# Web & Browser Security

## Chapter Six: Sandboxing & Random Bits

Let's build our own JavaScript sandbox, shall we? And then stories.

A lecture by Dr.-Ing. Mario Heiderich
mario@cure53.de || mario.heiderich@rub.de

# Our Dear Lecturer

- **Dr.-Ing. Mario Heiderich**
  - **Ex-Researcher and now Lecturer, Ruhr-Uni Bochum**
    - PhD Thesis about Client Side Security and Defense
  - **Founder & Director of Cure53**
    - Pentest- & Security-Firm located in Berlin
    - Security, Consulting, Workshops, Trainings
    - Ask for an internship if the force is strong with you
  - **Published Author and Speaker**
    - Specialized on HTML5, DOM and SVG Security
    - JavaScript, XSS and Client Side Attacks
  - **Maintains DOMPurify**
    - A top notch JS-only Sanitizer, also, couple of other projects
  - **Can be contacted but prefers not to be**
    - mario@cure53.de
    - mario.heiderich@rub.de

## Act One

## Sandboxing

## Why?

- Imagine the following situation
  - A web application maintainer allows their users to submit and execute JavaScript
  - Not “allows” in terms of XSS and JavaScript injection, they actually want user-controlled script
  - This will then execute in the browser, for other people. Or even on the server.
  - The script is supposed to be as powerful as possible
- Why would anyone ever do that?
- **The answer is: Why not!**

## User Controlled Code 1/2

- Not a new concept, has been around for ages
- Usually for testing and debugging
  - https://ideone.com
  - http://phptester.net
  - https://jsfiddle.com
  - http://phpfiddle.org
  - And so on
- Here, the website maintainer rarely cares about continuity
  - You run a script
  - You check the outputs
  - Afterwards things get destroyed. Or not.
  - These are not websites that hold valuable data
- Often there is little to no protection for the server

## User Controlled Code 2/2

- Sometimes however, it’s about more than debugging or “just running” code
- Some web applications allow their users to create their own web applications as a service
- Like, for example, the Salesforce Lightning Component Framework, SLCF
  - Run your own web applications on top of their web application
  - Your own HTML, CSS and JavaScript.
  - Everything, even SVG.
- How do you do that?

cure53

## Try and try again

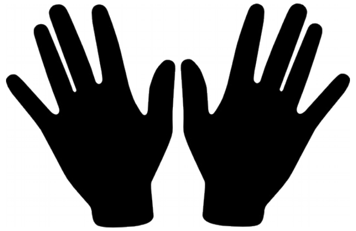
- Basically you need to build a sandbox
  - Find a way to execute that user code without being able to cause harm
  - But still able to support enough features
- This is not easy, especially if your web application is more than just a  test tool
  - So, let's do it, let's build our own sandbox

## JavaScript Sandboxing

- What is our sandbox supposed to offer?
    - Execute arbitrary JavaScript code
    - User the browser for it, no weird re-writer, expression language or alike
    - Offer as many JavaScript & DOM features as possible
- What is our sandbox supposed to prevent?
    - Have the JavaScript steal sensitive data
    - Have the JavaScript with the user's perception
    - Have the JavaScript cause any harm, essentially

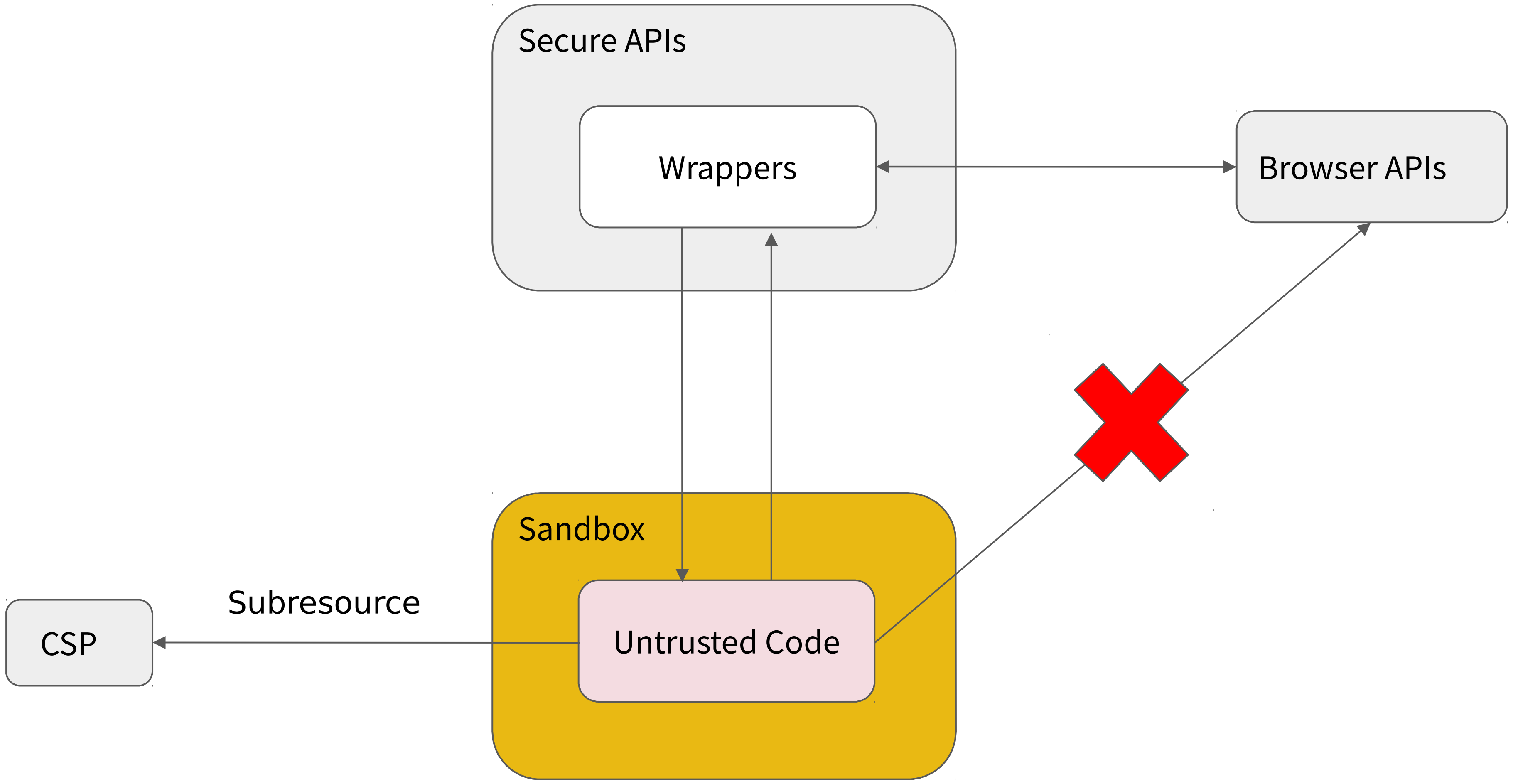
## Questions

1) Why do people need JavaScript sandboxes?
2) What challenges might they face?
3) Who might be to blame for those?



## Well, let’s do it!

## How it should look like



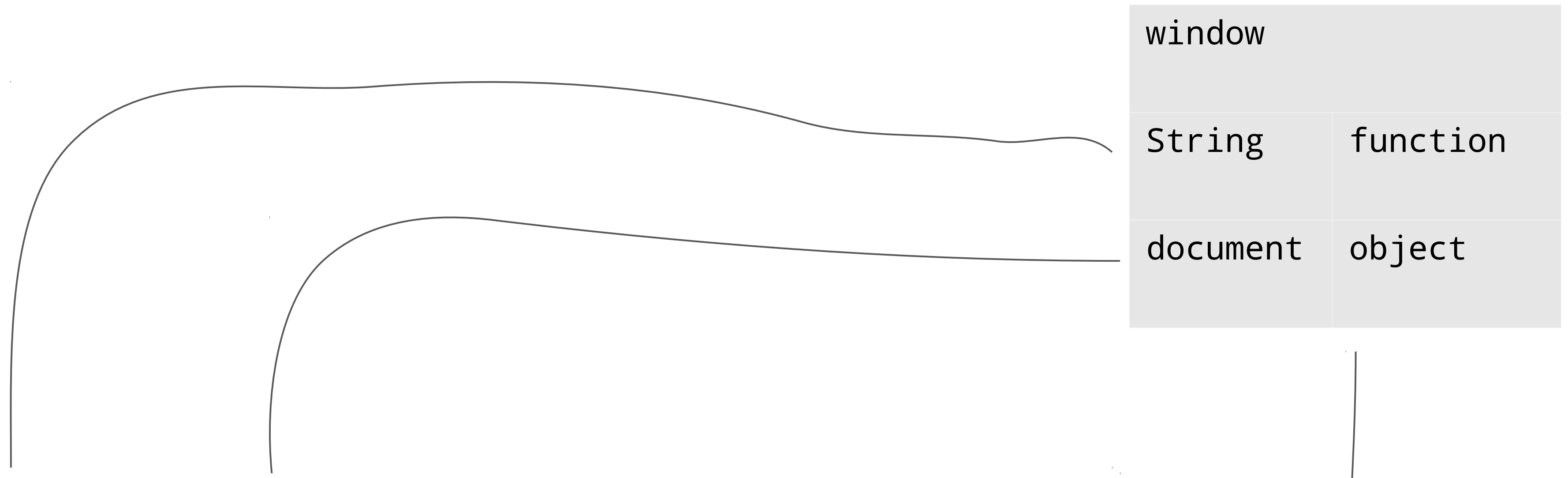
## Implementation

- **Basic Idea: Start with a minimal JavaScript environment**
  - No browser APIs can be accessible
  - No eval or alike can be offered
  - JavaScript to be executed in Strict Mode only
- **Only pass safe objects into the sandbox**
  - Objects from browser APIs must be wrapped into secure objects

## Strategic Implementation

- **Being able to reference <u>any</u> browser object directly is critical**
    - Any reference to `window` can do anything
    - Getting window from `document` is easy
        - `document.defaultView`
    - Getting `document` from DOM nodes (elements, attributes):
        - `node.ownerDocument`
    - Getting `window` from DOM events
        - `event.view`
- **No inline script, event handler or script import**
    - JavaScript in those places run outside of the sandbox
    - CSP controls this behavior in the browser level

# Variable Access



| window | |
|---|---|
| String | function |
| document | object |

```
> String(document);
< [object HTMLDocument]
```

This is a scope object (global)

## Disabling native API access

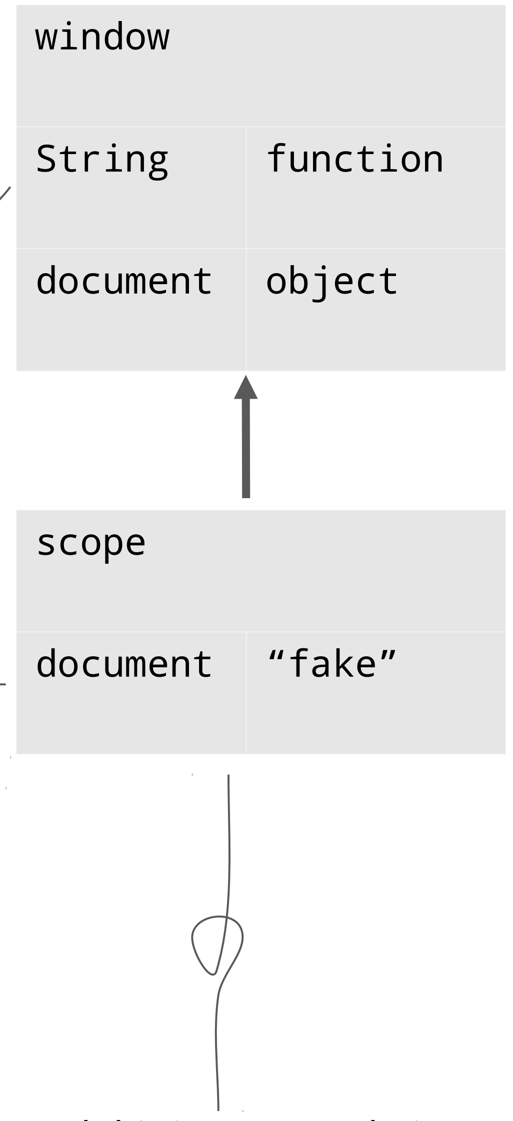
- **Accessing variables is the same as accessing properties from the scope object**
  - By default, the scope object is `window`
- **There is no browser API to make a minimal JS scope**
  - Deleting all dangerous properties doesn't work either
  - Why is that? 🙌
  - Try executing the following code in developer tools
    - `delete window.window`

## Disabling native API access

- **Solution: Shadow all dangerous global variables using `with`**
    - Enumerates all global variables, then shadows them with another scope object with higher precedence
- **Alternative:**
    - Use normal variables for shadowing
- **Danger:**
    - This is, at some level, a block-list!
    - Sounds terrible, is terrible but all we have at hands.
- **All untrusted JS must be wrapped in scoping wrapper**

## Why not use the `with` scope

```
var scope = {
    document: "fake"
};

> with(scope) String(document);
< fake
```



...and this is a scope chain

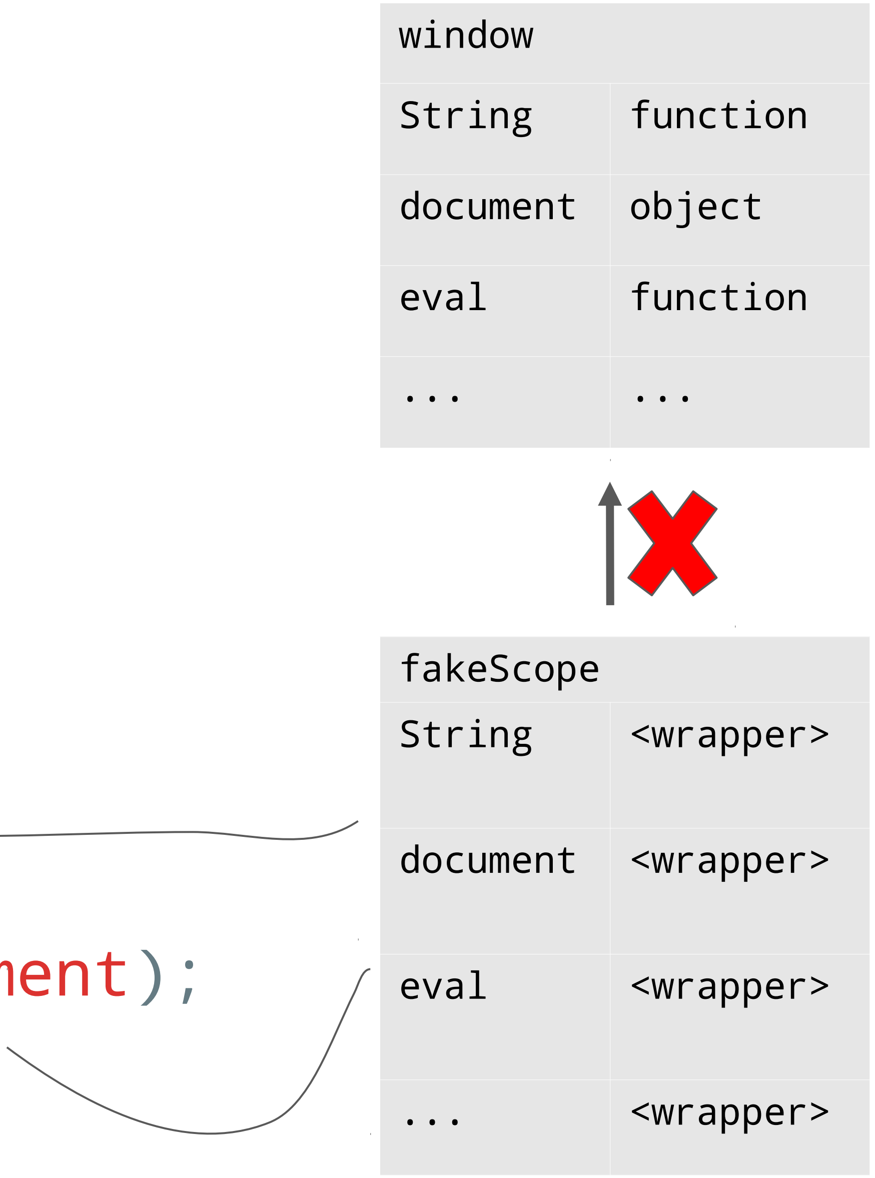
## Let’s talk about the Scope Chain

- JavaScript defines **3 different types of scope**
  - The global scope
  - The with scope
  - The function scope
- Embedded code accesses variables by traversing **upwards** in the scope chain
  - If the requested variable is found, traversal will stop there
  - If not, the journey continues
- We can therefore shadow variables access, based on this principal
- Instead of deleting existing APIs, we **shadow** them with proxied ones

# Shadowing access to global scope

```
var fakeScope = {
   String: <wrapper>,
   document: <wrapper>,
   eval: <wrapper>
   …: <wrapper>
};
```

```
with(fakeScope) String(document);
```

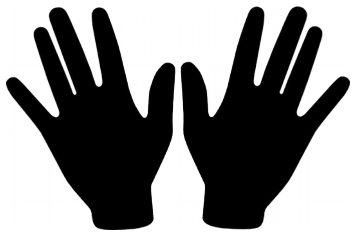


| window | |
|---|---|
| String | function |
| document | object |
| eval | function |
| ... | ... |

| fakeScope | |
|---|---|
| String | <wrapper> |
| document | <wrapper> |
| eval | <wrapper> |
| ... | <wrapper> |

## Questions

1) How do we best disable native API access?
2) What are the three relevant scopes in JS?
3) What is shadowing, why is it problematic?



## Hold on, a challenge appears!

- **https://sf.prompt.ml/sandbox/{level}**

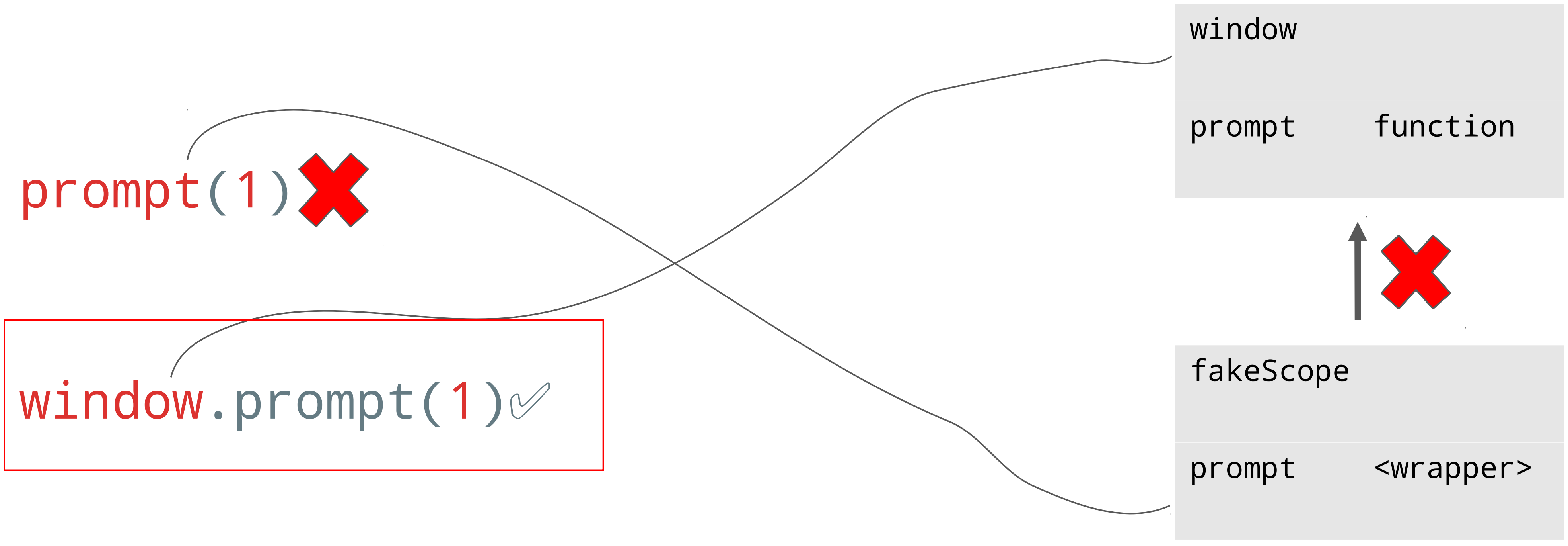
- Goal is always: execute JS code `prompt(1)`
- Google Chrome and developer tools are recommended
- You may have a solution that works across multiple levels (remember please, some levels are for beginners)
- While you could cheat the system by modifying the request, please refrain from doing that
- If you already **have** a solution, try aiming for a **shorter** one
- Refresh to see your score after completing a challenge

## Q: Is this code already safe enough?

```
var fakeScope = {
   String: <wrapper>,
   document: <wrapper>,
   eval: <wrapper>
   …: <wrapper>
};
with(fakeScope) <untrusted expression>
```

🙌 (warm-up) Find out here! https://sf.prompt.ml/sandbox/0

## Solution: Getting access to Window



**Recap**: All browser APIs reside in the `window` object.
This is the stuff we want to access.

## Now, enough for the warm-up...

- 🙌 https://sf.prompt.ml/sandbox/1
- **Stick to the original plan:**
  - to shadow **all** variables
- **Note:**
  - there are multiple solutions to this one
- **Hint:**
  - this is not much more difficult!

## Issue #1: `this` returns the global object

```
var fakeWindow = {
  this: <wrapper>
};

> with(fakeScope) String(this)
< [object Window]
```

**11.1.1 The this Keyword # Ⓣ Ⓡ Ⓖ**

The **this keyword** evaluates to the value of the ThisBinding of the current execution context.

```
this.prompt(1)✓
```

A keyword is **not** a variable!

## Issue #2: Namespace pollution

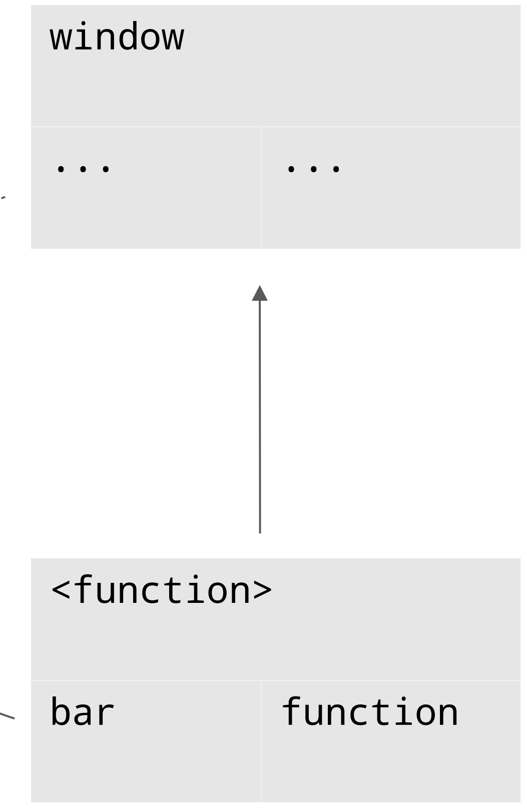
```
with (fakeScope) {
  function onclick() {}
}

> window.onclick
< fn onclick
```

`with` scope has no effect on function declaration.
**Note:** Does not work in the challenge as **{** is filtered
But it might have in real life!

## Function scope to the rescue

```
(function() {
   function bar() {};
   this; // [object Number]
}).call(1)
```

```
> bar
< undefined
```



## Function Scope

- Function scope provides two extra functionalities compared to the with scope
  - Has effect on function declaration
  - Controls the this value even if it is used outside of a class
- Technically, we can just use `Function` to manipulate the scope chain…
  - Has no programmatic way to write many variables into the scope object
  - As opposed to `with(fakeScope) { code }`
  - It needs to be `(function(var1, var2, …) { code })()`
- Best is to use a hybrid method that combines the two

## The more you know!

“I think that looks goofy, 'cause what we're talking about is the whole invocation, but we got these things hanging outside of it looking sorta like ... dog balls.”

D. Crockford, http://www.youtube.com/watch?v=taaEzHI9xyY

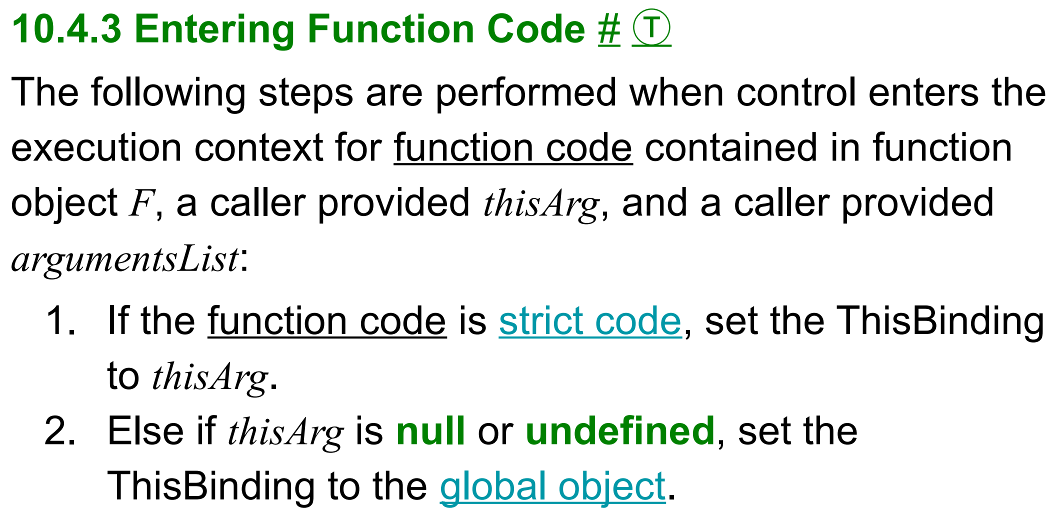
## Q: Is this Code safe?

```
with(fakeScope) (function() {
    <untrusted expression>
}).call(fakeScope)
```

☕ No challenge for this one, but feel free to discuss 😃

## Issue #1: this Coercion

```
with(fakeScope) (function() {
  (function(){ alert(this) })()
}).call(fakeScope)
// alerts [object Window]
```



**10.4.3 Entering Function Code # Ⓣ**

The following steps are performed when control enters the execution context for function code contained in function object *F*, a caller provided *thisArg*, and a caller provided *argumentsList*:

1. If the function code is strict code, set the ThisBinding to *thisArg*.
2. Else if *thisArg* is **null** or **undefined**, set the ThisBinding to the global object.

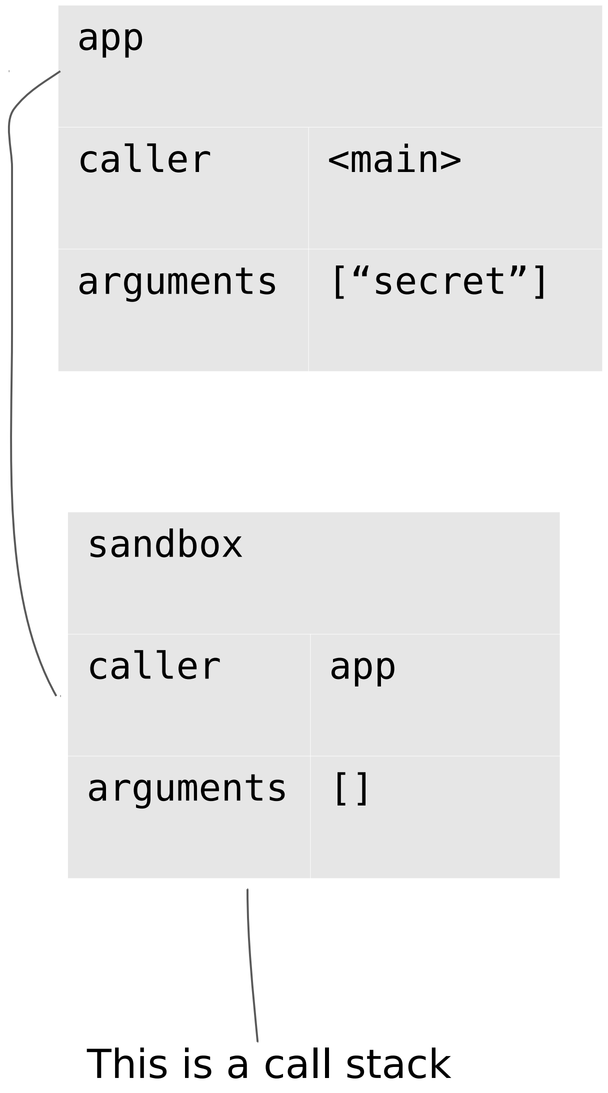
this leaks window in inner functions in non-strict mode

## Inner function scope

- Although we can control the `this` value in a function scope, It does have  no effect on an inner function scope
- This is because `this` is bound to the current scope
- The value changes on different scope
- By embedding an inner function we create a new scope, where the this value is not affected by the outer function scope
- In non-strict mode, it is... `window`

## Issue #1: caller traversal and arguments

```
function app(key) { locker(); }
function locker() {
   alert(locker.caller.arguments[0]);
}

app("secret");
// alerts secret
```



## Traversing the call stack

- Calling a function within another function call creates a call stack
- JavaScript allows us to access the call stack via `caller` and `callee` of a function
- `callee` refers to the current function being called
- `caller` refers to the function that calls the current function
- A function also has a special property, called `arguments`
  - "arguments is an Array-like object accessible inside functions that contains the values of the arguments passed to that function."
- In short, it returns the arguments of a function
- **Note**: directly interacting with the call stack is considered dangerous, thus this is **<u>forbidden in strict mode</u>**

# Sandbox bypasses by walking up the caller chain

```
this.fn.caller.caller.
caller.caller.caller.caller.arguments[1].view
```

Created in non-strict mode

## Issue #2: Yet another namespace pollution

```
with(fakeScope) (function() {
   foo = 1;
}).call(fakeScope)

> window.foo
< 1
```

## Defining variables without `var`

- This is yet another legacy JavaScript thingy
- We are supposed to define a variable with either prefix `var` or `let` (or `const` for constants)
- For compatibility reasons however, we can ignore that
- When we *just* assign a value to an undefined variable, it goes directly to the global scope (`window`)
- Although shadowing still works since we have the variables defined, it **pollutes the global scope**

# ES5 strict mode

- Both issues mentioned can be fixed by enabling strict mode for all sandbox code
- But what exactly is strict mode?
- ECMAScript 5's strict mode is a way to opt in to a **restricted variant** of JavaScript
  - `undefined/null thisArg` is no longer coerced (not default to `window` any longer)
  - `caller/callee/arguments` are no longer accessible as function properties
  - undeclared variable assignment is no longer allowed
  - ...and many more restrictions!
- Check them restrictions out in the MDN docs
  - https://is.gd/8cuqG6

## So… is this finally the perfect sandbox?

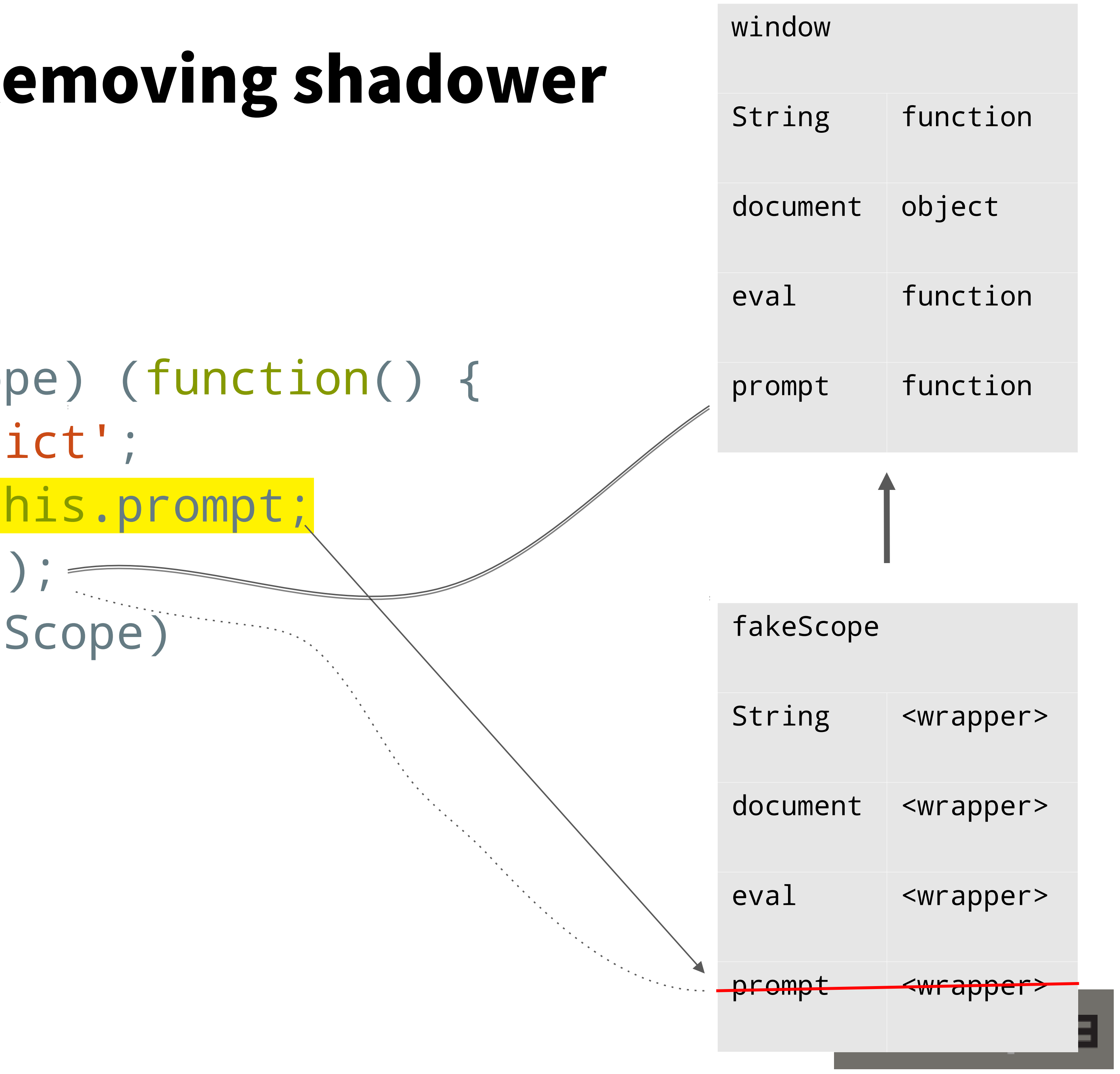
```
with(fakeScope) (function() {
   'use strict';
   <untrusted expression>
}).call(fakeScope)
```

Try it: https://sf.prompt.ml/sandbox/2

**Hint 1:** We can still interact with the fake scope object
**Hint 2:** Try to disable the property shadowing by de...

## Solution: Removing shadower

```
with(fakeScope) (function() {
    'use strict';
    delete this.prompt;
    prompt(1);
}).call(fakeScope)
```



| window | |
|---|---|
| String | function |
| document | object |
| eval | function |
| prompt | function |

| fakeScope | |
|---|---|
| String | <wrapper> |
| document | <wrapper> |
| eval | <wrapper> |
| ~~prompt~~ | ~~<wrapper>~~ |

# Controlling modification behavior

- We need to prevent the scope object from being modified
- JavaScript provides some functions for controlling the modification behavior

- `Object.defineProperty`
  - Defines getter/setter
  - Defines configurability

- `Object.freeze`
  - Prevents adding properties
  - Prevents redefining properties
  - Prevents deleting properties

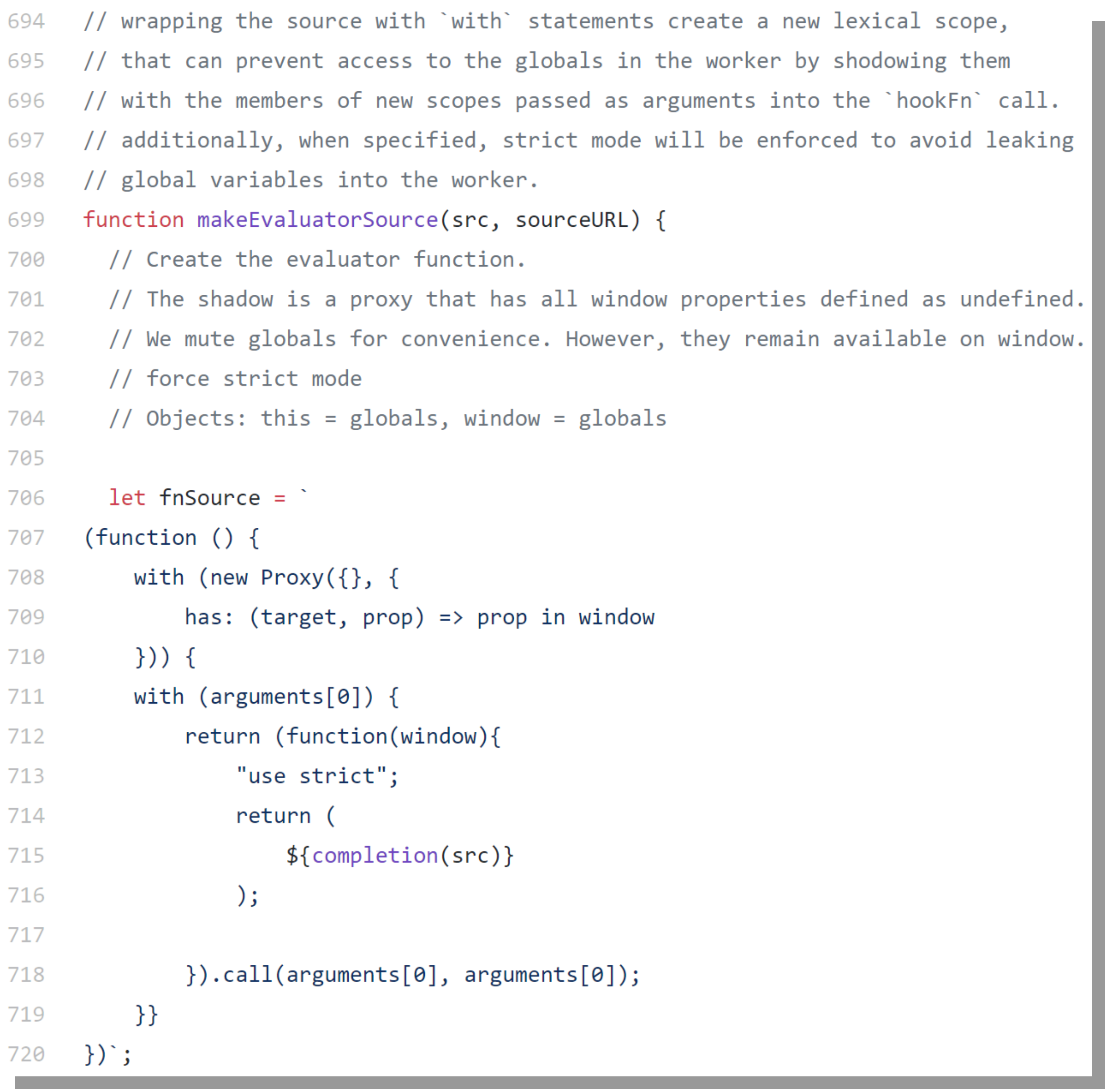
```
var fakeScope = Object.freeze({prompt: null})
> delete fakeScope.prompt
< false
```

🙌 Please try this out!

## Our Refined Sandbox

```
defineProperty(fakeScope, {...});
fakeScope = Object.freeze(fakeScope);
with(fakeScope) (function() {
  'use strict';
  <untrusted expression>
}).call(fakeScope)
```

# ...actually quite similar to i.e. Locker



```
// wrapping the source with `with` statements create a new lexical scope,
// that can prevent access to the globals in the worker by shodowing them
// with the members of new scopes passed as arguments into the `hookFn` call.
// additionally, when specified, strict mode will be enforced to avoid leaking
// global variables into the worker.
function makeEvaluatorSource(src, sourceURL) {
  // Create the evaluator function.
  // The shadow is a proxy that has all window properties defined as undefined.
  // We mute globals for convenience. However, they remain available on window.
  // force strict mode
  // Objects: this = globals, window = globals

  let fnSource = `
(function () {
    with (new Proxy({}, {
        has: (target, prop) => prop in window
    })) {
    with (arguments[0]) {
        return (function(window){
            "use strict";
            return (
                ${completion(src)}
            );

        }).call(arguments[0], arguments[0]);
    }}
})`;
```

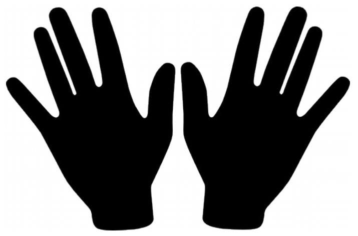
## Finally, a perfect sandbox!

- Well, not quite
  - In fact, it is not even close. It is just the tip of the iceberg
  - Next, we will get to an in-depth analysis of sandbox bypasses
  - Now, we may take a break and grab ☕

Fig.: Developer fixing sandbox bypasses

# Questions

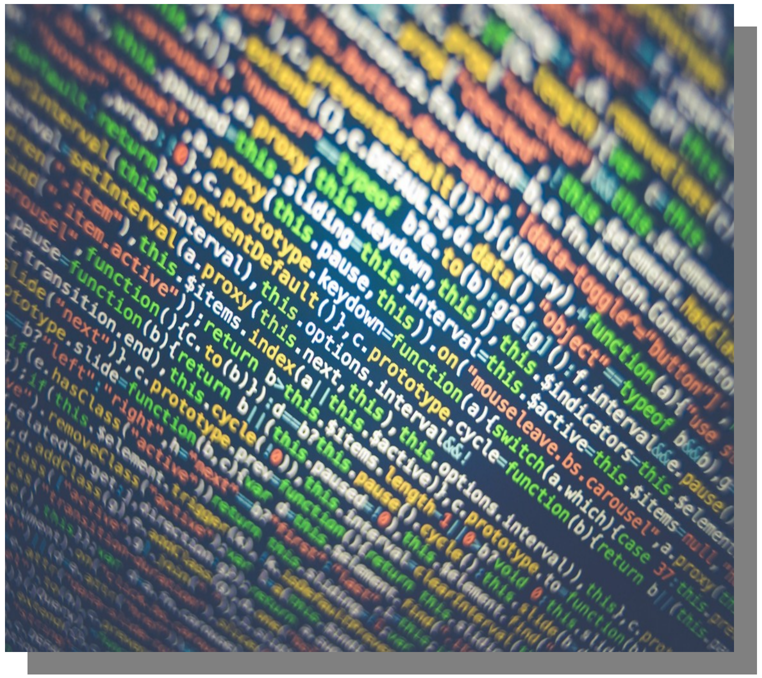
1) Why is JavaScript Sandboxing hard?
2) What does Strict-Mode do to help?
3) Is the 100% secure JavaScript sandbox possible?



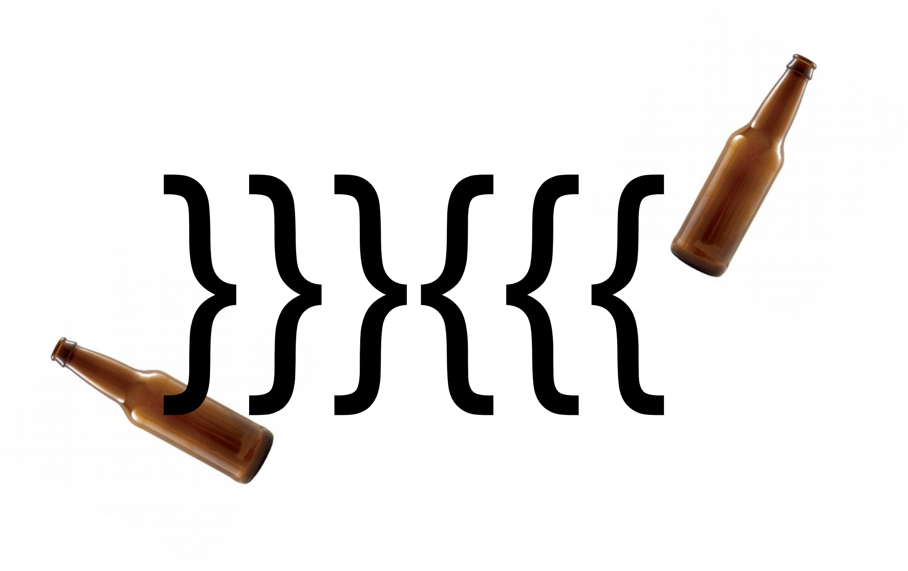
## More Challenges

- **Format Injections**
  - The user controlled code has to arrive somewhere
  - Usually inside curly braces around `<untrusted expression>`
- **HTML inadvertently leaking window**
  - SVG, MathML, XML, Flash doing the same
- **New JavaScript features that are not yet blocked**

# WHO WOULD WIN?



A sophisticated JavaScript sandbox with 10k+ LoC



Some curly bois

## Format Injection

```
with(fakeScope) (function() {
    'use strict';
    <custom expression>
}).call(fakeScope)
```

## Format Injection

```
with(fakeScope) (function() {
   'use strict';
   });alert(window);({
}).call(fakeScope)
```

## The weakest link

- Even if a sandbox itself is fairly strong
- Maybe the code that runs the untrusted code in the sandbox is not
- i.e. the rewriting logic of the untrusted code
- Why do we not just break the code that runs the sandbox?

**Q: What to do against that?**

## String-to-code and `unsafe-eval`

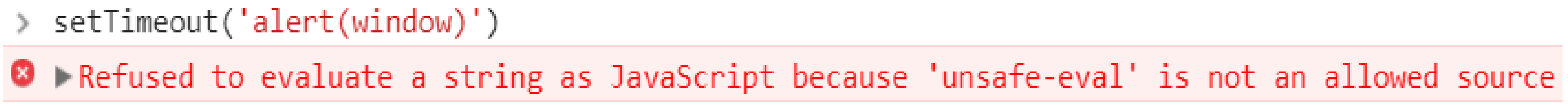
## CSP’s unsafe-eval

The following JavaScript execution sinks are gated on the "unsafe-eval" source expression:

- `eval()`
- `Function()`
- `setTimeout()` with an initial argument which is not callable.
- `setInterval()` with an initial argument which is not callable.

Note: If any User Agent implements non-standard sinks like `setImmediate()` or `execScript()`, they **SHOULD** also be gated on "unsafe-eval".

## Without unsafe-eval



```
> setTimeout('alert(window)')
✖ ▶Refused to evaluate a string as JavaScript because 'unsafe-eval' is not an allowed source
```

## Why allow something with the name unsafe?

- Some frameworks use `eval`
- Sandboxes might have `unsafe-eval` disabled
- However, because untrusted code is now put in `Function` to validate the syntax...
  - It is enabled to prevent sandbox breakout despite of the risk
  - Classic trade-off it seems
- Instead, now the editor checks if the code contains the use of `eval` related functions

## And why is it dangerous?

- We can use it to execute any code outside the sandbox
- The code is basically executed on the **global scope/context**
- So any scope shadowing will not have any effect on it

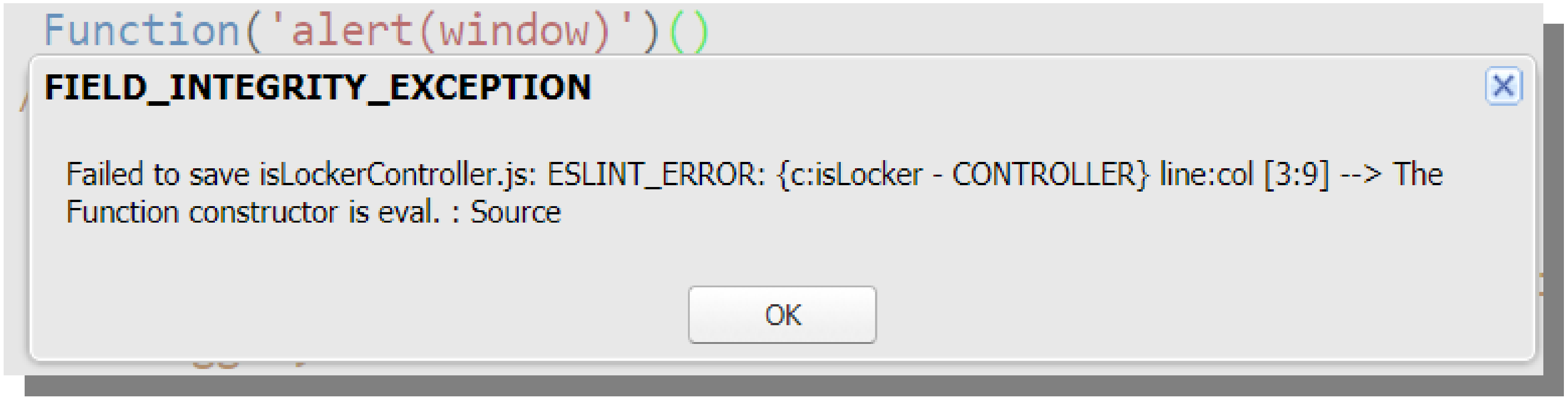
```
Function('alert(window)')
```

=

```
function() {
   alert(window);
}
```

cure53

## Computer says No

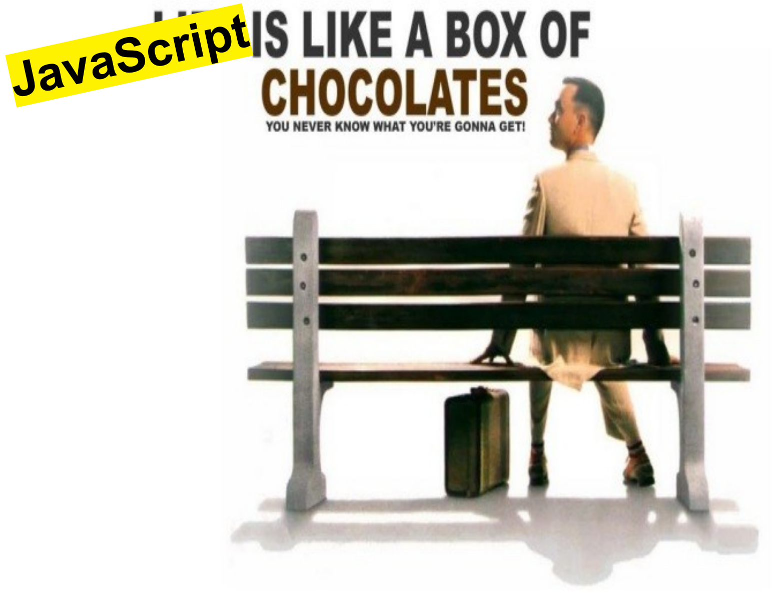


```
Function('alert(window)')()
```

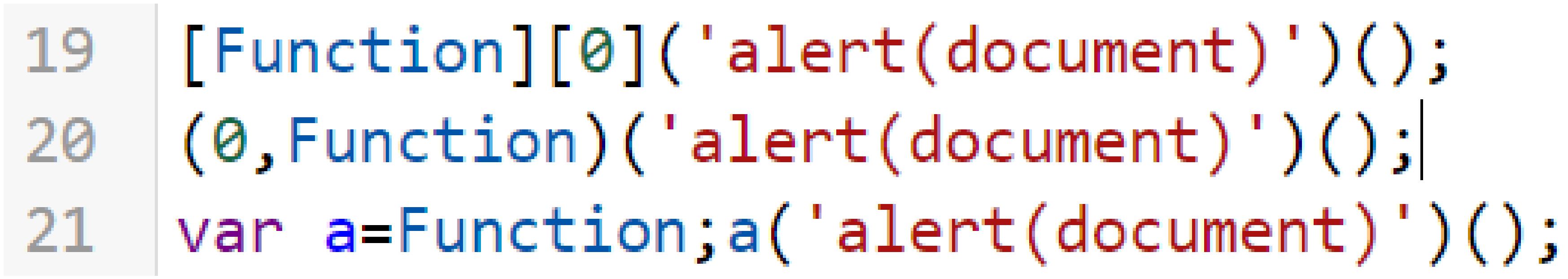
Failed to save isLockerController.js: ESLINT_ERROR: {c:isLocker - CONTROLLER} line:col [3:9] --> The Function constructor is eval. : Source

## What could go wrong?

- We have already known that sandbox parsers can be problematic
- And JavaScript is a **highly** dynamic language
- You never know what you are gonna get
- There are many ways to reference a function
- Static analysis will not help either



## Computer says yes



```
[Function][0]('alert(document)')();
(0,Function)('alert(document)')();
var a=Function;a('alert(document)')();
```

- First one uses array access to get to Function
- Second one uses the comma operator to get to Function
- Third one uses variable to get to Function

# How many ways can you get to Function?

- 🙌 Try it: https://sf.prompt.ml/sandbox/4
- The word "Function" is now filtered
- There are many, many ways to do it
- Below is an example that uses property access to get to Function
- You should come up with more ways of doing so!

```
fakeScope['Functi'+'on']('prompt(1)')()
```

IT'S OVER
9000!!!!

## Just to list a few

```
Fun\u0063tion('prompt(1)')() // Unicode Identifier

this.constructor.constructor('prompt(1)')() // Prototype chain
```

## How to fix?

We could just avoid exposing Function

```
Fun\u0063tion('prompt(1)')() // Unicode Identifier

this.constructor.constructor('prompt(1)')() // Prototype chain
```

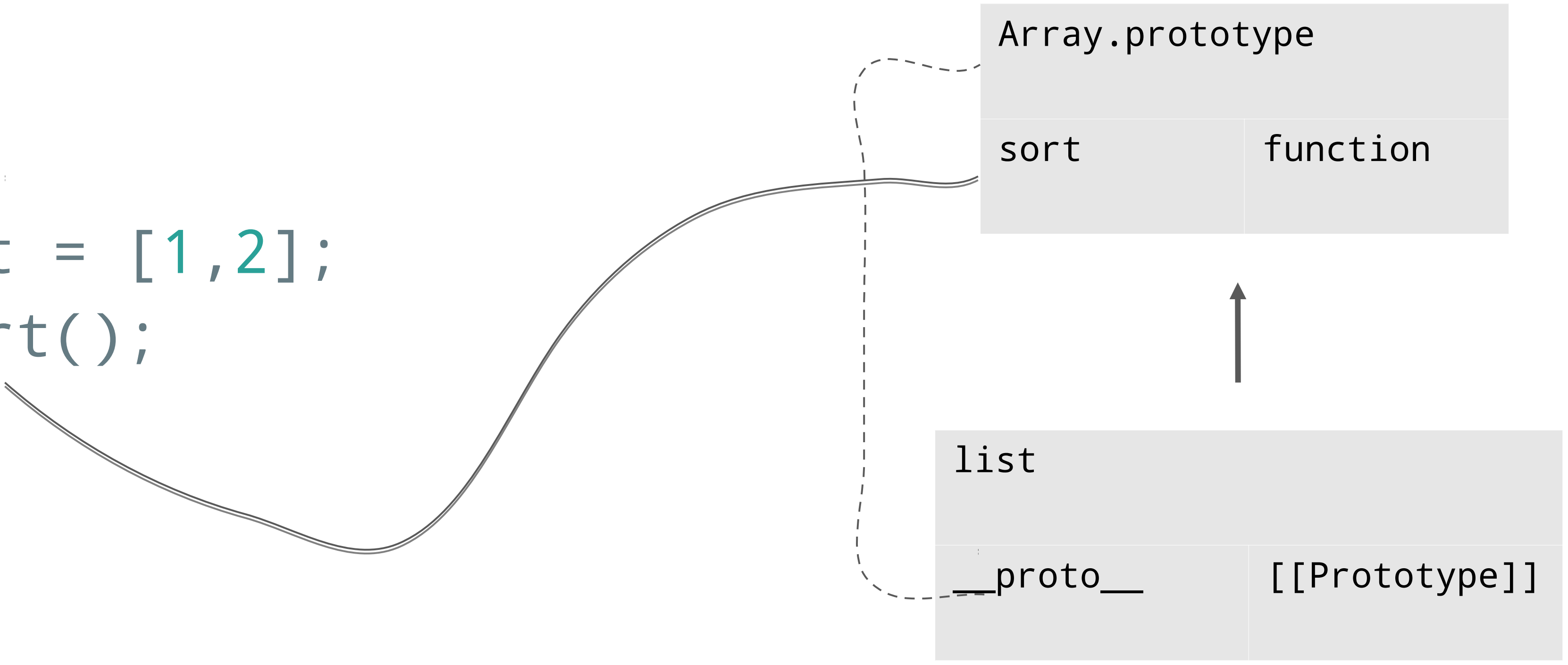
And now what ???

## Prototype Chain and `constructor`

- Objects in JavaScript are inherited using prototype chain
- The constructor that creates the object can be retrieved using **`.constructor`**
- A constructor has a **`.prototype`** property
- When accessing a property of an object where the property **does not exist**, it will try to look up the prototype chain
  - Almost as the with scope thingy traversing upwards
- The chain can be accessed via the **`.__proto__`** property

## Prototype Chain and constructor

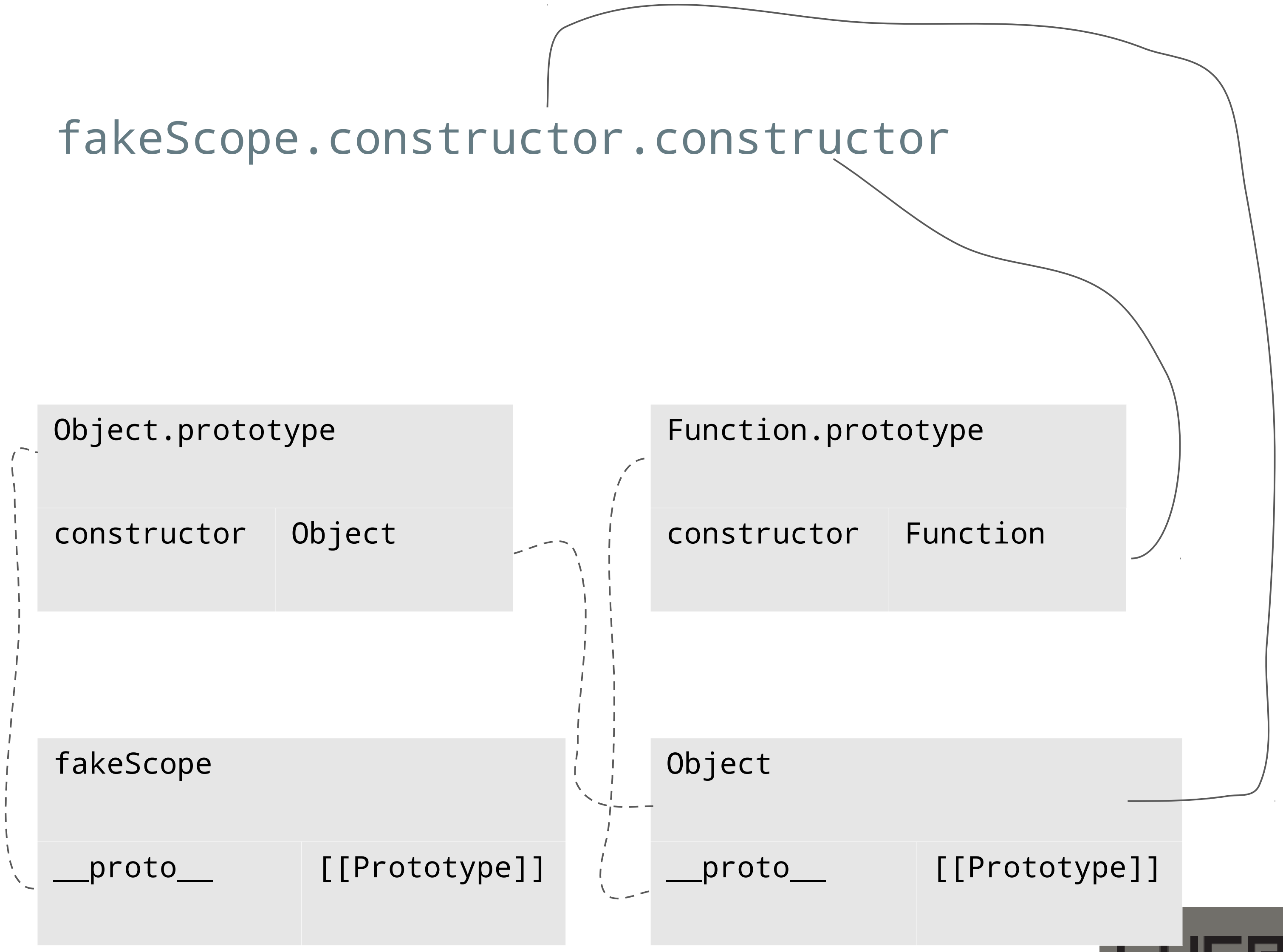
```
var list = [1,2];
list.sort();
```



## Prototype Chain and `constructor`

- In JavaScript, every `constructor` is a function
- And every object has a `constructor`
- By traversing to constructor, we can eventually get to `Function`
- This may sound a bit tongue-in-cheek so let us look at the diagram...

# Prototype Chain and `constructor`



## Prototype Chain and constructor (Cont.)

- `fakeScope` is an object
- `fakeScope.constructor` refers to the object constructor, which is `Object`
- `fakeScope.constructor.constructor == Object.constructor`
- `Object.constructor == Object.constructor`
- Voila, we now have a reference to `Function`!

## Classic Sandbox Bypasses

```
[].constructor.constructor('alert(document)')();
```

```
(x=>x).constructor('alert(document)')();
```

## How were they fixed?

- In order to traverse to `Function`, `Function.prototype.constructor` must be accessed
- Can we create a trap on the .constructor property on `Function.prototype`?
- If it is accessed in a userland context, it returns an empty function instead of `Function`
- Something like the following:

```
Object.defineProperty(Function.prototype, 'constructor',
    {get: function() {}})
```

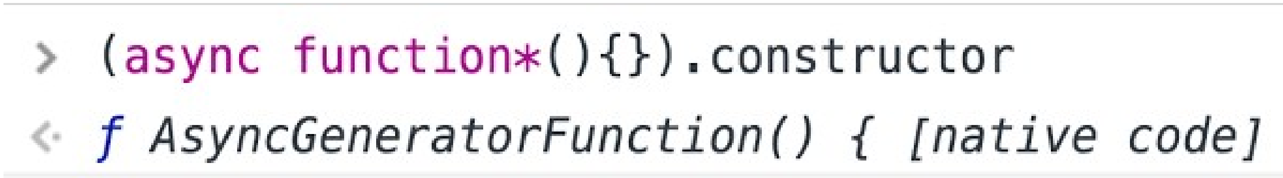
## In fact, that's kinda how AngularJS did it!

```
function ensureSafeObject(obj, fullExpression) {
  // nifty check if obj is Function that is fast and works across iframes and other contexts
  if (obj) {
    if (obj.constructor === obj) {
      throw $parseMinErr('isecfn',
          'Referencing Function in Angular expressions is disallowed! Expression: {0}',
          fullExpression);
```

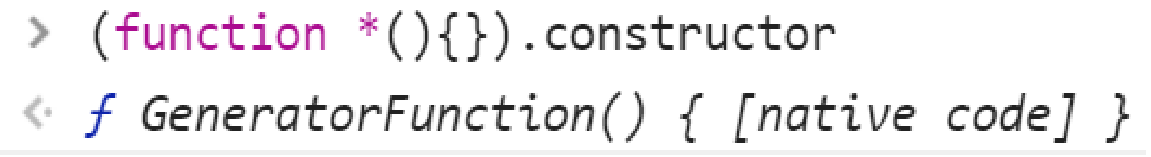
```
{{constructor.constructor('alert(1)')()}}
```

# The ever-growing Function family

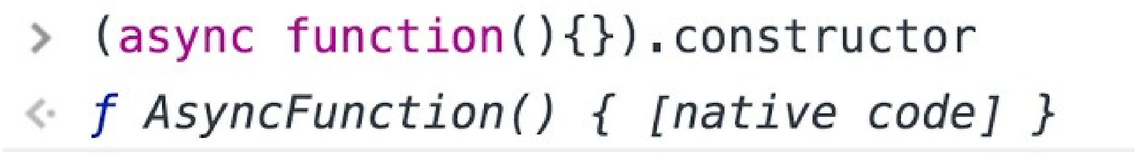
- In ES5, there was the one and only `Function`
- But ES6/2017 introduce some variations of `Function`!
  - `GeneratorFunction`
  - `AsyncFunction`
  - `AsyncGeneratorFunction`
- And their constructors?
- Those need to be blocked as well!



```
> (async function*(){}).constructor
< f AsyncGeneratorFunction() { [native code]
```

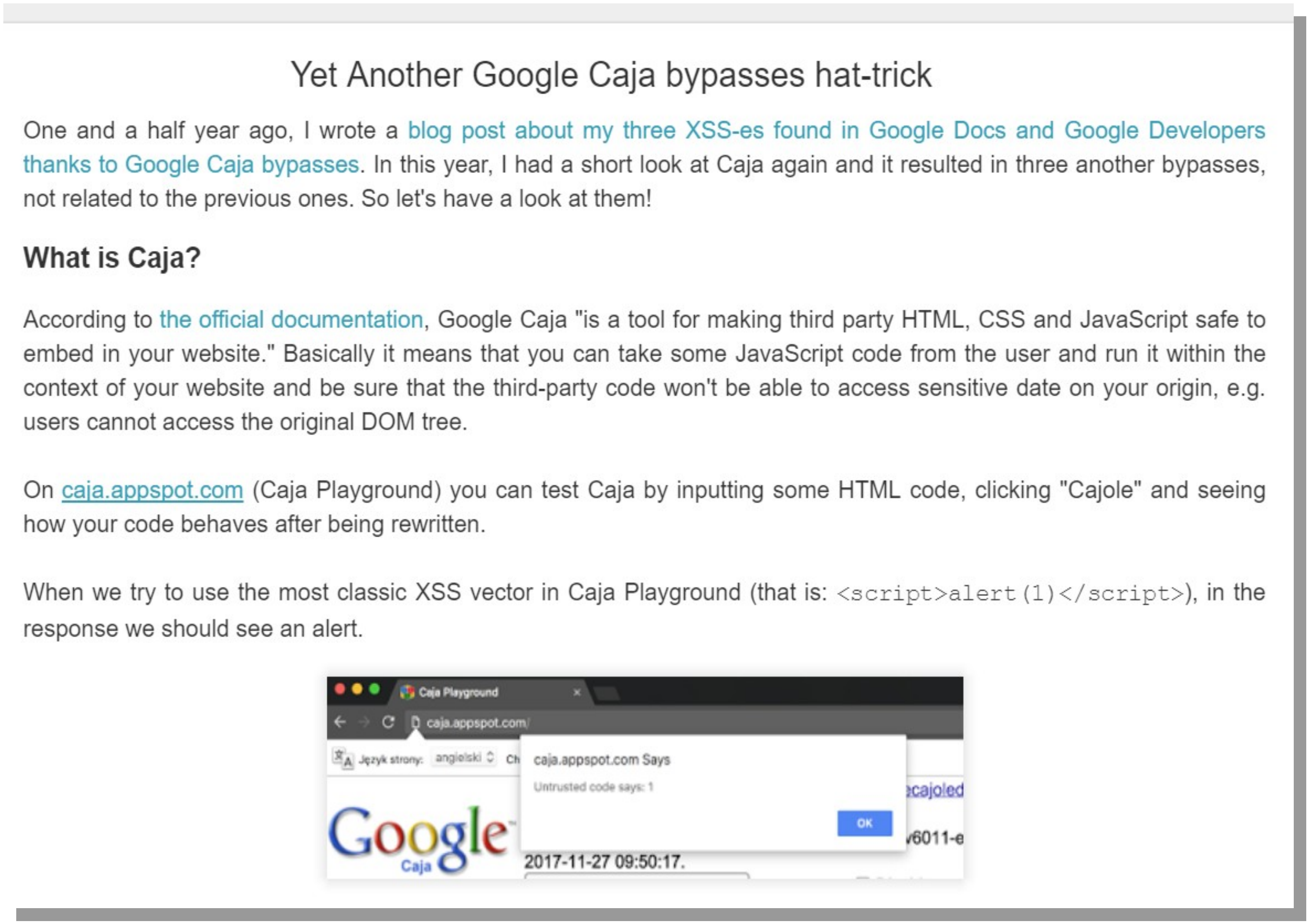


```
> (function *(){}).constructor
< f GeneratorFunction() { [native code] }
```



```
> (async function(){}).constructor
< f AsyncFunction() { [native code] }
```

# Google Caja was vulnerable too



## Yet Another Google Caja bypasses hat-trick

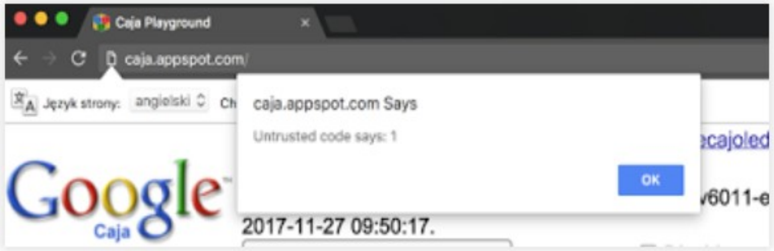
One and a half year ago, I wrote a blog post about my three XSS-es found in Google Docs and Google Developers thanks to Google Caja bypasses. In this year, I had a short look at Caja again and it resulted in three another bypasses, not related to the previous ones. So let's have a look at them!

### What is Caja?

According to the official documentation, Google Caja "is a tool for making third party HTML, CSS and JavaScript safe to embed in your website." Basically it means that you can take some JavaScript code from the user and run it within the context of your website and be sure that the third-party code won't be able to access sensitive date on your origin, e.g. users cannot access the original DOM tree.

On caja.appspot.com (Caja Playground) you can test Caja by inputting some HTML code, clicking "Cajole" and seeing how your code behaves after being rewritten.

When we try to use the most classic XSS vector in Caja Playground (that is: `<script>alert(1)</script>`), in the response we should see an alert.



caja.appspot.com Says

Untrusted code says: 1

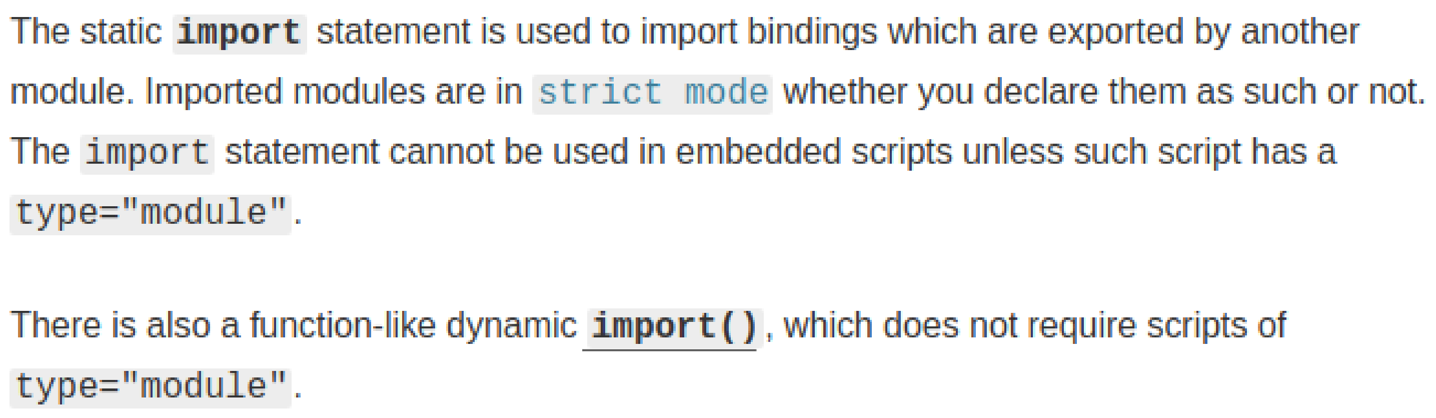
OK

2017-11-27 09:50:17.

## Which fix approach to pick?

- **Block access to Function and all its buddies**
  - Hide `Function` and trap `Function.prototype.constructor`
  - Can also trap other types of function constructor
  - However, this is a block-list approach which is **kind of** risky
  - If a new function is added in ECMAScript, then the list needs to update. Happened to AngularJS.
- **Try to avoid `unsafe-eval`**
  - String-to-code is taken care of on the browser level
  - Hence almost not bypassable (and not your fault if so)
  - However the untrusted code needs to be validated strictly since we cannot use `Function`

## Another problem - dynamic imports

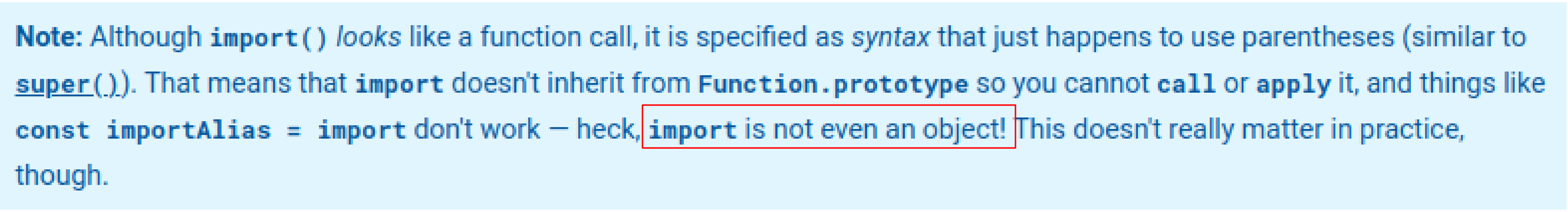


The static `import` statement is used to import bindings which are exported by another module. Imported modules are in `strict mode` whether you declare them as such or not. The `import` statement cannot be used in embedded scripts unless such script has a `type="module"`.

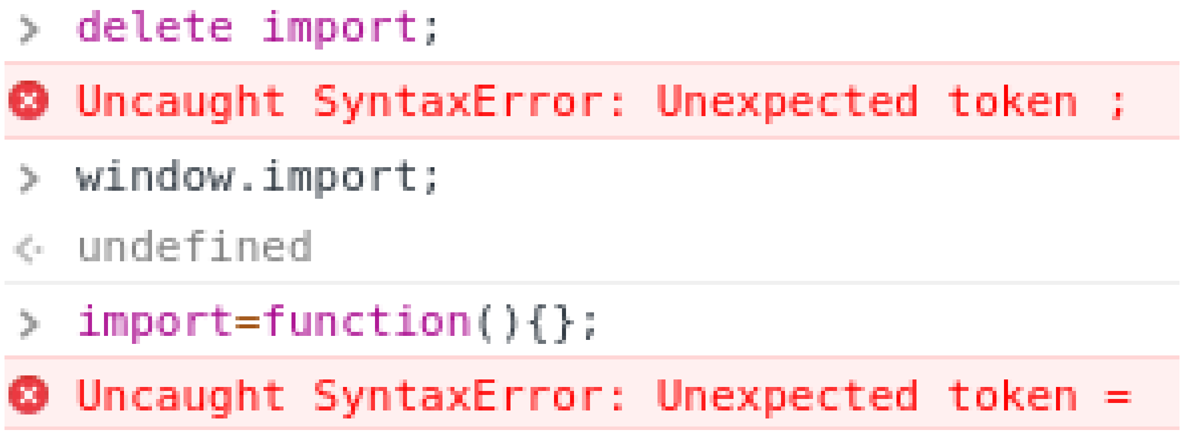
There is also a function-like dynamic `import()`, which does not require scripts of `type="module"`.

```
import("/module.js");
//Google Chrome
import("data:text/javascript,alert(window)")
```

## So what is the problem?


**Note:** Although `import()` *looks* like a function call, it is specified as *syntax* that just happens to use parentheses (similar to `super()`). That means that `import` doesn't inherit from `Function.prototype` so you cannot `call` or `apply` it, and things like `const importAlias = import` don't work — heck, `import` is not even an object! This doesn't really matter in practice, though.

### Think of it as something like `typeof()`


```
> delete import;
⊗ Uncaught SyntaxError: Unexpected token ;
> window.import;
<· undefined
> import=function(){};
⊗ Uncaught SyntaxError: Unexpected token =
```

## Window access via `import`

```
eval("import(`${location.origin}/resource/xss`)");
```

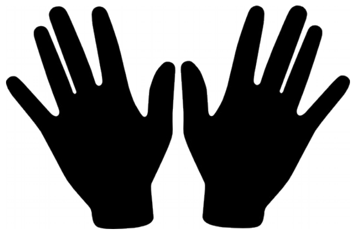
- `import()` cannot be removed or wrapped as it is considered **JS syntax**
- CSP `script-src` controls protocols/domains of import - `unsafe-eval` has no effect.
- Only solution: Filter the string for an import statement before passing it to eval.
  - Horrible approach!
  - False positives/false negatives
  - Obfuscation: JavaScript Strings, New Lines and more...

## Questions

1)Is a working JavaScript sandbox possible?

2)Is the browser helping us with that?

3)What other challenges come to mind?



## Act Two

## Random Bits

## Real World stories 1/4

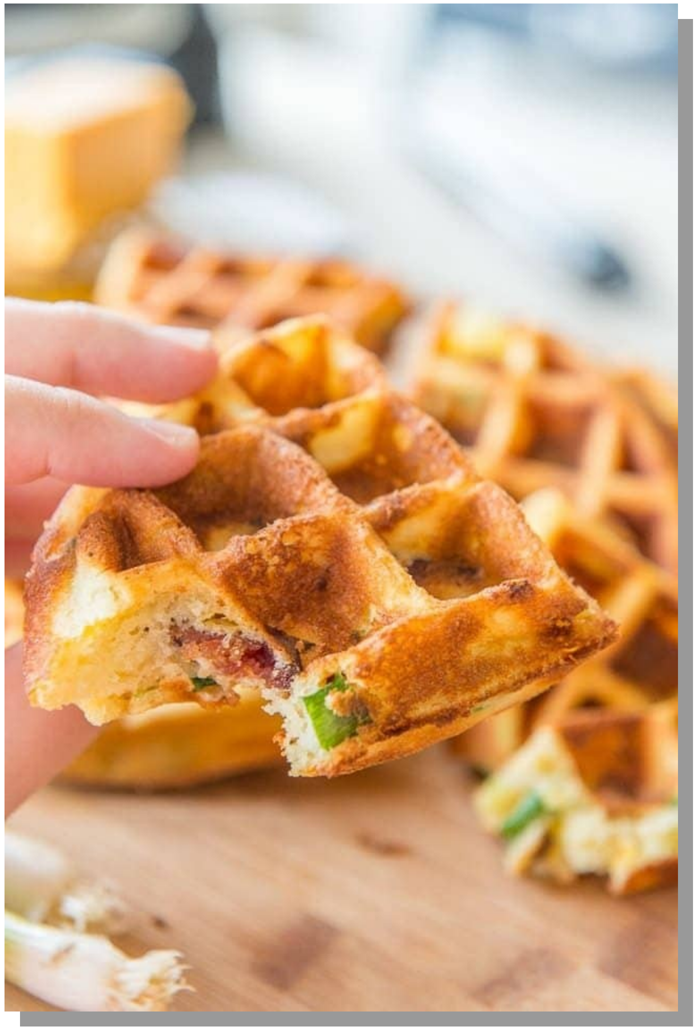
- The Japanese Cookie
  - HTTP Errors
  - Cookie Stuffing
  - XSS via XSS Filter

## Real World stories 2/4

- “Not code, just Python!”
  - 3rd party Software
  - Surprising features
  - Horrendous design

## Real World stories 3/4

- Bacon Waffles
  - eCommerce scenario
  - Complex relations & 3rd parties
  - Client-side pricing attacks



## Real World Stories 4/4

- “We trust our users”
  - Financial enterprise
  - Wants to give users a feedback channel
  - Comes up with a terrible idea

## Act Three

Humans

# Remember the Failure Slide?

- This is not the only stressful aspect of working in this field
- Not only technical skills are needed to master all this, soft skills needed too!
  - Stay organized
  - Don't forget anything, ever
  - Organize your knowledge
  - Keep everything in writing
  - Build circles of trust
  - Do not trust anyone outside the inner circles
  - Get good at reading legal language
- And most importantly grow a thick skin

# Thick Skin

- So, this is a feat that will help you in any field – but particularly in info-sec
  - Learn to separate work from private life
  - Avoid being emotional
  - Don’t shit-post on Twitter
  - Don’t go for revengeful crusades
  - Know your legal boundaries
  - Keep your ego under control
  - Reflect and observe
  - Stay healthy and eat your vitamins!
- Being in this field will not be easy and it’s often less cool than it looks like
- Be ready for that and work on **both** your skills and your stability
  - All those cool skills become **utterly useless** if you are a nerve-wreck destroying your own career in public cake-fights

## So, that was it

- Feedback is welcome
- Let me know what needs to get better
- And now get ready for the final phase...

## Next up, Exams

- You best preparation is to work with the slides
  - Look at the material, follow the links, understand the topics we talked about
  - The majority of questions will be similar yet not identical to the ones we discussed here
  - Topics we did not cover in this seminar will not be part of the exams
- Difficulty levels are expected to be moderate
  - The exams aim to evaluate how much of the course you understood
  - You might have to take knowledge from one chapter and combine it with knowledge from another
- **Good luck, everyone!**

## Final Chapter: Done

- Thanks a lot!
- Exams soon.
- Any questions? Ping me.
  - mario@cure53.de