NahamStore: XSS

Introduction

The XSS, also called HTML Injection vulnerability, involves injecting code with HTML characters such as **<>'** that **interact with the web page**, allowing an attacker to ***inject code*** and ***change the appearance of a web page***.

This flaw, as with **SQLi**, is primarily due to the fact that, when programming a website, **appropriate sanitization was not performed** on the characters reserved for **HTML**, and instead of escaping them, they end up interacting with the web page. Although this may seem harmless at first glance, it has many security implications.

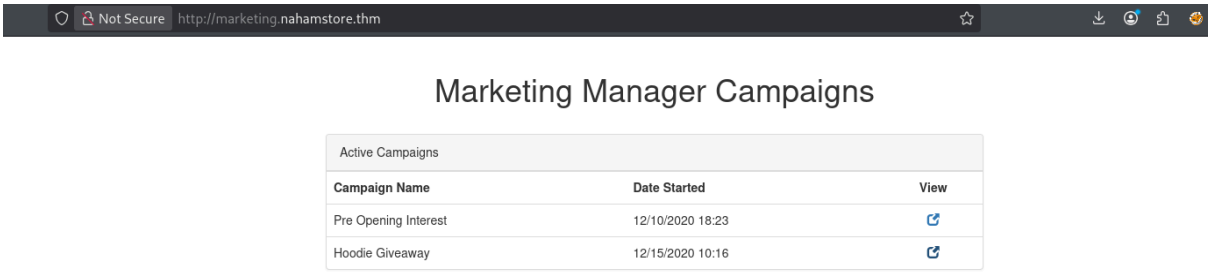
There are several types of **Cross-site scripting,** main ones are **Reflected XSS**, **Stored XSS** and **DOOM XSS.**

That's why in this section we're looking for all possible XSS errors that might exist in NahamStore.

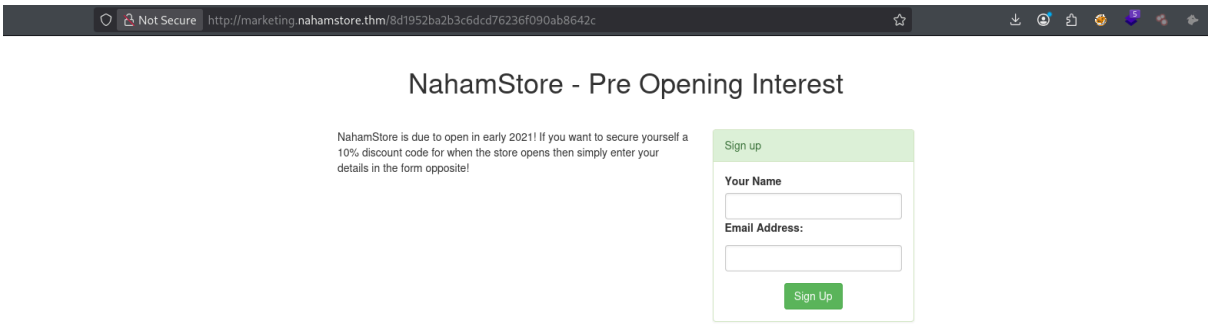
Looking for XSS

First XSS

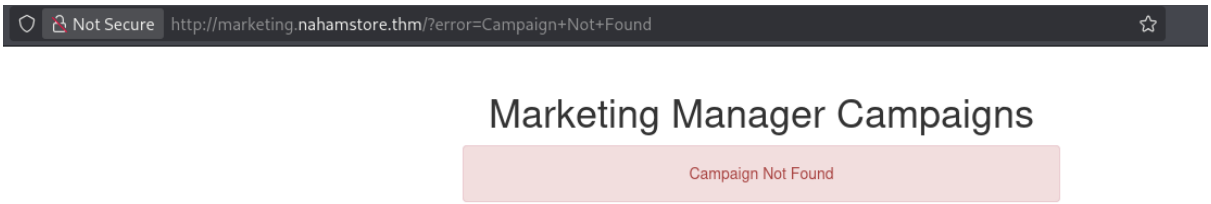
Starting with the subdomains, we didn't find anything particularly interesting except for the subdomain **marketing.nahamstore.thm**.



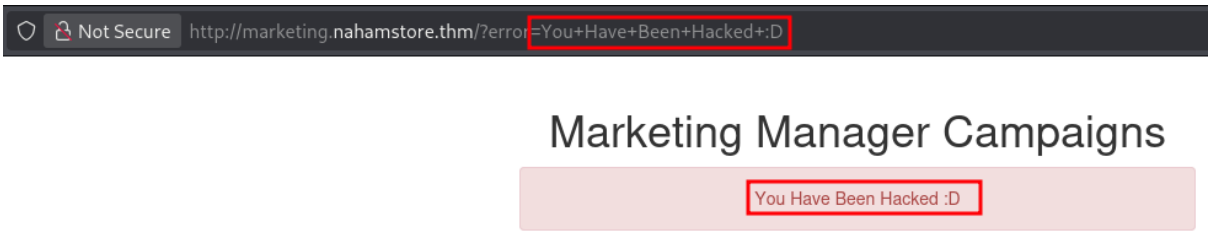
Clicking on one of the two links below "view" will take you to a Sign up panel.



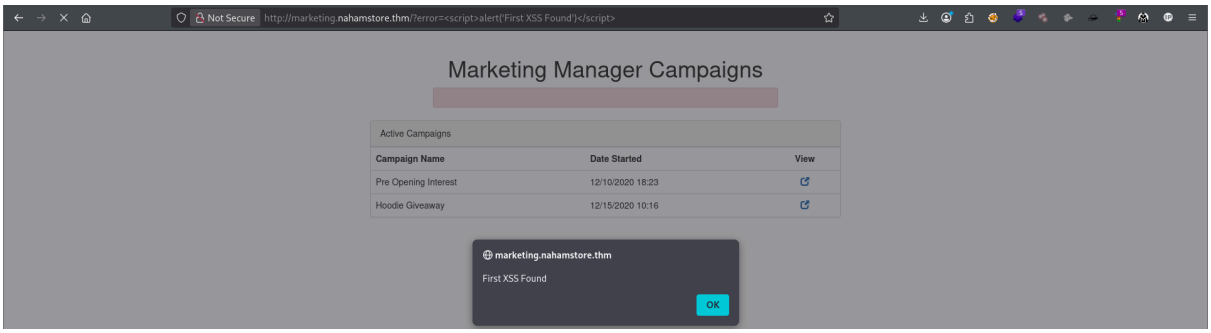
In the "Your Name" and "Email Address" input fields, we launched several XSS payloads, but they weren't vulnerable. However, when we tried to cause an error in the URL by changing a letter in the link, something quite interesting happened.



We can see the error message ***"/?error=Campaign+Not+Found"*** reflected on the website. Therefore, I tried changing the error message from the URL to a custom one; this was the result:



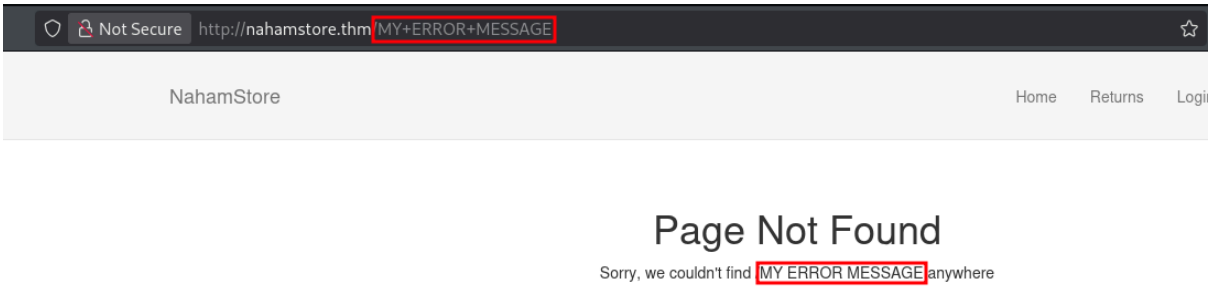
With this picture now clear, a simple XSS payload was injected into the URL with this result



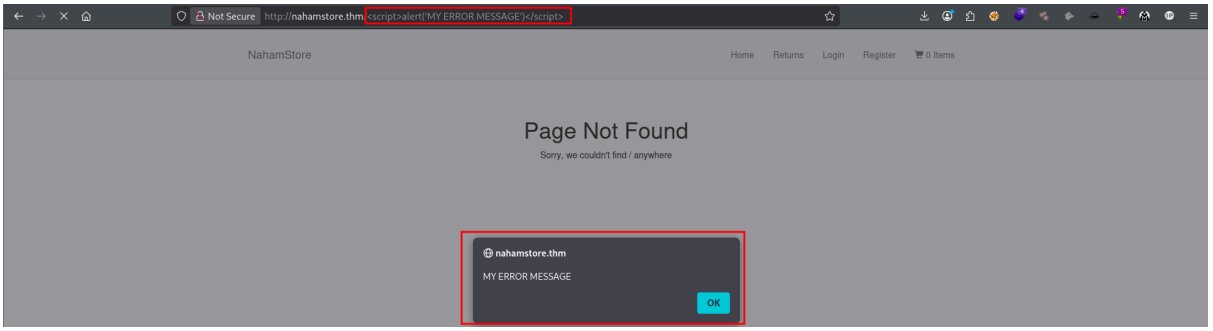
First xss found!!

❌ Second XSS

Looking for more XSS vulnerabilities, we return to the main website, where we will test for a similar error to the one we saw in the marketing subdomain from the URL, to see if it is reflected on the website



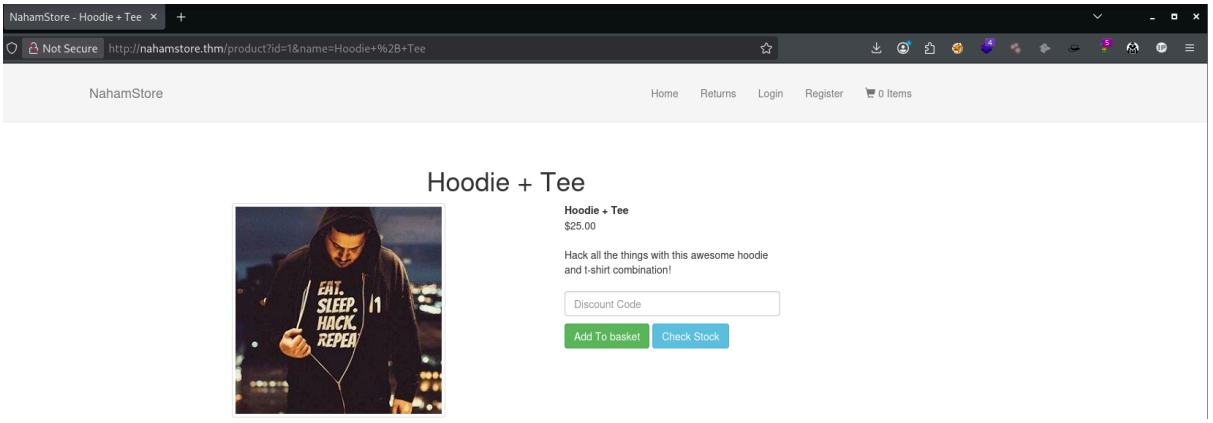
This type of error may seem harmless, and appears less obvious than the previous case; however, it never hurts to try.



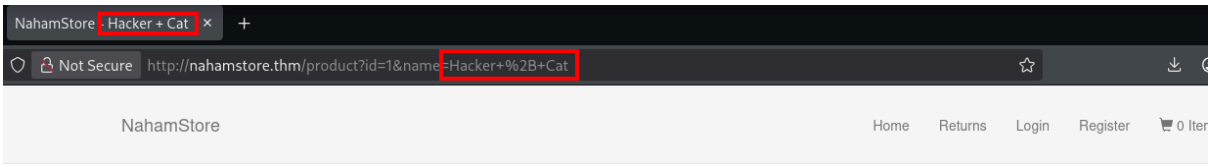
We have found our second XSS!

❌ Third XSS

The third section where we find the next XSS vulnerability is the section that refers to the products.

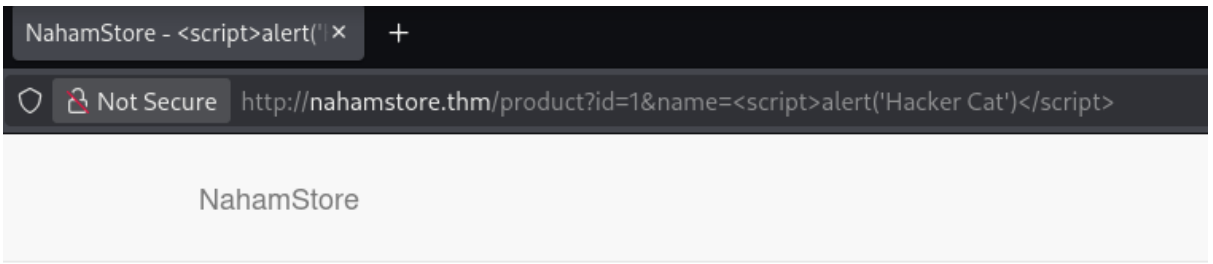


Again, let's play with the URL parameters and see what happens. I changed for “Hacked Cat”



Hoodie + Tee

As we can see, the URL parameter is manipulating the tab title; it's being placed in the text we've written there, and it's not including the part of the website that might seem more obvious. Now i will try a simple Payload to exploit XSS on this endpoint

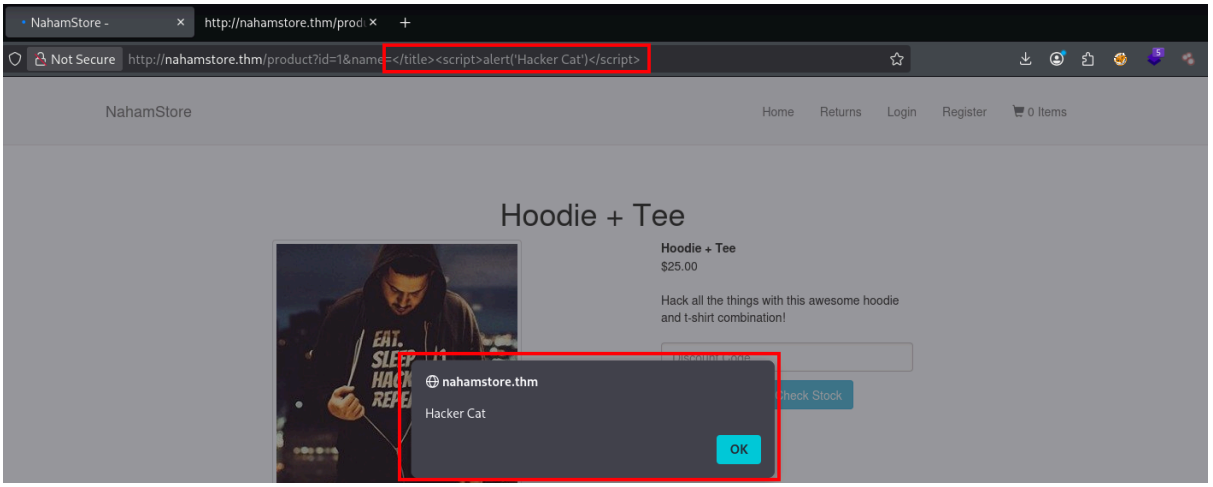


Although what we do is reflected in the tab title, it is not executed; to understand what is happening, we have to go to the source code.

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>NahamStore - <script>alert('Hacker Cat')</script></title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7
</head>
```

As we can see here, this part of the website we're controlling is contained within ***"title" tags***, which means that everything we place here will become a string unless we allow it to escape. ***We do this by closing the tag so that our payload can escape and be executed.***

To do this we need to place the title tag closed and then place our payload. It would be something like this ***</title><script>alert('Hacker Cat')</script>***

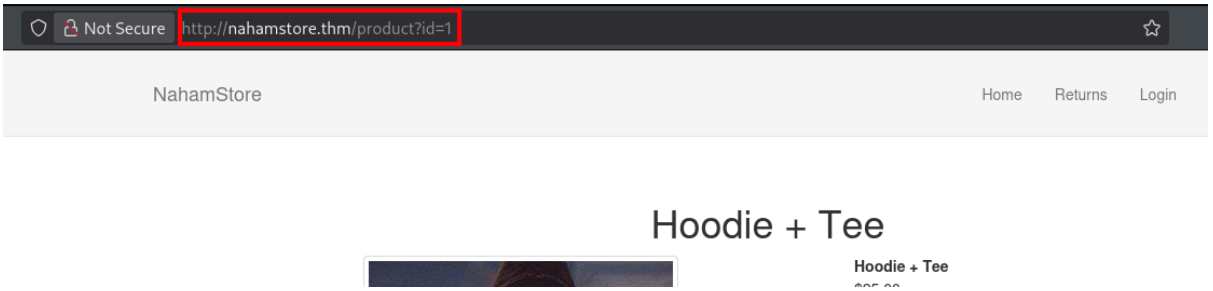


We have successfully exploited our third XSS!!

That's why, when searching for XSS, we have to be very observant.

❌ Fourth XSS

Our next XSS will be found in the same section, only it will be in the input that refers to the product ID.



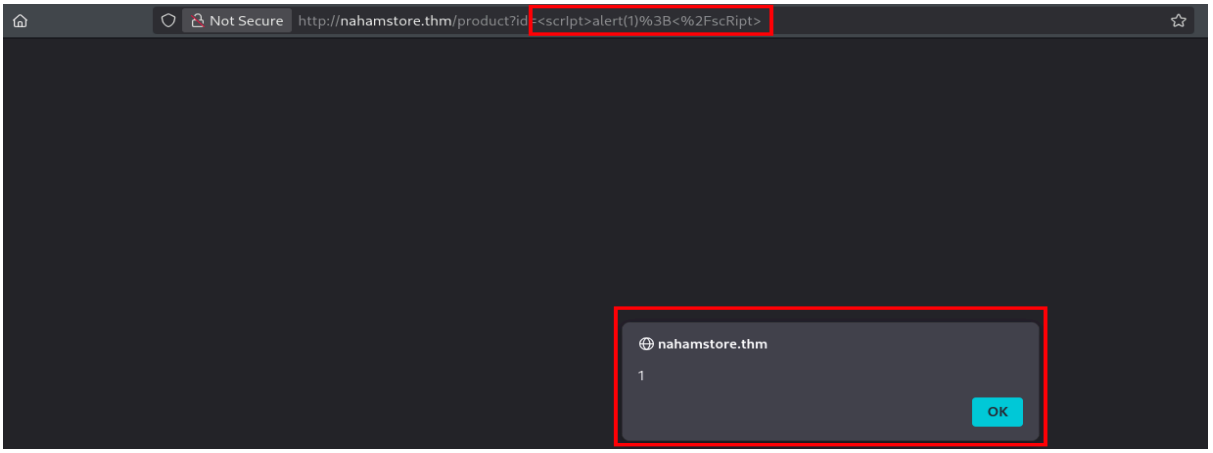
Interestingly, during our vulnerability analysis, **OWASP ZAP Proxy** detected that this endpoint is vulnerable to XSS.

http://nahamstore.thm (3)

Cross Site Scripting (Reflected) (1)

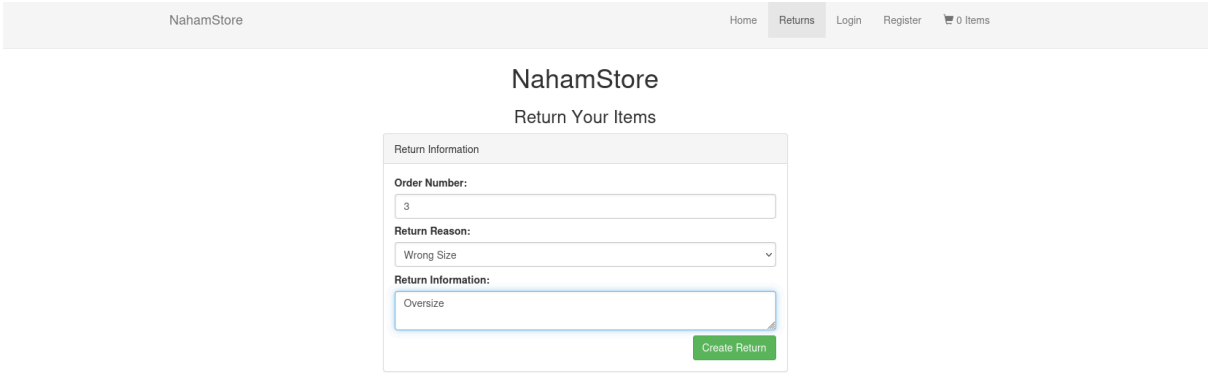
▶ POST http://nahamstore.thm/product?id=%3CscrIpt%3Ealert%281%29%3B%3C%2FscRipt%3E

By copying the link with the payload used by Zap Proxy, we will see this:

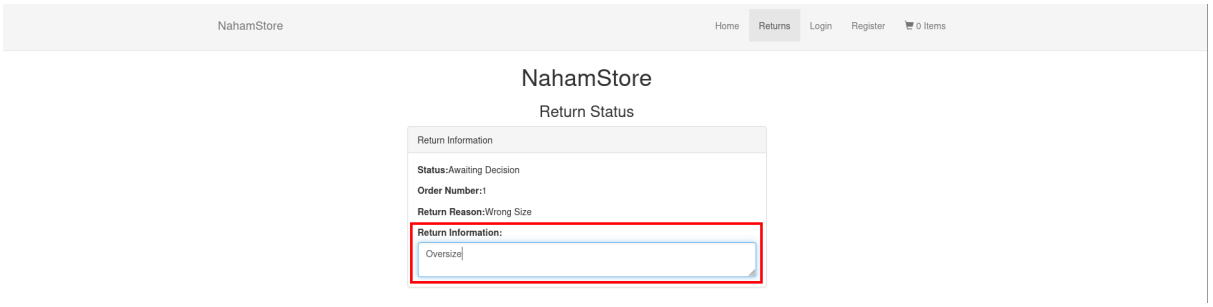


❌ Fifth XSS

For the next XSS, we go to the /returns section.



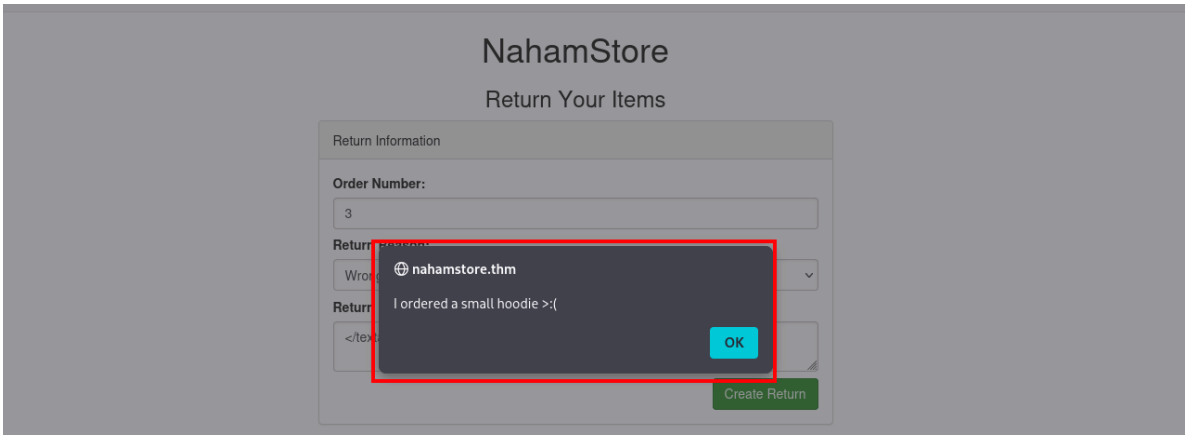
By filling it out, we see that we can, in a way, manipulate the information placed in the "Return Information" field.



If we look at this in the source code, we'll notice something interesting.

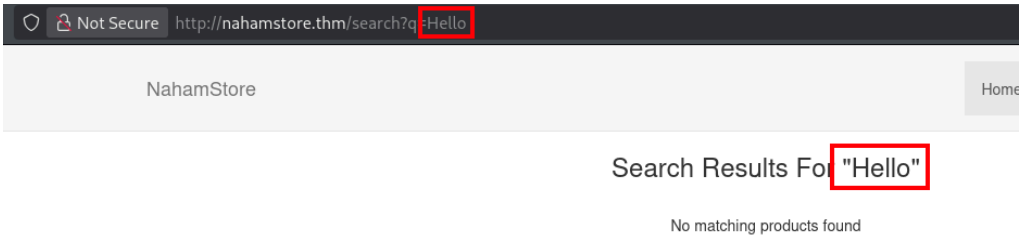
```
<div style="margin-top:7px"><label>Return Information:</label></div>
<div><textarea class="form-control">Oversize</textarea></div>
</div>
```

We can see that the information placed in the Return Information is between **"textarea"** tags. This is quite similar to the third case. Knowing this, for our payload to work, we must close the "textarea" tag and then place our payload inside. The payload would be something like this **</textarea><script>alert('XSS')</script>.** The result would be this:



❌ Sixth XSS

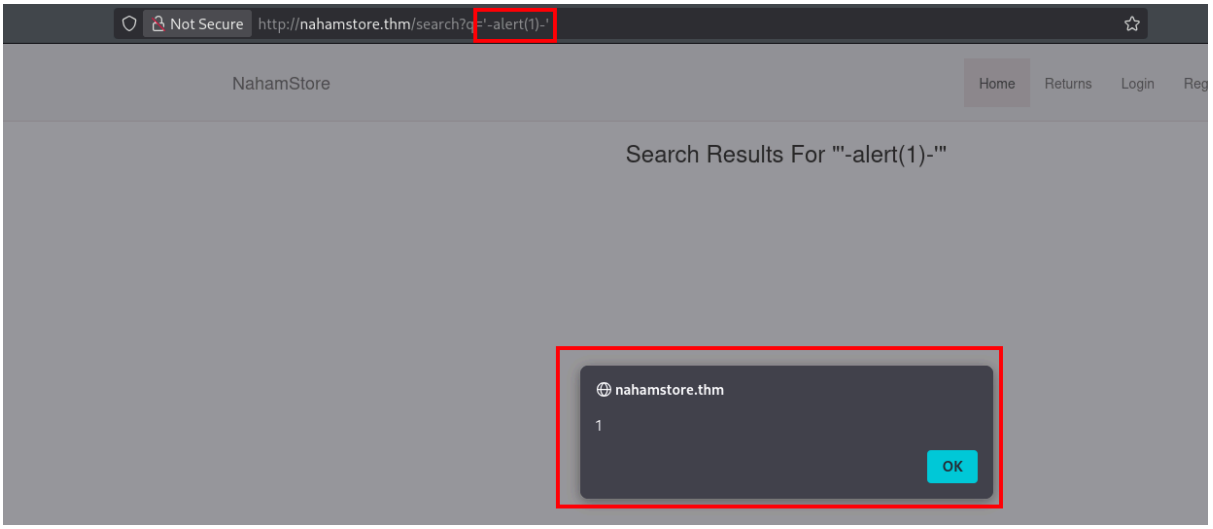
In this case, we need to go to the Search For Product bar. And to be honest, this was the XSS attack that took me the longest to identify.



During testing, we can see signs of XSS since what we wrote is reflected on the website. However, after several failed XSS attempts, reviewing the source code, and noticing any clear indication of vulnerability, I decided to use XSStrike with a wordlist of payloads, and the result was this:

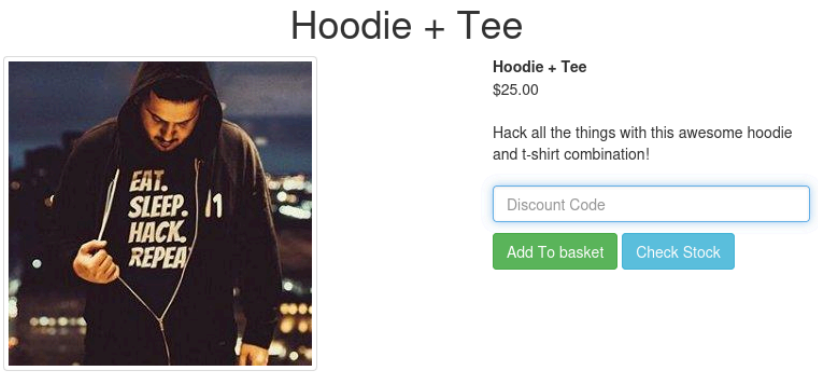
```
┌──(kali㉿kali)-[~/…/THM/Machines/NahamStore/XSStrike]
└─$ python3 xsstrike.py -u "http://nahamstore.thm/search?q=" -f /usr/share/wordlists/seclists/Fuzzing/XSS/human-friendly/XSS-BruteLogic.txt

      XSStrike v3.1.5

[!] [+] "onmouseover=alert(1)//
[!] [+] "autofocus/onfocus=alert(1)//
[!] [+] '-alert(1)-'  5/114
[!] [+] '-alert(1)//  6/114
[!] [+] \'-alert(1)// 7/114
[!] [+] %3Cx onxxx=alert(1)
 ~] Bruteforcing [q]: 114/114
```

The parameter is vulnerable, and when one of the payloads was taken and used, the result was this.

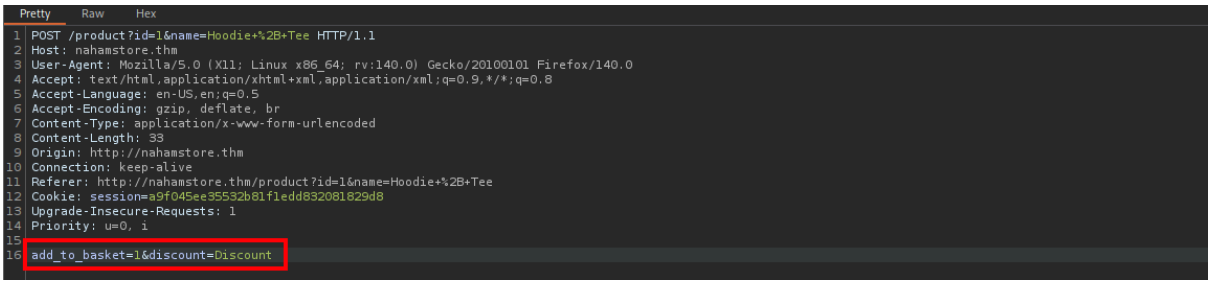


❌ Seventh XSS

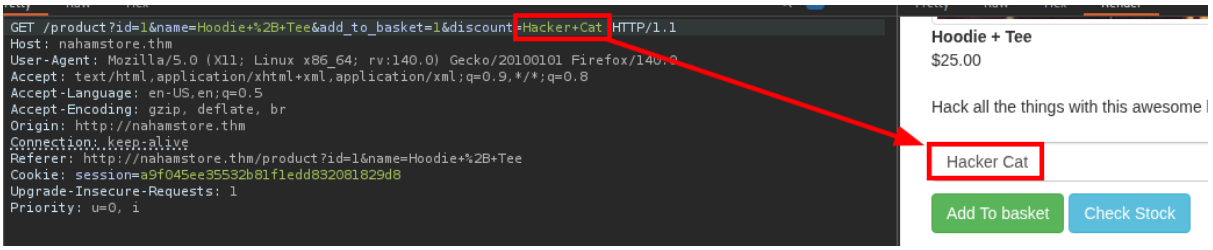
The next one can be found in the section **"/product?id=1&name=Hoodie+%2B+Tee".** Here we will focus on the **Discount Code** section.



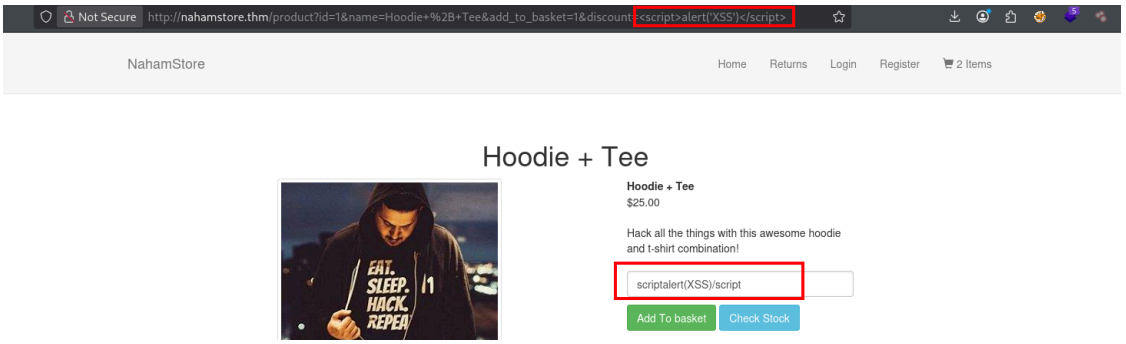
Here, when you add a discount code, or anything else we enter, an item will be created and added to the **Shopping Basket**. At first glance it doesn't seem like much, but if we intercept the request with Burp Suite we'll see more clearly the field where this "Discount" is being entered.



Here we can see more clearly the input fields, and that they are processed via the **POST** method. When sending the request to the Repeater and changing the request type to **GET**, we will see something interesting:

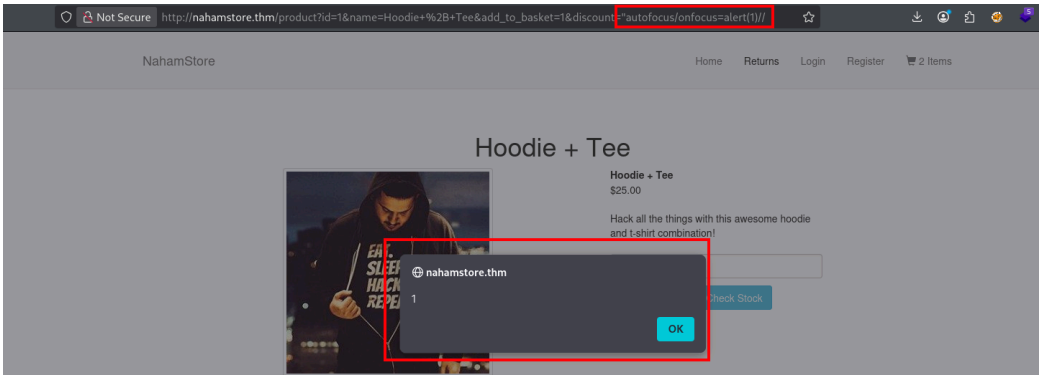


When changing and processing the request, we can see that anything we write from the URL will be reflected in the entry for "Discount Code". So I took the get response and put it in the search bar, and in the **“discount= input”** I used a simple XSS payload.



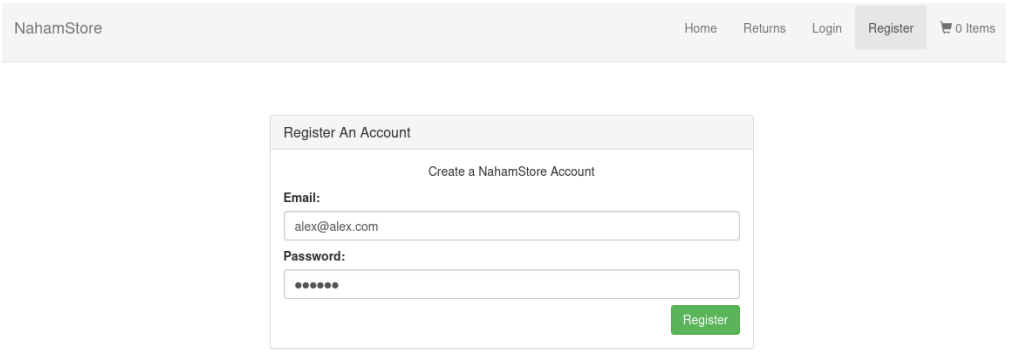
But this did not trigger an XSS. So I used XSStrike once again in search of a working payload.

And now, using it triggered our seventh XSS:

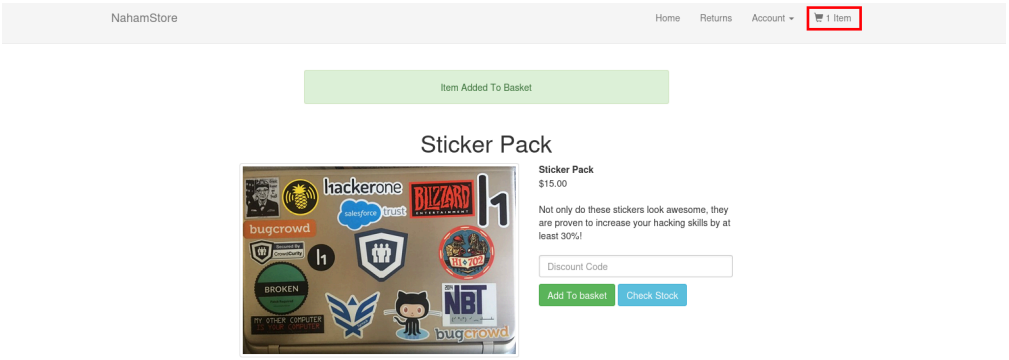


❌ **Last XSS**

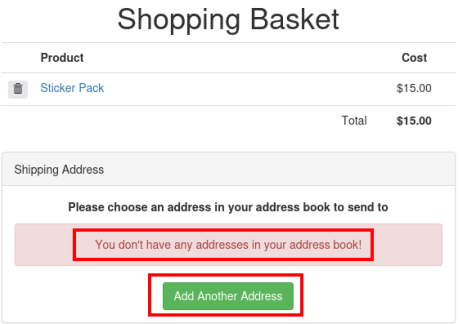
For that last step, we'll need to create an account and log in. The account to create was this one.



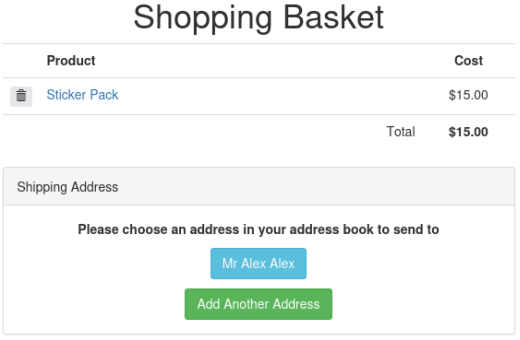
Once logged in, it will simulate the purchase of a product from the website.



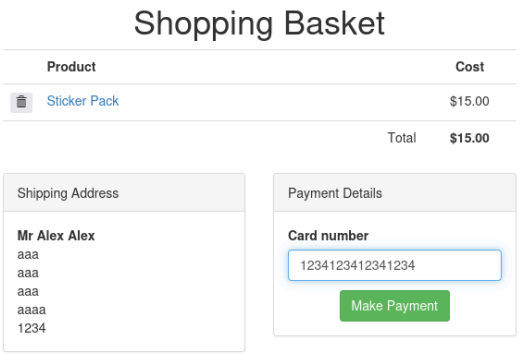
Once the product is added to the basket, we go there



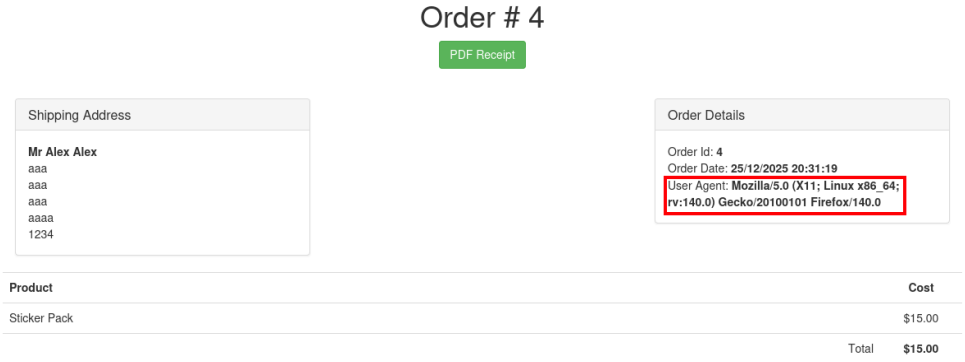
Here we can see that, to complete the payment, we need to add an address; we click on "Add Another Address" and we fill out the form with whatever we think is appropriate. And once the form is filled out, the Shopping basket section will look like this:



We completed the purchase by adding a "card".



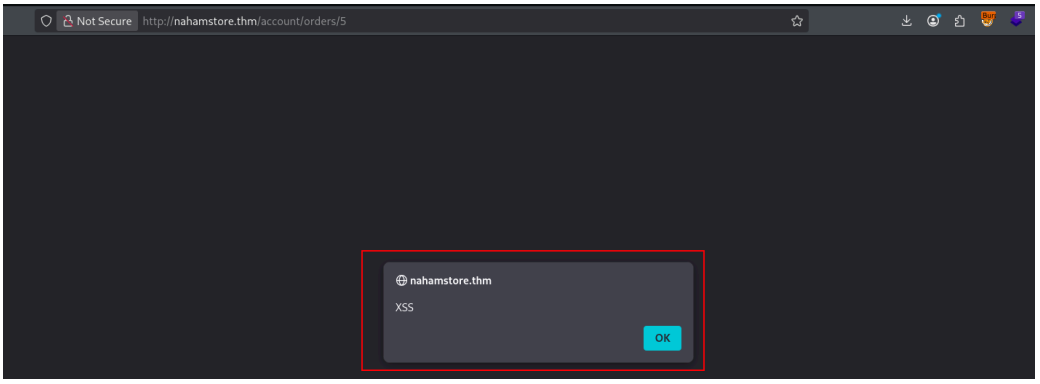
And at the end of the purchase process we will see this:



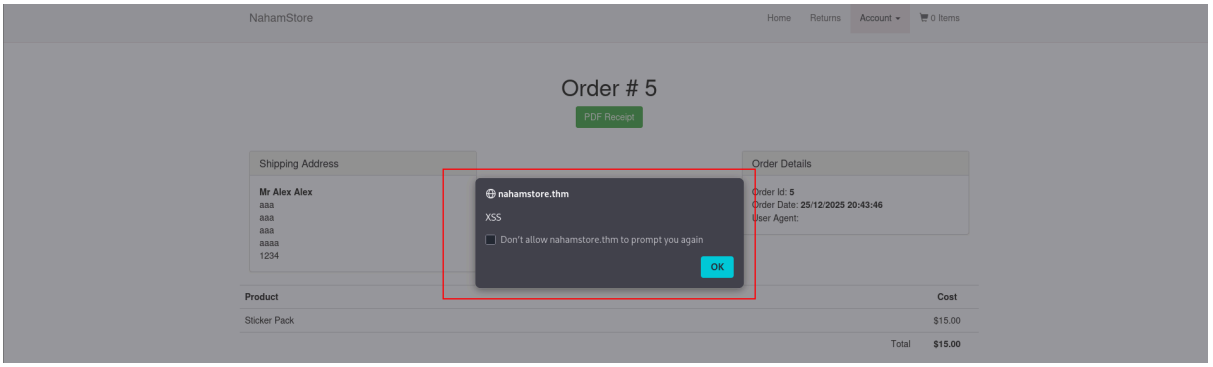
Something unusual is happening here: ***the user agent, the process handled behind the scenes of the request, is reflected on the website***. This should be a red flag. That's why I decided to repeat the process, but intercepting the request with Burp Suite and I modified the request in the User Agent content and placed a simple XSS payload.

```
POST /basket HTTP/1.1
Host: nahamstore.thm
User-Agent: <script>alert('XSS')</script>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 37
Origin: http://nahamstore.thm
Connection: keep-alive
Referer: http://nahamstore.thm/basket
Cookie: session=303711f8b643a32f5a8fdb8c966f6199; token=8af7c6341ae4a49a5db3e20584b84940
Upgrade-Insecure-Requests: 1
Priority: u=0, i

address_id=7&card_no=1234123412341234
```

And this was the result:



We have Found the last XSS! But the interesting thing here is that this is a ***Stored XSS***, which means that the payload will be activated every time someone accesses the section where this purchase order is located.



Conclusion

And with this we conclude the long XSS section for the NahamStore machine. In the next section we will continue with ***Open Redirect.***