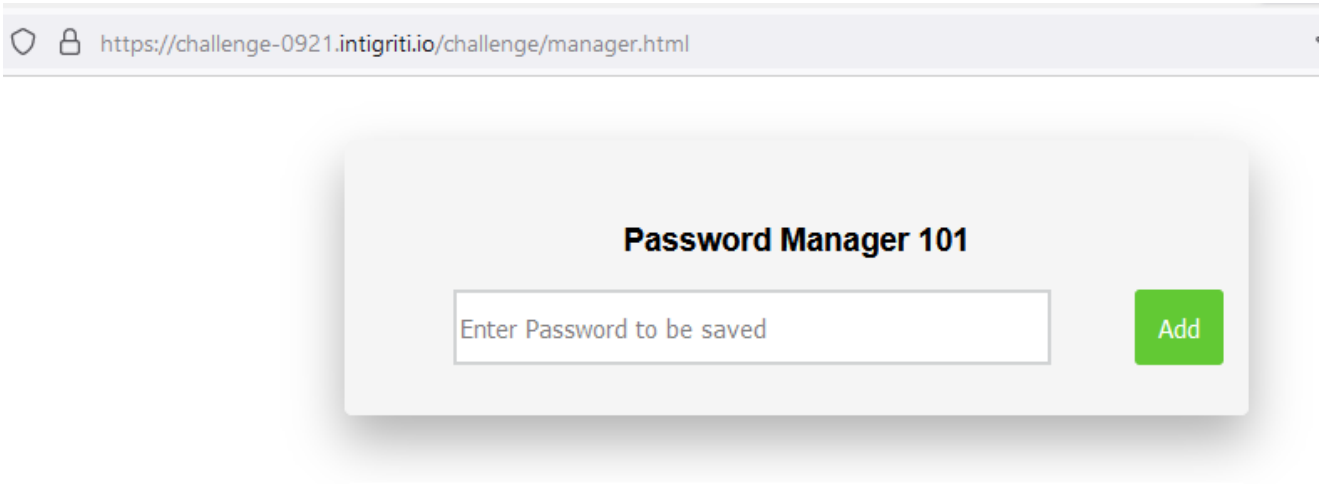
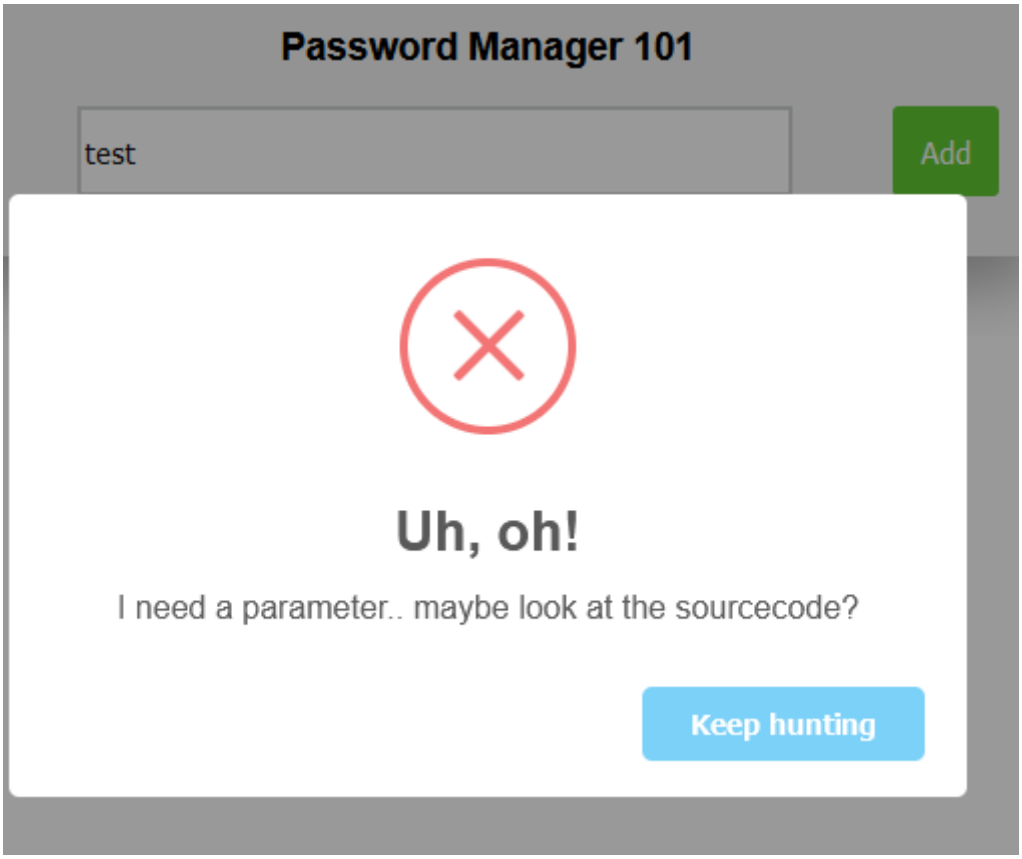
Phase-1 : WTF???

This time we are presented with a password manager



Trying to input some random password in there and hitting `add` gives us an error



So to proceed we check the page source first, we can see 2 scripts are included here

```
<script async src="manager.js"></script>
<script src="sweetalert.min.js"></script>
</body>
```

Notice how we have a async source for the manager.js. This simply means that the script is fetched asynchronously, and when it's ready the HTML parsing is paused to execute the script, then it's resumed. Just a way to efficiently load the script (check references for more info).

Doing a quick check to see if sweetalert.min.js has any vulns leads to a conclusion that there are no vulns!

But we can pretty much be sure that this is what produces that prettified alert box asking us to look for the parameter in the source sourcecode.

Checking the manager.js to inspect the sourcecode (WTFFF?????).

```
}(a, 0x9a1c7 + 0xa21 * -0x4 + -0x1 * -0x

function b(c, d) {
    var e = a();
    b = function(f, g) {
        f = f - (0x242 + -0x1feb * -0x1
        var h = e[f];
        if (b['HZYYok'] === undefined) {
            var i = function(m) {
                var n = 'abcdefghijklmnc
                var o = '';
                var p = '';
                for (var q = -0xfcd * -0
                    s = n['indexOf'](s);
                }
                for (var u = 0xd6 * 0x24
                    p += '%' + ('00' + c
                }
                return decodeURIComponer
            };
            b['hPvxJz'] = i;
            c = arguments;
            b['HZYYok'] = !![];
        }
```

Here we are dealing with obfuscated javascript and most likely the next step is to figure out this `parameter` to be able to proceed.

Phase 2 - Find Injection Point

To be able to find this parameter, we dig a little deeper into this manager.js file. On a quick sift, we can figure out a few things.

We notice these 2 big arrays which are referenced throughout and they contain some weird obfuscated entries.

```
function a() {
    var g4 = ['AgLNAa', 'r2PczKi', 'C3LTyM9S', 'BwfZA3vUA
        s0vfuf9dt05uru5u', 'i2fKza', 'r0PxEg0', 'BxrHyMXL
        'mtm3ndvUBu51AgC', 'zw5JDhLWzq', 'C3rYB2TL', 'B3b
        z3zJCMC', 'C3zN', 'Cgf0DgvYBNrYyw5ZzM9YBq', 'sxnm
        BgvNzw5K', 'zgvWDgG', 'zgvS', 'Eg1SoNnWywnL', 'Ag
        ', 'BM9ICG', 'y2fSBa', 'y2vSBhnWywnPBMC', 'DMzeB0
        ywnYB255Bq', 'D2LKDgG', 'Aw5Uzxjive1m', 's3Lnuwq'
        uKvuvvjox0rptv9guKfhtuvova', 'i25LDY1WyxnZD29Yzcb
         'Cg9ZDgvY', 'y3vYCMvUDfnJCMLWDa', 'zM9UDc1Myw1PB
```

```
}
var _0x5195 = [fn(-0x1b1, -0x112), fn(-0x23d, -0
    , fn(-0x75, -0x30), fn(0x11b, 0x1f), '1', fn
    , fn(-0x70, -0x11), fn(-0x22, -0x94), fn(-0x
    fn(-0x35f, -0x205), fn(-0x8a, -0x7a), fn(-0x
    -0xd9, -0x1d6), fn(-0x2e, -0x170), fn(-0x12e
    0x9b, -0x12e), fn(-0x350, -0x219), fn(-0xea,
    0x18), fn(-0x407, -0x265), fn(0x162, -0x13),
    276), 'mi', 'mn', 'mo', 'ms', fn(0xd8, -0x1d
```

So surely, at some point of time during execution we will know what these values actually are right?

We go ahead and put a breakpoint anywhere there is a reference to them and wait to see what values we got.

```
64 function b(c, d) {
65     var e = a();
66     b = function(f, g) {
67         f = f - (0x242 + -0x1feb * -0x1
68         var h = e[f];
69         if (b['HZYYok'] === undefined)
70             var i = function(m) {
```

Upon execution our break point hits and we can inspect what the values inside that big _0×5195 array are.

```
» _0x5195
← ▼ Array(712) [ "use strict", "]
    ▶ [0…99]
    ▶ [100…199]
    ▶ [200…299]
    ▶ [300…399]
    ▶ [400…499]
    ▶ [500…599]
    ▶ [600…699]
    ▶ [700…711]
      length: 712
    ▶ <prototype>: Array []
```

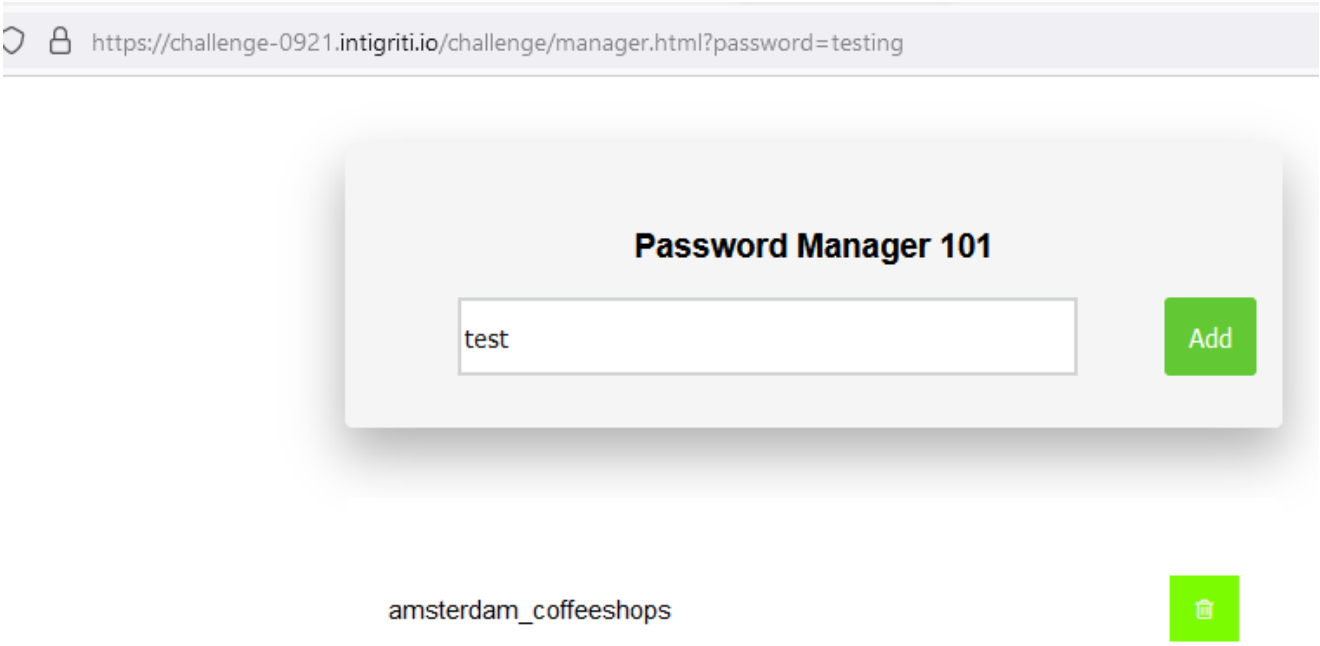
So we can see its length is 712 and since its referenced everywhere in the code, surely that parameter should be in there somewhere right?

Also when we say paramater, my brain thinks something like `/manager.html?PARAMATER=something`

There seem to be an interesting value like this in the array.

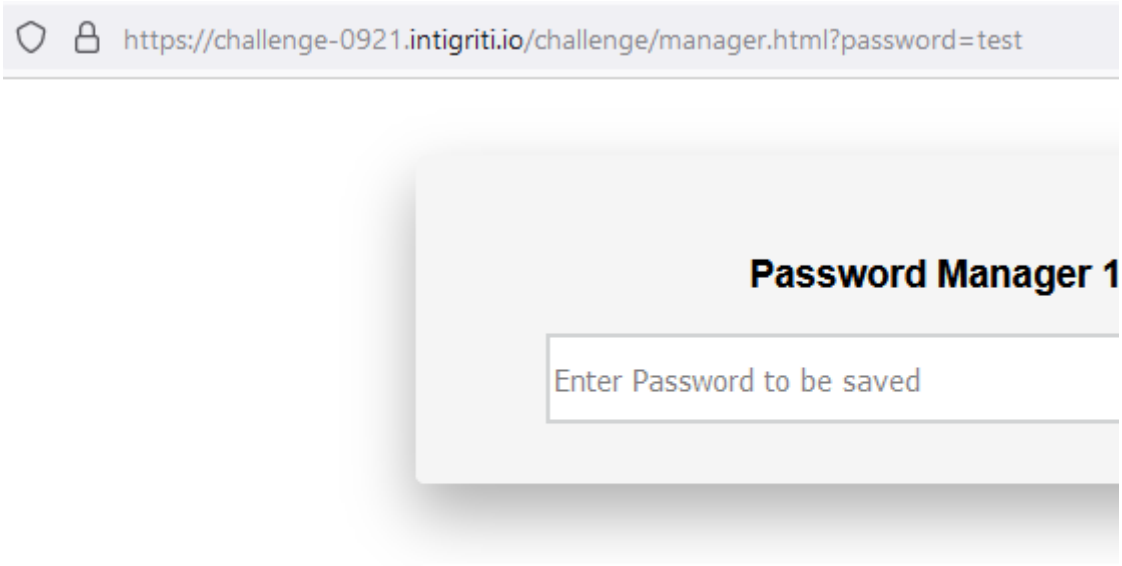
```
694: "error"
695: "Keep hunting"
696: "?password="
697: "+"
698: "replaceAll"
```

So seems like we have found the parameter, now lets try and put it in the URL and see what happens?



So now I am thinking, "WTFF is going on here?" how come putting `testing` in there leads us to this coffeeshop.

Weird enough putting some other values (in this case `test`) in there leads to some garbage



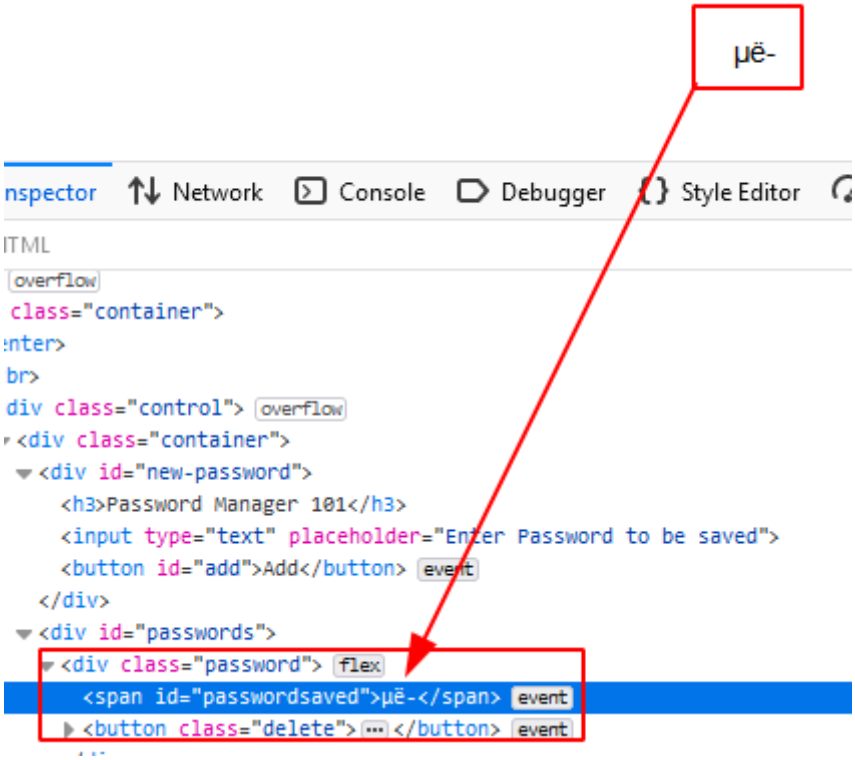
µë-

Experimenting with this a little more we come to a conclusion that any string with `LENGTH%4===0` will lead to garbage and anything else goes to the coffee shop

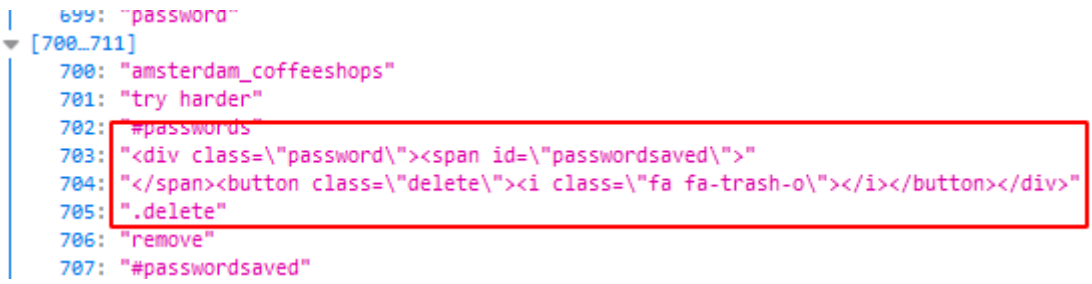
To understand what really this garbage is, we have to get a little deeper into debugging and find the point where this takes place in the code.

Phase 3 - The Garbage Mystery

To figure out the garbage mystery have to figure out what really happens with the string we provide to the parameter. We also notice that this garbage is embedded into the DOM as well.



Surely this is been done somewhere in that JS code? We head back to that big array of length 712 we found previously and look for something which might give us clue on how this thing is put in there.



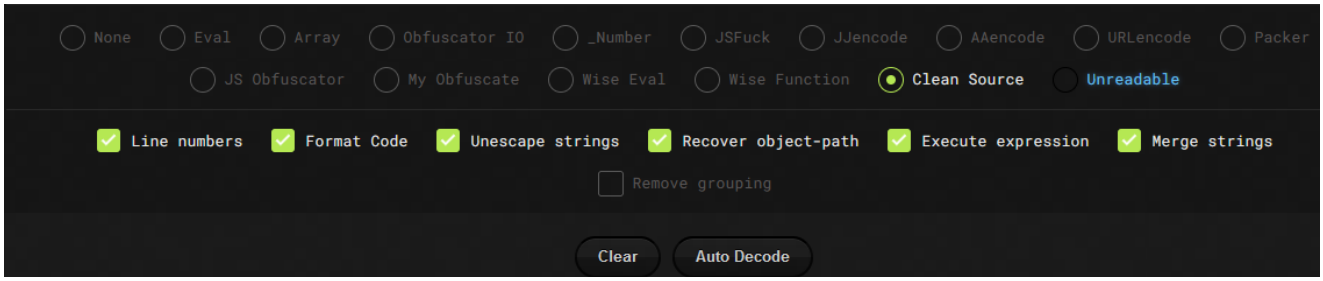
Looks like the HTML code is indeed stored in that array to perform this operation and its on number 703 inside the array. This means that in the JS code it MUST be retrieving it from the array something like this: `_0x5195[703]`.

But, there is a problem! we cannot eyeball or search it inside the code because those values are not placed directly in there. They look like this

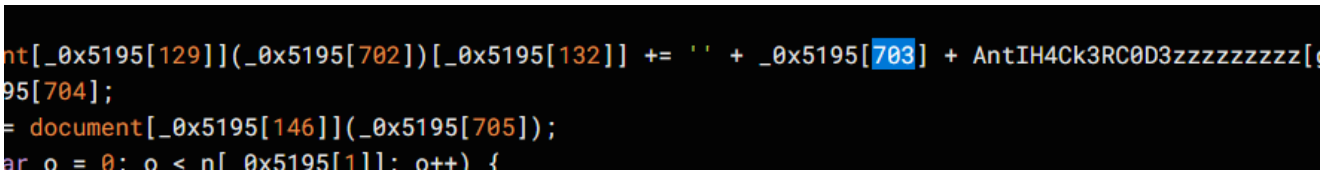
```
_0x5195[-0x184 * 0x8 + -0xa4f + 0x1674]);
```

To proceed from here, we need to calculate the value of all these expressions and figure out which one has value 703.

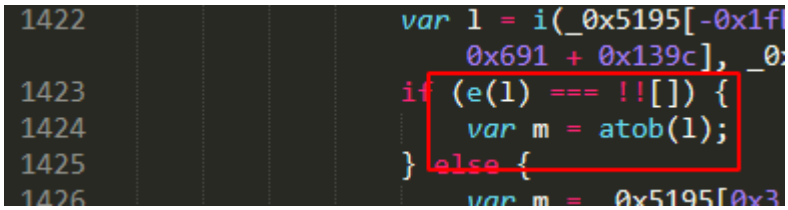
We head to this website: https://lelinhtinh.github.io/de4js/
We dump our code in there and simplify it to solve this expressions for us and look for `_0x5195[703]`



We can see this value does indeed come up in the code, precisely at line number 1429 in the original code.

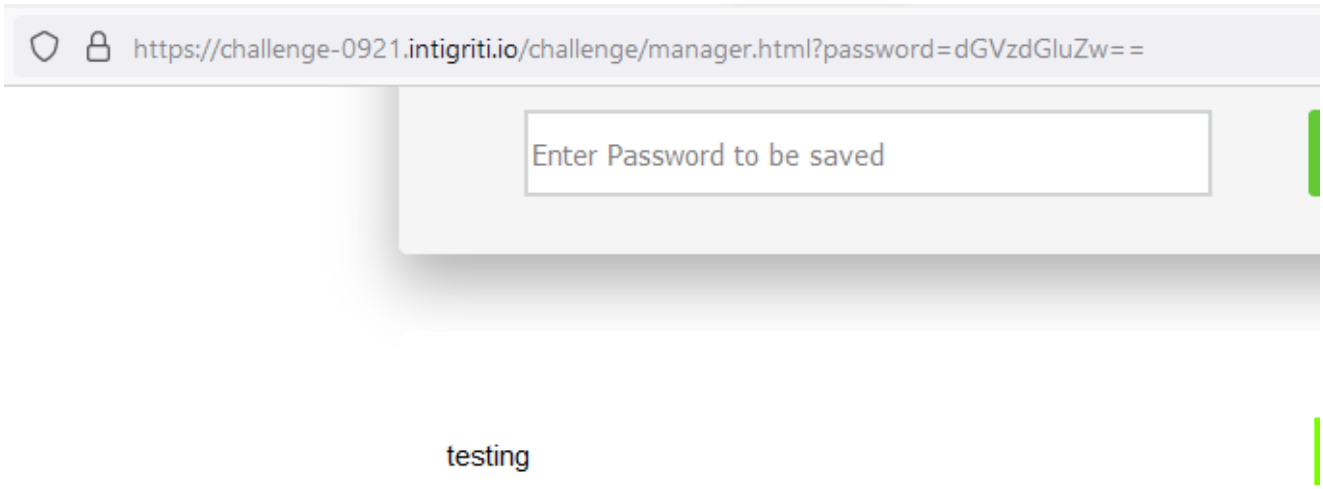


Looking at the code above this (before it puts into the DOM) we can see there is a call to `atob` at around line 1424,



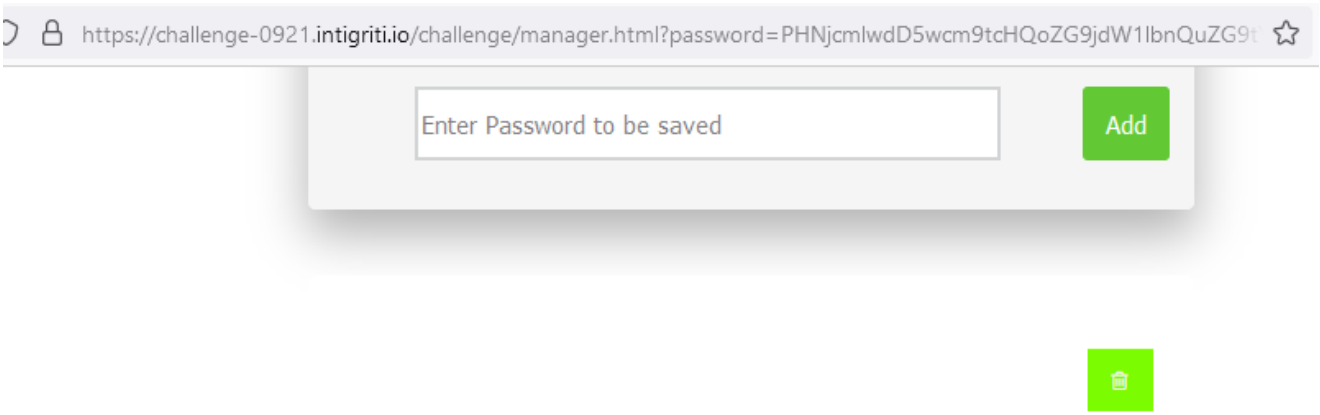
`atob` basically used to base64 decode a string, does that mean we are supposed to pass this thing a base64 encoded string which gets put into the DOM later on at line number 1429?

To confirm this theory we base64 encode a string named "testing" and pass it to the password parameter and if our theory is correct then "testing" should appear below, instead of garbage.



Looks like we are correct! "testing" does indeed appear in there, that means we should be able to get an XSS by simply base64 encoding `<script>prompt(document.domain)</script>`

We go ahead and try that payload:

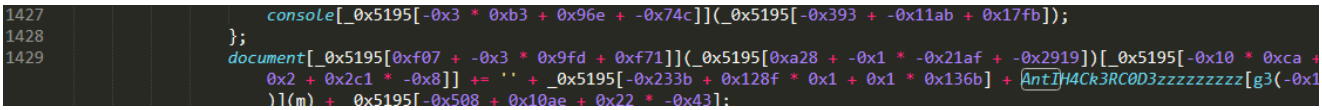


Nothing happens! its empty, which means there is something more happening which we don't know yet and further analysis is required.

Phase 4 - The hacker police

It looks like some sort of sanitisation might be happening in the JS code because of which our payload is rendered useless.

We notice that in the JS code there was a reference to something `antihacker`. This is on that same line number 1429, the time at which our payload is base64 decoded and inserted into the DOM.



This means before it gets put into the DOM some sanitization happens.

Lets decode what exactly that line means, going back to our deobfuscated code where these math expressions were evaluated, this what we get:

```
document[_0x5195[129]](_0x5195[702])[_0x5195[132]] += '' + _0x5195[703] +
AntIH4Ck3RC0D3zzzzzzzzz[g3(-475, -447)](m) + _0x5195[704];
```

Going back to that array to figure out what exactly these values means, we get the following:

```
_0x5195[129] => "querySelector"
_0x5195[132] => "innerHTML"
_0x5195[702] => "#passwords"
_0x5195[703] => "<div class=\"password\"><span id=\"passwordsaved\">"
_0x5195[704] => "</span><button class=\"delete\"><i class=\"fa fa-trash-o\">
</i></button></div>"
```

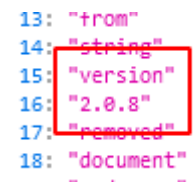
Which evaluates to:

```
document[querySelector](#passwords)[innerHTML] += '' + <div class="password">
<span id="passwordsaved"> + AntIH4Ck3RC0D3zzzzzzzzz[g3(-475, -447)](m) +
</span><button class="delete"><i class="fa fa-trash-o"></i></button></div>
```

So now we understand how this gets embedded in the DOM and here we can take a guess that `AntIH4Ck3RC0D3zzzzzzzzz[g3(-475, -447)](m)` is basically sanitizing our payload that we provide which is about to be embedded in the DOM.

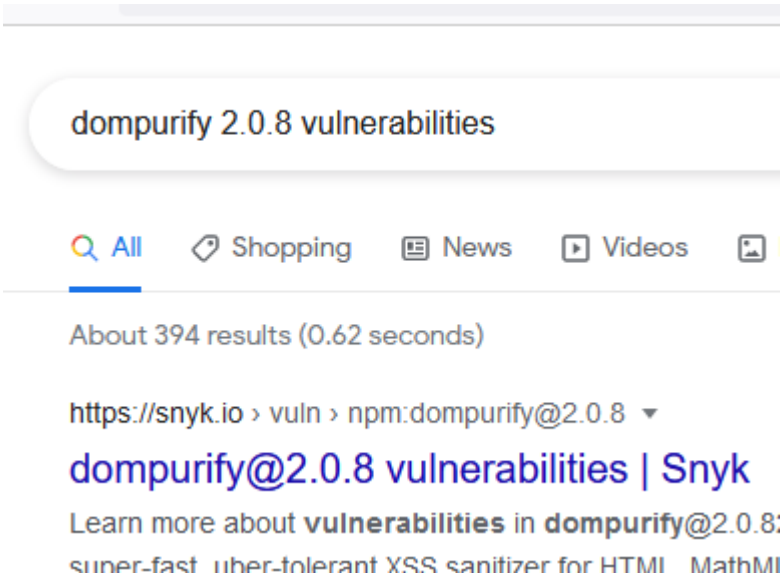
Now the main question is, what is this thing and how does it sanitize? Trying to debug it in the console lead me to nowhere and I hit a wall!

Looking back at that big array again we can see there is a version number in there

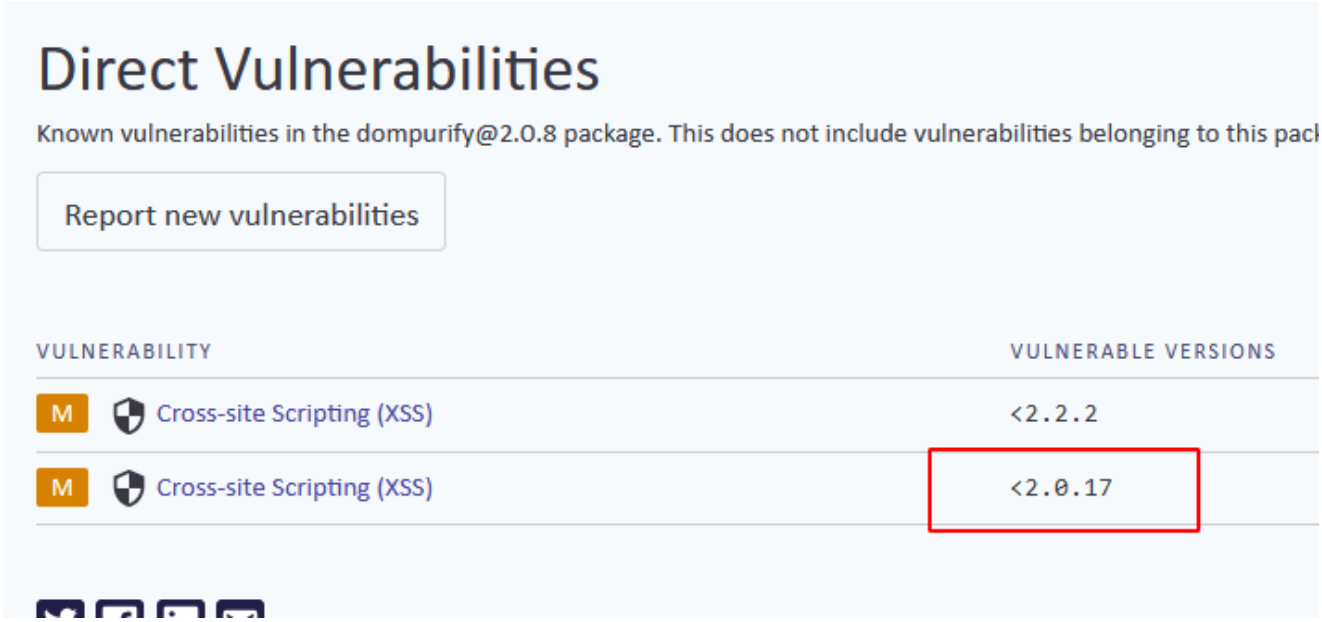


Maybe this thing uses some 3rd party stuff to sanitize the input we provide and this might be the version number of that.

Looking for sanitizers there are a few ones for example dompurify, sanitize-html. So we simply google "SCRIPT NAME 2.0.8 vulnerabilities"

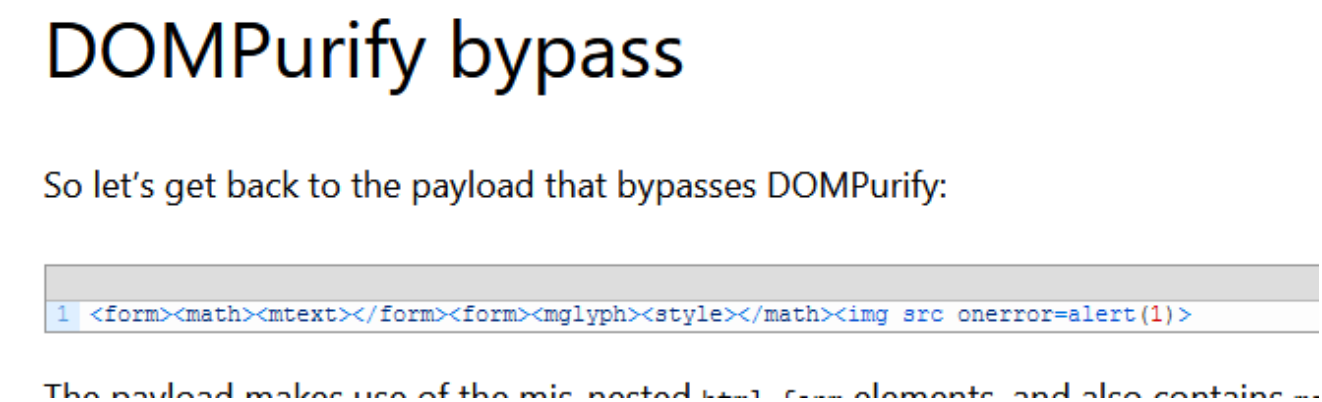


Doing this we see that this is most likely using dompurify (because results show up only for this with version 2.0.8) to sanitize our payload and version 2.0.8 already has an XSS vulnerability.



Phase 5 - Victory!

Following the research article linked in there, we can see that dompurify 2.0.8 already has an XSS vulnerability and the following payload should trigger this (check references for article):



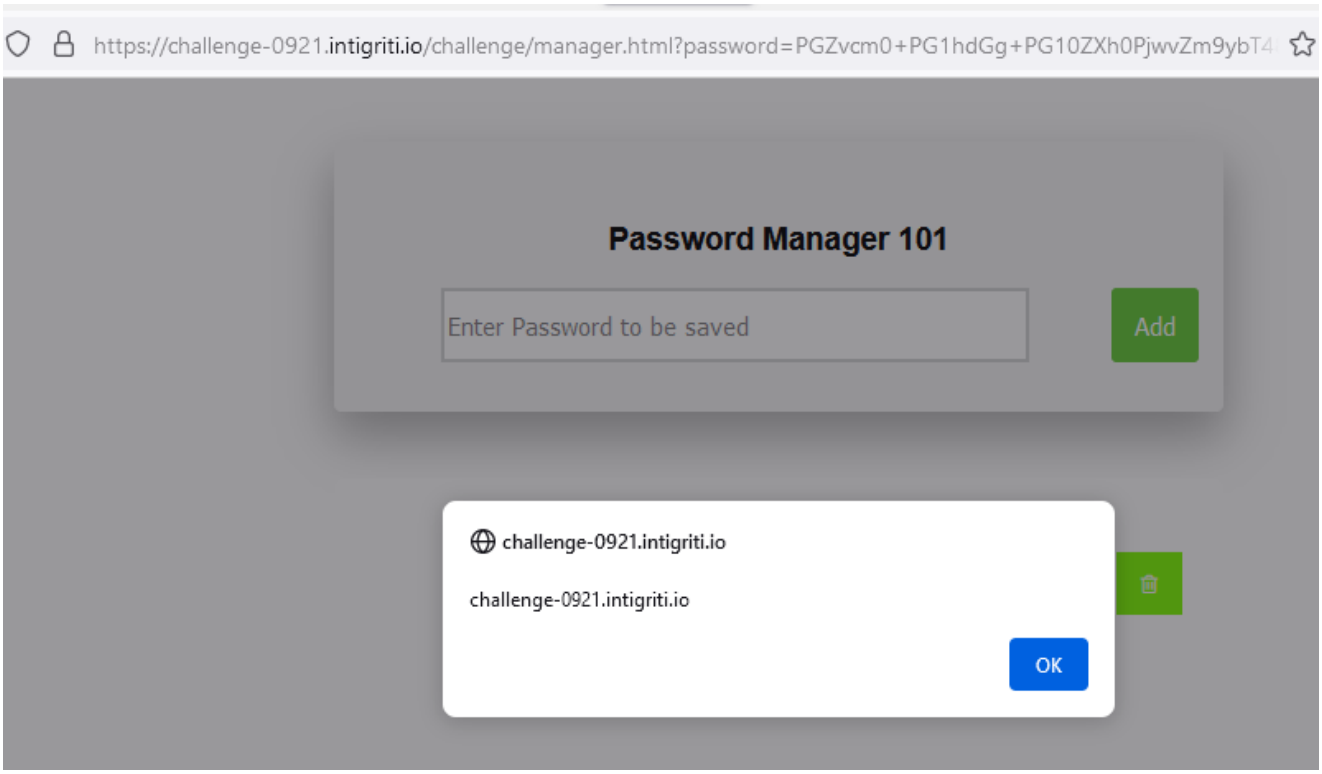
So in conclusion the following payload should give us XSS:

```
<form><math><mtext></form><form><mglyph><style></math><img src
onerror=alert(document.domain)>
```

As seen previously, we base64 encode this and send this to the password parameter.

```
PGZvcm0+PG1hdGg+PG10ZXh0PjwvZm9ybT48Zm9ybT48bWdseXBoPjxzdHlsZT48L21hdGg+PGltZy
BzcmMgb25lcnJvcj1hbGVydChkb2N1bWVudC5kb21haW4pPg==
```

Upon putting some random text in password manager box and clicking on "Add", we finally get our XSS.



PoC URL: `https://challenge-0921.intigriti.io/challenge/manager.html?password=PGZvcm0+PG1hdGg+PG10ZXh0PjwvZm9ybT48Zm9ybT48bWdseXBoPjxzdHlsZT48L21hdGg+PGltZyBzcmMgb25lcnJvcj1hbGVydChkb2N1bWVudC5kb21haW4pPg==`

References

https://stackoverflow.com/questions/10808109/script-tag-async-defer

https://flaviocopes.com/javascript-async-defer/

https://lelinhtinh.github.io/de4js/

https://snyk.io/vuln/npm:dompurify@2.0.8

https://research.securitum.com/mutation-xss-via-mathml-mutation-dompurify-2-0-17-bypass/