Integriti - Window Maker

Nandayo.

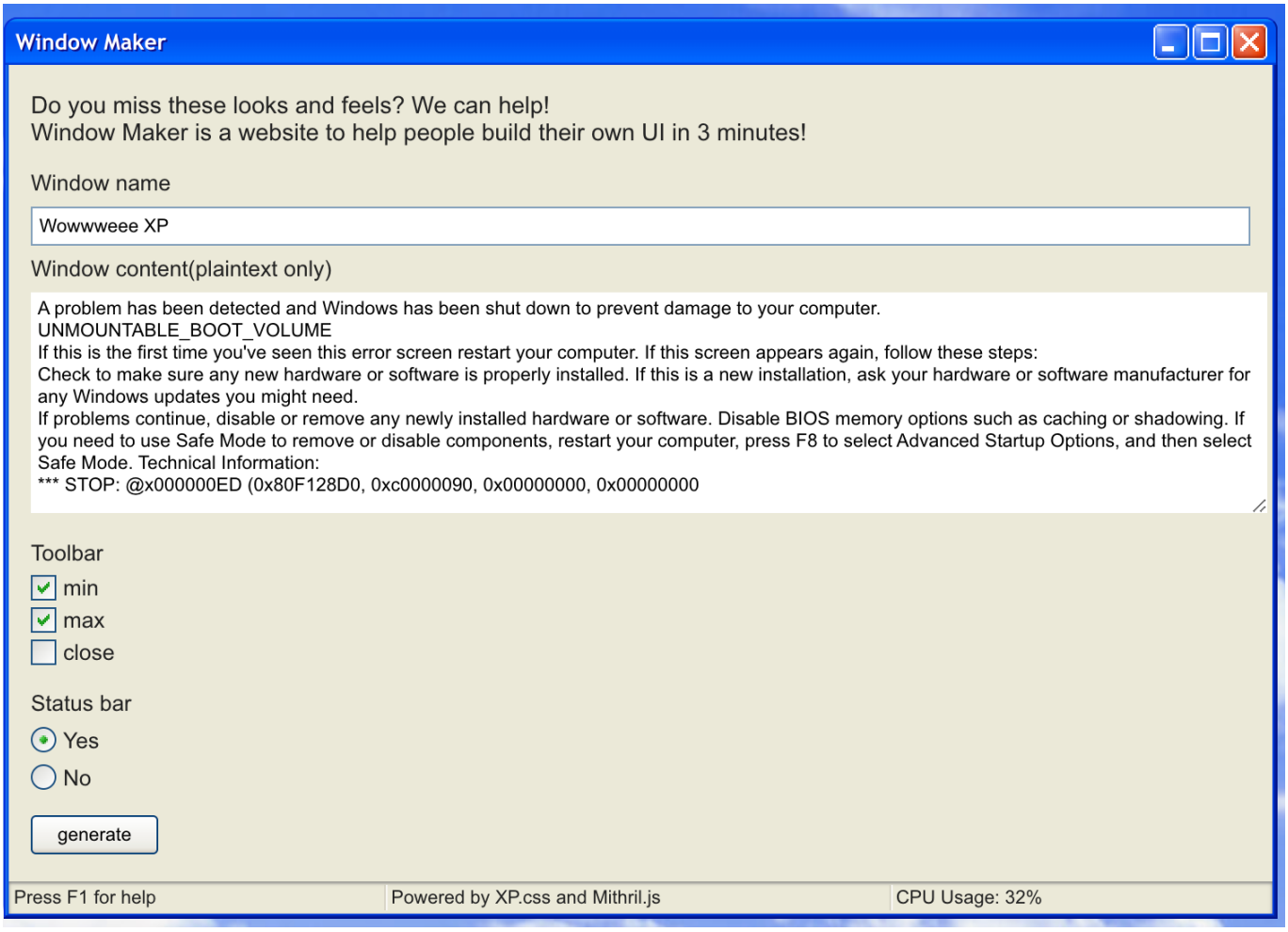
Introduction

This is my first real writeup on a web challenge, so please bare with me.

It was quite the journey. XSS is something I'm not new to, but it's not something I've really looked into a lot either. A friend and work colleague got me onto the Integriti challenges, and they've been very interesting. This one really got me though, at first glance it looked like it was an impossible task... but that was obviously not the case, and I learn a lot along the way.

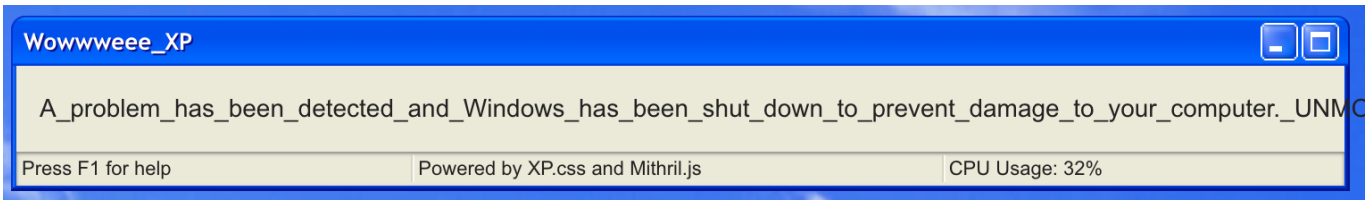
The challenge page (`https://challenge-0422.intigriti.io/challenge/Window%20Maker.html`) presented me with a window making web application built upon the Mithril.js library.

The web application



It allows the user to add some text and select some components before generating and rendering the custom window to the page (love the Windows XP aesthetics by the way).

The Mithril.js version was "2.0.4", so the first thing I did was jump onto the GitHub repo for some recently disclosed issues. It turns out there were no active issues related to security. Dang. Looks like we've got some work to do.



The first thing I noticed while looking through the source code of the web application were the `appConfig` and `devSettings` objects in the `main()` function. It is also important to note that these objects are instantiated with `Object.create(null)`. Basically this means the object has no properties. Not even prototypes. It's completely empty. This will become important later. The `appConfig` object was initialised with some default values before being passed to a `merge()` function along with `qs.config`, which was defined at the start of the `main()` function using `m.parseQueryString(location.search)`.

```
const qs = m.parseQueryString(location.search)

let appConfig = Object.create(null)
appConfig["version"] = 1337
appConfig["mode"] = "production"
appConfig["window-name"] = "Window"
appConfig["window-content"] = "default content"
appConfig["window-toolbar"] = ["close"]
appConfig["window-statusbar"] = false
appConfig["customMode"] = false

if (qs.config) {
    merge(appConfig, qs.config)
    appConfig["customMode"] = true
}
```

The `m` variable was the object used to interface with the Mithril.js library, and reading the documentation on `parseQueryString` reveals this function would parse the "query parameter" string provided and produce a object. The result doesn't just have to be a flat object though, the function also handles type casting of booleans (true, false) and deep data structures such as:

```
"test[1]=first&test[2]=second" => { test: [ "first", "second" ]  }
```

The documentation for `parseQueryString` can be found here: https://mithril.js.org/parseQueryString.html

The produced object was passed to the `merge()` function along with the `appConfig`.

What does the merge function do?

Just as the name suggests, the `merge()` function takes two objects, iterates over the second one provided, and copies the key / value pairs into the first target object. This process was protected by a list of `protectedKeys` which contains a deny list of keys. From the quick on the spot research it seems these banned keys are often used in prototype pollution attacks.

The function also takes into account depth, recursively iterating the object's structure if a particular value was not "primative". This was determined by the `isPrimitive()` function.

```
function merge(target, source) {
    let protectedKeys = ['__proto__', "mode", "version",
                         "location", "src", "data", "m"]

    for(let key in source) {
        if (protectedKeys.includes(key)) continue

        if (isPrimitive(target[key])) {
            target[key] = sanitize(source[key])
        } else {
            merge(target[key], source[key])
        }
    }
}
```

The `isPrimitive()` function simply returns whether or not the value provided was one of the five (5) defined primitive types (null, undefined, string, boolean, and number), as these cannot be iterated any further by the `merge()` function.

```
function isPrimitive(n) {
    return n === null || n === undefined || typeof n === 'string'
               || typeof n === 'boolean' || typeof n === 'number'
}
```

Back to the `merge()` function for a second. Looking through previous writeups on prototype pollution revealed custom merge functions to a major contributor to these types of attacks. We've pointed out already that the `__proto__` key was banned, which was not great on an initial first look, however, there are ways around this... sort of. The use of `constructor.prototype` can be handy for influencing prototypes of some adjacent prototypes. It's not on the deny list, so we'll keep it in the back of our minds for now.

This article (What is the difference between prototype and **proto** in JavaScript?) clears it up quite well.

Continuing the investigation

```
let devSettings = Object.create(null)
devSettings["root"] = document.createElement('main')
devSettings["isDebug"] = false
devSettings["location"] = 'challenge-0422.intigriti.io'
devSettings["isTestHostOrPort"] = false
```

```
if (checkHost()) {
    devSettings["isTestHostOrPort"] = true
    merge(devSettings, qs.settings)
}

if (devSettings["isTestHostOrPort"] || devSettings["isDebug"]) {
    console.log('appConfig', appConfig)
    console.log('devSettings', devSettings)
}
```

The second interesting object was `devSettings`. Like `appConfig` it was an empty object populated with some default values.

A call to `checkHost()` was then perfomed:

```
function checkHost() {
    const temp = location.host.split(':')
    const hostname = temp[0]
    const port = Number(temp[1]) || 443
    return hostname === 'localhost' || port === 8080
}
```

The `checkHost()` function checked the hostname and port in the URL for `localhost` OR `8080` respectively. We'll call this barrier #1.

If this was bypassed then two things would happen:

- A second `merge` would occur where another set of user controlled parameters are merged into `devSettings`.
- `console.log`s would fire, letting us know we're in some sort of developer mode.

So...

What now? Pollute!!!

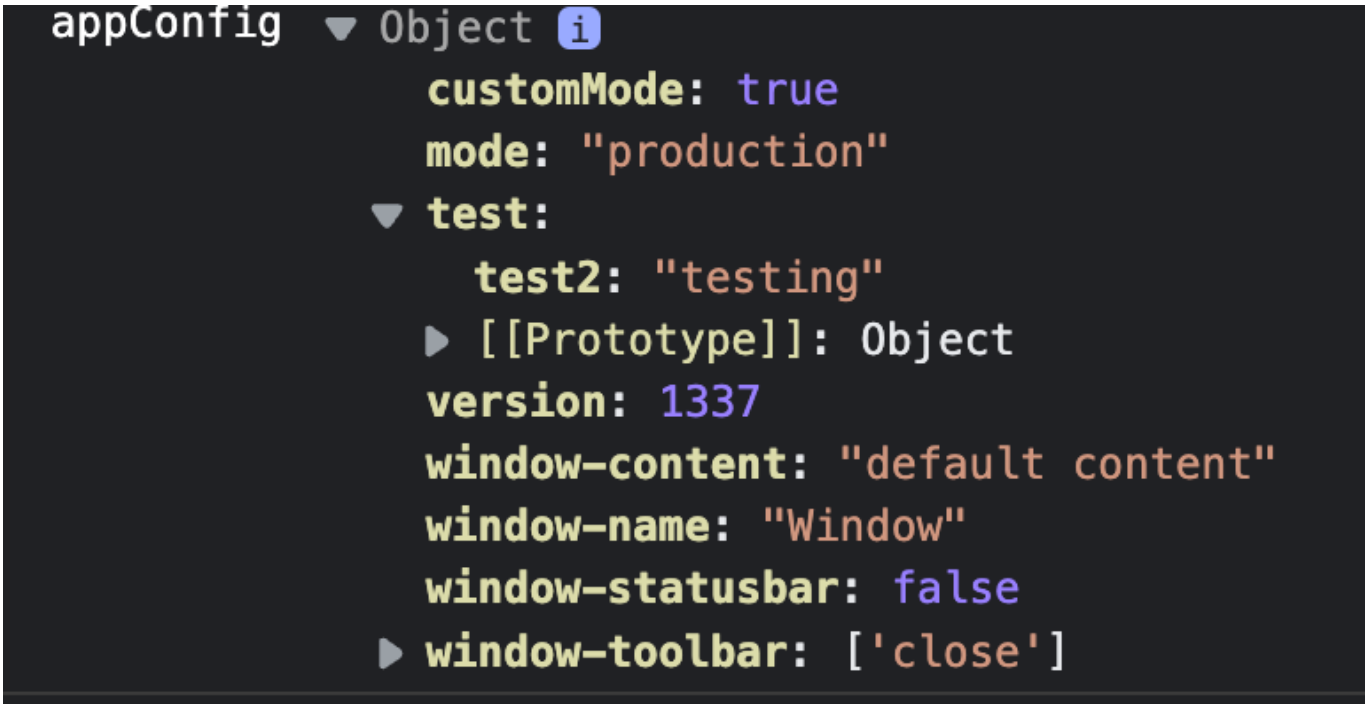
From the looks of things, prototype pollution still semmed like the most viable option.

The first thing I did was make sure to use a local copy for testing. This way I could also modify the `appConfig` and `devSettings` variables such that they were accessible from a global scope. This would allow me to inspect them in the console without any restrictions.

Running the query:

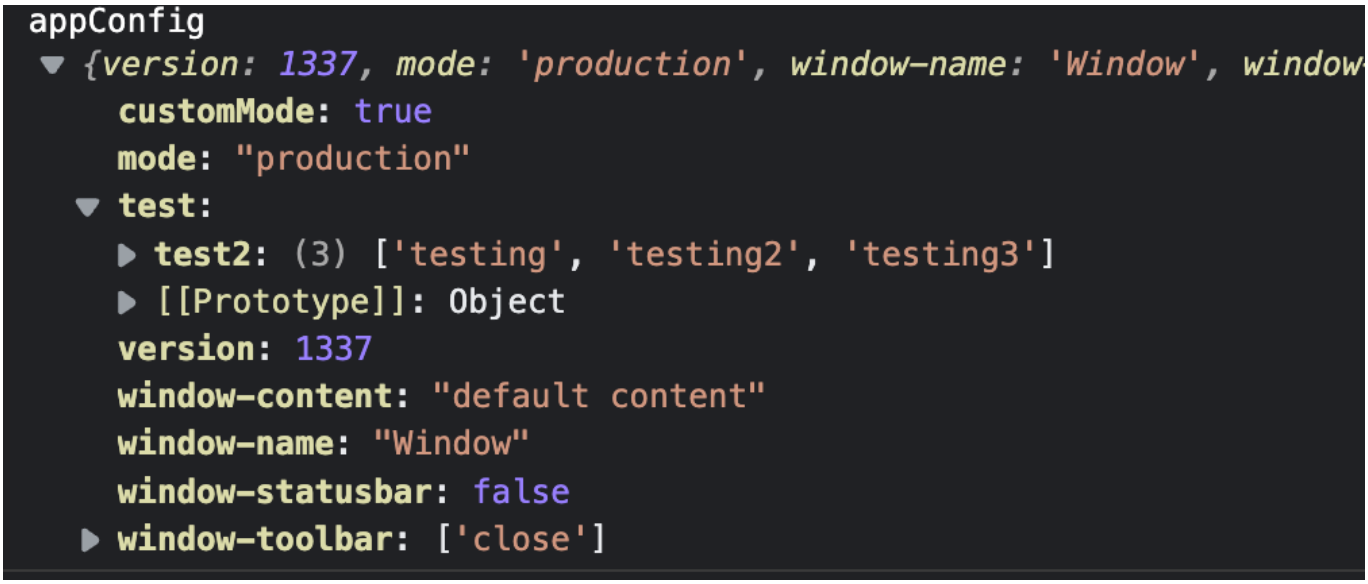
http://localhost:8000/Window%20Maker.html?config[test][test2]=testing

Results in the following:



Playing around it was also possible to create arrays using the following query:

http://localhost:8000/Window%20Maker.html?config[test][test2][]=testing&config[test][test2][]=testing2&config[test][test2][]=testing3



Using the chrome debug tool, it I emulated the query syntax and viewed the `constructor.prototype` for our new array:

```
> appConfig['test']['constructor']['prototype']
< ▼ {constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ,
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
```

I then tried adding a new field using:

http://localhost:8000/Window%20Maker.html?config[test][test2][]=testing&config[test][constructor][prototype][testing123]=test321

```
> appConfig['test']['constructor']['prototype']['testing123']
< 'test321'
```

And there it was! But just to be sure, I checked a new empty array for the same field:

```
> ([])['constructor']['prototype']['testing123']
< undefined
```

Sadly, it wasn't there.

After a bit of playing around I ended up with the following query:

http://localhost:8000/Window%20Maker.html?config[window-toolbar][constructor][prototype][testing123]=test321

```
> ([])['constructor']['prototype']['testing123']
< 'test321'
```

`config[window-toolbar]` was an array field set in the `appConfig` object before the merge, so potentially this was why the custom one didn't work? I'm still unsure about this, but we can overwrite the prototype and have some Array prototype pollution!

Breaking down the first barrier

What can we do with this new-found power? Well, not a whole lot, really. Ideally when it comes to prototype pollution you want to be polluting the "Object" object. This leads to every object inheriting the new field. Also, we can only write booleans, strings, or arrays of the two. That doesn't give us much.

The next thing I wanted to do was look more at the first barrier, the `checkHost()` function. Somehow we need to meet one of the two criteria:

1. Set our host to `localhost` in the URL, or
2. Set our port to `8080` in the URL.

Looking again at the code:

```
function checkHost() {
    const temp = location.host.split(':')
    const hostname = temp[0]
    const port = Number(temp[1]) || 443
    return hostname === 'localhost' || port === 8080
}
```

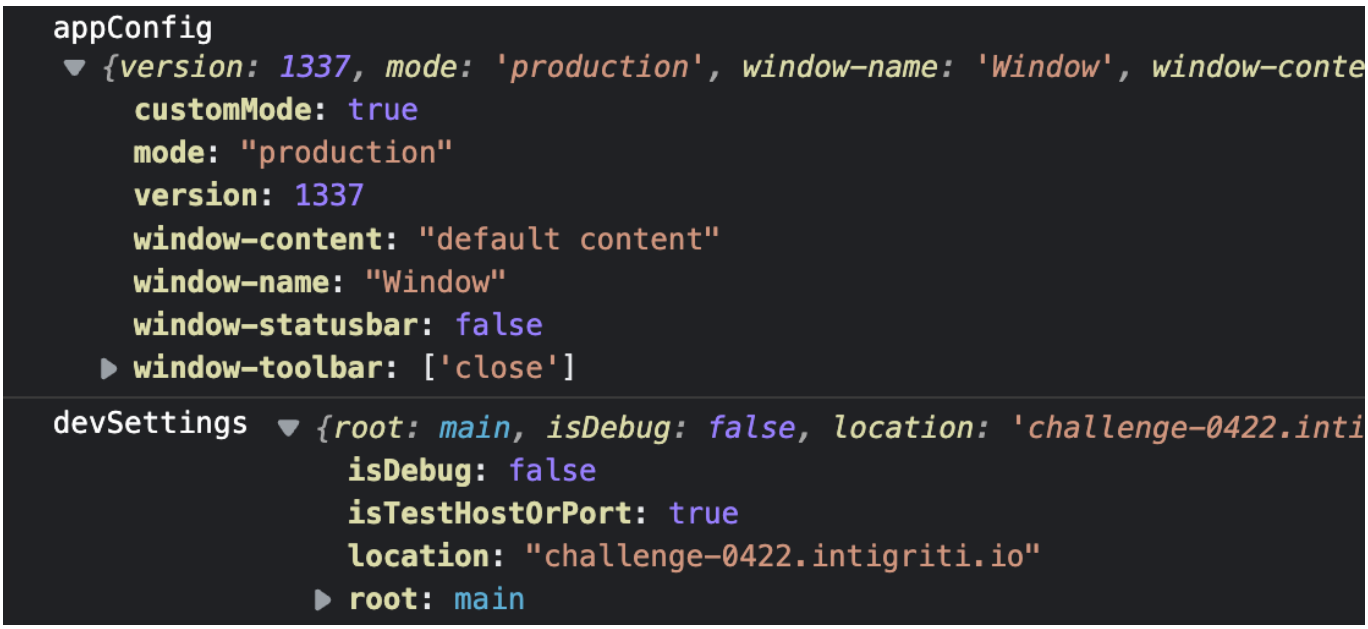
Array operations are being performed on `temp[0]` and `temp[1]`. These look like something we can target.

After some trial and error, setting the `1` key of the array prototype would lead to the `const port = Number(temp[1]) || 443` line returning a value of our choice, in this case `8080`. Even though we can only write strings, the string was converted to a number for us, so this isn't an issue. The `0` was tested, but it did not give us developer mode. Why? No clue.

The query now looked like this:

http://localhost:8000/Window%20Maker.html?config[window-toolbar][constructor][prototype][1]=8080

Running this now reveals the `devSettings` object in the console, as expected:



We now have a new object to play around with, this time with a more interesting field: `root`. It's referencing `document.createElement('main')`, which created an HTML element object. This surely has some

interesting properties.

How do we get some alerts popping? The second barrier.

This stage of the exploit took a lot longer than expected. Reading through the number of fields on the `root` HTML element took a while. I was really unsure where to go at this point. We can write a string almost anywhere (apart from readonly fields), but nothing really stood out.

I tried:

- Overwriting / constructing various `vnodes` and `vnode` properties that were created by Mitrhil.js.
- Inspecting prototypes of properties for the "Object" object.
- Replacing style sheet backgrounds with base64 encoded data SVGs in the hopes I could get an XSS out of that...

Little did I know a third (3rd) barrier was in the way, but we'll get to that shortly.

The real spark hit when I found the `ownerDocument` property (Read more here) [https://developer.mozilla.org/en-US/docs/Web/API/Node/ownerDocument]. Basically this references the `Document` Node in the DOM. This gave me a lot more to go through and waste a lot of time, retrying failed attempts, and trying to overwrite / construct nodes out of arrays.

I did end up overwriting the first `sctipt` tag with an `alert(1)`, however this was a dead end as `script` tags would not re-execute after the page load.

The payload looked like this:

```
http://localhost:8000/Window%20Maker.html?config[window-toolbar][constructor]
[prototype][1]=8080&settings[root][ownerDocument][scripts][0]
[innerHTML]=alert(1)
```

Eventually I came across the `body` node element sitting in the property of the same name. Turns out when you edit the `body` element with `innerHTML` it actually renders the HTML after the page load!

It's taken a while at this point, but we're able to write stuff onto the page. But it's not actually HTML we're writing. It's just text. The reason being...

Barrier 2.5? I forgot about one function

It turns out there was also a `sanitize()` function. It was in the `merge()` code I referenced before, but I didn't mention it... here's our old friend `merge()`:

```
function merge(target, source) {
    let protectedKeys = ['__proto__', "mode", "version",
                         "location", "src", "data", "m"]

    for(let key in source) {
        if (protectedKeys.includes(key)) continue

        if (isPrimitive(target[key])) {
            target[key] = sanitize(source[key])
```

```
            } else {
                merge(target[key], source[key])
            }
        }
    }
```

As you can see, before writing the value in our user controlled `source` object to the `target` object, the value is passed to the `sanitize()` function.

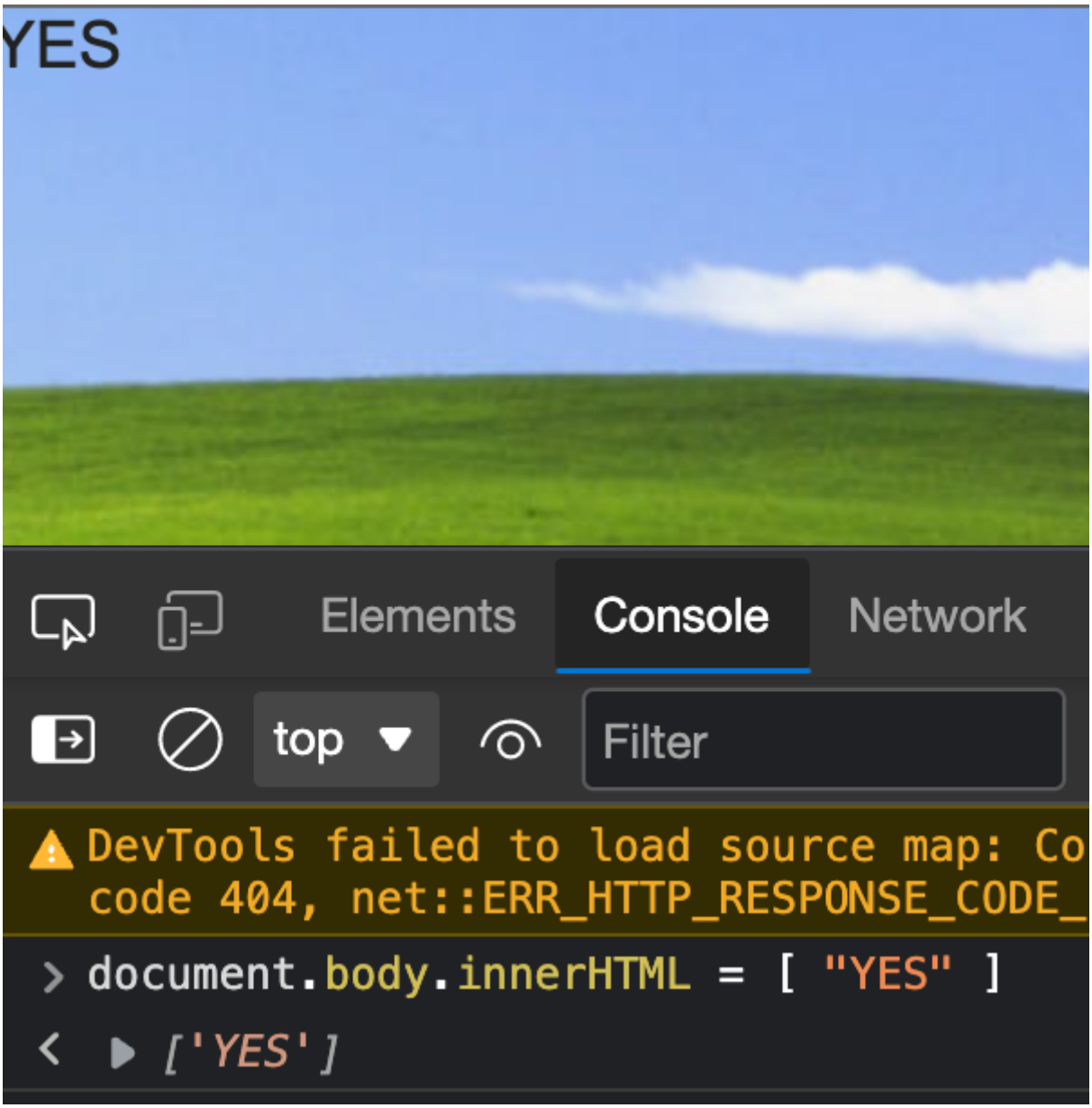
```
function sanitize(data) {
    if (typeof data !== 'string') return data
    return data.replace(/[<>%&\$\s\\]/g, '_')
               .replace(/script/gi, '_')
}
```

The function was straight forward, we pass in a `data` variable, and if it was a string it was cleaned of a bunch of useful characters and the word `script`. Anything I tried writing the page was being partially replaced with underscores.

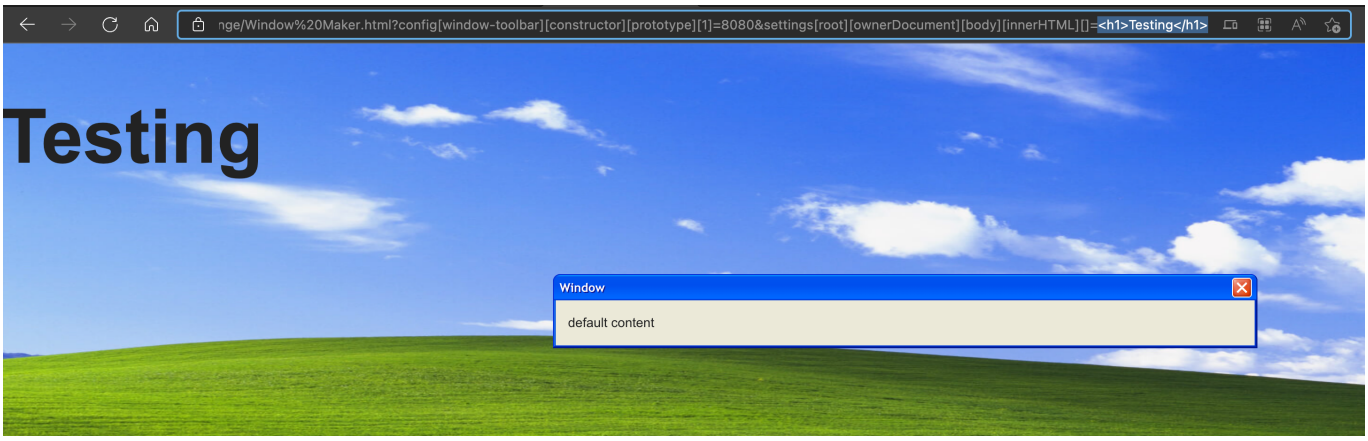
I tried to bypass this, but I knew it looked solid. I tried UTF-7, UTF-16, changing a `meta` tag to `charset=UTF-7`, changing the `html` tag's `lang` attribute... nothing worked. It was a long shot, but I came to the realisation there's got to be a better way. Something that was staring me in the face... something I've missed... Again.

That's right, the `sanitize()` function again! The first line, `if (typeof data !== 'string') return data`, it'll just return `data` if it's not a string, so what if we just don't use a string? We know we can write booleans, strings, and arrays of one of both to a field on an object, what if we just write an array with one string? JavaScript is good with letting things like this go. If it bypasses the check, then we just write to the DOM using `innerHTML` with an array of one string, instead of with a string directly.

Does this even work?



No errors, let's give it a shot! To test the `sanitize` bypass I just used `<h1>Testing</h1>`:

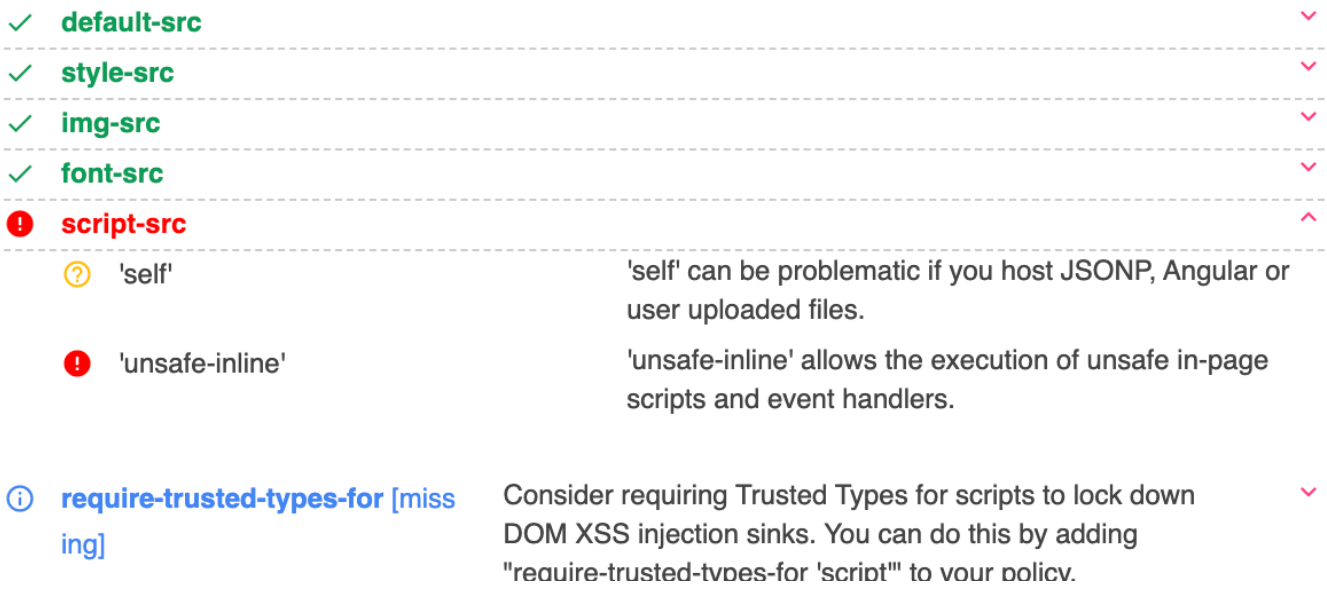


Very nice, we have some HTML rendering!

Barrier 3. I missed one more thing...

At this point popping the alert seems trivial, and then I CSP was a thing. I better check it:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'none';
style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self';
script-src 'self' 'unsafe-inline';">
```



Okay, nevermind. I'm lucky I didn't spend any time on default-src, style-src, img-src, or font-src in my previous ramblings (I did spend a lot of time on these, I just forgot about CSP until now 😄).

Final Payload

Alright, time to construct the final paylod. Okay, so I tried a few different payloads. URL encoded, not encoded, hacktricks, etc. Eventually I stumbled across: `%3Cimg%20src%3Dx%20onerror%3Dalert(document.domain)%3E`.

It's just `<img src=x onerror=alert(document.domain)>` but all the characters are URL encoded, not entirely sure why this one stays and pops the alert, but it does.

Final payload:

```
https://challenge-0422.intigriti.io/challenge/Window%20Maker.html?
config[window-toolbar][constructor][prototype][1]=8080&settings[root]
[ownerDocument][body][innerHTML]
[]=%3Cimg%20src%3Dx%20onerror%3Dalert(document.domain)%3E
```

Thanks for reading

This was a really difficult challenge for me. The writeup may seem like I knew what I was doing, but web is still relatively new to me and there's so much for me to learn. Thanks Integriti for putting this together, it was a blast. I'm going to go to bed now.