KU LEUVEN

DistriNet

Designing “least-authority” JavaScript apps

Tom Van Cutsem
DistriNet KU Leuven

tvcutsem.github.io be.linkedin.com/in/tomvc github.com/tvcutsem x.com/tvcutsem @tvcutsem@techhub.social

Web application security

same-origin policy

certificate pinning

OAuth

cookies

content security policy

CSRF

html sanitization

HSTS

OWASP

Open Web Application
Security Project

A **software engineering view** of Web application security

~~same-origin policy~~

modules

~~certificate pinning~~

functions

encapsulation

~~OAuth~~

~~cookies~~

dependencies

~~content security policy~~

immutability

dataflow

~~CSRF~~

~~html sanitization~~

~~HSTS~~

isolation

A **software engineering view** of ~~Web~~ application security

“Security is just the extreme of Modularity”

- Mark S. Miller
(Chief Scientist, Agoric)

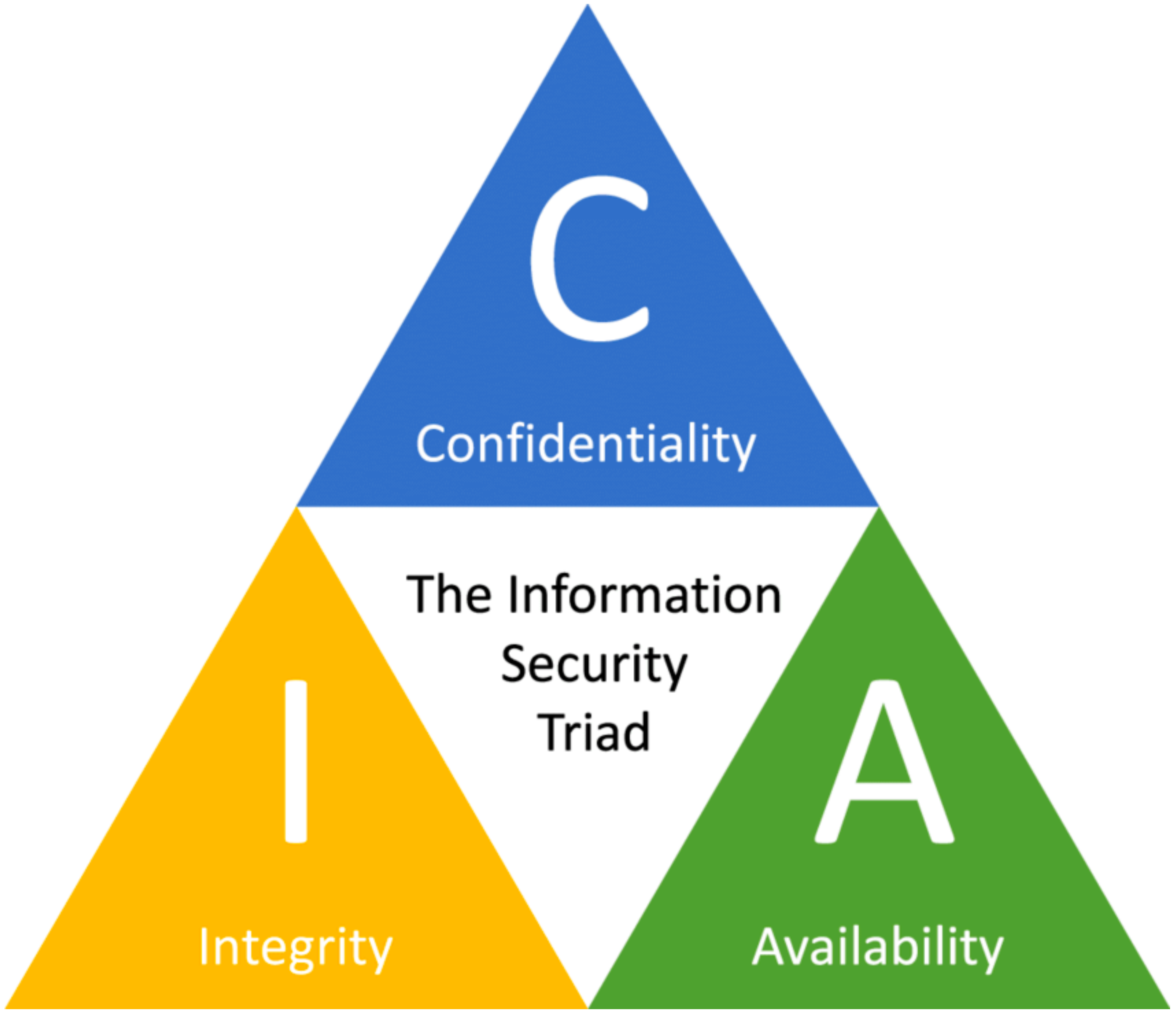
Modularity: avoid needless dependencies (to prevent bugs)

Security: avoid needless vulnerabilities (to prevent exploits)

The CIA triad from an application security perspective

- **Confidentiality** (a.k.a. Secrecy): No one can infer information they are not supposed to know. Confidentiality usually rests on **cryptography** to keep information secret.
  - Example violation: “*Bob learns how much money Alice has in her bank account*”
  - Example threat: side channel attack.
- **Integrity** (a.k.a. Safety): No “bad” things happen. Integrity usually rests on **access control** determining what agents can cause what effects.
  - Example violation: “*Bob steals Alice’s money*”
  - Example threat: confused deputy attack.
- **Availability** (a.k.a. Liveness): “Good” things continue to happen.
  - Example violation: “*Bob prevents Alice from spending her money as she wants*”
  - Example threat: a denial of service attack.

(Source: Miller, “A Taxonomy of Security Issues”, 2021, https://agoric.com/blog/technology/a-taxonomy-of-security-issues)



(Image source: Nikander, Jussi & Manninen, Onni & Laajalahti, Mikko. (2020). Requirements for cybersecurity in agricultural communication networks. Computers and Electronics in Agriculture.)

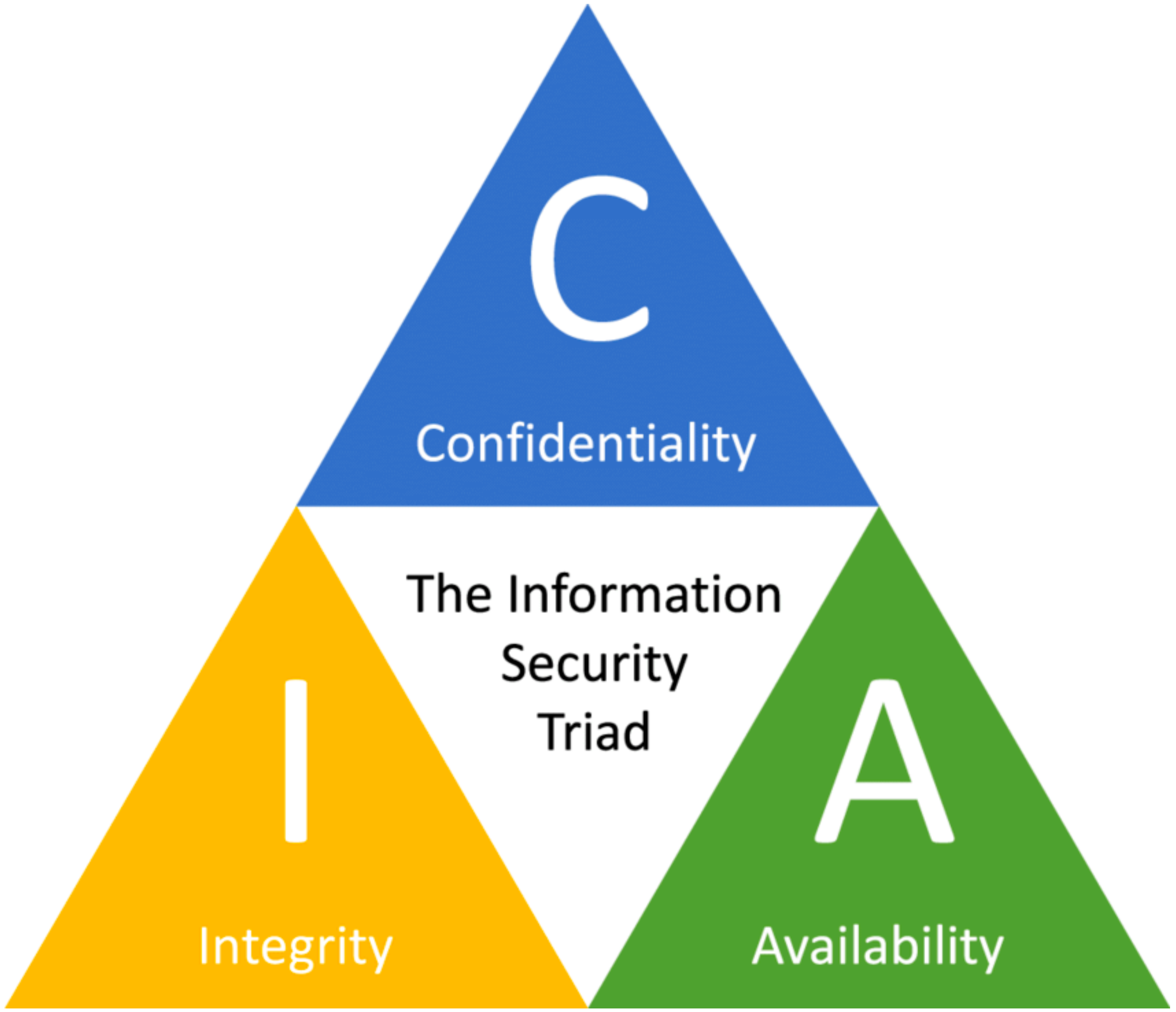
The CIA triad from an application security perspective

- **Confidentiality** (a.k.a. Secrecy): No one can infer information they are not supposed to know. Confidentiality usually rests on **cryptography** to keep information secret.
  - Example violation: “*Bob learns how much money Alice has in her bank account*”
  - Example threat: side channel attack.

Our focus

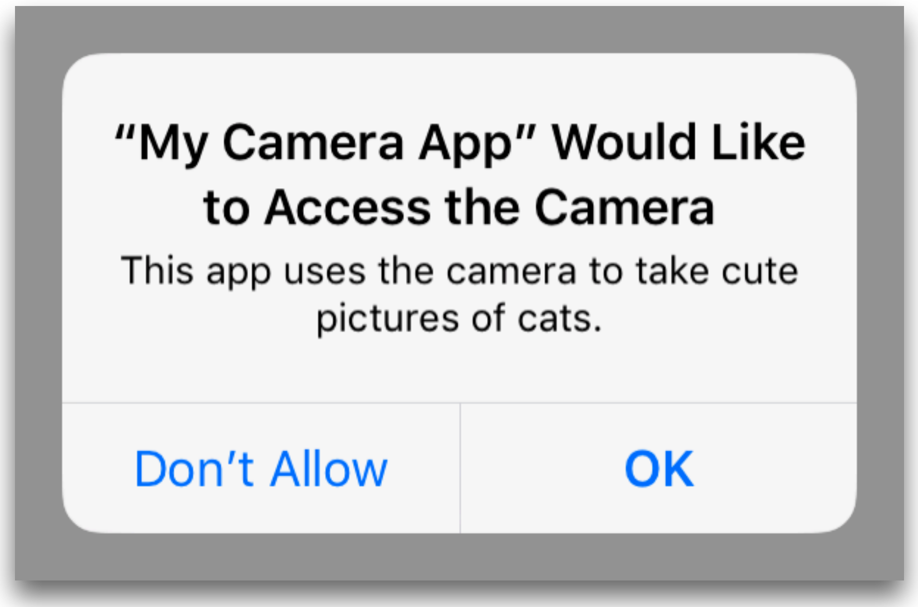
- **Integrity** (a.k.a. Safety): No “bad” things happen. Integrity usually rests on **access control** determining what agents can cause what effects.
  - Example violation: “*Bob steals Alice’s money*”
  - Example threat: confused deputy attack.
- **Availability** (a.k.a. Liveness): “Good” things continue to happen.
  - Example violation: “*Bob prevents Alice from spending her money as she wants*”
  - Example threat: a denial of service attack.

(Source: Miller, “A Taxonomy of Security Issues”, 2021, https://agoric.com/blog/technology/a-taxonomy-of-security-issues)



(Image source: Nikander, Jussi & Manninen, Onni & Laajalahti, Mikko. (2020). Requirements for cybersecurity in agricultural communication networks. Computers and Electronics in Agriculture.)

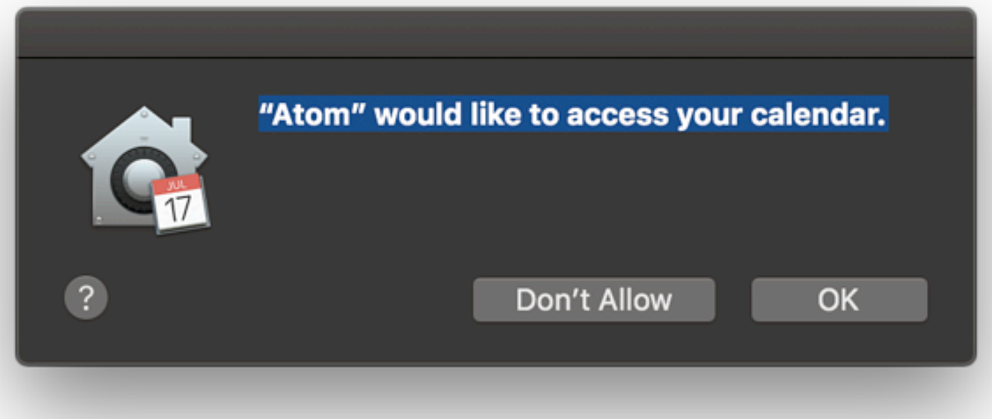
Application integrity & access control



“My Camera App” Would Like to Access the Camera

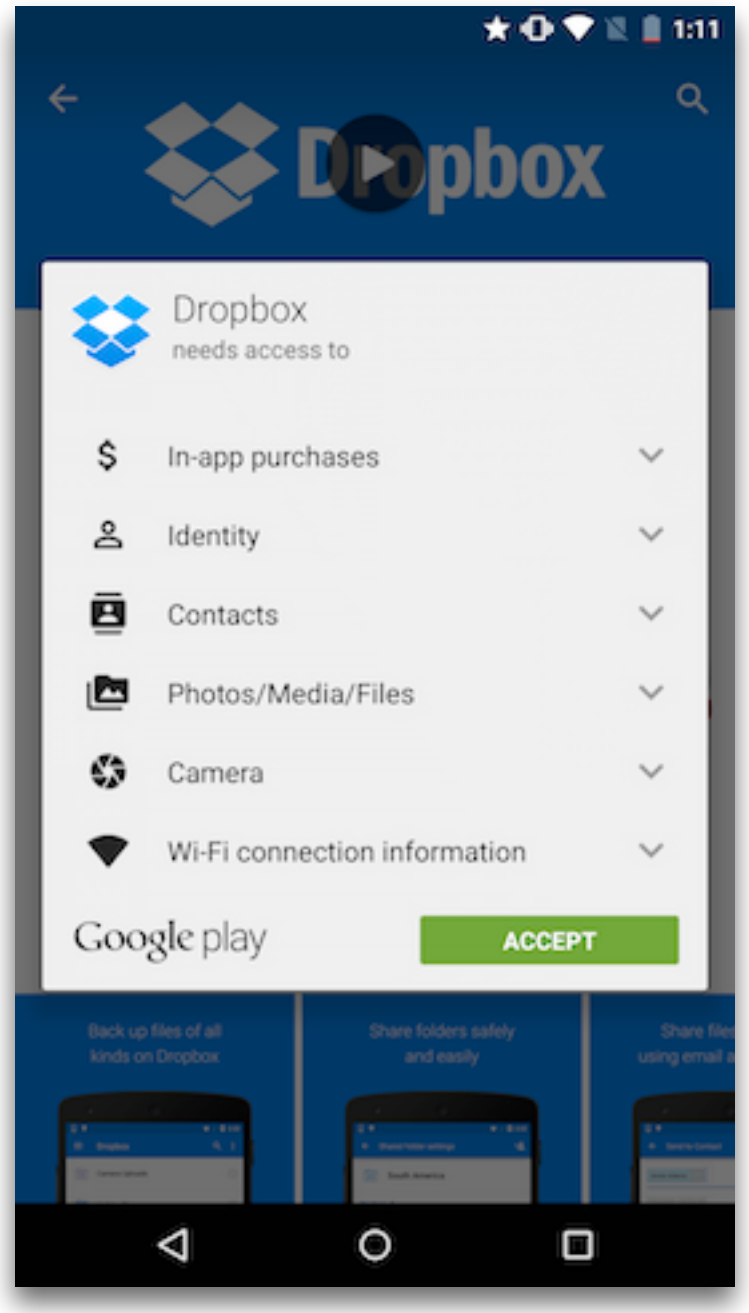
This app uses the camera to take cute pictures of cats.

Don’t Allow | OK



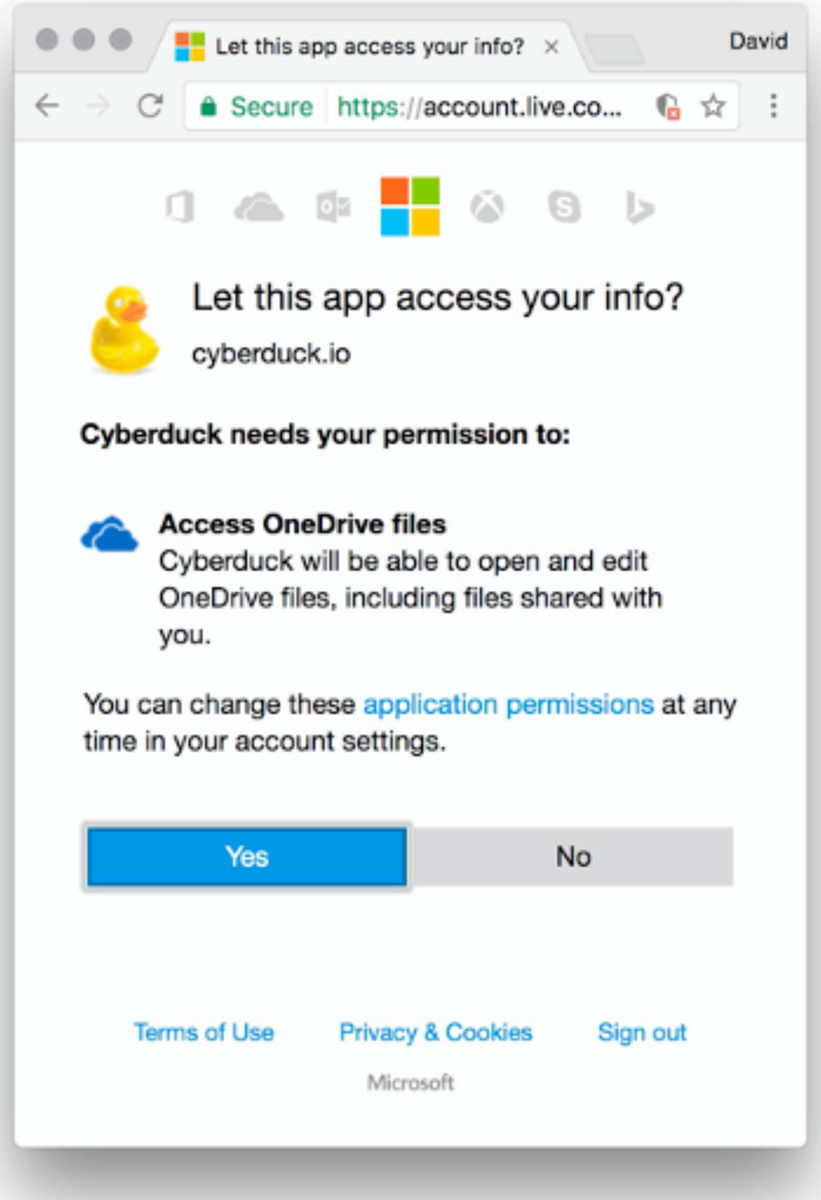
“Atom” would like to access your calendar.

Don’t Allow | OK



Dropbox
needs access to

- In-app purchases
- Identity
- Contacts
- Photos/Media/Files
- Camera
- Wi-Fi connection information



Let this app access your info?
David
Secure https://account.live.co...

Let this app access your info?
cyberduck.io

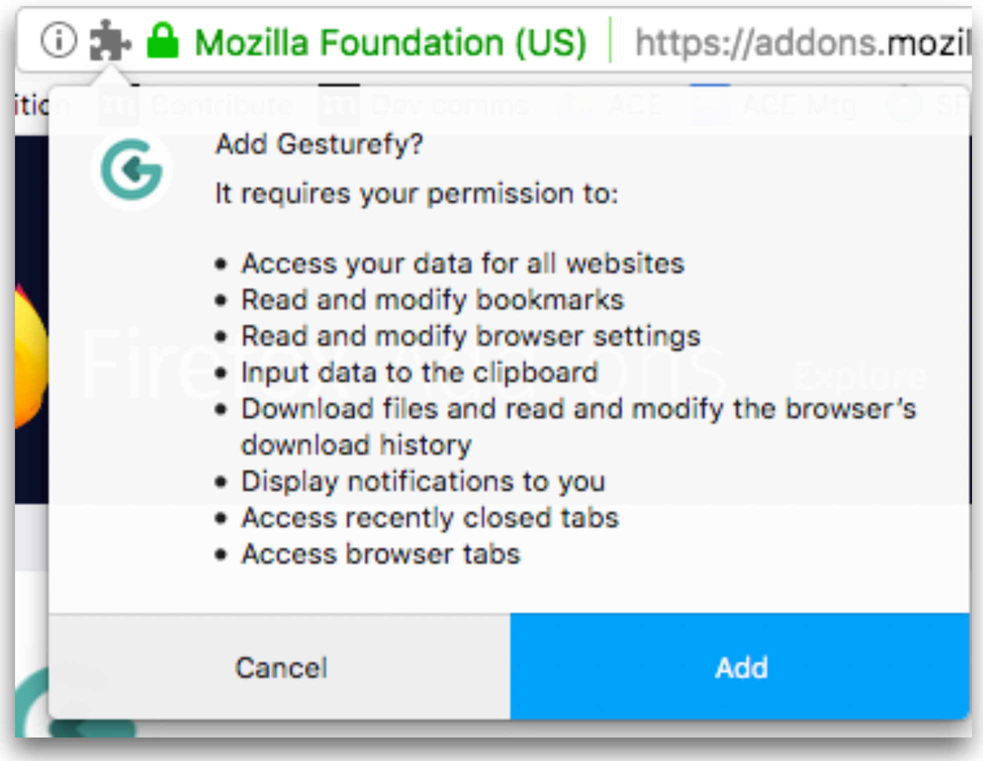
Cyberduck needs your permission to:

Access OneDrive files
Cyberduck will be able to open and edit OneDrive files, including files shared with you.

You can change these application permissions at any time in your account settings.

Yes | No

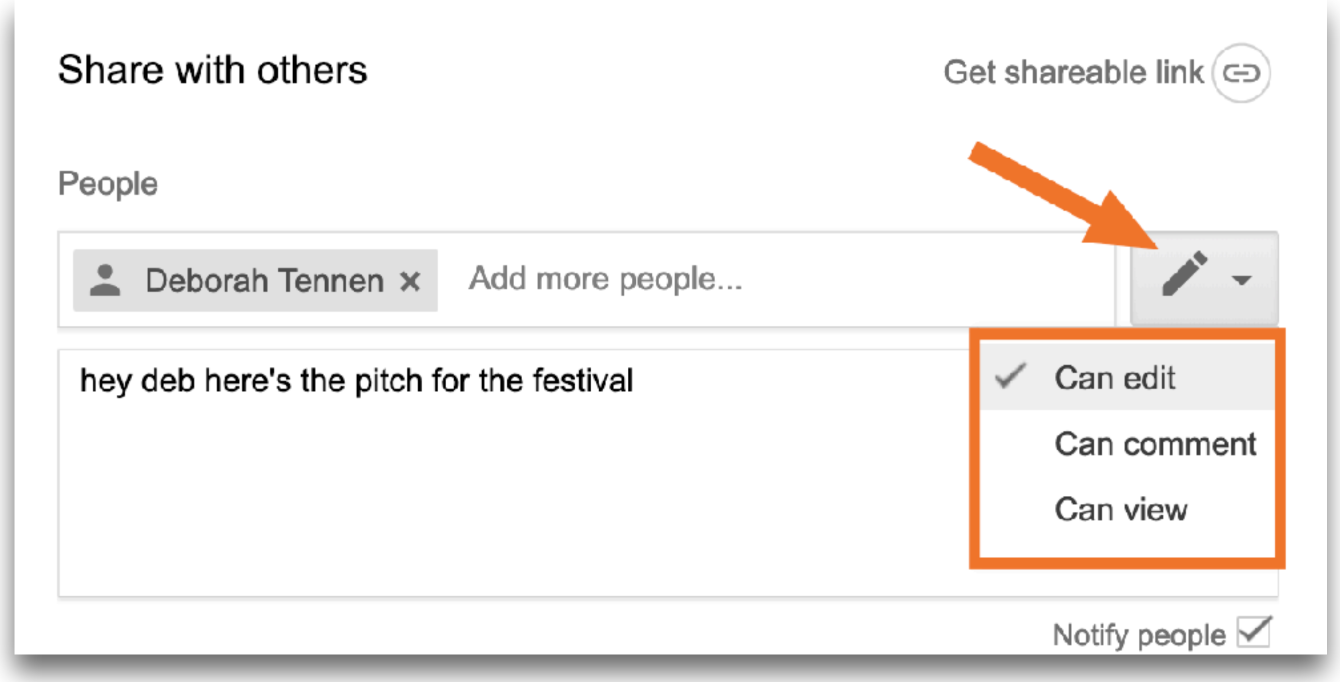
Terms of Use
Privacy & Cookies
Sign out
Microsoft



Add Gesturefy?
It requires your permission to:

- Access your data for all websites
- Read and modify bookmarks
- Read and modify browser settings
- Input data to the clipboard
- Download files and read and modify the browser’s download history
- Display notifications to you
- Access recently closed tabs
- Access browser tabs

Cancel | Add



Share with others

People

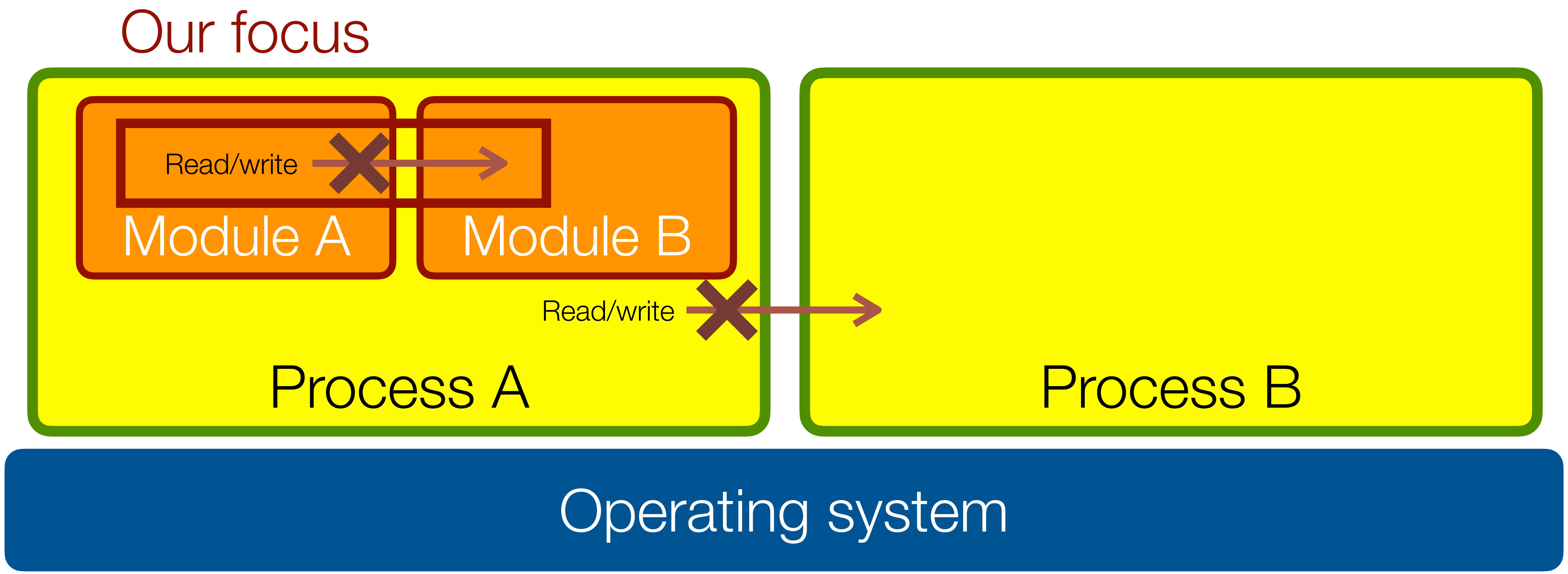
Deborah Tennen × Add more people...

hey deb here's the pitch for the festival

- ✓ Can edit
- Can comment
- Can view

Notify people ☑

Application integrity (safety): going beyond OS process isolation

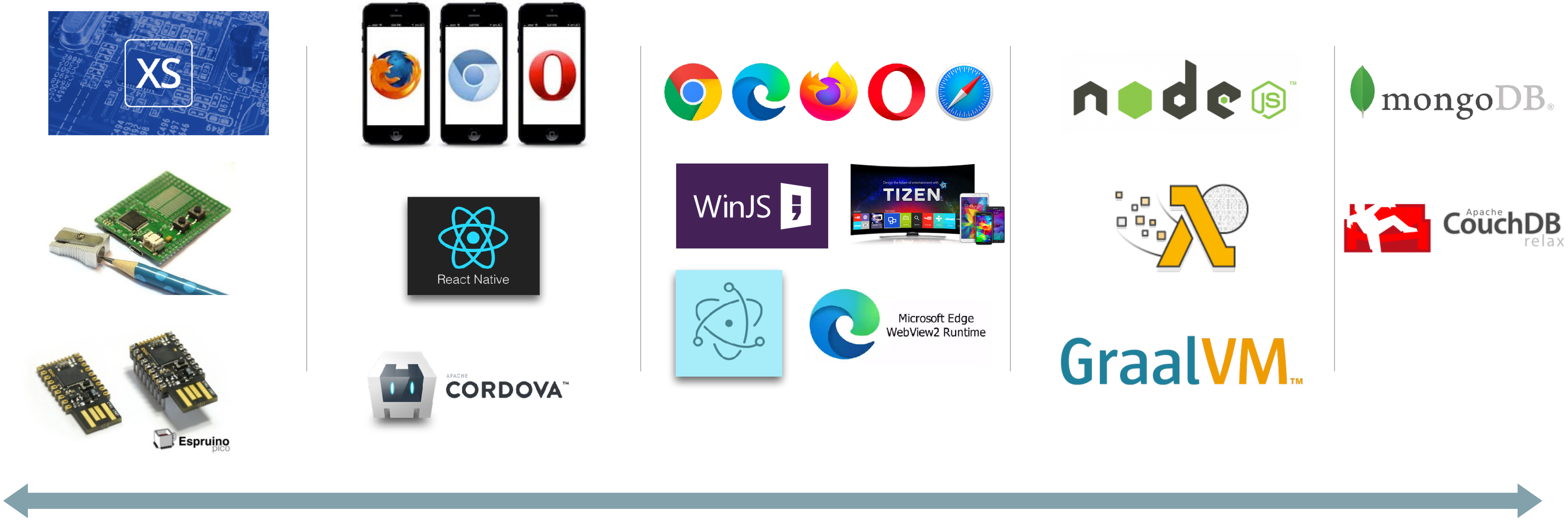


This Lecture

- Part I: **why module isolation** is critical to modern JavaScript applications
- Part II: the **Principle of Least Authority**, by example (in JavaScript)
- Part III: safely composing modules using **least-authority patterns**

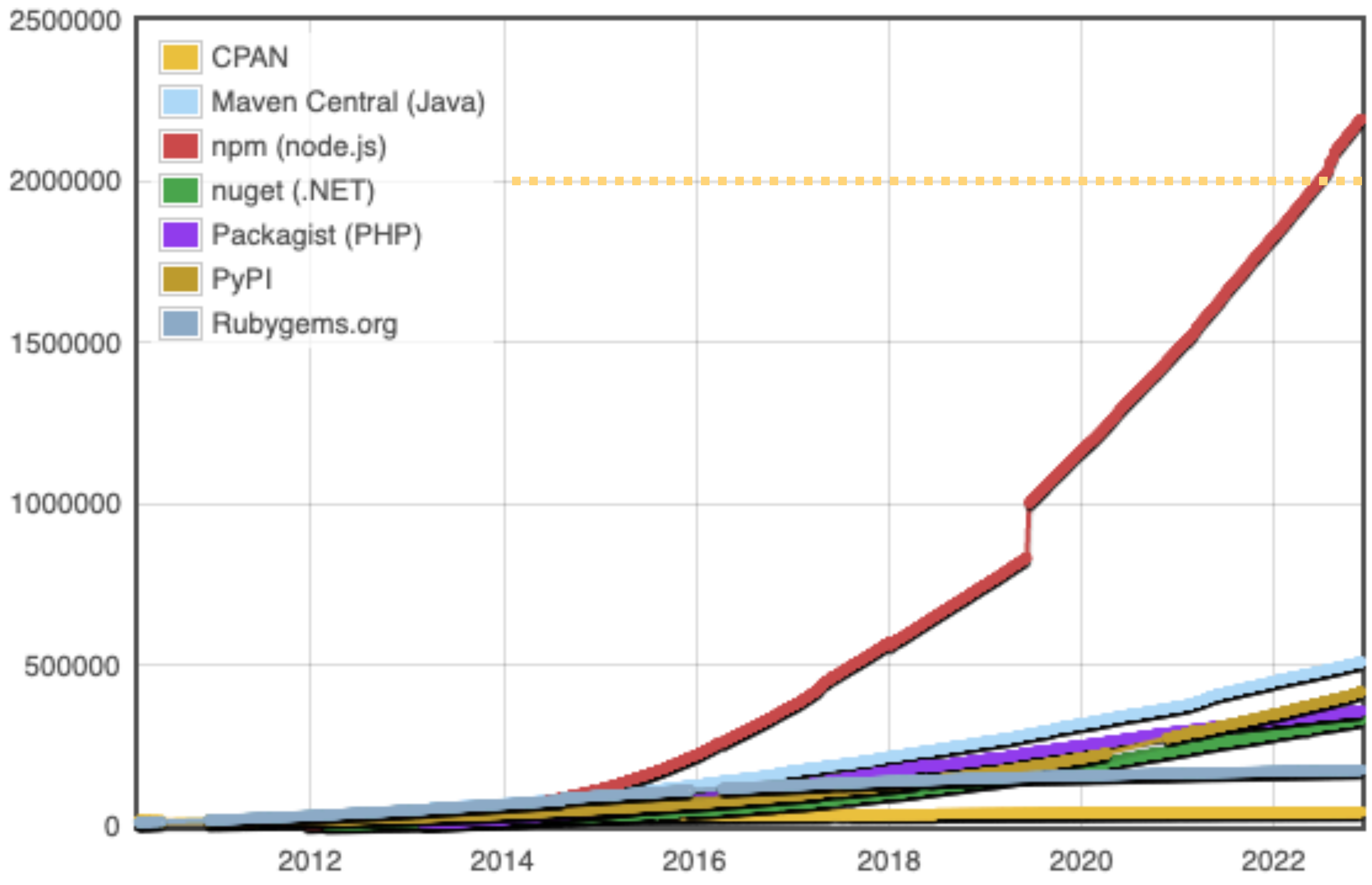
Part I
Why module isolation is critical to modern JavaScript applications

JavaScript is no longer just about the Web. Used widely across *all* tiers.



Embedded Mobile Desktop/Native Server Database

Modern JavaScript applications are built from thousands of modules



2,000,000 modules on NPM

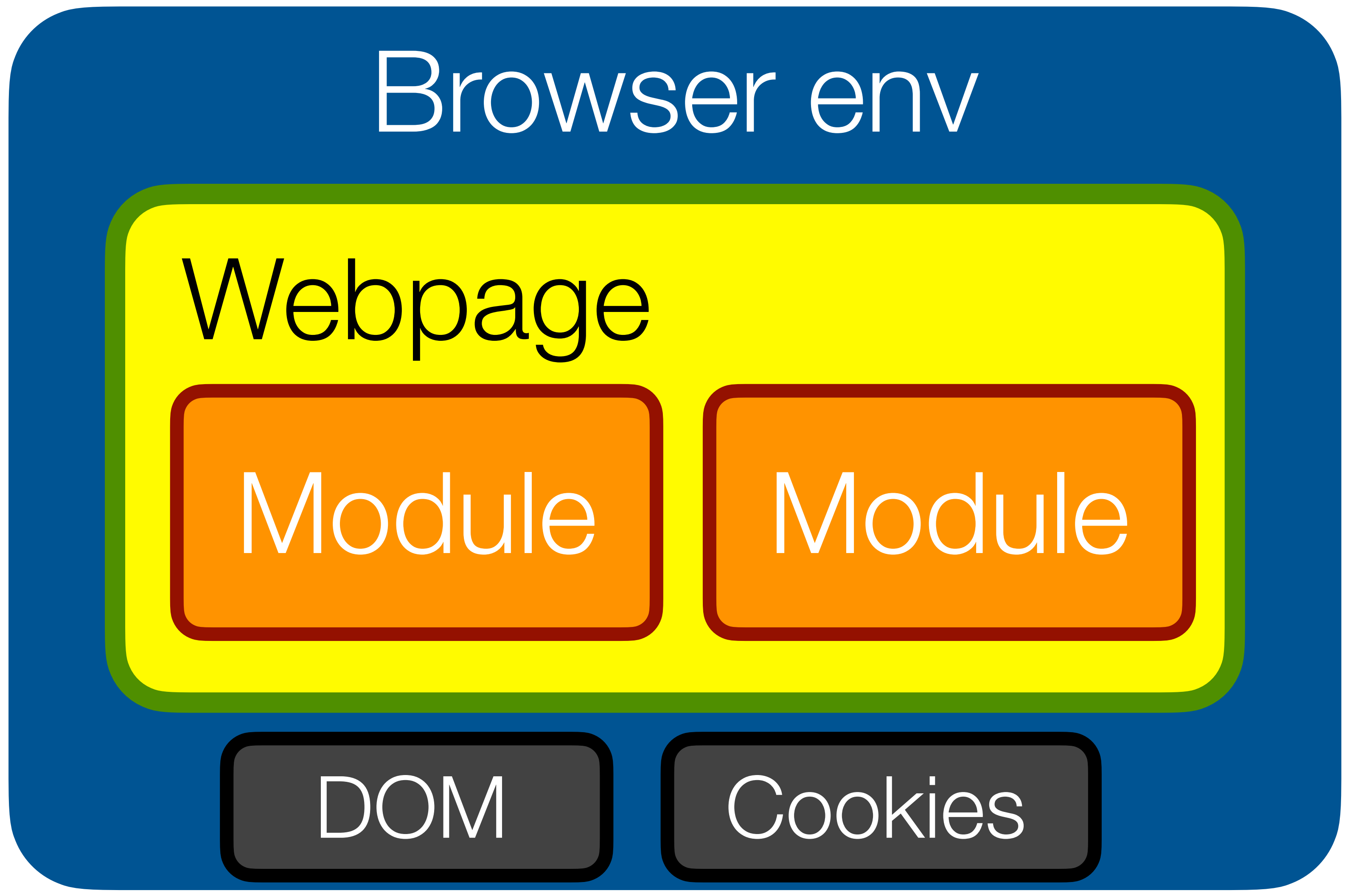
(source: modulecounts.com, Nov 2022)

“The average modern web application has over 1000 modules […] ***97% of the code in a modern web application comes from npm***. An individual developer is responsible only for the final 3% that makes their application unique and useful.”

(source: npm blog, December 2018)

Composing modules: it’s all about **trust**

It is exceedingly common to run code you don’t know or trust in a common environment



Browser env

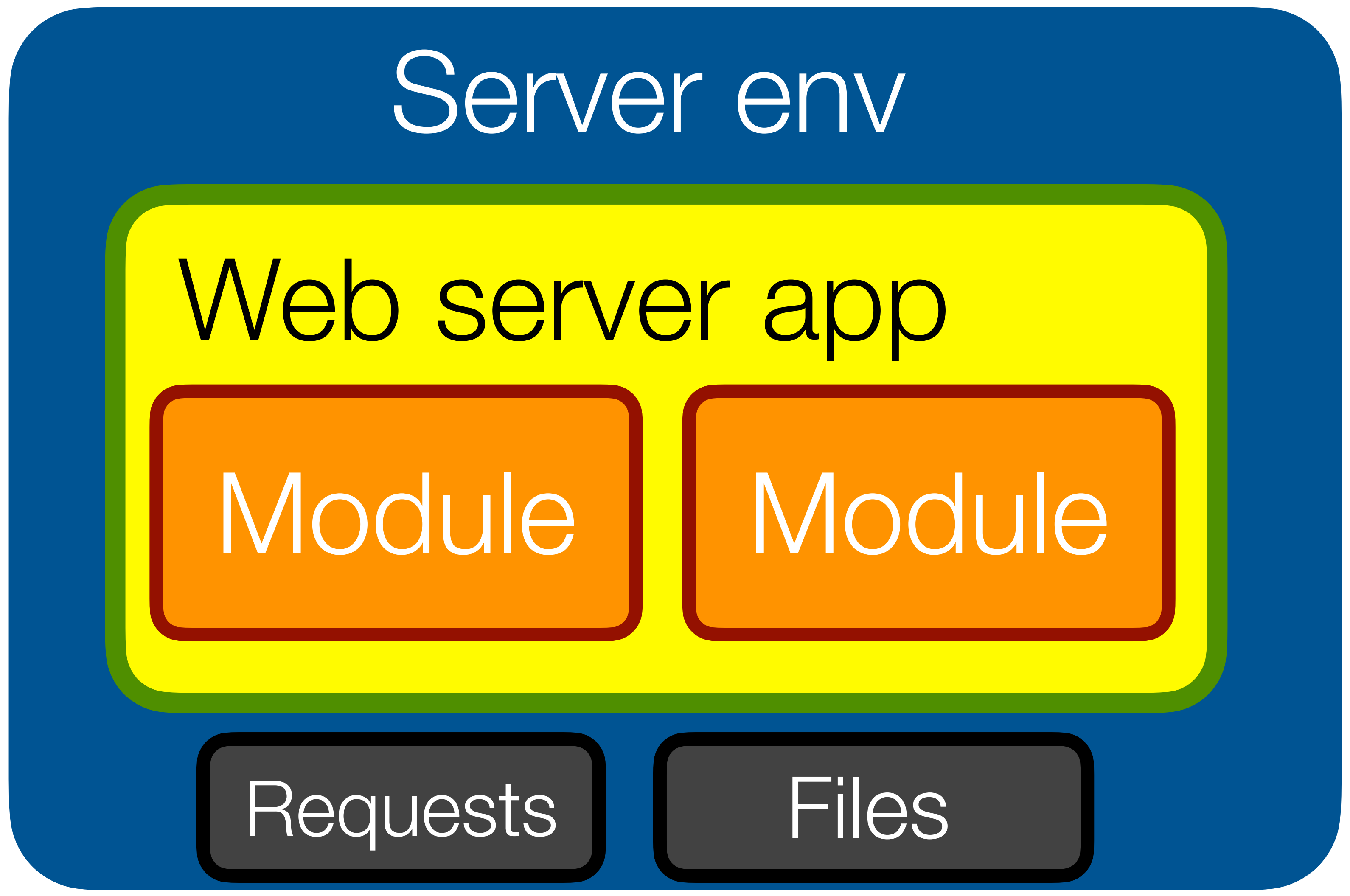
Webpage

Module

Module

DOM

Cookies



Server env

Web server app

Module

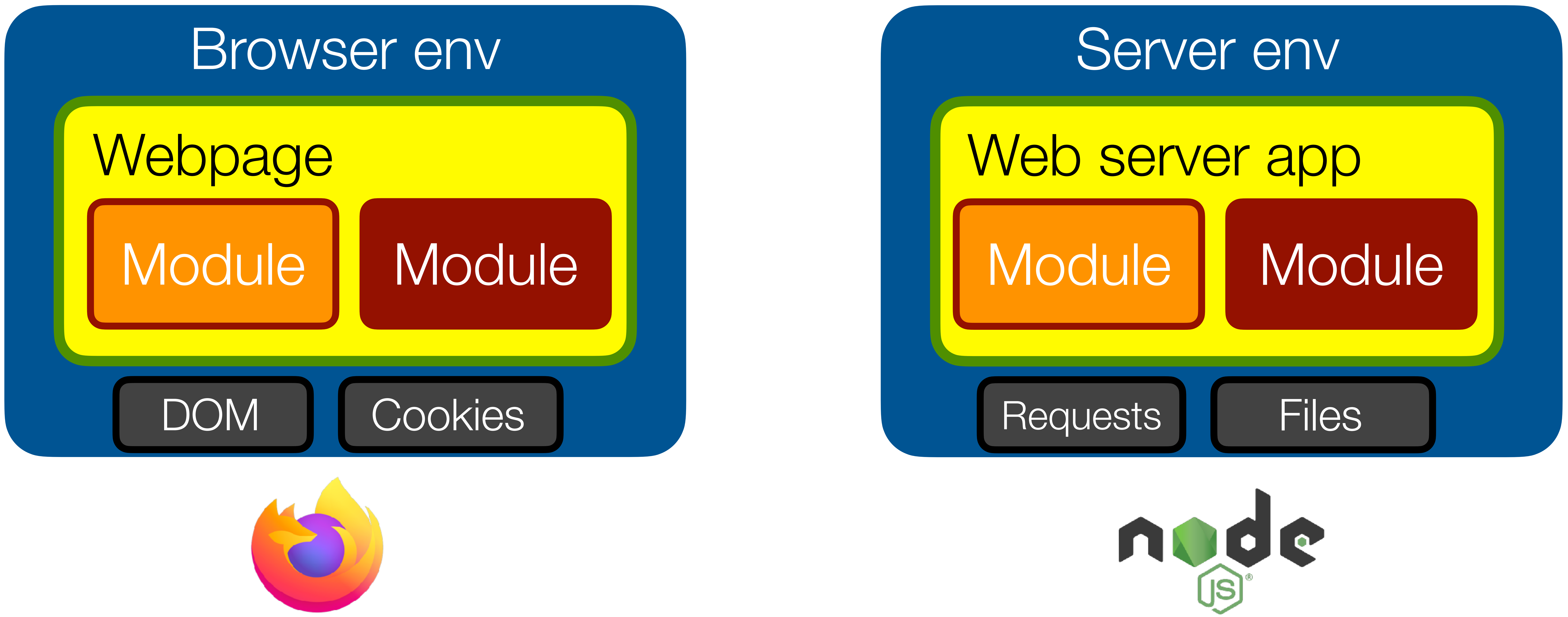
Module

Requests

Files

What can happen when a module goes **rogue**?

It is exceedingly common to run code you don’t know or trust in a common environment



Browser env

Webpage

Module

Module

DOM

Cookies

Server env

Web server app

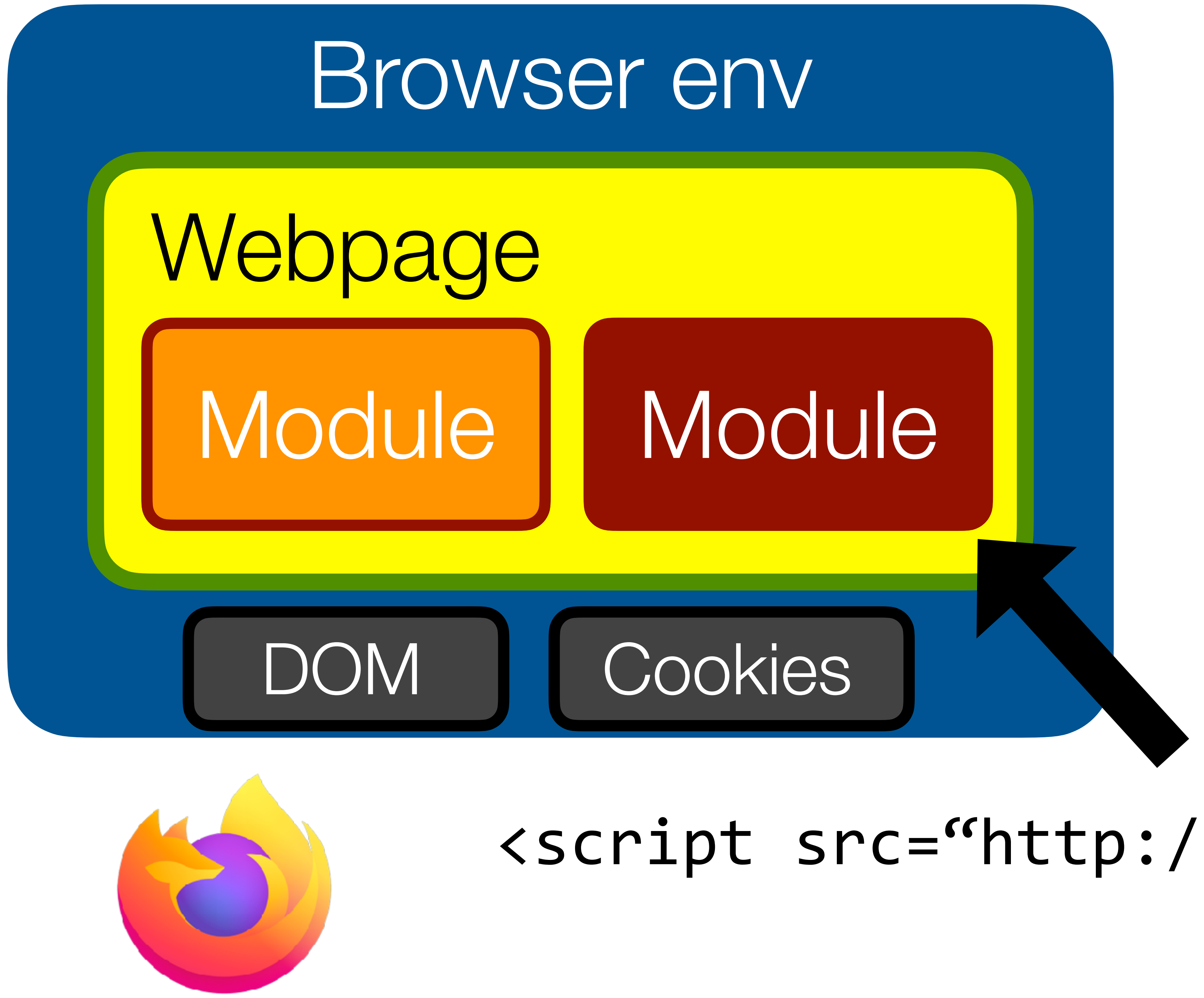
Module

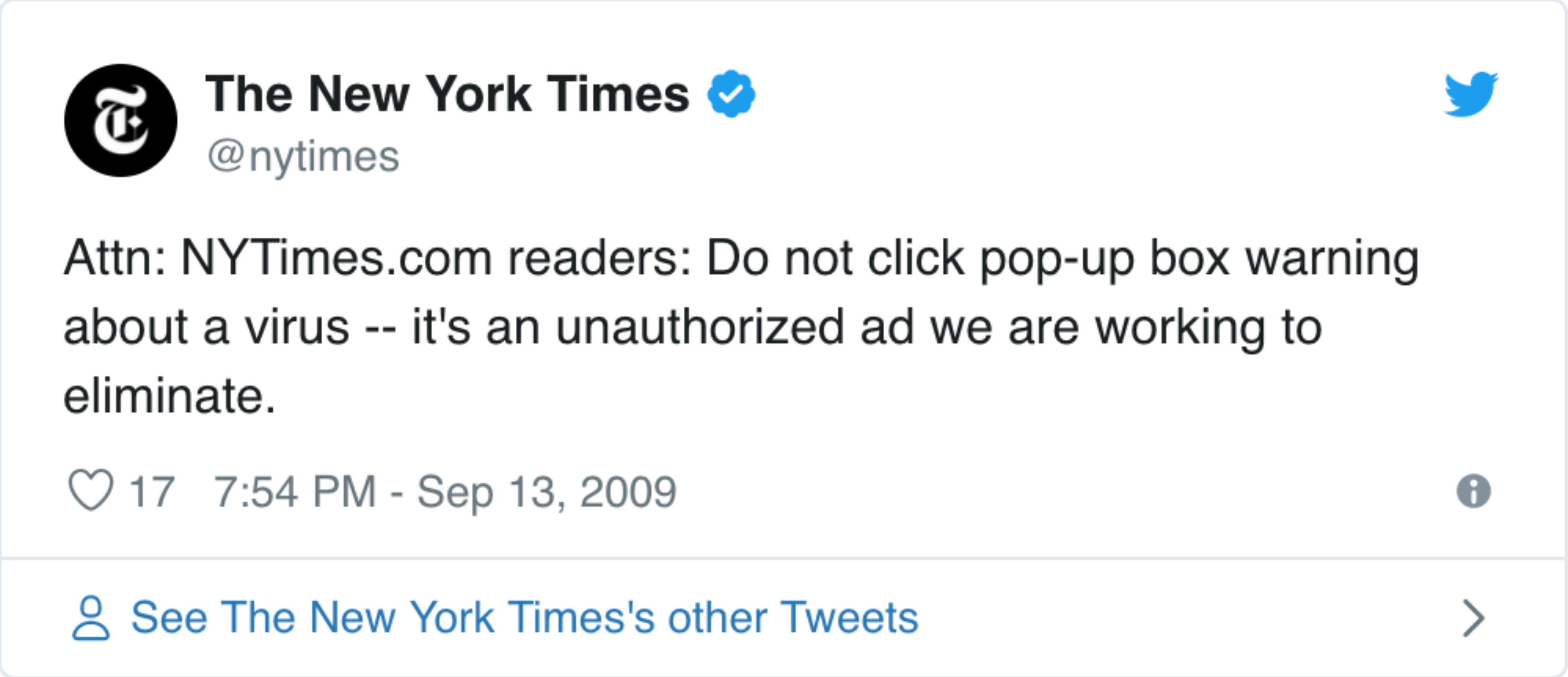
Module

Requests

Files

What can happen when a module goes **rogue**?





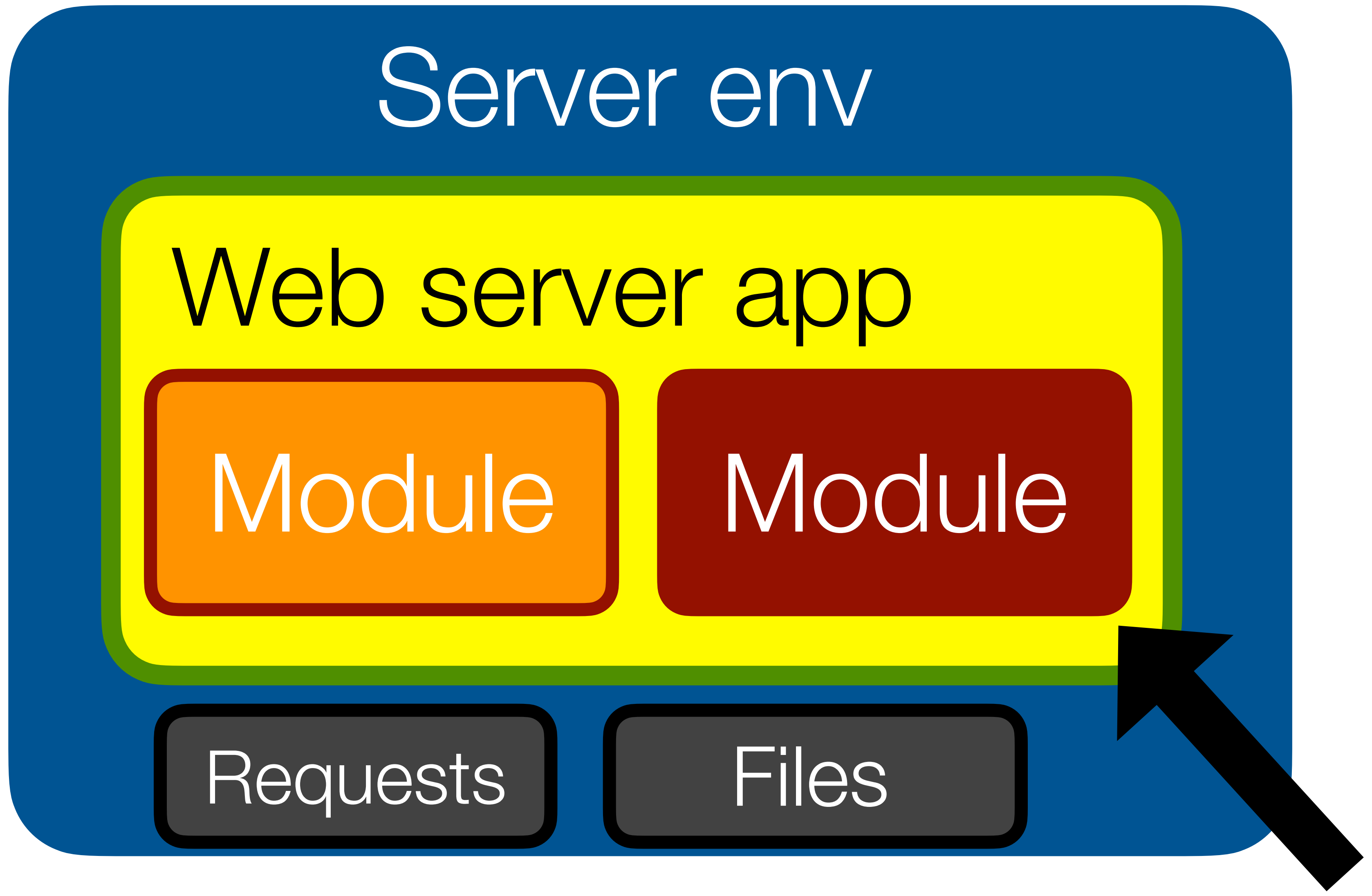
Attn: NYTimes.com readers: Do not click pop-up box warning about a virus -- it's an unauthorized ad we are working to eliminate.

17 7:54 PM - Sep 13, 2009

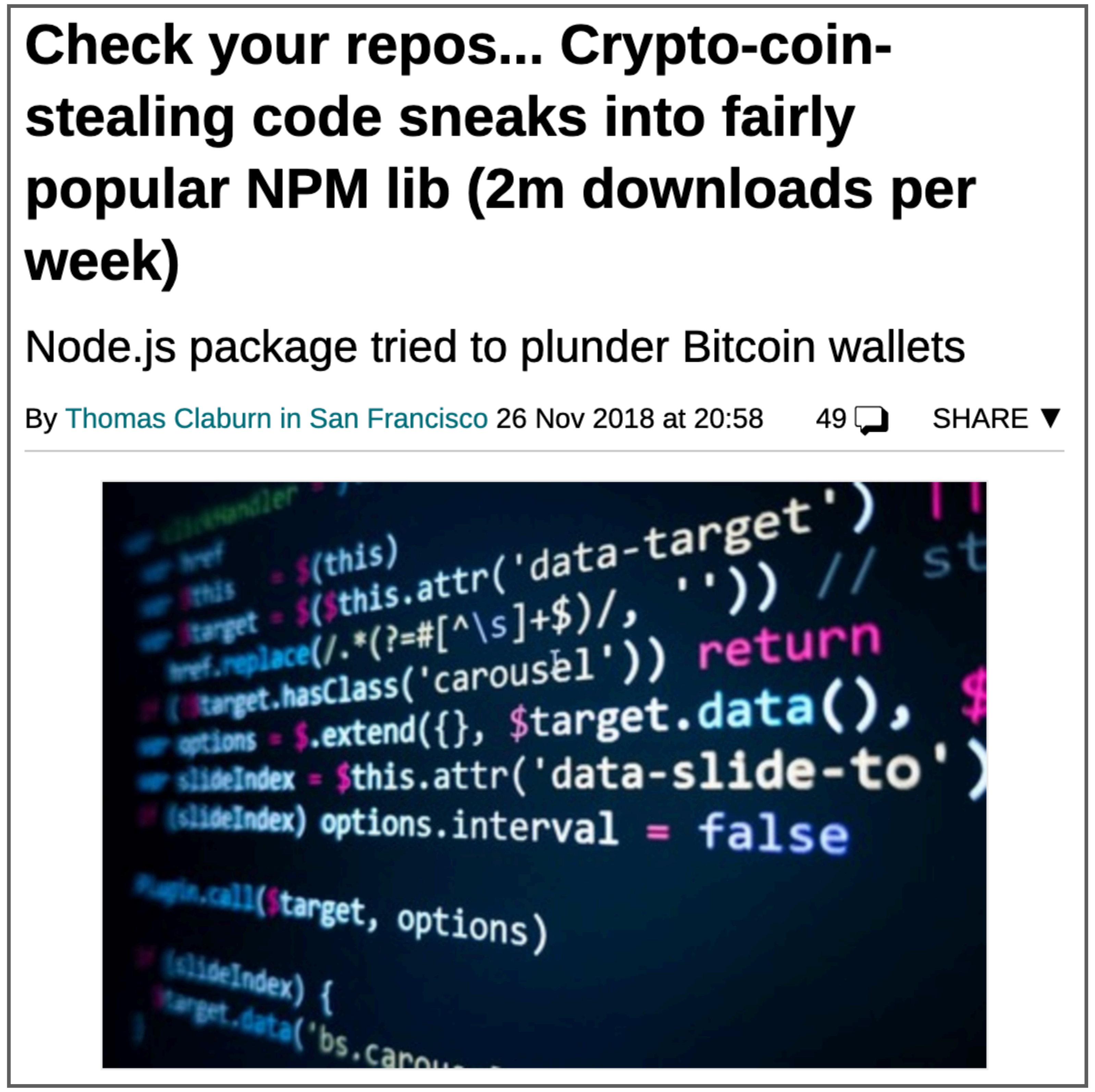
See The New York Times's other Tweets

```
<script src=“http://evil.com/ad.js”>
```

What can happen when a module goes **rogue**?



```
npm install event-stream
```



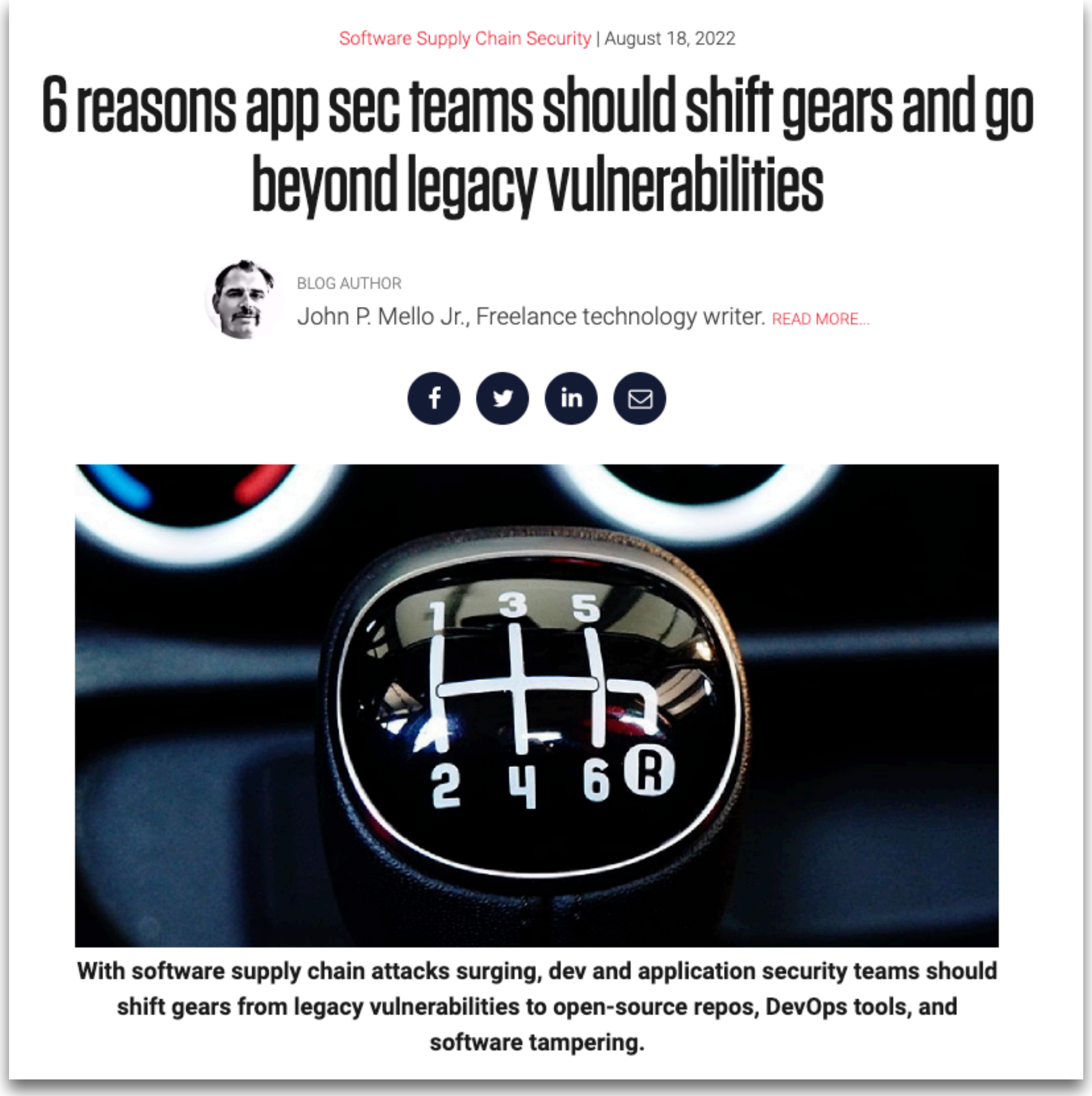
Check your repos... Crypto-coin-stealing code sneaks into fairly popular NPM lib (2m downloads per week)

Node.js package tried to plunder Bitcoin wallets

By Thomas Claburn in San Francisco 26 Nov 2018 at 20:58 49 SHARE ▼

(source: theregister.co.uk)

These are examples of **software supply chain** attacks

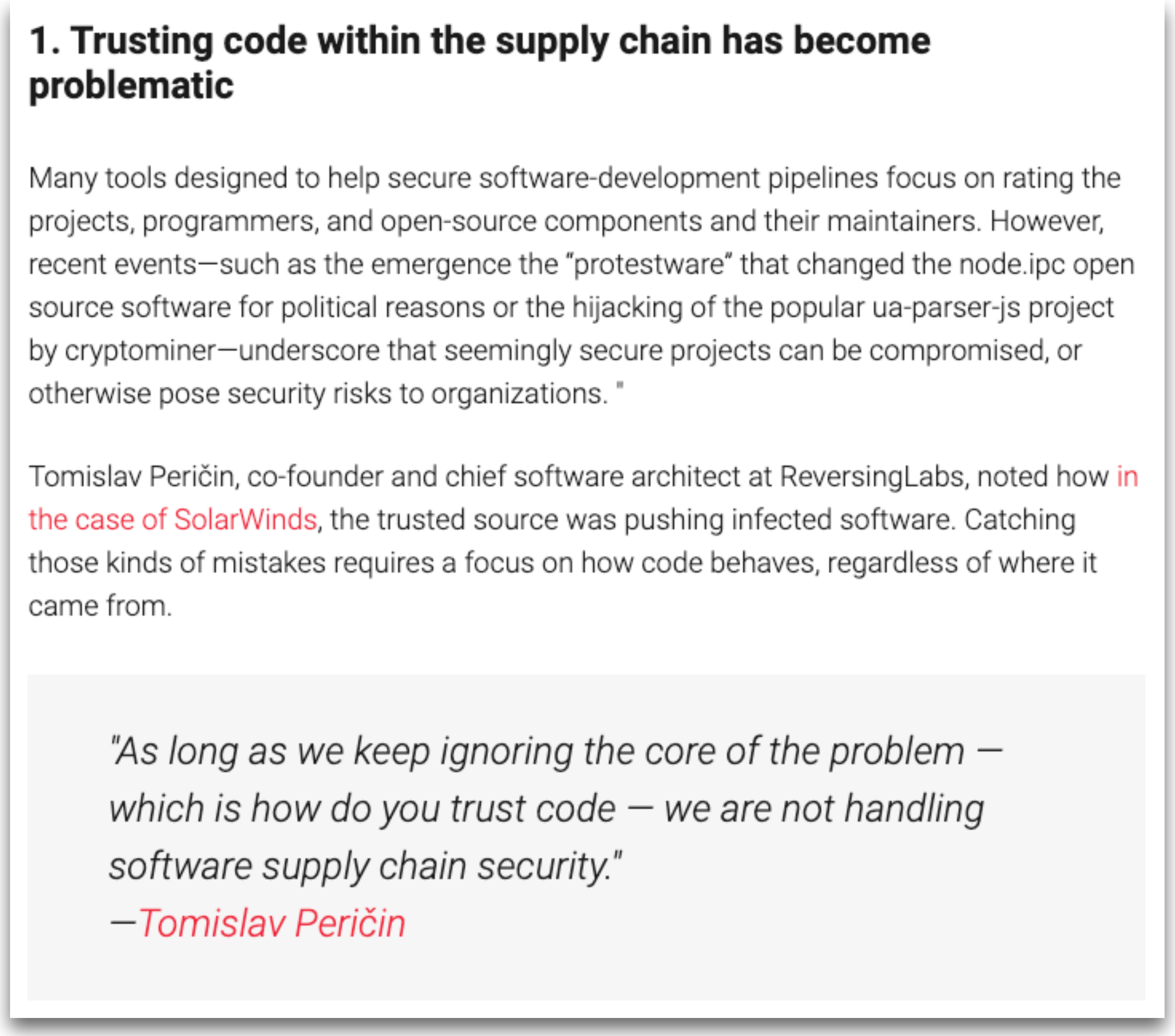


Software Supply Chain Security | August 18, 2022

6 reasons app sec teams should shift gears and go beyond legacy vulnerabilities

BLOG AUTHOR
John P. Mello Jr., Freelance technology writer. READ MORE...

With software supply chain attacks surging, dev and application security teams should shift gears from legacy vulnerabilities to open-source repos, DevOps tools, and software tampering.



1. Trusting code within the supply chain has become problematic

Many tools designed to help secure software-development pipelines focus on rating the projects, programmers, and open-source components and their maintainers. However, recent events—such as the emergence the "protestware" that changed the node.ipc open source software for political reasons or the hijacking of the popular ua-parser-js project by cryptominer—underscore that seemingly secure projects can be compromised, or otherwise pose security risks to organizations. "

Tomislav Peričin, co-founder and chief software architect at ReversingLabs, noted how in the case of SolarWinds, the trusted source was pushing infected software. Catching those kinds of mistakes requires a focus on how code behaves, regardless of where it came from.

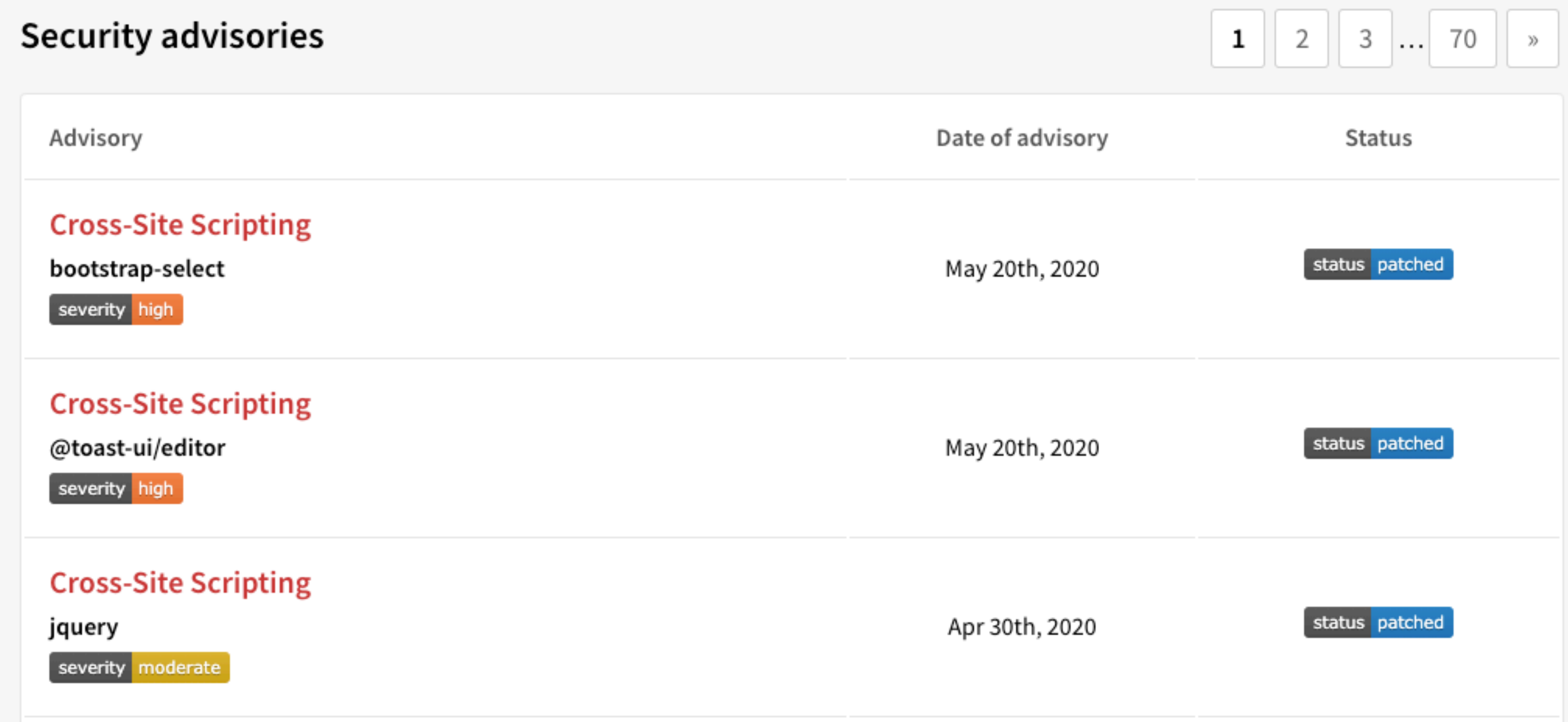
> *"As long as we keep ignoring the core of the problem — which is how do you trust code — we are not handling software supply chain security."*
> *—Tomislav Peričin*

(Source: https://develop.secure.software/6-reasons-software-security-teams-need-to-go-beyond-vulnerability-response, august 2022)

Increasing awareness

Great tools, but address the symptoms, not the root cause

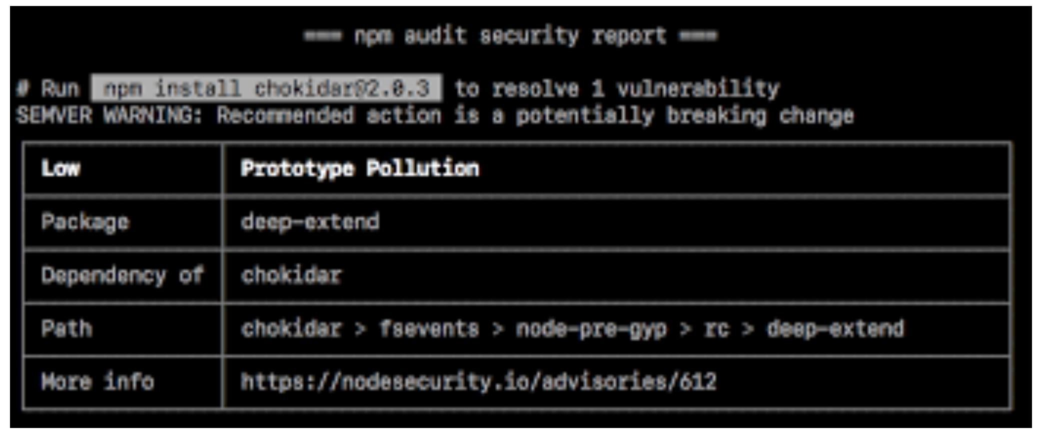
npm security advisories



Security advisories

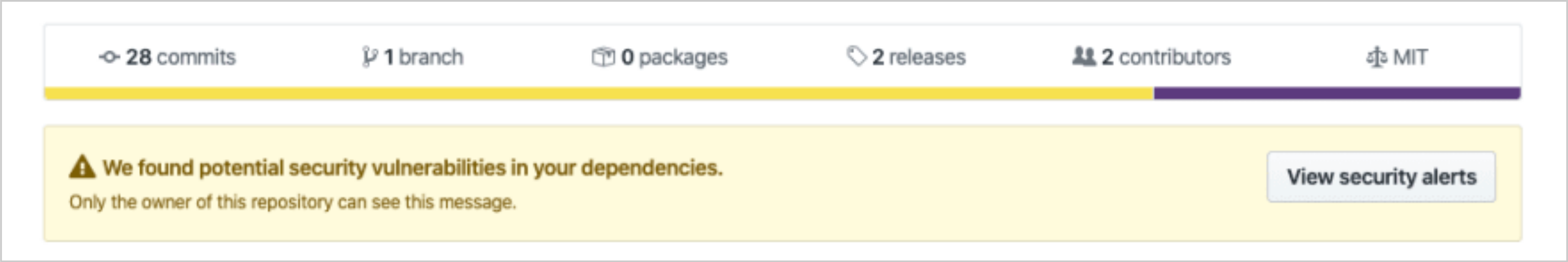
Advisory	Date of advisory	Status
Cross-Site Scripting bootstrap-select severity high	May 20th, 2020	status patched
Cross-Site Scripting @toast-ui/editor severity high	May 20th, 2020	status patched
Cross-Site Scripting jquery severity moderate	Apr 30th, 2020	status patched

npm audit



```
=== npm audit security report ===

# Run  npm install chokidar@2.0.3  to resolve 1 vulnerability
SEMVER WARNING: Recommended action is a potentially breaking change
┌───────────────┬──────────────────────────────────────────────────┐
│ Low           │ Prototype Pollution                              │
├───────────────┼──────────────────────────────────────────────────┤
│ Package       │ deep-extend                                      │
├───────────────┼──────────────────────────────────────────────────┤
│ Dependency of │ chokidar                                         │
├───────────────┼──────────────────────────────────────────────────┤
│ Path          │ chokidar > fsevents > node-pre-gyp > rc > deep-extend │
├───────────────┼──────────────────────────────────────────────────┤
│ More info     │ https://nodesecurity.io/advisories/612           │
└───────────────┴──────────────────────────────────────────────────┘
```

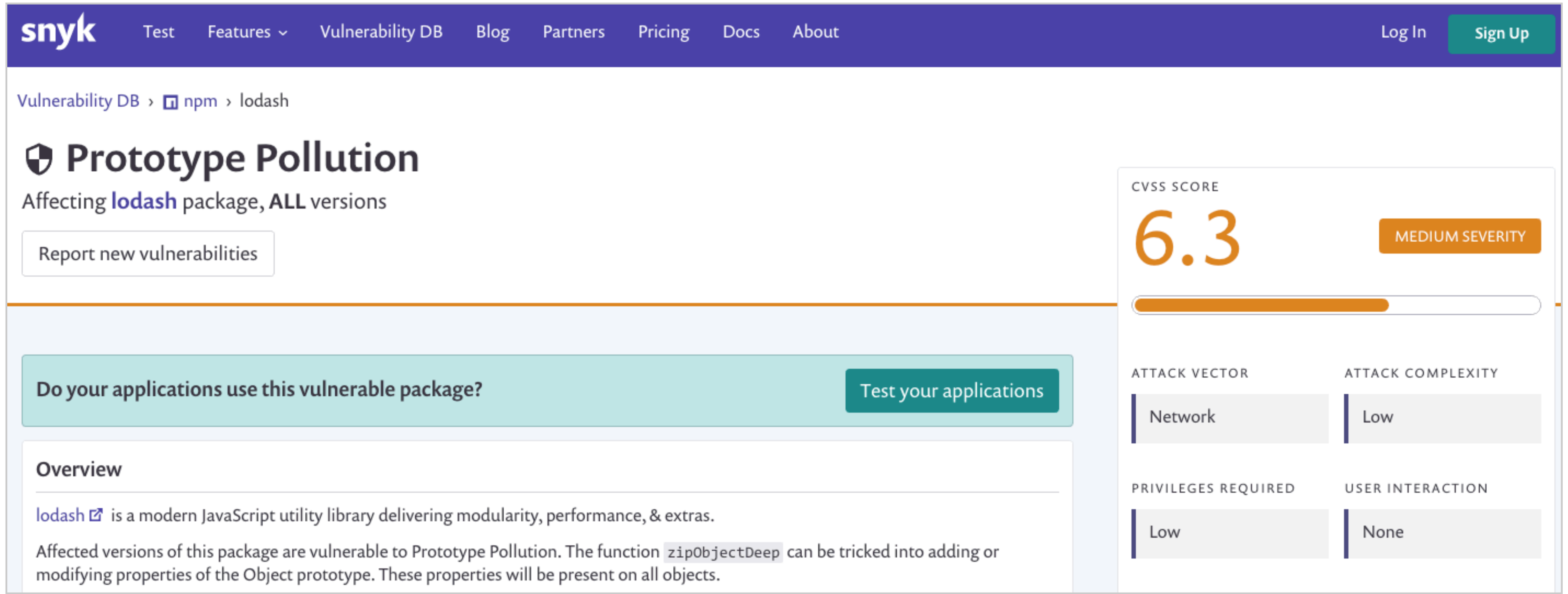
GitHub security alerts



⚠ We found potential security vulnerabilities in your dependencies.

Only the owner of this repository can see this message.

Snyk vulnerability DB



Vulnerability DB › npm › lodash

Prototype Pollution

Affecting **lodash** package, **ALL** versions

CVSS SCORE 6.3 MEDIUM SEVERITY

ATTACK VECTOR: Network

ATTACK COMPLEXITY: Low

PRIVILEGES REQUIRED: Low

USER INTERACTION: None

Do your applications use this vulnerable package?

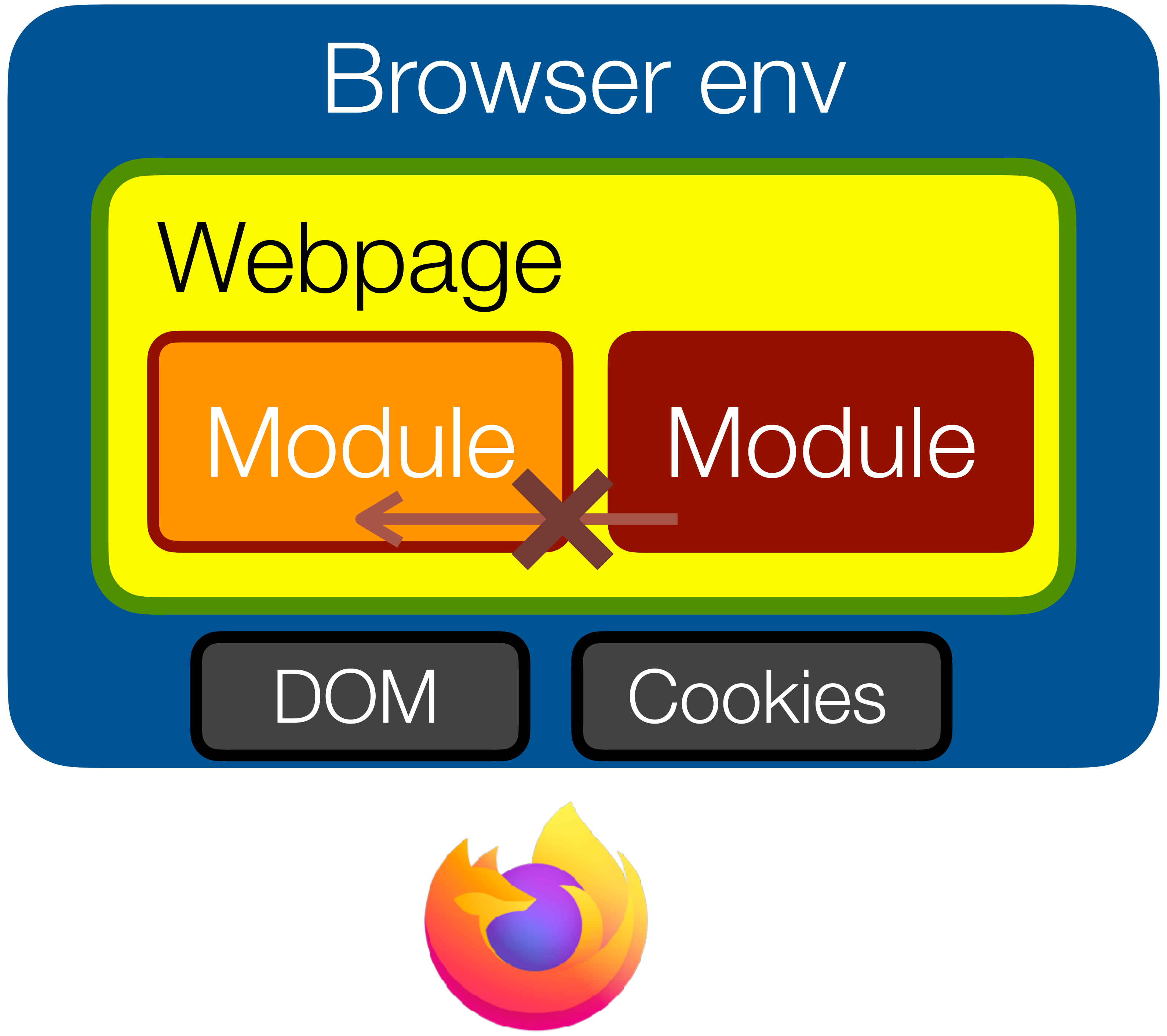
Overview

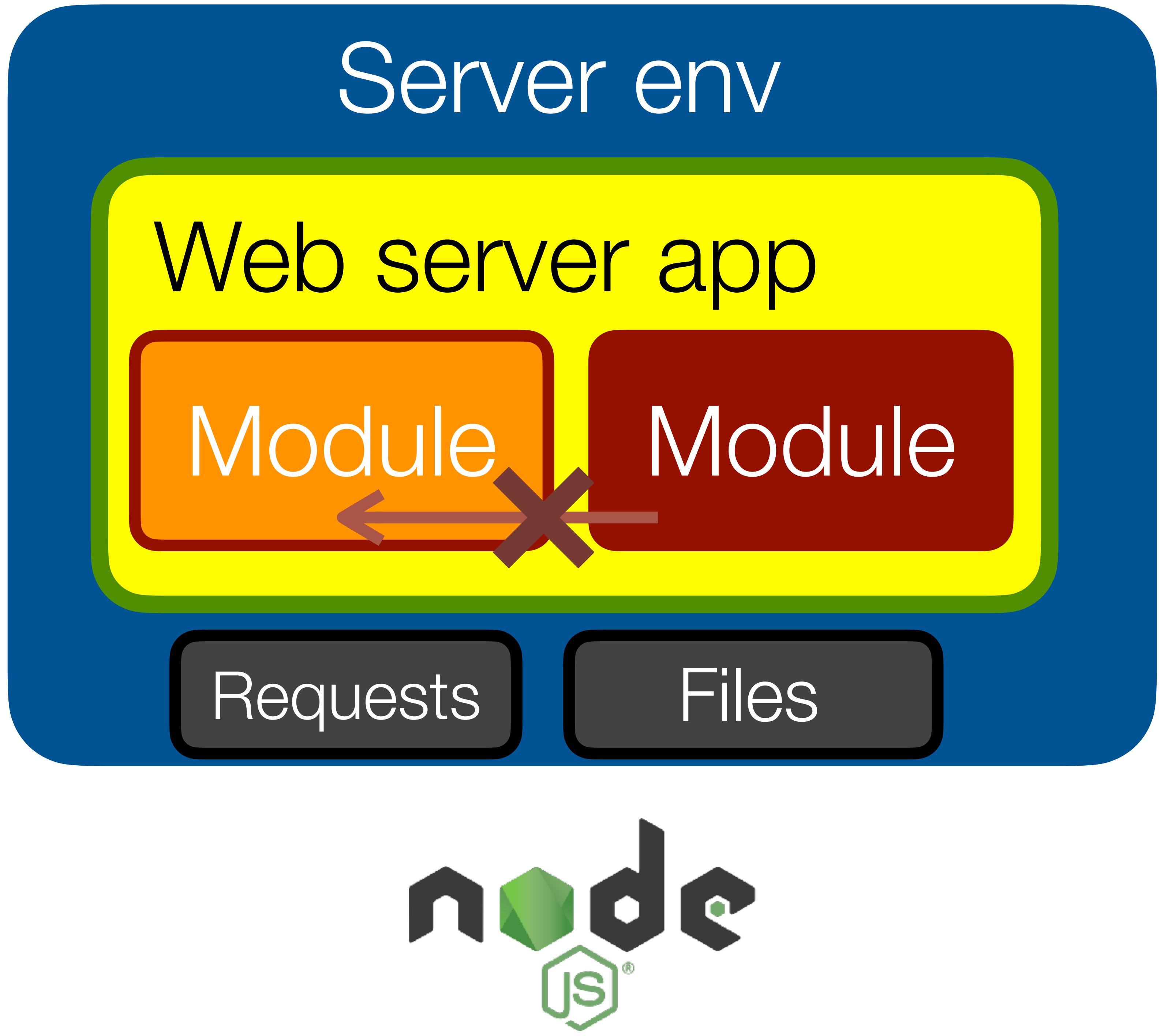
lodash is a modern JavaScript utility library delivering modularity, performance, & extras.

Affected versions of this package are vulnerable to Prototype Pollution. The function `zipObjectDeep` can be tricked into adding or modifying properties of the Object prototype. These properties will be present on all objects.

Avoiding interference is the name of the game

- Shield important resources/APIs from modules that don’t need access
- Apply **Principle of Least Authority** (POLA) to application design

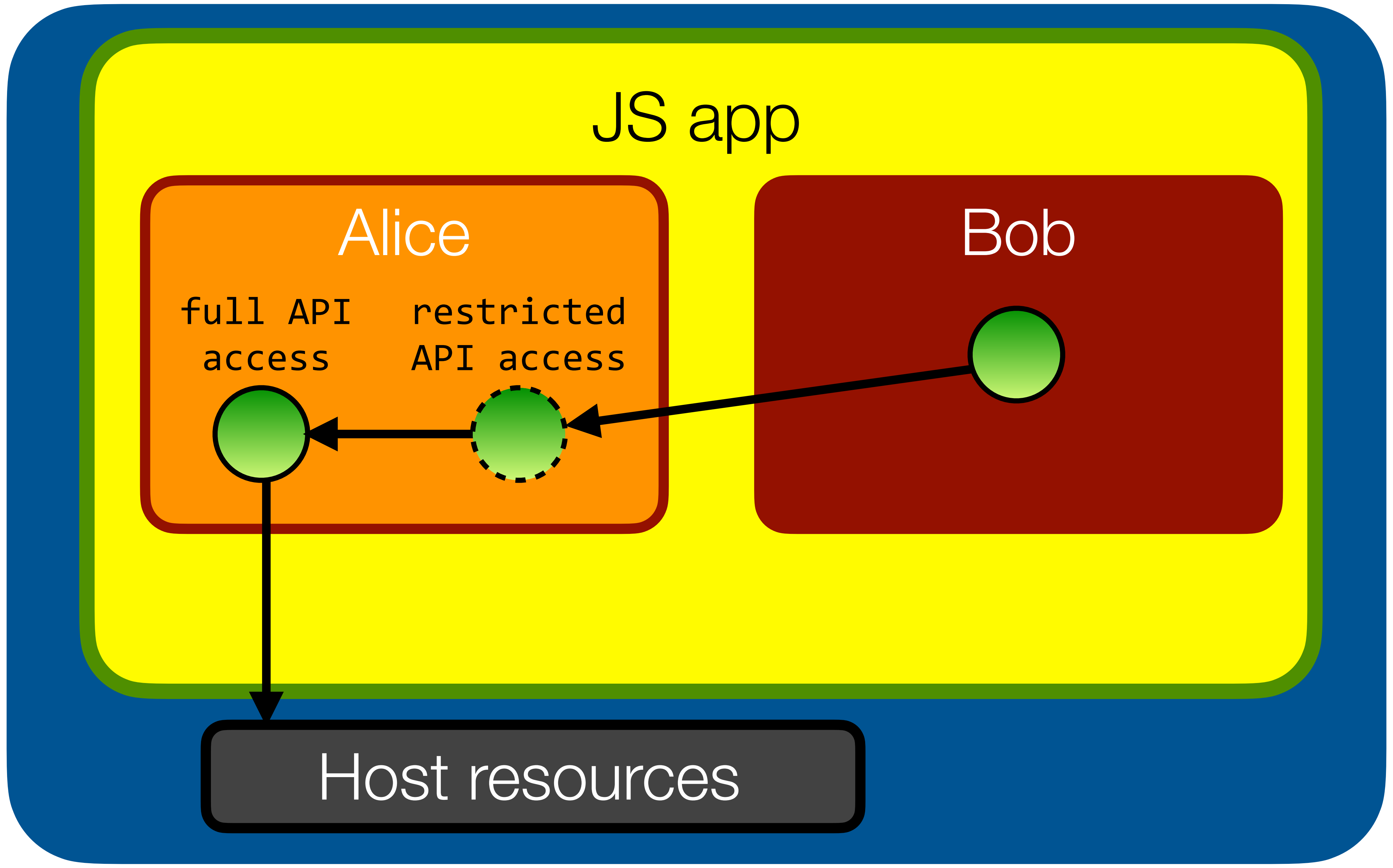




Part II
The Principle of Least Authority, by example (in JavaScript)

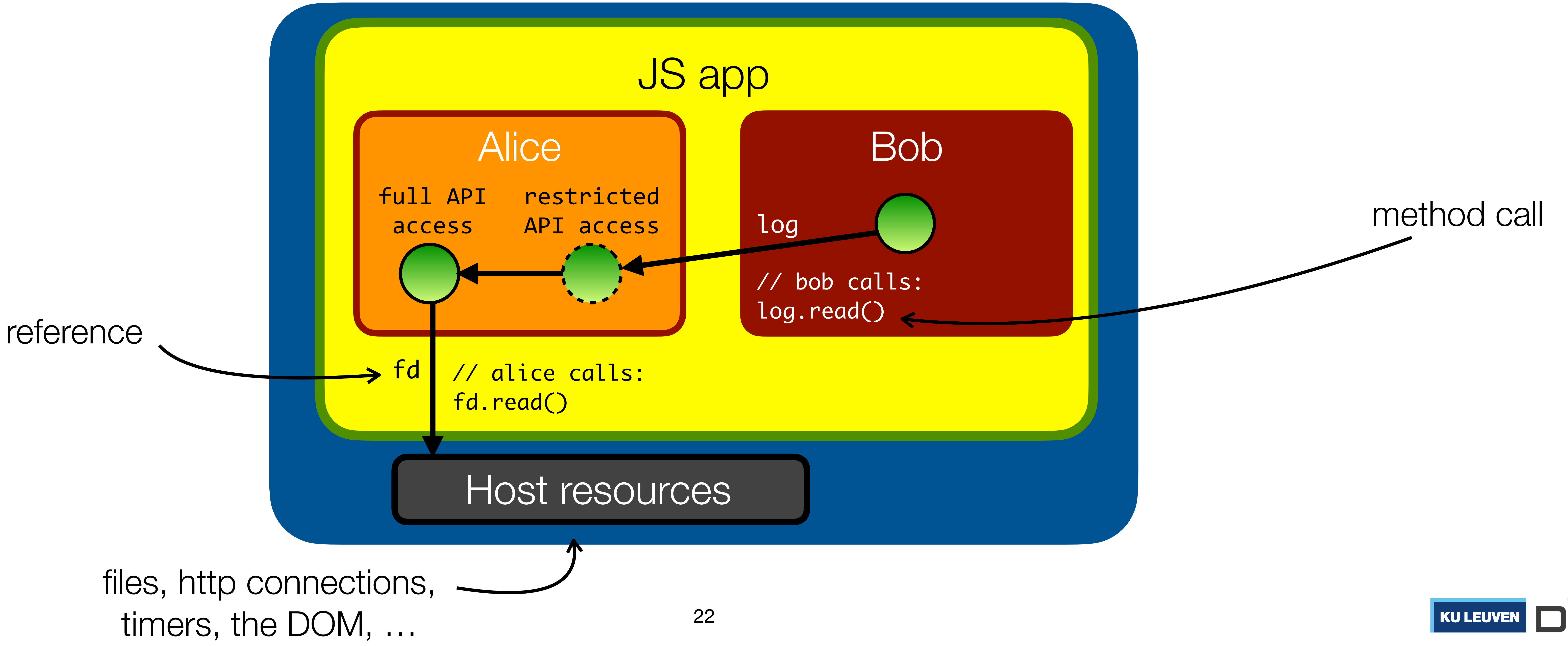
Principle of Least Authority (POLA)

- A module should only be given the authority it needs to do its job, and nothing more



What is “authority” in a JavaScript app?

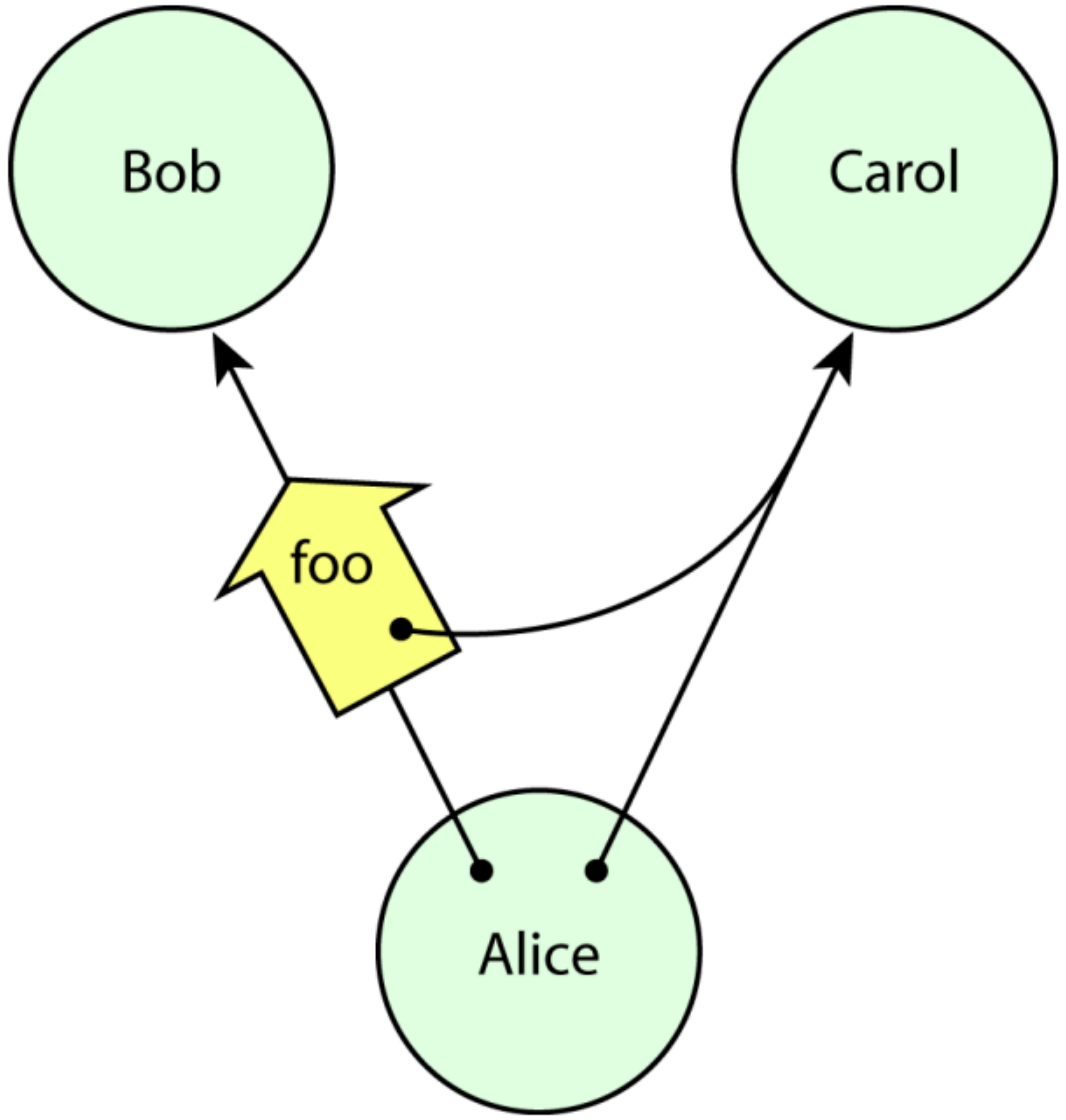
- Authority is linked to resources represented as objects (or functions)
- Objects can hold references (“pointers”) to resource objects
- The authority to use a resource is expressed by calling a method/function on a reference



```
// bob calls:
log.read()
```

```
// alice calls:
fd.read()
```

Delegating authority == sharing references, under the right assumptions



(source: Miller *et al.* “Capability myths demolished”, 2003)

Example: Alice wants to give Bob access to Carol, and *only* to Carol:

```
// alice calls:
bob.foo(carol)
```

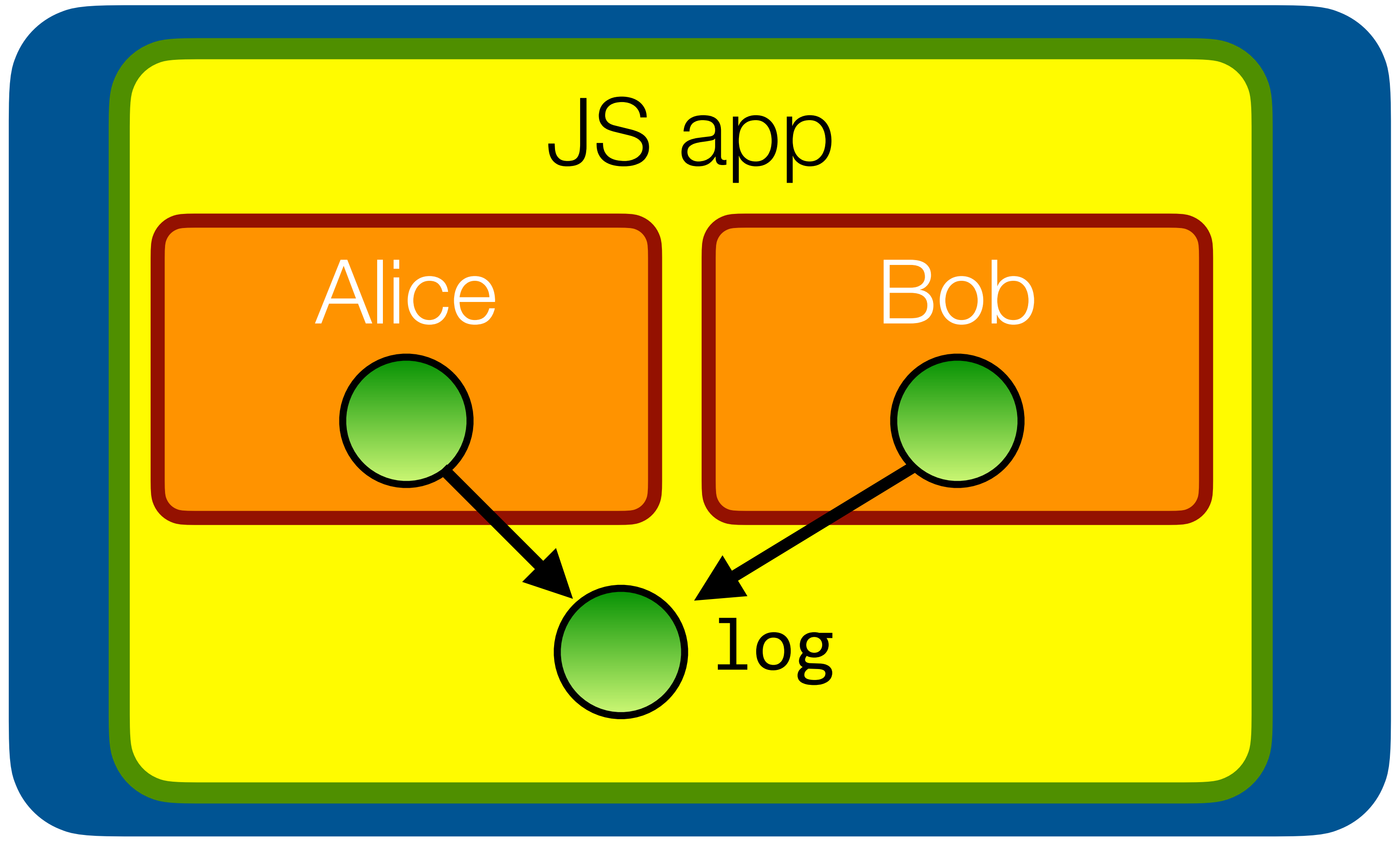
Assumptions:

- Pointers (references) are unforgeable
✅ JavaScript is **memory-safe**
- Pointers (references) can be privately stored
✅ JavaScript supports **hiding** access to private state through scoping rules
- There is no global mutable state
⚠️ Ensure that all exported objects/functions in a module are **immutable**
- There is no undeniable (“ambient”) authority
⚠️ Need to load each module in its **own**, initially empty, **global environment**

Running example: apply POLA to a basic shared log

We would like Alice to only **write** to the log, and Bob to only **read** from the log.

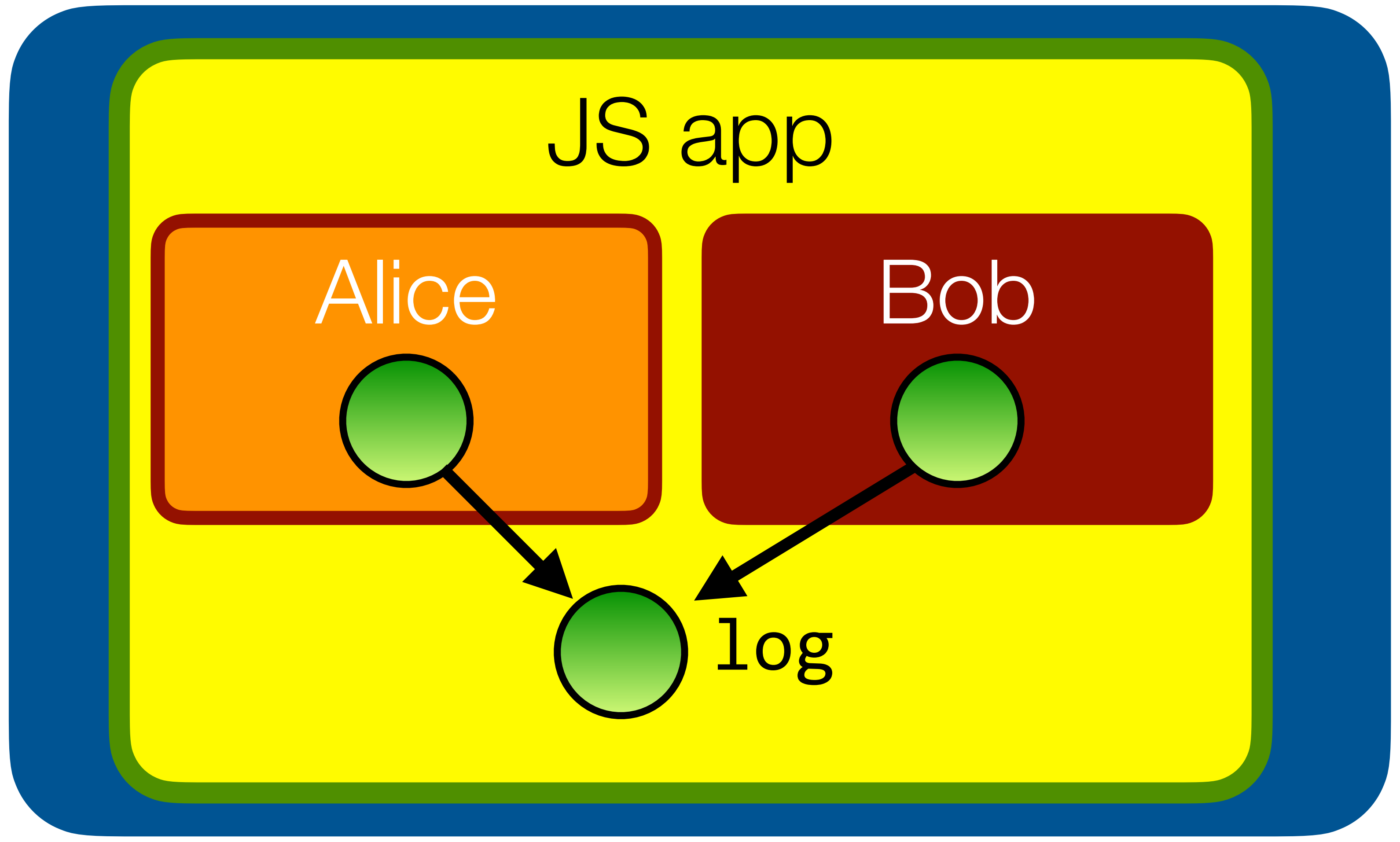
```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = new Log();
alice(log);
bob(log);
```



Running example: apply POLA to a basic shared log

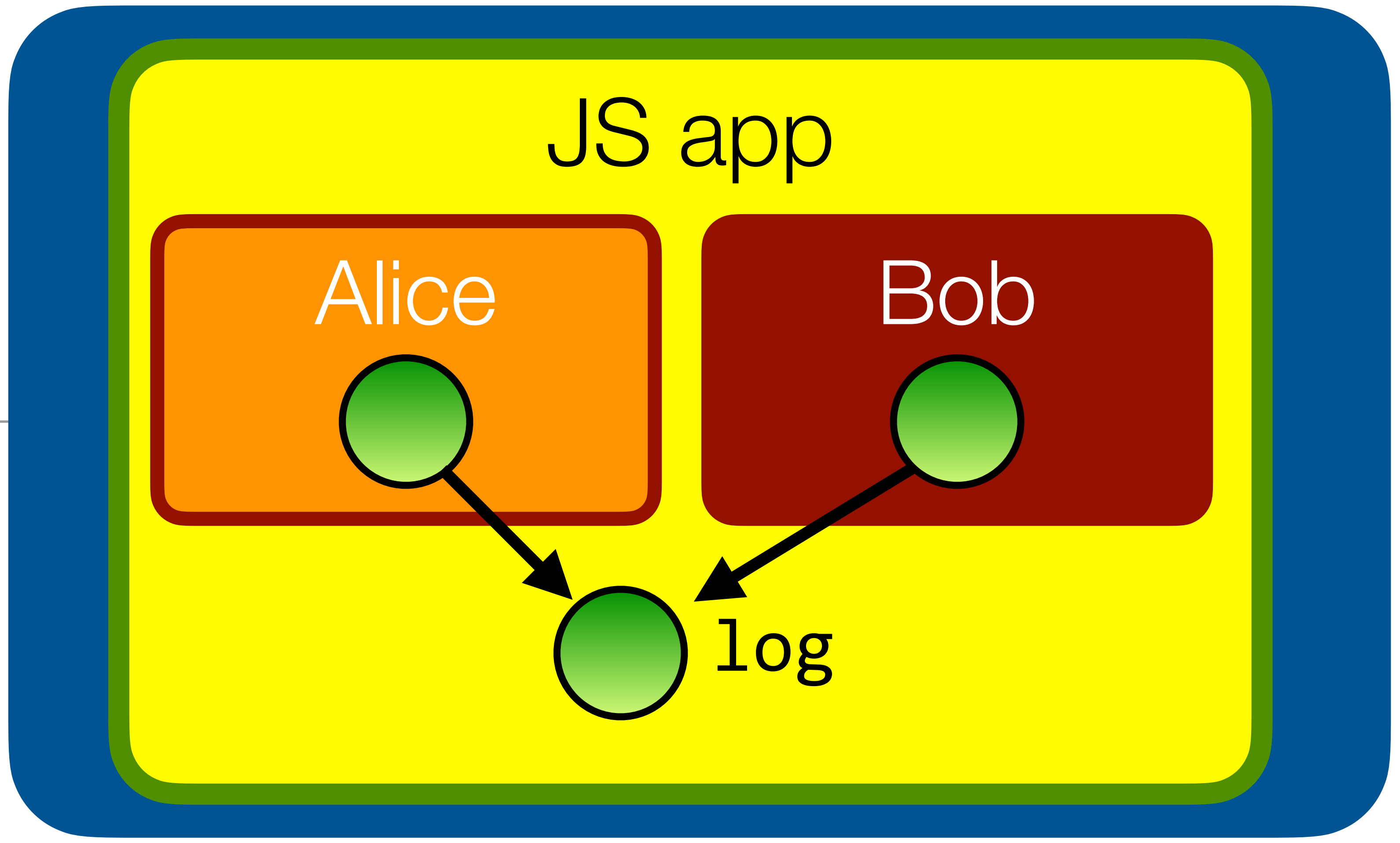
If Bob goes rogue, what could go wrong?

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = new Log();
alice(log);
bob(log);
```



Bob has way too much authority!

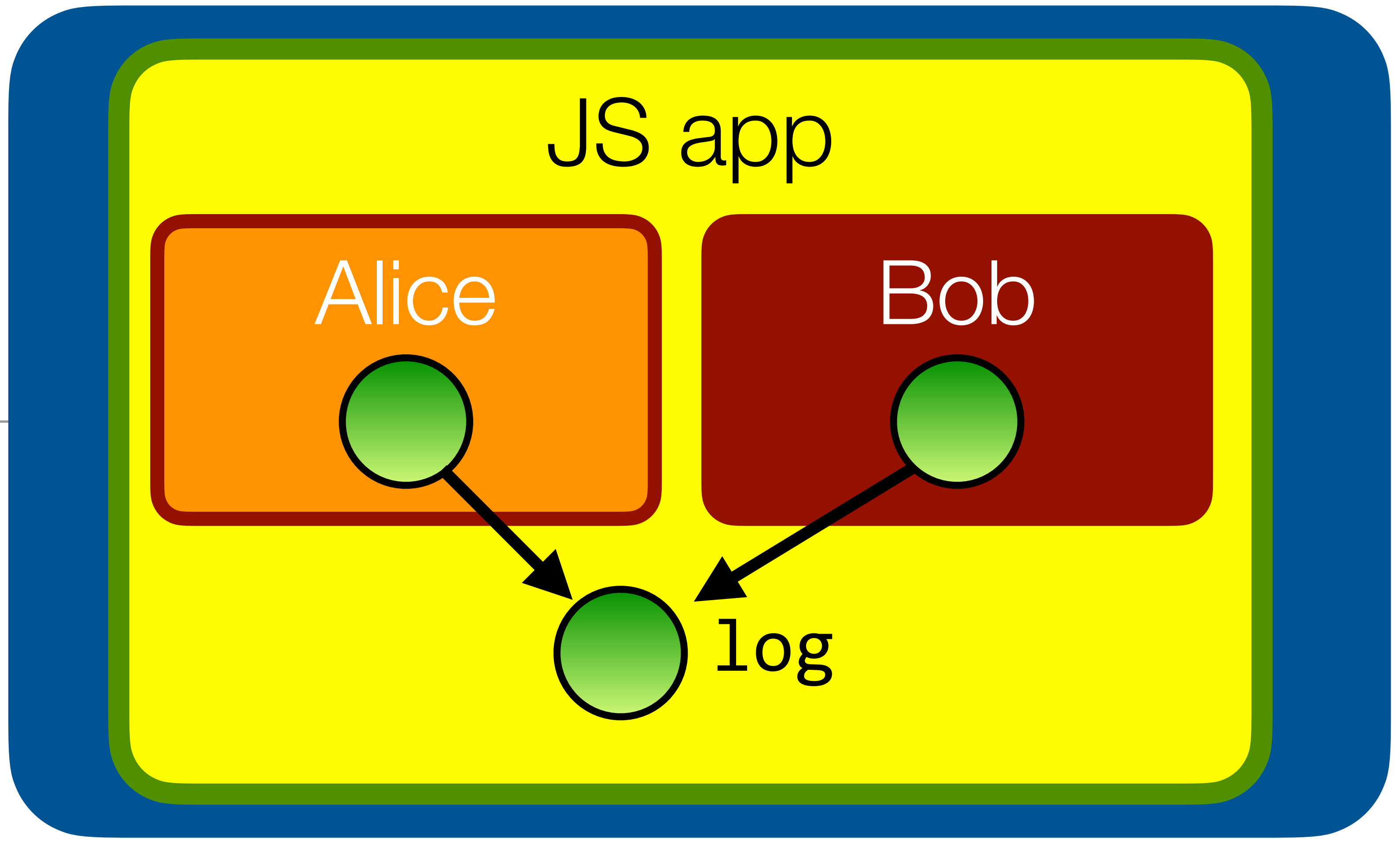
If Bob goes rogue, what could go wrong?



```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = new Log();
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0

// Bob can replace the ‘write’ function
log.write = function(msg) {
  console.log(“I’m not logging anything”);
}
// Bob can replace the Array built-ins
Array.prototype.push = function(msg) {
  console.log(“I’m not logging anything”);
}
```

How to solve “prototype poisoning” attacks?



Load each module in its own environment, with its own set of “primordial” objects

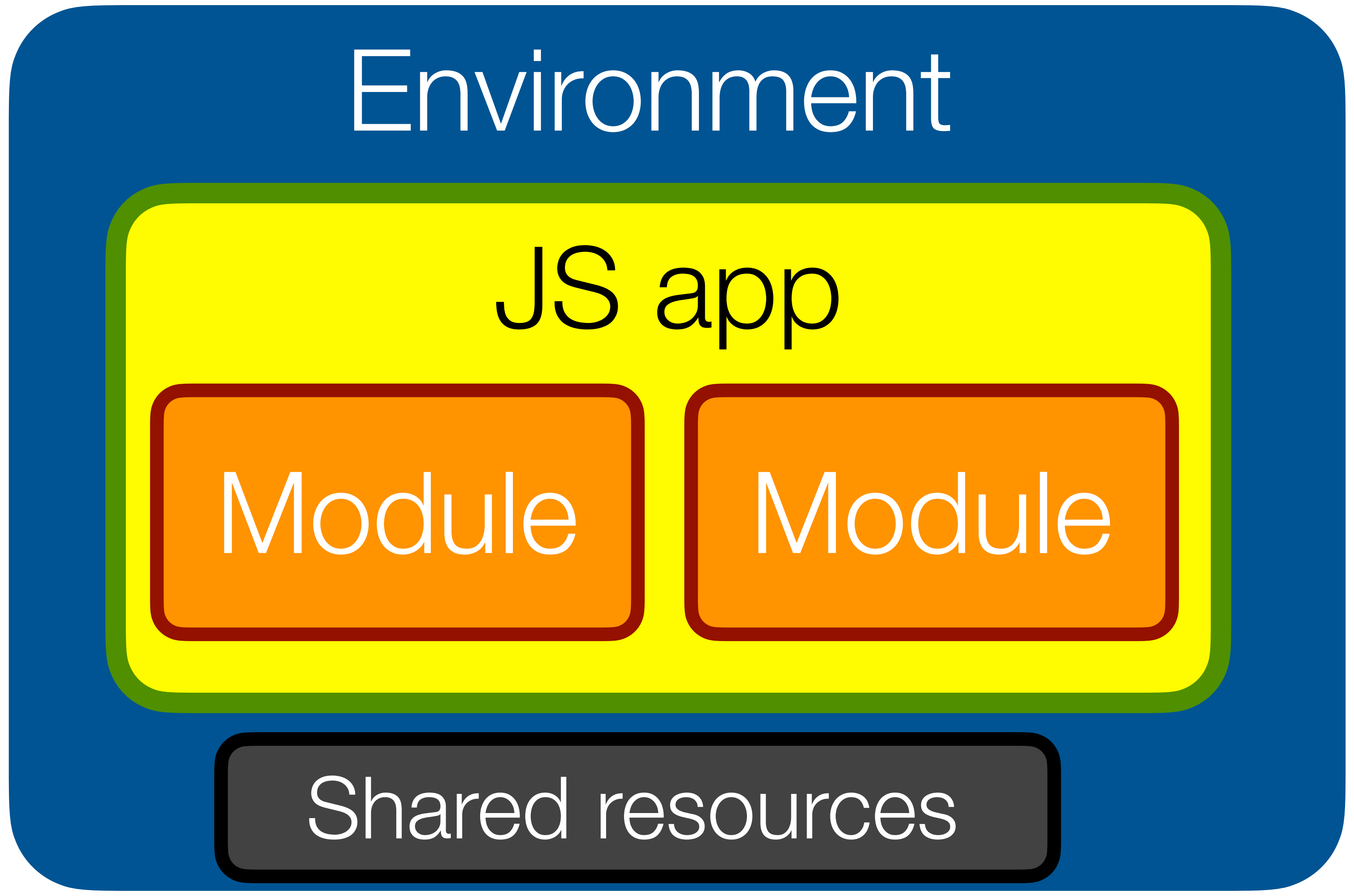
```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = new Log();
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0

// Bob can replace the ‘write’ function
log.write = function(msg) {
  console.log(“I’m not logging anything”);
}
```

```
// Bob can replace the Array built-ins
Array.prototype.push = function(msg) {
  console.log(“I’m not logging anything”);
}
```

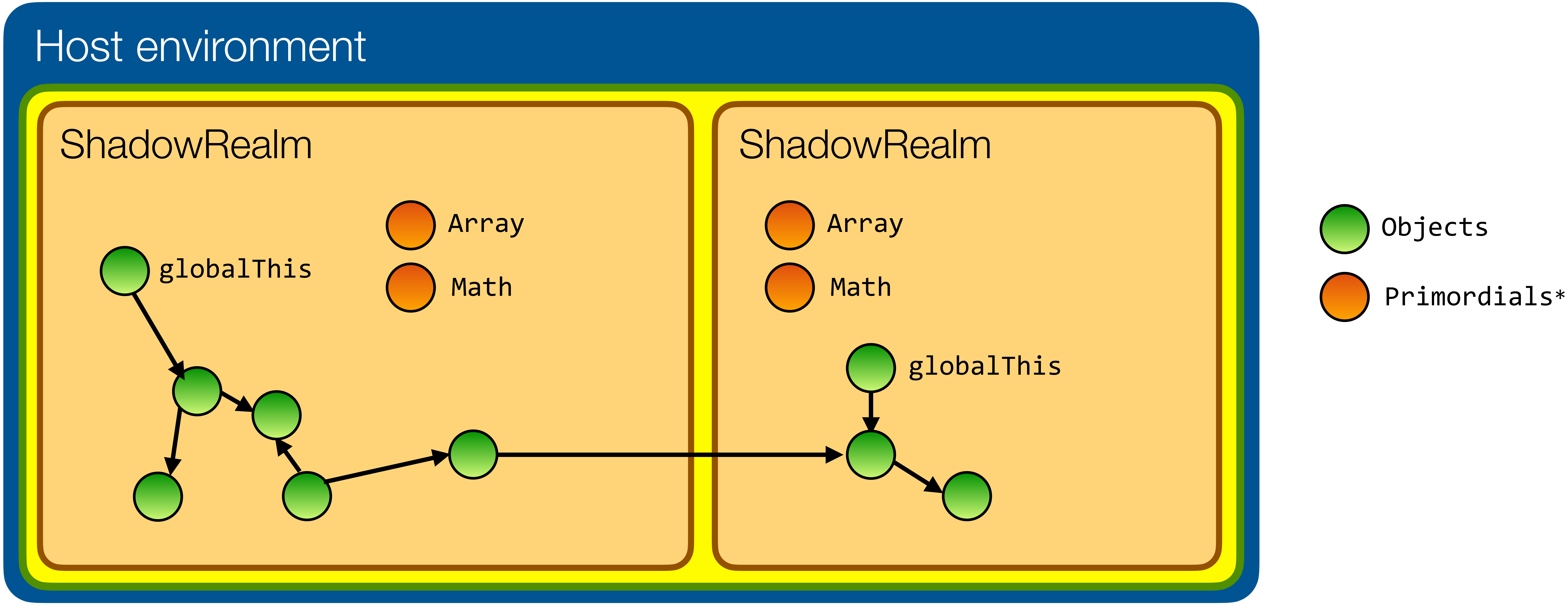
Prerequisite: isolating JavaScript modules

- Today: JavaScript offers no standard way to isolate a module (load it in a separate environment)
- Lots of host-specific isolation mechanisms, but non-portable and ill-defined:
  - **Web Workers**: no shared memory, can only communicate using message-passing
  - **iframes**: mutable primordials, “identity discontinuity”
  - **nodejs** `vm` **module**: same issues



ShadowRealms (ECMA TC39 Stage 2 proposal)

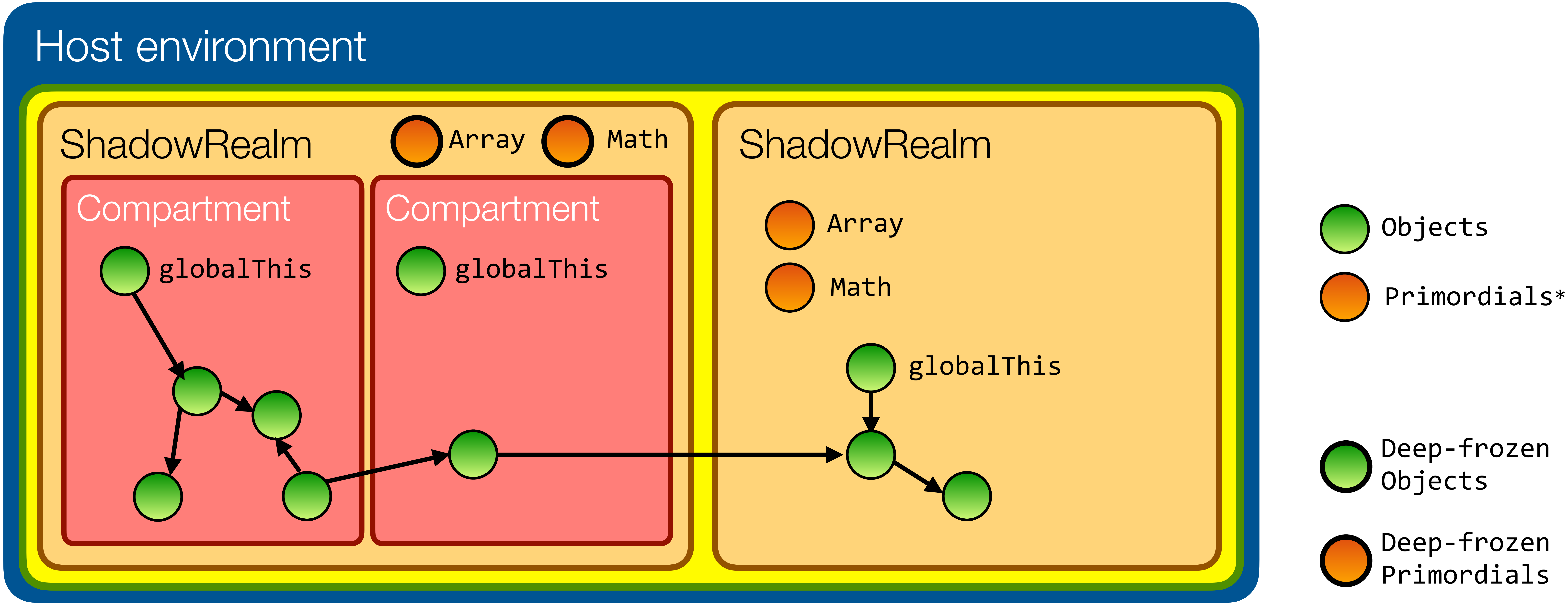
Intuitions: “iframe without DOM”, “principled version of node’s `vm` module”



* Primordials: built-in objects like `Object`, `Object.prototype`, `Array`, `Function`, `Math`, `JSON`, etc.

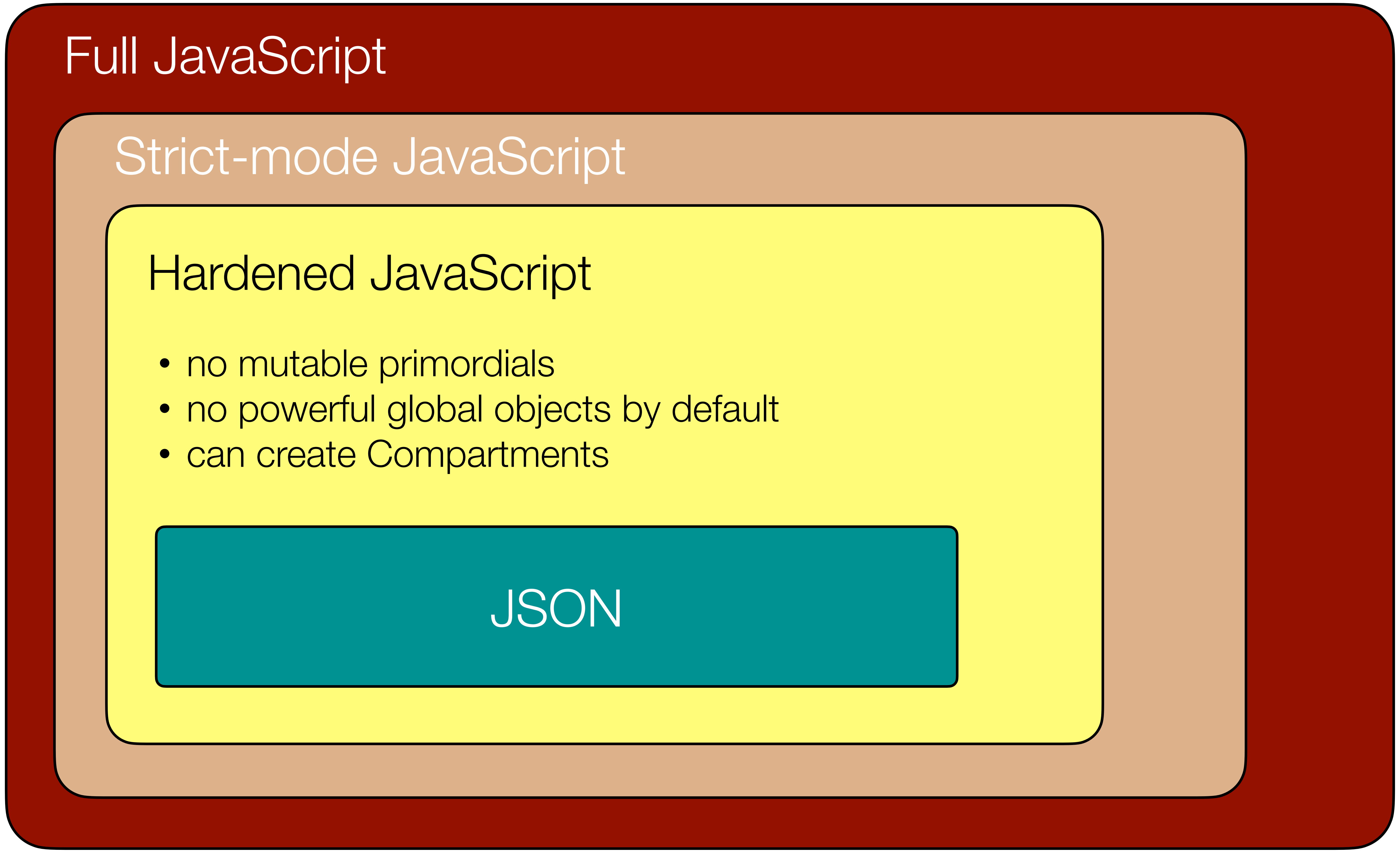
Compartments (ECMA TC39 Stage 1 proposal)

Each Compartment has its own global object but shared (immutable) primordials.



* Primordials: built-in objects like `Object`, `Object.prototype`, `Array`, `Function`, `Math`, `JSON`, etc.

Hardened JavaScript is a secure subset of standard JavaScript



- no mutable primordials
- no powerful global objects by default
- can create Compartments

(inspired by the diagram at https://github.com/Agoric/Jessie)

Key idea: code running in hardened JS can only affect the outside world through objects (capabilities) explicitly granted to it from outside.

Hardened JavaScript: some history

Google develops a project called “**Caja**” for **safe embedding** of dynamic web content (JavaScript scripts) in web pages

2009 Google Caja

Attempts are made to **standardize** core features that enable secure sandboxing as “**Secure ECMAScript**” (SES) at ECMA TC39

2015 TC 39

Standardisation process got stalled, but work continued on a modified node.js runtime called “**endo**”, supporting SES on the server

2018 endo

A company called Agoric **rebrands** SES **to “Hardened JavaScript”**, works with Moddable and Metamask on implementation and tooling

2020 AGORIC

HardenedJS is **used by several companies** to isolate JavaScript modules for IoT (Moddable), Web3 (Agoric), SaaS (Salesforce), …

Today moddable salesforce

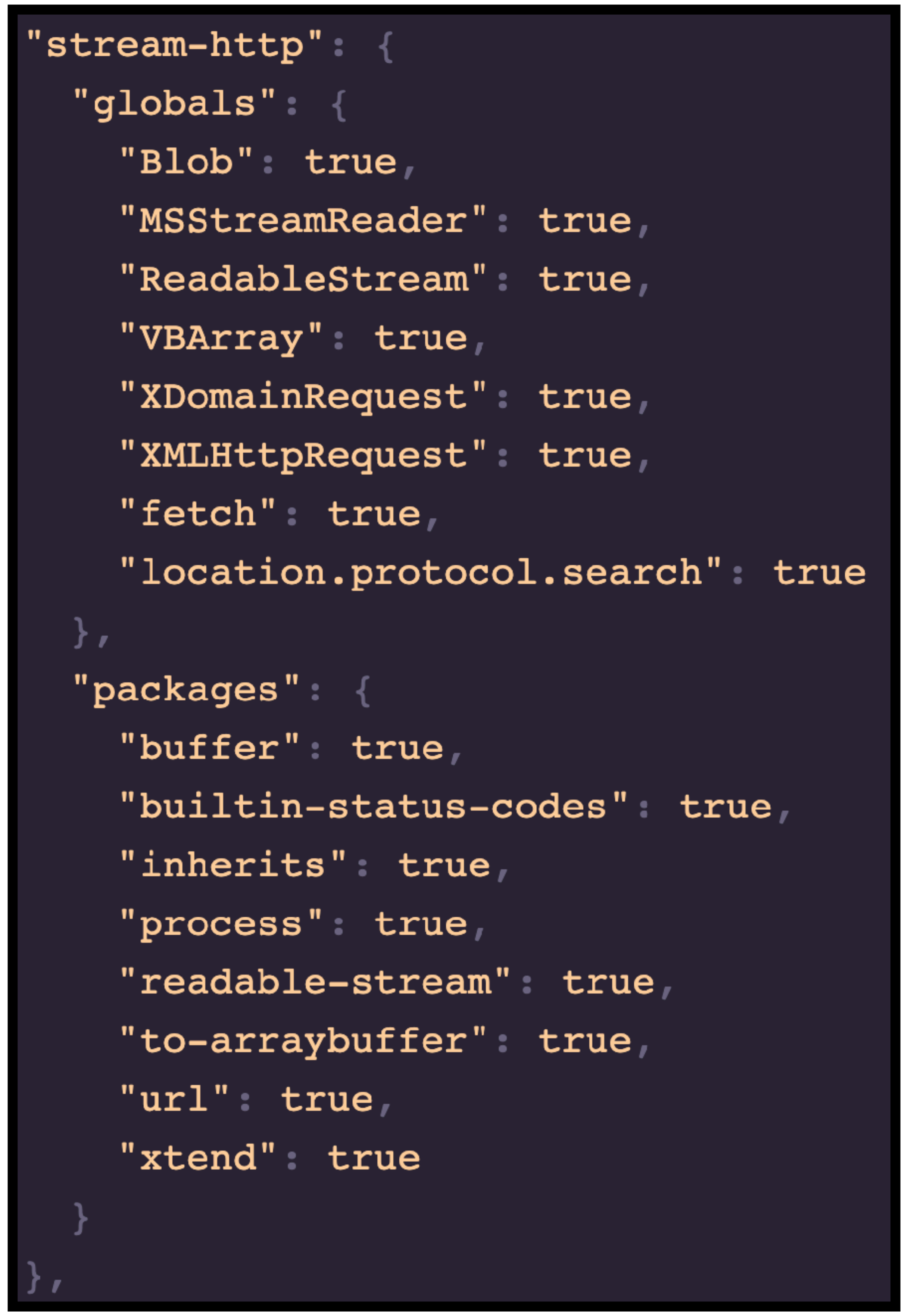
LavaMoat

- CLI tool that puts each package dependency into its own hardened JS sandbox environment
- Auto-generates config file indicating authority needed by each package
- Plugs into build tools like Webpack and Browserify

```
npm install -D lavamoat
npx lavamoat app.js --autopolicy
```

https://github.com/LavaMoat/lavamoat

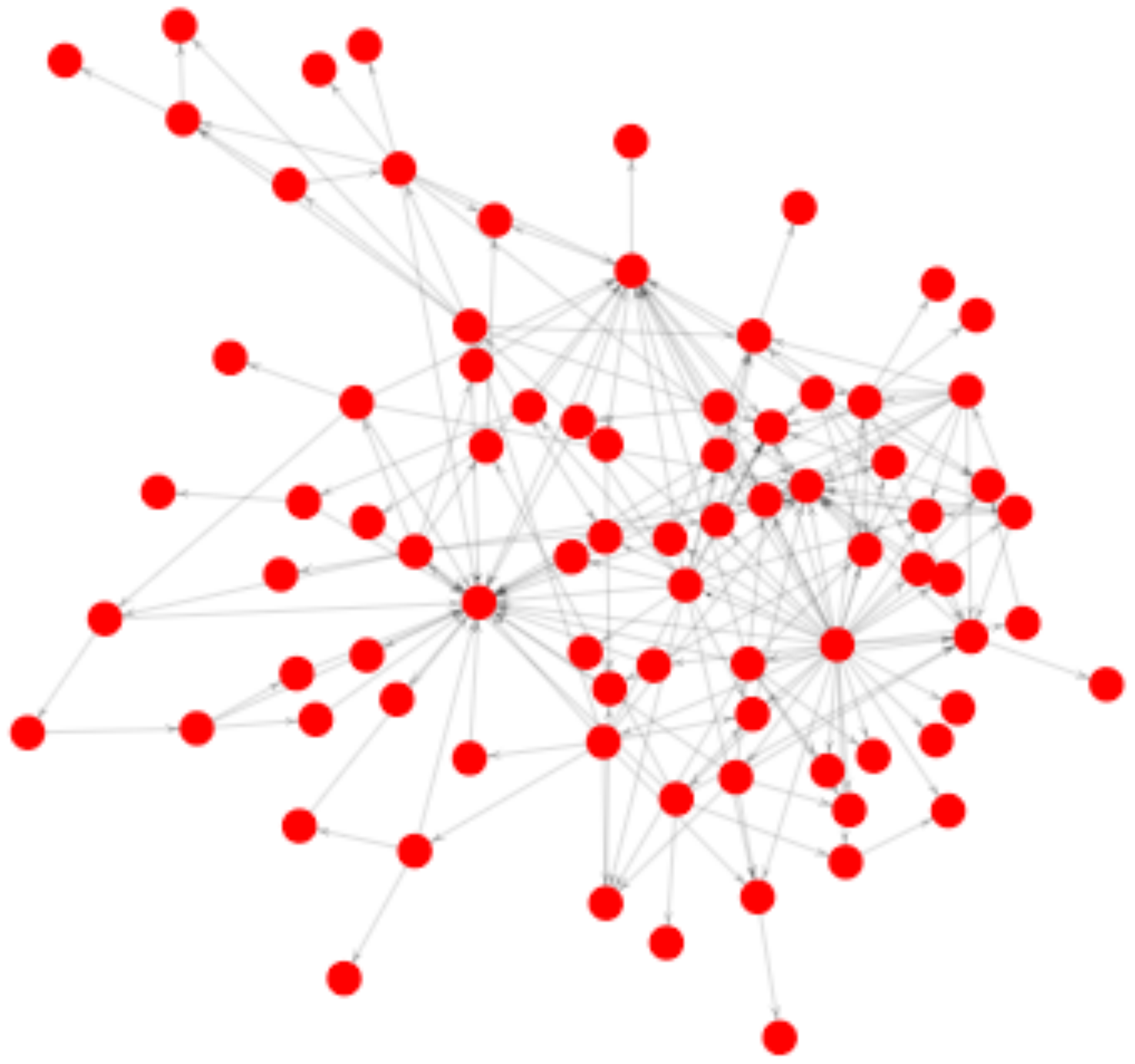
METAMASK



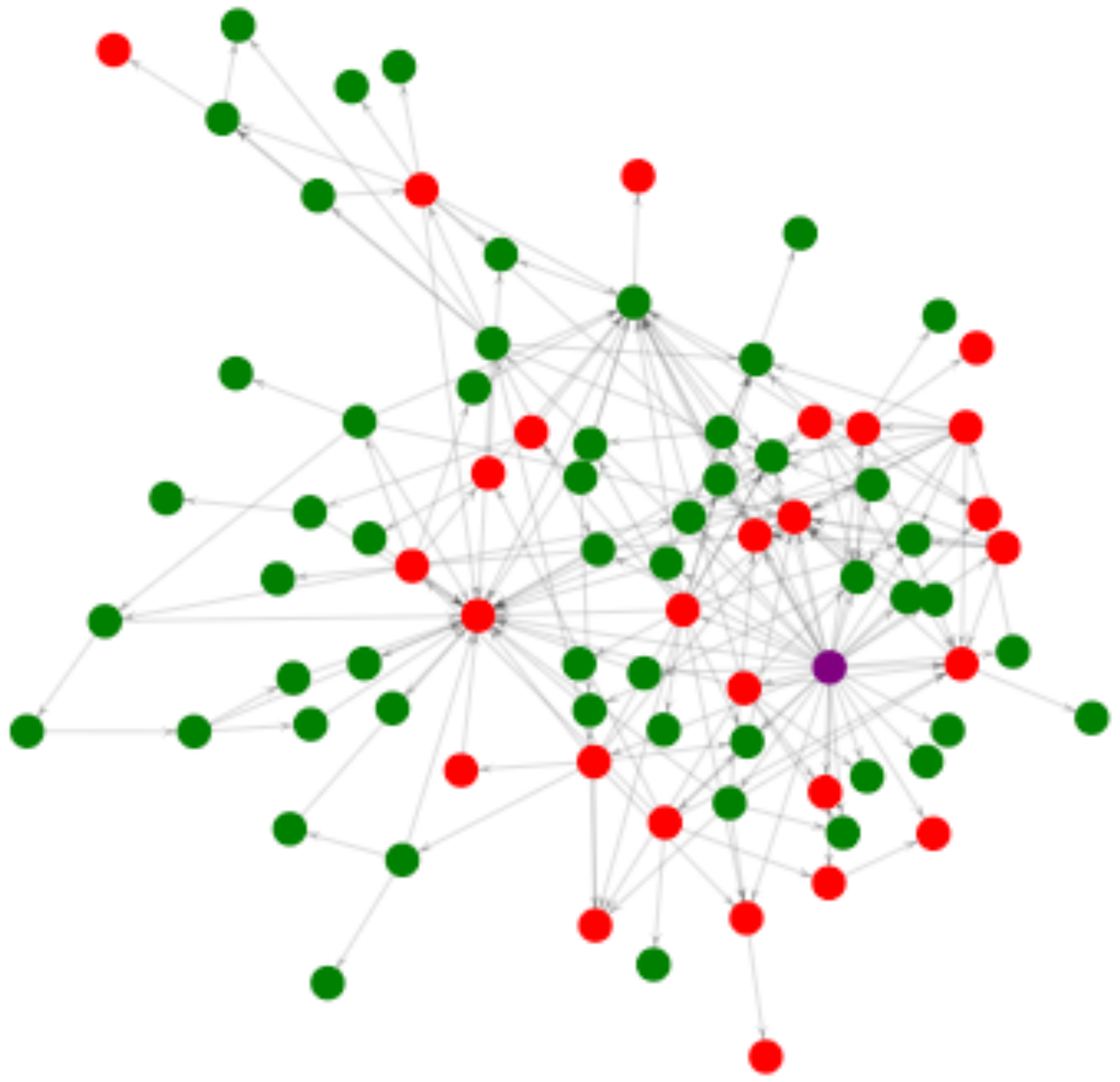
```
"stream-http": {
  "globals": {
    "Blob": true,
    "MSStreamReader": true,
    "ReadableStream": true,
    "VBArray": true,
    "XDomainRequest": true,
    "XMLHttpRequest": true,
    "fetch": true,
    "location.protocol.search": true
  },
  "packages": {
    "buffer": true,
    "builtin-status-codes": true,
    "inherits": true,
    "process": true,
    "readable-stream": true,
    "to-arraybuffer": true,
    "url": true,
    "xtend": true
  }
},
```

LavaMoat enables more focused security reviews

Exposure to package dependencies
without LavaMoat sandboxing



Exposure to package dependencies
with LavaMoat sandboxing



https://github.com/LavaMoat/lavamoat

Bonus: avoiding unwanted post-install scripts

- Package managers like `npm` allow packages to run install scripts

- A compromised dependency can exploit this to run code as part of your project installation script

- Lavamoat's `allow-scripts` tool configures your project to disable running install scripts by default

- Edit allowed packages in `package.json`

- New install scripts entering your dependency tree will no longer run automatically unless approved

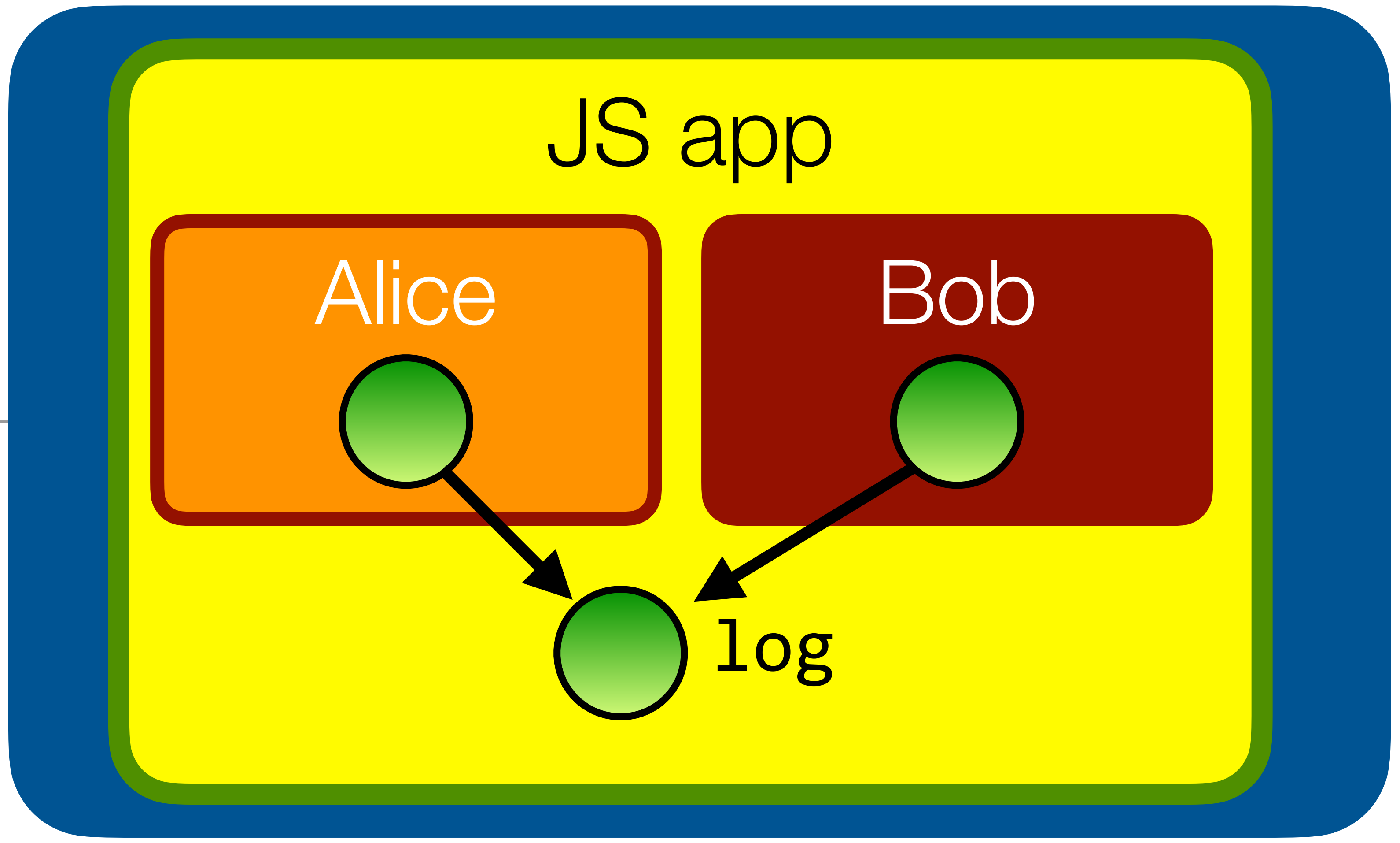
```
npm install -D @lavamoat/allow-scripts
npx --no-install allow-scripts auto
```

```
// in package.json
{
  "lavamoat": {
    "allowScripts": {
      "keccak": true,
      "core-js": false
    }
  }
}
```

https://www.npmjs.com/package/@lavamoat/allow-scripts

Back to our example

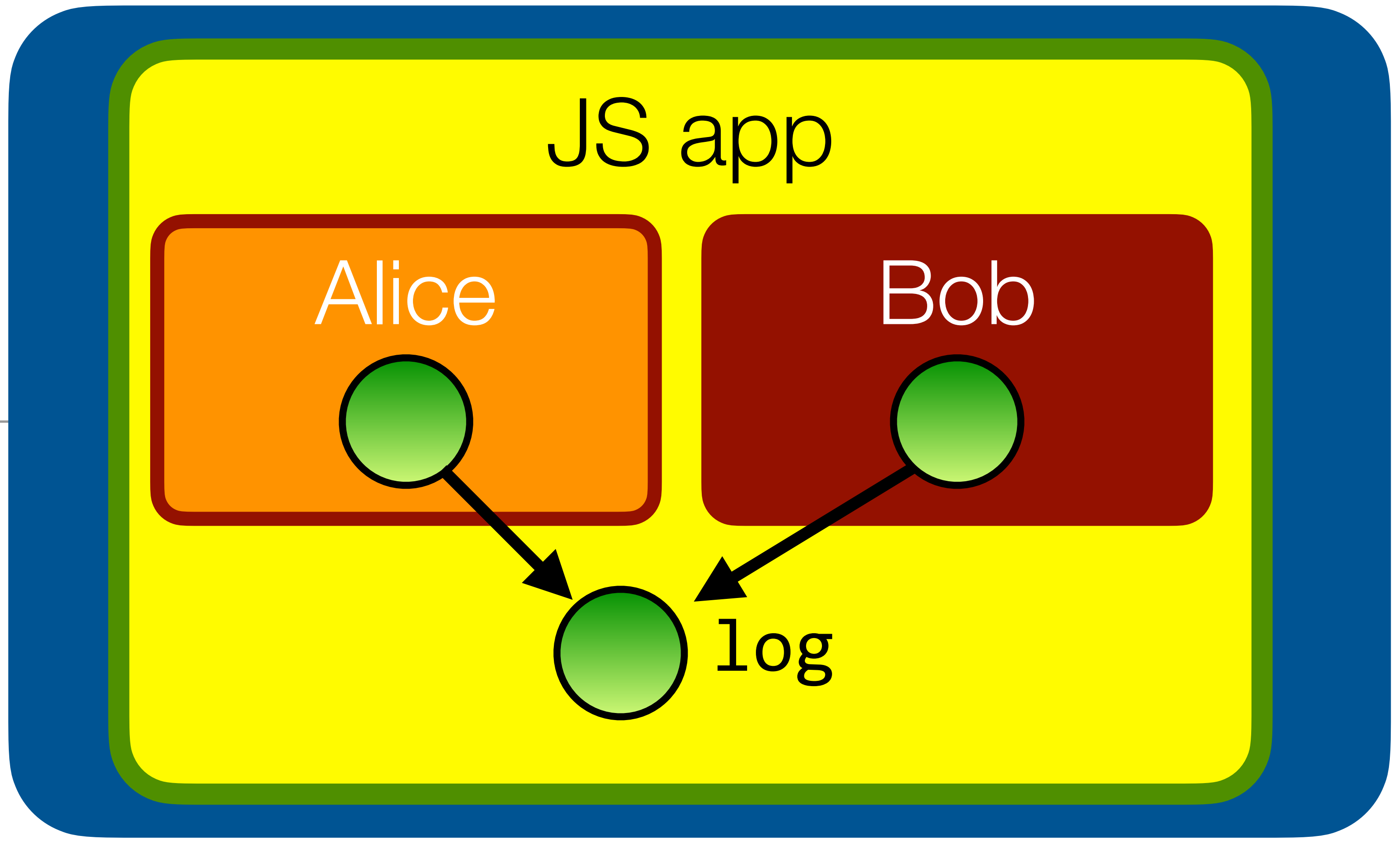
With Alice and Bob’s code running in their own Compartment, we mitigate the poisoning attack



```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = new Log();
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0

// Bob can replace the ‘write’ function
log.write = function(msg) {
  console.log(“I’m not logging anything”);
}
~~// Bob can replace the Array built-ins~~
~~Array.prototype.push = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~
```

One down, three to go



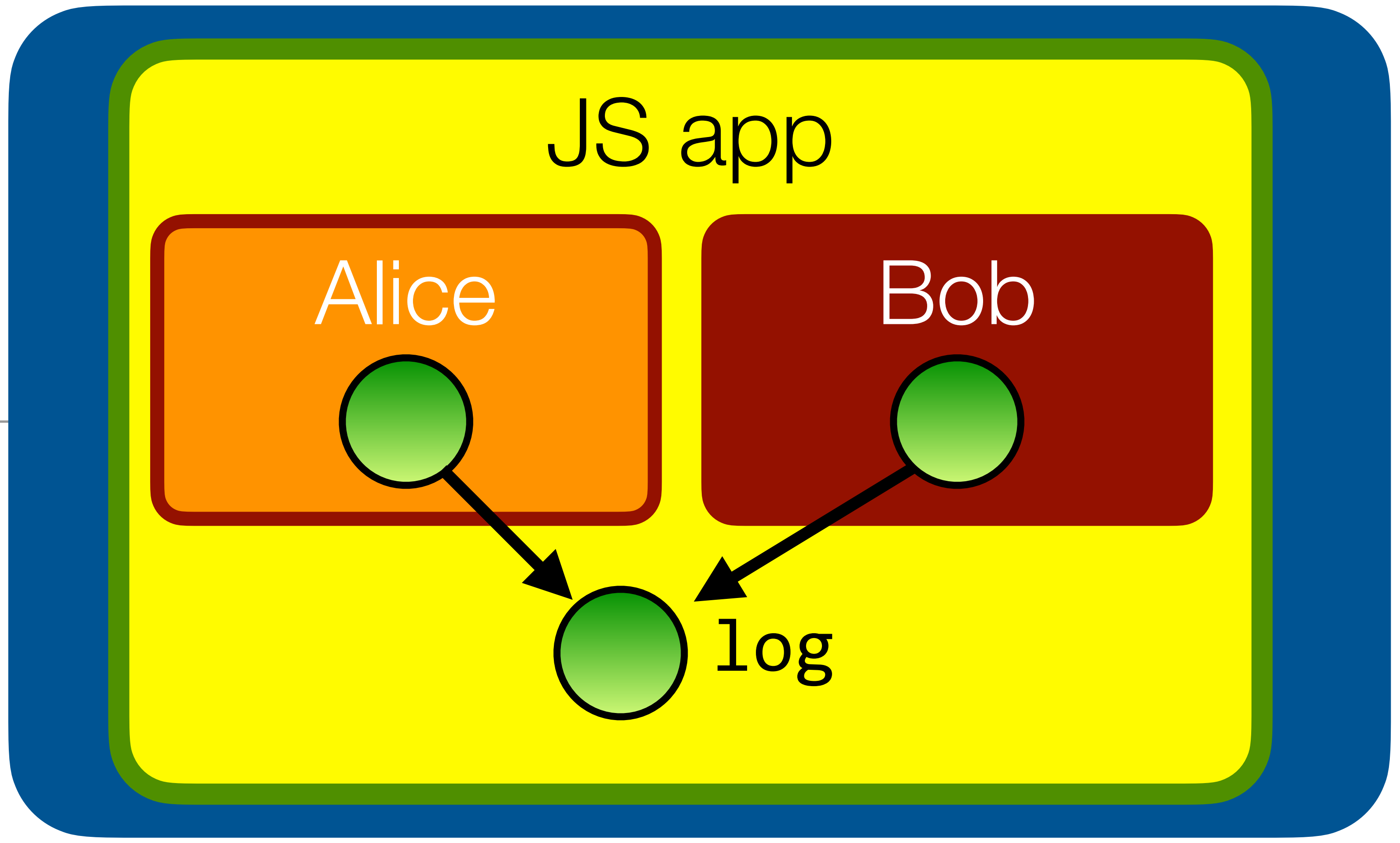
POLA: we would like Alice to only write to the log, and Bob to only read from the log.

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = new Log();
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0
```

```
// Bob can replace the ‘write’ function
log.write = function(msg) {
  console.log(“I’m not logging anything”);
}
```

Make the log’s interface **tamper-proof**



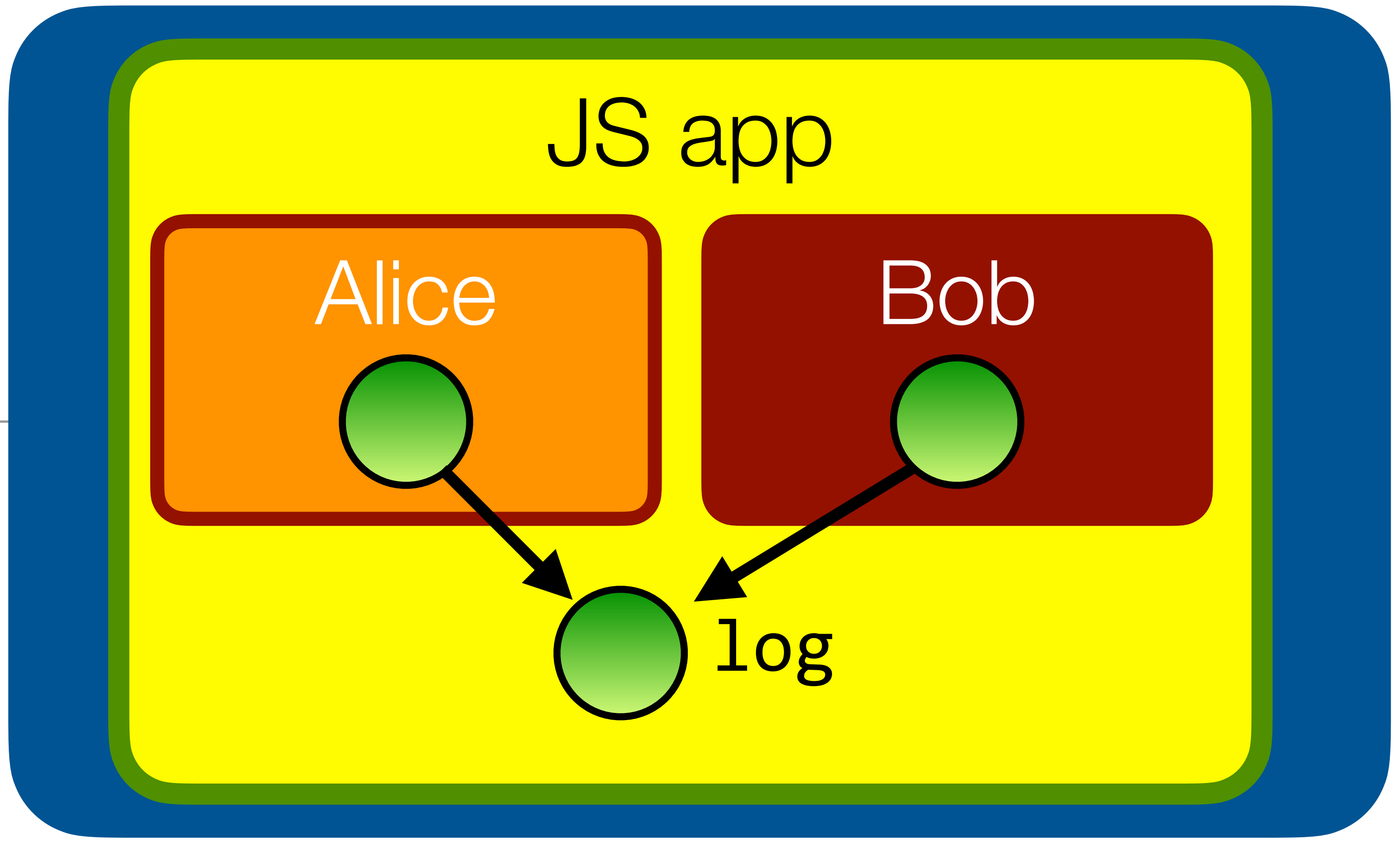
Object.freeze makes property bindings (not their values) immutable

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = Object.freeze(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0

// Bob can replace the ‘write’ function
log.write = function(msg) {
  console.log(“I’m not logging anything”);
}
```

Make the log’s interface tamper-proof. Oops.

Functions are mutable too. Freeze doesn’t recursively freeze the object’s functions.



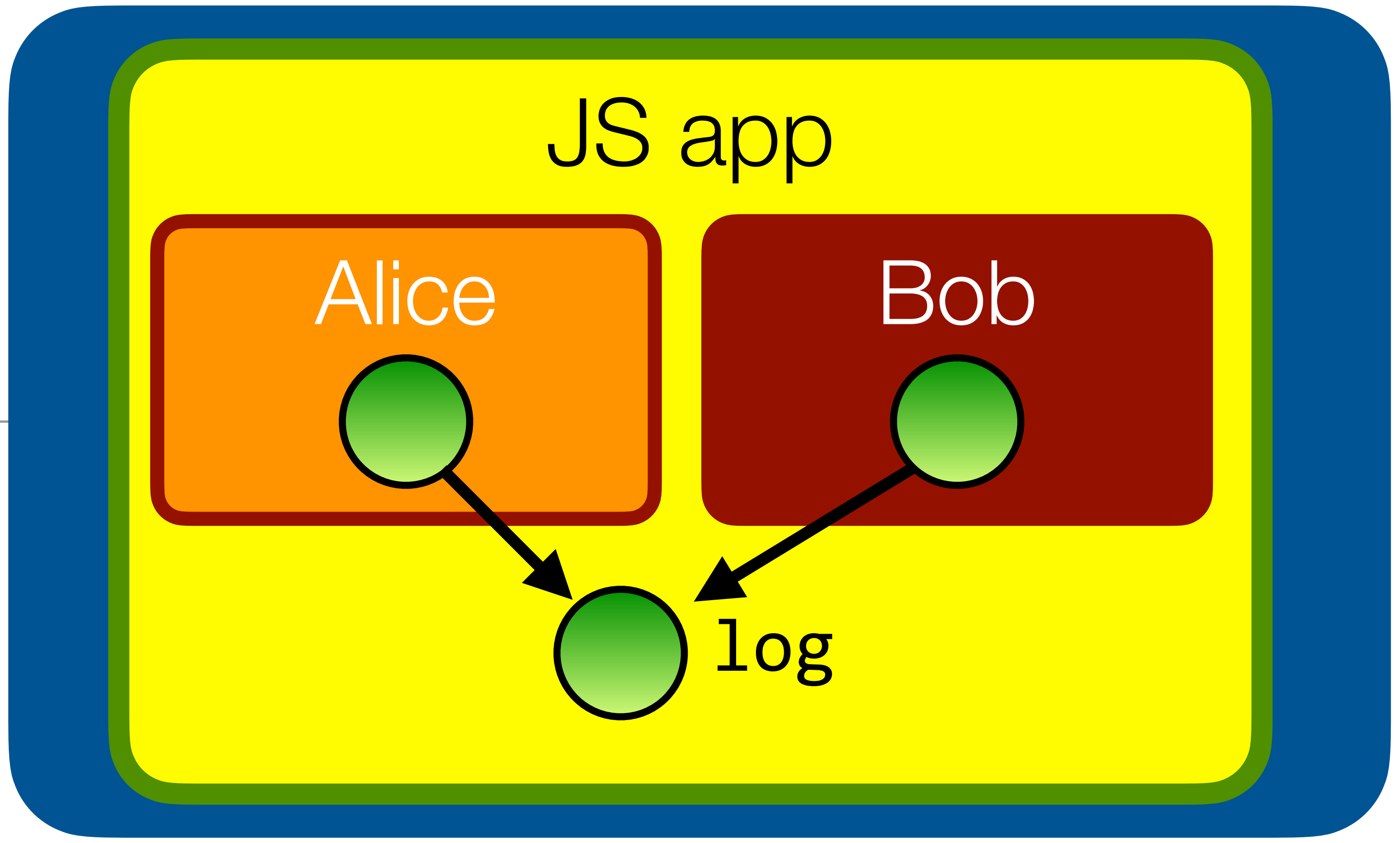
```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = Object.freeze(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0
```

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~ console.log(“I’m not logging anything”);~~
~~}~~

```
// Bob can still modify the write function
log.write.apply = function() { “gotcha” };
```

Make the log’s interface tamper-proof

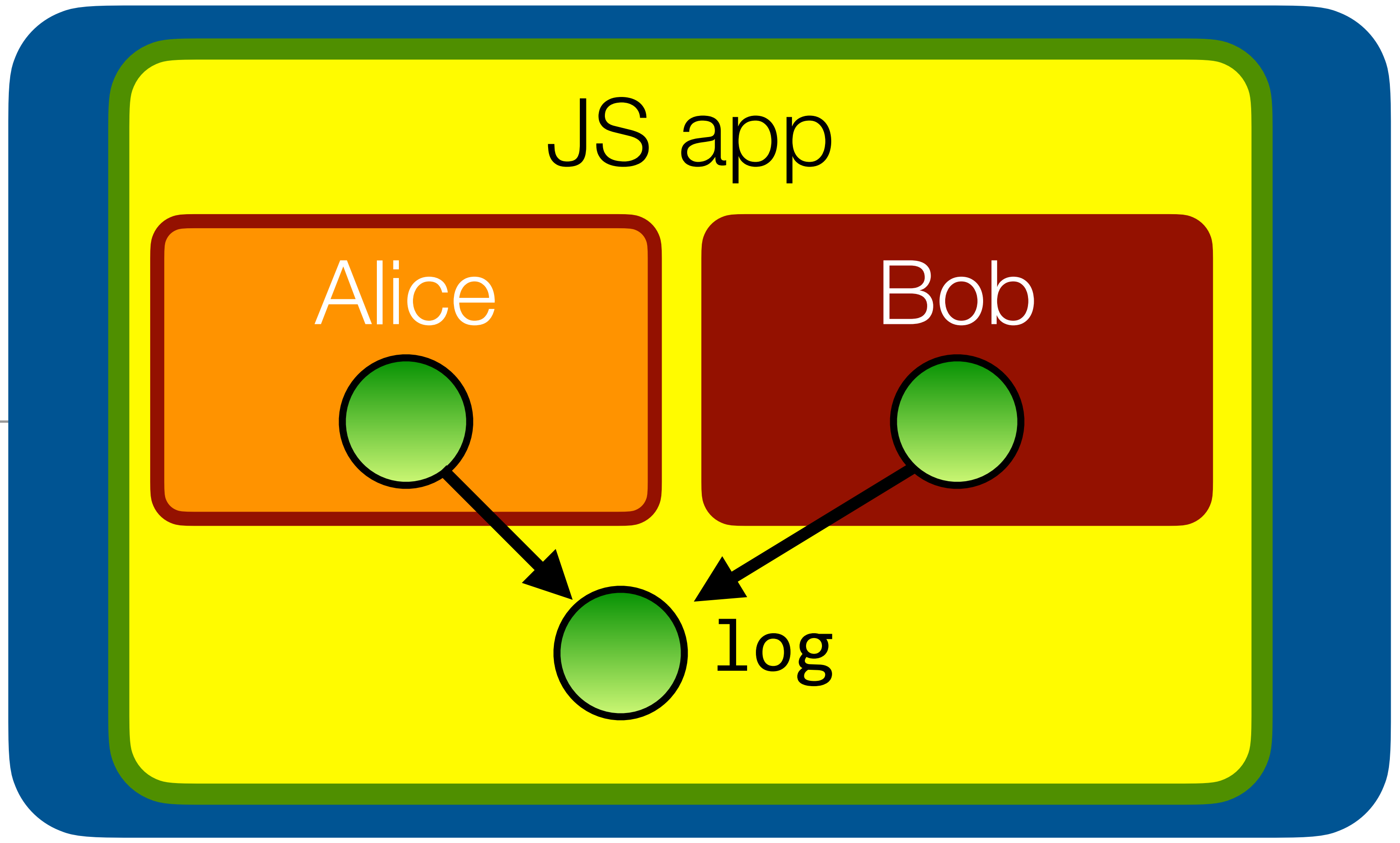


Hardened JavaScript provides a `harden` function that “deep-freezes” an object

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = harden(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~

// Bob can still modify the write function
log.write.apply = function() { “gotcha” };
```

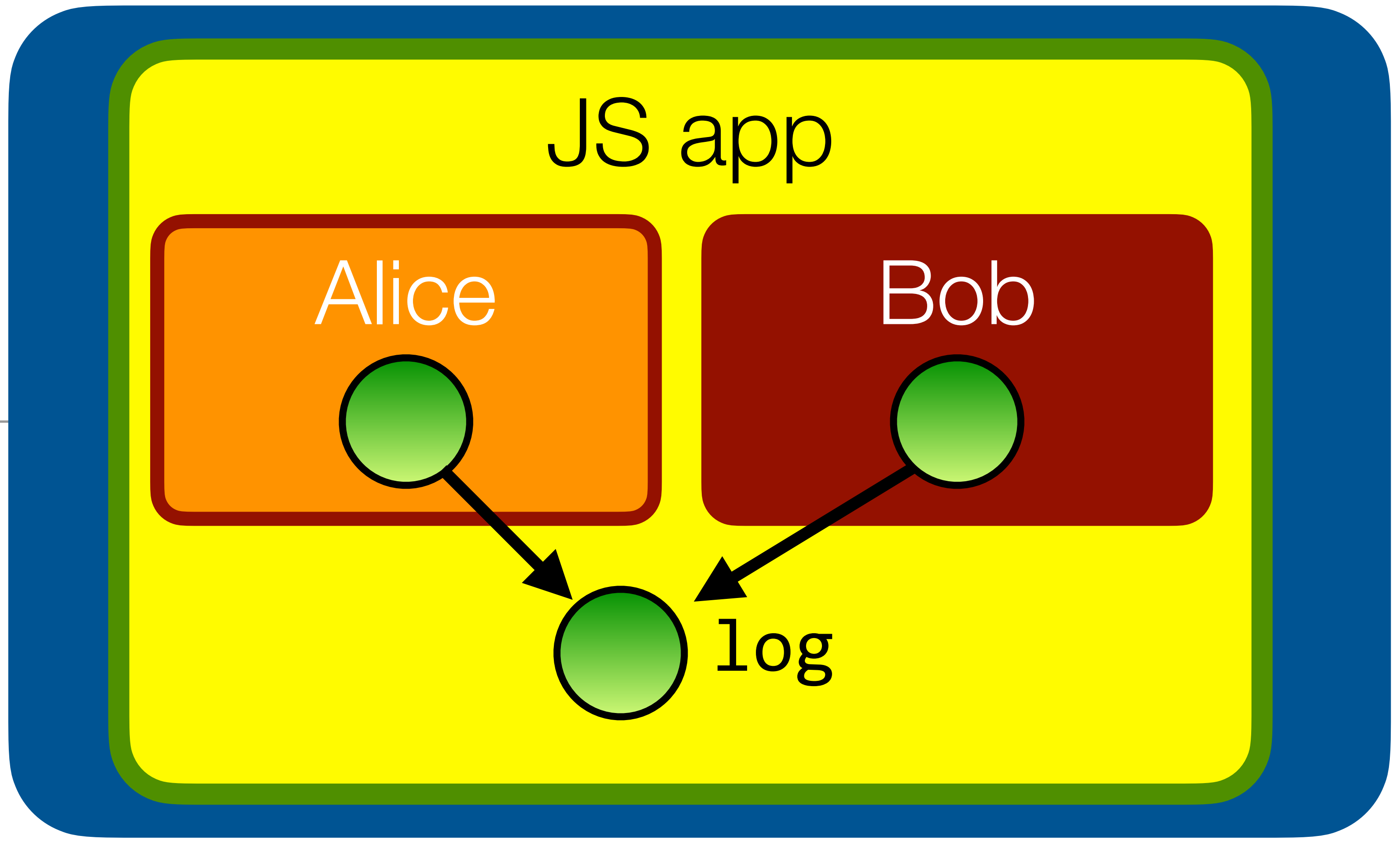
Two down, two to go



```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = harden(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~
```

Two down, two to go



```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return this.messages_; }
}

let log = harden(new Log());
alice(log);
bob(log);
```

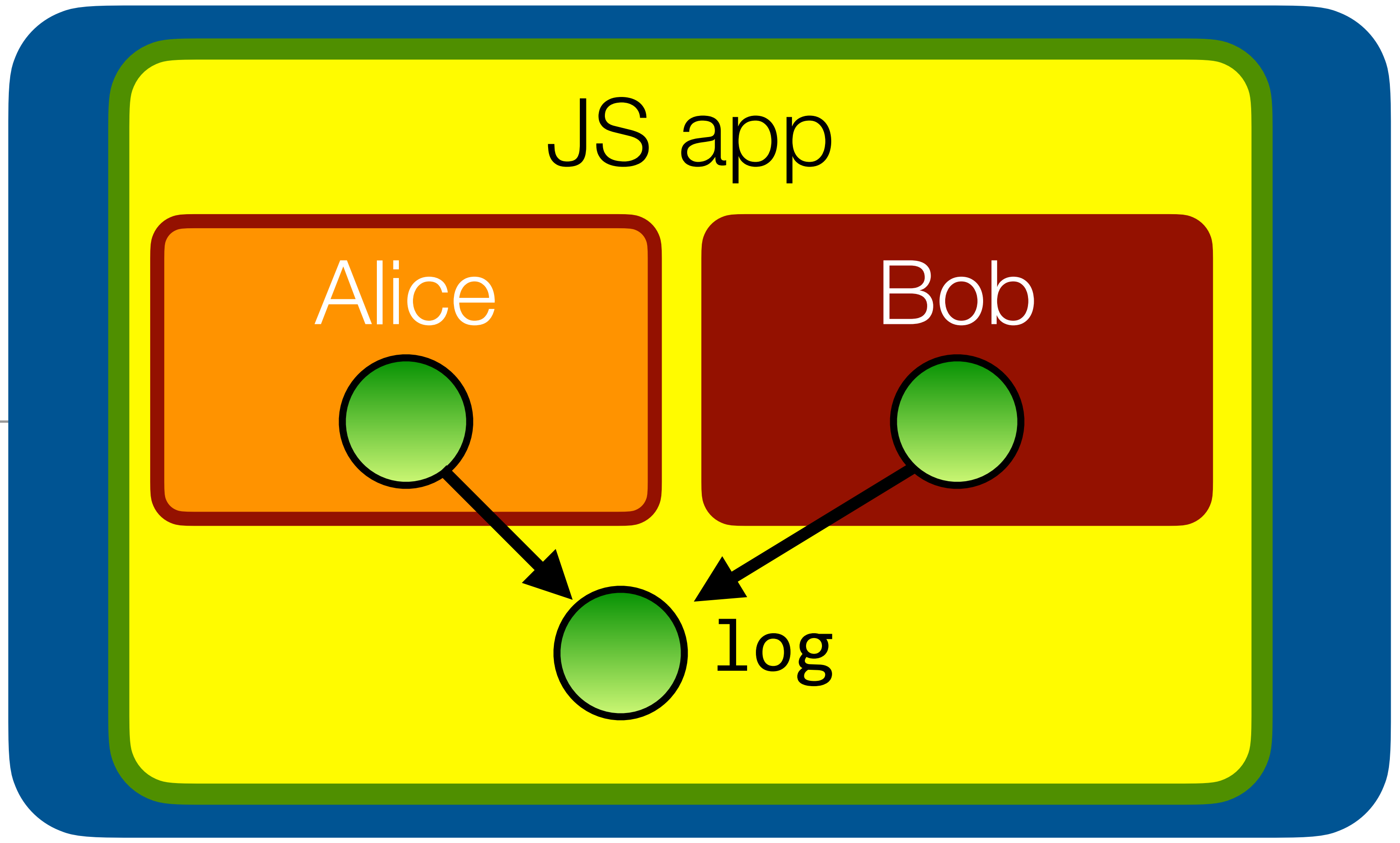
```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)
```

```
// Bob can delete the entire log
log.read().length = 0
```

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~

Don’t share access to mutable internals



- Modify `read()` to return a copy of the mutable state.
- Even better would be to use a more efficient copy-on-write or “immutable” data structure (see immutable-js.com)

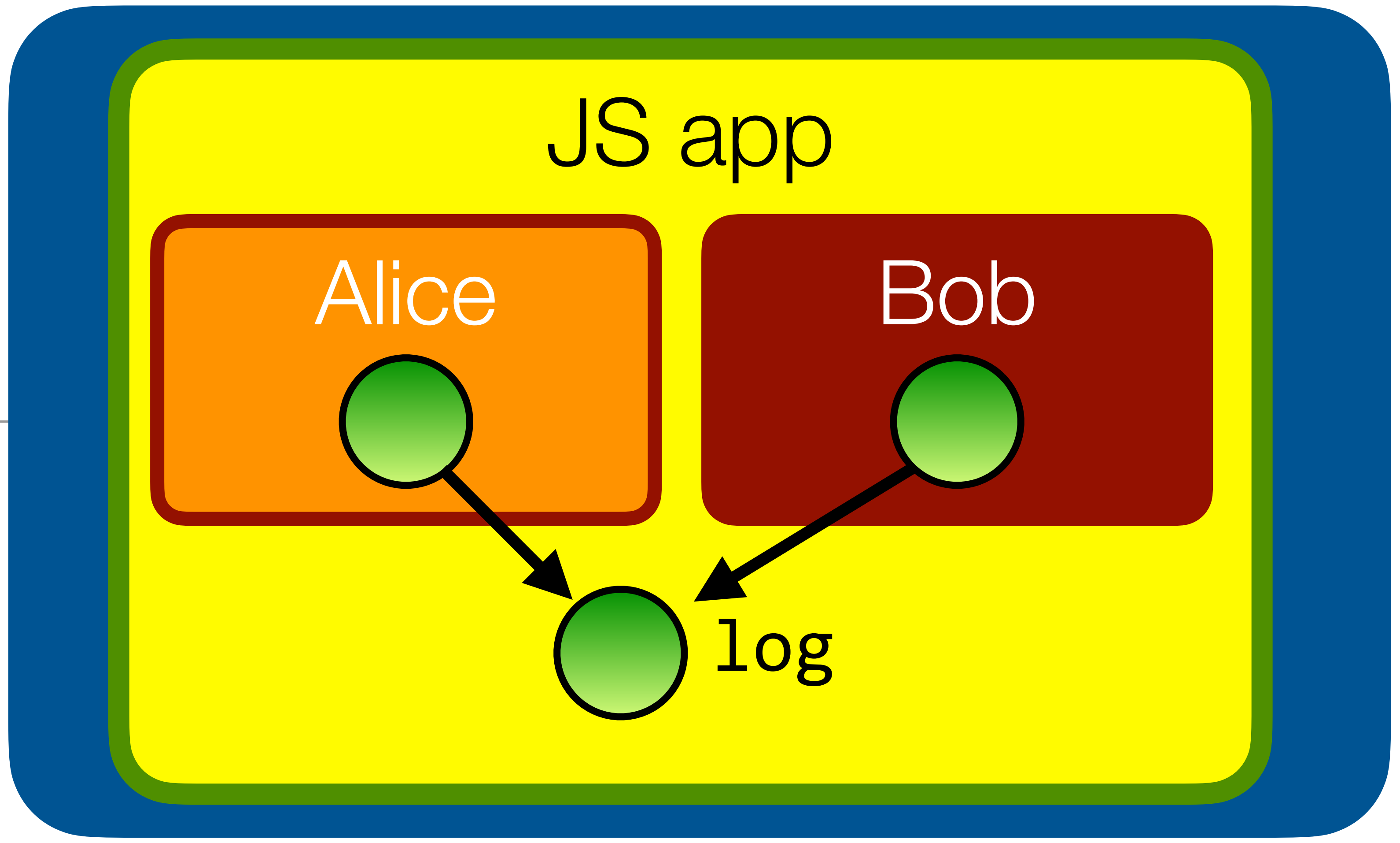
```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = harden(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

// Bob can delete the entire log
log.read().length = 0
```

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~ console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~

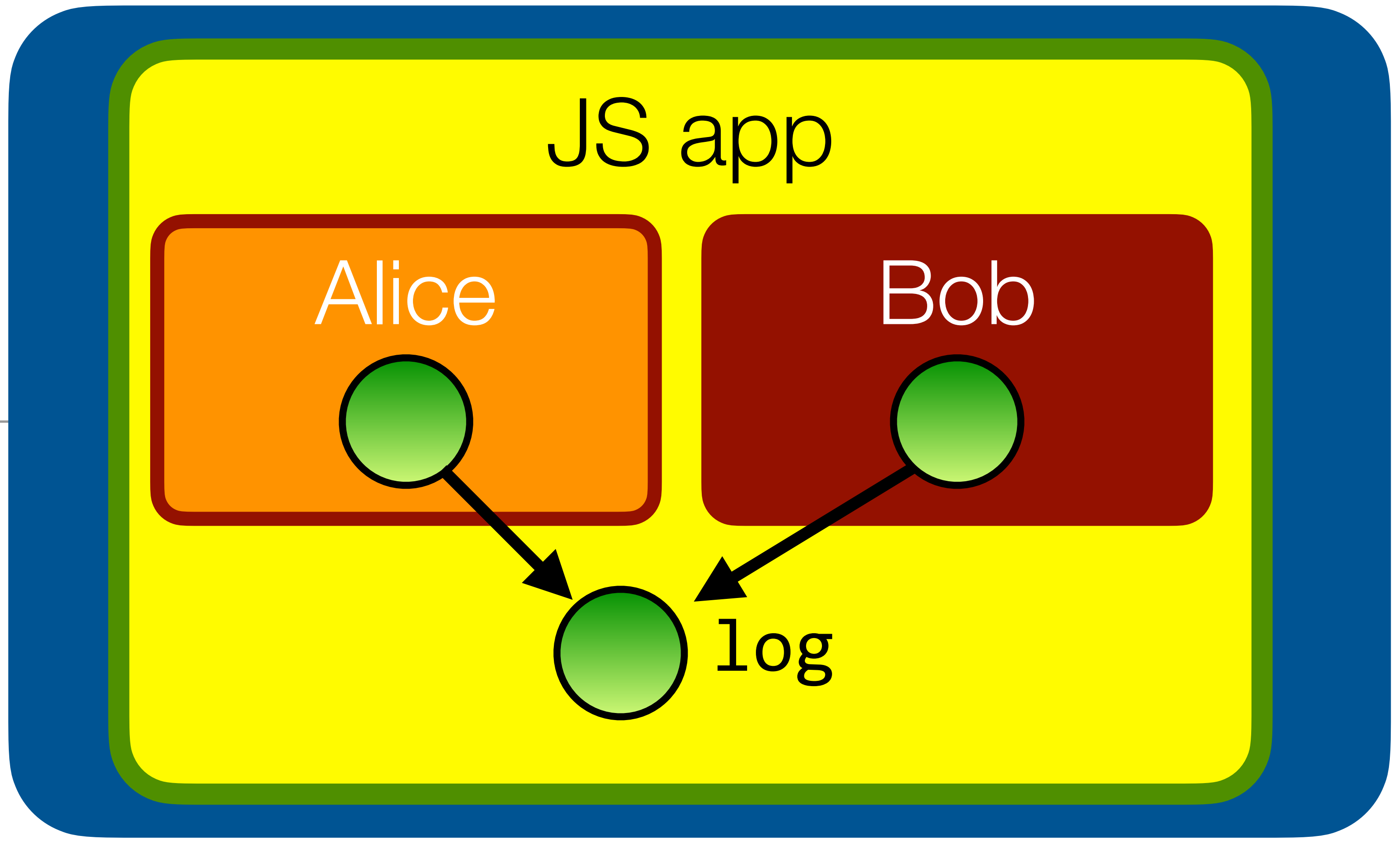
Three down, one to go



```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = harden(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)

~~// Bob can delete the entire log~~
~~log.read().length = 0~~

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~
```

Three down, one to go



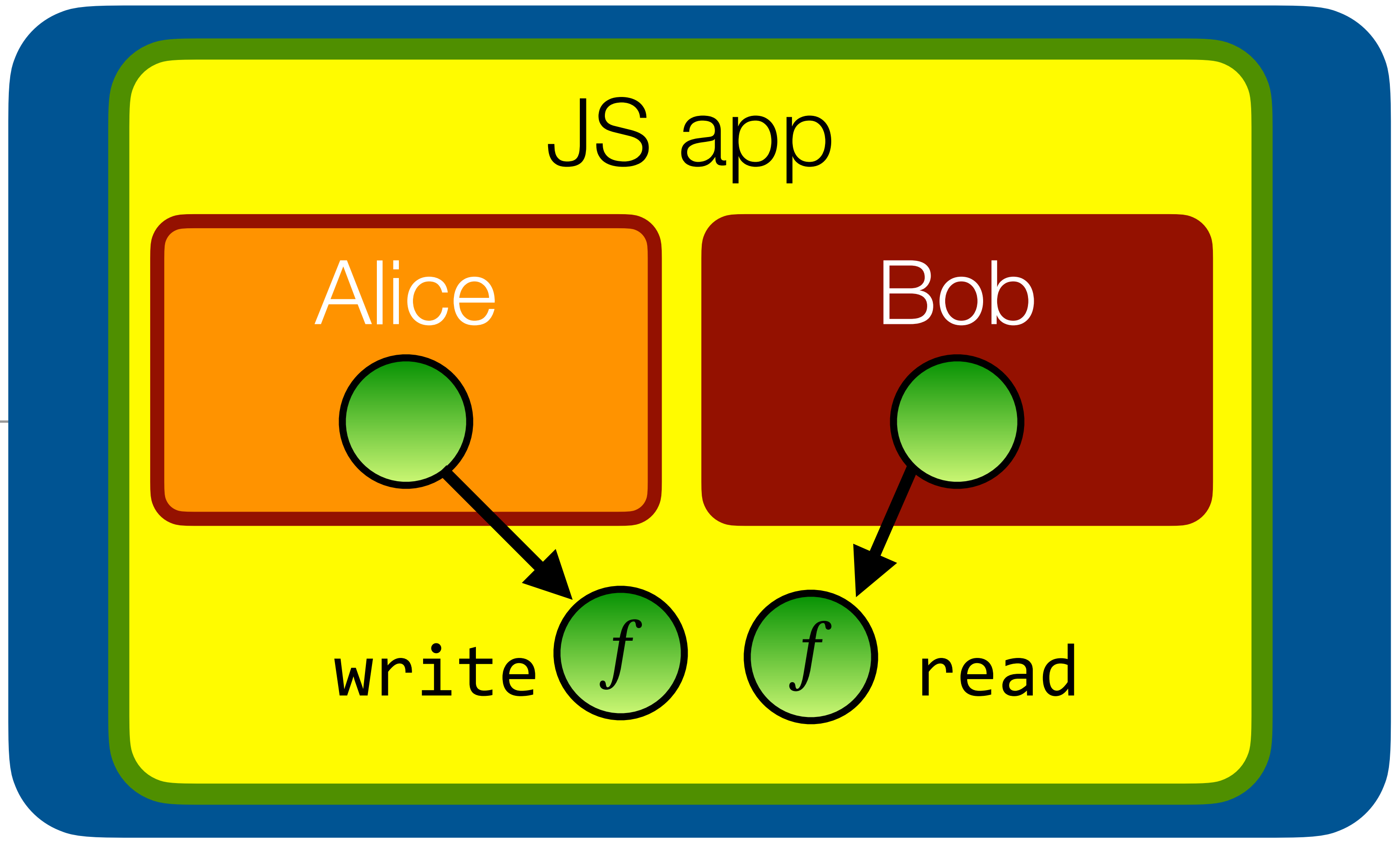
- Recall: we would like Alice to only write to the log, and Bob to only read from the log.
- Bob receives too much authority. How to limit?

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = harden(new Log());
alice(log);
bob(log);
```

```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)
```

```
~~// Bob can delete the entire log~~
~~log.read().length = 0~~

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~
```

Pass only the authority that Bob needs.



Just pass the write function to Alice and the read function to Bob.

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = new Log();
let read = harden(() => log.read());
let write = harden((msg) => log.write(msg));
alice(write);
bob(read);
```

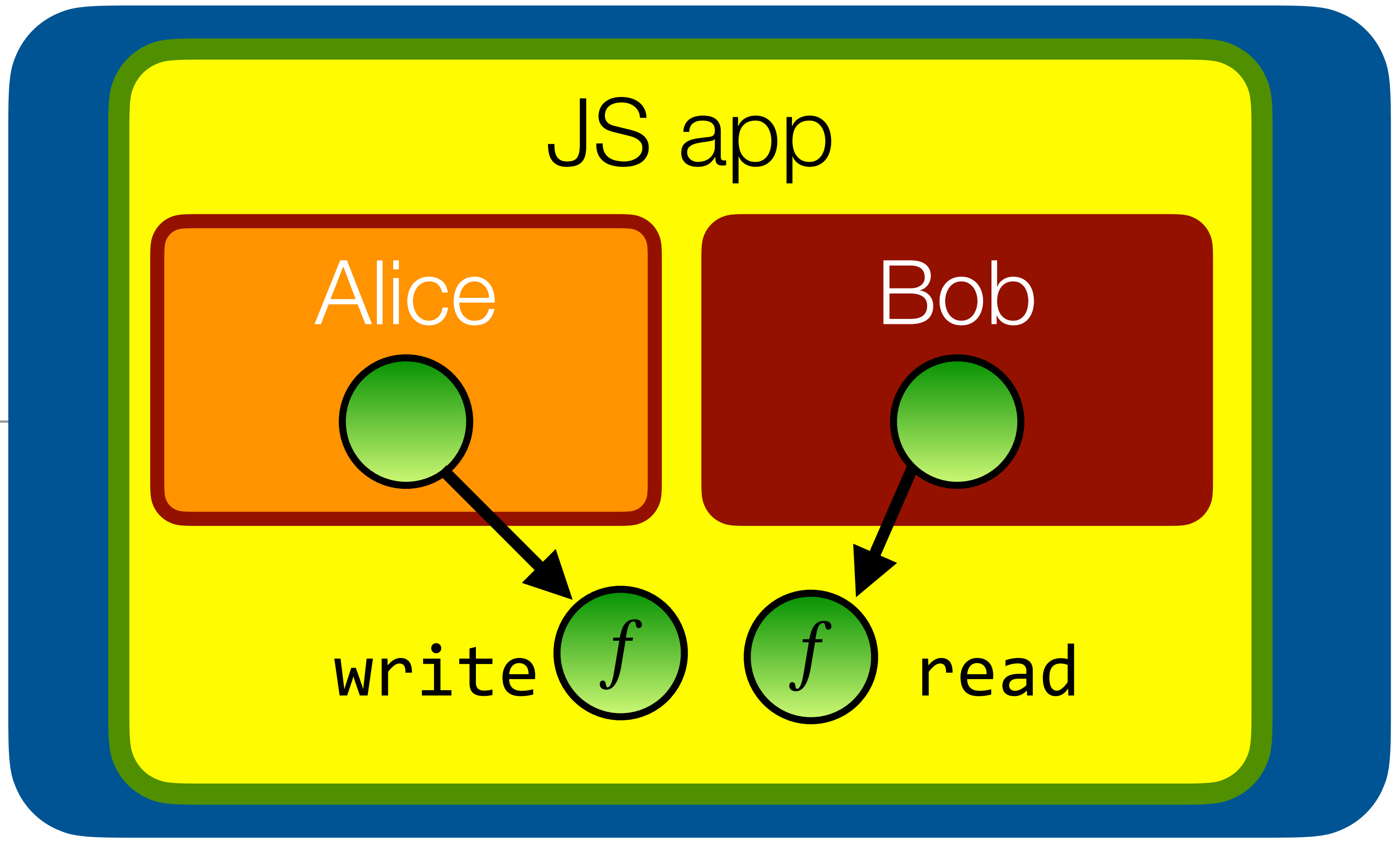
```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)
```

~~// Bob can delete the entire log~~
~~log.read().length = 0~~

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~

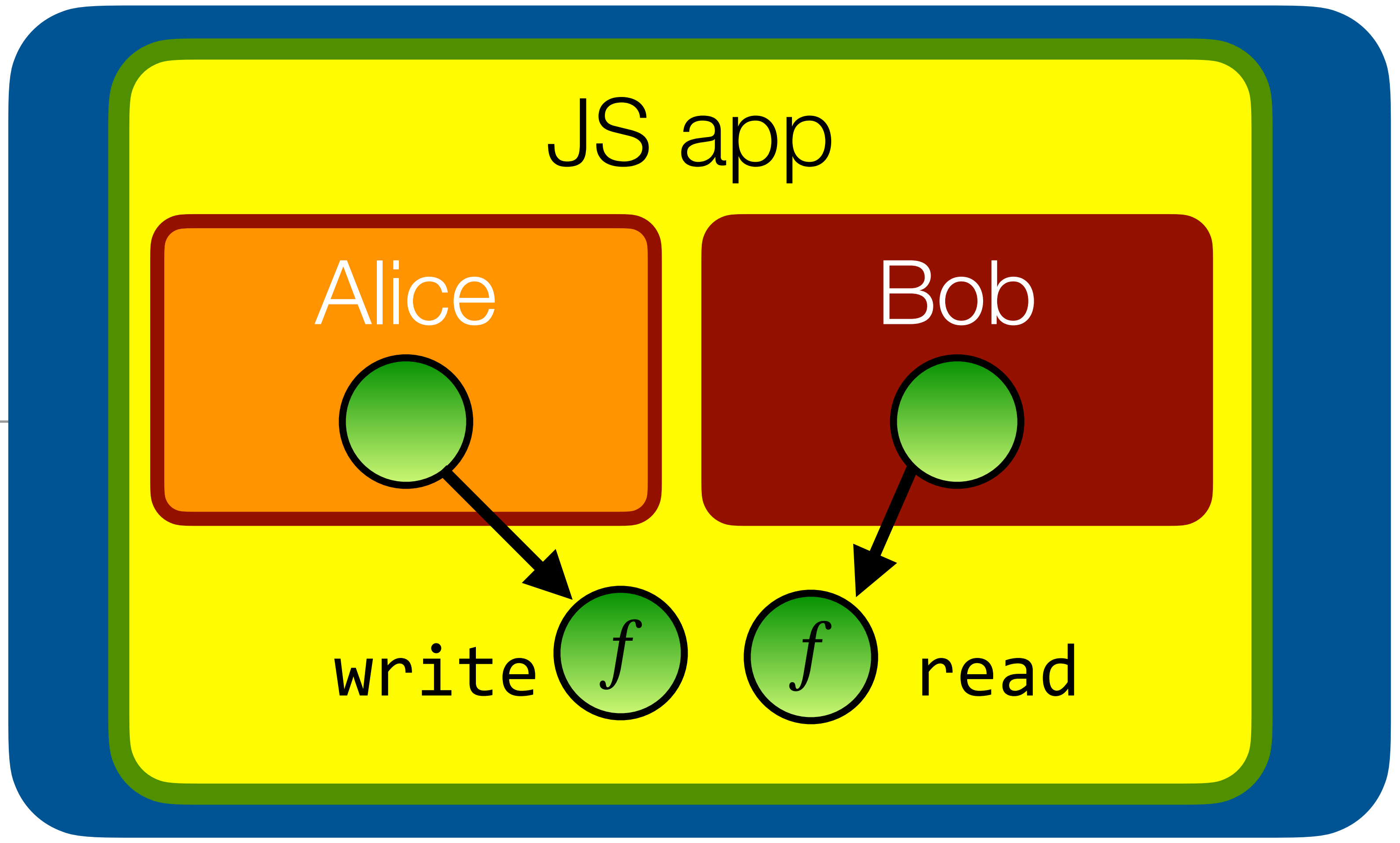
Success! We thwarted all of Evil Bob's attacks.



```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = new Log();
let read = harden(() => log.read());
let write = harden((msg) => log.write(msg));
alice(write);
bob(read);
```

```
// in bob.js
~~// Bob can just write to the log~~
~~log.write(“I’m polluting the log”)~~

~~// Bob can delete the entire log~~
~~log.read().length = 0~~

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~
```

Is there a better way to write this code?

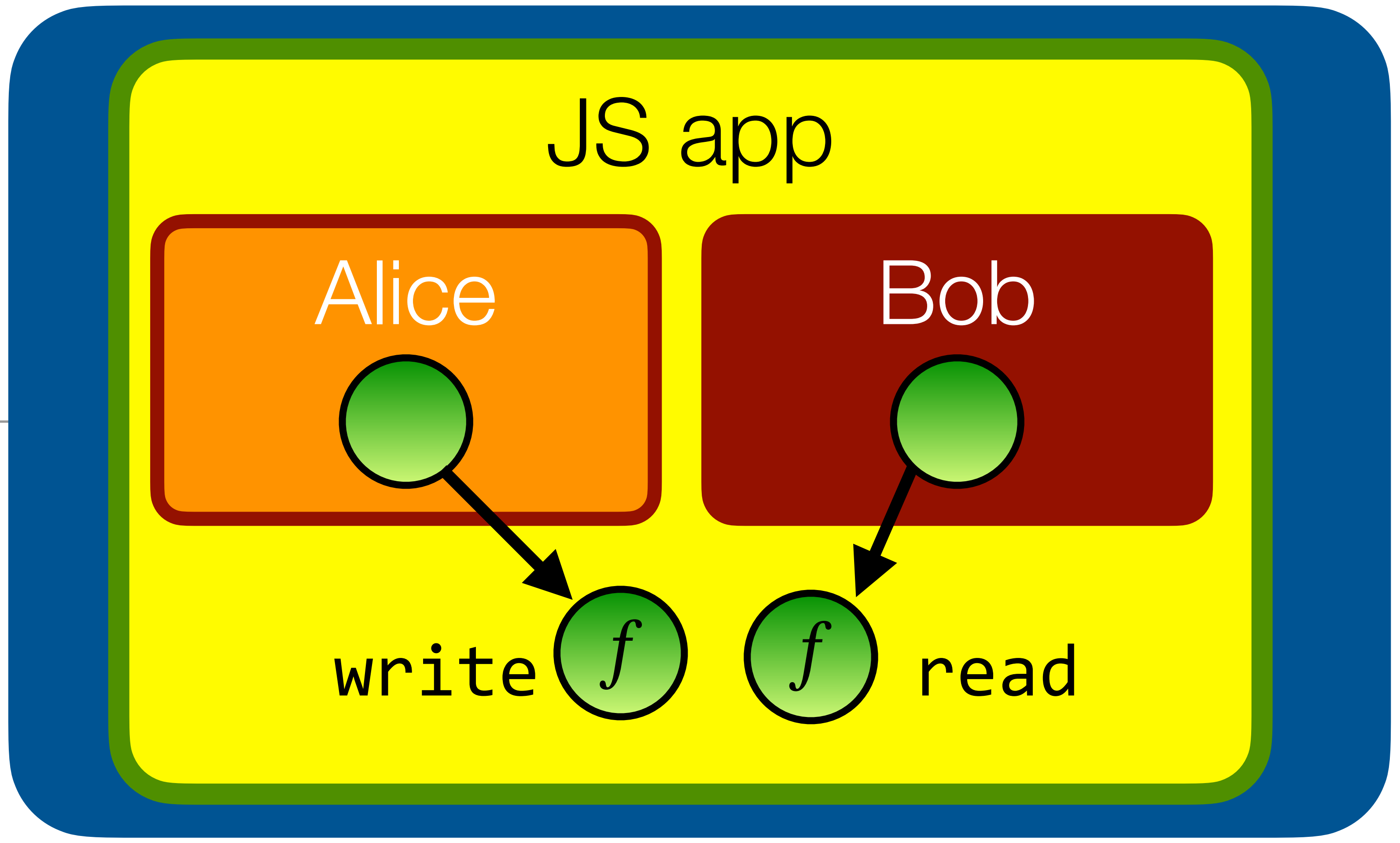


The burden of correct use is on the *client* of the class. Can we avoid this?

```
import * as alice from "alice.js";
import * as bob from “bob.js";

class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = new Log();
let read = harden(() => log.read());
let write = harden((msg) => log.write(msg));
alice(write);
bob(read);
```

```
// in bob.js
~~// Bob can just write to the log~~
~~log.write(“I’m polluting the log”)~~

~~// Bob can delete the entire log~~
~~log.read().length = 0~~

~~// Bob can replace the ‘write’ function~~
~~log.write = function(msg) {~~
~~  console.log(“I’m not logging anything”);~~
~~}~~

~~// Bob can still modify the write function~~
~~log.write.apply = function() { “gotcha” };~~
```

Use the **Function as Object** pattern



- A record of closures hiding state is a fine representation of an object of methods hiding instance vars
- Pattern long advocated by Doug Crockford instead of using classes or prototypes

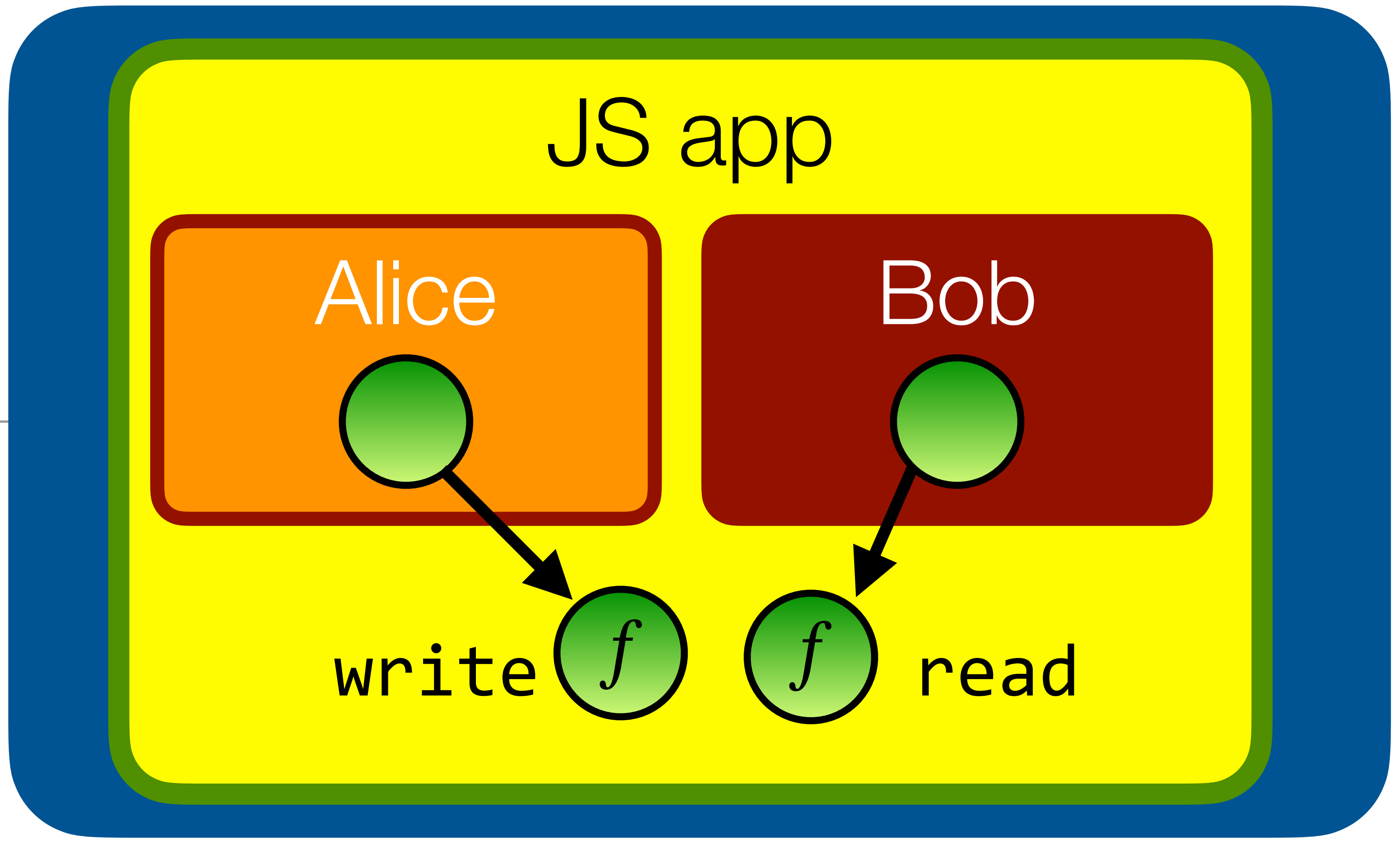
```
class Log {
  constructor() {
    this.messages_ = [];
  }
  write(msg) { this.messages_.push(msg); }
  read() { return [...this.messages_]; }
}

let log = new Log();
let read = harden(() => log.read());
let write = harden((msg) => log.write(msg));
alice(write);
bob(read);
```

⇨

```
function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```

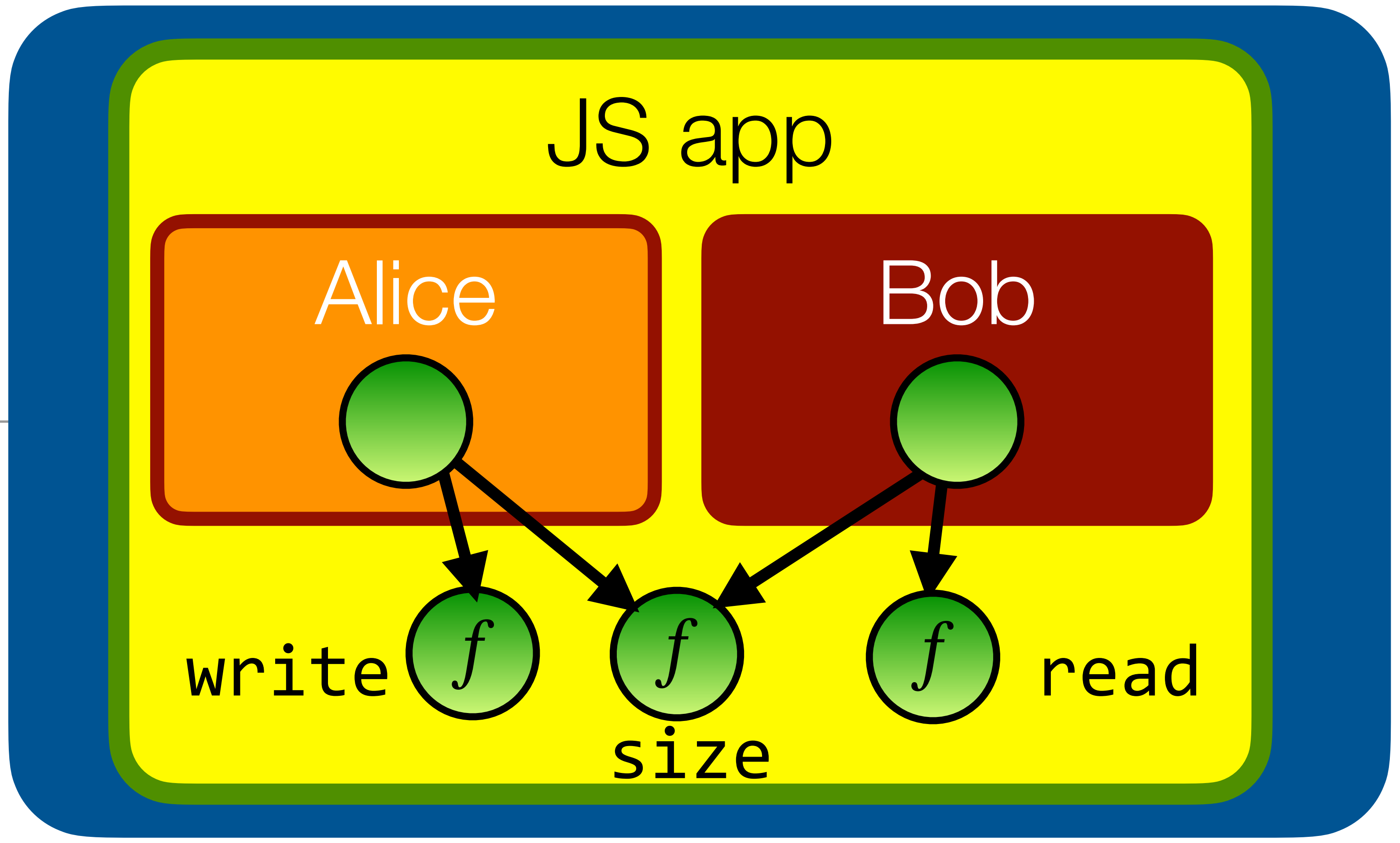
(See also https://martinfowler.com/bliki/FunctionAsObject.html)

Use the Function as Object pattern



```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```

What if Alice and Bob need more authority?



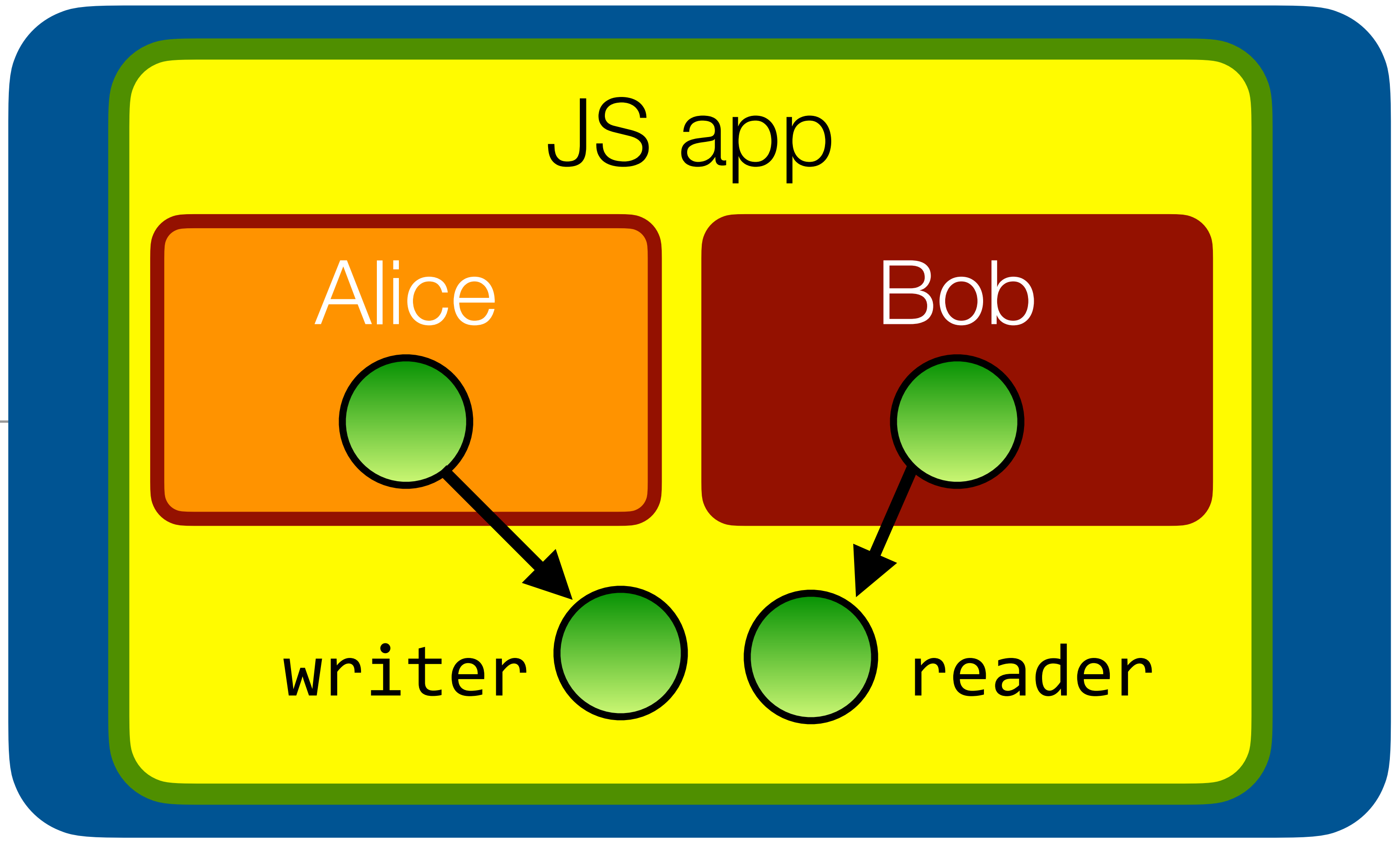
If over time we want to expose more functionality to Alice and Bob, we need to refactor all of our code.

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  function size() { return messages.length(); }
  return harden({read, write, size});
}

let log = makeLog();
alice(log.write, log.size);
bob(log.read, log.size);
```

Expose distinct authorities through **facets**

Easily deconstruct the API of a single powerful object into separate interfaces by nesting objects



```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  function size() { return messages.length(); }
  return harden({read, write, size});
}

let log = makeLog();
alice(log.write, log.size);
bob(log.read, log.size);
```

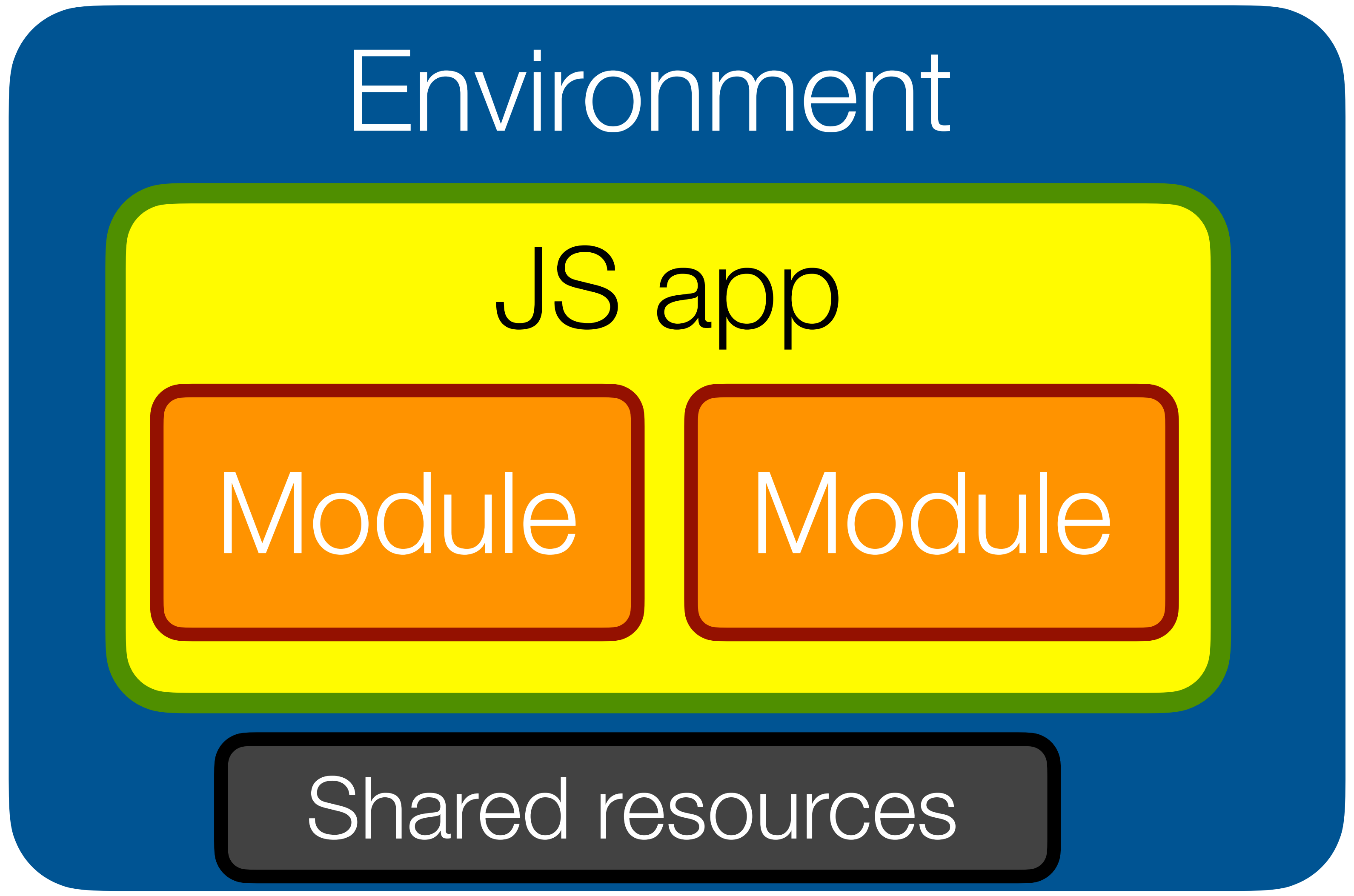
```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  function size() { return messages.length(); }
  return harden({
    reader: {read, size},
    writer: {write, size}
  });
}

let log = makeLog();
alice(log.writer);
bob(log.reader);
```

Demo

https://github.com/tvcutsem/lavamoat-demo

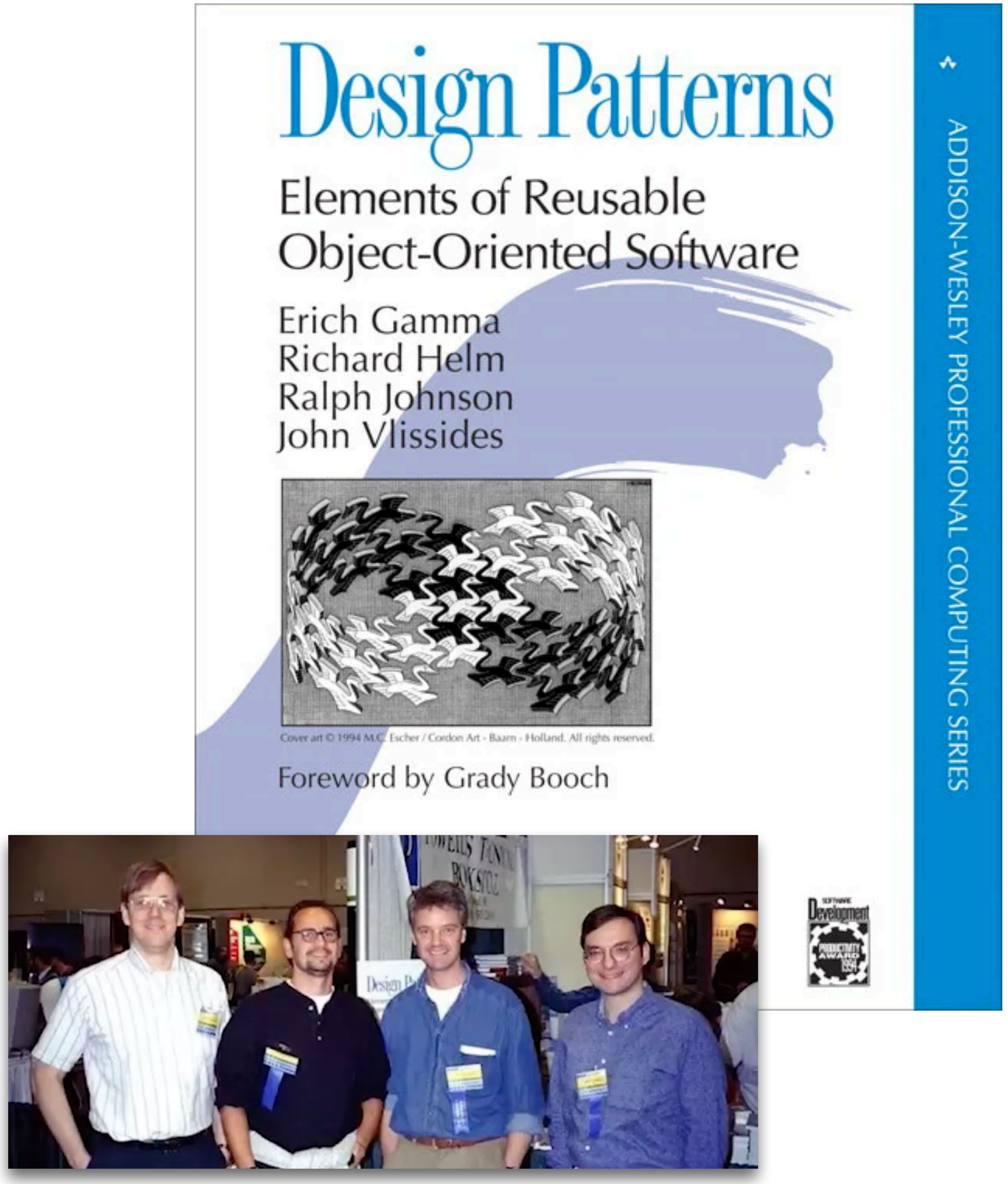
End of Part II: recap

- Modern JS apps are composed from many modules. You can’t trust them all.
- Traditional security boundaries don’t exist between modules. Compartments add basic isolation.
- **Isolated modules must still interact!**
- **Compose** functionality from untrusted modules in a **least-authority** manner
- This can be done via **reusable programming patterns** that rely on object-capability security



Part III
Safely composing modules using least-authority patterns

Design Patterns ("Gang of Four", 1994)



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

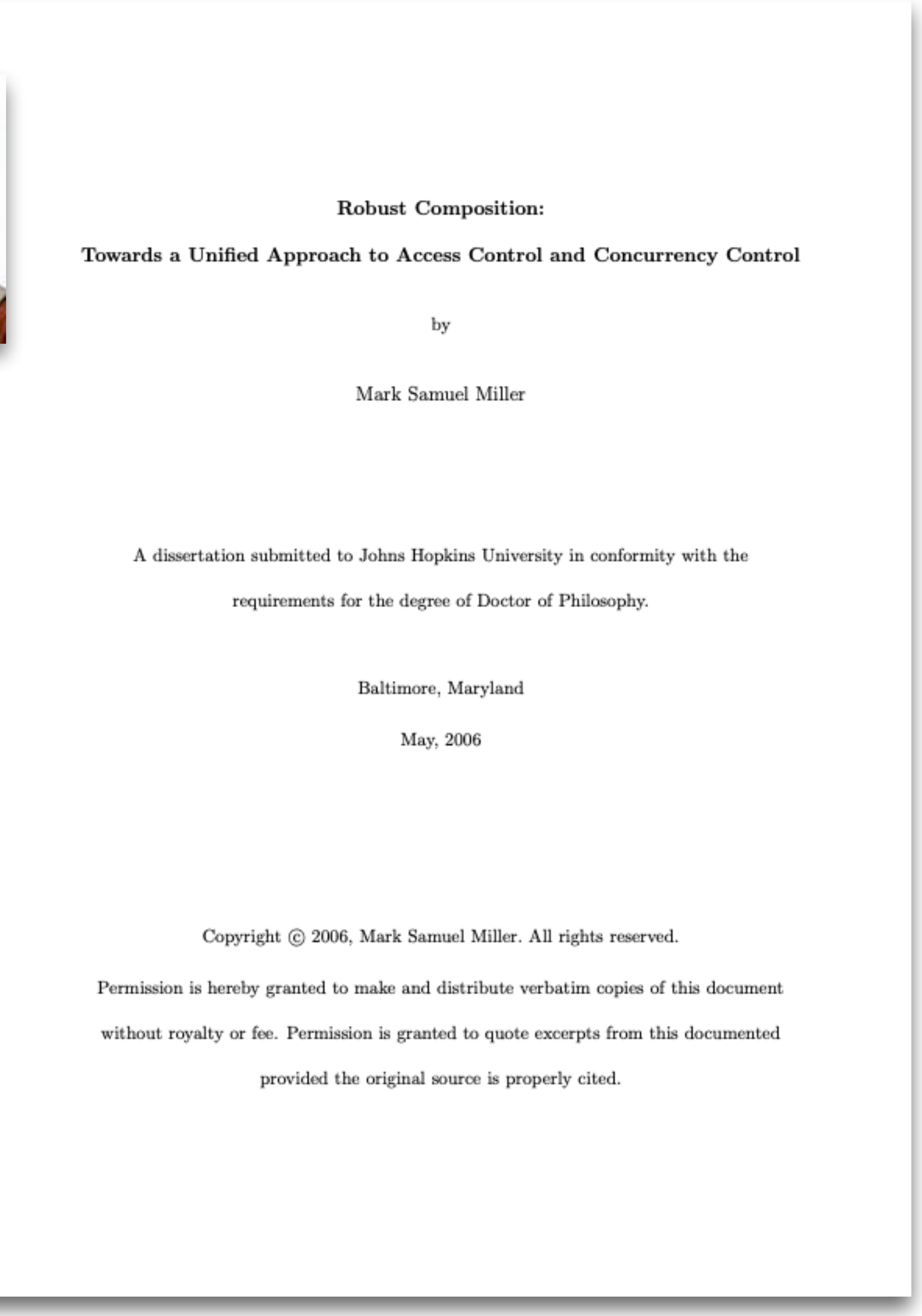
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Visitor
- Factory
- Observer
- Singleton
- State
- …

Design Patterns for **robust composition** (Mark S. Miller, 2006)



Robust Composition:

Towards a Unified Approach to Access Control and Concurrency Control

by

Mark Samuel Miller

A dissertation submitted to Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

May, 2006

Copyright © 2006, Mark Samuel Miller. All rights reserved.

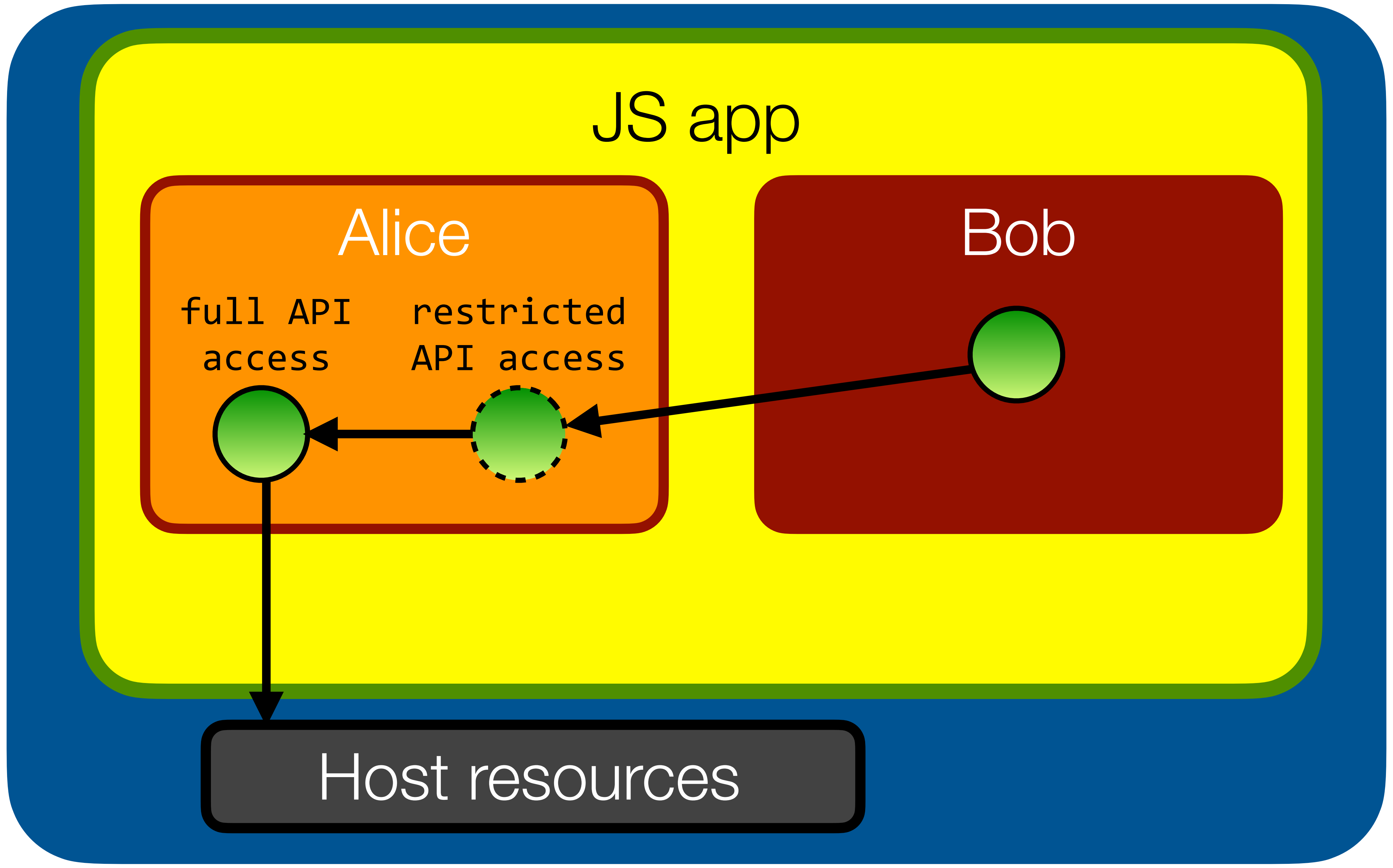
Permission is hereby granted to make and distribute verbatim copies of this document without royalty or fee. Permission is granted to quote excerpts from this documented provided the original source is properly cited.

http://www.erights.org/talks/thesis/markm-thesis.pdf

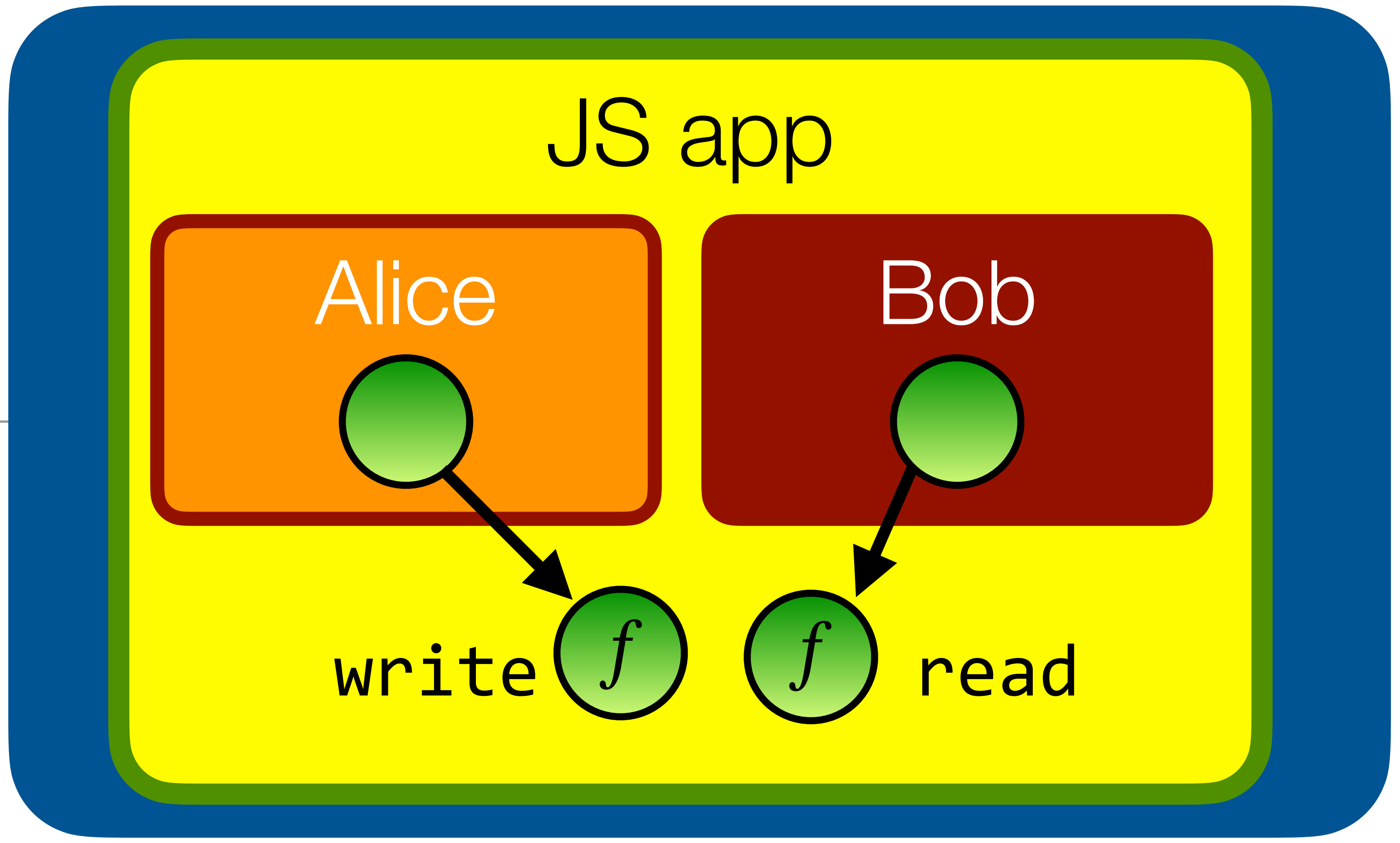
- Facets
- Taming
- Caretaker
- Membrane
- Sealer/unsealer pair
- …

Recall: the Principle of Least Authority (POLA)

- A module should only be given the authority it needs to do its job, and nothing more



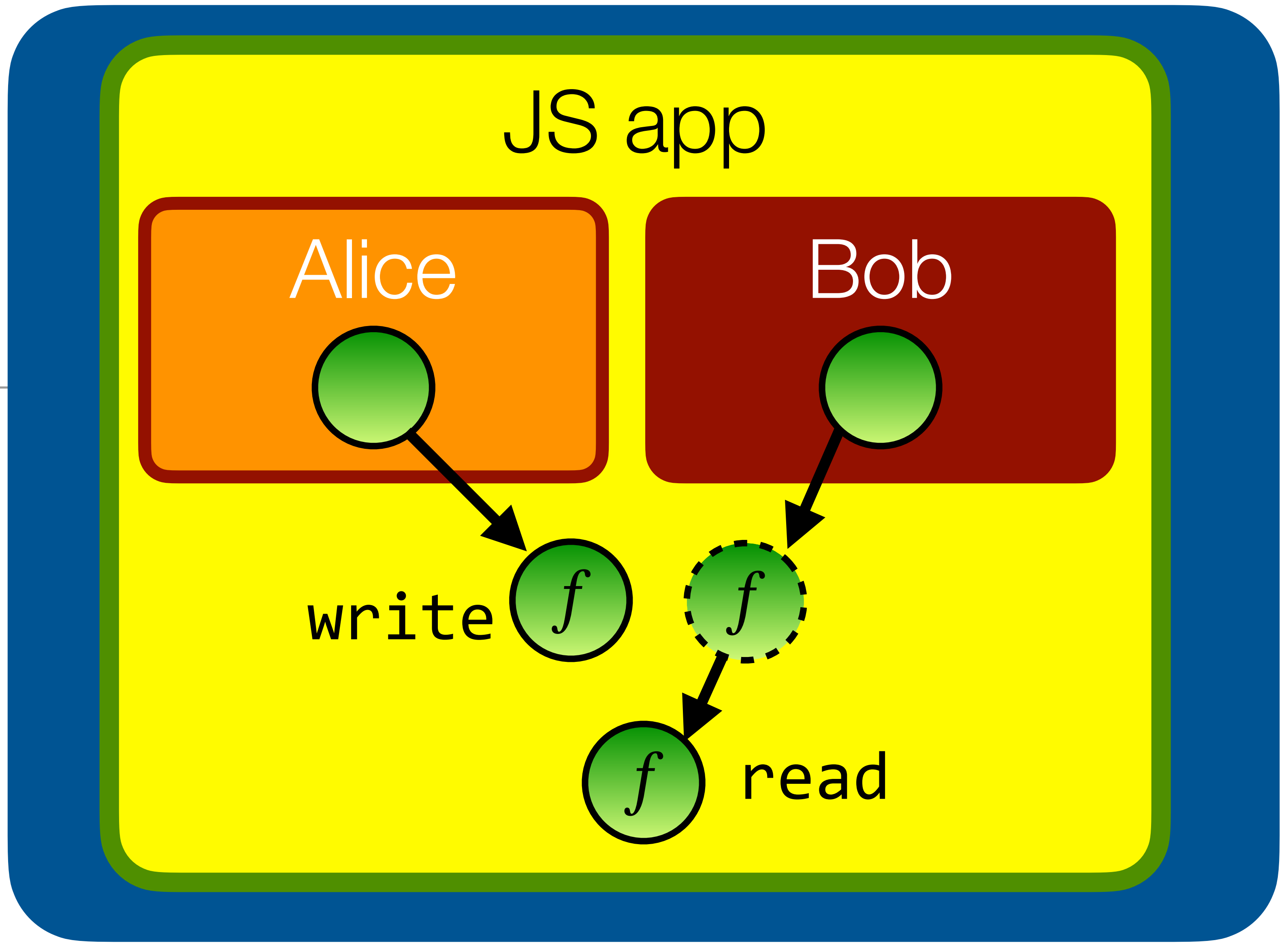
Further limiting Bob’s authority



We would like to give Bob only **temporary** read access to the log.

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();

alice(log.write);
bob(log.read);
```

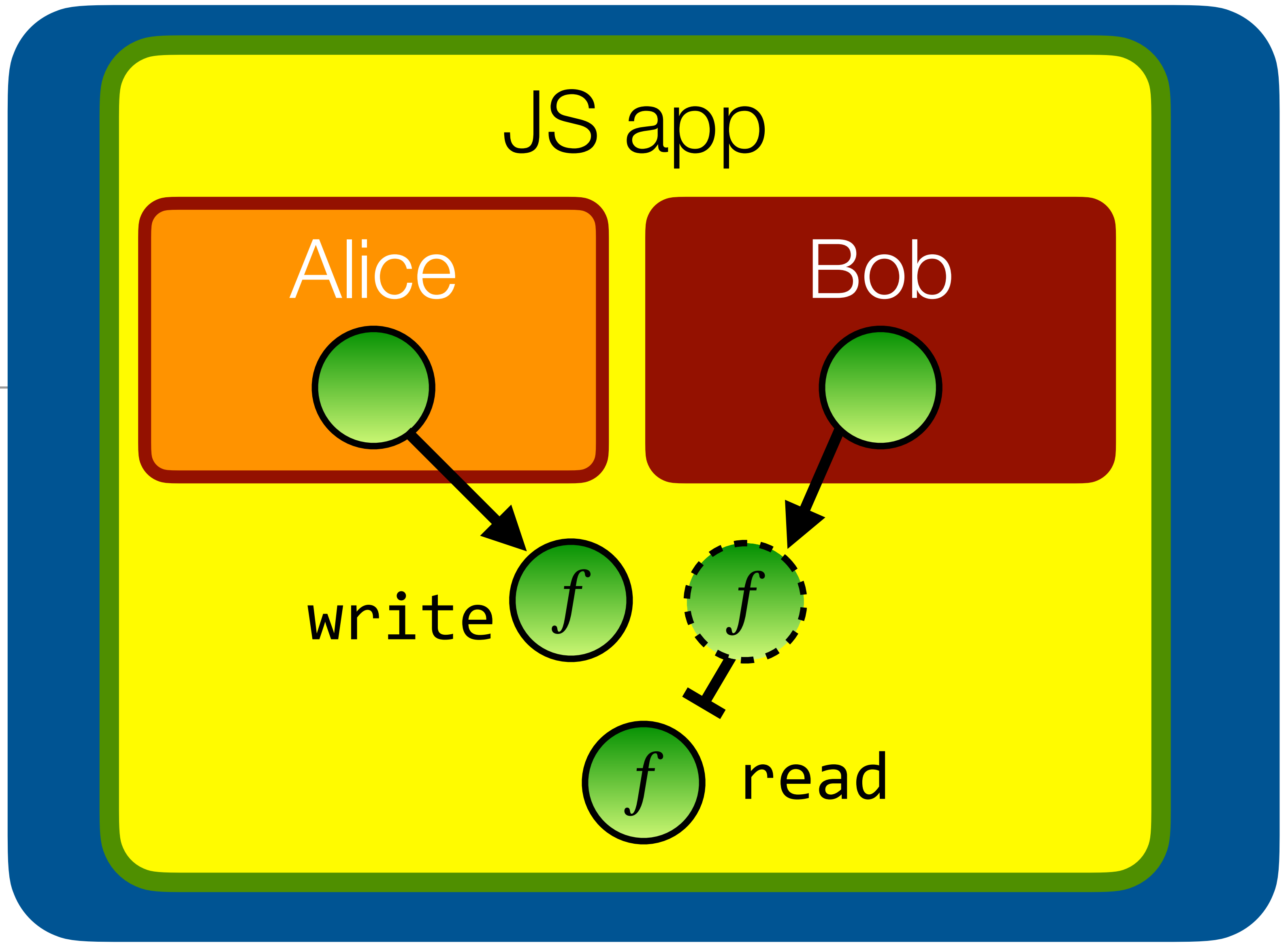
Use **caretaker** to insert access control logic



We would like to give Bob only **temporary** read access to the log.

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();
let [rlog, revoke] = makeRevokableLog(log);
alice(log.write);
bob(rlog.read);
```

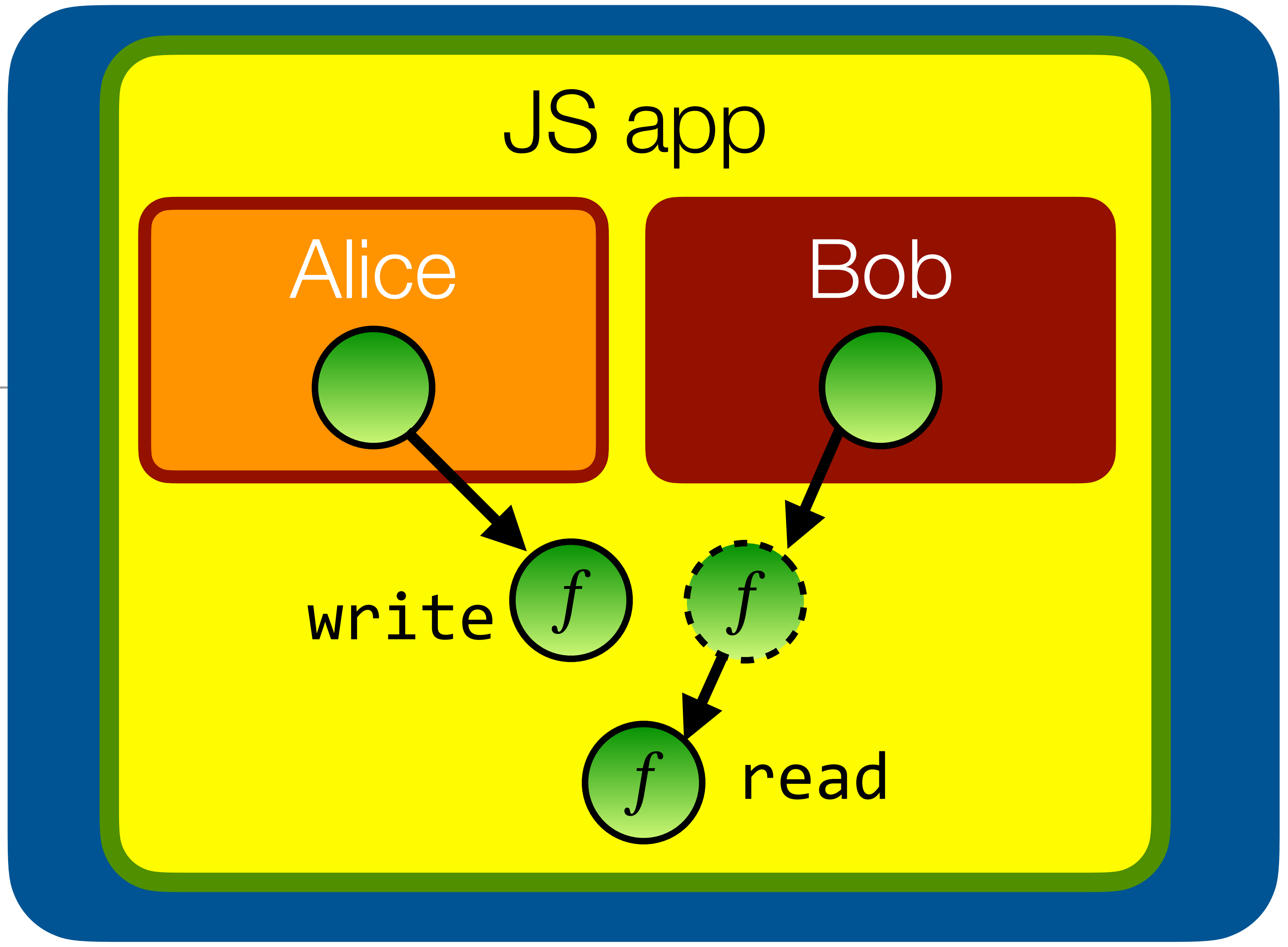
Use **caretaker** to insert access control logic



We would like to give Bob only **temporary** read access to the log.

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();
let [rlog, revoke] = makeRevokableLog(log);
alice(log.write);
bob(rlog.read);

// to revoke Bob’s access:
revoke();
```

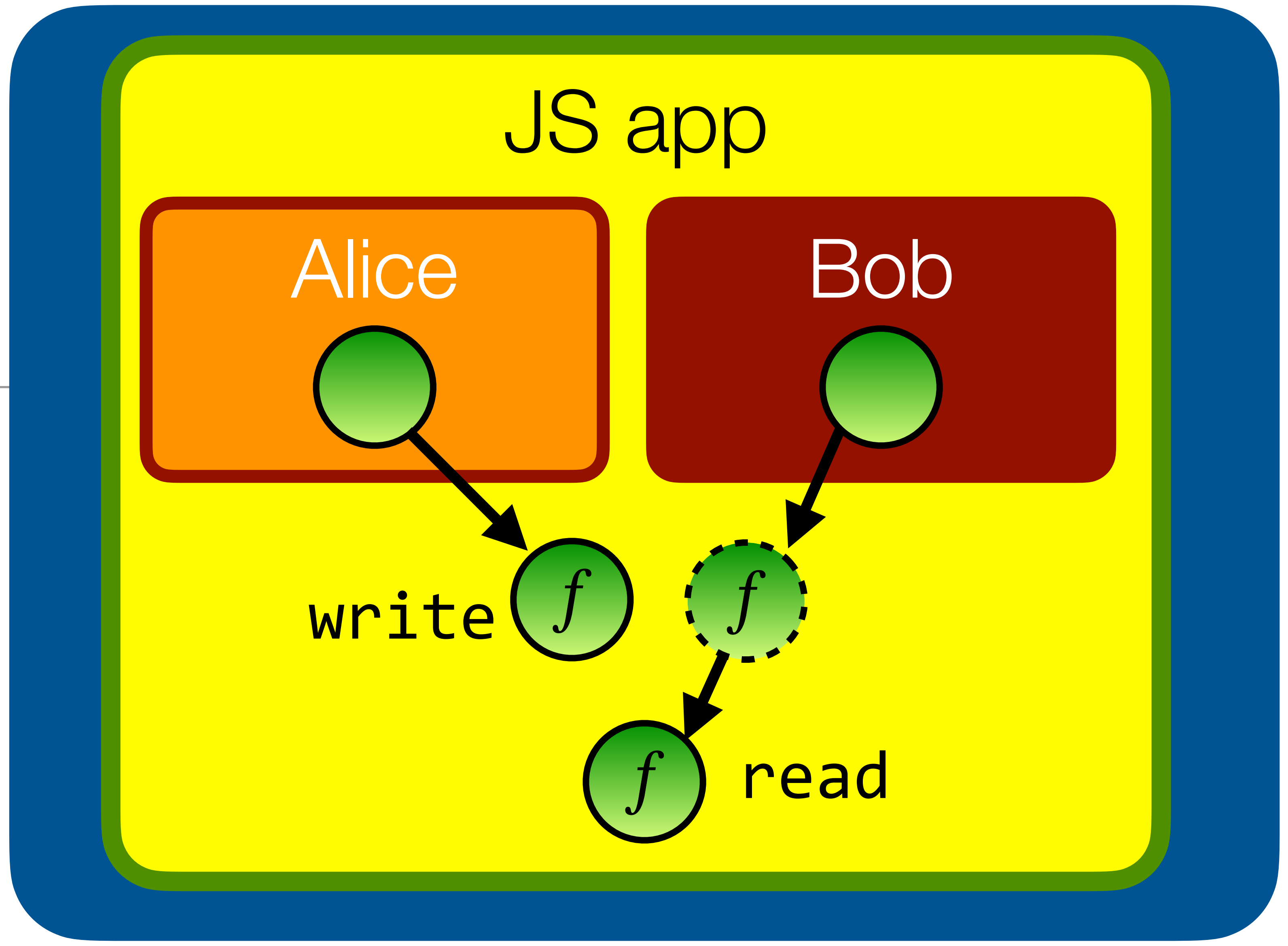
Use **caretaker** to insert access control logic



```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();
let [rlog, revoke] = makeRevokableLog(log);
alice(log.write);
bob(rlog.read);

// to revoke Bob’s access:
revoke();
```

```
function makeRevokableLog(log) {
  function revoke() { log = null; };
  let proxy = {
    write(msg) { log.write(msg); }
    read() { return log.read(); }
  };
  return harden([proxy, revoke]);
}
```

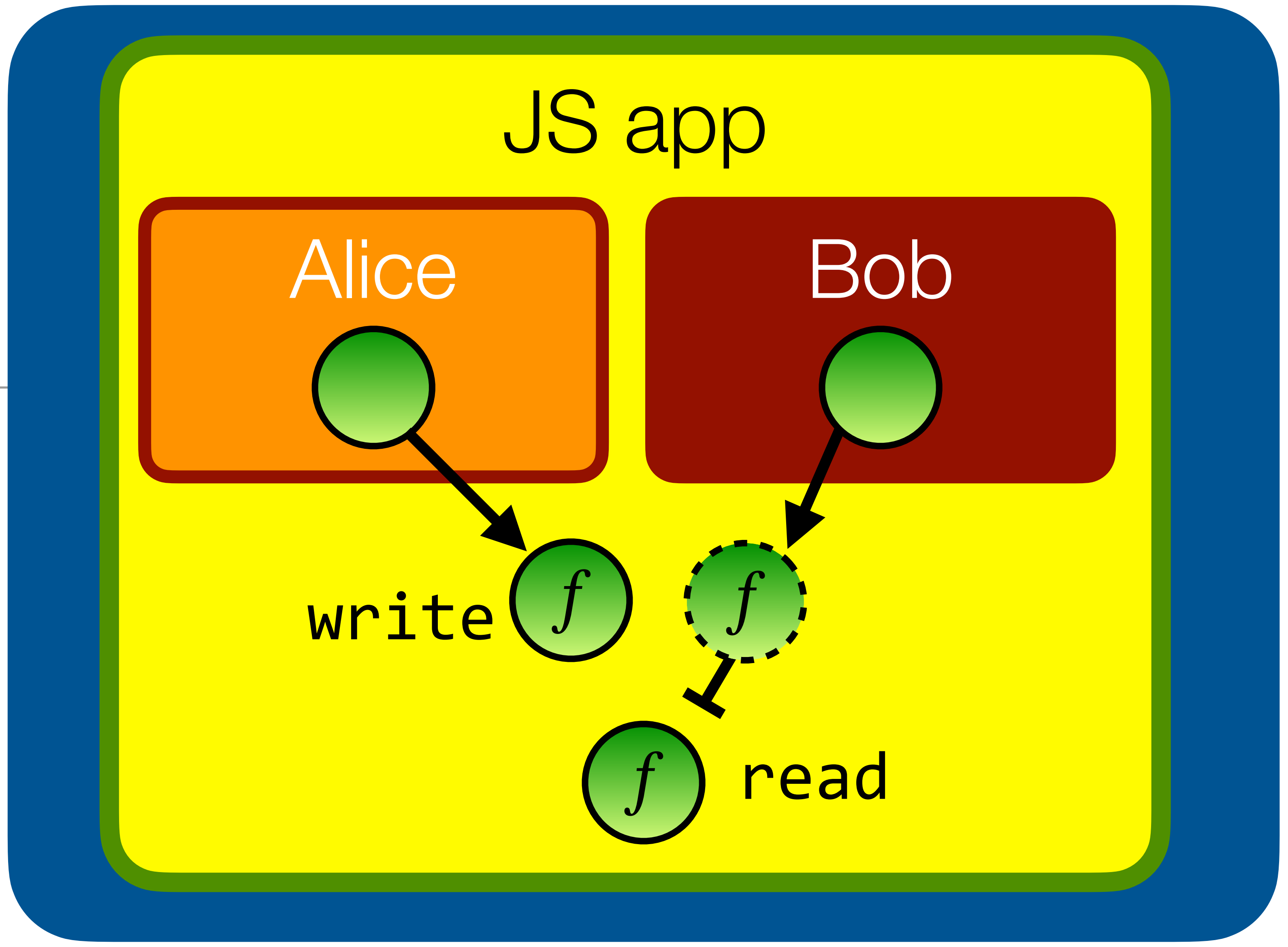
A caretaker is just a proxy object



```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();
let [rlog, revoke] = makeRevokableLog(log);
alice(log.write);
bob(rlog.read);

// to revoke Bob’s access:
revoke();
```

```
function makeRevokableLog(log) {
  function revoke() { log = null; };
  let proxy = {
    write(msg) { log.write(msg); }
    read() { return log.read(); }
  };
  return harden([proxy, revoke]);
}
```

A caretaker is just a proxy object

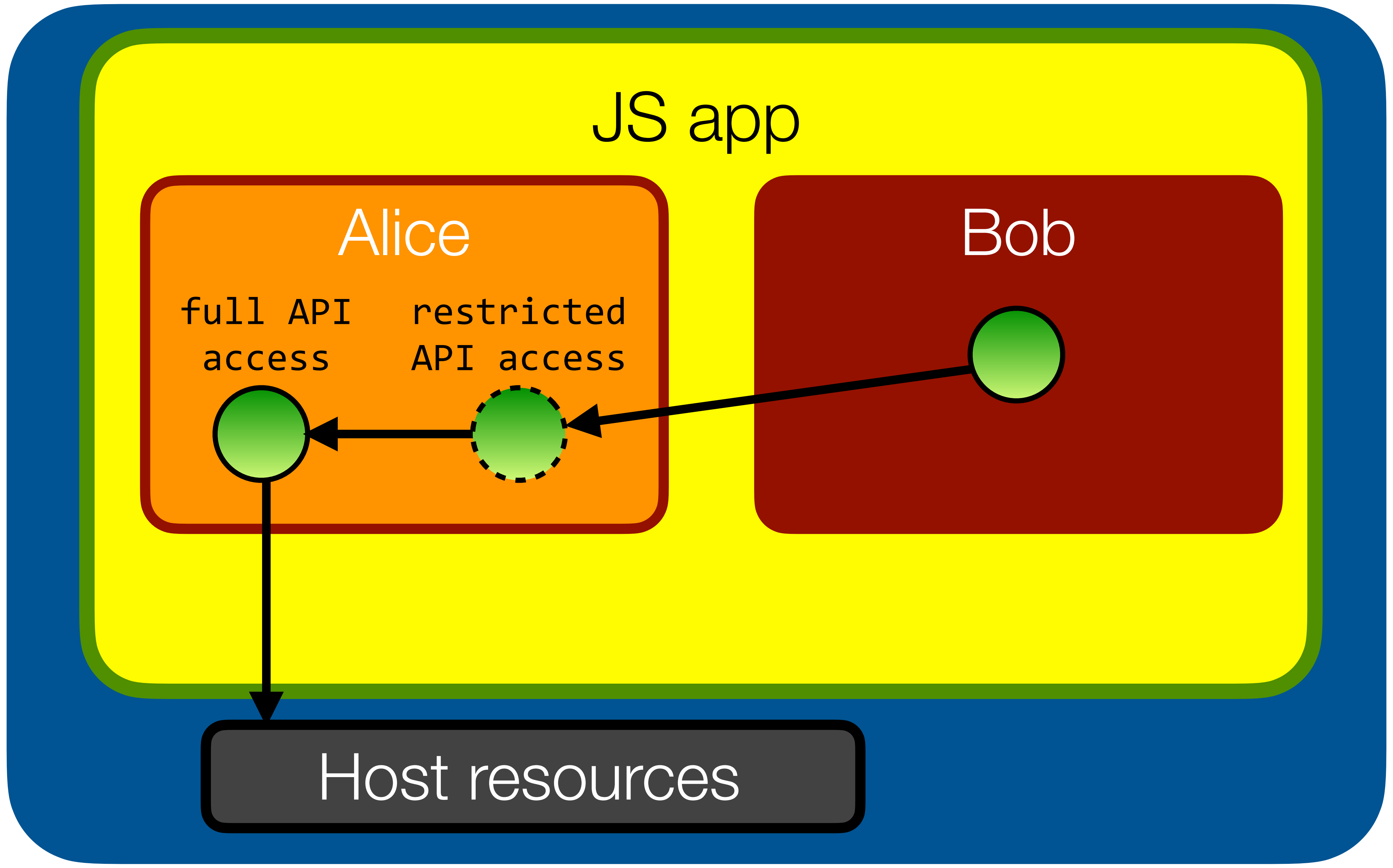


```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();
let [rlog, revoke] = makeRevokableLog(log);
alice(log.write);
bob(rlog.read);

// to revoke Bob’s access:
revoke();
```

```
function makeRevokableLog(log) {
  function revoke() { log = null; };
  let proxy = {
    write(msg) { log.write(msg); }
    read() { return log.read(); }
  };
  return harden([proxy, revoke]);
}
```

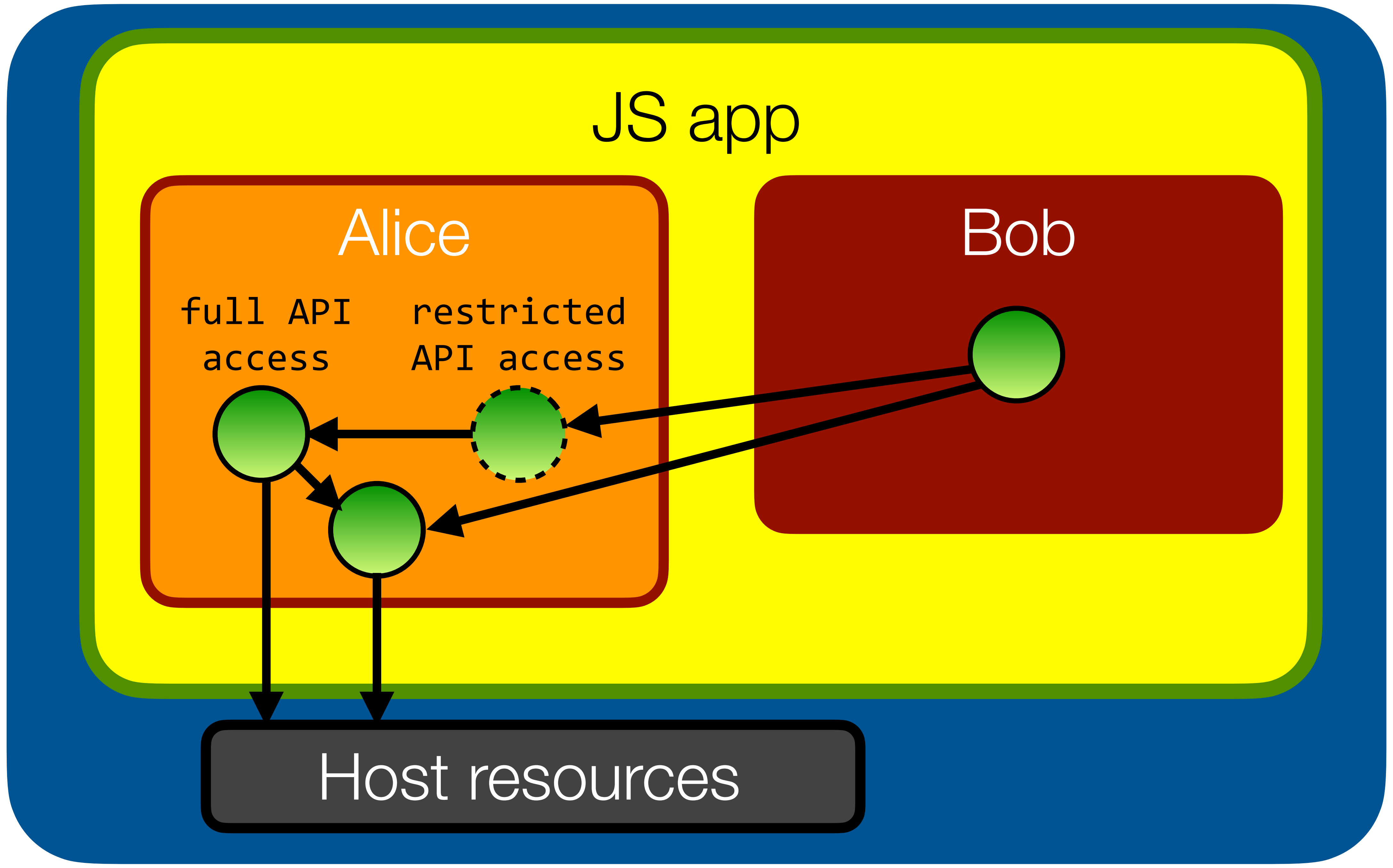
Taming is the process of restricting access to powerful APIs

- Expose powerful objects through restrictive proxies to third-party code
- E.g. Alice might give Bob read-only access to a specific subdirectory of her file system



Taming is the process of restricting access to powerful APIs

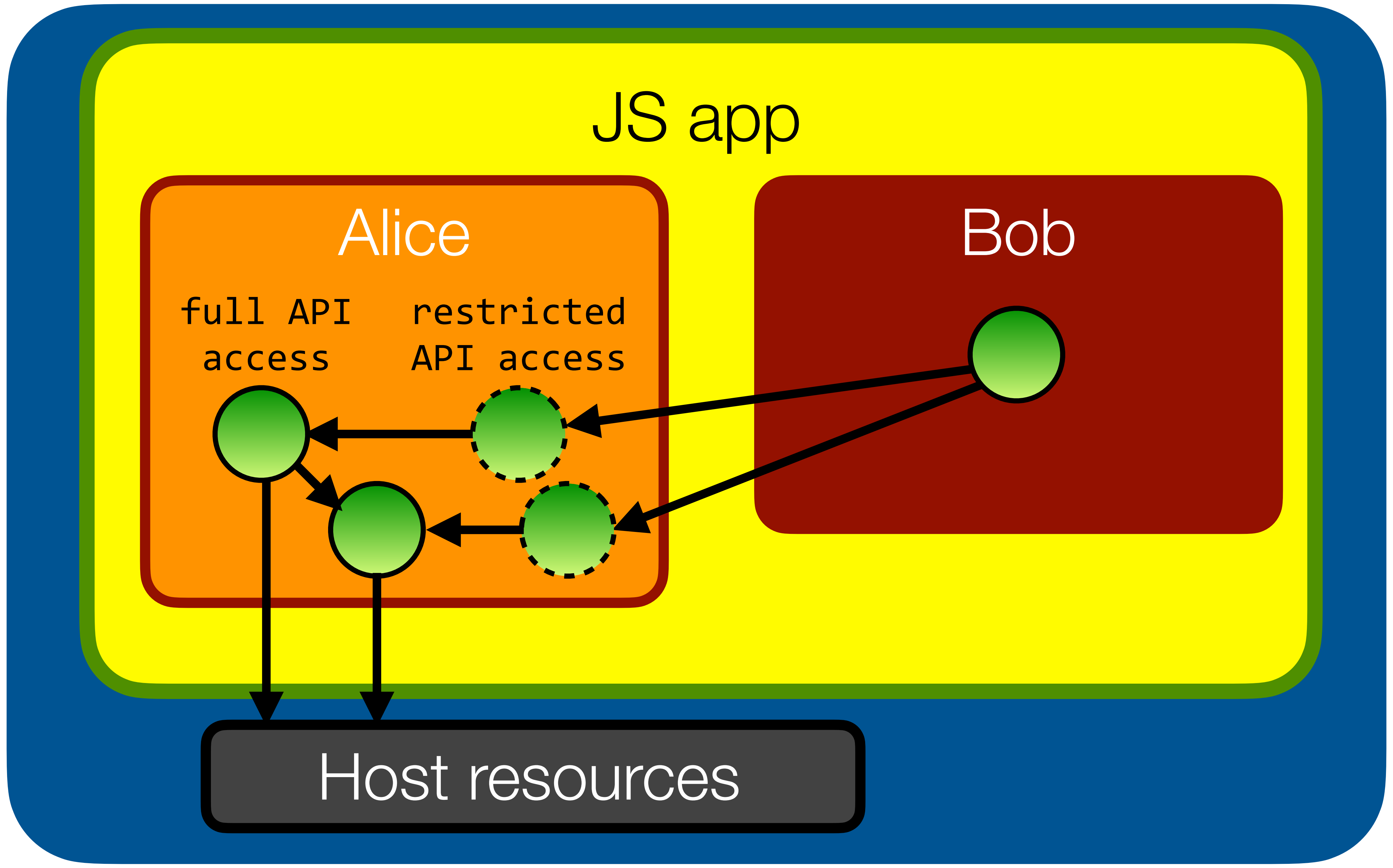
Potential **hazard:** the taming proxy must ensure it does not “leak” privileged access to host resources through the tamed API (e.g. through return values)



Taming is the process of restricting access to powerful APIs

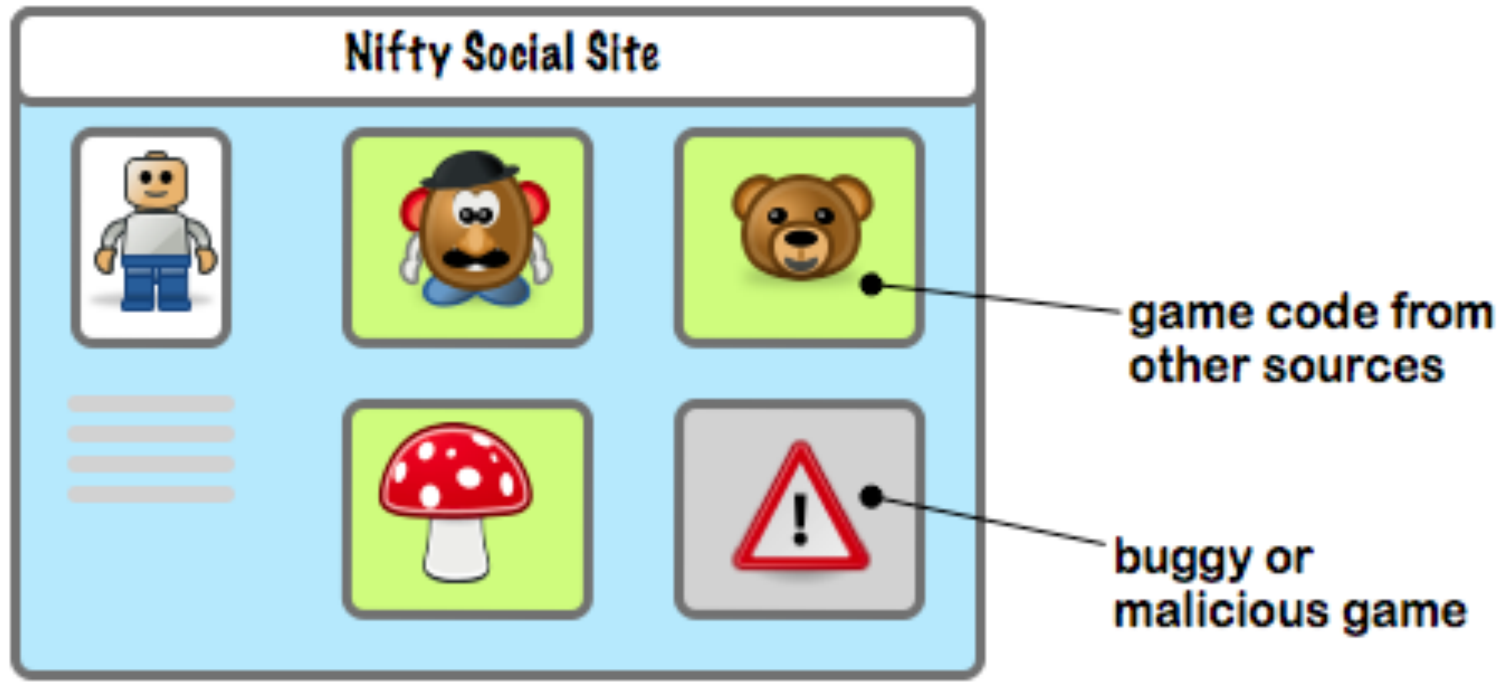
The **solution** is to transitively apply the proxy pattern to return values as well. This pattern is called a “**membrane**”

Deep dive blog post at tvcutsem.github.io/membranes

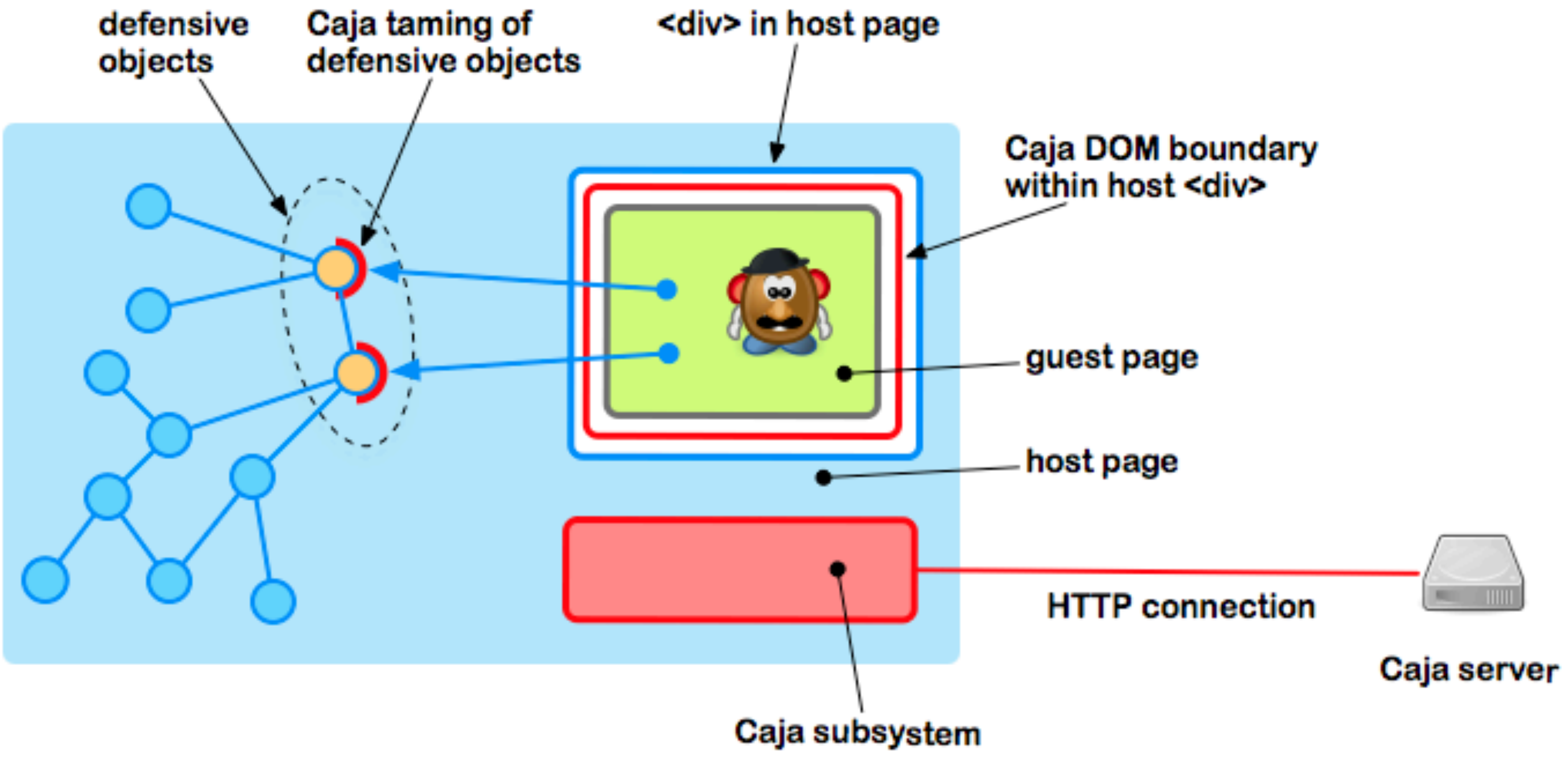


Least-authority patterns are used in industry

Example: how Google Caja uses **taming** to restrict access to the browser DOM



Google Caja



(source: Google Caja documentation: https://developers.google.com/caja/docs/about)

Least-authority patterns are used in industry

moddable

Moddable XS

Uses **Compartments** for safe end-user scripting of IoT products

METAMASK

MetaMask Snaps

Uses **LavaMoat** to sandbox plugins in their crypto web wallet

AGORIC

Agoric Zoe

Uses **Hardened JS** for writing smart contracts and Dapps

Google Caja

Uses **taming** for safe html embedding of third-party content

Mozilla Firefox

Uses **membranes** to isolate site origins from privileged JS code

salesforce

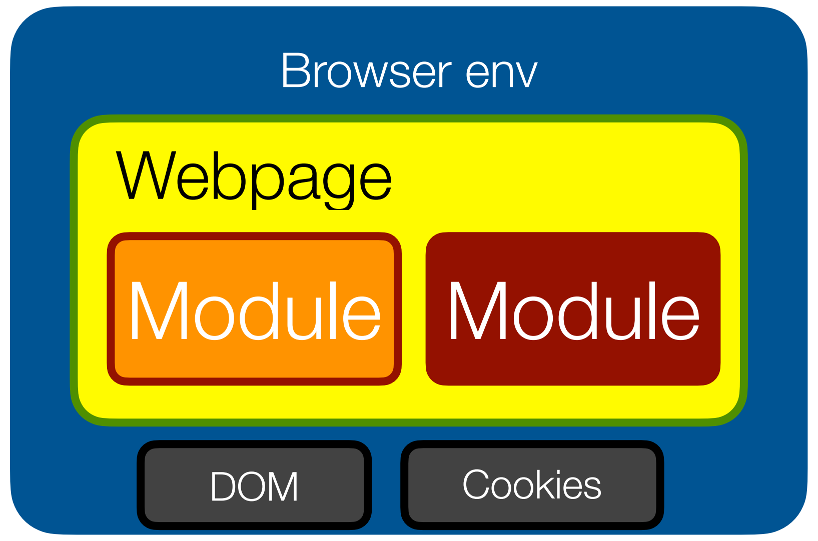
Salesforce Lightning

Uses **realms** and **membranes** to isolate & observe UI components

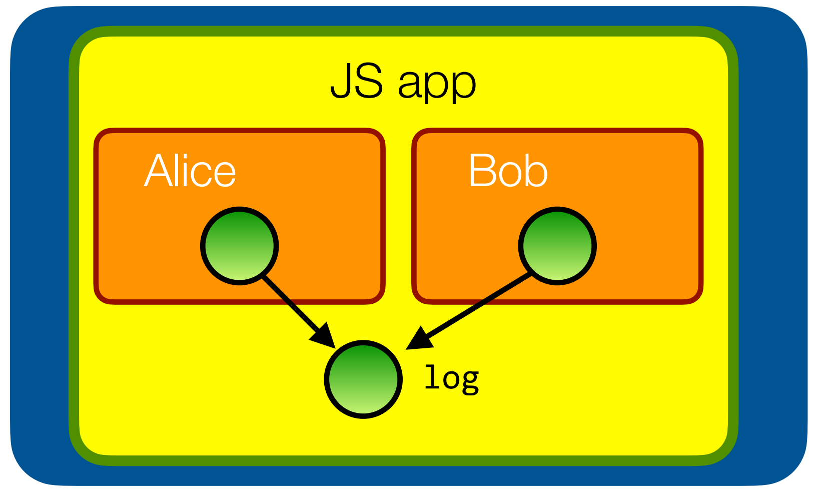
Summary

This Lecture: Recap

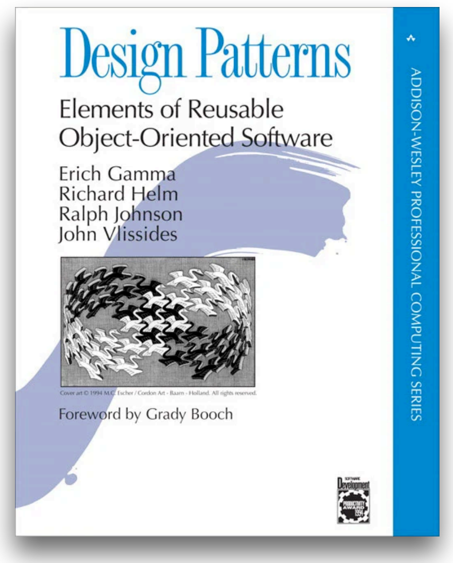
- Part I: **why module isolation** is critical to modern JavaScript applications

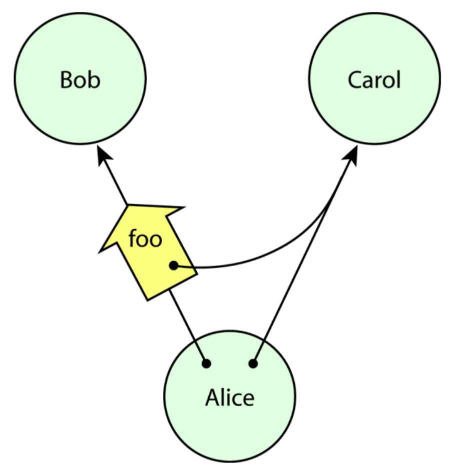


- Part II: the **Principle of Least Authority**, by example



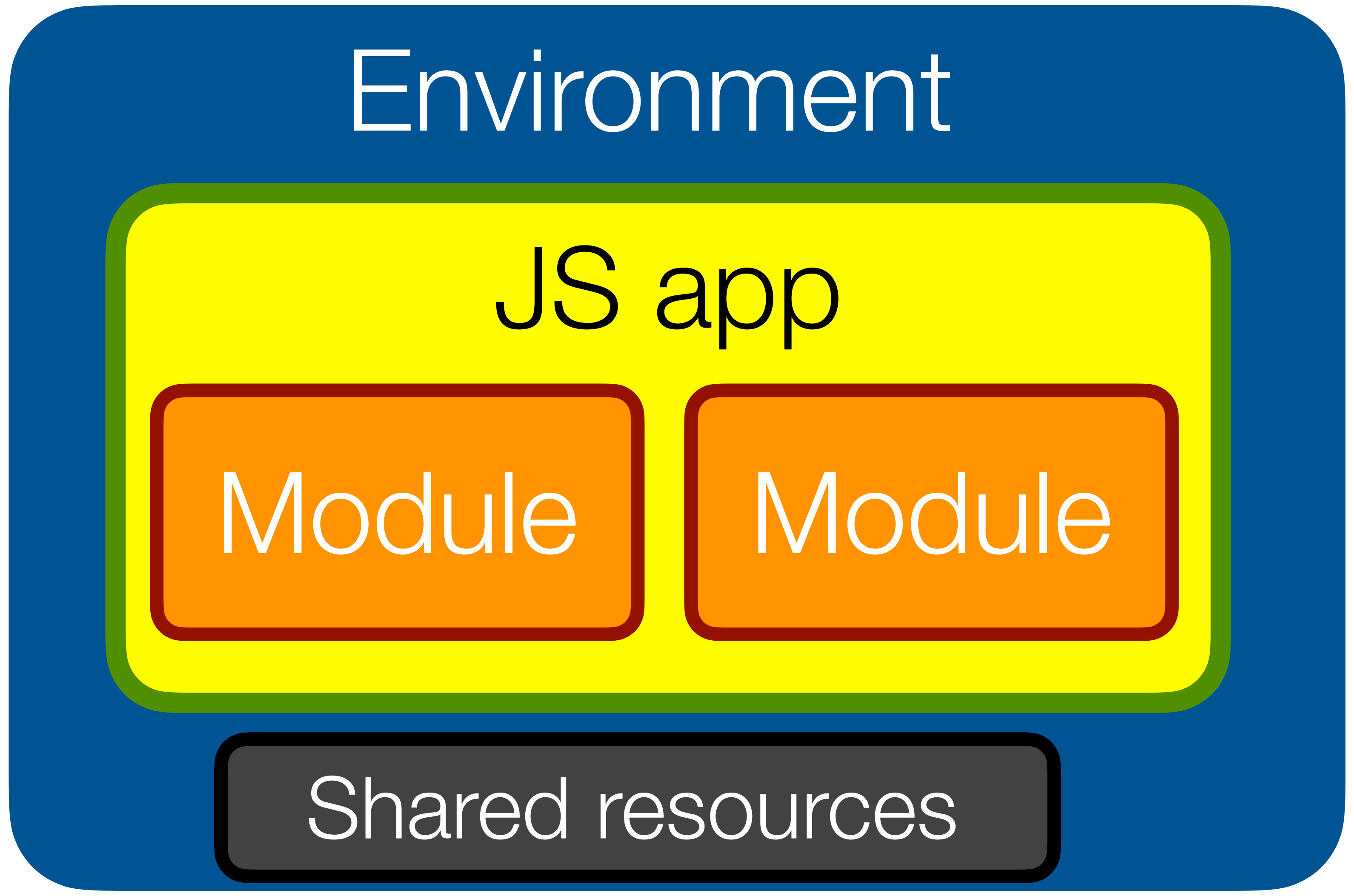
- Part III: safely composing modules using **least-authority patterns**


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



The take-away messages

- Modern applications are **composed from many modules**.
- You can't trust them all (**software supply chain attacks**)
- Apply the "principle of least authority" to **limit trust**.
  - Step 1: **Isolate modules** (Hardened JS & Lavamoat)
  - Step 2: Let modules **interact with "least authority"** (using reusable programming patterns)
- Understanding these patterns is **important in a world of > 2,000,000 NPM modules.**
- Even more critical **in the emerging "Web3"** where code can access valuable digital assets (think: tokens, NFTs, …)



KU LEUVEN

DistriNet

Designing “least-authority” JavaScript apps

Tom Van Cutsem
DistriNet KU Leuven

Questions?
tom.vancutsem@kuleuven.be

tvcutsem.github.io be.linkedin.com/in/tomvc github.com/tvcutsem x.com/tvcutsem @tvcutsem@techhub.social

Further Reading

- Mark Miller, Ka-Ping Yee, Jonathan Shapiro, “Capability Myths Demolished”: https://srl.cs.jhu.edu/pubs/SRL2003-02.pdf
- Compartments: https://github.com/tc39/proposal-compartments and https://github.com/Agoric/ses-shim
- ShadowRealms: https://github.com/tc39/proposal-realms and github.com/Agoric/realms-shim
- Hardened JS (SES): https://github.com/tc39/proposal-ses and https://github.com/endojs/endo/tree/master/packages/ses
- Subsetting ECMAScript: https://github.com/Agoric/Jessie
- Kris Kowal (Agoric): “Hardened JavaScript” https://www.youtube.com/watch?v=RoodZSIL-DE
- Making Javascript Safe and Secure: Talks by Mark S. Miller (Agoric), Peter Hoddie (Moddable), and Dan Finlay (MetaMask): https://www.youtube.com/playlist?list=PLzDw4TTug5O25J5M3fwErKlmrjOrqGikj
- Moddable: XS: Secure, Private JavaScript for Embedded IoT: https://blog.moddable.com/blog/secureprivate/
- Membranes in JavaScript: tvcutsem.github.io/js-membranes and tvcutsem.github.io/membranes
- Caja: https://developers.google.com/caja (Capability-secure subset of JavaScript)
- Chip Morningstar, “What are capabilities”: http://habitatchronicles.com/2017/05/what-are-capabilities/ (broad historical perspective)
- Why KeyKOS is fascinating: https://github.com/void4/notes/issues/41 (sketches the early history of capabilities as used in operating systems)
- Neil Madden, “Capability-Based Security and Macaroons” https://freecontent.manning.com/capability-based-security-and-macaroons/#id_ftn3 (capabilities in REST APIs)

Acknowledgements

- Mark S. Miller (for the inspiring and ground-breaking work on Object-capabilities, Robust Composition, E, Caja, JavaScript and Secure ECMAScript)
- Marc Stiegler's "PictureBook of secure cooperation" (2004) is a great source of inspiration for patterns of robust composition
- Doug Crockford's "JS: the Good Parts" and "How JS Works" books provide a highly opinionated take on how to write clean, good, robust JavaScript code
- Kate Sills and Kris Kowal at Agoric for helpful comments on earlier versions of these slides
- The Cap-talk and Friam community for inspiration on capability-security and capability-secure design patterns
- TC39 and the es-discuss community, for the interactions during the design of ECMAScript 2015, and in particular all the feedback on the Proxy API
- The SES secure coding guide: https://github.com/endojs/endo/blob/master/packages/ses/docs/secure-coding-guide.md