

cross-site-scripting/reflected (2)

Web Security Academy » Cross-site scripting » Contexts » Lab

Lab: Reflected XSS into HTML context with most tags and attributes blocked

LAB Solved

This lab contains a **reflected cross-site scripting** vulnerability in the search functionality but uses a web application firewall (WAF) to protect against common **XSS** vectors.

To solve the lab, perform a cross-site scripting attack that bypasses the WAF and alerts `document.cookie`.

Note

Your solution must not require any user interaction. Manually triggering an alert in your own browser will not solve the lab.

[Access the lab](#)

Solution

1. Inject a standard XSS payload such as: ``
2. Observe that this payload gets blocked. In the next few steps, we'll use use Burp Intruder to test which tags and attributes are being blocked.
3. With your browser proxying traffic through Burp Suite, use the search function in the lab. Send the resulting request to Burp Intruder.
4. In Burp Intruder, in the Positions tab, click "Clear \$". Replace the value of the search term with: `<>`
5. Place the cursor between the angle brackets and click "Add \$" twice, to create a payload position. The value of the search term should now look like: `<$>`

Track your progress

Learning materials: [View all](#)

0%

Vulnerability labs: [View all](#)

28%

Level progress:

25 of 47 Apprentice 27 of 123 Practitioner 1 of 27 Expert

Your level: **NEWBIE**

Solve 22 more labs to become an apprentice.

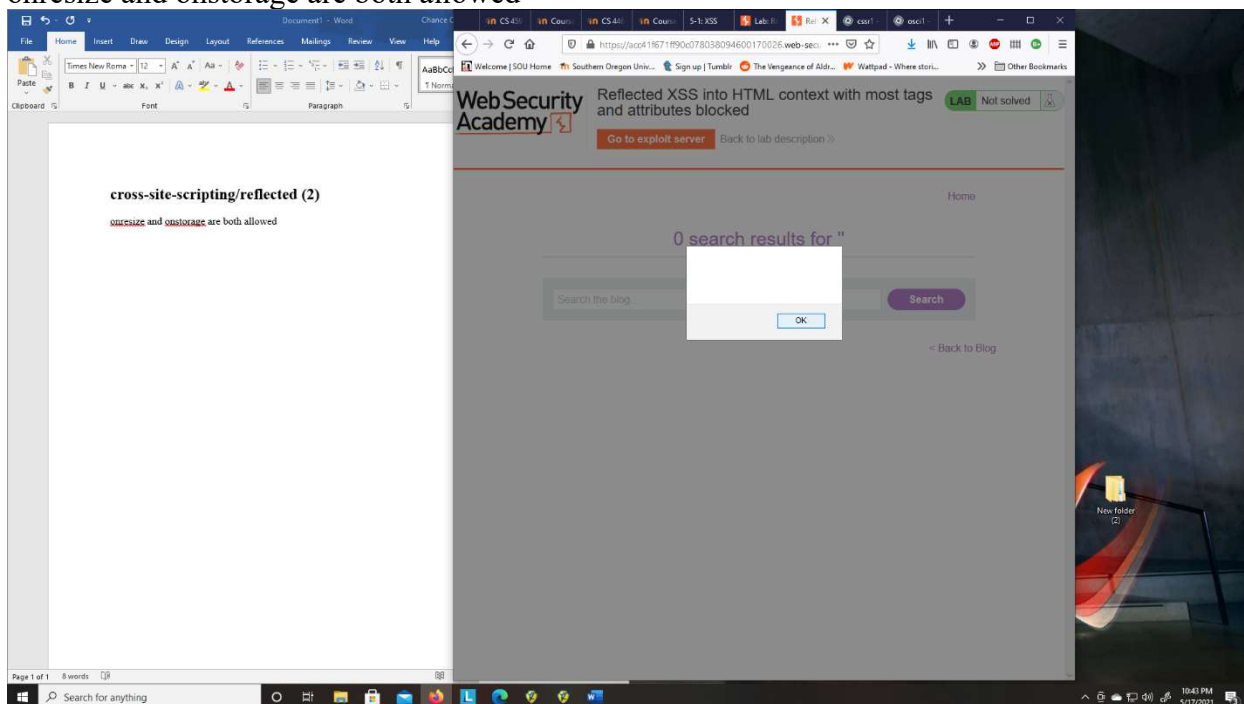
See where you rank on our Hall of Fame >>

In this topic

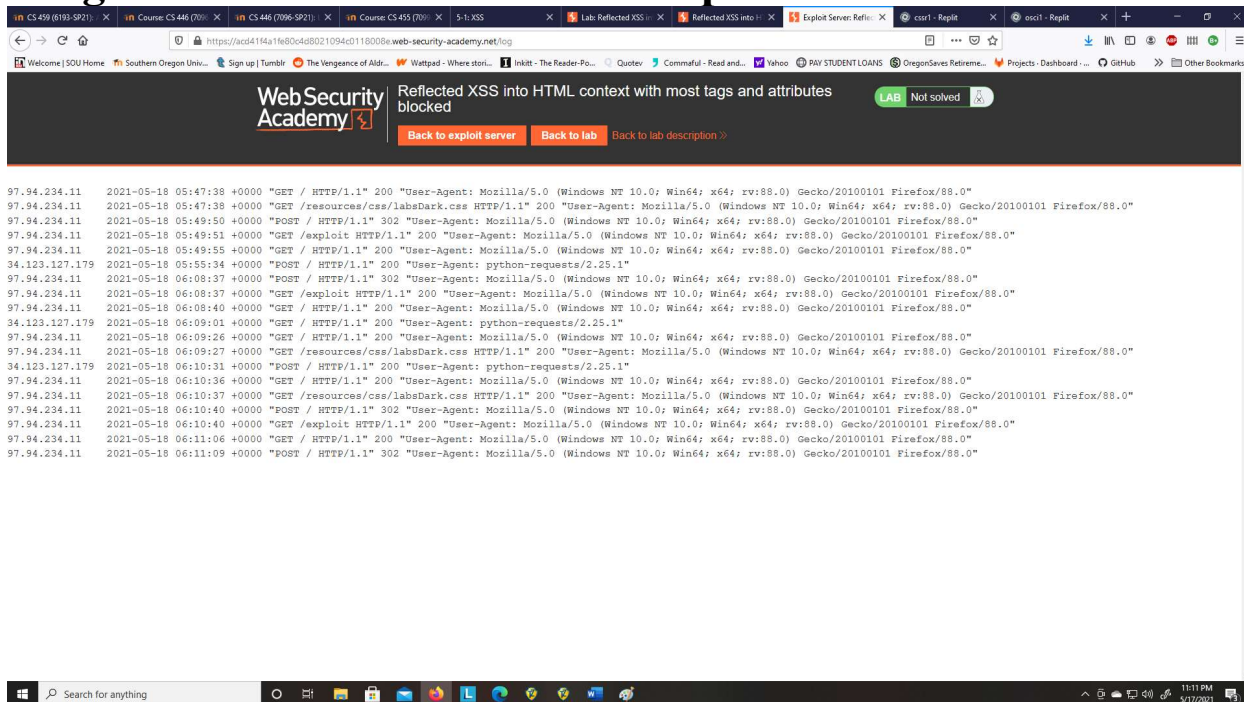
- Cross-site scripting >>
- Reflected XSS >>
- Stored XSS >>
- DOM-based XSS >>

cross-site-scripting/reflected (2)

onresize and onstorage are both allowed



Programmatic interaction with exploit server



Course CS 455 (7099-SP21)

Lab: Reflected XSS into HTML context with most tags and attributes blocked

Reflected XSS into HTML context with most tags and attributes blocked

onchange Event

HTML, onclick Event Attribute

css1 - Replit

← → ↻ 🏠

https://portswigger.net/web-security/cross-site-scripting/contexts/lab-html-context-with-most-tags-and-attributes-blocked

Welcome | SOU Home | Southern Oregon Univ... | Sign up | Tumblr | The Vengeance of Aldi... | Wattpad - Where story... | Inkitt - The Reader-Po... | Quotv | Commaful - Read and... | Yahoo | PAY STUDENT LOANS | OregonSaves Retirement... | Projects - Dashboard ... | GitHub | Other Bookmarks

PortSwigger

Log out MY ACCOUNT

Products | Solutions | Research | Academy | Daily Swig | Support |

Web Security Academy » Cross-site scripting » Contexts » Lab

Lab: Reflected XSS into HTML context with most tags and attributes blocked

🐦 🗨️ 📧 📧 📧 📧

FRAGMENTOR

LAB Solved

This lab contains a **reflected cross-site scripting** vulnerability in the search functionality but uses a web application firewall (WAF) to protect against common **XSS** vectors.

To solve the lab, perform a cross-site scripting attack that bypasses the WAF and alerts document.cookie.

Note

Your solution must not require any user interaction. Manually triggering an alert in your own browser will not solve the lab.

Access the lab

Solution

Community solutions

Track your progress

Learning materials: View all

0%

Vulnerability labs: View all

26%

Level progress:

25 of 47

27 of 123

1 of 27

Apprentice

Practitioner

Expert

Your level:

NEWBIE

Solve 22 more labs to become an apprentice.

See where you rank on our Hall of Fame >>

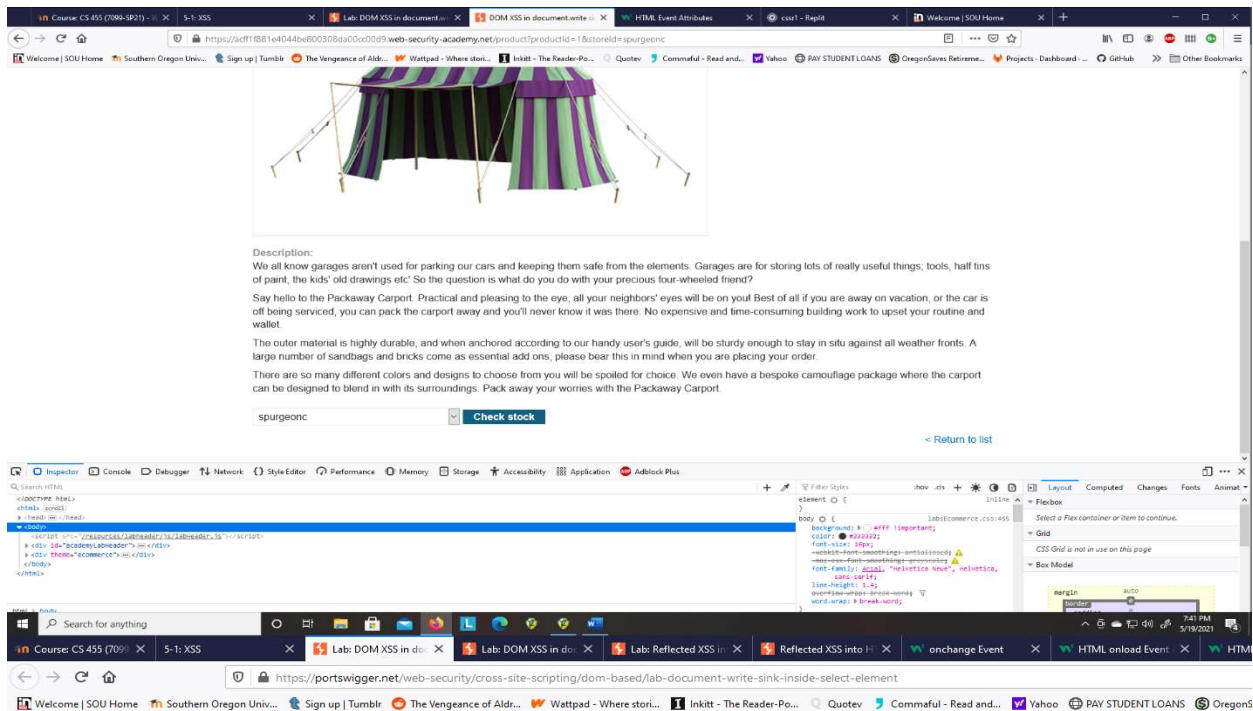
In this topic

The screenshot displays the PortSwigger Web Security Academy website. The main heading is "Lab: DOM XSS in document.write sink using source.location.search". Below the heading, it indicates the lab is "Solved". The description states: "This lab contains a DOM-based cross-site scripting vulnerability in the search query tracking functionality. It uses the JavaScript document.write function, which writes data out to the page. The document.write function is called with data from location.search, which you can control using the website URL. To solve this lab, perform a cross-site scripting attack that calls the alert function." A green button labeled "Access the lab" is present. Below the lab description, there are sections for "Solution" and "Community solutions". On the right side, a "Track your progress" sidebar shows: "Learning materials: 0%", "Vulnerability labs: 26%", and "Level progress" with three circular progress indicators for "Apprentice" (25 of 47), "Practitioner" (27 of 123), and "Expert" (1 of 27). The "Your level" section shows a "NEWBIE" badge and a message: "Solve 22 more labs to become an apprentice." The bottom of the page shows a Windows taskbar with the search bar and various application icons.

cross-site-scripting/dom-based (2)

It gets picked from a list using a function called URLSearchParams. One can never assume anything from the user.

For the first, it states that if the store variable is already set before the page is loaded, then to print it. The second is within a for loop that selects location from the list, and says that if the location in that loop matches the current store value, then to print it.



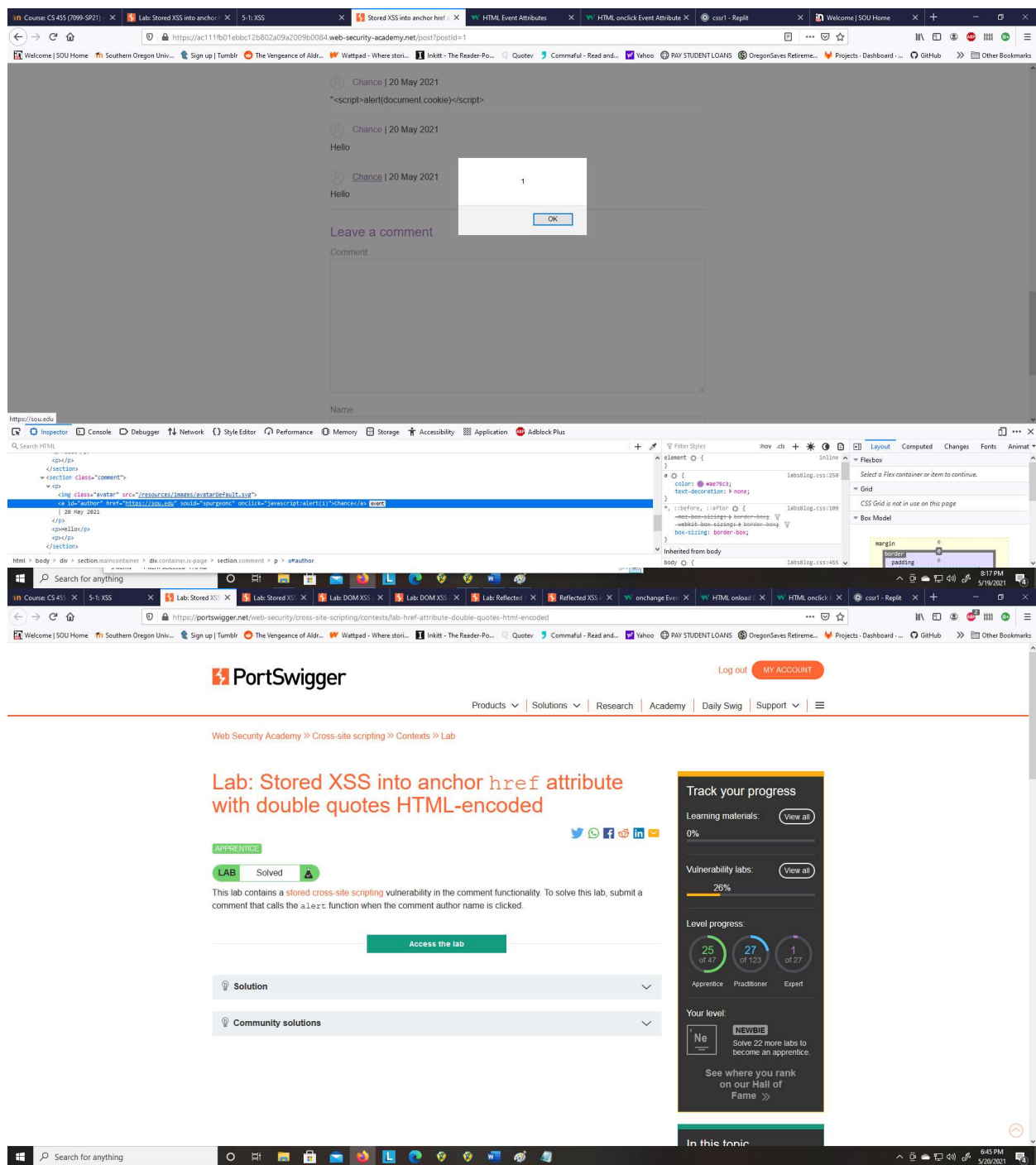
cross-site-scripting/stored (1)

The screenshot shows the PortSwigger Web Security Academy interface. The main heading is "Lab: Stored XSS into HTML context with nothing encoded". Below it, there's a "LAB Solved" badge. The description states: "This lab contains a **stored cross-site scripting** vulnerability in the comment functionality. To solve this lab, submit a comment that calls the `alert()` function when the blog post is viewed." A green button labeled "Access the lab" is present. On the right, a "Track your progress" sidebar shows learning materials at 0%, vulnerability labs at 26%, and level progress for Apprentice (25/47), Practitioner (27/123), and Expert (1/27). The user's level is "NEWBIE".

cross-site-scripting/stored (2)

The screenshot shows a web application with a blog post and a comments section. The blog post text is: "The moral of the story, if there is one, is about balance. What some people like to call, 'white lies'. The gentle untruths that spare people's feelings, and make them happy, or happier than they were when you first ask them that potentially explosive question, 'Do you like my new boots?'". The comments section shows two comments: one by Mike Plesure dated 03 May 2021, and another by Chance dated 20 May 2021. The "Leave a comment" form is visible at the bottom. The DevTools console is open, showing the HTML structure of the comments section, with the following code highlighted:

```
<div class="comment">
  
  <div class="comment-body">
    <div class="comment-text">
      Hello
    </div>
  </div>
</div>
```

11. Upload exploit and test

The screenshot shows a web browser with a JSON response in the console. The JSON object is:

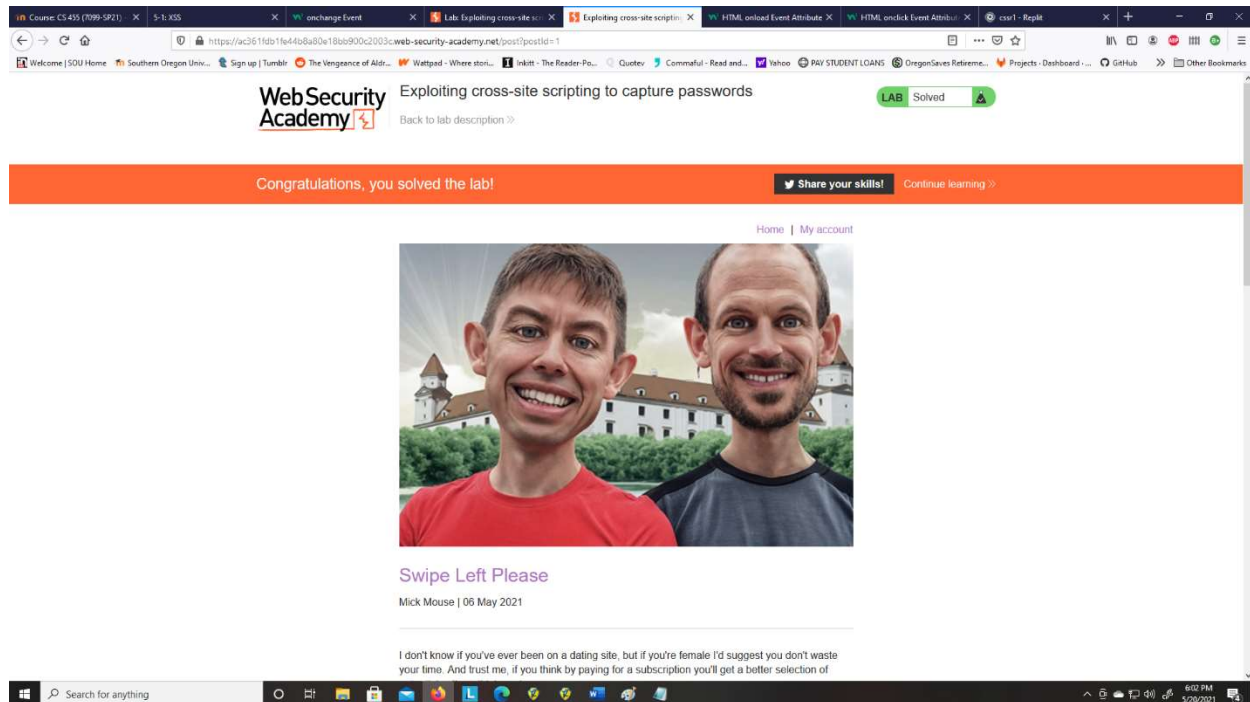
```
{
  "name": "Chance",
  "comment": "",
  "email": "spurgone@osu.edu",
  "website": "https://www.burpcollaborator.net/"
}
```

Below the JSON, the Network tab of the browser's developer tools is open, showing a POST request to `aca519021e7485678061d5a000e0001e.web-security-academy.net/post/comment`. The request body is a JSON object with the same data as the response.

Solve the level

The screenshot shows the Web Security Academy lab page for the level "Exploiting cross-site scripting to steal cookies". The lab is marked as "Solved". The page displays a congratulatory message: "Congratulations, you solved the lab!". Below this, there is a section titled "WE LIKE TO BLOG" with a photo of a man and a woman sitting on a couch. The bottom of the screenshot shows the browser's developer tools with the Network tab open, displaying a list of requests including `labHeader.js?2` and `favicon.ico`.

cross-site-scripting/exploiting (2)



Reflections

None of the input fields for submitting a comment are sanitized, and if a user knows this, they can input any code they please, even by typing said code in the comment itself. While theoretically fun, it is a large security risk if there's any sensitive data stored on the site or in the cookies, or if users can be tricked into sending it.

The solution is to keep data and code as separate as possible, and sanitize input in as many ways as possible. One should encode and/or remove characters required for code injection, as well as for code itself. One may even consider loading strings into a buffer and/or injecting zero-width spaces to break the text into chunks too small for code.