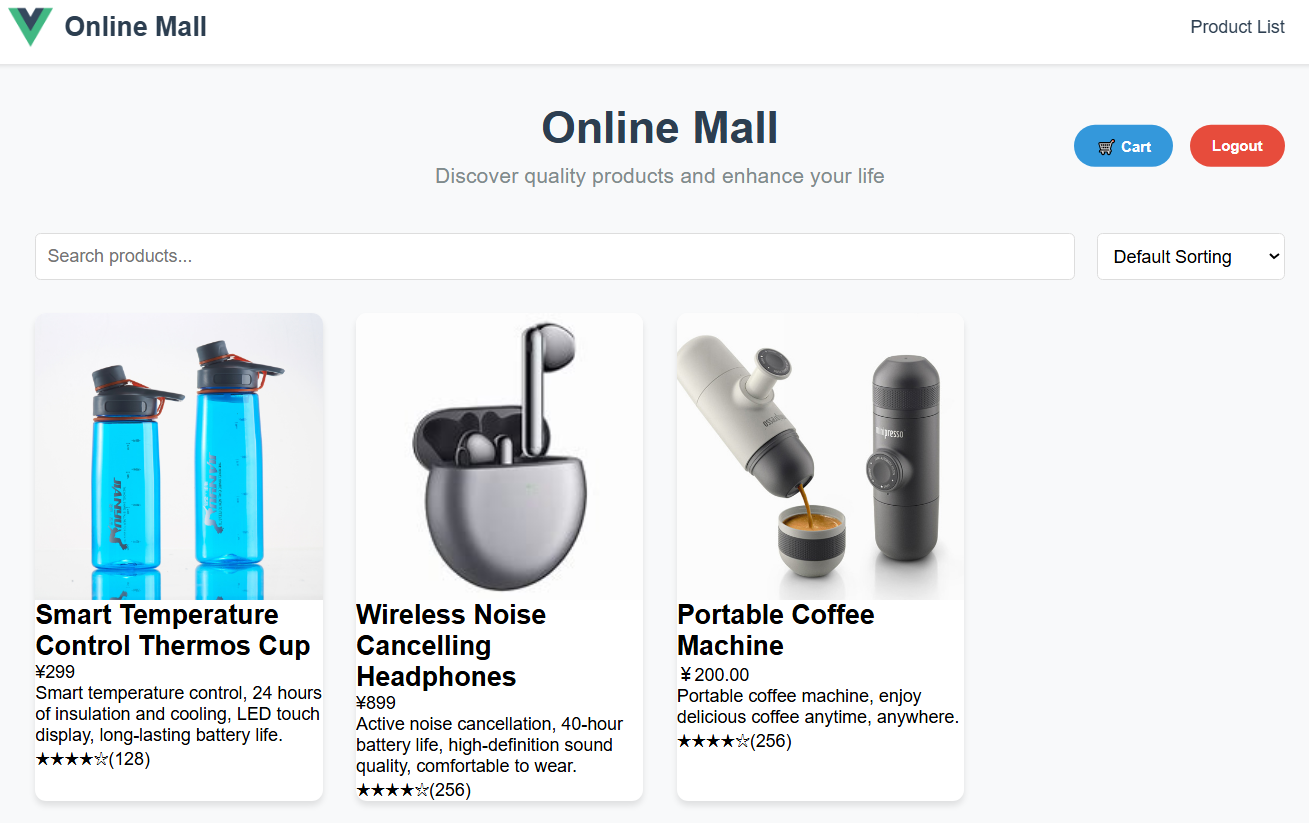


However, Vue also offers directives such as v-html to directly insert HTML content. While this can be very useful in certain scenarios, it poses a significant XSS risk if sufficient input filtering and escaping are not performed. For example, in the shopping cart or comment modules of an e-commerce system, if user-submitted JSON data is rendered directly through v-html without filtering, attackers can inject malicious HTML or JavaScript code via the name field. When this data is displayed on the page, the browser will execute the malicious code, posing a potential security threat.

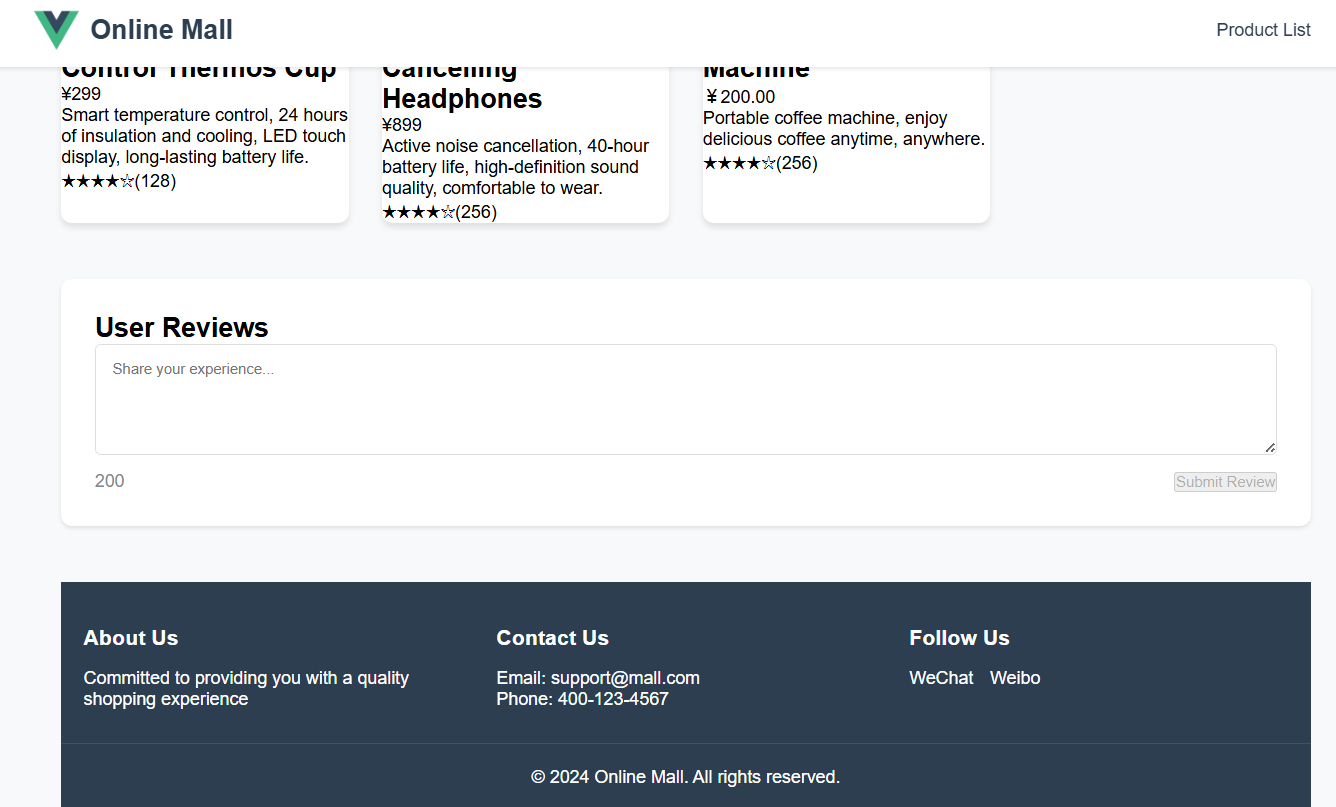
In ProductComponent.vue, there is an example of a stored XSS vulnerability. The component directly renders the product description using v-html. If the product description contains malicious scripts, these scripts will be executed.

In Products.vue, content is dynamically generated based on URL parameters and directly rendered on the page. The code uses v-html to render the name parameter input by the user. This can lead to events like onerror in the HTML being executed, triggering an XSS attack. Attackers can construct malicious URLs and inject parameters containing <script> tags or other HTML tags into the page, thereby executing unauthorized scripts.

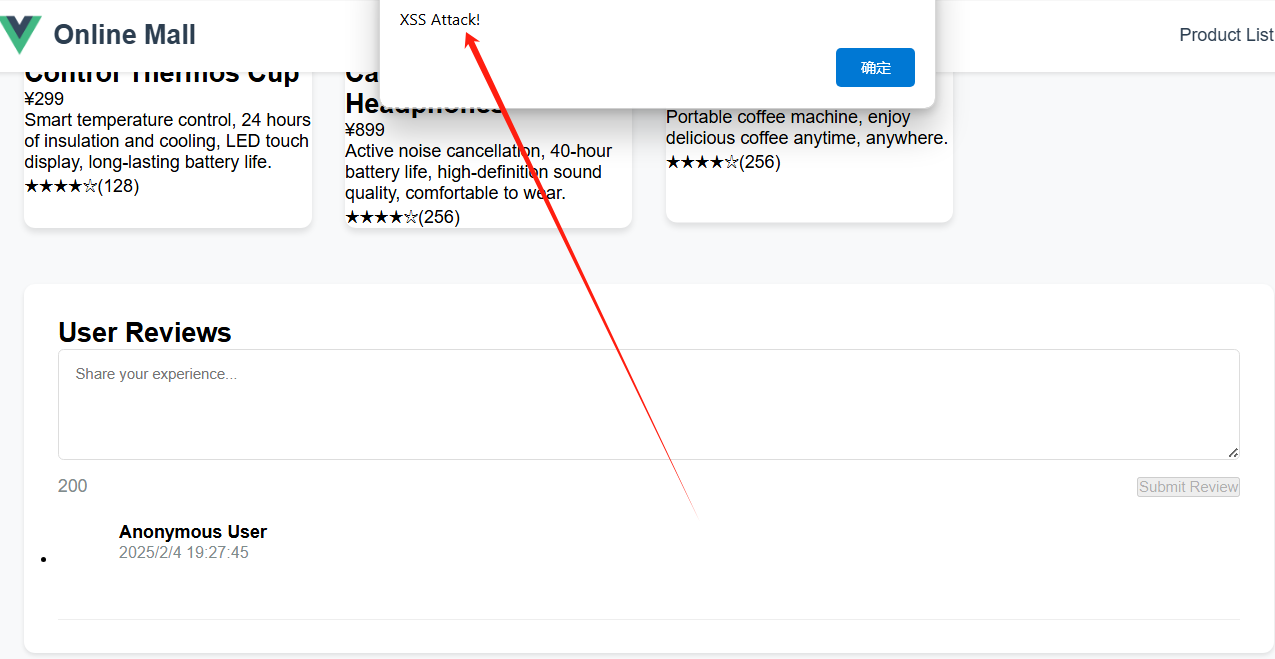
## Stored XSS (Cross-site Scripting) Attacks:



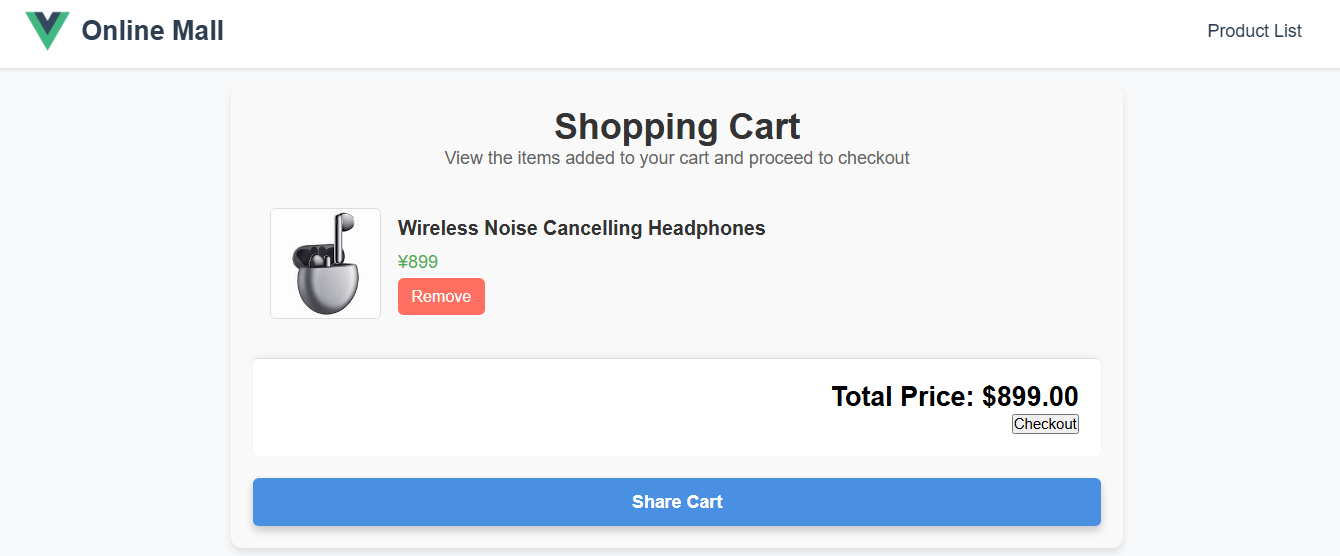
Upon examining the e-commerce system, it was noted that user comments are immediately displayed on the homepage. This suggests the potential presence of a stored XSS vulnerability, where attackers could inject malicious scripts using something like <img src="nonexistent.jpg" onerror="alert('XSS Attack!')" />.



Once the attack is successful, the malicious code will persist and be triggered with every page refresh.

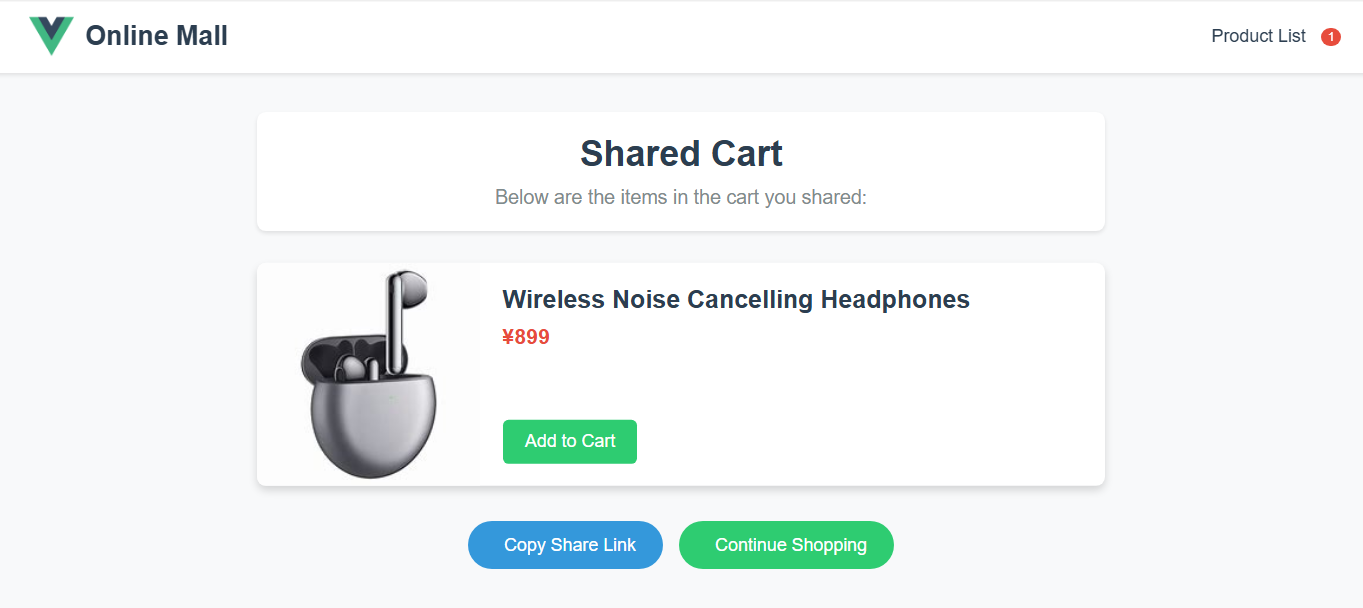


## Reflected XSS:

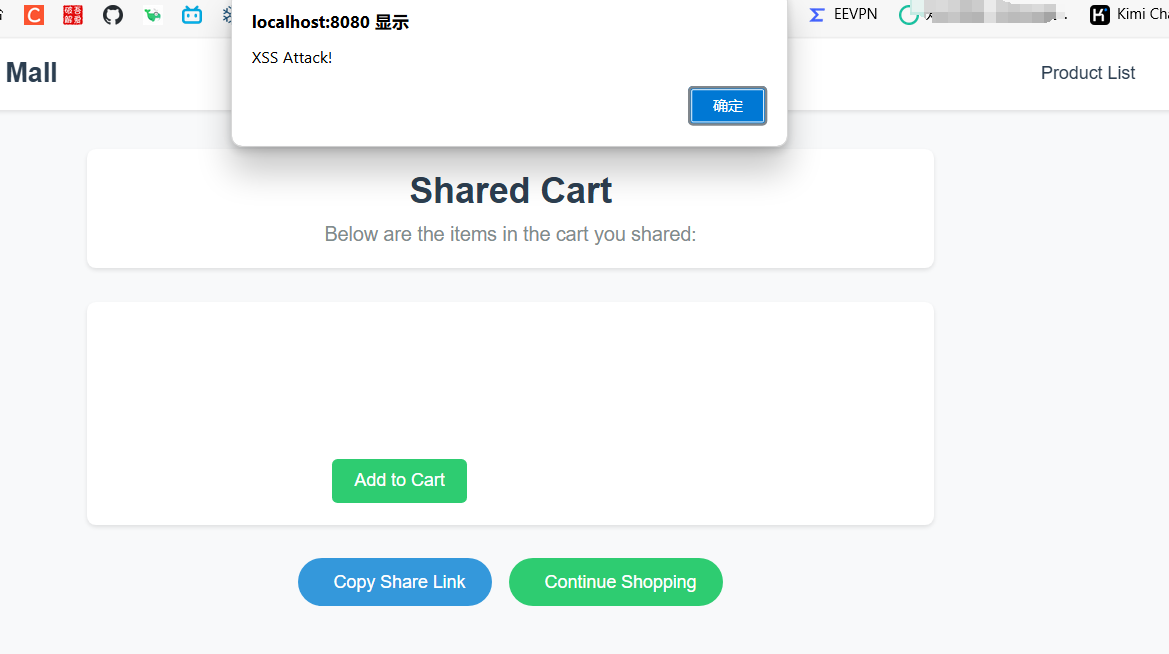


A reflected XSS vulnerability has been identified in the shared shopping cart feature. When clicked, it attempts to access a specific URL.

http://localhost:8080/share?id=1&name=Wireless%20Noise%20Cancelling%20Headphones&price=%C2%A5899&image=https%3A%2F%2Fth.bing.com%2Fth%2Fid%2FOIP.RVJZ2Nmvn24ILGPA7vAeTQHaHa%3Fw%3D189%26h%3D189%26c%3D7%26r%3D0%26o%3D5%26pid%3D1.7&description=.

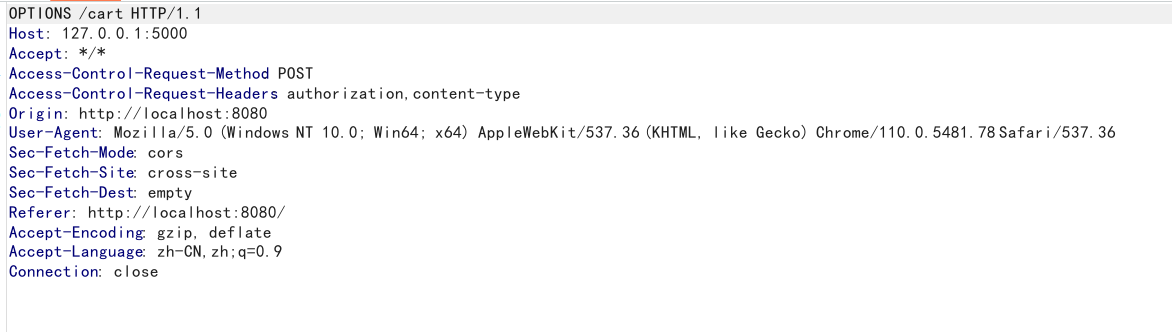


There is a possibility that the URL contains a malicious injection script: <http://localhost:3000/share?id=1&name=%3Cimg%20src=%22nonexistent.jpg%22%20onerror=%22alert(%27XSS%20Attack!%27)%22%20/%3E.> An XSS attack could potentially be executed using this link with an XSS payload.

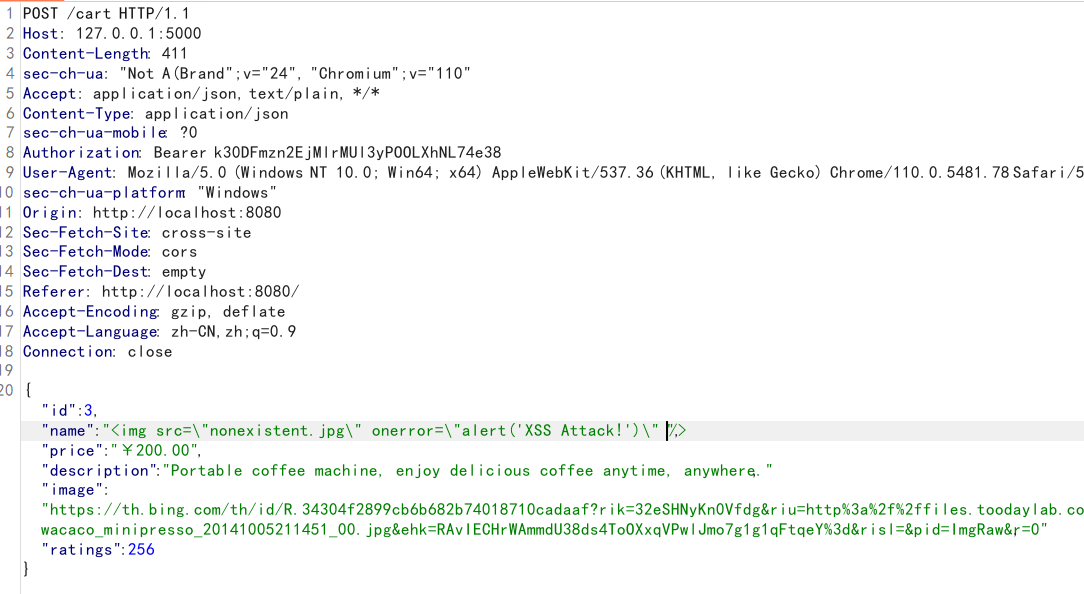


## JSON XSS:

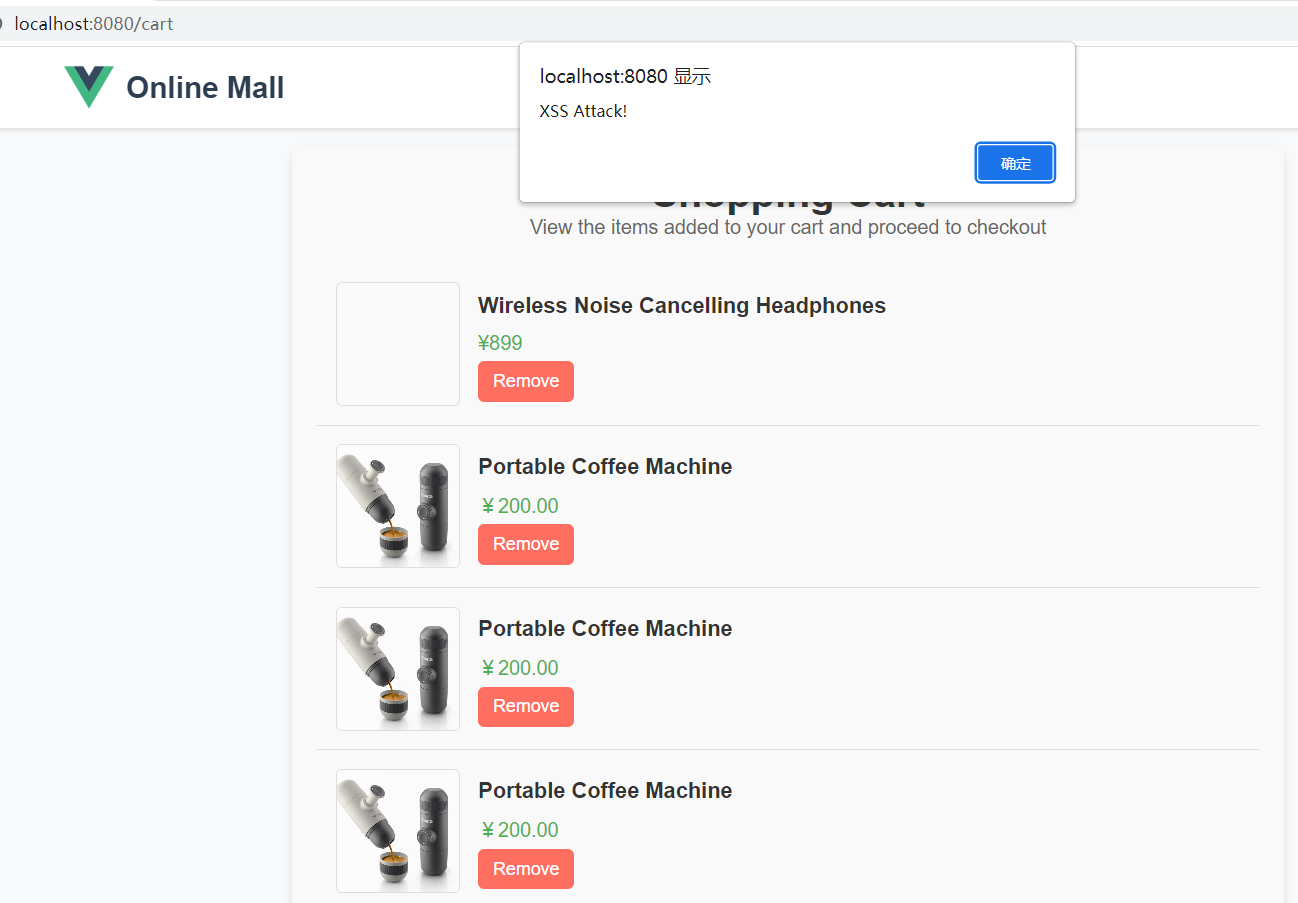
When intercepting the network traffic with Burpsuite, it was discovered that the request sent to add items to the shopping cart included detailed product information.



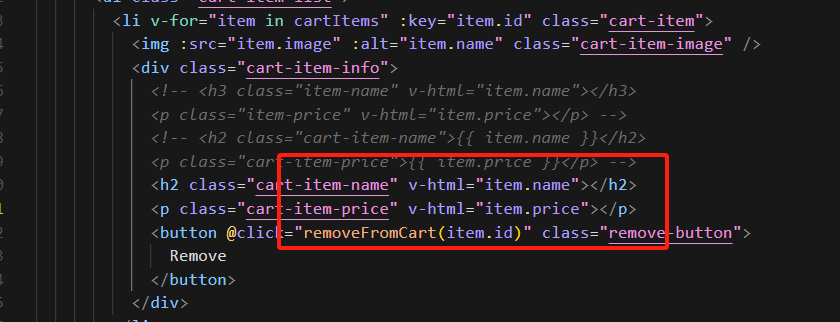
Modify the product name to Payload:"name":"<img src=\"nonexistent.jpg\" onerror=\"alert('XSS Attack!')\" />".



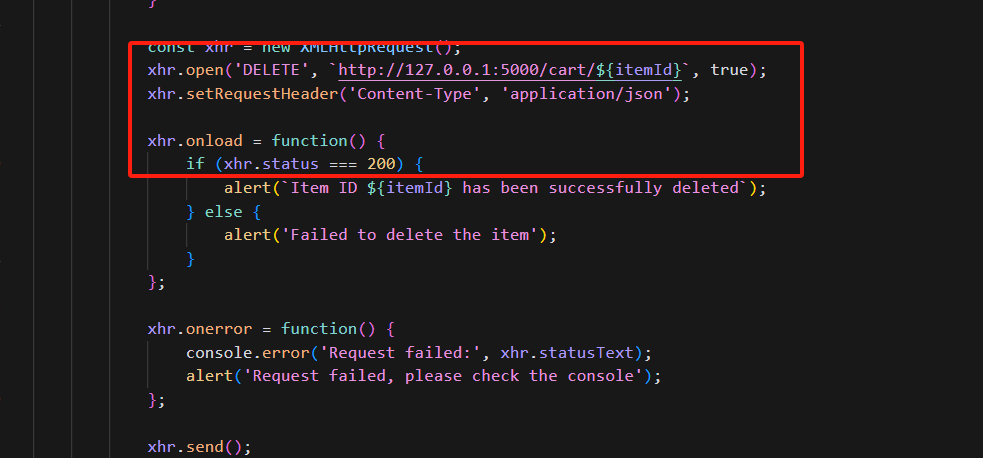
After placing the package, it was successfully added, and then the shopping cart was checked, and XSS was successfully exploited.



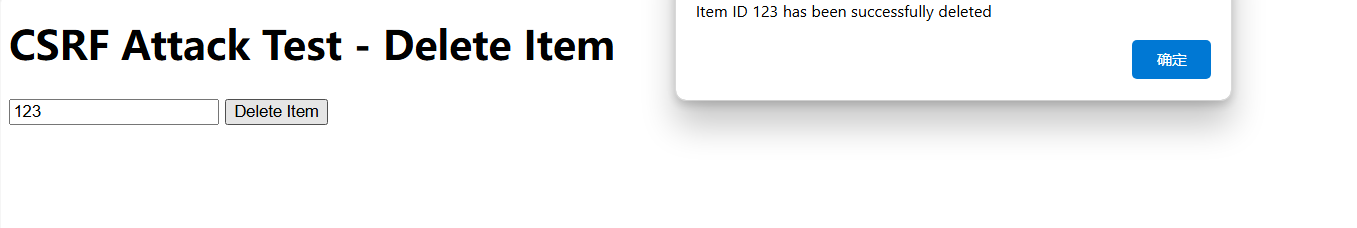
By analyzing the code for displaying shopping cart items, we can also find that an unsafe rendering function is used. This function causes user input to be executed as code, thus causing a vulnerability.



## CSRF:

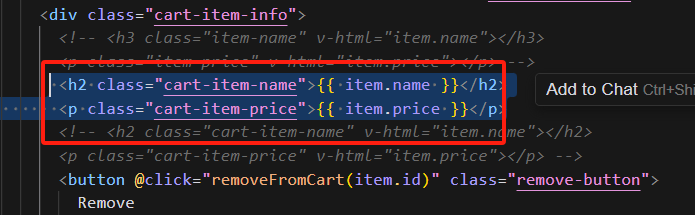


To test CSRF attacks, I have developed a test page named test.html. As illustrated in the figure above, the relevant trigger code is displayed. Upon clicking the button, the CSRF code is activated, prompting the browser to send a request to the backend to delete the shopping cart item.



## How to protect:

In Vue.js, the v-html directive is used to directly render HTML strings into the template. While this provides convenience for displaying dynamic content, it bypasses Vue's default HTML escaping mechanism, which can leave the application vulnerable to Cross-Site Scripting (XSS) attacks. Therefore, it is recommended to avoid using v-html when developing Vue applications. Here are some alternative solutions and best practices:



Using Text Interpolation: For most use cases, you can safely insert content using text interpolation (Mustache syntax {{ }}). Vue automatically escapes all content inserted via Mustache syntax, effectively preventing XSS attacks.

<template>

<div>

<p>{{ rawHtml }}</p>

</div>

</template>

In the above code, even if the rawHtml variable contains HTML tags or JavaScript code, they will be escaped and rendered as plain text, preventing execution.

Using Data Binding: If you need to dynamically handle styles or classes instead of directly inserting HTML, you can use Vue's binding capabilities, such as v-bind or the shorthand :, to safely bind data.

<template>

<div>

<p :class="dynamicClass">{{ content }}</p>

</div>

</template>