Designing “least-authority” JavaScript apps

Tom Van Cutsem
KU Leuven

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

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

“Security is just an extreme form of Modularity”

- Mark S. Miller
(Chief Scientist, Agoric)

Modularity: avoid needless software dependencies to protect against *unintended bugs*

Security: avoid needless software dependencies to protect against *deliberate exploits*

KU LEUVEN DistriNet

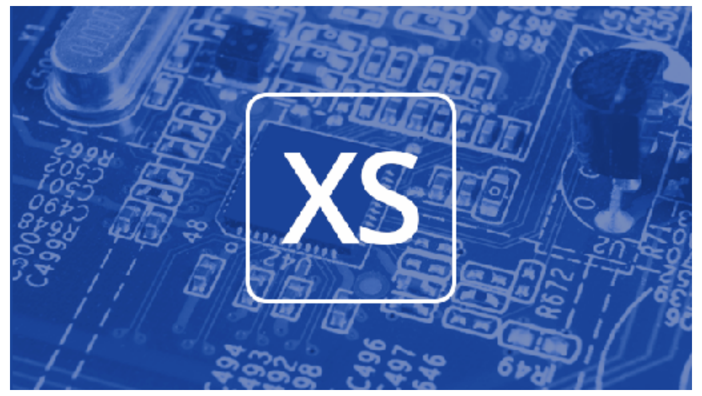
This Lecture

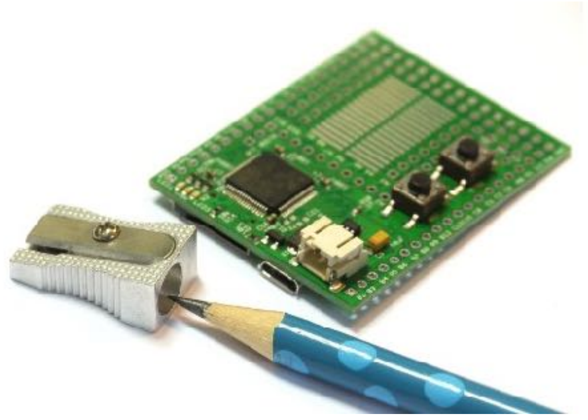
- Part I: **why module isolation** is critical to modern JavaScript applications
- Part II: the **Principle of Least Authority**, by example (in JavaScript)
- Part III: safely composing modules using **least-authority patterns**

KU LEUVEN DistriNet

Part I
Why module isolation is critical to modern JavaScript applications

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





React Native

Apache Cordova

WinJS

Microsoft Edge
WebView2 Runtime

node js

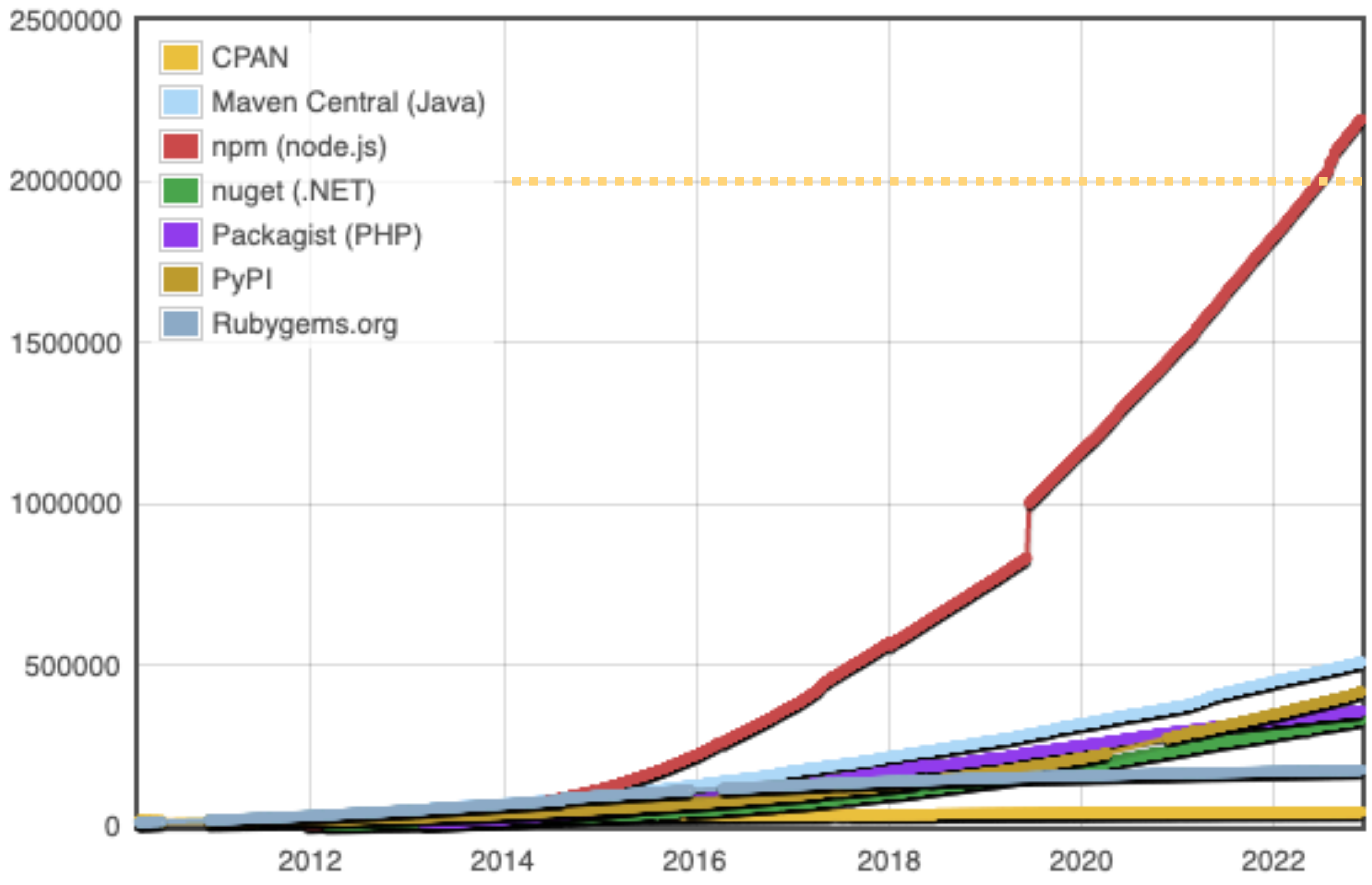
GraalVM

mongoDB

Apache CouchDB relax

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)

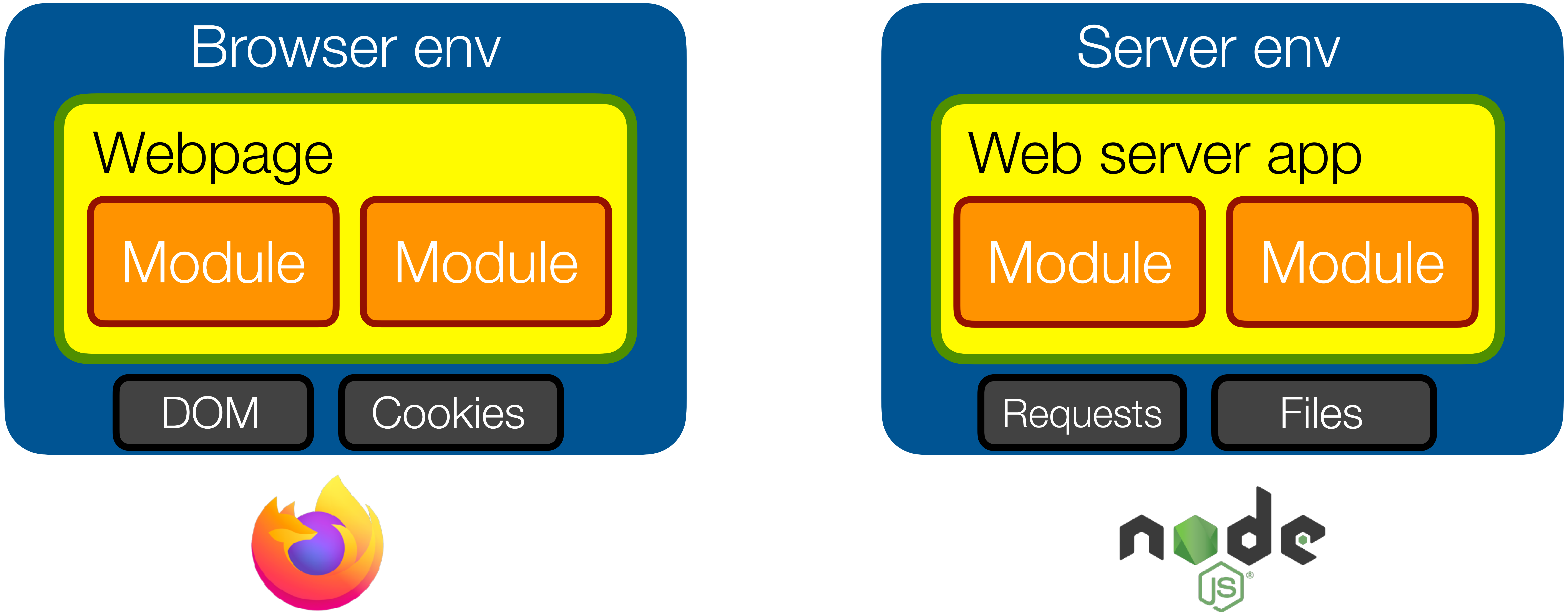
"The average modern web application has over 1000 modules [...] ***97% of the code in a modern web application comes from npm***. An individual developer is responsible only for the final 3% that makes their application unique and useful."

(source: npm blog, December 2018)

KU LEUVEN DistriNet

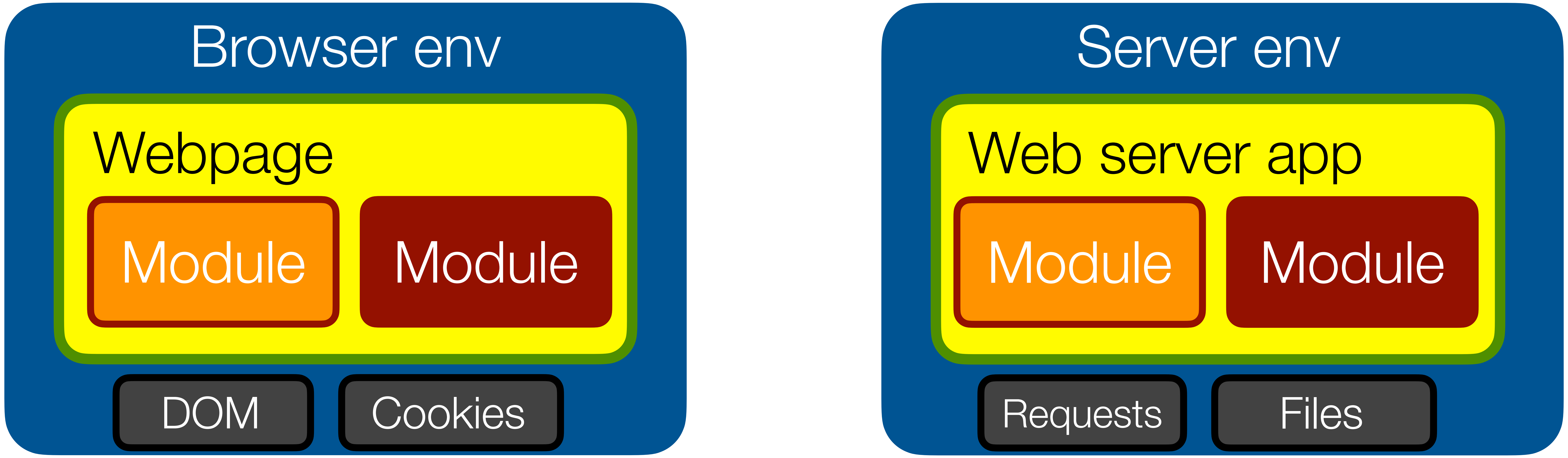
Composing modules: it’s all about **trust**

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

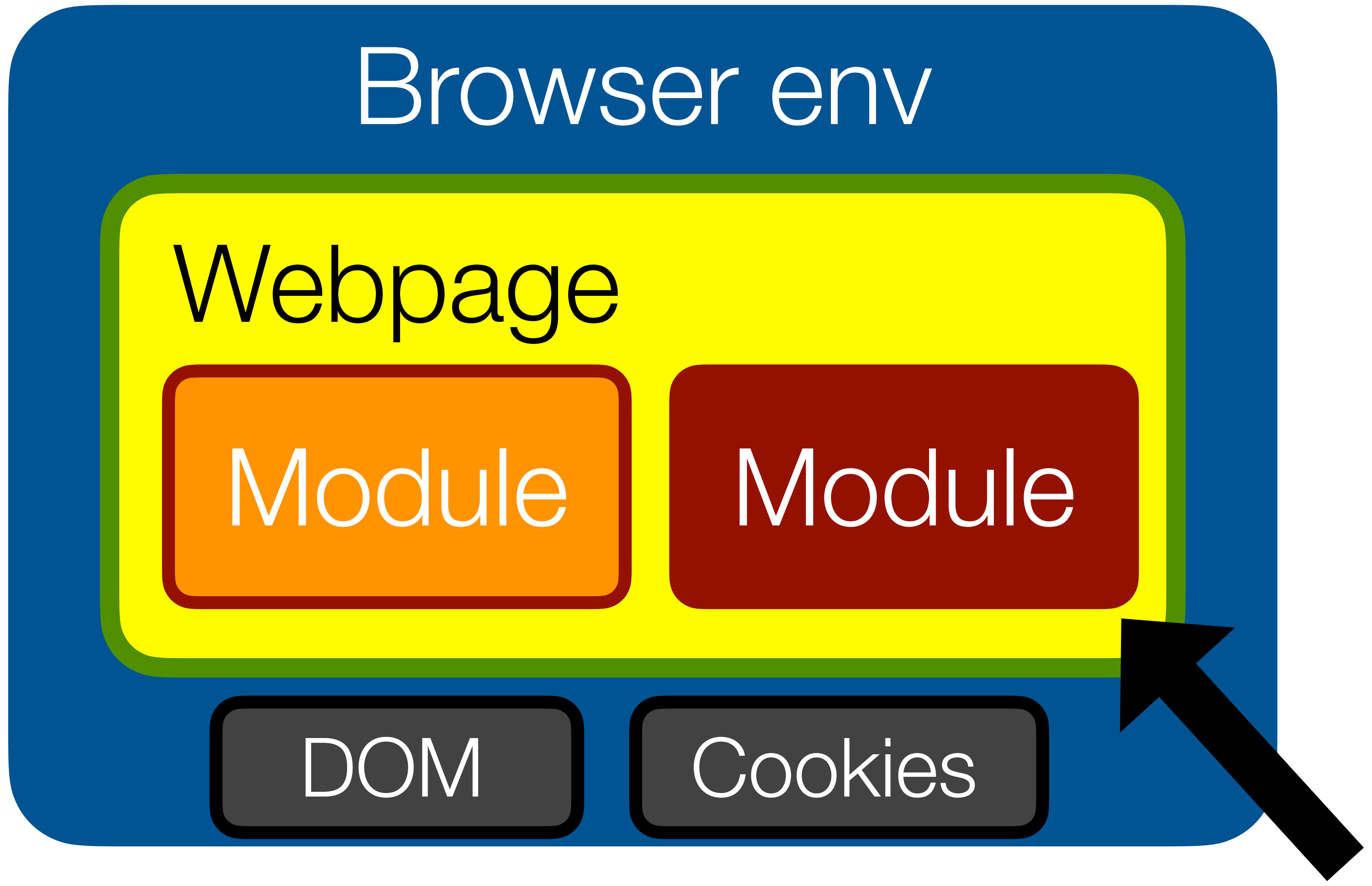


What can happen when a module goes **rogue**?

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



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




The New York Times
@nytimes

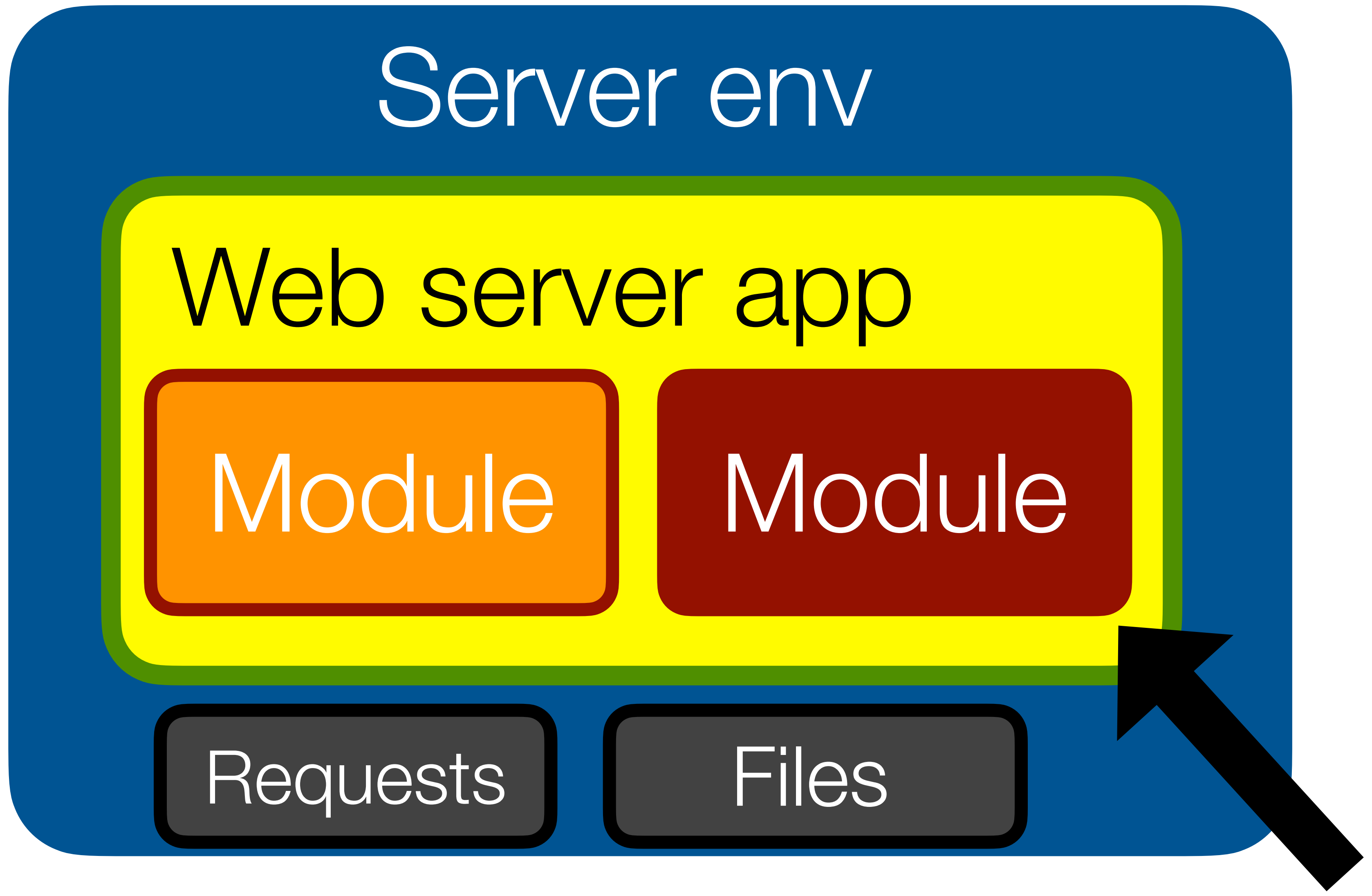
Attn: NYTimes.com readers: Do not click pop-up box warning about a virus -- it's an unauthorized ad we are working to eliminate.

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)

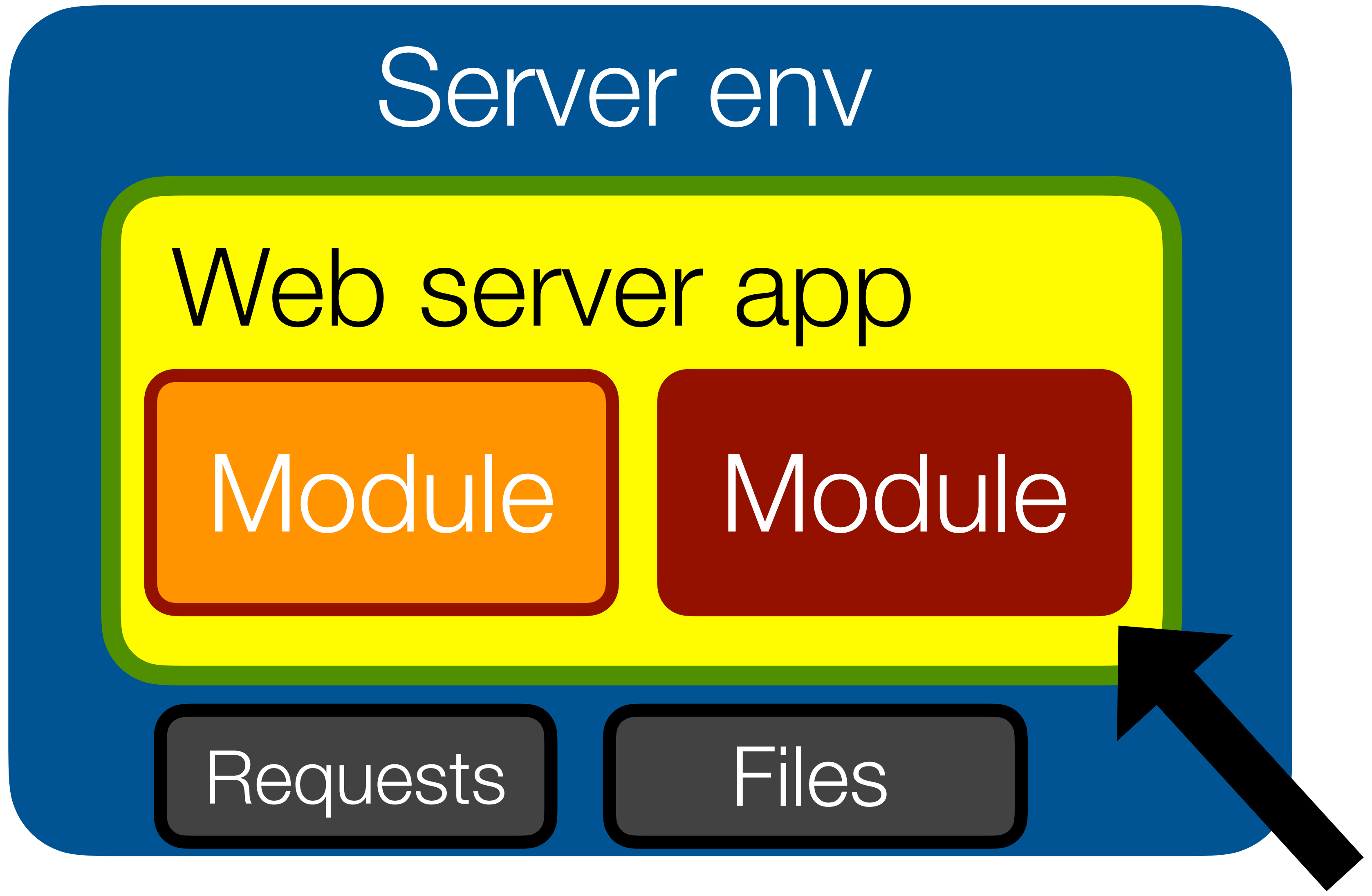
Node.js package tried to plunder Bitcoin wallets

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

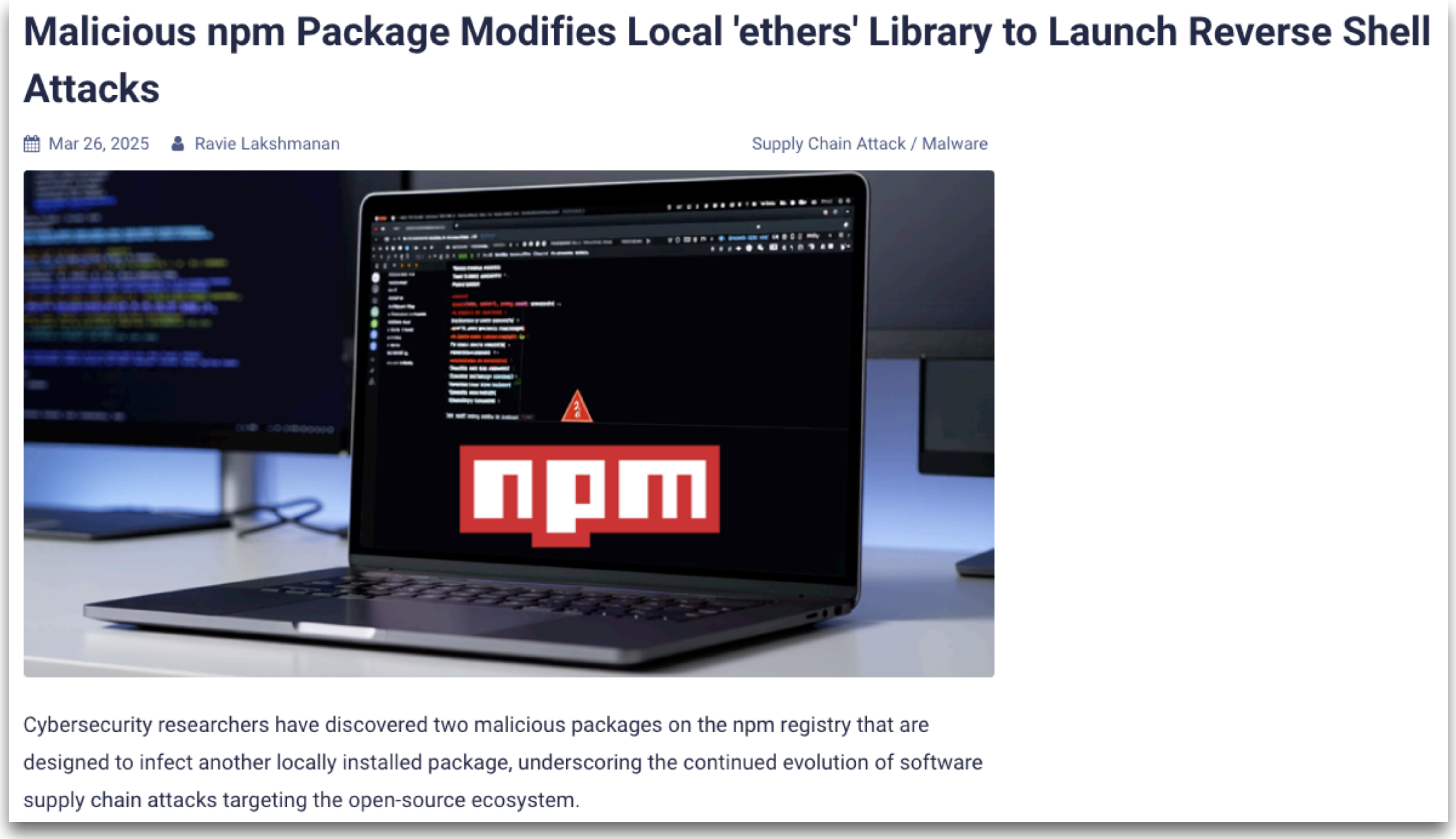
(source: theregister.co.uk)

KU LEUVEN DistriNet

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



npm install ethers


Malicious npm Package Modifies Local 'ethers' Library to Launch Reverse Shell Attacks

Mar 26, 2025 · Ravie Lakshmanan · Supply Chain Attack / Malware

Cybersecurity researchers have discovered two malicious packages on the npm registry that are designed to infect another locally installed package, underscoring the continued evolution of software supply chain attacks targeting the open-source ecosystem.

(source: https://thehackernews.com/ March 2025)

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...

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

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. "

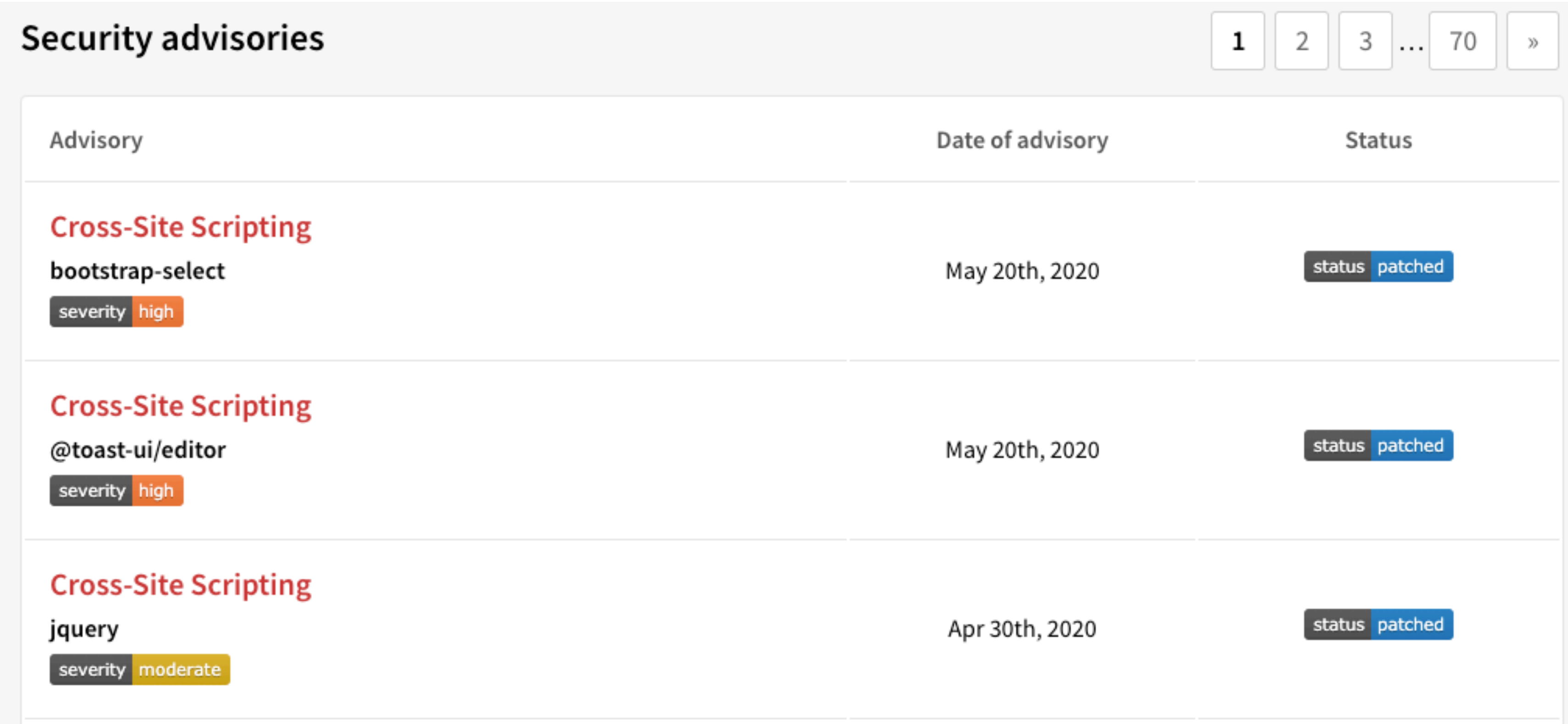
Tomislav Peričin, co-founder and chief software architect at ReversingLabs, noted how in the case of SolarWinds, the trusted source was pushing infected software. Catching those kinds of mistakes requires a focus on how code behaves, regardless of where it came from.

> *"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*

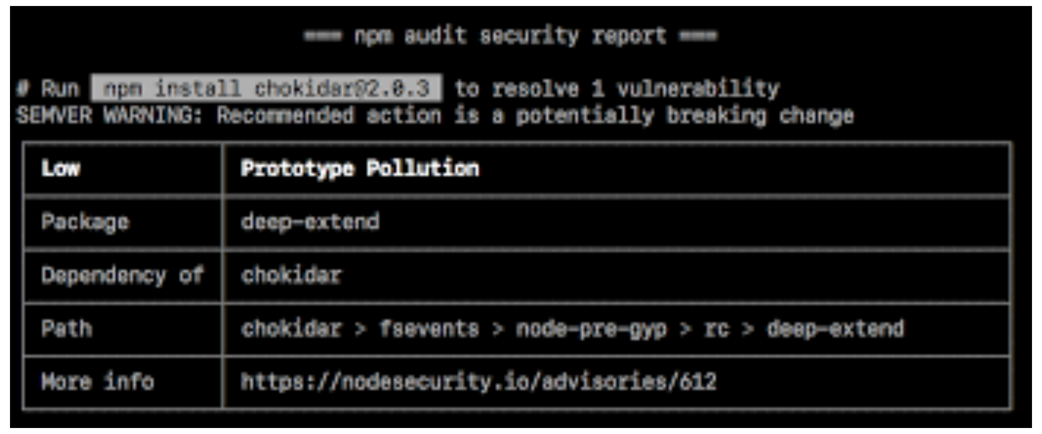
Increasing awareness

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

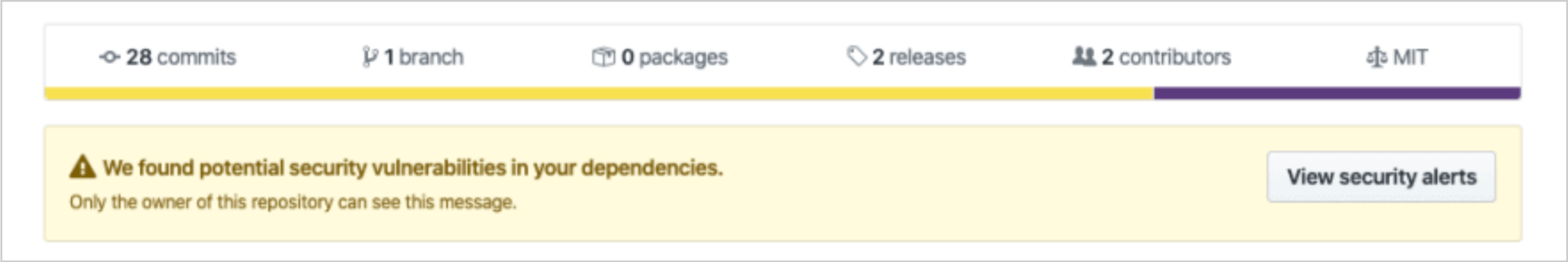
npm security advisories



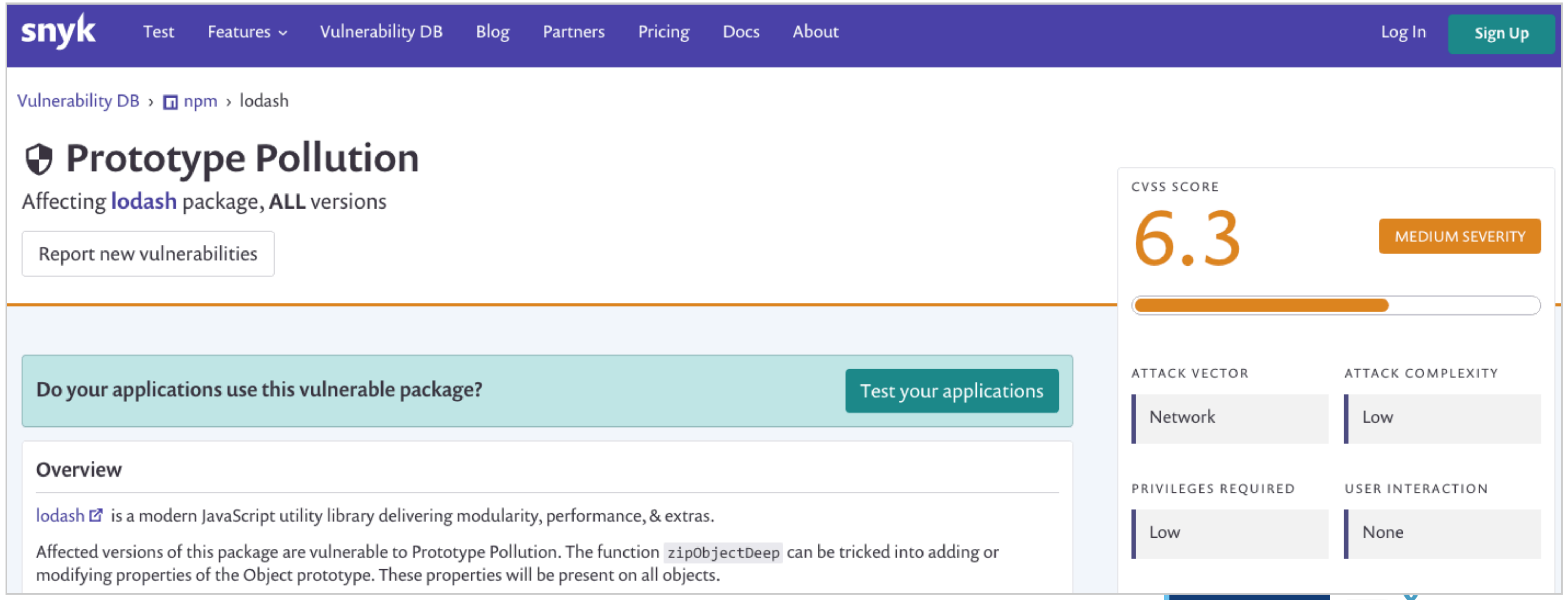
npm audit



GitHub security alerts

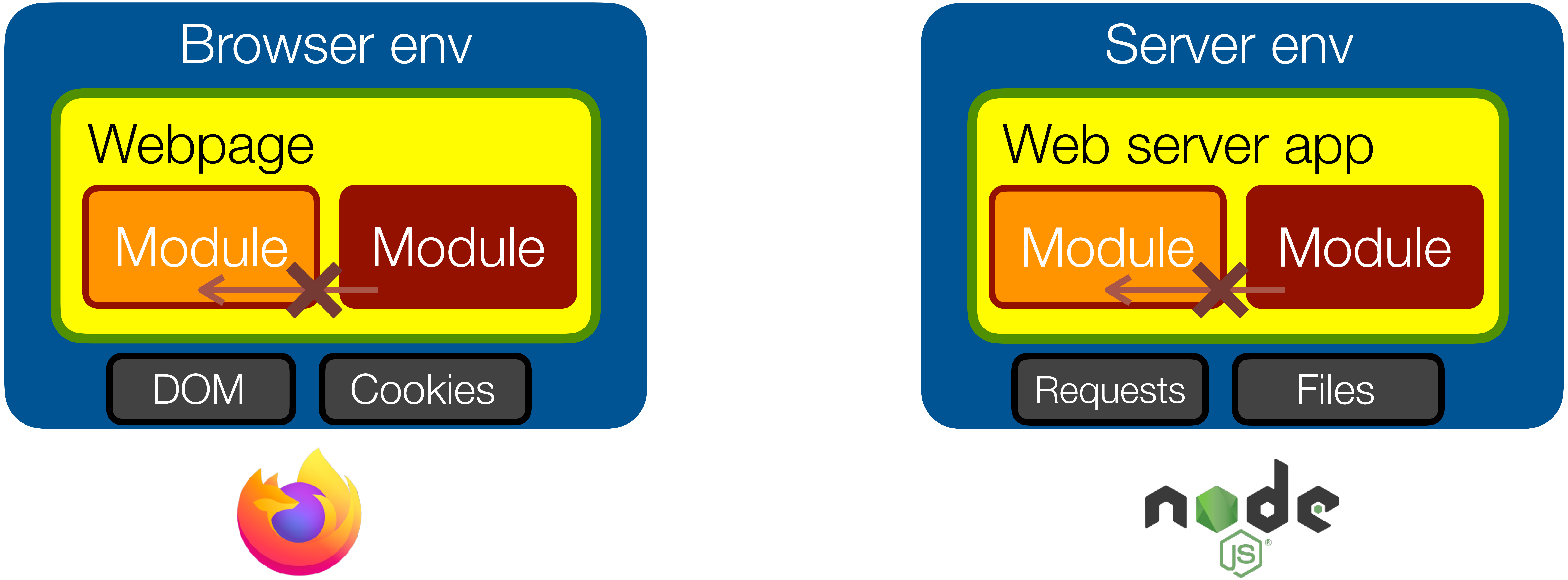


Snyk vulnerability DB

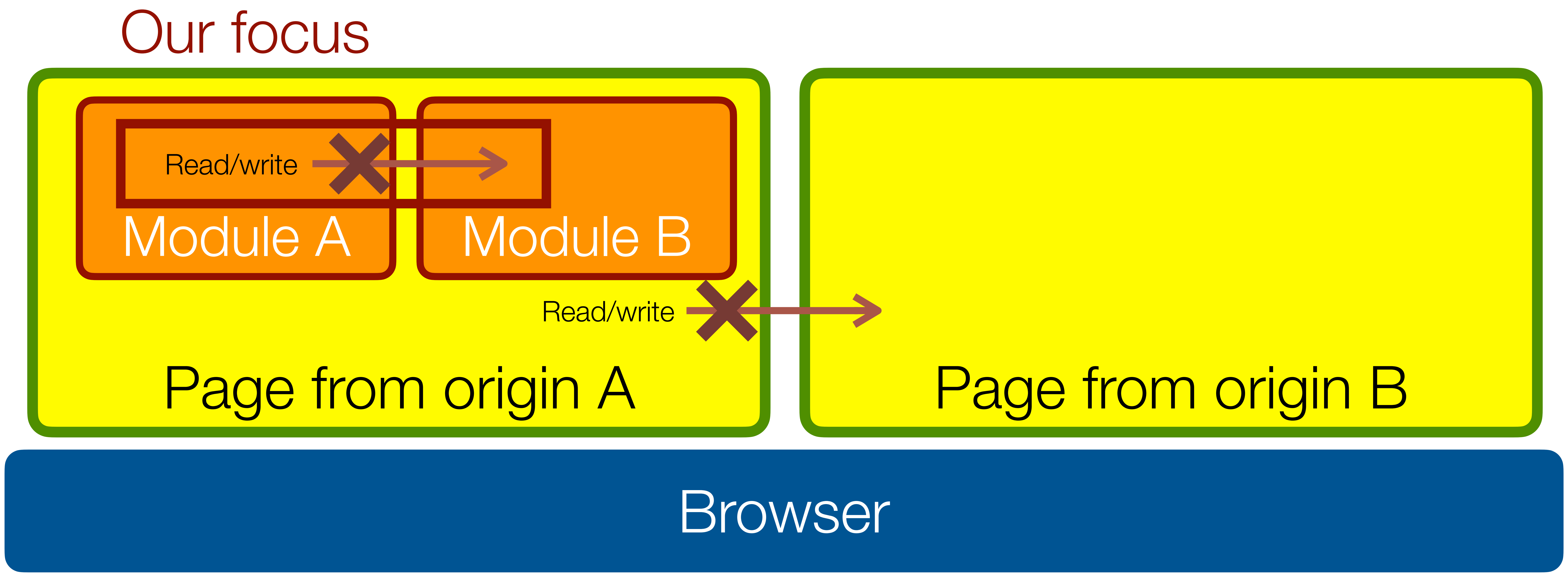


Avoiding interference is the name of the game

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



We'll need more than simply relying on Browser Same-origin Policy

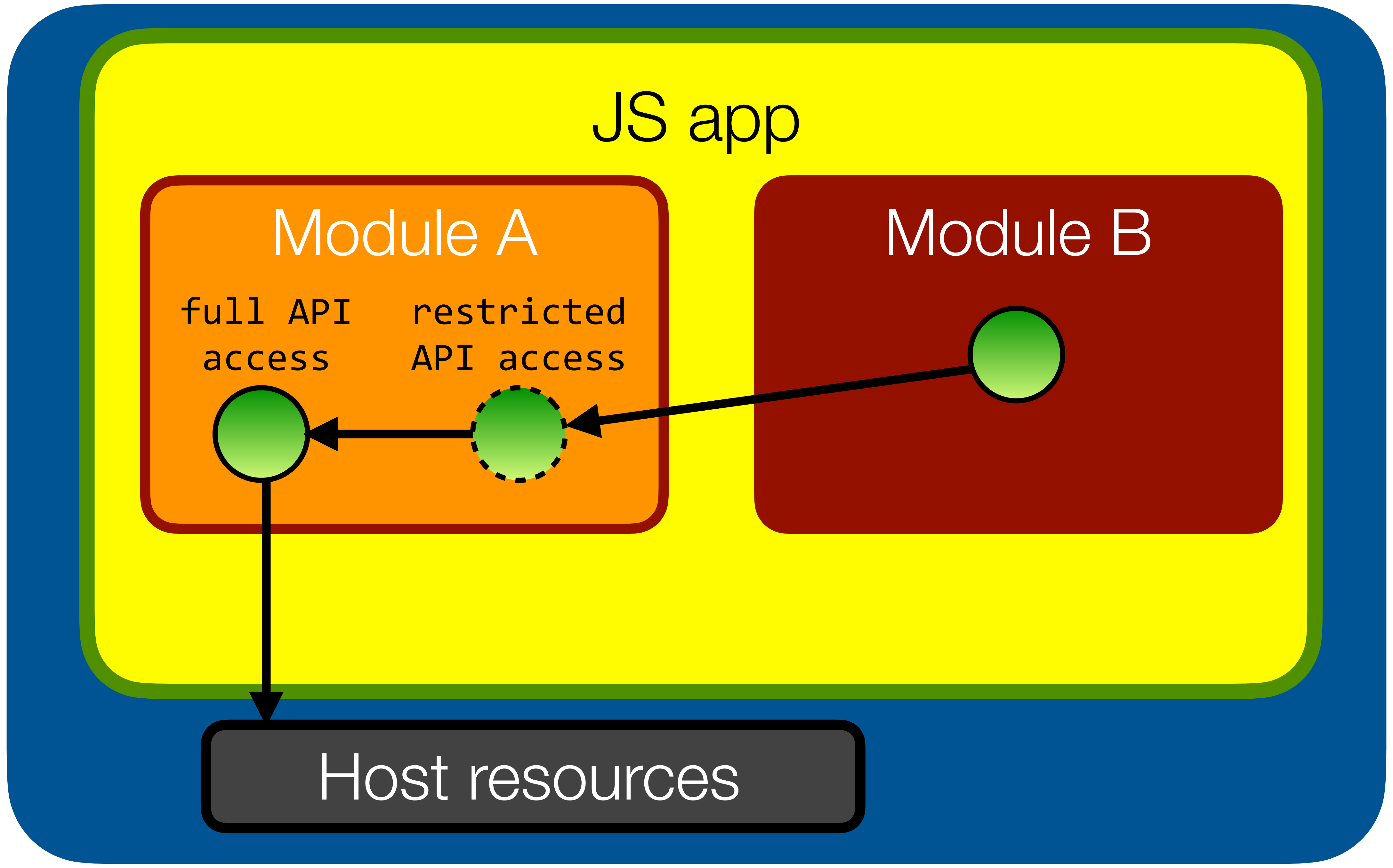


Part II
The Principle of Least Authority, by example (in JavaScript)

KU LEUVEN DistriNet

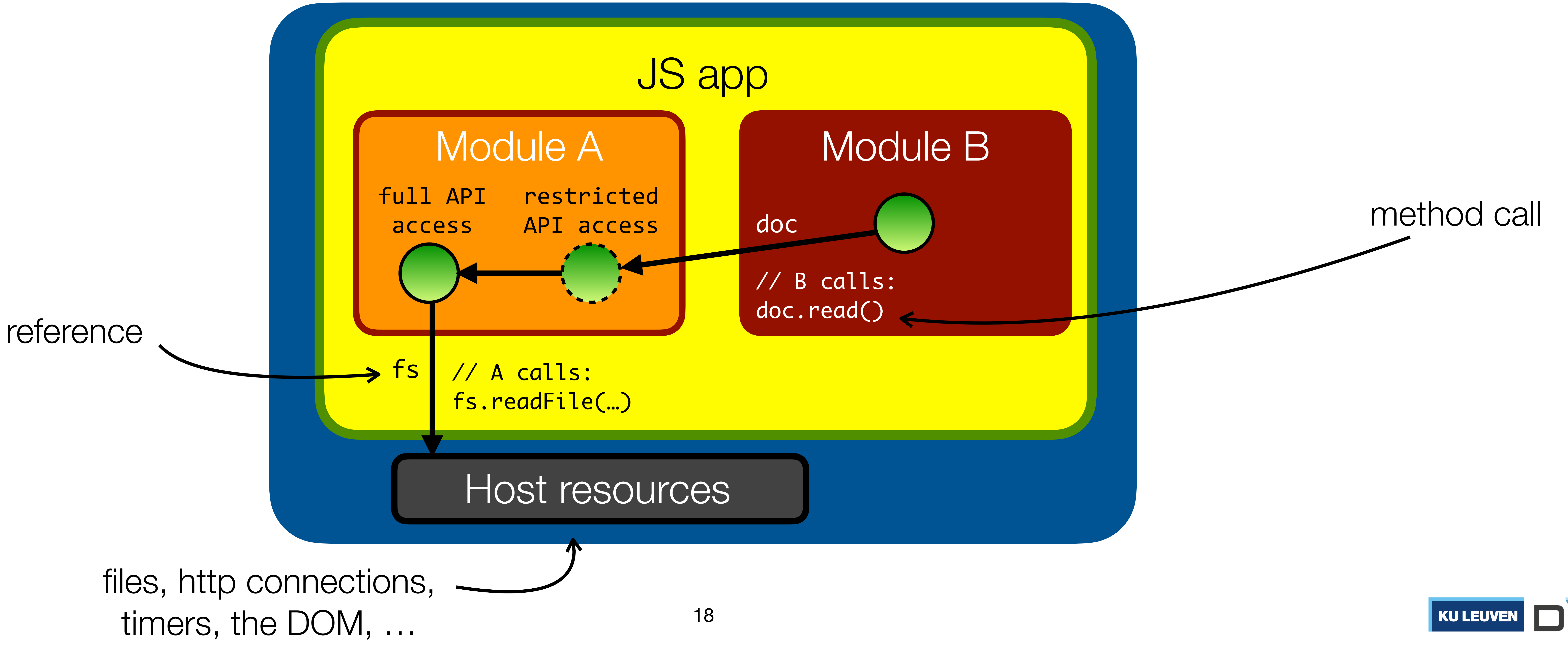
Principle of Least Authority (POLA)

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

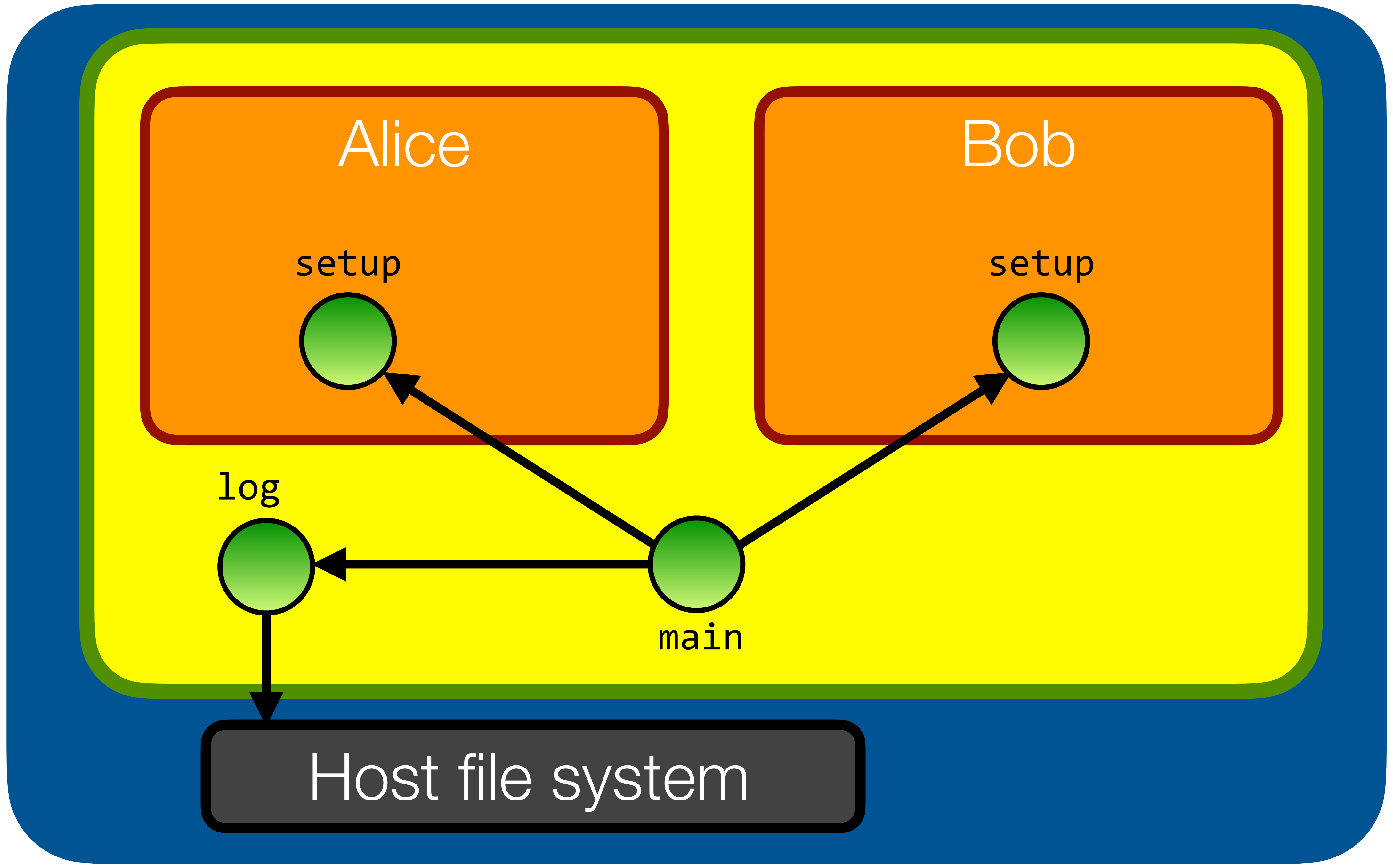


What is “authority” in a JavaScript app?

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



Delegating authority == sharing references, under the right assumptions



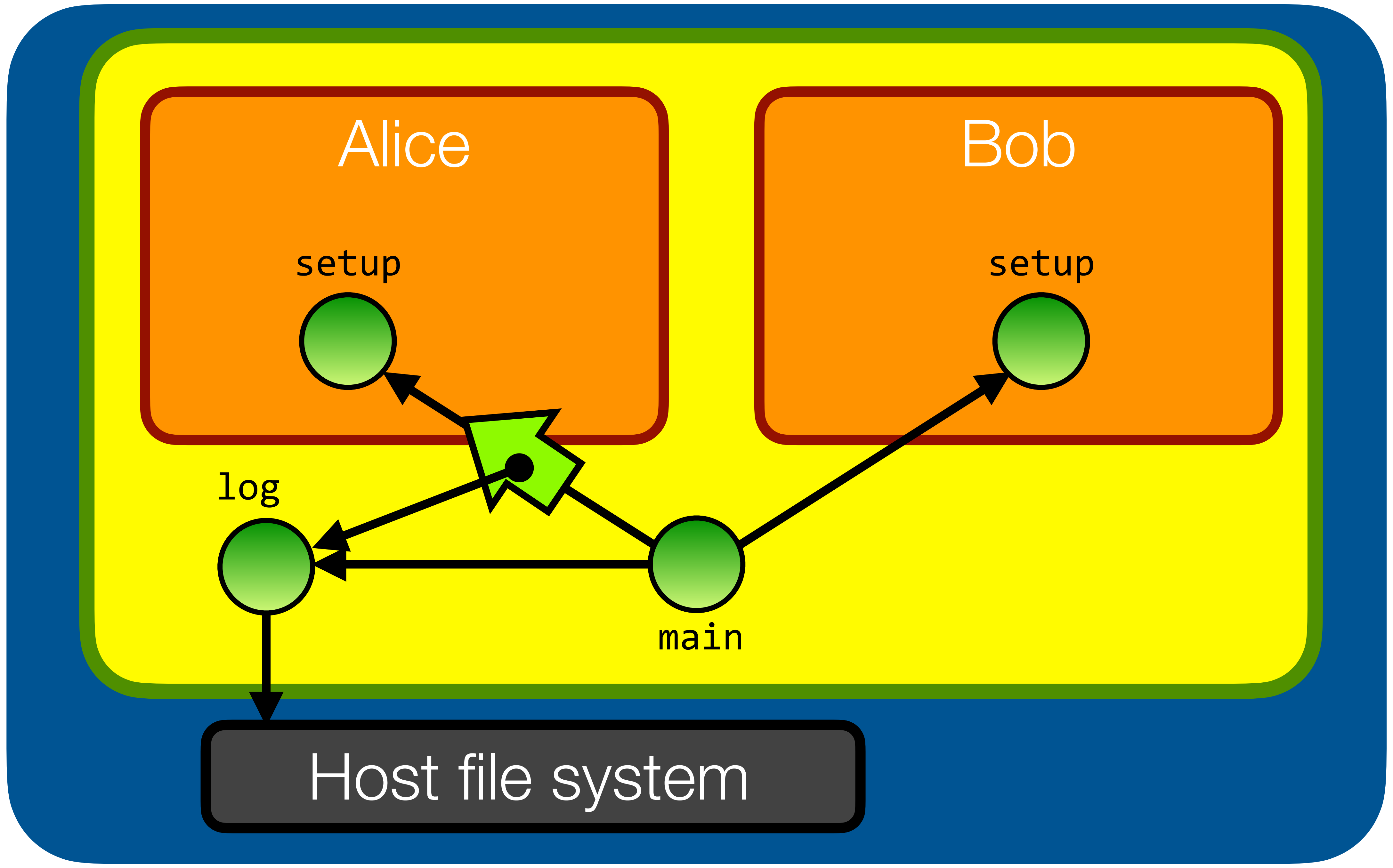
Consider an app maintaining a message log.

The app loads two untrusted modules Alice and Bob.

We would like Alice and Bob to *only* have access to this log file *and nothing more*.

```
// in our app's main function:
let log = new Log();
alice.setup(log)
bob.setup(log)
```

Delegating authority == sharing references, under the right assumptions



Consider an app maintaining a message log.

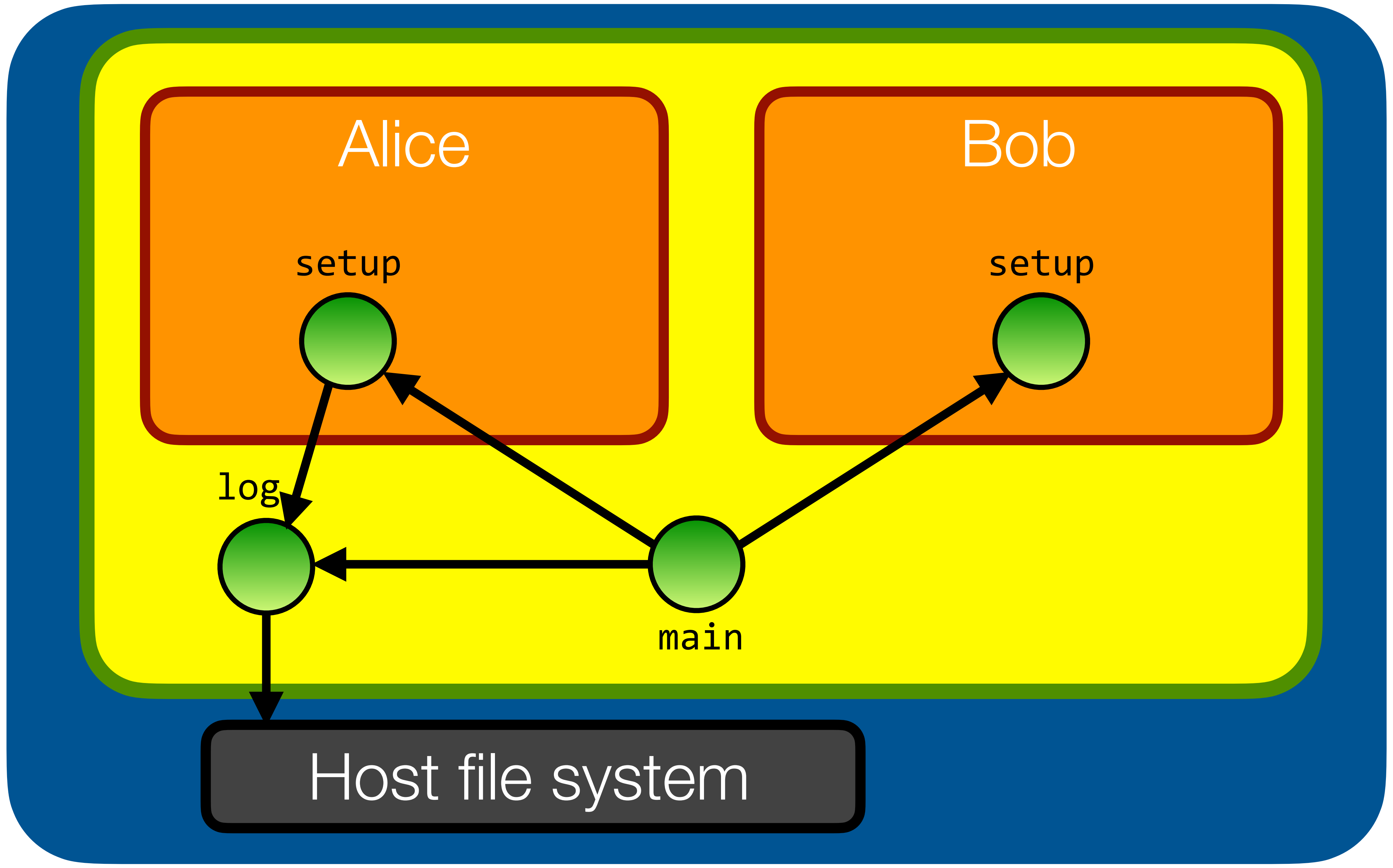
The app loads two untrusted modules Alice and Bob.

We would like Alice and Bob to *only* have access to this log file *and nothing more*.

```
// in our app's main function:
let log = new Log();
alice.setup(log)
bob.setup(log)
```


KU LEUVEN DistriNet

Delegating authority == sharing references, under the right assumptions



Consider an app maintaining a message log.

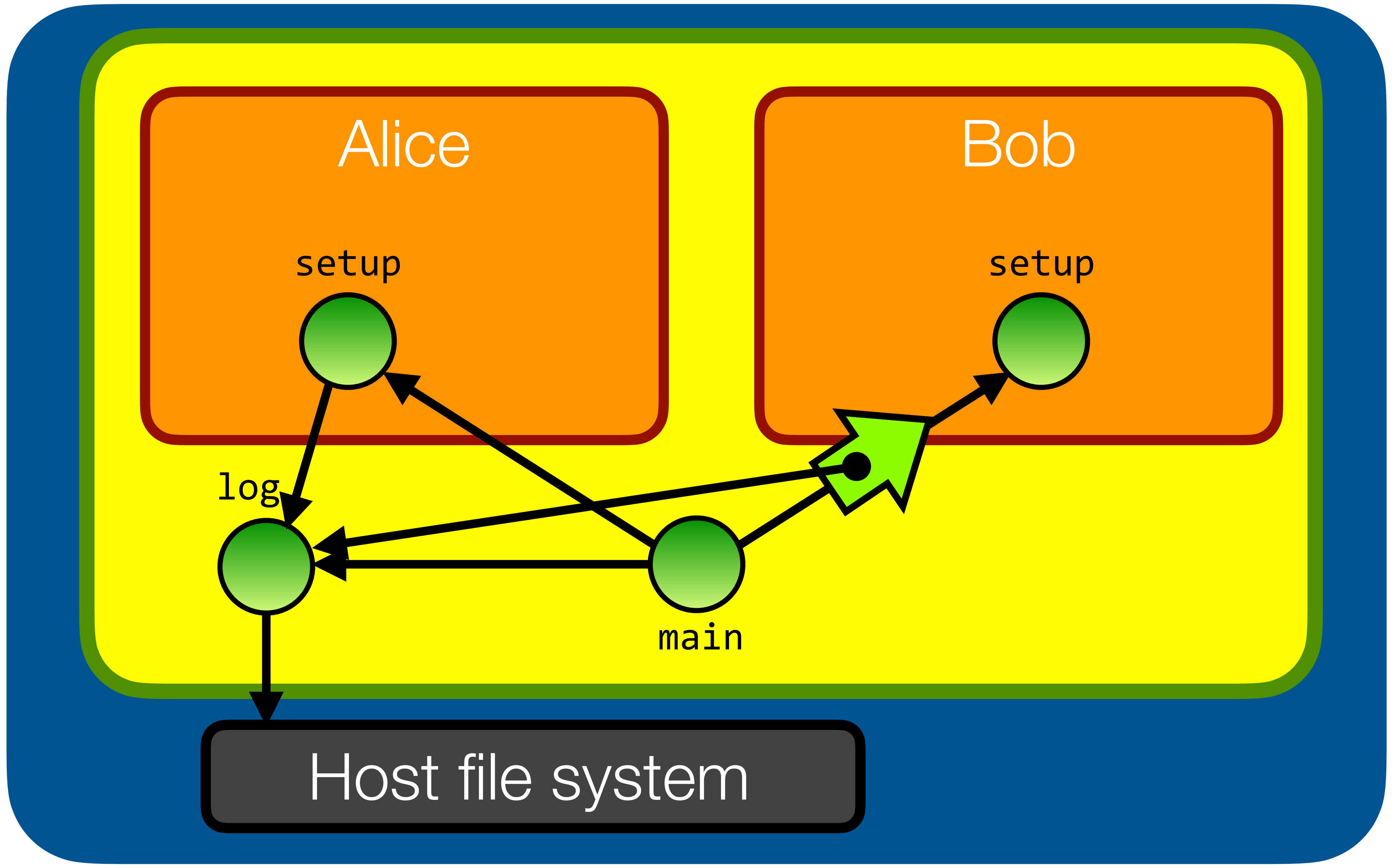
The app loads two untrusted modules Alice and Bob.

We would like Alice and Bob to *only* have access to this log file *and nothing more*.

```
// in our app’s main function:
let log = new Log();
alice.setup(log)
bob.setup(log)
```


KU LEUVEN DistriNet

Delegating authority == sharing references, under the right assumptions



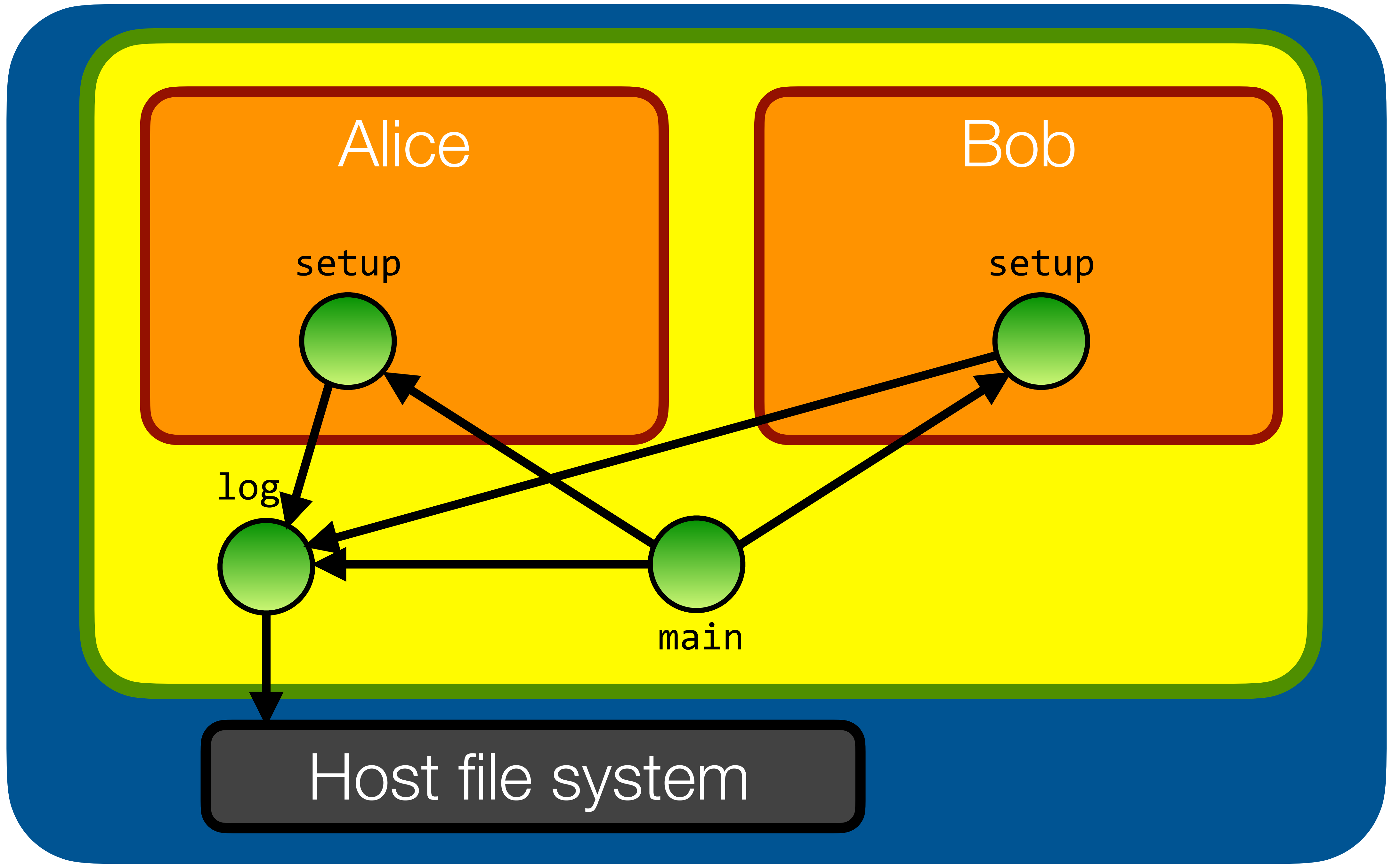
Consider an app maintaining a message log.

The app loads two untrusted modules Alice and Bob.

We would like Alice and Bob to *only* have access to this log file *and nothing more*.

```
// in our app's main function:
let log = new Log();
alice.setup(log)
bob.setup(log)
```

Delegating authority == sharing references, under the right assumptions



Consider an app maintaining a message log.

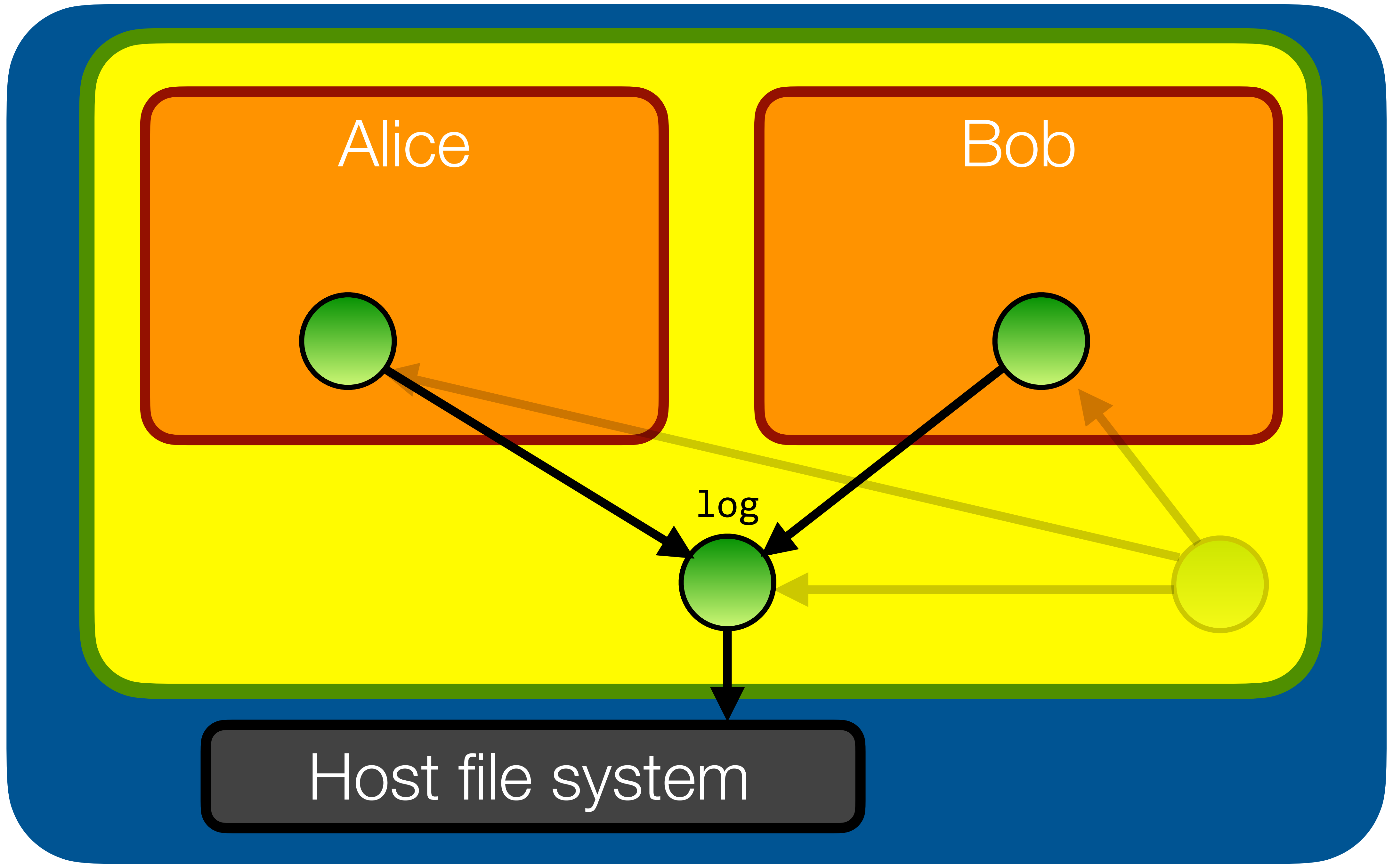
The app loads two untrusted modules Alice and Bob.

We would like Alice and Bob to *only* have access to this log file *and nothing more*.

```
// in our app's main function:
let log = new Log();
alice.setup(log)
bob.setup(log)
```


KU LEUVEN DistriNet

Delegating authority == sharing references, under the right assumptions



Consider an app maintaining a message log.

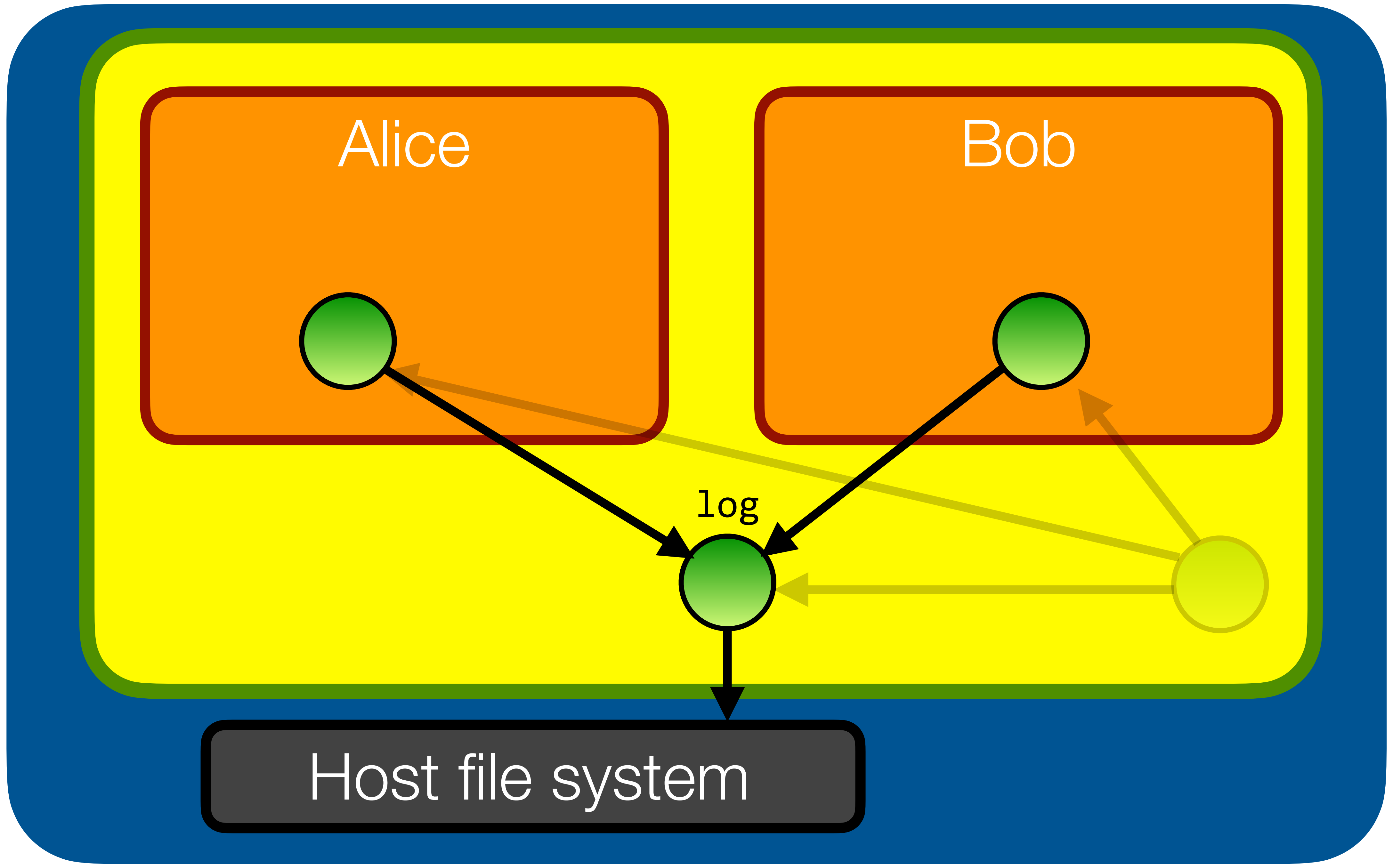
The app loads two untrusted modules Alice and Bob.

We would like Alice and Bob to *only* have access to this log file *and nothing more*.

```
// in our app's main function:
let log = new Log();
alice.setup(log)
bob.setup(log)
```


KU LEUVEN DistriNet

What are our assumptions?



- Alice and Bob cannot create references to other app objects.
✅ JavaScript is **memory-safe.** References are unforgeable.

- The log can hide its reference to the host file system from Alice and Bob.
✅ JavaScript has strict **lexical scoping** rules that support **hiding** pointers in **private** local variables.

- Alice and Bob cannot communicate via global mutable vars.
⚠️ App must ensure that there is **no global mutable state**!

- Alice and Bob cannot circumvent the log's public API.
⚠️ App must ensure that all exported API objects are **immutable**.

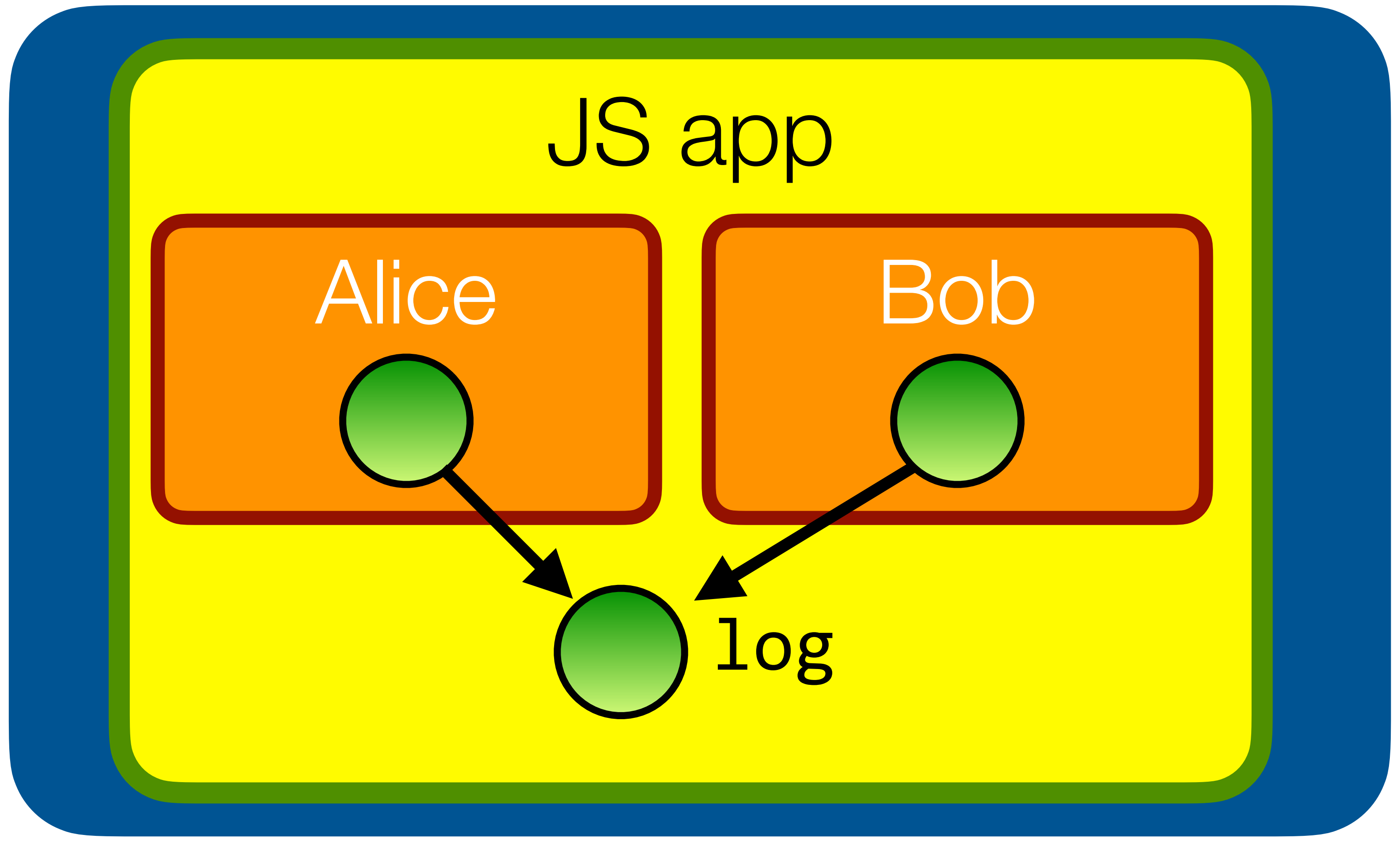
- Alice and Bob cannot access the host file system by default.
⚠️ App must ensure each module starts out with **no references to powerful globals** by default.

KU LEUVEN DistriNet

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.setup(log);
bob.setup(log);
```

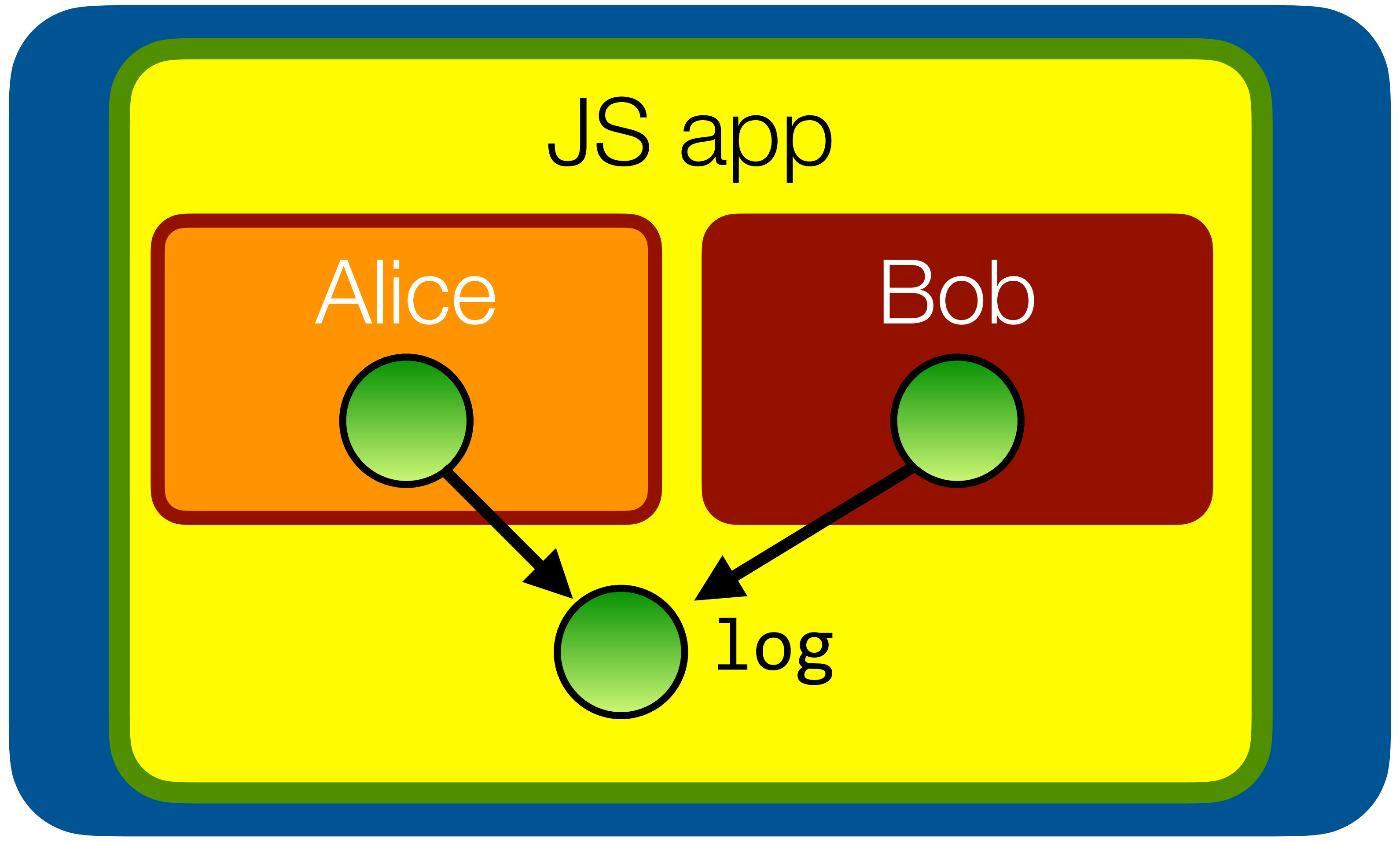


KU LEUVEN DistriNet

Running example: apply POLA to a basic shared log

If Bob goes rogue, what could go wrong?

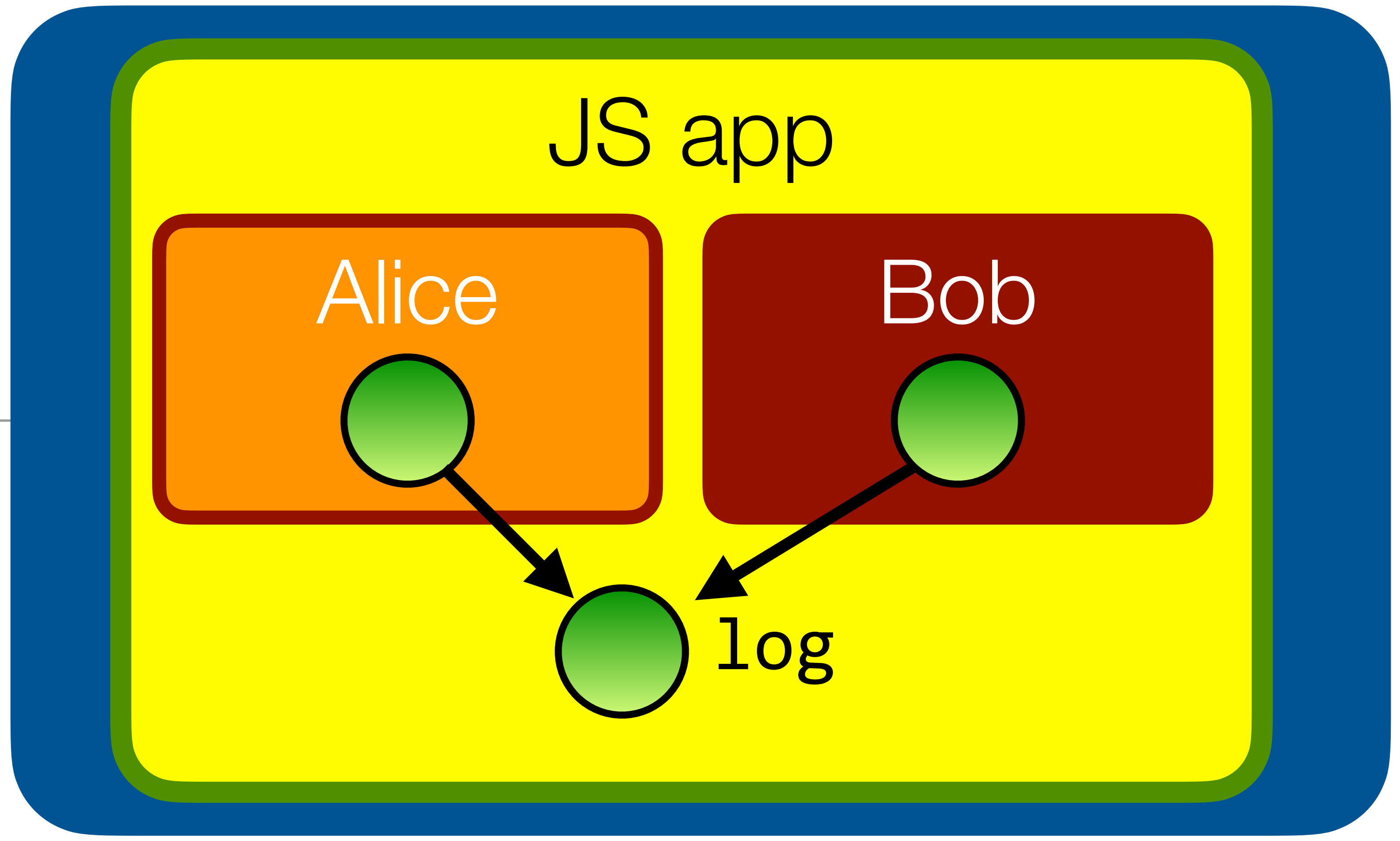
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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



KU LEUVEN DistriNet

Bob has way too much authority!

If Bob goes rogue, what could go wrong?

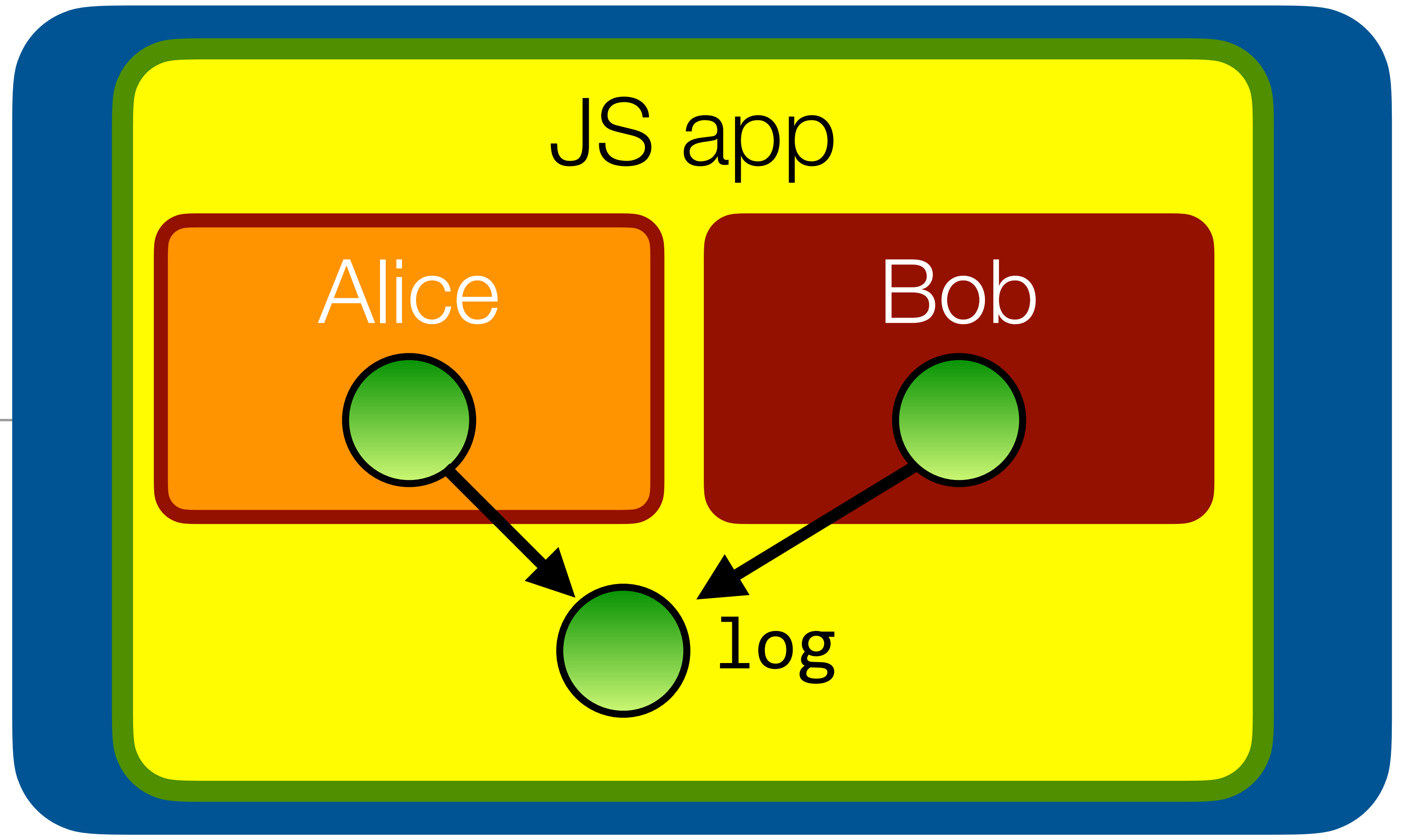



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

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

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

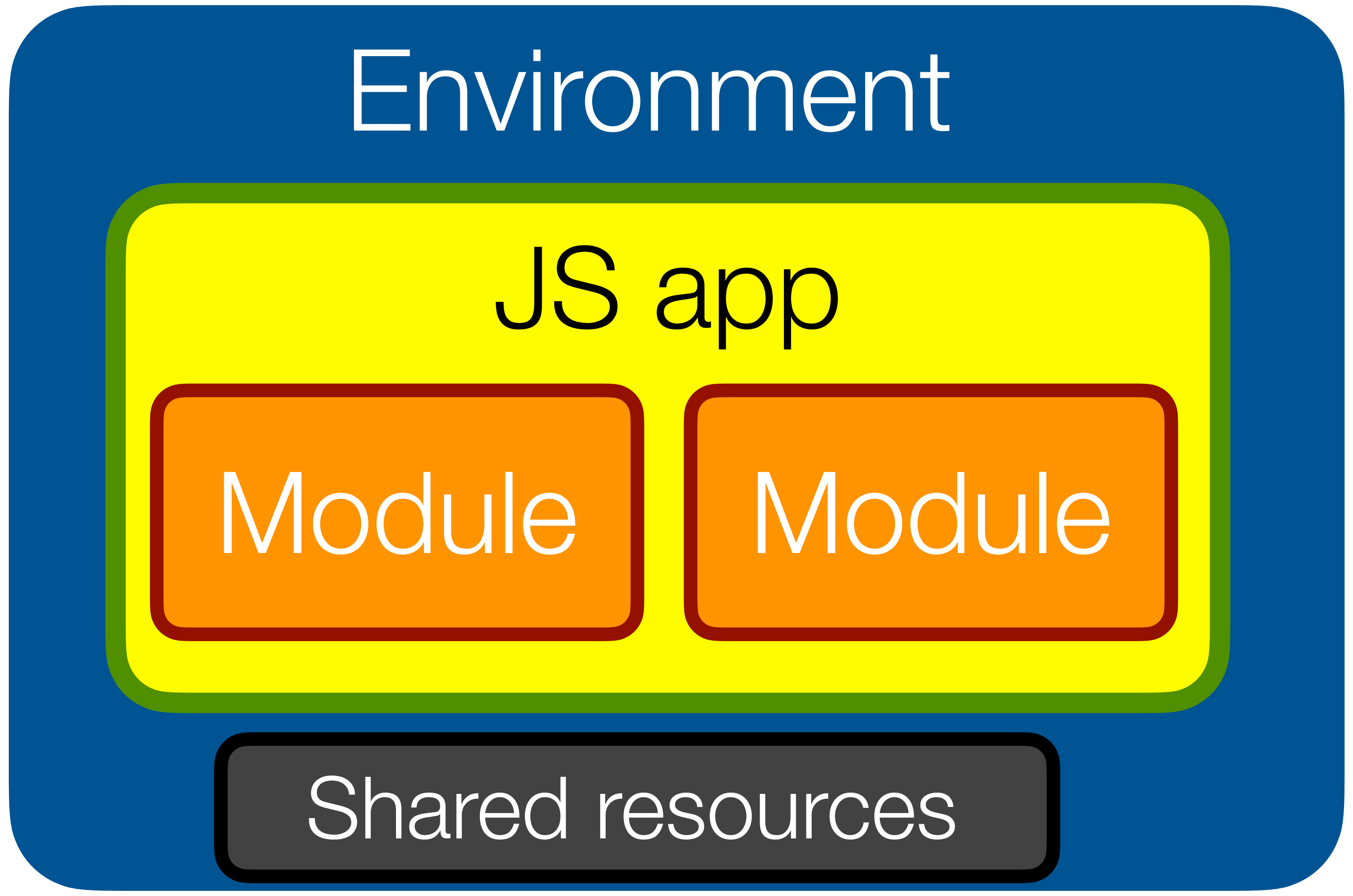


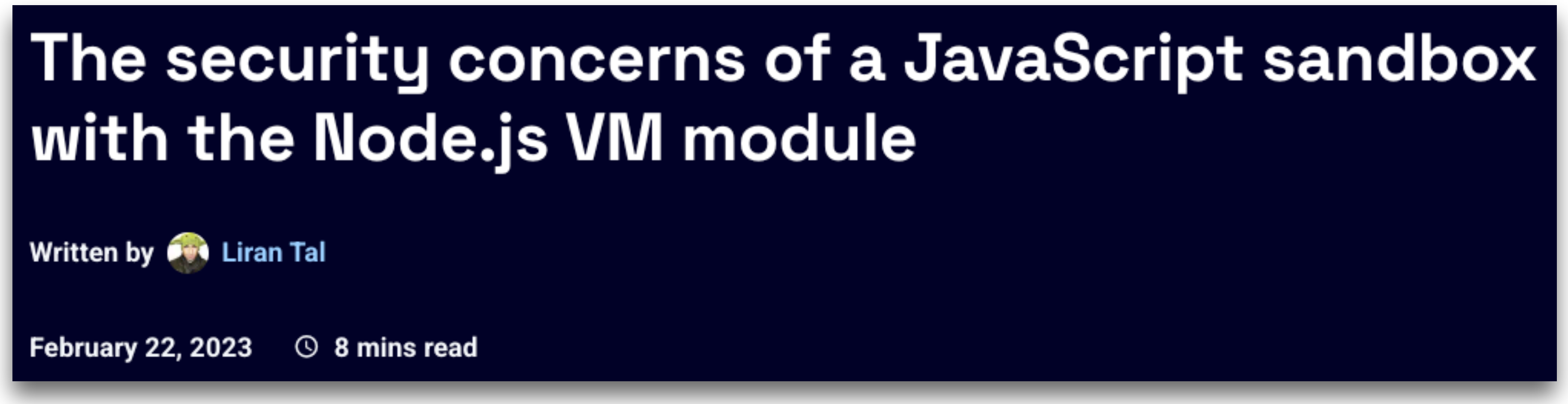

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

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

// Bob can replace the ‘write’ function (api poisoning)
log.write = function(msg) {
  console.log(“I’m not logging anything”);
}
// Bob can replace the built-ins (prototype poisoning)
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**: *not* designed for running untrusted code! See article on Snyk blog ->



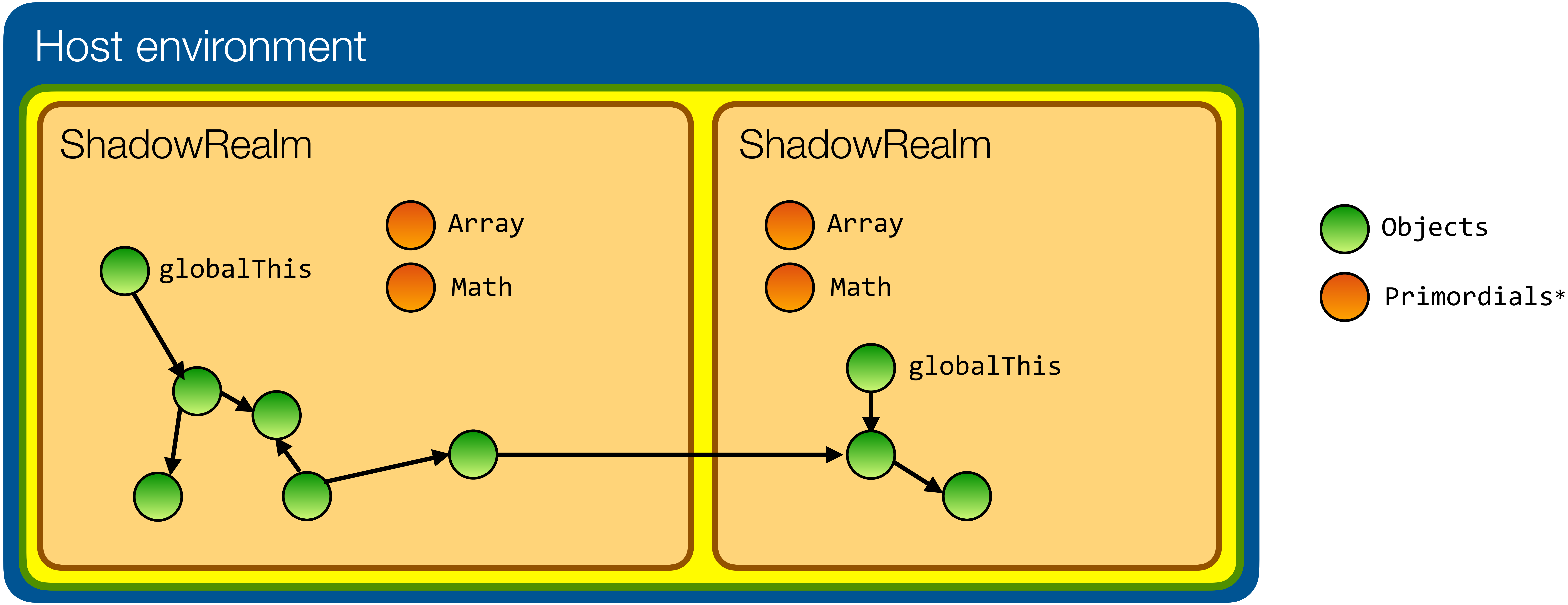


(Source: snyk.io blog, 2023)

KU LEUVEN DistriNet

ShadowRealms (ECMA TC39 Stage 2.7 proposal)

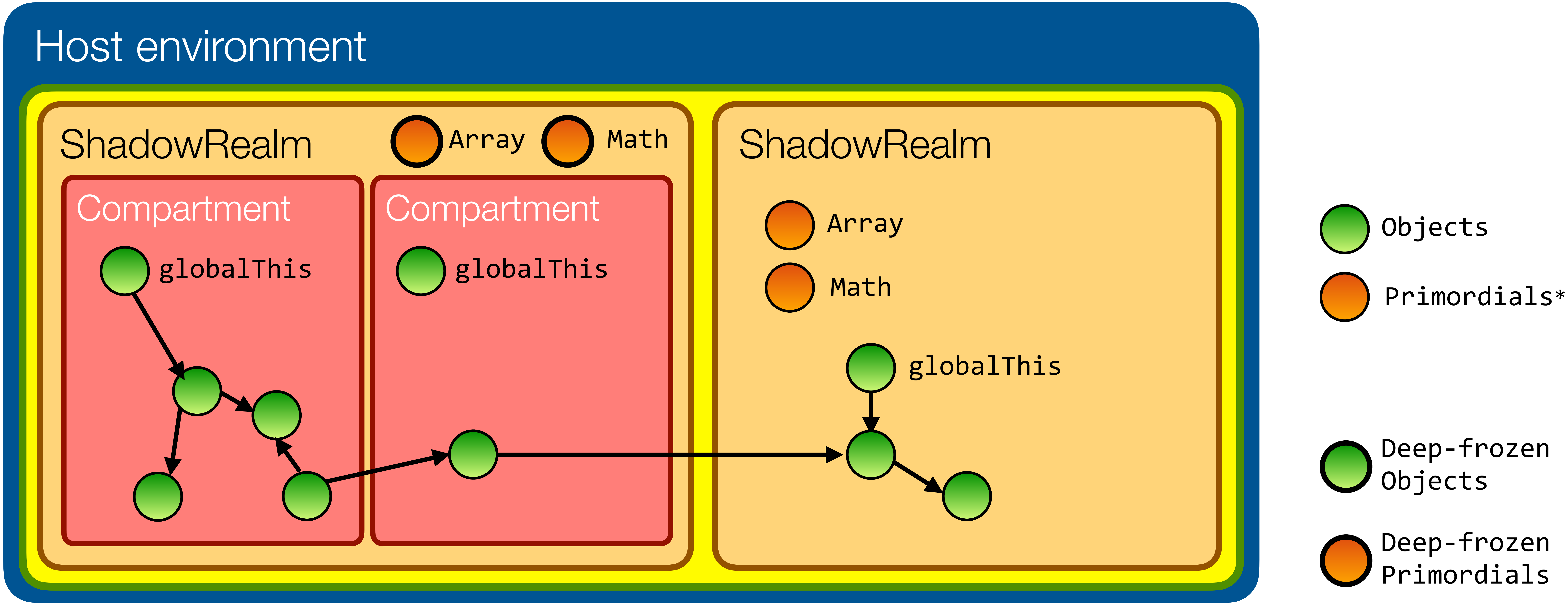
Intuitions: “iframe without DOM”, “principled version of node’s `vm` module”



* 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.

KU LEUVEN DistriNet

Hardened JavaScript is a secure subset of standard JavaScript

Full JavaScript

- everything mutable by default, can mess up the global environment
- powerful globals like `window` or `process` accessible by default
- no easy way to eval untrusted code in a sandboxed environment

Hardened JavaScript

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

JSON (Data only)

(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.

hardenedjs.org/

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 — JS

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

Today — moddable, salesforce

LavaMoat

- **Command-line tool** that puts each package dependency into its own hardened JS sandbox environment
- Auto-generates config file indicating authority needed by each package
- For node.js and Web. Plugs into build tools like Webpack

https://lavamoat.github.io/

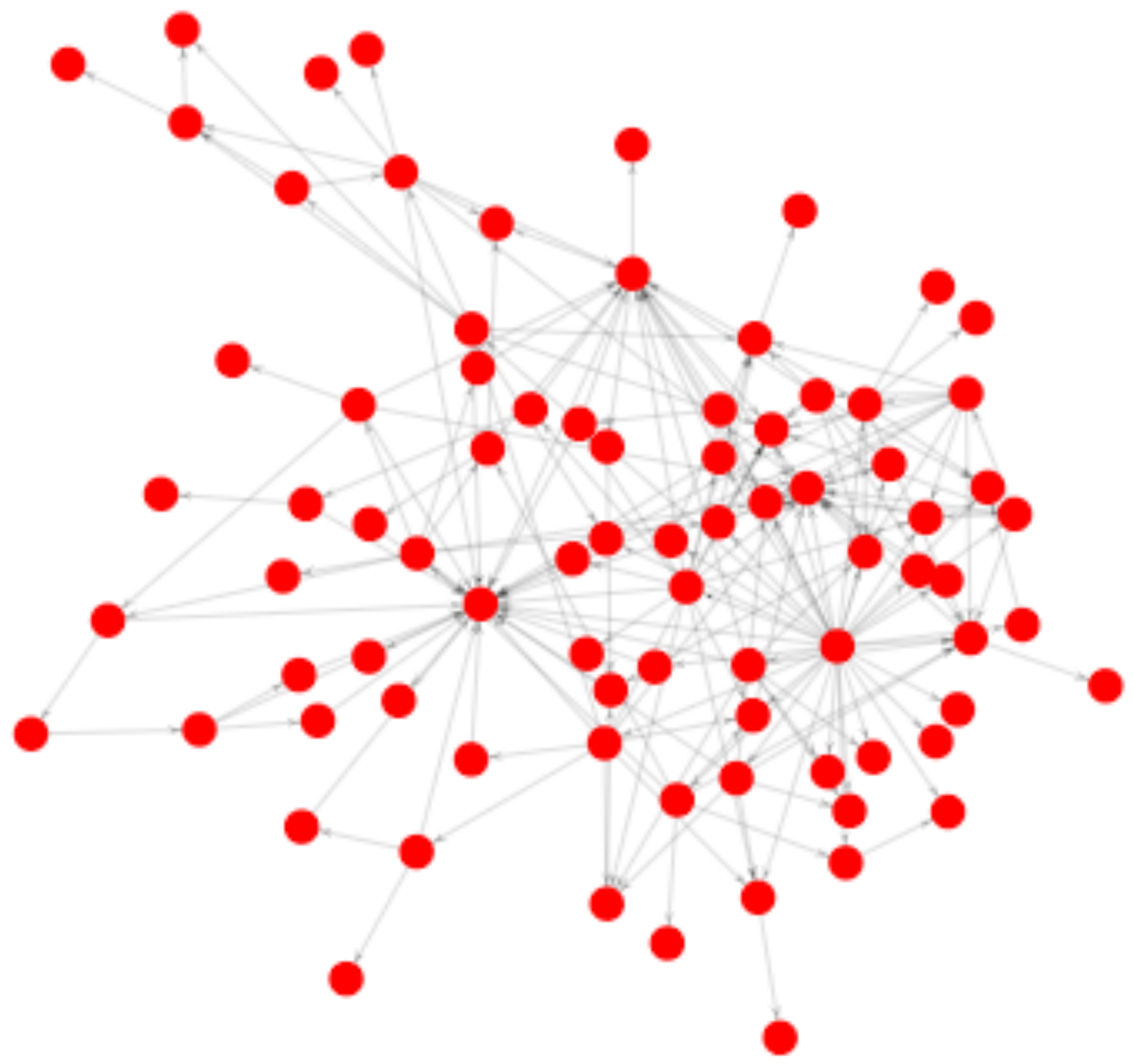
```
app.js
└─┬ some-package
  └─┬ entropoetry
    └── bn.js
```

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

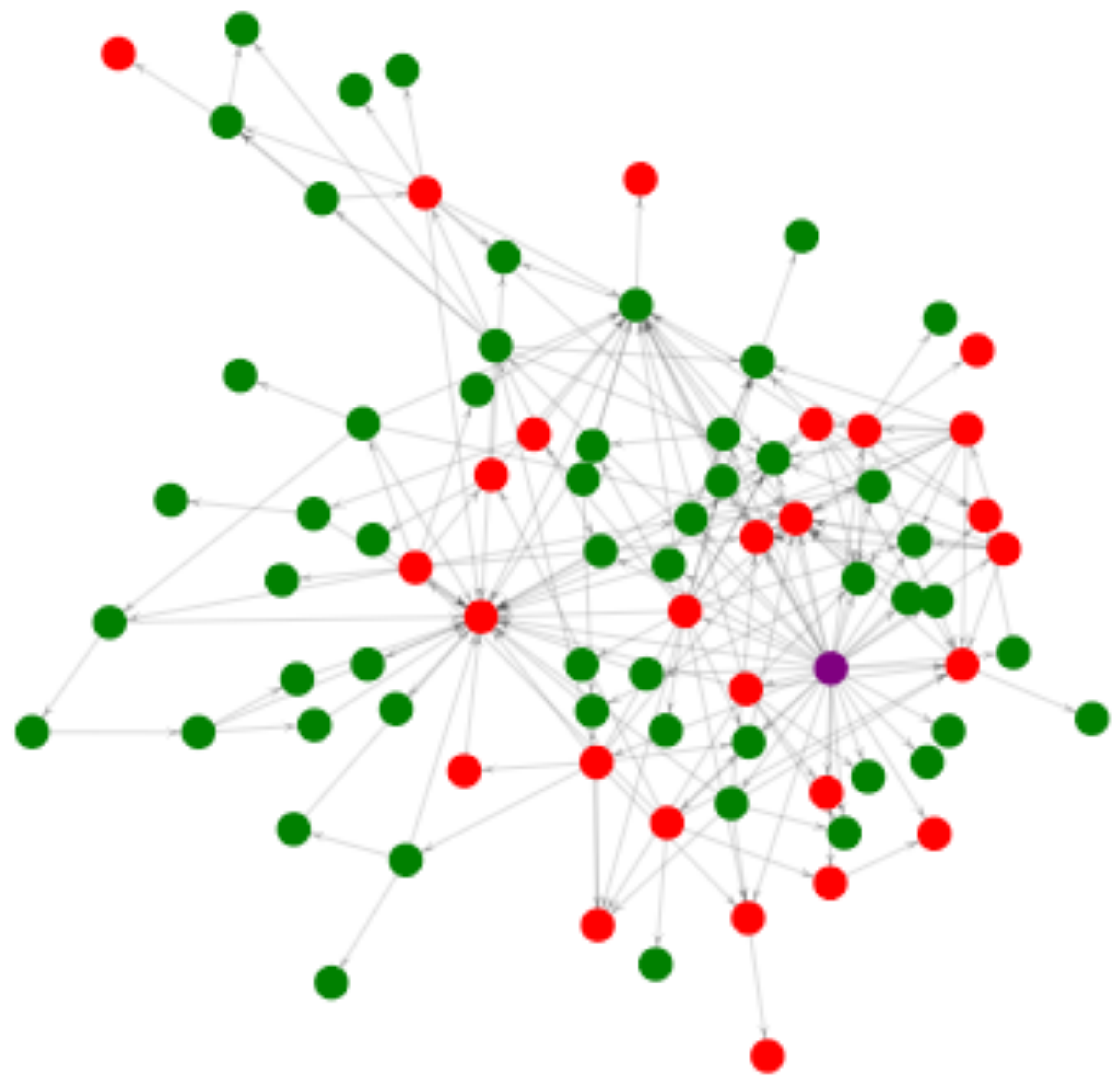
```
{
  "resources": {
    "some-package": {
      "globals": {
        "Buffer.from": true
      },
      "packages": {
        "some-package>entropoetry": true
      }
    },
    "some-package>entropoetry": {
      "builtin": {
        "assert": true,
        "buffer.Buffer": true,
        "zlib": true
      },
      "globals": {
        "console": true,
        "process.exitCode": "write"
      },
      "packages": {
        "some-package>entropoetry>bn.js": true
      }
    },
    "some-package>entropoetry>bn.js": {
      "builtin": {
        "buffer.Buffer": true
      },
      "globals": {
        "Buffer": true
      }
    }
  }
}
```

LavaMoat enables more focused security reviews

Exposure to package dependencies
without LavaMoat sandboxing



Exposure to package dependencies
with LavaMoat sandboxing



lavamoat-viz: https://github.com/LavaMoat/LavaMoat/tree/lavamoat-viz

Bonus: avoiding unwanted post-install scripts

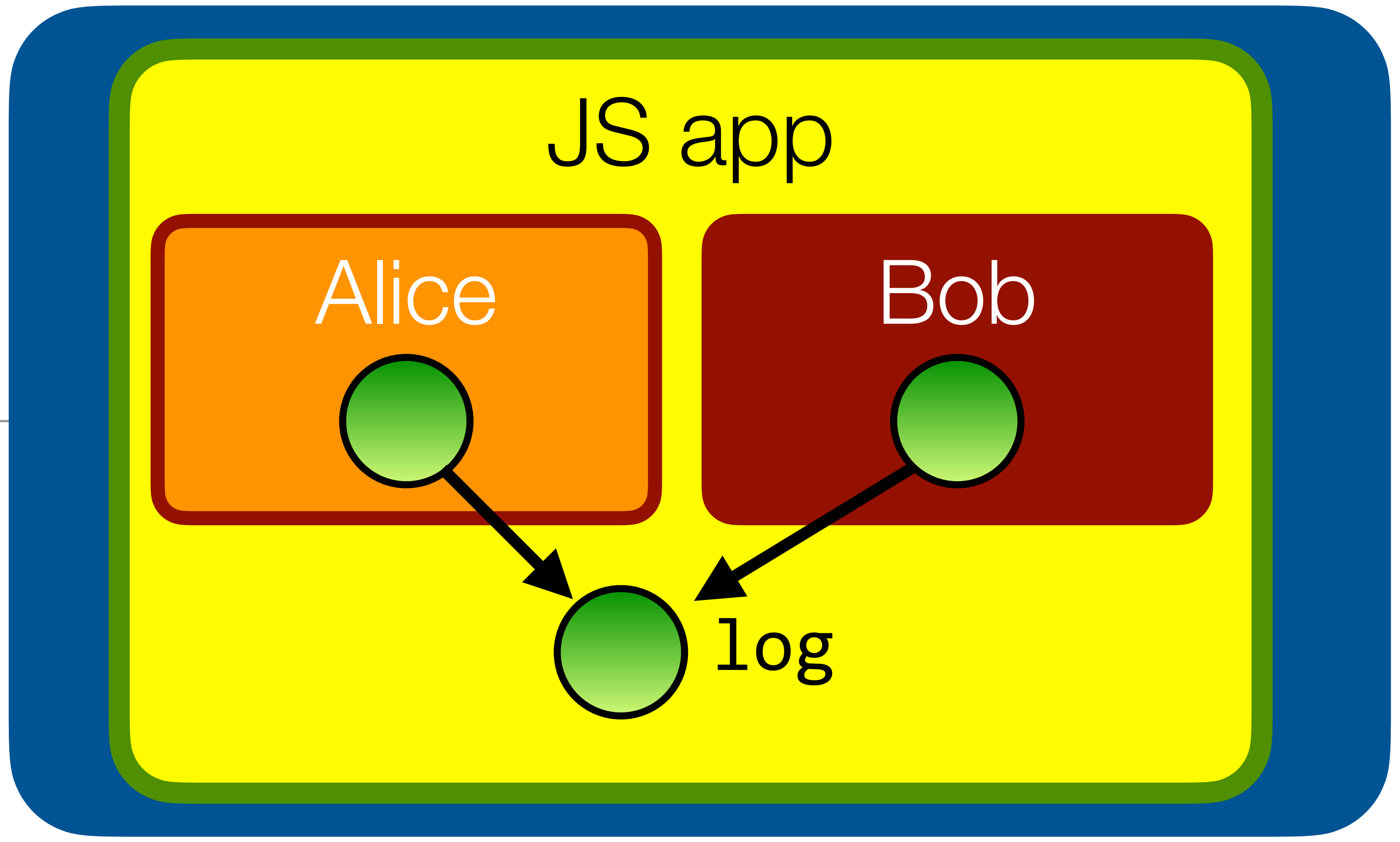
- Package managers like `npm` allow packages to run install scripts
- A compromised dependency can exploit this to run code as part of your project installation script
- Lavamoat's `allow-scripts` tool configures your project to disable running install scripts by default
- Edit allowed packages in `package.json`
- **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 prototype 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.setup(log);
bob.setup(log);
```

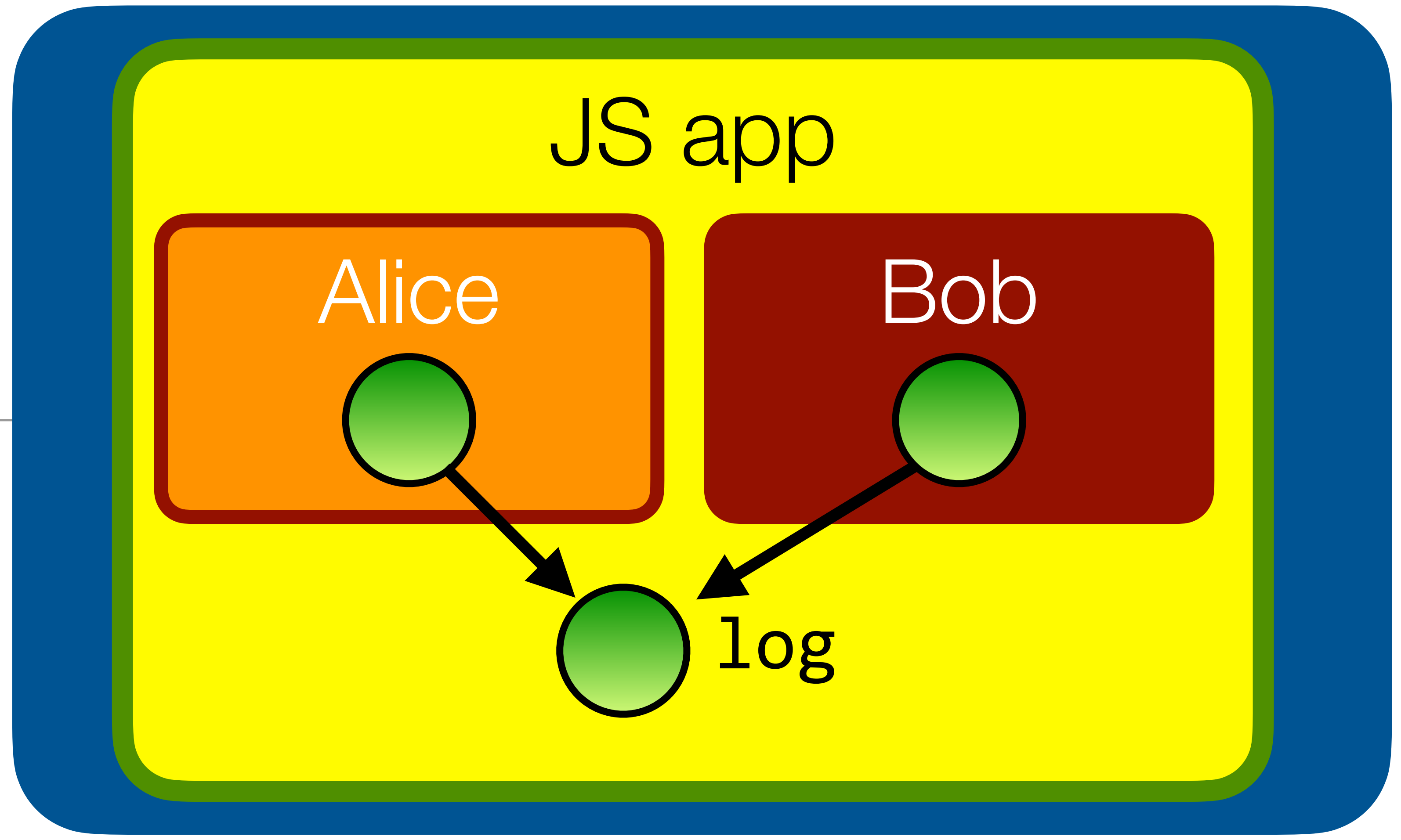
```
// in bob.js
// Bob can just write to the log (excess authority)
log.write("I'm polluting the log")

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

// Bob can replace the 'write' function (api poisoning)
log.write = function(msg) {
  console.log("I'm not logging anything");
}
```

~~// Bob can replace the built-ins (prototype poisoning)~~
~~Array.prototype.push = function(msg) {~~
~~console.log("I'm not logging anything");~~
~~}~~

KU LEUVEN DistriNet

One down, three to go

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

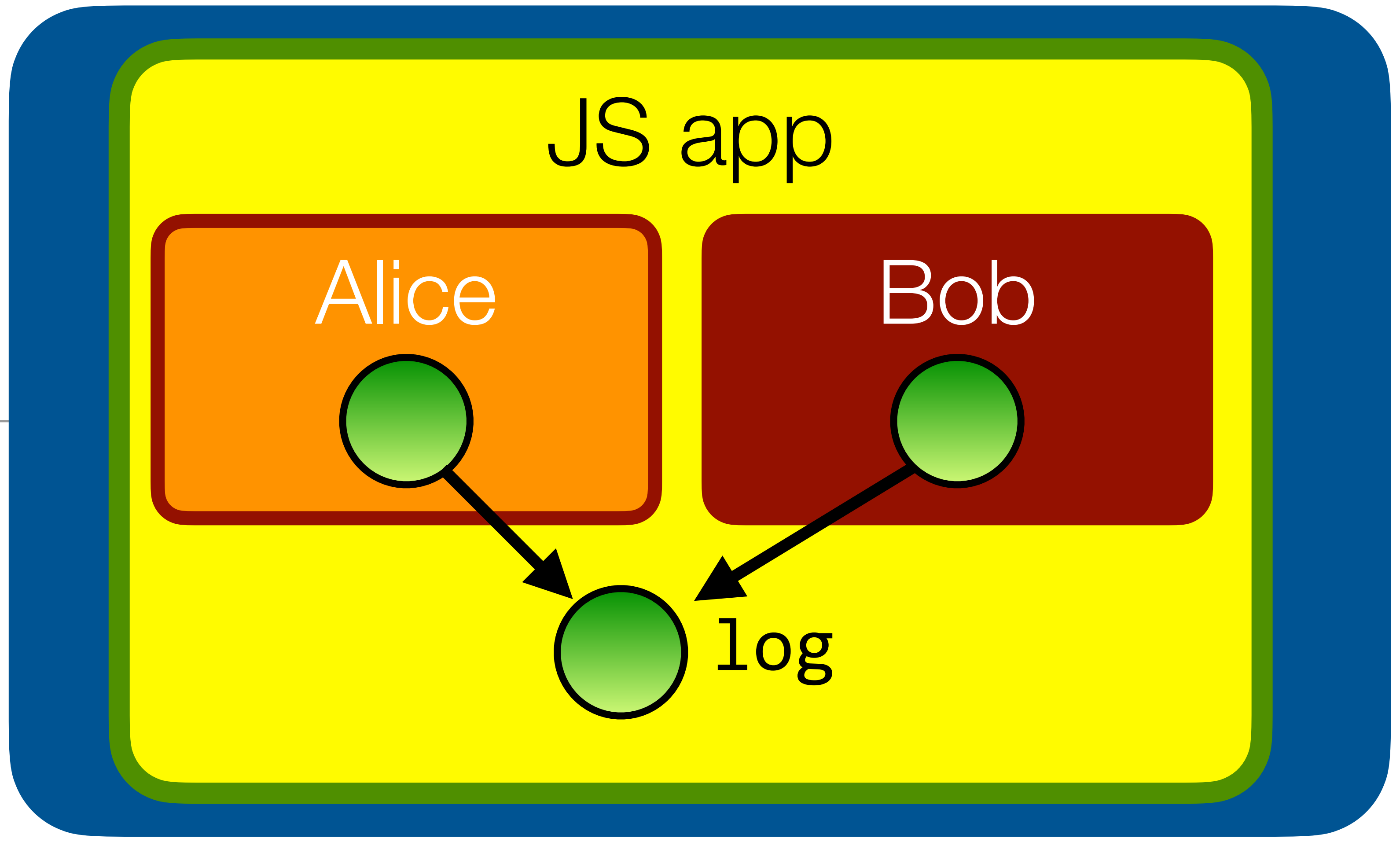



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

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

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

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



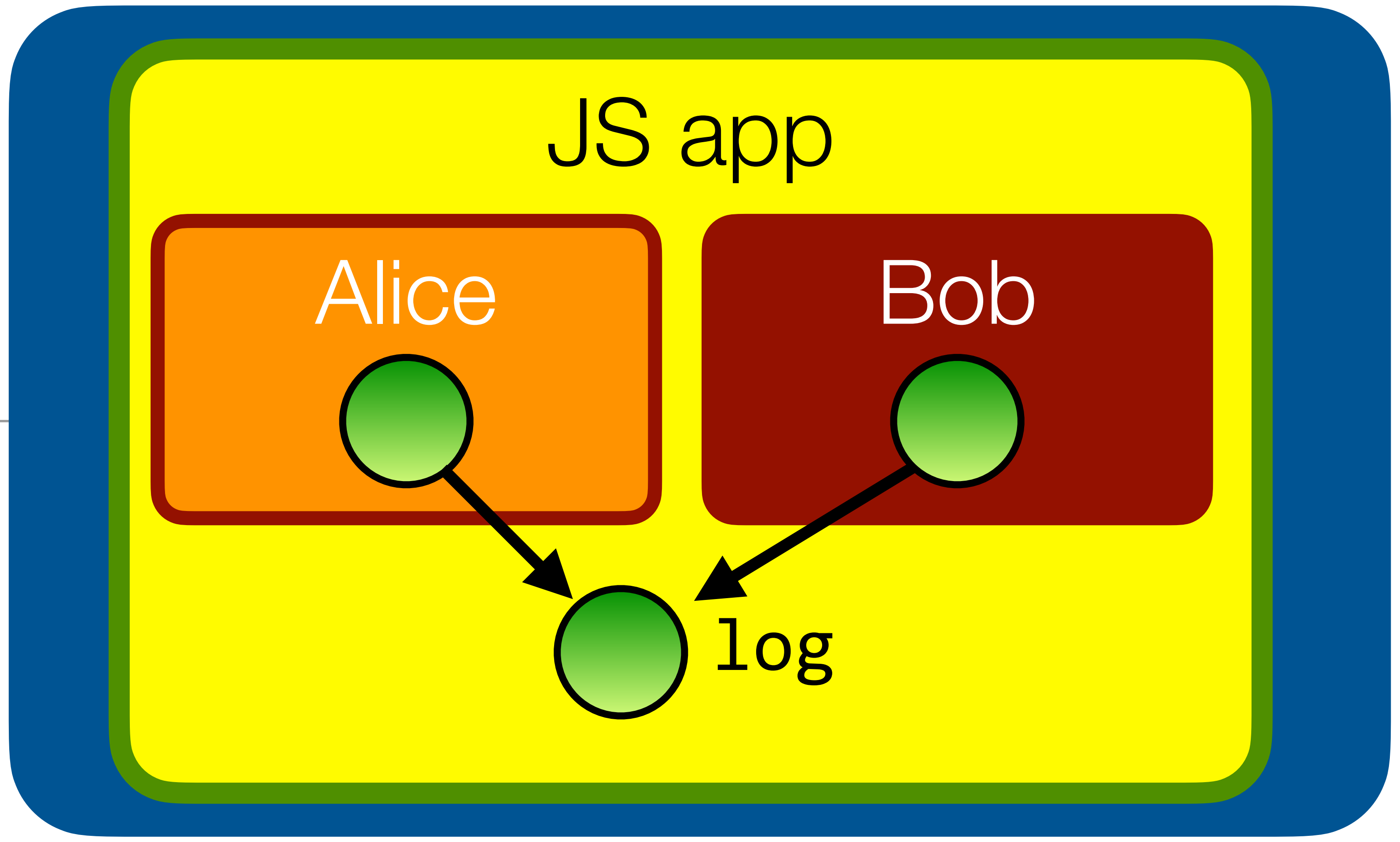
Object.freeze makes property bindings (not their values) immutable

```
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.setup(log);
bob.setup(log);
```

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

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

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


KU LEUVEN DistriNet

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



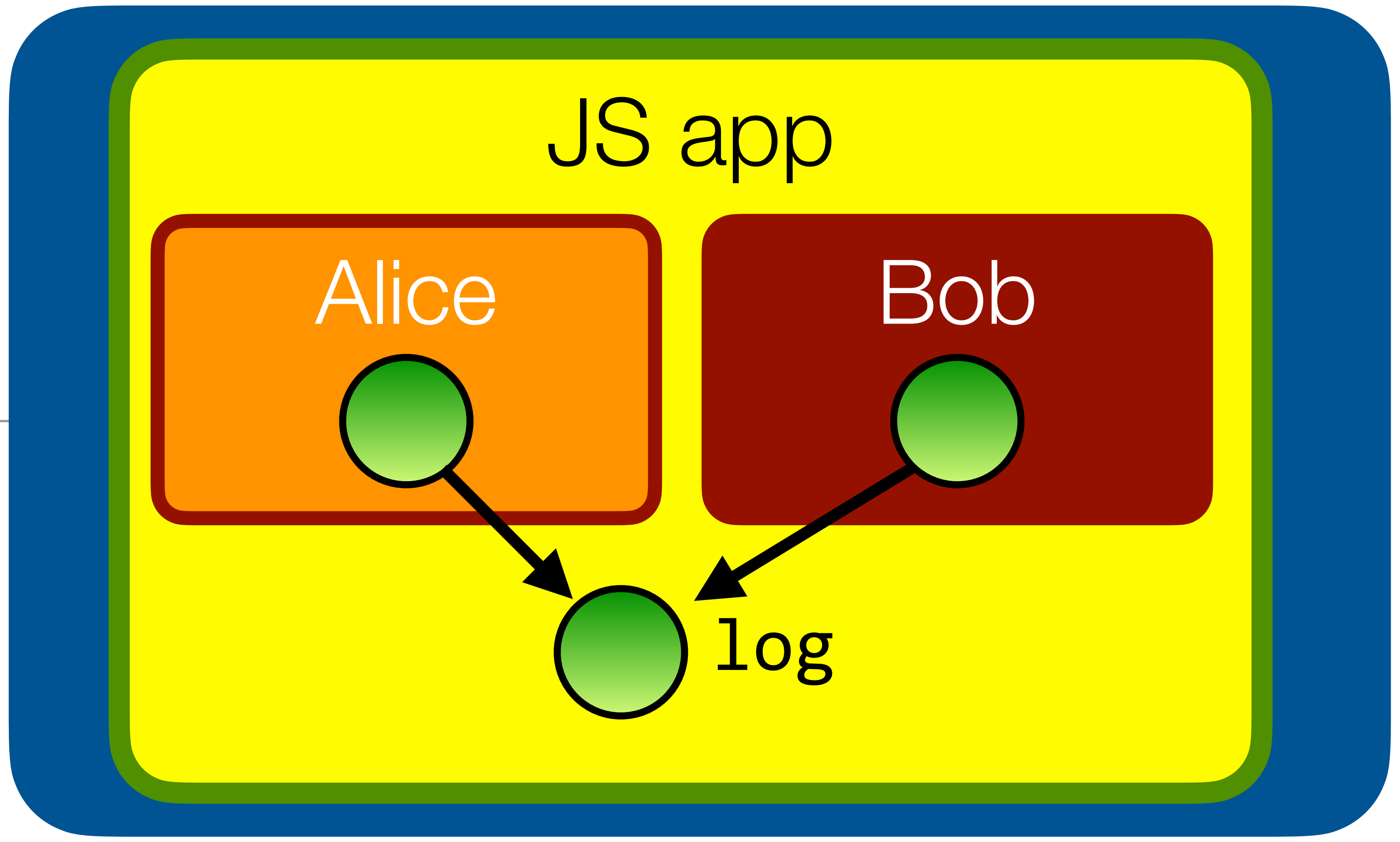
Functions are mutable too. Freeze doesn’t recursively freeze the object’s functions.

```
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.setup(log);
bob.setup(log);
```

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

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

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

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


KU LEUVEN DistriNet

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.setup(log);
bob.setup(log);
```

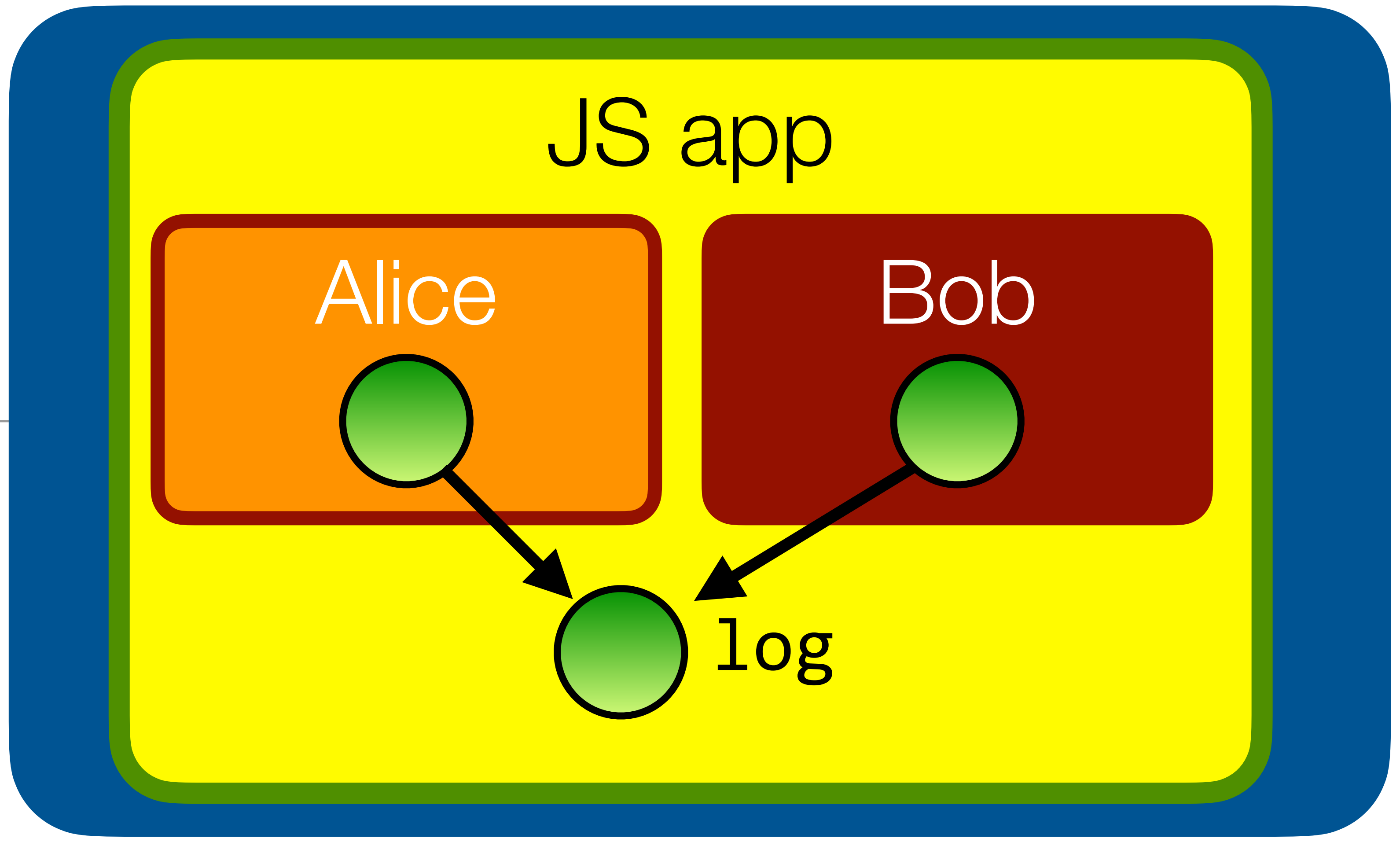
```
// in bob.js
// Bob can just write to the log (excess authority)
log.write("I'm polluting the log")

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

~~// Bob can replace the 'write' function (api poisoning)~~
~~log.write = function(msg) {~~
~~console.log("I'm not logging anything");~~
~~}~~

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


KU LEUVEN DistriNet

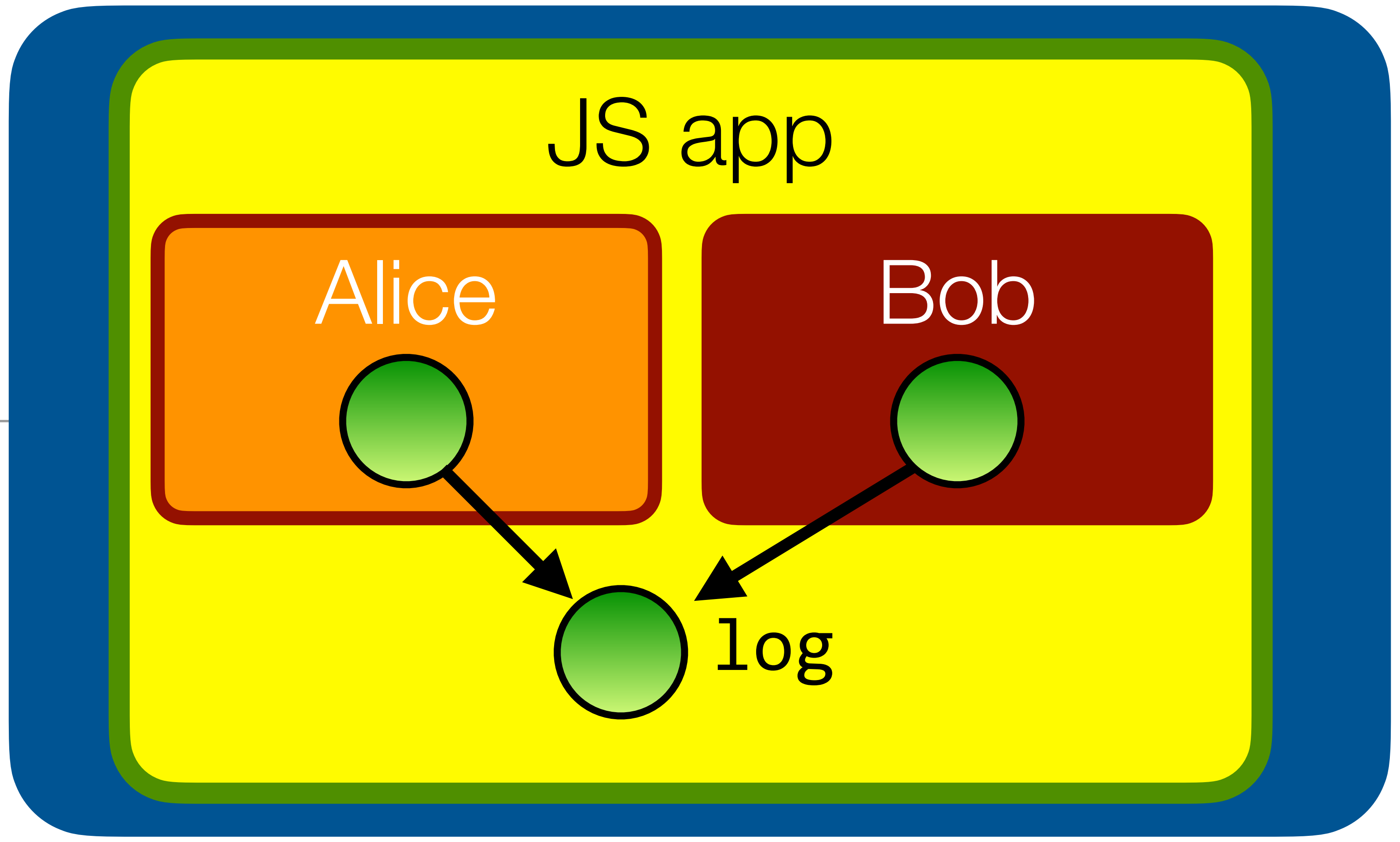
Two down, two to go




```
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.setup(log);
bob.setup(log);
```

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

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

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

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

Two down, two to go

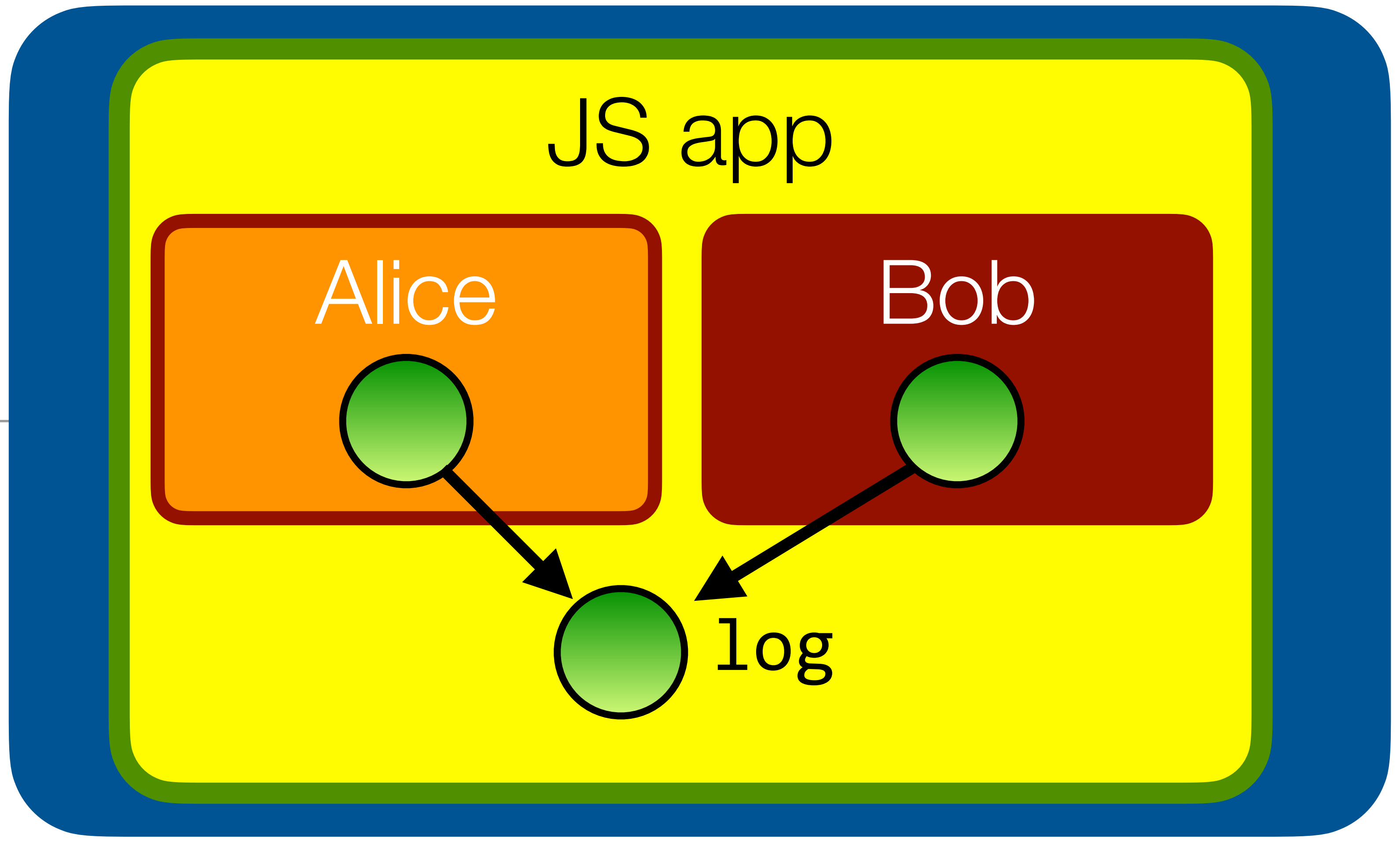



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

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

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)

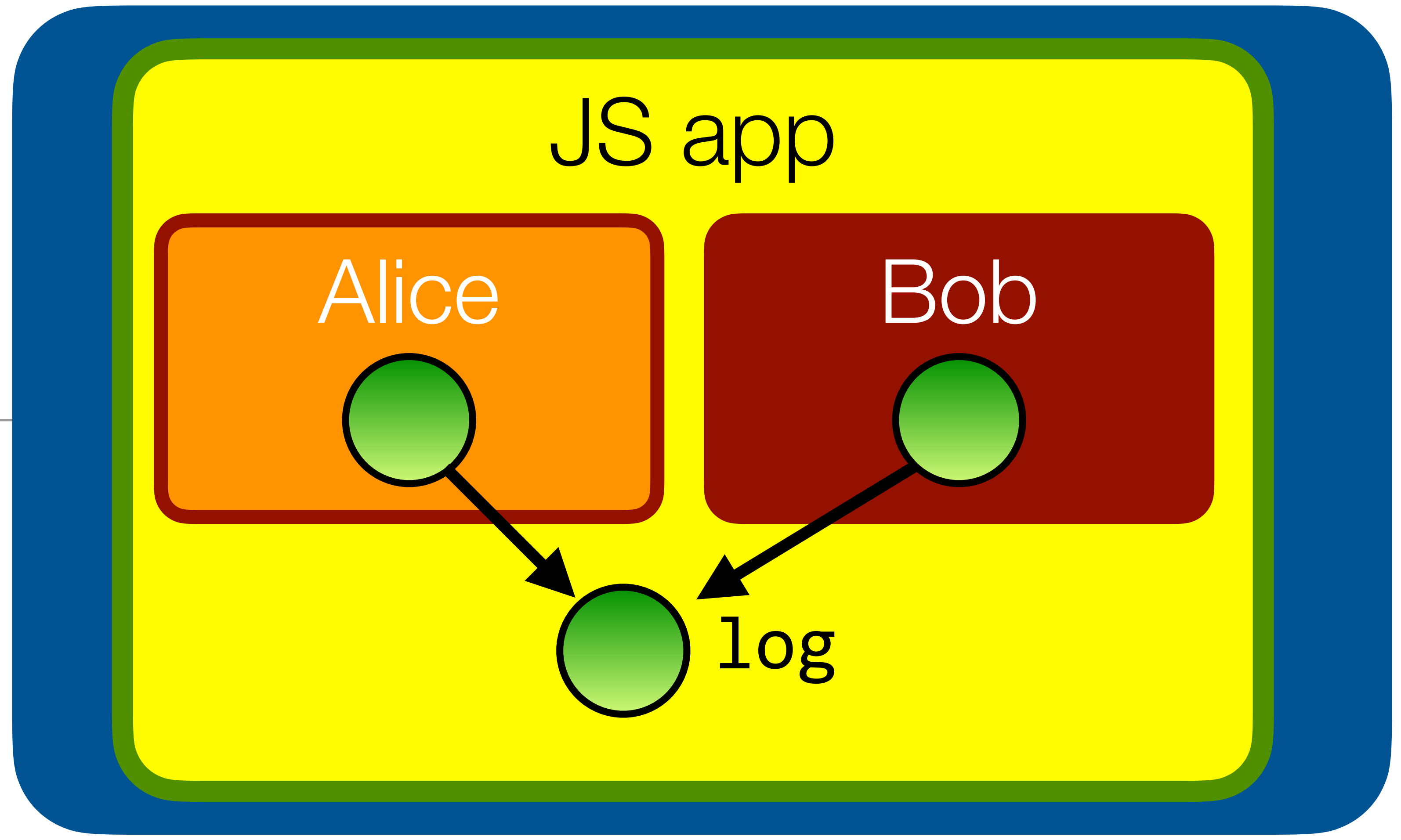



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

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


KU LEUVEN DistriNet

Three down, one to go

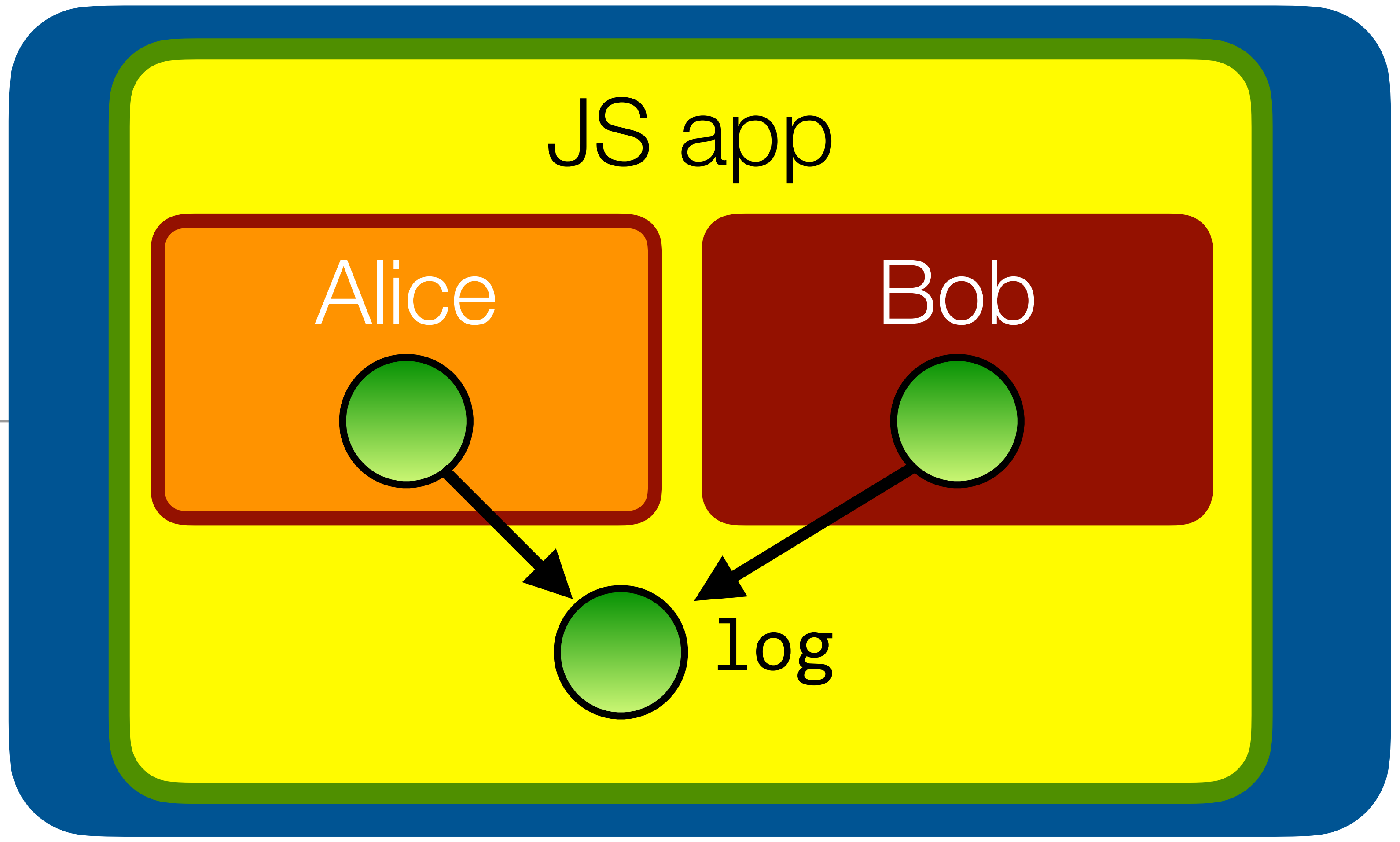



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

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


KU LEUVEN DistriNet

Three down, one to go

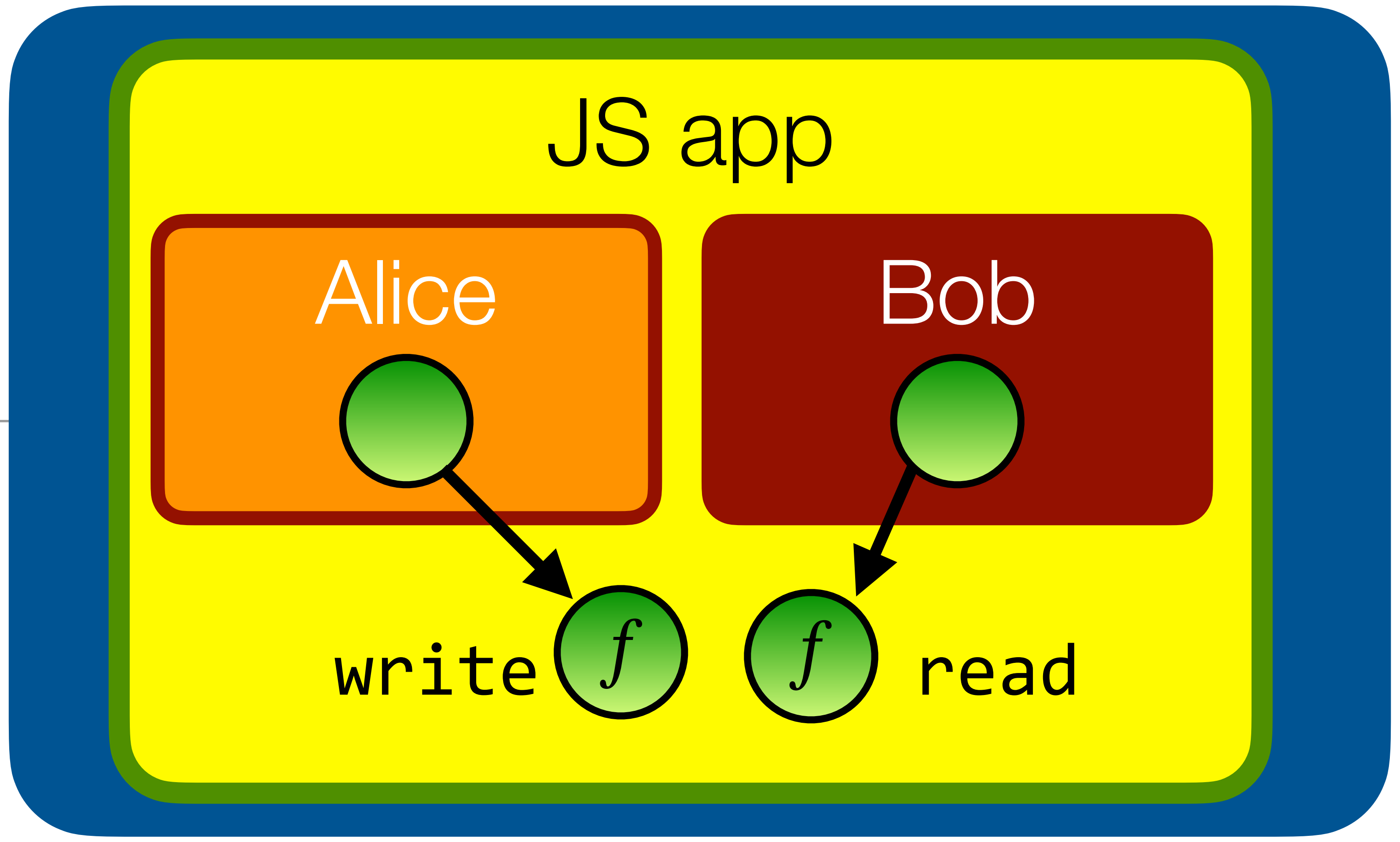


- Recall: we would like Alice to only write to the log, and Bob to only read from the log.
- Bob receives too much authority. How to limit?

```
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.setup(log);
bob.setup(log);
```

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

Pass only the authority that Bob needs.

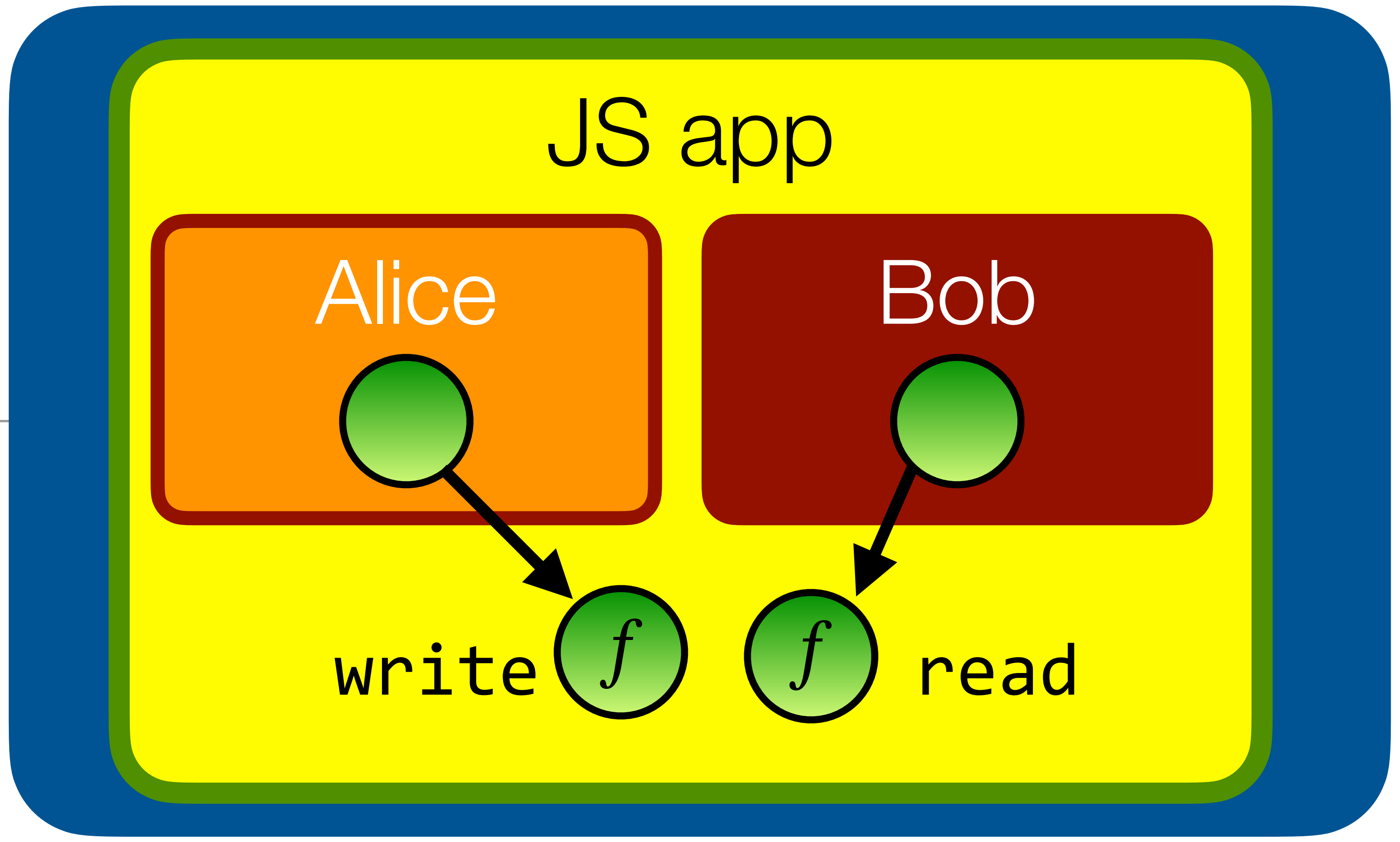


Just pass the write function to Alice and the read function to Bob.

```
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.setup(write);
bob.setup(read);
```

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

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



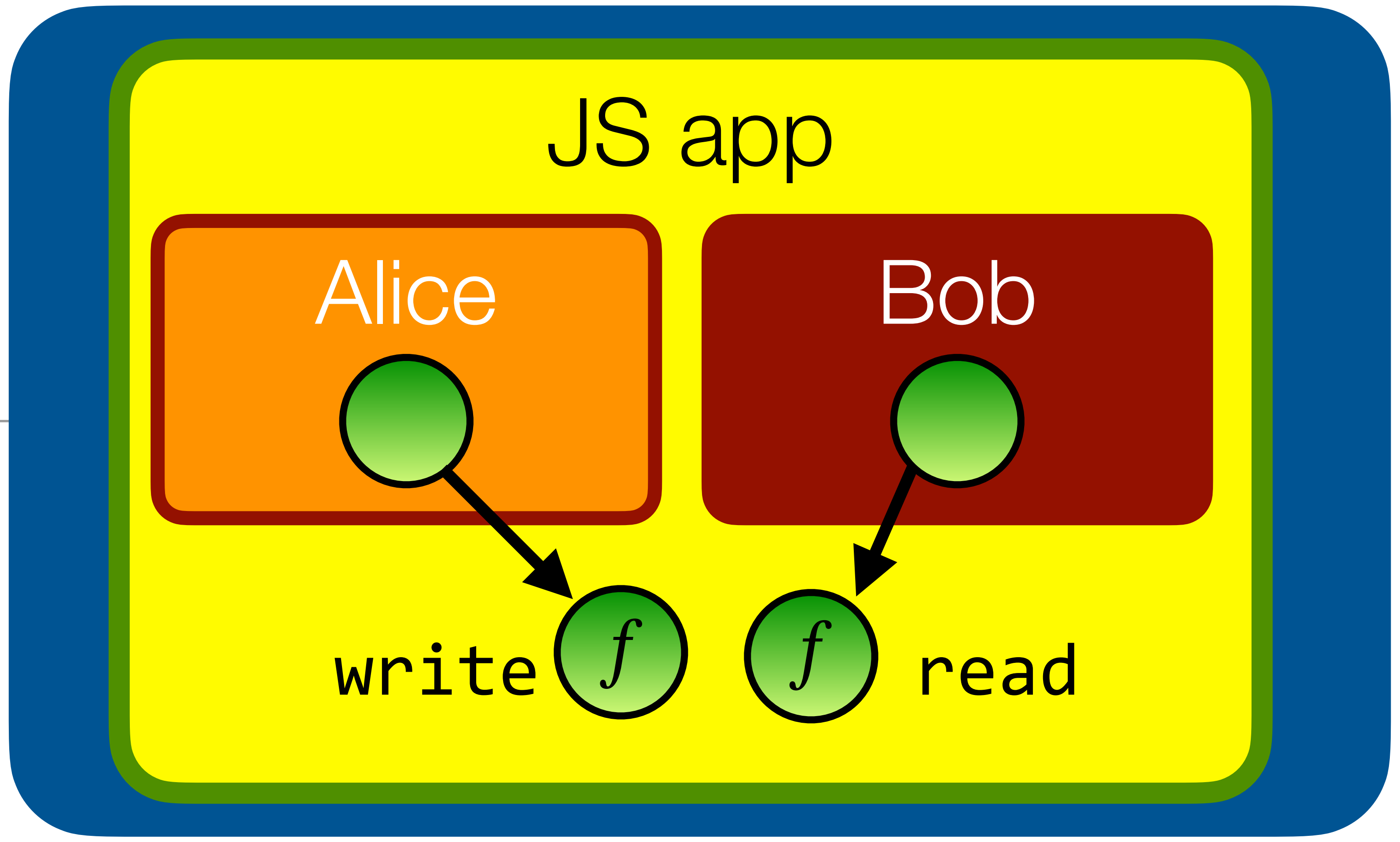

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.setup(write);
bob.setup(read);
```

```
// in bob.js
```

~~// Bob can just write to the log (excess authority)~~
~~log.write(“I’m polluting the log”)~~

KU LEUVEN DistriNet

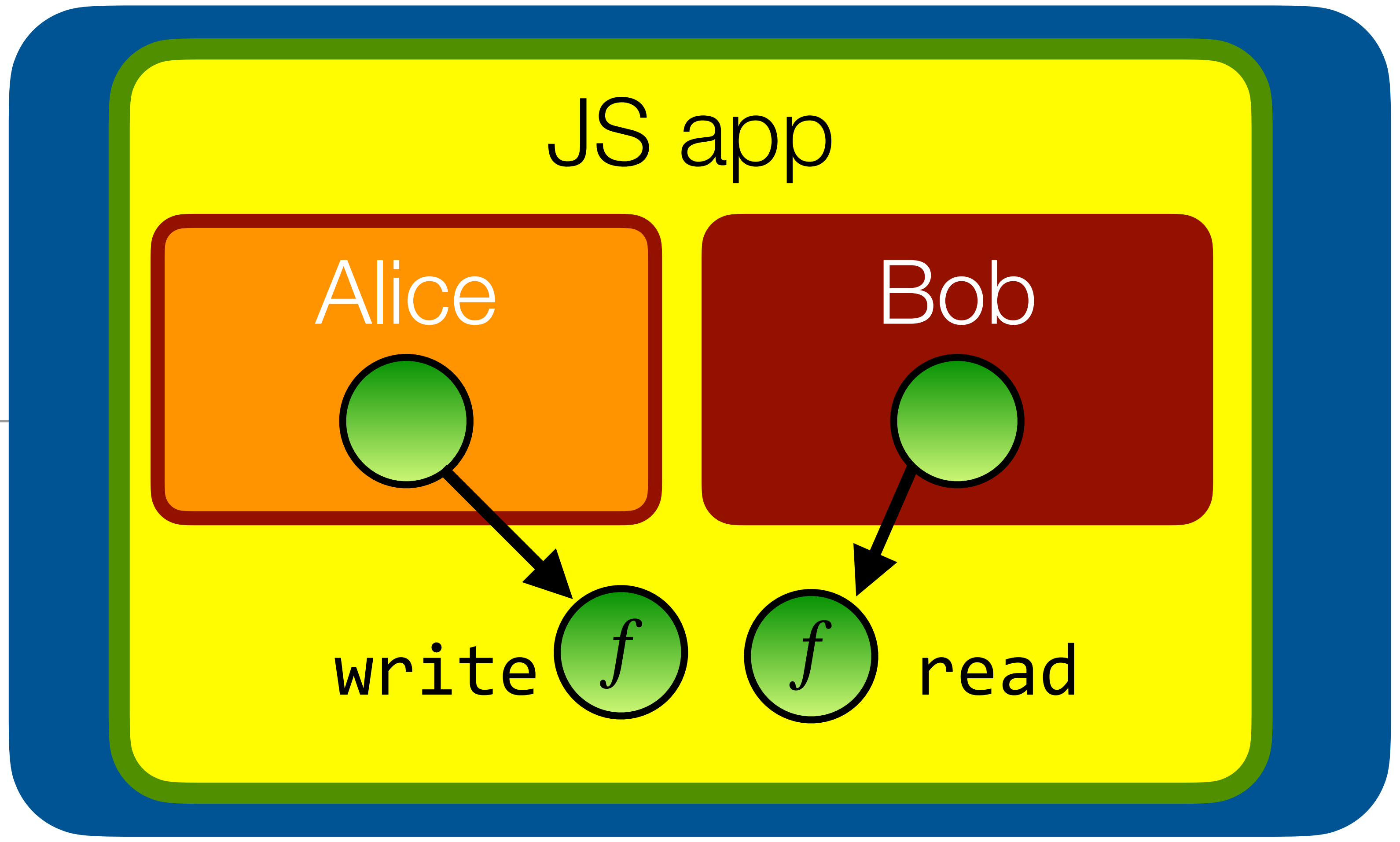
Is there a better way to write this code?



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.setup(write);
bob.setup(read);
```

Use the **Function as Object** pattern



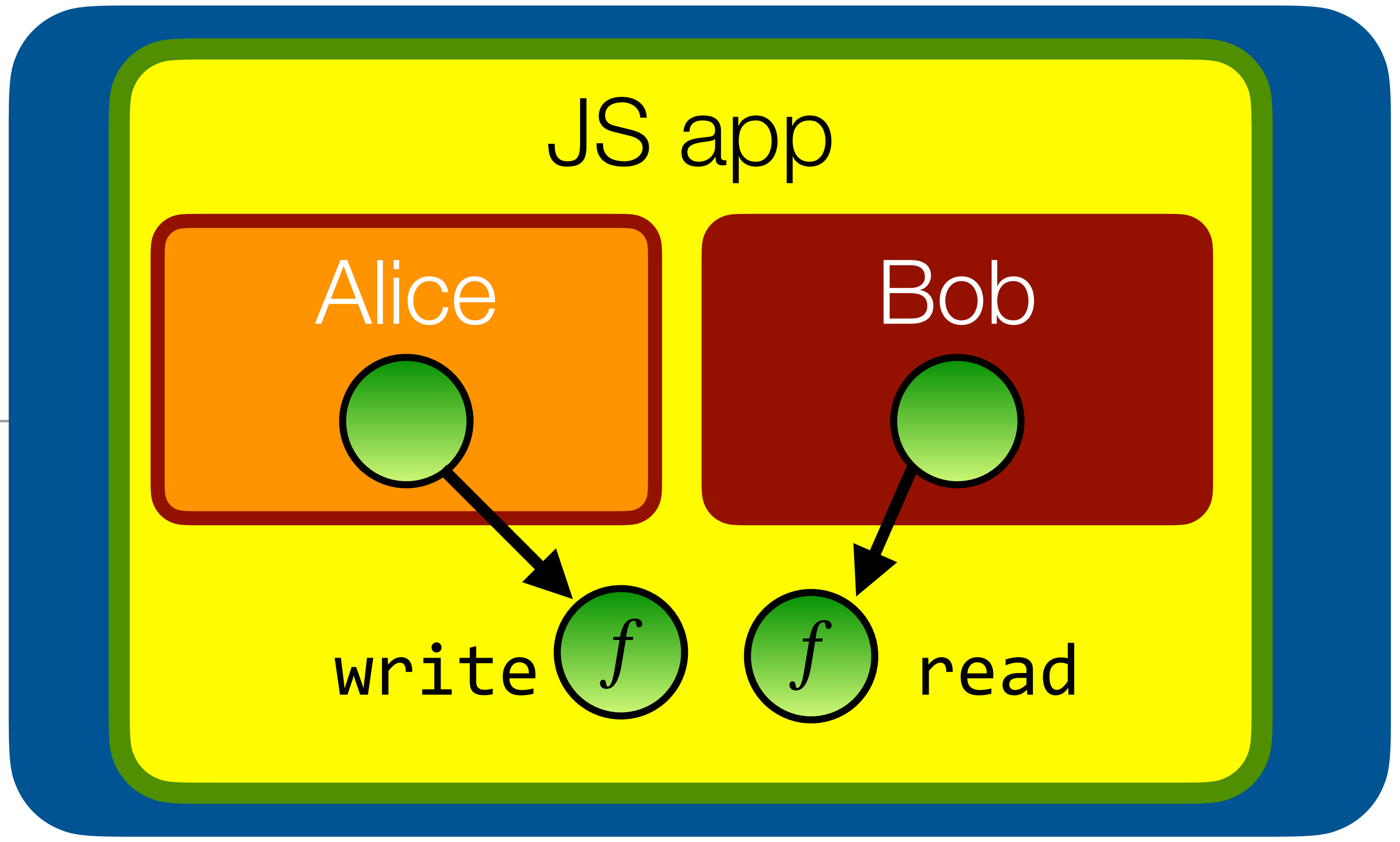
- A record of closures hiding state is a fine representation of an object of methods hiding instance vars
- Pattern long advocated by Doug Crockford instead of using classes or prototypes

```
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.setup(write);
bob.setup(read);
```

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


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

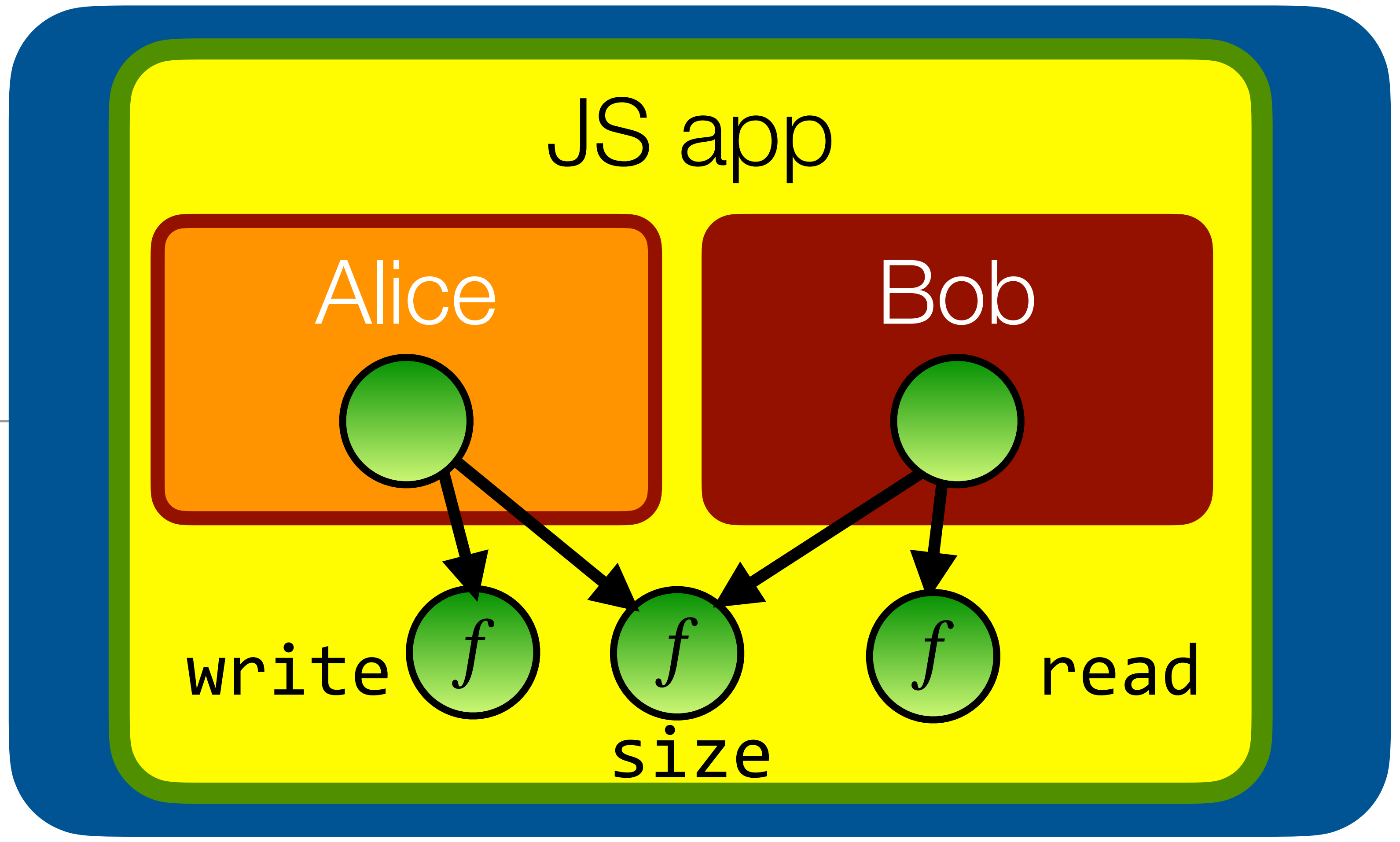
(See also https://martinfowler.com/bliki/FunctionAsObject.html)

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

What if Alice and Bob need more authority?



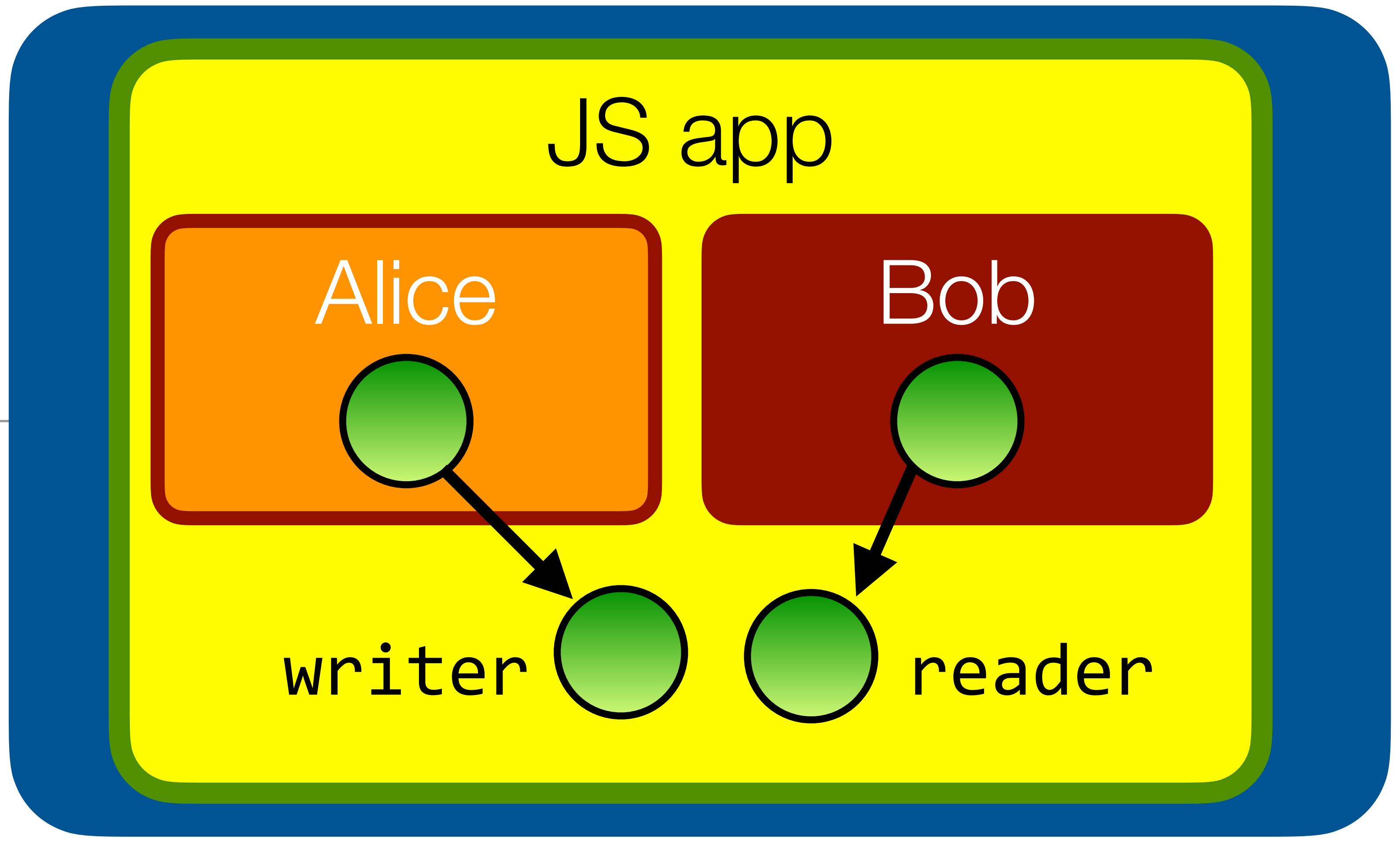
If over time we want to expose more functionality to Alice and Bob, we need to refactor all of our code.

```
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.setup(log.write);
bob.setup(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.setup(log.write, log.size);
bob.setup(log.read, log.size);
```


KU LEUVEN DistriNet

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.setup(log.write, log.size);
bob.setup(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.setup(log.writer);
bob.setup(log.reader);
```

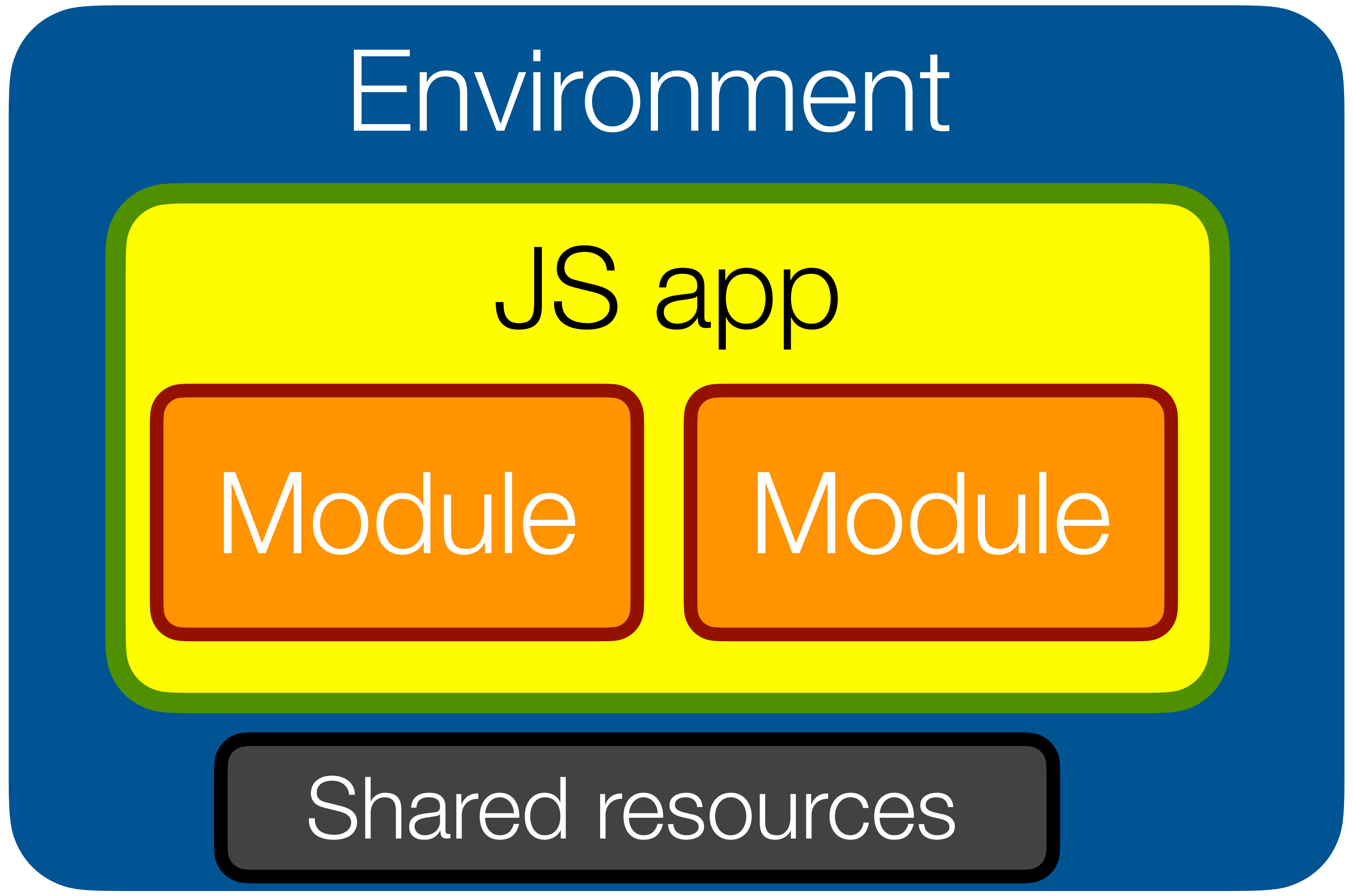

KU LEUVEN DistriNet

Demo

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

End of Part II: recap

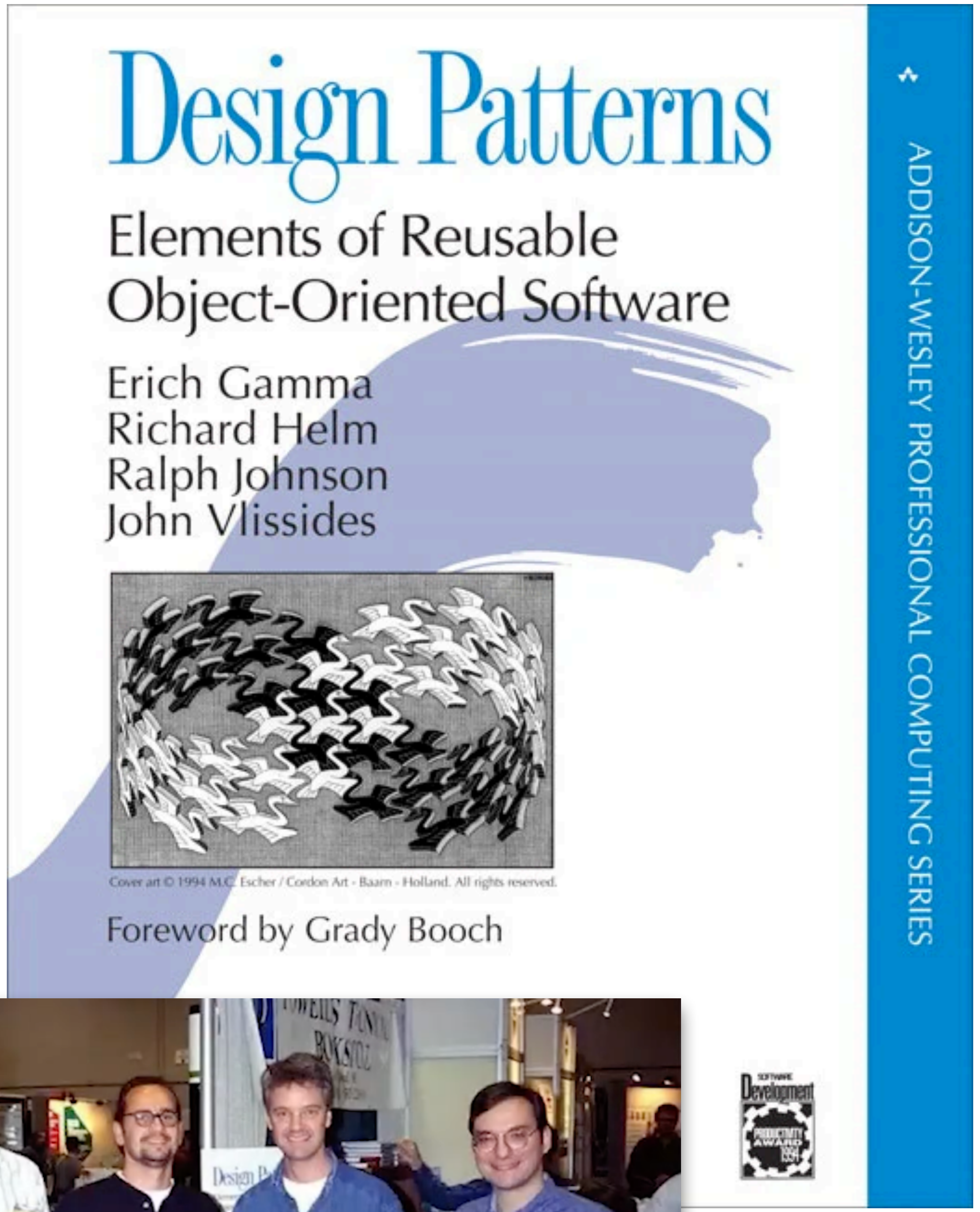
- Modern JS apps are composed from many modules. You can't trust them all.
- Traditional security boundaries don't exist between modules. Compartments add basic isolation.
- **Isolated modules must still interact!**
- **Compose** functionality from untrusted modules in a **least-authority** manner
-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



Part III
Safely composing modules using least-authority patterns

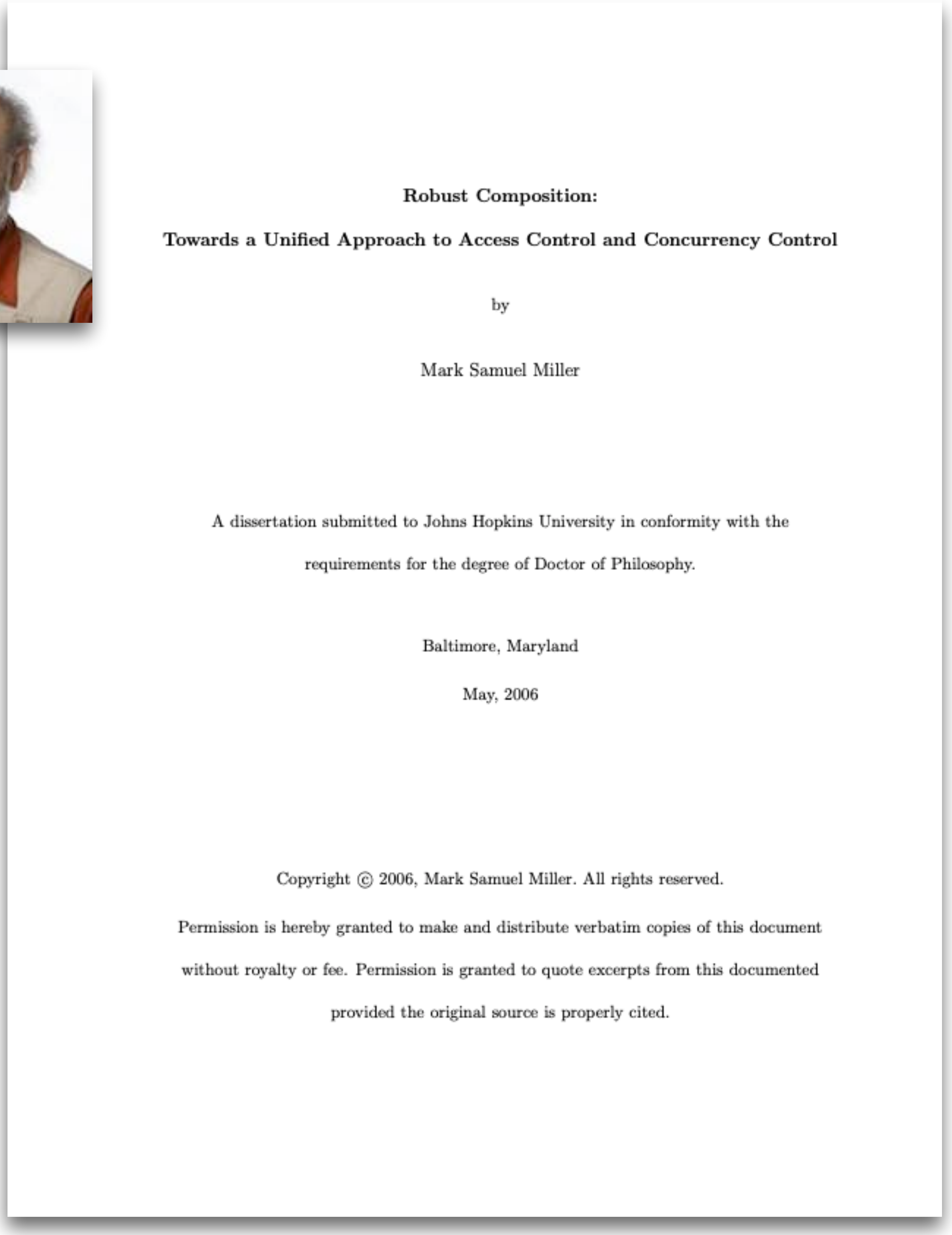
KU LEUVEN DistriNet

Design Patterns (“Gang of Four”, 1994)



- 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.

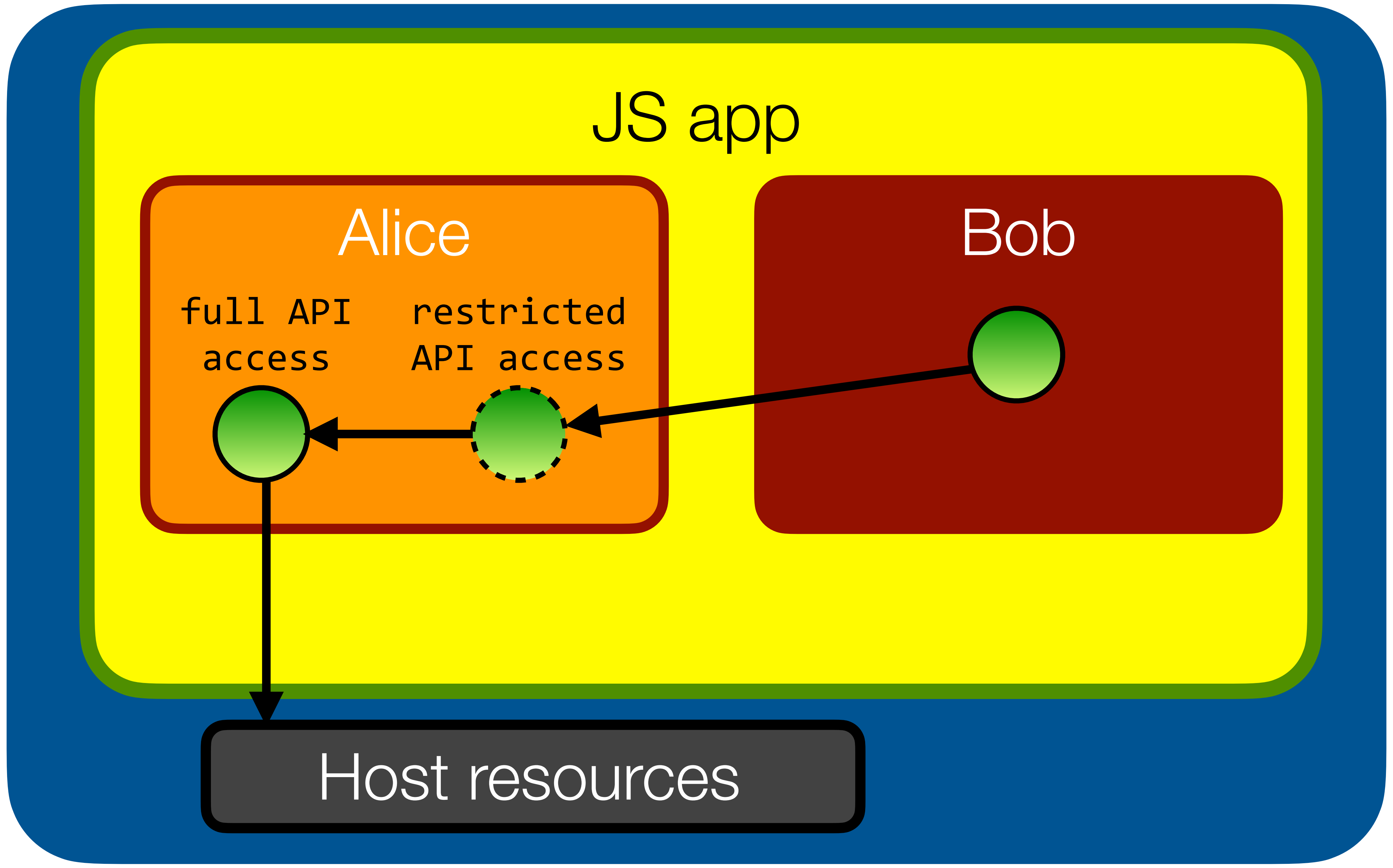
Permission is hereby granted to make and distribute verbatim copies of this document without royalty or fee. Permission is granted to quote excerpts from this documented provided the original source is properly cited.

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

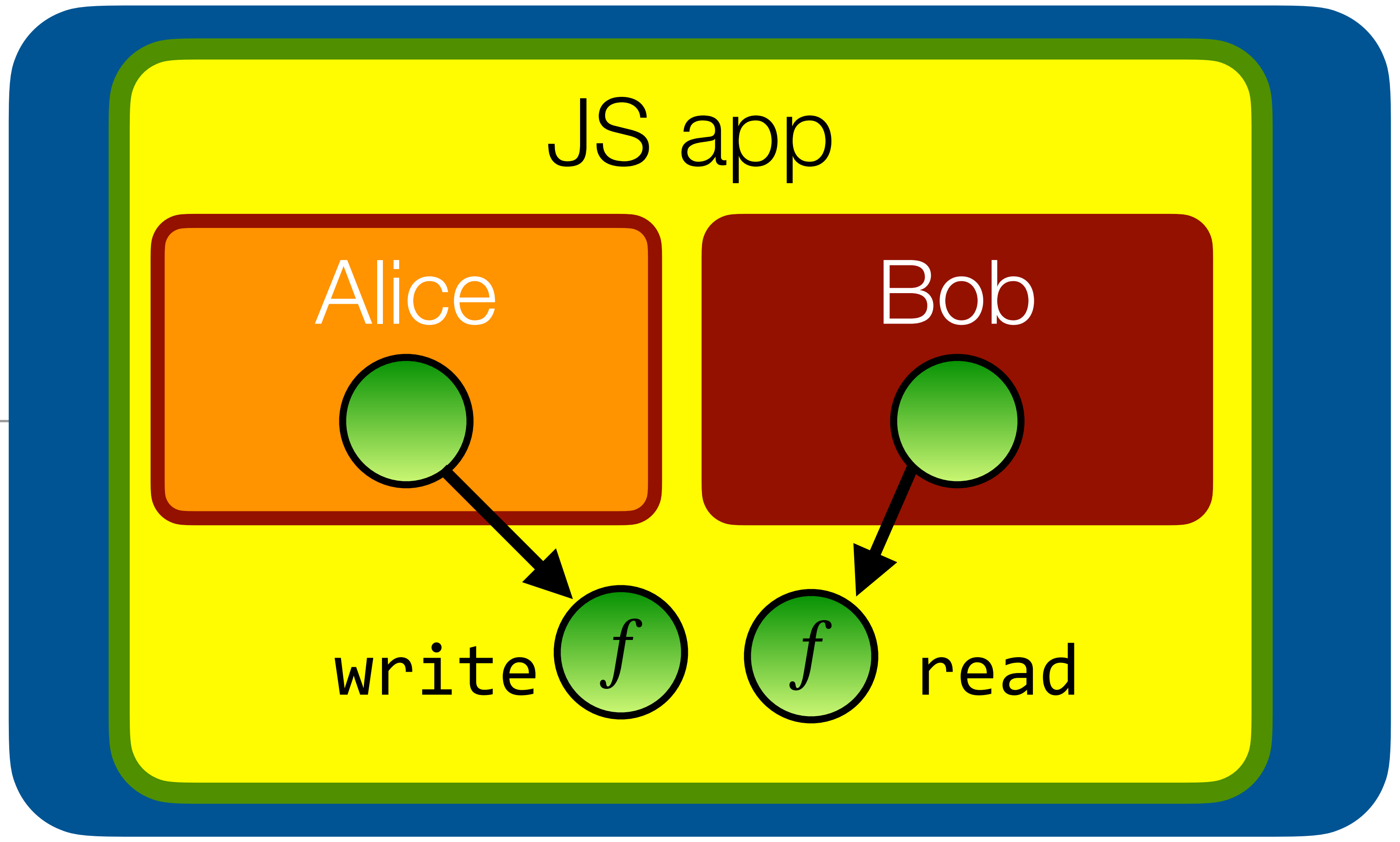
- Facets
- Taming
- Caretaker
- Membrane
- Sealer/unsealer pair
- …

Recall: the Principle of Least Authority (POLA)

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



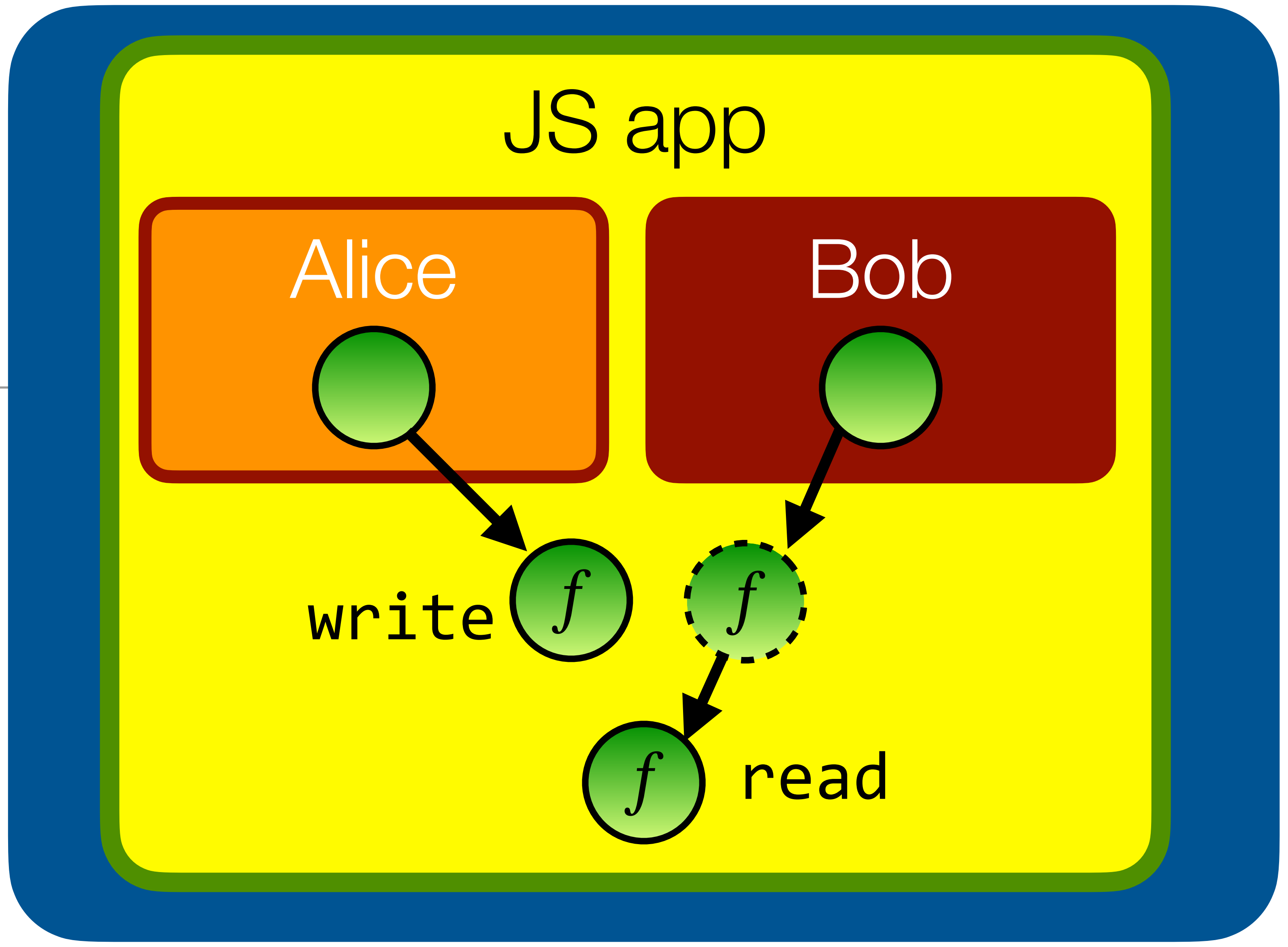
Further limiting Bob's authority



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

Use **caretaker** to insert access control logic



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

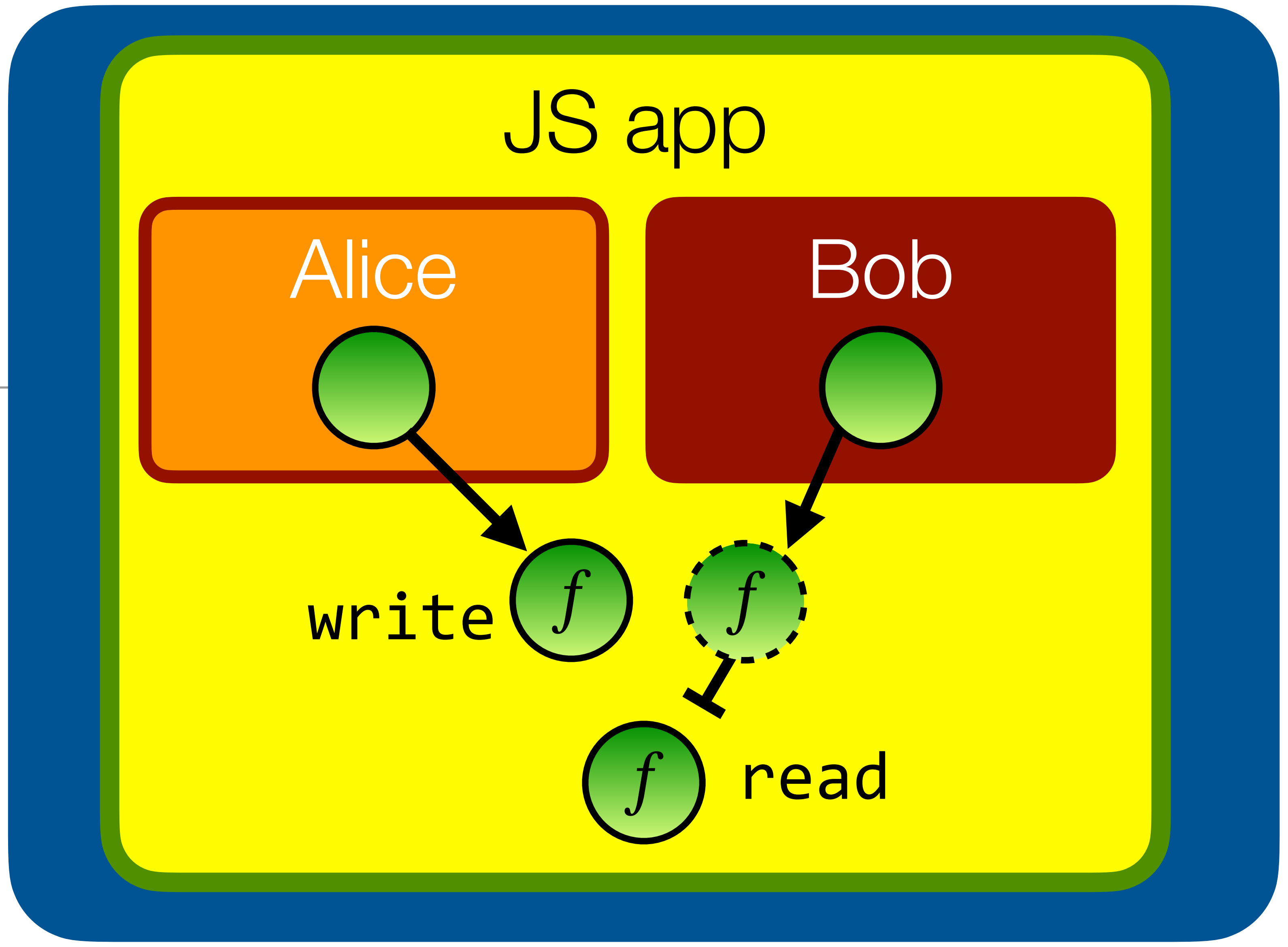
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.setup(log.write);
bob.setup(rlog.read);
```


KU LEUVEN DistriNet

Use **caretaker** to insert access control logic

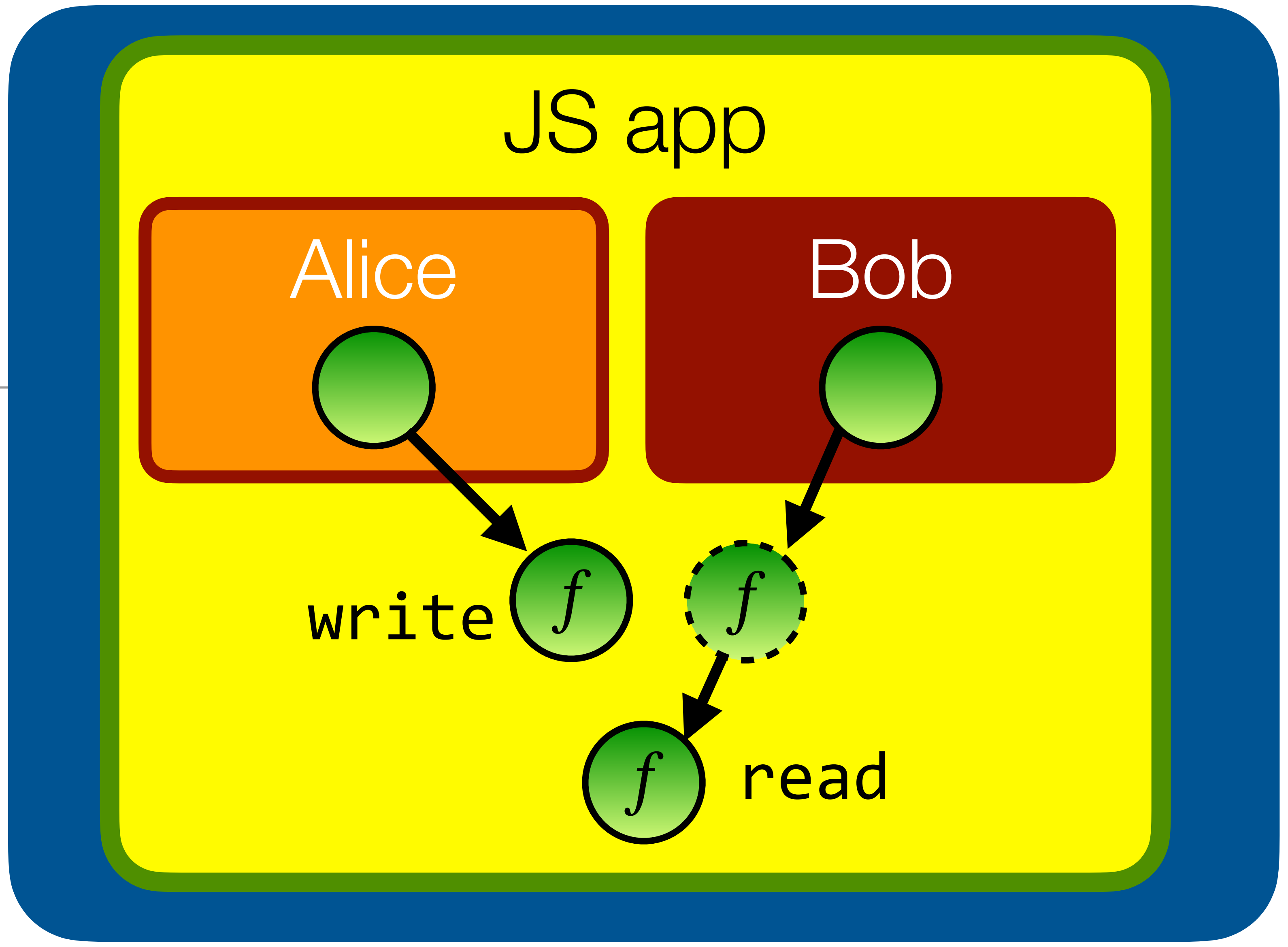
We would like to give Bob only **temporary** read access to the log.

```
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.setup(log.write);
bob.setup(rlog.read);

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



KU LEUVEN DistriNet

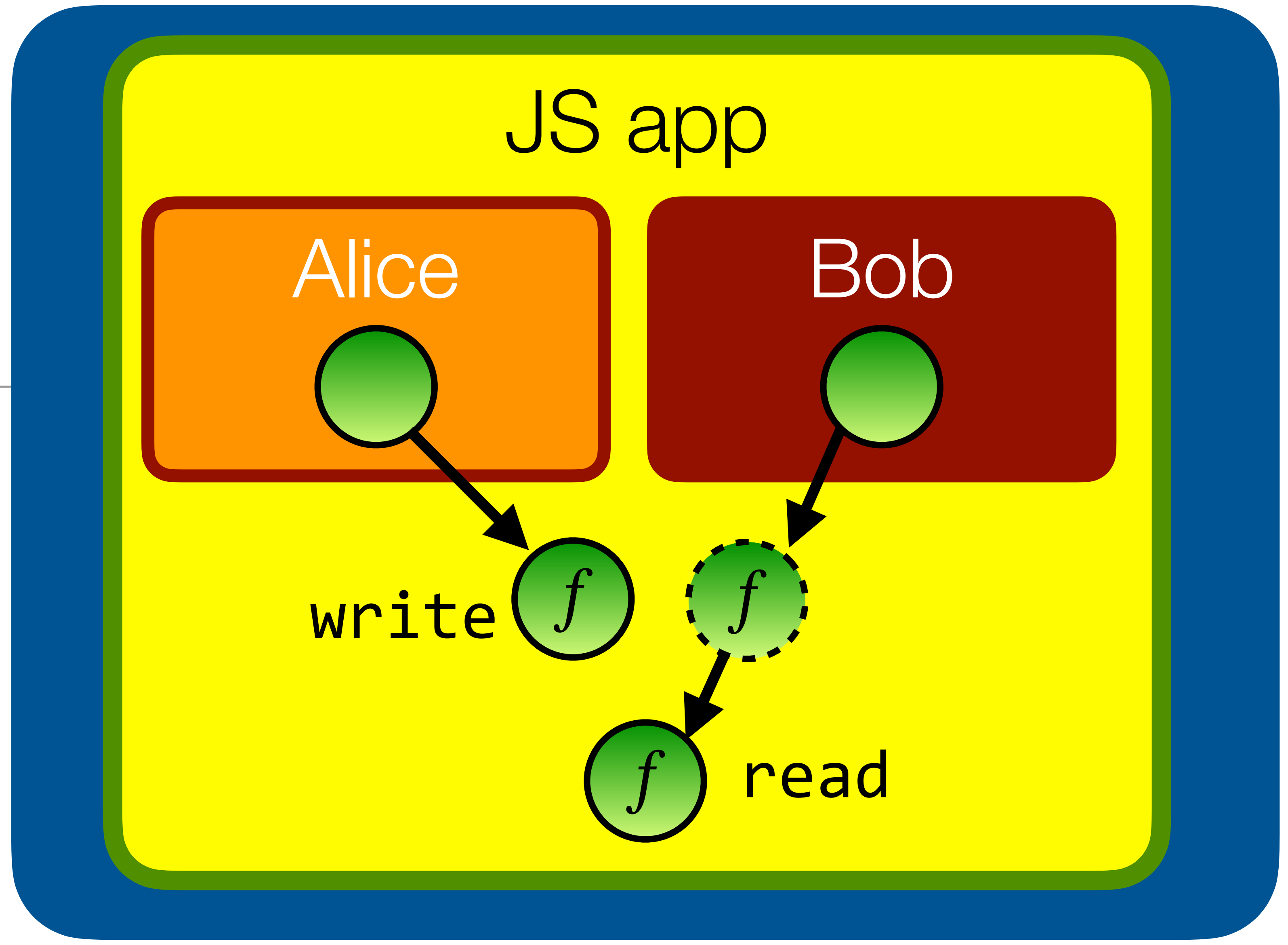
Use **caretaker** to insert access control logic




```
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.setup(log.write);
bob.setup(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]);
}
```

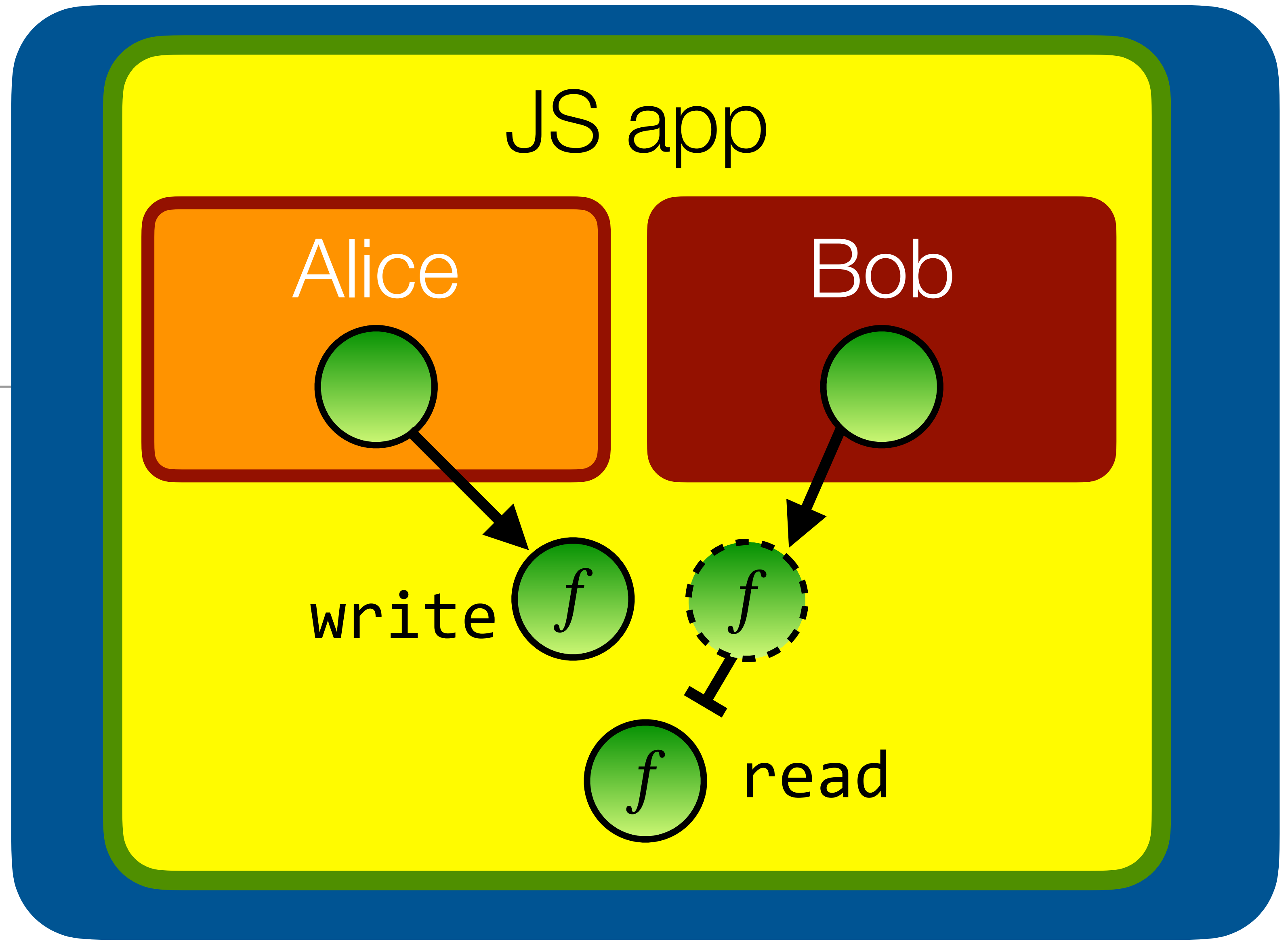
A caretaker is just a proxy object




```
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.setup(log.write);
bob.setup(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



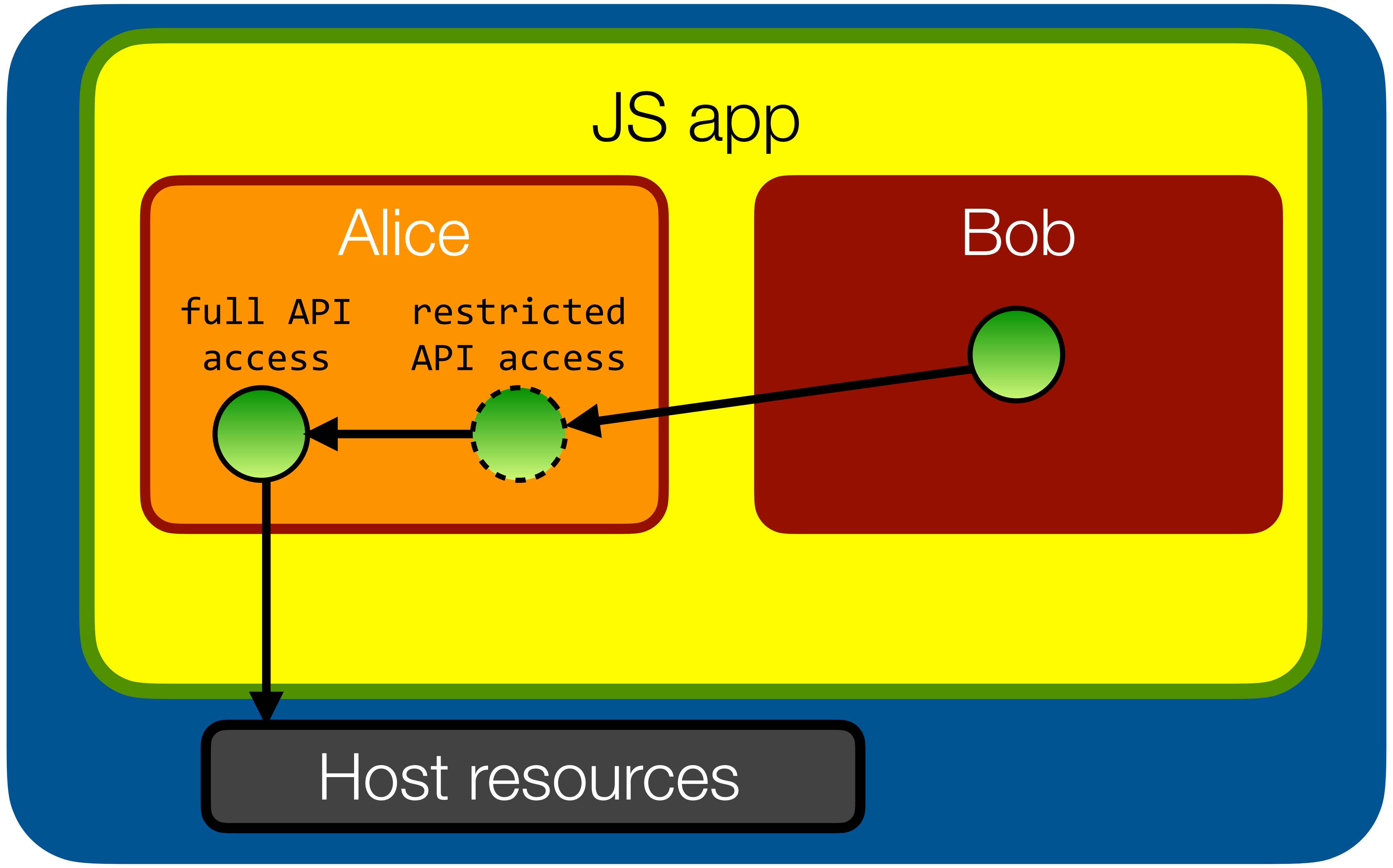

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.setup(log.write);
bob.setup(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]);
}
```


KU LEUVEN DistriNet

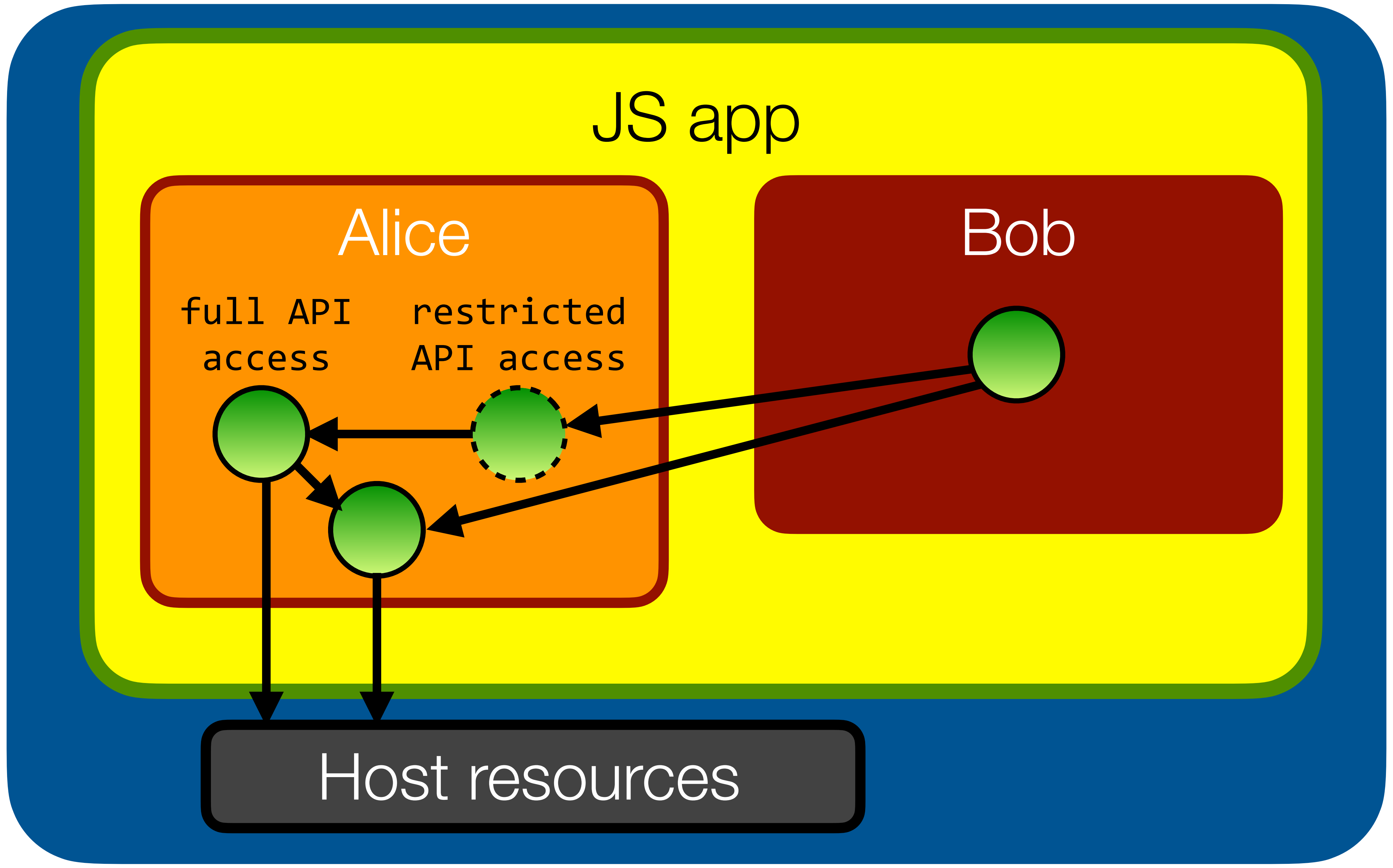
Taming is the process of restricting access to powerful APIs

- 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

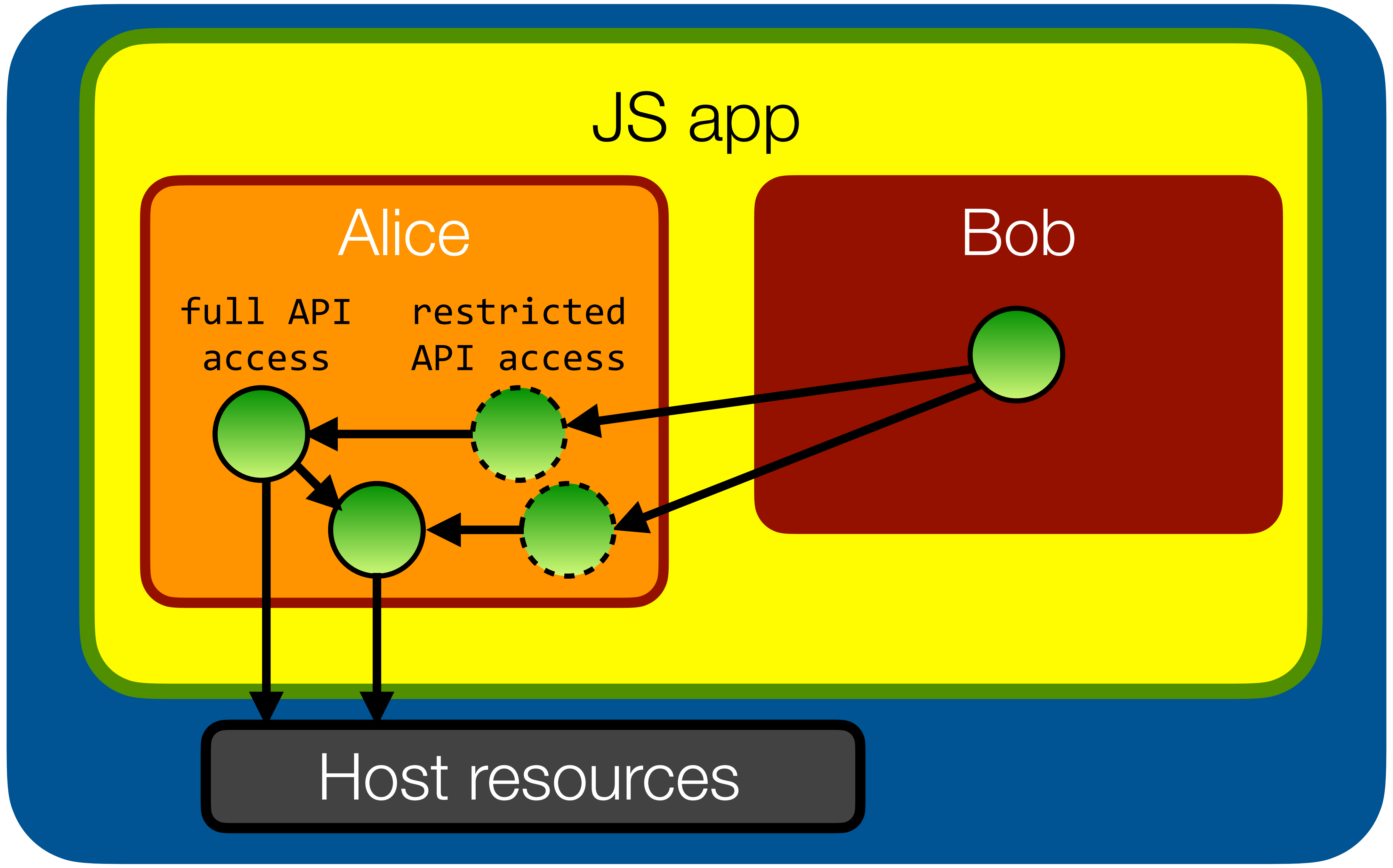
Potential **hazard**: the taming proxy must ensure it does not “leak” privileged access to host resources through the tamed API (e.g. through return values)



Taming is the process of restricting access to powerful APIs

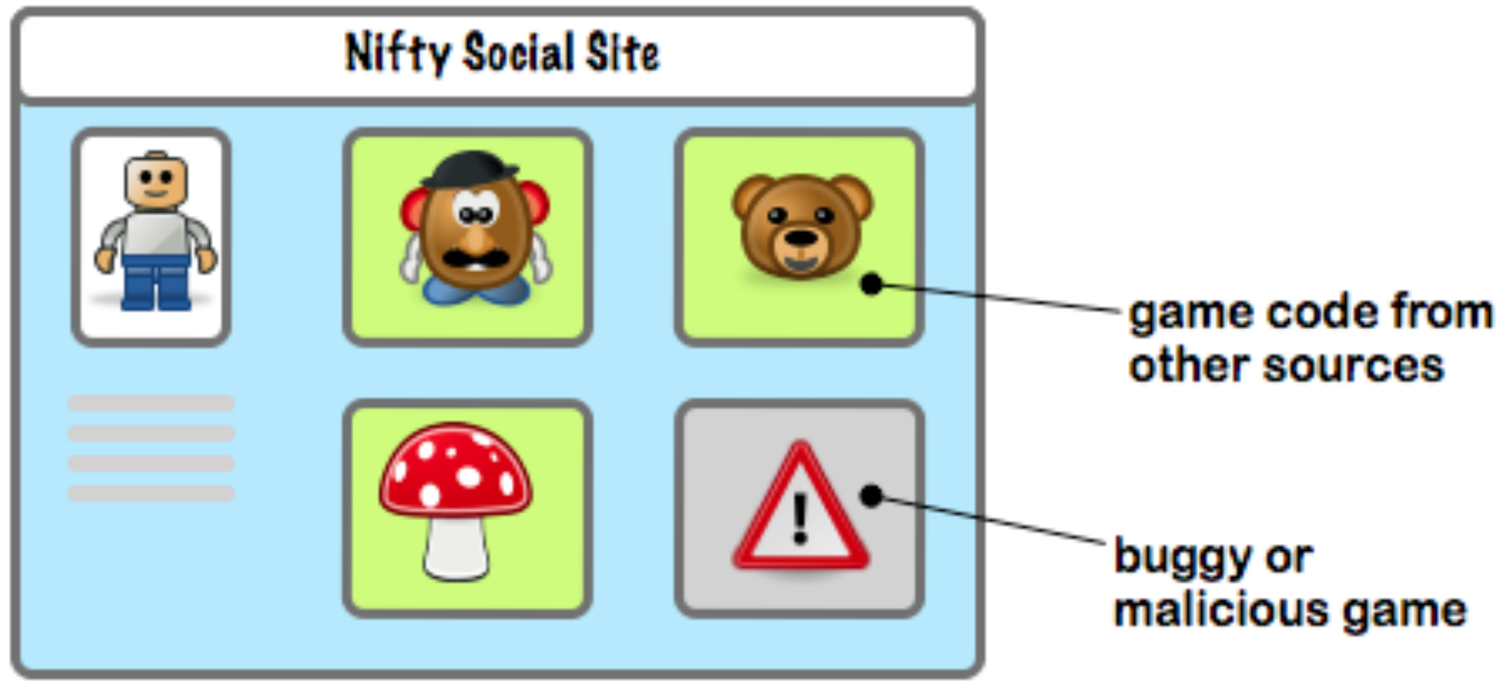
The **solution** is to transitively apply the proxy pattern to return values as well. This pattern is called a “**membrane**”

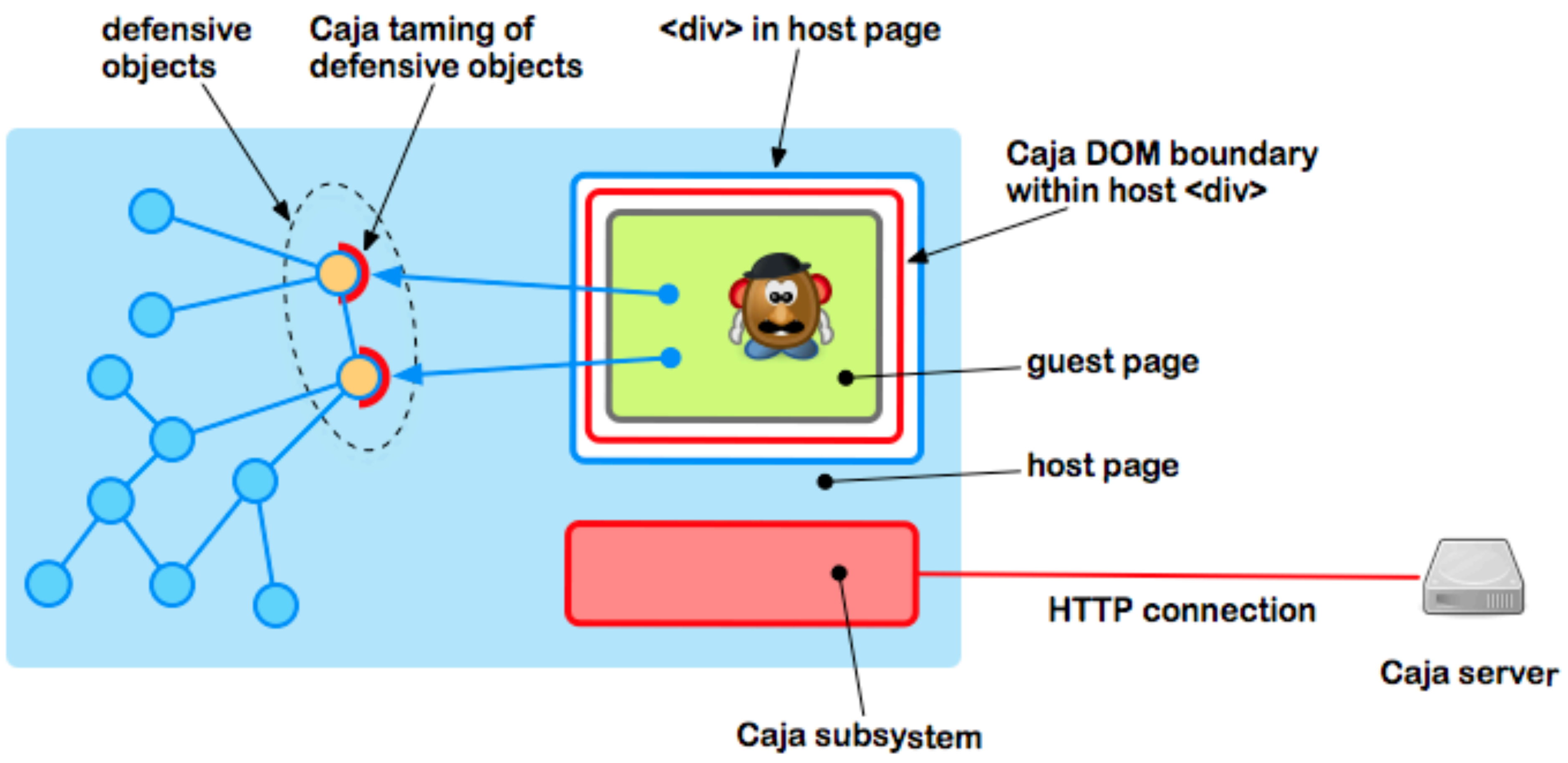
Deep dive blog post at tvcutsem.github.io/membranes



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 XS

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

MetaMask Snaps

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

Agoric Zoe

Uses **Hardened JS** to write smart contracts and Dapps

Figma plugins

Used **Realms** and **membranes** to embed third-party plugins for their editor

Mozilla Firefox

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

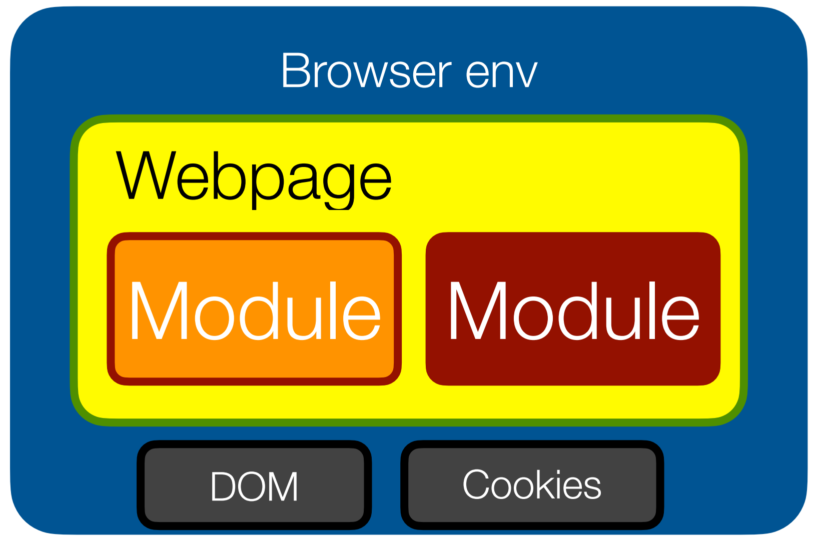
Salesforce Lightning

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

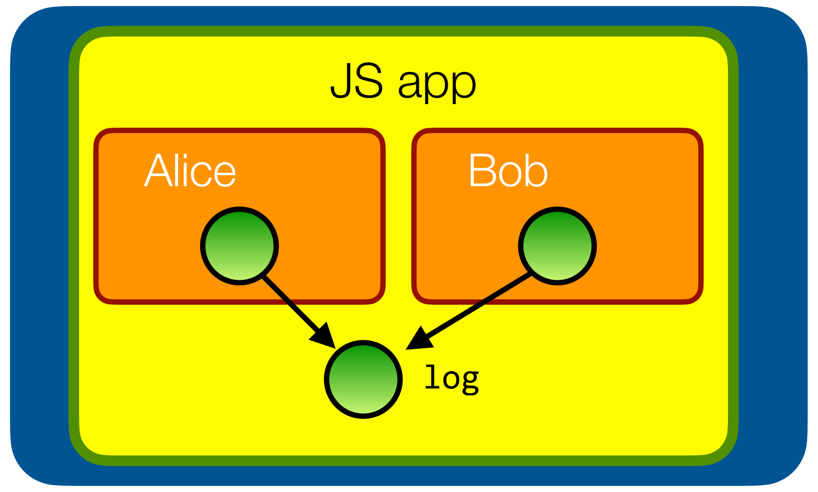
Summary

This Lecture: Recap

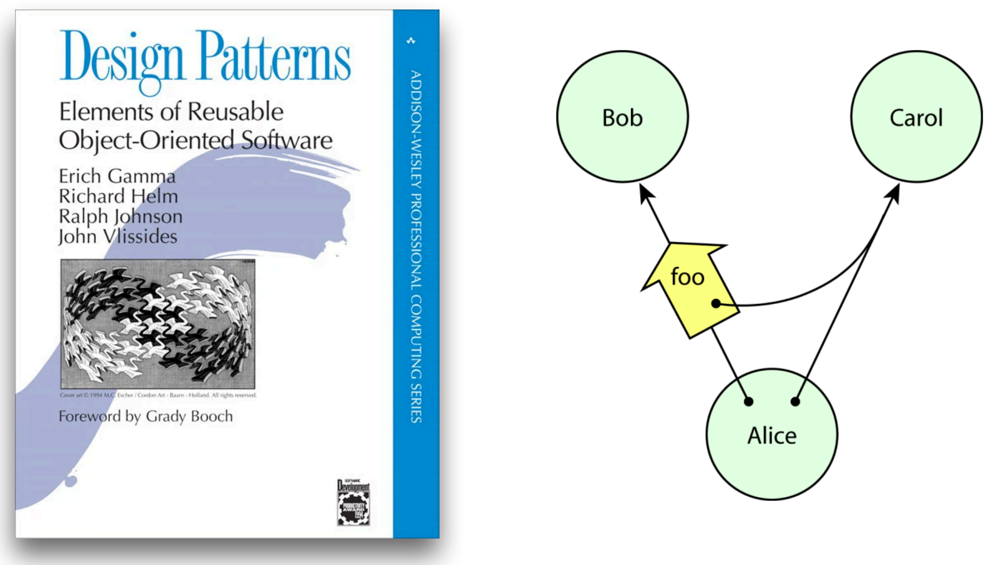
- Part I: **why module isolation** is critical to modern JavaScript applications



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



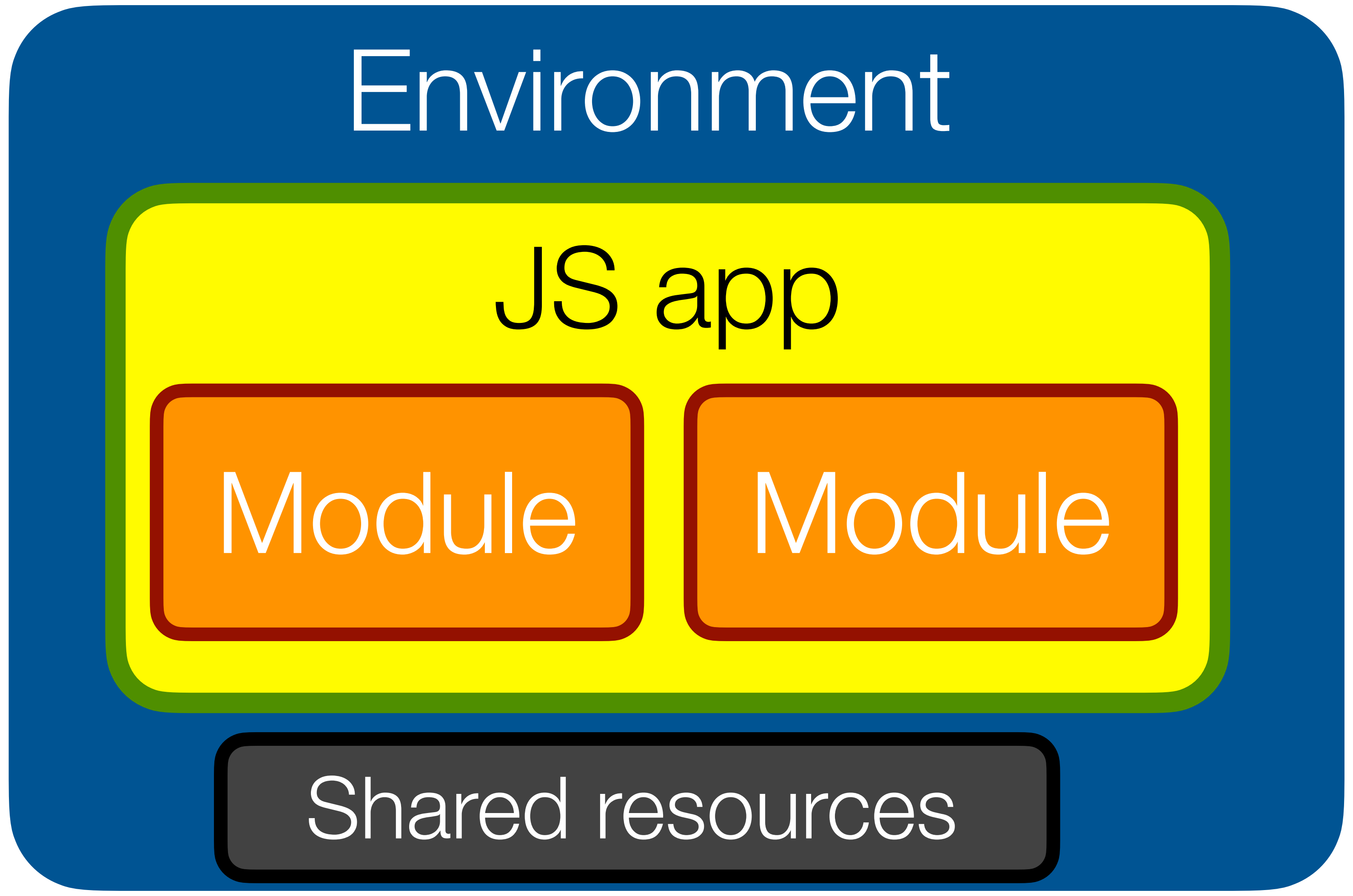
- Part III: safely composing modules using **least-authority patterns**



KU LEUVEN DistriNet

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: Use repeatable programming **patterns** to let modules **interact with "least authority"**
- Understanding these patterns is **important in a world of > 2,000,000 NPM modules** and an increasingly **hostile** threat landscape



“Security is just an extreme form of Modularity”

- Mark S. Miller
(Chief Scientist, Agoric)

Designing “least-authority” JavaScript apps

Tom Van Cutsem
KU Leuven

Questions?
tom.vancutsem@kuleuven.be

Further Reading

- **JavaScript-specific tools and resources**
- Hardened JavaScript: https://hardenedjs.org/
- Lavamoat: https://lavamoat.github.io/
- 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=PLzDw4TTug5O25J5M3fwErKImrjOrqGikj
- 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)

- **General background on capability-based security and POLA**
- Mark Miller, Ka-Ping Yee, Jonathan Shapiro, "Capability Myths Demolished": https://srl.cs.jhu.edu/pubs/SRL2003-02.pdf
- Chip Morningstar, "What are capabilities": http://habitatchronicles.com/2017/05/what-are-capabilities/ (broad historical perspective)
- Thomas Leonard, "Lambda capabilities": https://roscidus.com/blog/blog/2023/04/26/lambda-capabilities/ (excellent intro to capabilities for functional programmers)
- 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
- Dan Finlay and the Metamask team for their work on Lavamoat