Intigriti May 2023 challenge

The challenge presents a form in which the user can insert Javascript code to be executed by the browser. The user provided code is embedded into a `script` element, as part of the `src` attribute, and the script is then appended to the document's head, resulting in the code execution.
The application code, prior to adding the script element, performs the following operations:

- retrieves the `xss` URL parameter
- checks that the `xss` paramter length is shorter than 100 characters
- checks that the `xss` parameter matches the regular expression `/^[a-zA-Z,'+\\.()]+$/`:
  - i.e., it is composed only of alphabetical characters, comma, dot, plus sign (+) and parentheses
- checks that the parameter doesn't match any of the `words` keywords (case insensitive), in an attempt to prevent XSS payloads.

```
<script>
  (()=>{
    opener=null;
    name='';
    const xss = new URL(location).searchParams.get("xss") || '';
    const characters = /^[a-zA-Z,'+\\.()]+$/;
    const words =
    /alert|prompt|eval|setTimeout|setInterval|Function|location|open|document|script|url|
    HTML|Element|href|String|Object|Array|Number|atob|call|apply|replace|assign|on|write|
    import|navigator|navigation|fetch|Symbol|name|this|window|self|top|parent|globalThis|
    new|proto|construct|xss/i;
    if(xss.length<100 && characters.test(xss) && !words.test(xss)){

      script = document.createElement('script');
      script.src='data:,'+xss
      document.head.appendChild(script)
    }
    else{
      console.log("try harder");
    }
  }
  )()
  </script>
```

Additionally, the application implements the following Content-Security-Policy:
`<meta http-equiv="Content-Security-Policy" content="script-src 'none'; script-src-elem data: 'unsafe-inline'">`

that restricts the operations that can be performed through the Javascript code (e.g. no `eval` is allowed, on top of being filtered by the `words` regular expression).

The challenge sanitization code is fairly restrictive, since it limits the usage of most of the relevant keywords needed to execute meaningful code (including the challenge requested

`alert(document.domain)`). Specifically, the `words` regular expression makes it (likely?) impossible to reach the `window` and `document` objects directly. Similarly, the `alert`, `eval` and several other potentially useful functions are detected, preventing the code execution.

To execute the `alert(document.domain)` function call, then, we need a way to indirectly reach the blacklisted components (i.e. `alert` and `document`, specifically).

ECMAScript 6 introduced, among other things, the `Reflect` APIs that, much like other Object Oriented Programming languages, allow to manipulate properties, variables, and object methods at runtime.

Among the exposed APIs (listed here), the `Reflect` namespace object exposes the `Reflect.get` and `Reflect.set` methods that, respectively, allow to get and set an object's property value.

With the mentioned functions, consequently, it is possible to retrieve and set properties of any object, without calling them directly. As an example:

`Reflect.get(window,'alert') is equivalent to window.alert`.

In this way, we have however retrieved the `alert` property by referencing it through a string. This allows to easily obfuscate it and bypass the `words` regular expression check.

Good. Now we need to retrieve the different pieces of the function to execute, always remembering that we also have a size constraint to respect as well.

Let's try to create a payload as short as possible. One of the first things that came up to my mind was to rely on the `Function` constructor, to directly create a function like:

`function a(){alert(document.domain)}`.

To do this, we can rely on several objects that are exposed by default by Javascript, such as `Map`, and take its `constructor` , which is the `Function` constructor. Through the `Reflect` API we can do it like this:

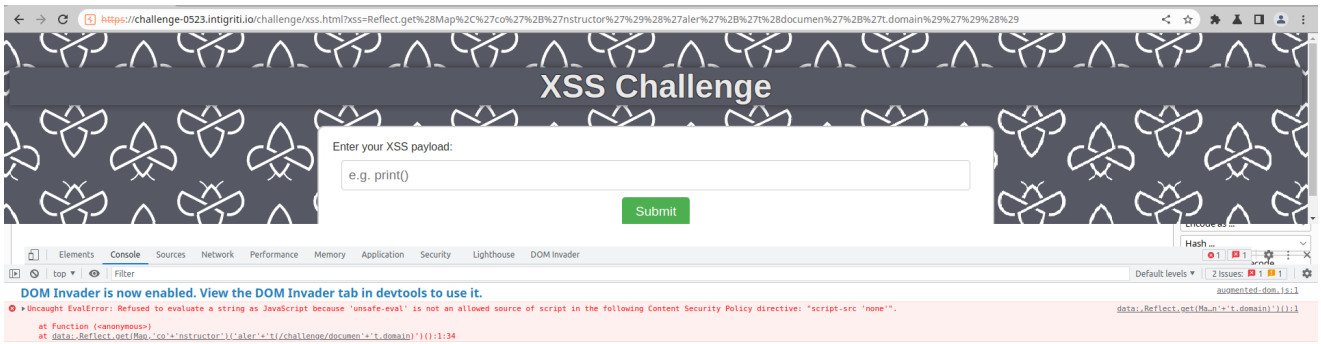
```
Reflection.get(Map,'co'+'nstructor')
```

note that the `constructor` string has been split into `co` and `nstrcutor` in order not to match the `words` regular expression. Now that we have the `Function` constructor we can then create a function that calls `alert(document.domain)` and directly call it, as follows:

```
Reflect.get(Map,'co'+'nstructor')('aler'+'t(documen'+'t.domain)')()
```

note that, as for the `constructor`, even `alert` and `document` have been properly split into multiple strings to bypass the regex check.

Nice, easy and short, except...



Well, turns out that the `Function` constructor actually relies internally on `eval`, which is blocked by the Content-Security-Policy. The list of functions that are blocked by the Content-Security-Policy without the `unsafe-eval` directive is reported by the Mozilla docs here.

Let's look for another route.

We know for sure that we have to rely on the `Reflect` API, however, without relying on a custom function, or an `eval` equivalent, we necessarily need to directly use the `window` and `document` objects. Let's try to retrieve them then.

Javascript, when trying to access an object, by default searches for all the objects under the `this` object, which, outside of any object method, is the `window` object. This means that, normally, we can reach any object under `window`, if not in the regular expression blacklist. So, we want then to reach the `window` object from one of these. To do this, we can search for those objects that have the `top` or `parent` property element pointing to `window` (since `top` is shorter than `parent` we will first look for that).

```
>> for(el in window){
     try{
       if(window[el].hasOwnProperty('top') && window[el]['top']===window){console.log(el)}
     }catch(err){}
   }
   0
   self
   frames
   parent
⚠ InstallTrigger is deprecated and will be removed in the future.
⚠ onmozfullscreenchange is deprecated.
⚠ onmozfullscreenerror is deprecated.
   window
   top
← undefined
```

Looks like the `frames` object is actually not in the `words` list and, consequently, a viable option.

Now for the `document` part:

```
>> for(el in window){
     try{
       if(window[el].hasOwnProperty('document') && window[el]['document']===document){console.log(el)}
     }catch(err){}
   }
   self
   frames
   parent
   window
   top
<- undefined
```

Same as before, the `frames` object is our target.
Now that we have an object that allows us to reach the `window` and `document` objects, we can start to create our payload:

- `Reflect.get(frames,'to'+'p')` gets the `window` object
- `Reflect.get(Reflect.get(frames,'to'+'p'),'aler'+'t')` gets the `alert` function from the `window` object, accessed through the Reflect API
- `Reflect.get(frames,'documen'+'t').domain` gets the `document.domain` element

Putting things togheter, we can call the `alert(document.location)` as follows:

Reflect.get(Reflect.get(frames,'to'+'p'),'aler'+'t')(Reflect.get(frames,'documen'+'t').domain)

The total string length is 94, below the 100 characters required.
The final URL payload is then: https://challenge-0523.intigriti.io/challenge/xss.html?xss=Reflect.get%28Reflect.get%28frames%2C%27to%27%2B%27p%27%29%2C%27aler%27%2B%27t%27%29%28Reflect.get%28frames%2C%27documen%27%2B%27t%27%29.domain%29