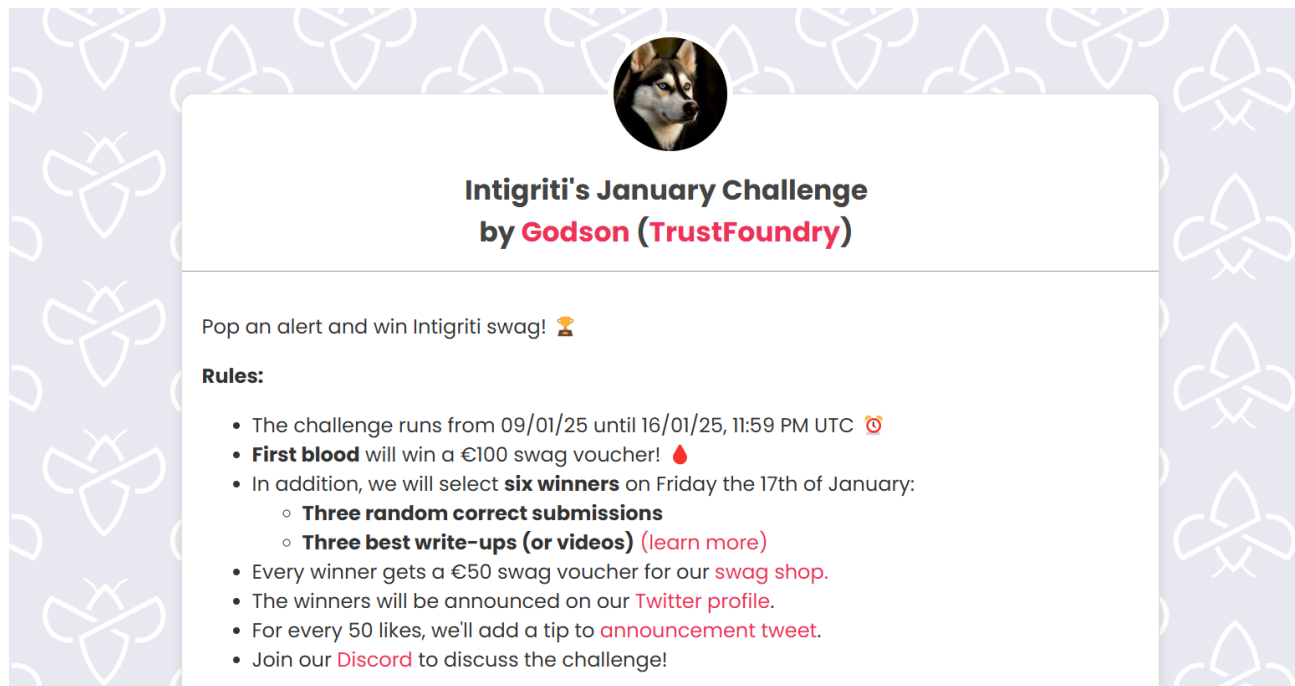


Intigrity January 2025 Challenge: XSS Challenge 0125 by 0xGodson_

In January ethical hacking platform Intigrity (<https://www.intigrity.com/>) launched a new Cross Site Scripting challenge. The challenge itself was created by community member 0xGodson_.

The graphic features a light blue background with a repeating pattern of white, stylized butterfly-like icons. At the top center is a circular profile picture of a husky dog. Below the profile picture, the text reads "Intigrity's January Challenge by Godson (TrustFoundry)". A horizontal line separates this header from the main content. The main content starts with the text "Pop an alert and win Intigrity swag! 🏆". Below this is a section titled "Rules:" followed by a bulleted list of challenge details.

Pop an alert and win Intigrity swag! 🏆

Rules:

- The challenge runs from 09/01/25 until 16/01/25, 11:59 PM UTC 🕒
- **First blood** will win a €100 swag voucher! 🩸
- In addition, we will select **six winners** on Friday the 17th of January:
 - **Three random correct submissions**
 - **Three best write-ups (or videos)** ([learn more](#))
- Every winner gets a €50 swag voucher for our [swag shop](#).
- The winners will be announced on our [Twitter profile](#).
- For every 50 likes, we'll add a tip to [announcement tweet](#).
- Join our [Discord](#) to discuss the challenge!

Rules of the challenge

- Should work on the latest version of Chrome **and** FireFox.
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks.
- Should require no user interaction.

Challenge

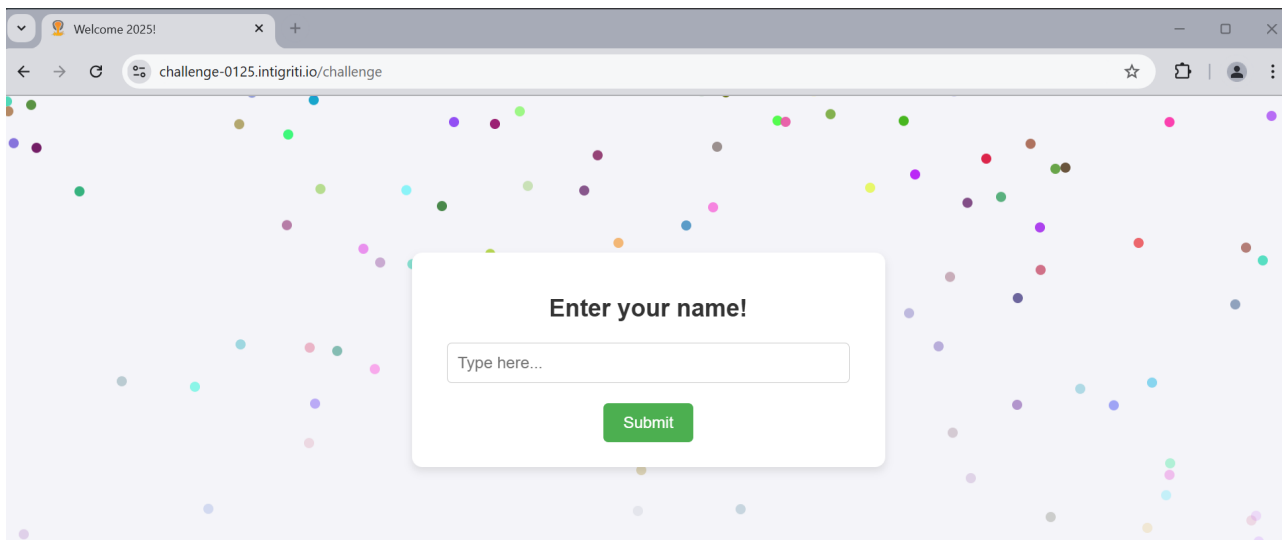
To simplify a victim needs to visit our crafted web URL for the challenge page and arbitrary JavaScript should be executed to launch a Cross Site Scripting (XSS) attack against our victim.

The XSS (Cross Site Scripting) attack

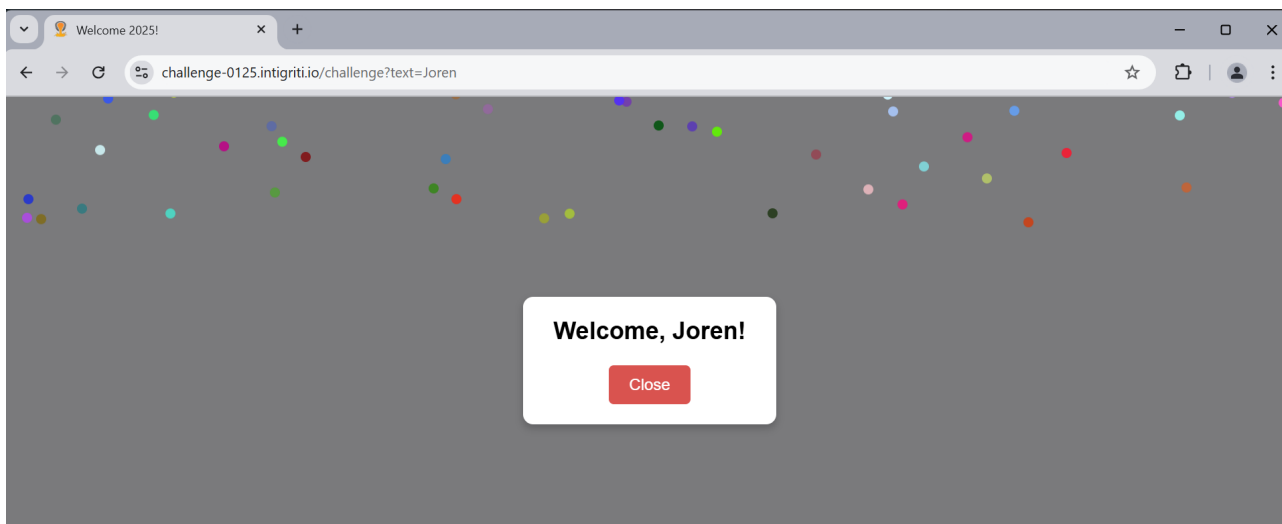
Step 1: Recon

It is always important to carefully check the target you are trying to attack and look around for possible weak spots. Use the web application and check the JavaScript source code. The better you know how an application works the more chance you will have to find vulnerabilities.

The challenge start at this URL: <https://challenge-0125.intigriti.io/> but shows payloads can be tested here: <https://challenge-0125.intigriti.io/challenge>



A simple web page where we can enter our name. We approach the page as a normal end user of the application and enter our name to see how the application reacts.



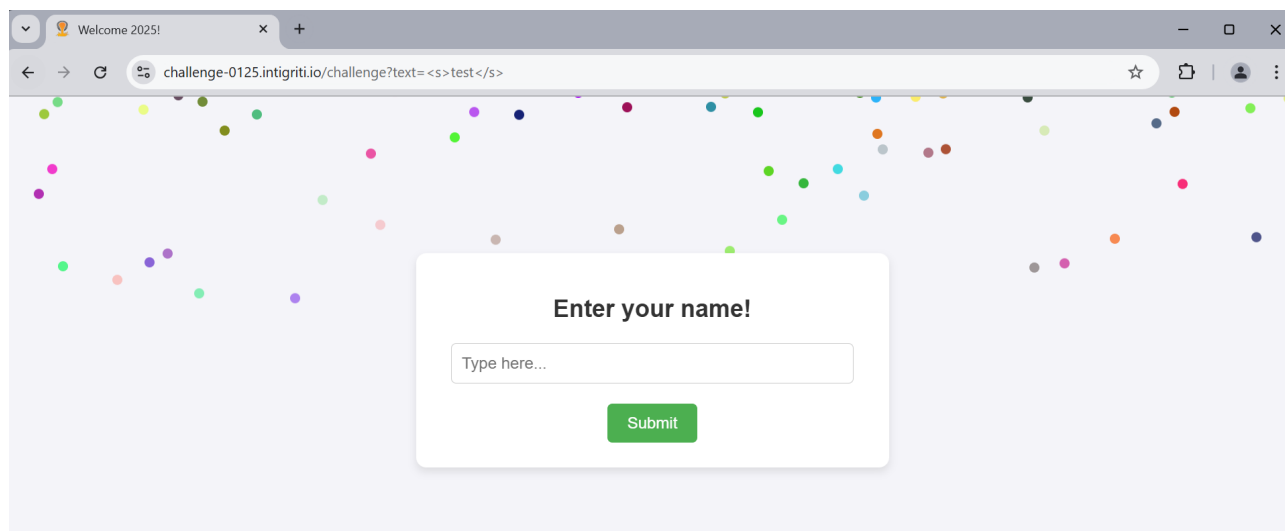
A popup appears that welcomes us with our first name. This first action reveals the application uses an URL parameter “text” for our input: <https://challenge-0125.intigriti.io/challenge?text=Joren>

When testing for XSS attacks I always start simple with HTML injection. Once I can achieve HTML injection I will try to build on that further to achieve XSS.

Lets try to achieve HTML injection with following payload: `<s>test</s>`. If we see `test` (with strike-through) reflected in the application we already achieve a first step.

The URL with payload will look like following: <https://challenge-0125.intigriti.io/challenge?text=<s>test</s>>

Testing this results in the application redirecting back to the home page to enter your name. This shows the application has some kind of filtering or WAF (Web Application Firewall) blocking our HTML injection payloads.



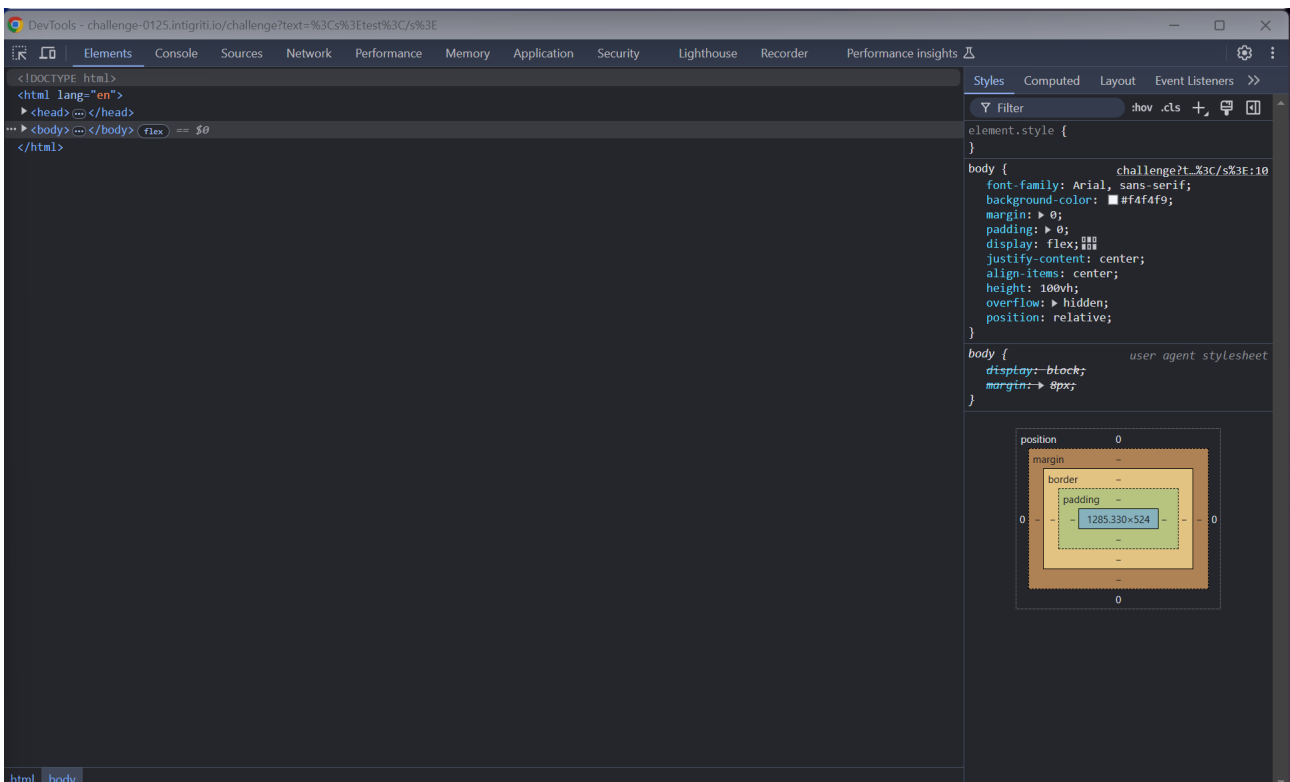
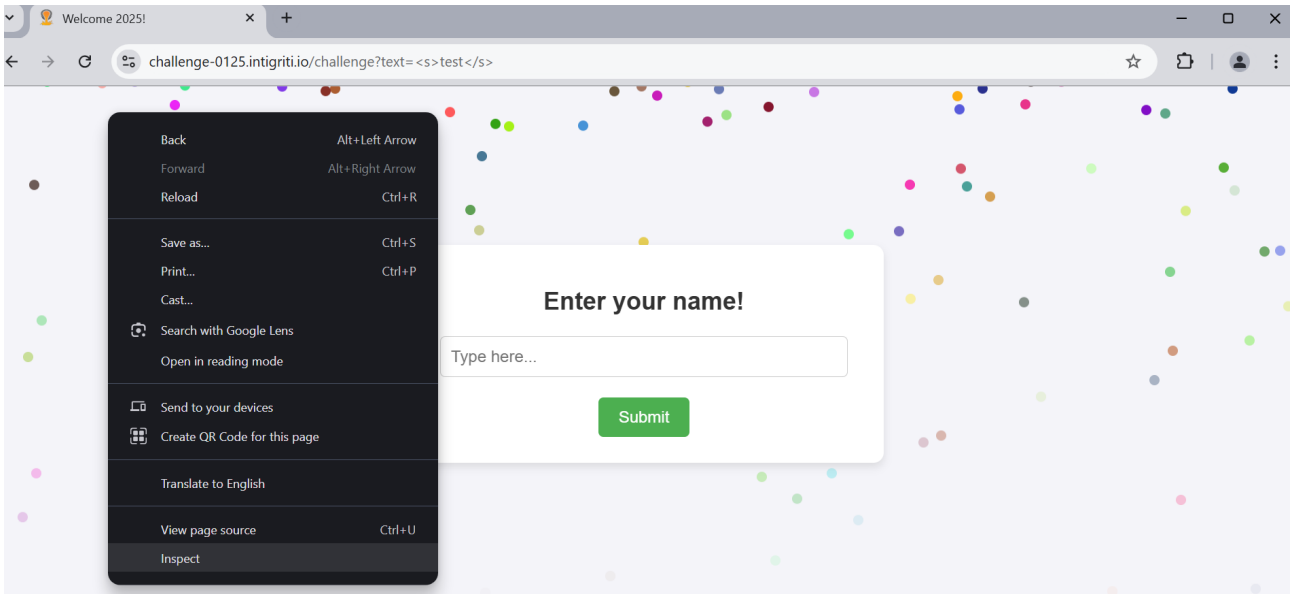
This means it would make no sense to start throwing XSS payloads blindly at the application as we will need to bypass the filter. At this point after our initial recon the next steps look like following.

- 1) Determine which filtering or WAF (Web Application Firewall) is used to block malicious payloads.
- 2) Achieve HTML injection.
- 3) Build further on our HTML injection and achieve XSS.

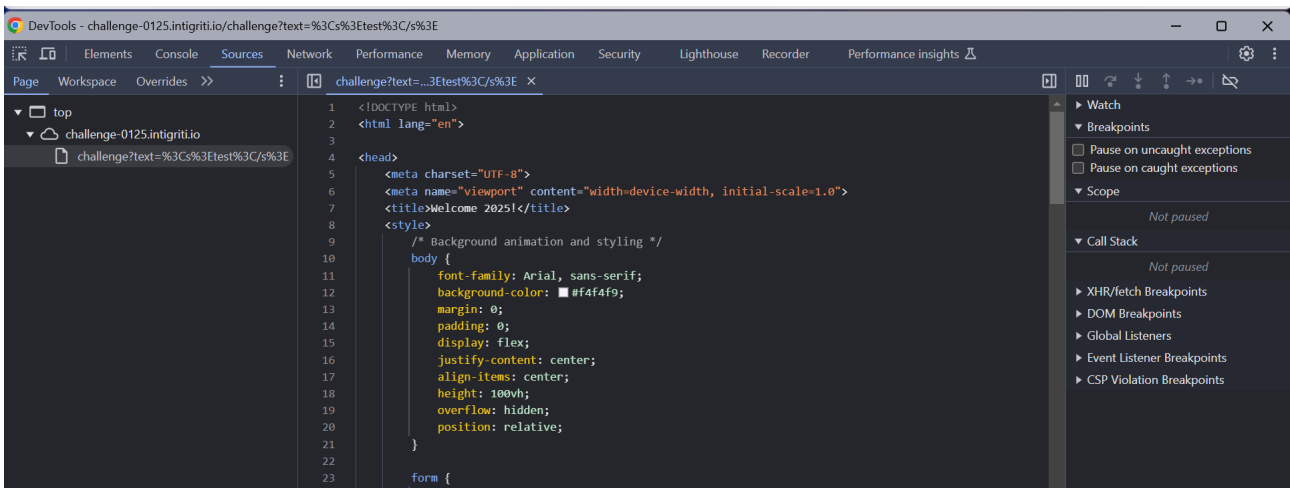
Step 2: Deep dive into the source code

Another important step to achieve the filtering bypass is to inspect the application source code. This will help in our recon to find weak spots in the applications defense.

Right click on the browser window and click inspect. This will open the DevTools.

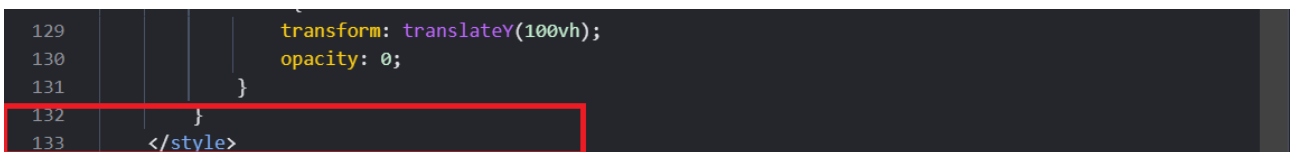
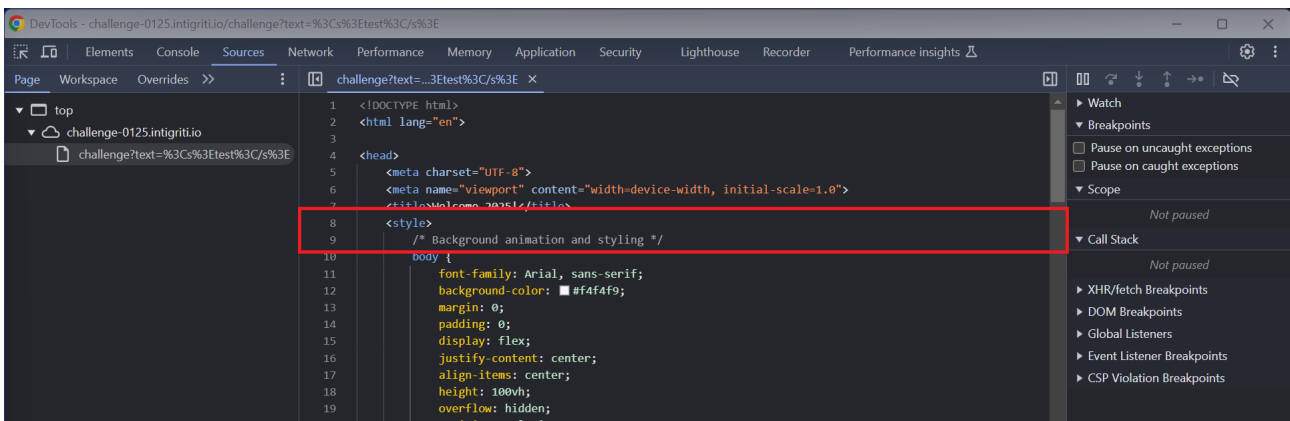


Once DevTools is opened move to the “Sources” tab and double click the challenge page to view the source code.



In this case all the web application code is embedded in one page. So CSS, HTML and JavaScript are put into the same page. Often the CSS and JavaScript will be split and organized into different files.

The top of the page contains the CSS styling and animation. This is not interesting for the challenge and can be ignored. So we can ignore everything between the `<style>body{...</style>` tags.



After the CSS styling we get the HTML body part. This contains the HTML code for the textForm where we enter our first name. As this part is static HTML code it can be ignored as not interesting for the challenge.

```
133 </style>
134 <link rel="icon" href="https://trustfoundry.net/wp-content/uploads/2022/07/cropped-fav-32x32.png" si
135 </head>
136
137 <body>
138 <form id="textForm" onsubmit="redirectToText(event)">
139   <h1>Enter your name!</h1>
140   <label for="inputBox"></label>
141   <input type="text" id="inputBox" name="inputBox" placeholder="Type here...">
142   <button type="submit">Submit</button>
143 </form>
144
145 <!-- Modal -->
146 <div id="modal" class="modal">
147   <div class="modal-content">
148     <h2 id="modalText"></h2>
149     <button onclick="closeModal()">Close</button>
150   </div>
151 </div>
152
153 <!-- Falling Particles -->
154 <div id="particles">
155   <!-- Falling particles will be generated here -->
156 </div>
```

Now the interesting part starts. Between `<script>...</script>` tags the JavaScript for this web application is defined. This is something we should check carefully.

```
157 <script>
158   function XSS() {
159     return decodeURIComponent(window.location.search).includes('<') || decodeURIComponent(window
160   }
161   function getParameterByName(name) {
162     var url = window.location.href;
163     name = name.replace(/[\/\?]/g, "\\$&");
164
225   }
226 }
227
228 // Generate particles when the page loads
229 window.onload = function () {
230   generateFallingParticles();
231   checkQueryParam();
232 };
233 </script>
```

Here a copy of the full JavaScript code between the script tags:

```
<script>
function XSS() {
  return decodeURIComponent(window.location.search).includes('<') || decodeURIComponent(window.location.search).includes('>') ||
  decodeURIComponent(window.location.hash).includes('<') || decodeURIComponent(window.location.hash).includes('>')
}
function getParameterByName(name) {
  var url = window.location.href;
  name = name.replace(/[[]]/g, "\\$&");
  var regex = new RegExp("[?&]" + name + "([^\&#]*|&|#|$)");
  results = regex.exec(url);
  if (!results) return null;
  if (!results[2]) return "";
  return decodeURIComponent(results[2].replace(/\+/g, " "));
}

// Function to redirect on form submit
function redirectToText(event) {
  event.preventDefault();
  const inputBox = document.getElementById('inputBox');
  const text = encodeURIComponent(inputBox.value);
  window.location.href = `challenge?text=${text}`;
}

// Function to display modal if 'text' query param exists
function checkQueryParam() {
  const text = getParameterByName('text');
  if (text && XSS() === false) {
    const modal = document.getElementById('modal');
    const modalText = document.getElementById('modalText');
    modalText.innerHTML = `Welcome, ${text}!`;
    textForm.remove()
    modal.style.display = 'flex';
  }
}

// Function to close the modal
function closeModal() {
  location.replace('/challenge')
}

// Function to generate random color
function getRandomColor() {
  const letters = '0123456789ABCDEF';
  let color = '#';
  for (let i = 0; i < 6; i++) {
    color += letters[Math.floor(Math.random() * 16)];
  }
  return color;
}

// Function to generate falling particles
function generateFallingParticles() {
  const particlesContainer = document.getElementById("particles");

  // Generate 100 particles with different animations, positions, and colors
  for (let i = 0; i < 100; i++) {
    let particle = document.createElement("div");
    particle.classList.add("falling-particle");

    // Randomize the particle's left position
    particle.style.left = Math.random() * 100 + "vw"; // Left position from 0 to 100% of viewport width

    // Randomize the particle's color
    particle.style.backgroundColor = getRandomColor();

    // Randomize animation delays
    particle.style.animationDelay = Math.random() * 5 + "s"; // Random delay for staggered fall
    particlesContainer.appendChild(particle);
  }
}

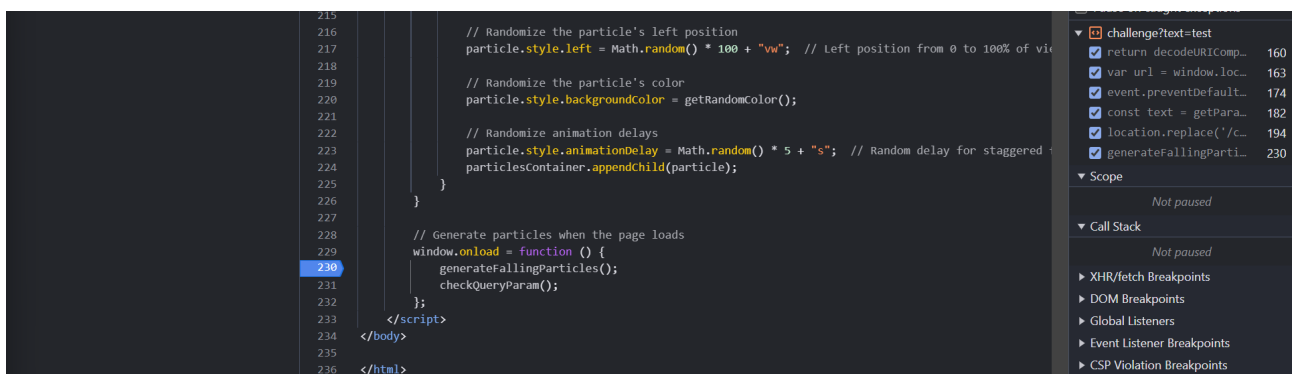
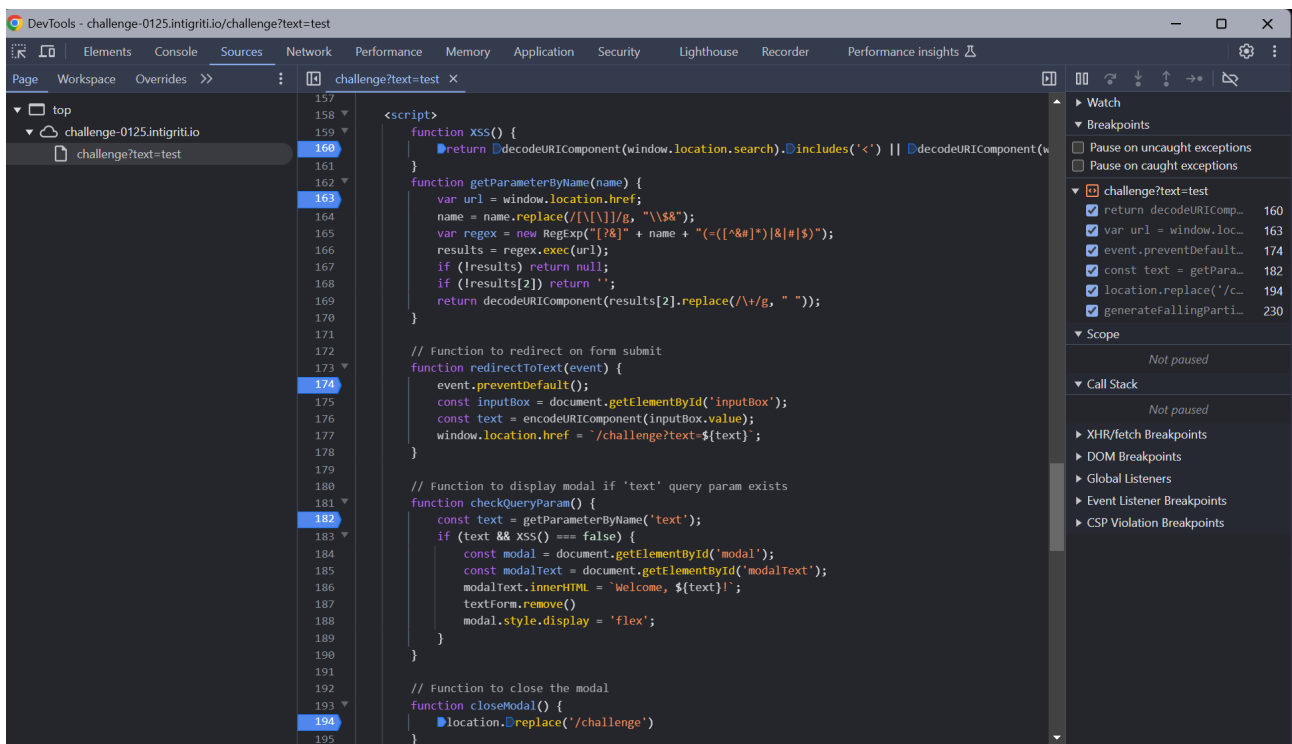
// Generate particles when the page loads
window.onload = function () {
  generateFallingParticles();
  checkQueryParam();
};
</script>
```

This can be pretty overwhelming at first sight. Splitting it in smaller pieces can help. Reading the JavaScript code we can see it is split into different functions by the developer who created the application. I approached it by checking each function one by one.

From the source code comments it is clear we can skip following functions as they are only used to generate and color the falling snow particles: “*function getRandomColor(), function generateFallingParticles()*”

Now I want to figure out what the JavaScript source code is doing when I enter a non malicious payload into the URL “text” parameter. We load following URL: <https://challenge-0125.intigriti.io/challenge?text=test>

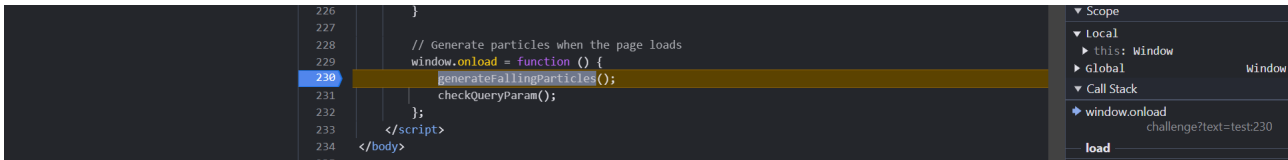
With the browser DevTools open we can set breakpoints where we want the JavaScript code execution to stop so we can inspect it better. I first placed a breakpoint after each interesting function to see which route the code takes. In the “Sources” tab open the source code and place breakpoint by clicking line number where you want the JavaScript code execution to stop.



Once the breakpoints are set reload the web page with the URL we used previously:

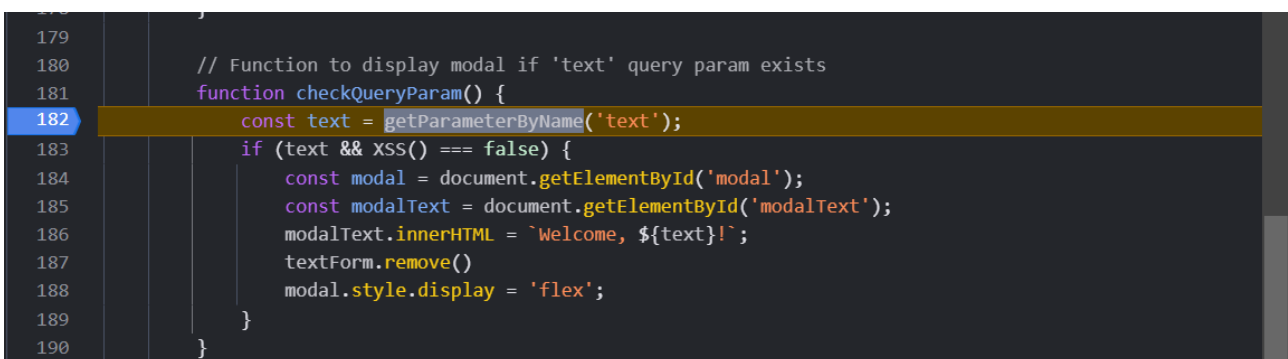
<https://challenge-0125.intigriti.io/challenge?text=test>

We first hit the “window.onload” function which is normal as we are reloading the page. The page onload function triggers the for us non interesting “generateFallingParticles()” function and also the “checkQueryParam()” function which will be our next breakpoint.



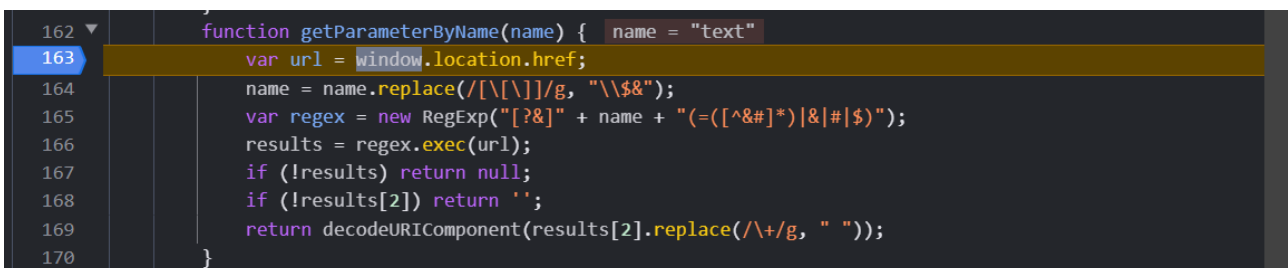
```
226     }
227
228     // Generate particles when the page loads
229     window.onload = function () {
230         generateFallingParticles();
231         checkQueryParam();
232     };
233 </script>
234 </body>
```

Press F8 to advance towards the next breakpoint. As we have seen in the previous breakpoint this will be the “checkQueryParam()” function we hit now.



```
179
180     // Function to display modal if 'text' query param exists
181     function checkQueryParam() {
182         const text = getParameterByName('text');
183         if (text && XSS() === false) {
184             const modal = document.getElementById('modal');
185             const modalText = document.getElementById('modalText');
186             modalText.innerHTML = `Welcome, ${text}!`;
187             textForm.remove();
188             modal.style.display = 'flex';
189         }
190     }
```

This “checkQueryParam()” function immediately invokes another function “getParameterByName”



```
162     function getParameterByName(name) { name = "text"
163         var url = window.location.href;
164         name = name.replace(/[\\[\]]/g, "\\$&");
165         var regex = new RegExp("[?&]" + name + "=(^[&#]*)|&#|)$");
166         results = regex.exec(url);
167         if (!results) return null;
168         if (!results[2]) return '';
169         return decodeURIComponent(results[2].replace(/\+/g, " "));
170     }
```

Here there is no code invoking a new function so it goes through all the steps of this “getParameterByName” function and at the end it returns something back towards the previous “checkQueryParam()” function.

We are returned to the “checkQueryParam()” function with the output of the “getParameterByName” function. If you hover over the “text” variable you will now see it contains our URL parameter input “test”

```
179
180 // Function to display modal if 'text' query param exists
181 function "test" checkQueryParam() {
182     const text = getParameterByName('text'); text = "test"
183     if (text && XSS() === false) {
184         const modal = document.getElementById('modal');
185         const modalText = document.getElementById('modalText');
186         modalText.innerHTML = `Welcome, ${text}!`;
187         textForm.remove();
188         modal.style.display = 'flex';
189     }
190 }
191
```

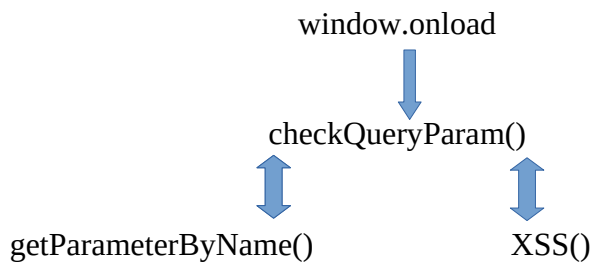
Back into the “checkQueryParam()” we see an if statement with a logical AND operator. Our returned “text” goes into the logical AND with the XSS() function. This step invokes the next function XSS().

```
158 </script>
159 function XSS() {
160     return decodeURIComponent(window.location.search).includes('<') || decodeURIComponent(w
161 }
```

The output coming from the XSS() function is returned back towards the “checkQueryParam()” function finishing the if statement there with the logical AND operator. This then loads the full web page meaning we went through all the JavaScript code.

After this first high level loop over the JavaScript code we can conclude following.

- The main function is “checkQueryParam()” which invokes the other functions who return the necessary data to load the web page. We have following JavaScript functions structure:



Step 3: The main JavaScript function: “checkQueryParam()”

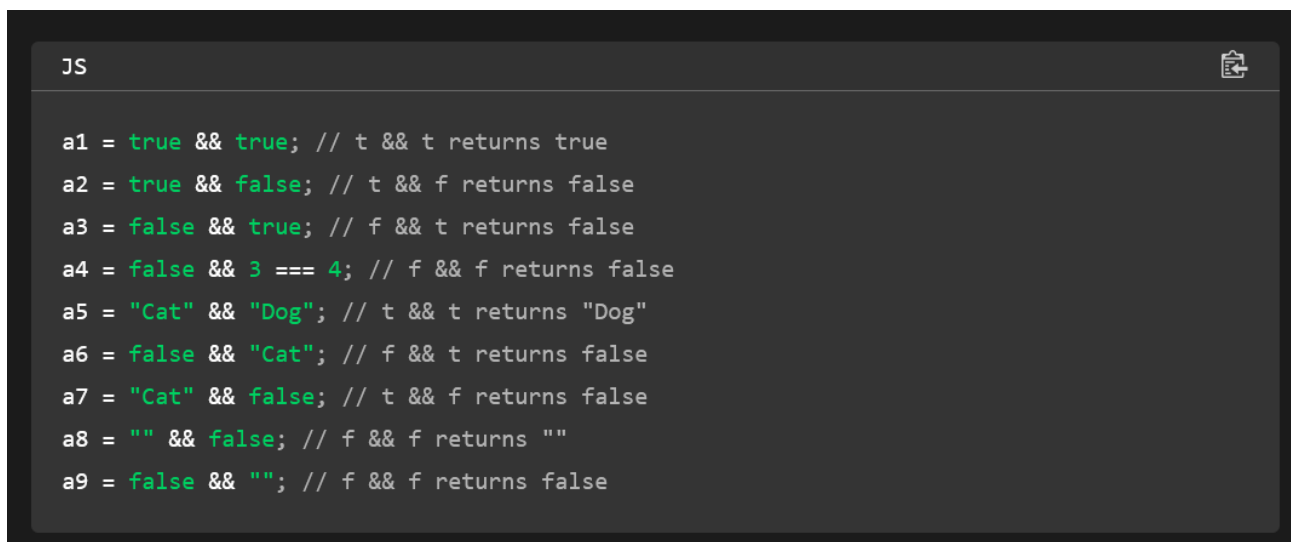
```
// Function to display modal if 'text' query param exists
function checkQueryParam() {
  const text = getParameterByName('text');
  if (text && XSS() === false) {
    const modal = document.getElementById('modal');
    const modalText = document.getElementById('modalText');
    modalText.innerHTML = `Welcome, ${text}!`;
    textForm.remove();
    modal.style.display = 'flex';
  }
}
```

The main JavaScript function as we noticed earlier first initiates the “getParameterByName” function so it gets a value returned for the “text” variable.

Then we enter an if loop with a logical AND operator which expects an outcome “false” to proceed inside with the JavaScript code inside the loop.

The logical AND is explained here:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND

A screenshot of a JavaScript console window with a dark background. The title bar says "JS" and there is a clipboard icon on the right. The console contains the following code:

```
a1 = true && true; // t && t returns true
a2 = true && false; // t && f returns false
a3 = false && true; // f && t returns false
a4 = false && 3 === 4; // f && f returns false
a5 = "Cat" && "Dog"; // t && t returns "Dog"
a6 = false && "Cat"; // f && t returns false
a7 = "Cat" && false; // t && f returns false
a8 = "" && false; // f && f returns ""
a9 = false && ""; // f && f returns false
```

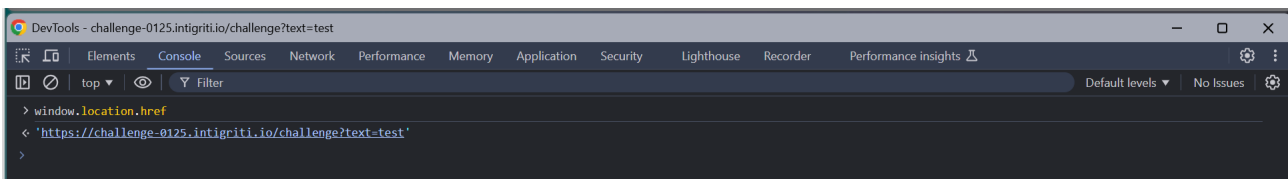
If we can get the if loop to be in the “false” condition the remaining JavaScript code will add the “text” variable into the HTML source code. This is our input name and thus the point where we would like to first inject HTML injection and later inject the XSS payload.

Step 4: The “getParameterByName” JavaScript function

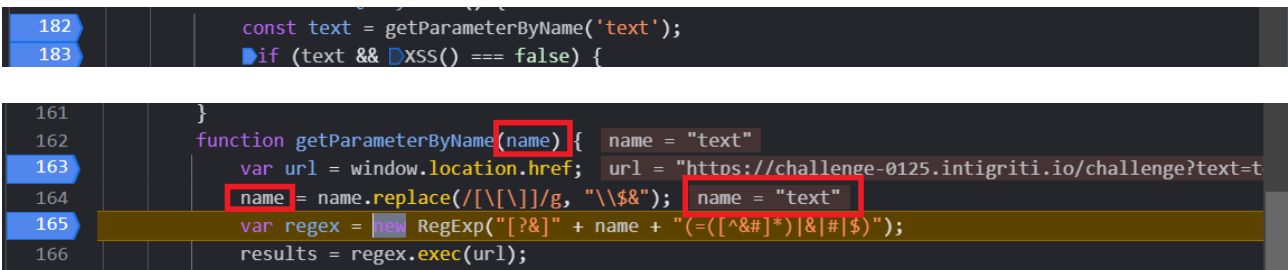
It would be tempting to immediately look into the JavaScript XSS() function and see if we can get it to return false towards our main “checkQueryParam()” function but first the “getParameterByName” function was called just before the if loop.

```
function getParameterByName(name) {
  var url = window.location.href;
  name = name.replace(/[\/]/g, "\\$&");
  var regex = new RegExp("[?&]" + name + "(=[^&#]*)|&#|$)");
  results = regex.exec(url);
  if (!results) return null;
  if (!results[2]) return "";
  return decodeURIComponent(results[2].replace(/\+/g, " "));
}
```

This function takes the “window.location.href” which if you check via the DevTools of your browser in the “console” tab results in the complete URL and puts this in the “url” variable.



The “name” variable will always be “text” as this is hard-coded send by the “checkQueryParam()” function. There is a “name.replace” done onto the “name” variable but as this is hard-coded set to “text” that does not have any impact.



The “regex” variable is then created with a regular expression: `[?&]text(=[^&#]*)|&#|$)`. This regular expression is used onto our “url” variable that contains the full URL: <https://challenge-0125.intigrity.io/challenge?text=test>

If you look into the regex it actually looks for following part inside the complete URL:

- it needs to start with ? Or &
- then contains text=
- It should not contain \$ or # multiple times or end with & or #

The regex roughly said looks in our complete URL for a query parameter like `?text=` or `&text=`

Here we should take a note that the web application developer has chosen to use the regex onto the complete URL by using “window.location.href” this is a bit weird as there are more strict JavaScript functions to get the parameters like “window.location.search”

```
166     results = regex.exec(url);
167     if (!results) return null;
168     if (!results[2]) return '';
169     return decodeURIComponent(results[2].replace(/\+/g, " "));
```

The regex output results in an array which is then checked by 2 if statement to be sure it is not empty. So if we load an URL without the parameter ?text= or &text= or we leave the “text” parameter empty this function will return “null” or nothing towards or main function and we will have our input not embedded into the web page.

Conclusion for this function is that the web developer maybe made a mistake by taking the complete URL to find the “text” parameter.

Step 5: The “XSS” JavaScript function

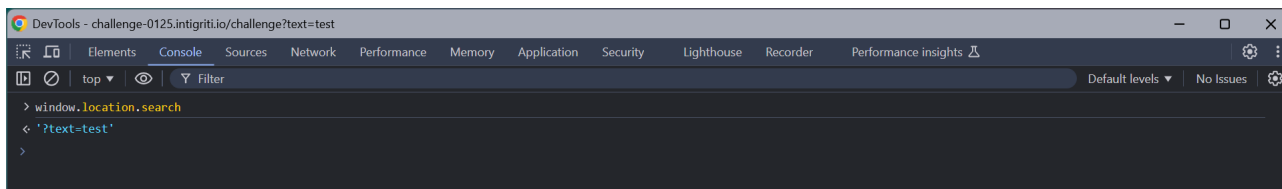
The last function is a short one:

```
function XSS() {
    return decodeURIComponent(window.location.search).includes('<') || decodeURIComponent(window.location.search).includes('>') ||
    decodeURIComponent(window.location.hash).includes('<') || decodeURIComponent(window.location.hash).includes('>')
}
```

The function first does URL decoding and then uses “window.location.search” to search in the URL for the parameters.

If we take our input URL: <https://challenge-0125.intigriti.io/challenge?text=test> then using the “window.location.search” JavaScript function results in “?text=test”

You can easily test this in the browsers DevTools.



Together with “window.location.search” it also uses “window.location.hash” to find the fragment or hash part of the URL. This can be demonstrated with following input URL: <https://challenge-0125.intigriti.io/challenge?text=test#testhash>

“window.location.hash” results in the output: “#testhash”



The XSS() JavaScript function thus looks in the URL parameters and hash part for the characters: “<” and “>”. This means that ones it finds “<” or “>” anywhere in the parameters or hash it will return true towards our main function.

Step 6: Putting the puzzle together

From our main JavaScript function “checkQueryParam()” we need the if loop “if (text && XSS() === false)” to return false so we can proceed to embedding our payload.

We saw earlier the “text” variable will be true as it cannot be empty. This text variable contains our input and thus we would like to use “<” or “>” here to attack the web application. We are faced with the XSS() function returning true if it finds a “<” or “>” in the URL parameters or hash fragment which we want to avoid as true && true will not be equal to false which we need.

If we look at the complete URL it means the parameter and hash part is useless for any attack vector:

<https://challenge-0125.intigriti.io/challenge?text=test#testhash>

But if we go back to the “getParameterByName()” JavaScript function it was a bit weird the developer has chosen to check the complete URL for a “?text=” or “&text=” part via the regex. This means we could embed “?text=” or “&text=” in another part of the URL that is not checked by the XSS() function.

There is a discrepancy between the “getParameterByName()” and “XSS()” function. The “text” parameter is allowed to be found in the complete URL while the XSS() function only checks the parameters or hash part of the URL.

Where can we hide our “?text=” or “&text=” somewhere else in the URL?

For this we need to dig a bit deeper into the web server how that behaves.

The web application runs at this URL: <https://challenge-0125.intigriti.io/challenge> which means there is a “challenge” folder on the web server.

The following screenshot shows a challenge folder being created with no content as an example.

```
joren@DESKTOP-HOT3NN4:~$ mkdir challenge
joren@DESKTOP-HOT3NN4:~$ cd challenge
joren@DESKTOP-HOT3NN4:~/challenge$ pwd
/home/joren/challenge
joren@DESKTOP-HOT3NN4:~/challenge$ ls -lah
total 8.0K
drwxr-xr-x  2 joren joren 4.0K Jan 12 21:09 .
drwxr-x---  7 joren joren 4.0K Jan 12 21:09 ..
joren@DESKTOP-HOT3NN4:~/challenge$
```

Notice that I am located in the directory /home/joren/challenge. If I then give the command ../ I will traverse downwards towards the /home/joren directory.

```
joren@DESKTOP-HOT3NN4:~/challenge$ pwd
/home/joren/challenge
joren@DESKTOP-HOT3NN4:~/challenge$ cd ../
joren@DESKTOP-HOT3NN4:~$ pwd
/home/joren
joren@DESKTOP-HOT3NN4:~$
```

On a web server we could do following by actually giving a non existing path but returning towards the existing “challenge” path with a traversal.

<https://challenge-0125.intigriti.io/challenge/nonexistingfolder/..%2F>

=> keep in mind the last / is URL encoded into %2F

This will return without any error to the challenge page and if we now check the XSS() and getParameterByName() function it becomes interesting.

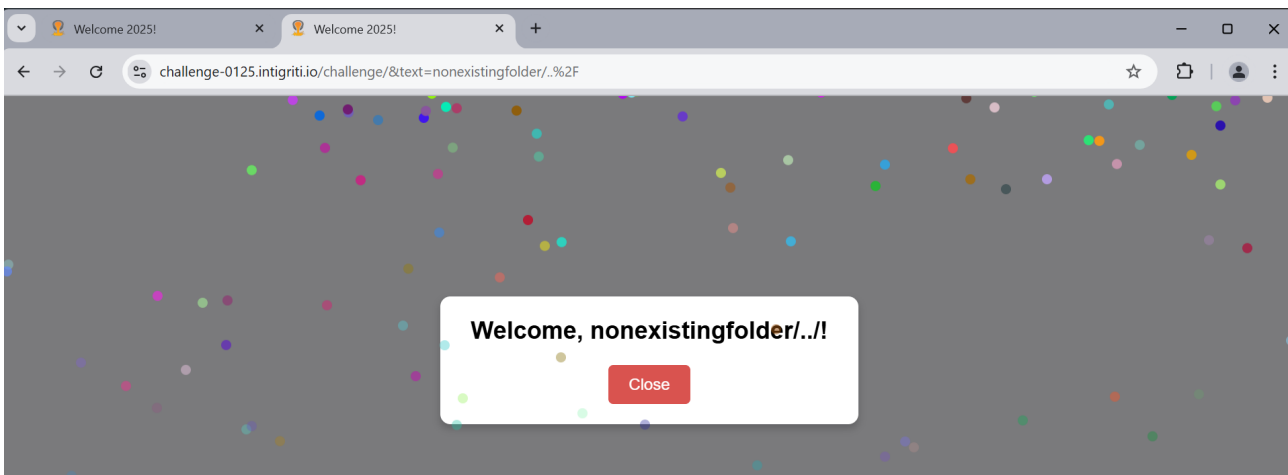
```
157
158     <script>
159         function XSS() {
160             return decodeURIComponent(window.location.search).includes('<') || decodeURIComponent(window.location.search).in
161         }
162         function getParameterByName(name) { name = "text"
163             var url = window.location.href; url = "https://challenge-0125.intigriti.io/challenge/nonexistingfolder/..%2F"
164             name = name.replace(/[\\\/]/g, "\\$&");
165             var regex = new RegExp("[&]" + name + "(=[^&#]*)|&#|$)");
166             results = regex.exec(url);
```

“window.location.search and window.location.hash” are empty as we never gave any parameters in the URL so the XSS() function will return false as it will not find any “<” or “>” in the parameters or hash which is perfect for us.

“windows.location.href” takes the complete URL including our “nonexistingfolder” to use in its regex. This means if we do something like following:

<https://challenge-0125.intigriti.io/challenge/&text=nonexistingfolder/..%2F>

We should end up with “nonexistingfolder/..” as our name displayed by the application as the regex will search for “?text= or &text=” in the complete URL and take the value it contains.



That works and we bypassed the XSS() function as it sees no parameters like ?text= and nothing in the hash part :-)

Lets build further to a HTML injection by adding the value “<s>test</s>” into our fake path (be sure to URL encode the / into %2F):

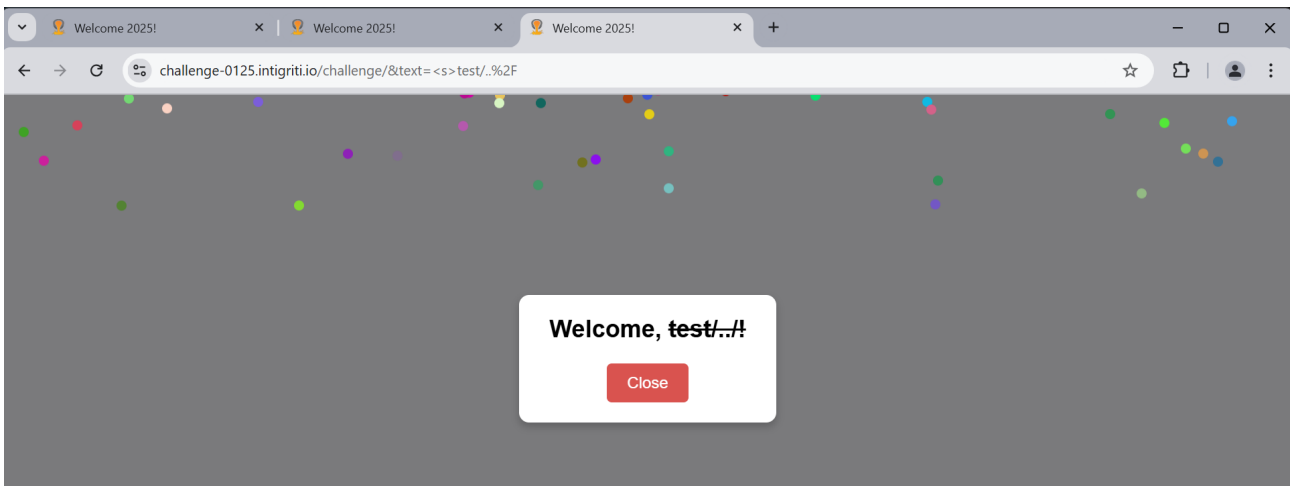
<https://challenge-0125.intigriti.io/challenge/&text=<s>test<%2Fs>/..%2F>



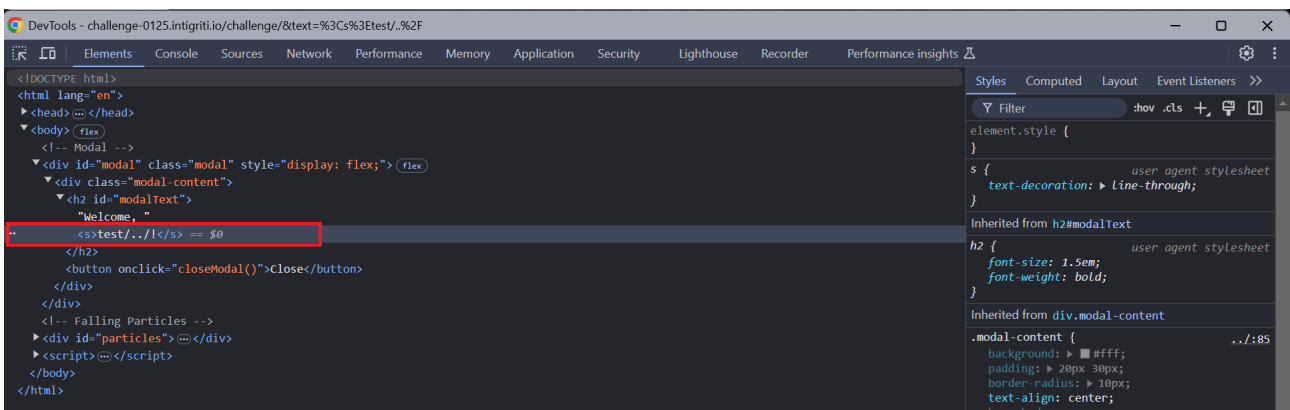
404 Not Found

The page you requested could not be found.

That does not work :(because we use </s> as the end of our payload which is seen as a new part of the URL path due to the / character. Luckily browsers are always trying to fix broken HTML code so we can input following: <https://challenge-0125.intigriti.io/challenge/&text=<s>test/..%2F>



Exactly what we want. We achieved HTML injection. The browser fixed our <s>test to working HTML in the source code and converted it to <s>test</s>.

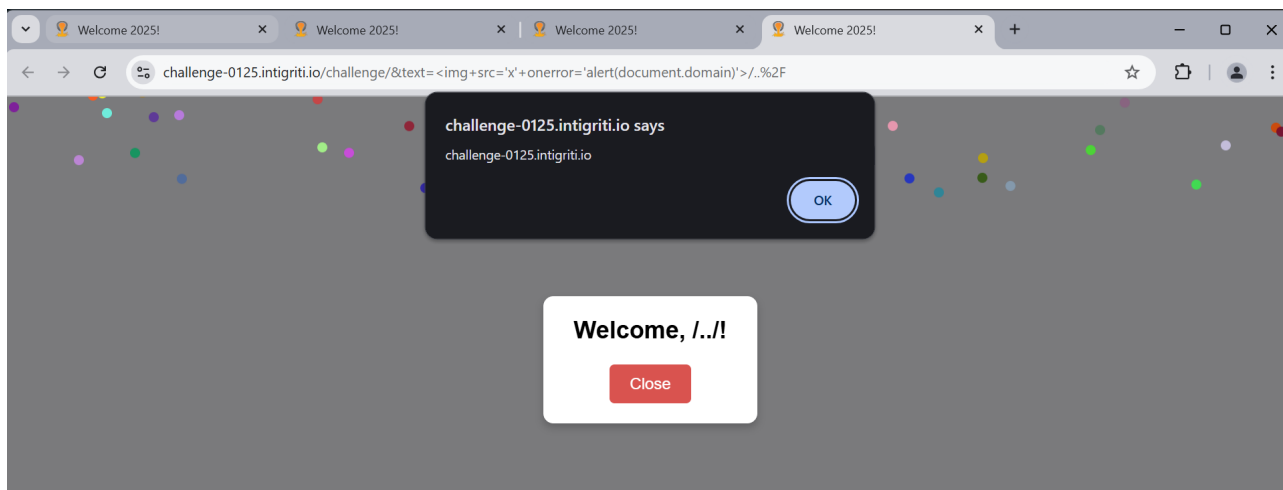


The shortest way to achieve XSS would now be the payload “<script>alert(document.domain)</script>” for example but the “checkQueryParam()” uses “innerHTML” to add our input into the source code. “innerHTML” will not execute script tags so we need to use another payload like following for example:

```
<img src='x' onerror='alert(document.domain)'\>
```

This gives following URL where spaces are replaced by + characters:

[https://challenge-0125.intigriti.io/challenge/&text=%3Cimg+src='x'+onerror='alert\(document.domain\)'\>/.%2F](https://challenge-0125.intigriti.io/challenge/&text=%3Cimg+src='x'+onerror='alert(document.domain)'\>/.%2F)



This one pops the alert and executes arbitrary JavaScript both in Chrome and Firefox and thus solves this challenge with a successful XSS attack.

The XSS() function which normally sanitizes input by searching URL parameters and hash fragments sees there are no parameters and also not a hash part in the URL because we did not use a “?” in the URL so the function returns false as it cannot find any malicious character.

The “getParameterByName()” function does check the complete url and finds “&text=” in our malicious URL as input. It takes the value of this “&text=” part and embeds it into the HTML source code.