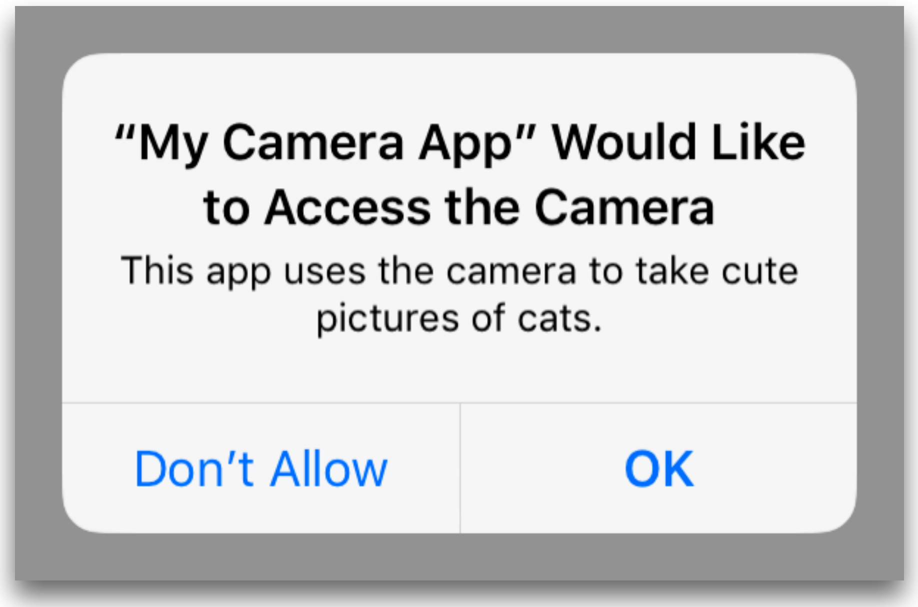
Object-capability security for JavaScript applications

Tom Van Cutsem
DistriNet KU Leuven

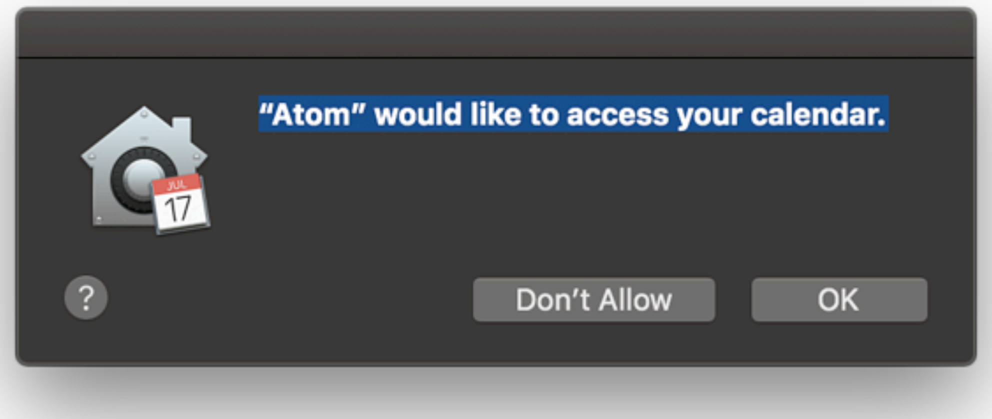
tvcutsem.github.io be.linkedin.com/in/tomvc github.com/tvcutsem twitter.com/tvcutsem

Application security & access control

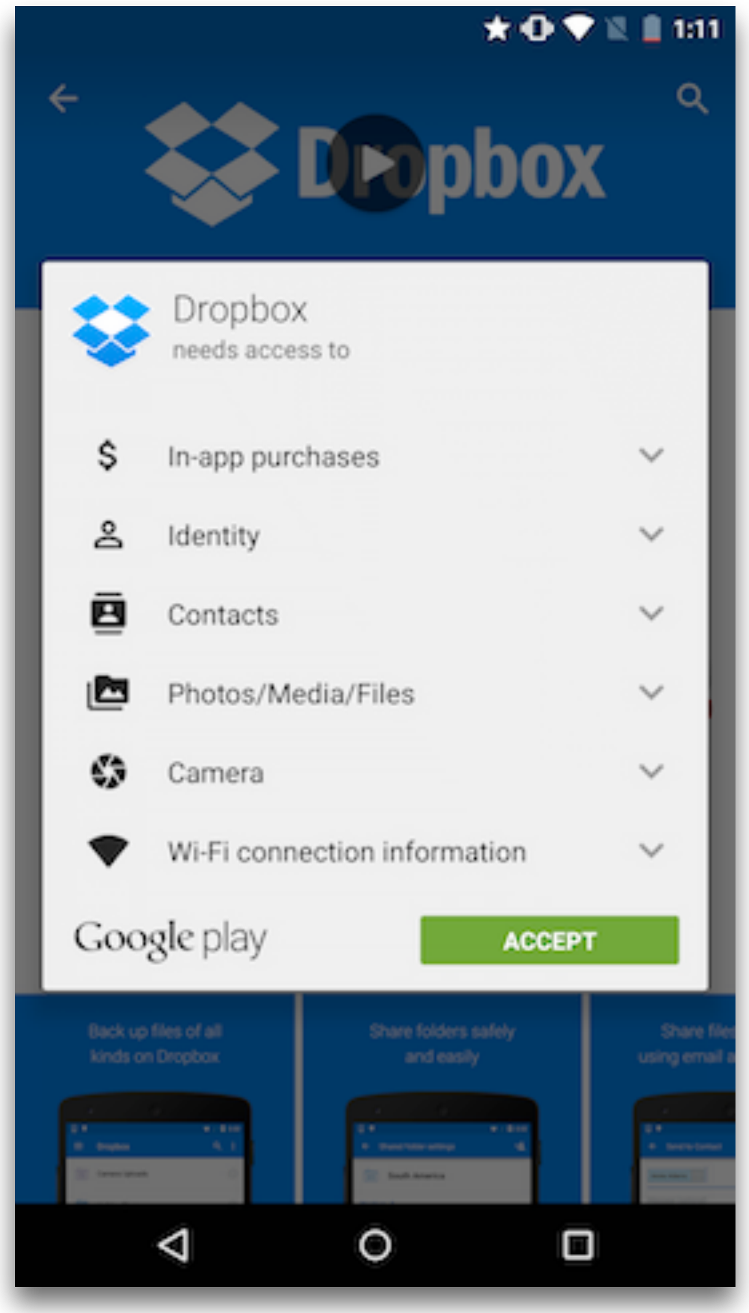


“My Camera App” Would Like to Access the Camera

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



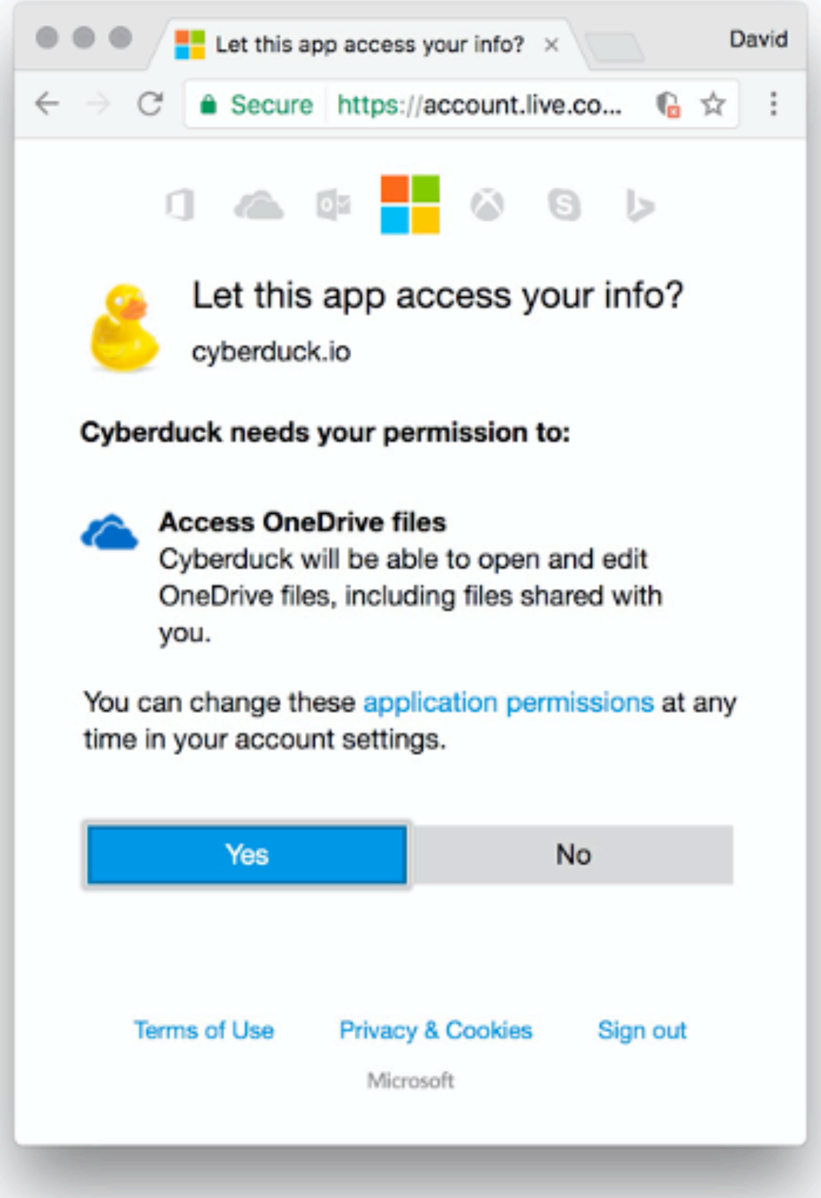
“Atom” would like to access your calendar.



Dropbox
needs access to

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

1:11
Dropbox
Google play
ACCEPT
Back up files of all kinds on Dropbox
Share folders safely and easily

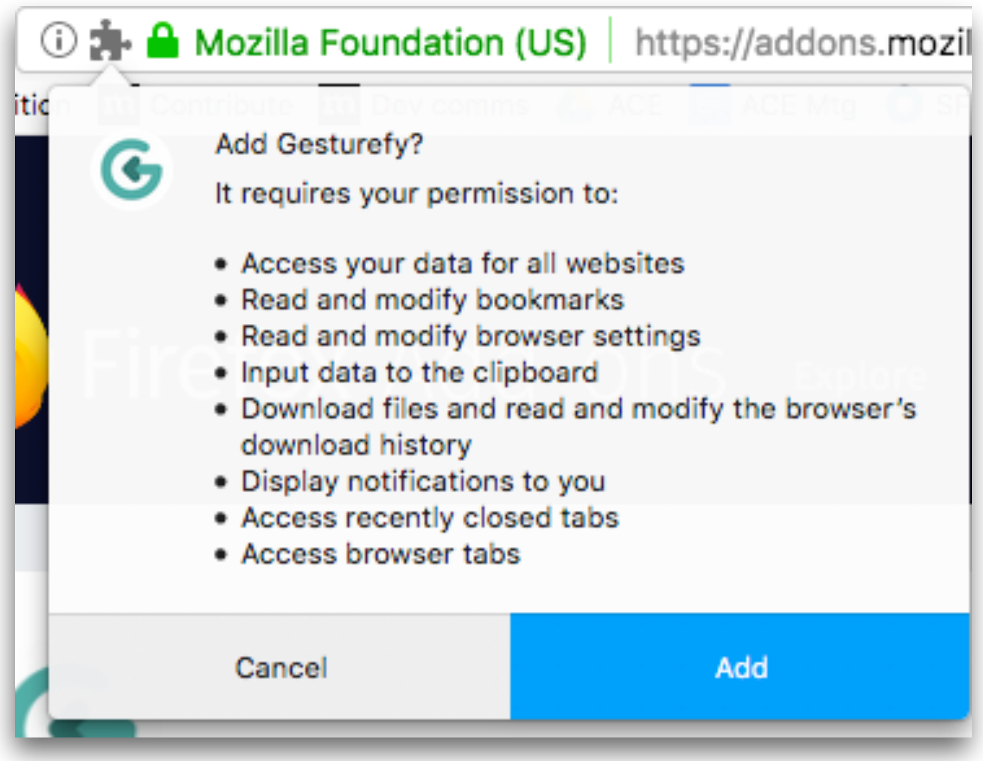


Let this app access your info?
cyberduck.io

Cyberduck needs your permission to:

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

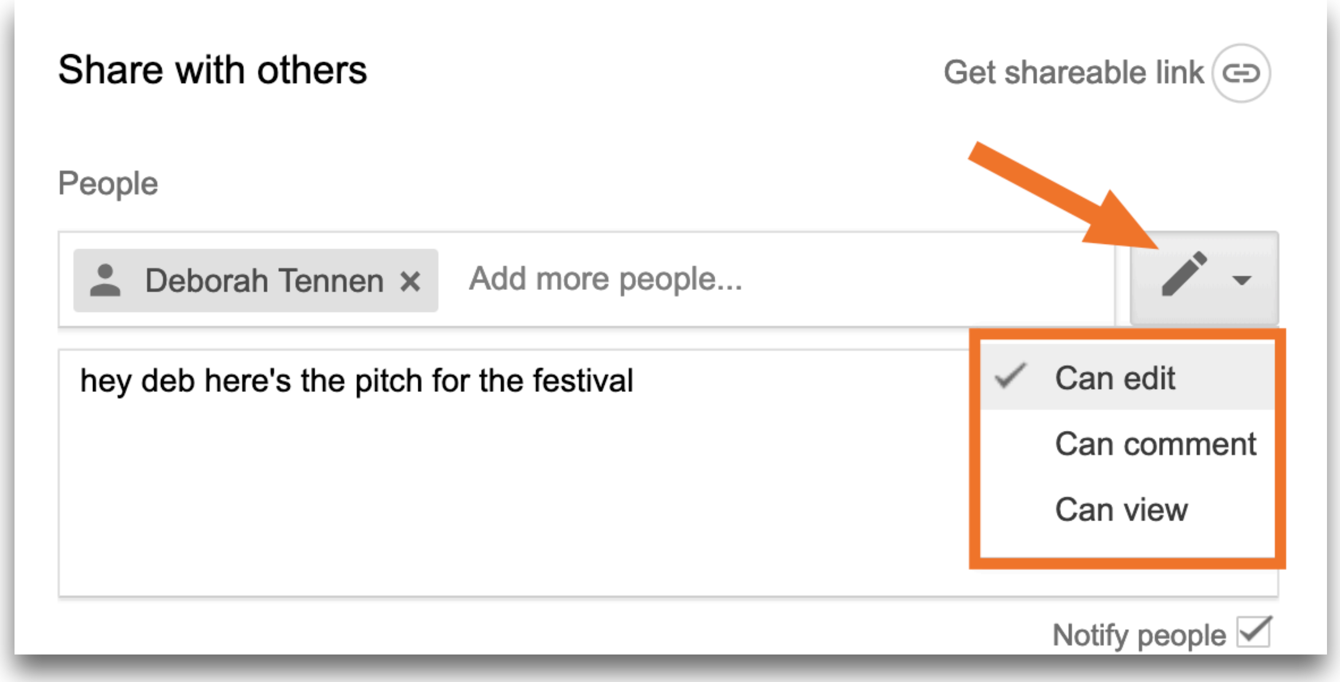
You can change these application permissions at any time in your account settings.



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



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 ☑

Web application security

same-origin policy

certificate pinning

OAuth

cookies

content security policy

CSRF

HSTS

html sanitization

OWASP

Open Web Application
Security Project

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

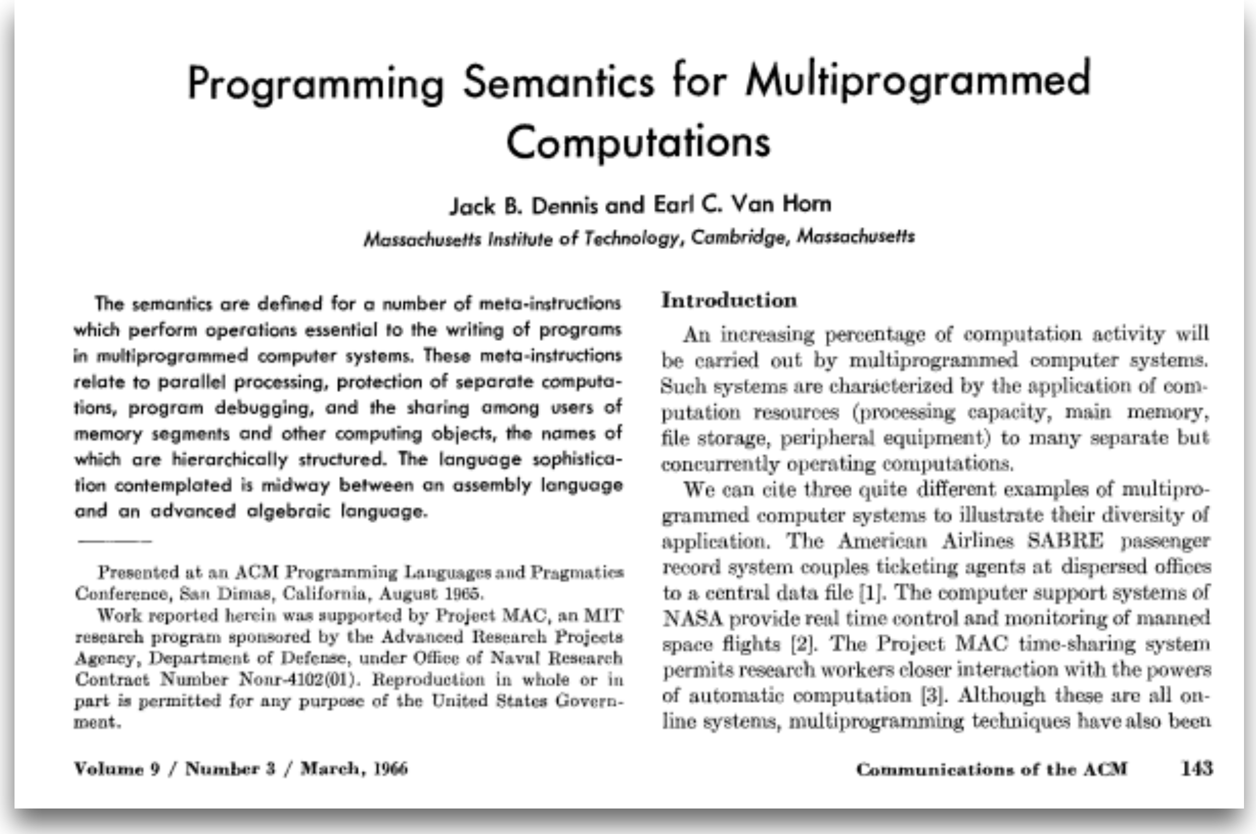
"Security is just the extreme of Modularity"

- Mark S. Miller
(Chief Scientist, Agoric)

Modularity: avoid needless dependencies (to prevent bugs)

Security: avoid needless vulnerabilities (to prevent exploits)

Object-capability security: a brief history



Programming Semantics for Multiprogrammed Computations

Jack B. Dennis and Earl C. Van Horn

Massachusetts Institute of Technology, Cambridge, Massachusetts

The semantics are defined for a number of meta-instructions which perform operations essential to the writing of programs in multiprogrammed computer systems. These meta-instructions relate to parallel processing, protection of separate computations, program debugging, and the sharing among users of memory segments and other computing objects, the names of which are hierarchically structured. The language sophistication contemplated is midway between an assembly language and an advanced algebraic language.

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

Work reported herein was supported by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

Introduction

An increasing percentage of computation activity will be carried out by multiprogrammed computer systems. Such systems are characterized by the application of computation resources (processing capacity, main memory, file storage, peripheral equipment) to many separate but concurrently operating computations.

We can cite three quite different examples of multiprogrammed computer systems to illustrate their diversity of application. The American Airlines SABRE passenger record system couples ticketing agents at dispersed offices to a central data file [1]. The computer support systems of NASA provide real time control and monitoring of manned space flights [2]. The Project MAC time-sharing system permits research workers closer interaction with the powers of automatic computation [3]. Although these are all on-line systems, multiprogramming techniques have also been

Volume 9 / Number 3 / March, 1966 **Communications of the ACM** 143

Communications of the ACM, Vol 9, No 3, March 1966

1966

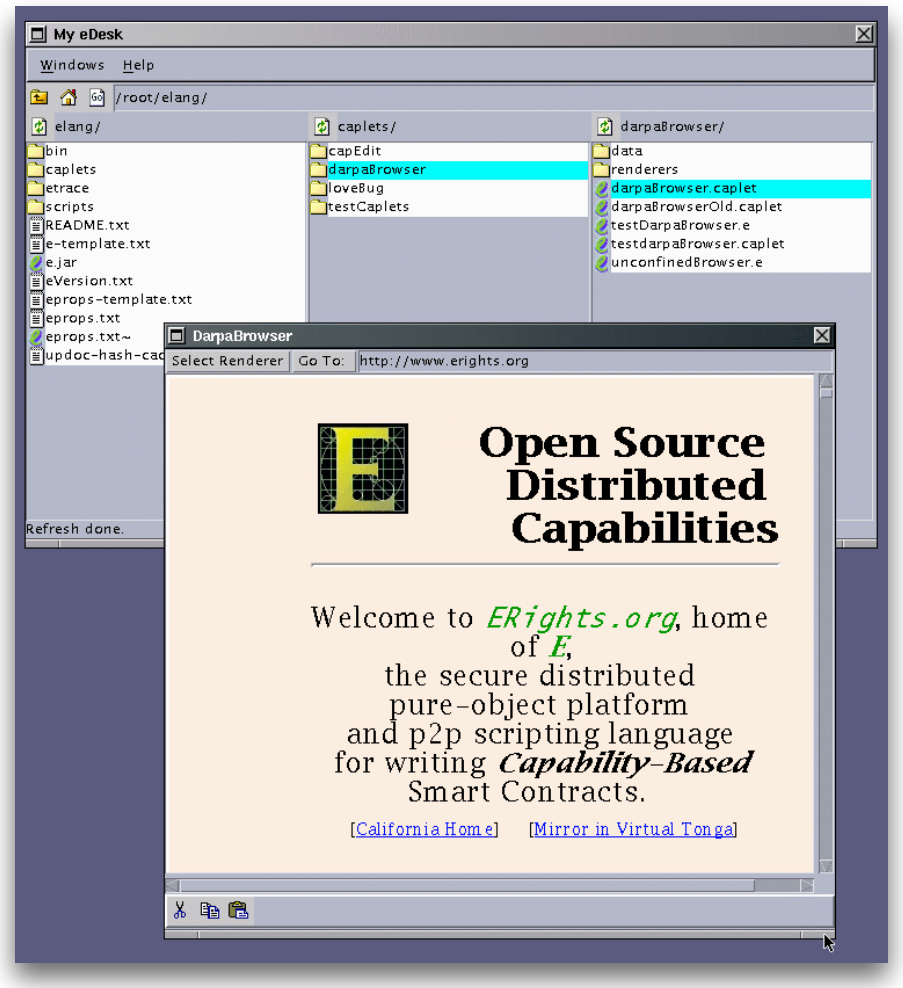
Seminal paper on capabilities in operating systems by **Dennis & Van Horn**

SDS 940 Time-sharing computer

See: Why KeyKOS is fascinating

1977

GNOSIS (later KeyKOS)
First capability-secure operating system developed by Tymshare



My eDesk

Windows Help

/root/elang/

elang/	caplets/	darpaBrowser/
bin	capEdit	data
caplets	darpaBrowser	renderers
etrace	loveBug	darpaBrowser.caplet
scripts	testCaplets	darpaBrowserOld.caplet
README.txt		testDarpaBrowser.e
e-template.txt		testdarpaBrowser.caplet
e.jar		unconfinedBrowser.e
eVersion.txt		
eprops-template.txt		
eprops.txt		
eprops.txt~		
updoc-hash-cac		

Refresh done.

DarpaBrowser

Select Renderer | Go To: | http://www.erights.org

Open Source Distributed Capabilities

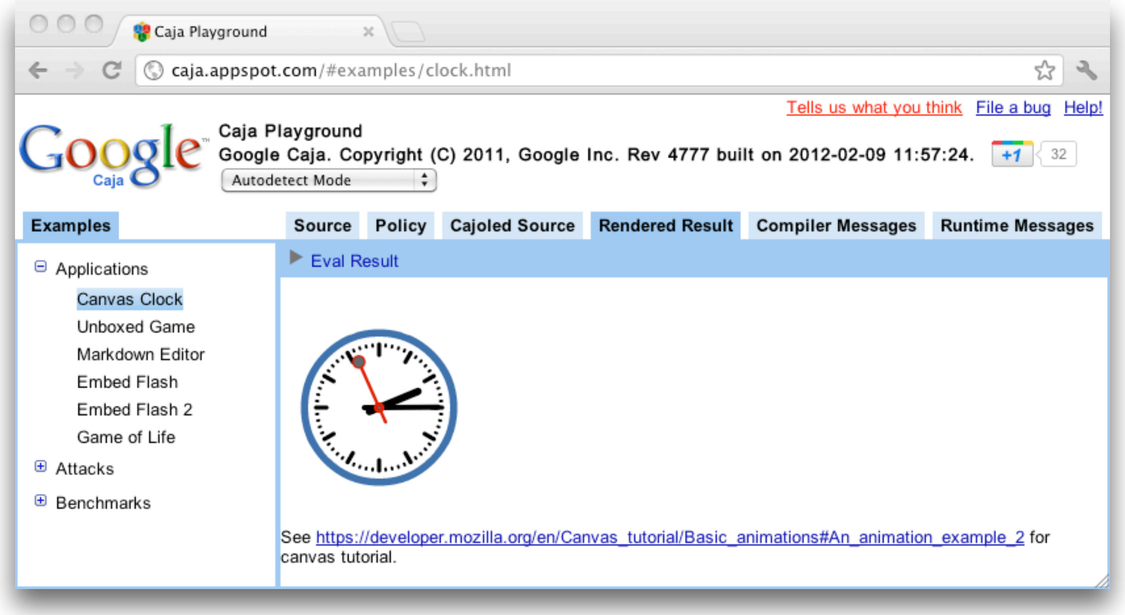
Welcome to *ERights.org*, home of *E*,
the secure distributed
pure-object platform
and p2p scripting language
for writing ***Capability-Based***
Smart Contracts.

[California Home] [Mirror in Virtual Tonga]

"Capdesk", a capability-based file browser, written in E

1997

E, a pure "object-capability" programming language developed by Electric Communities



Caja Playground

caja.appspot.com/#examples/clock.html

Tells us what you think File a bug Help!

Caja Playground
Google Caja. Copyright (C) 2011, Google Inc. Rev 4777 built on 2012-02-09 11:57:24.

Autodetect Mode

Examples | Source | Policy | Cajoled Source | Rendered Result | Compiler Messages | Runtime Messages

- Applications
  - Canvas Clock
  - Unboxed Game
  - Markdown Editor
  - Embed Flash
  - Embed Flash 2
  - Game of Life
- Attacks
- Benchmarks

Eval Result

See https://developer.mozilla.org/en/Canvas_tutorial/Basic_animations#An_animation_example_2 for canvas tutorial.

Google Caja enables safe embedding of dynamic Web content on a webpage

2008

Google **Caja** project creates a capability-secure subset of JavaScript for Web security

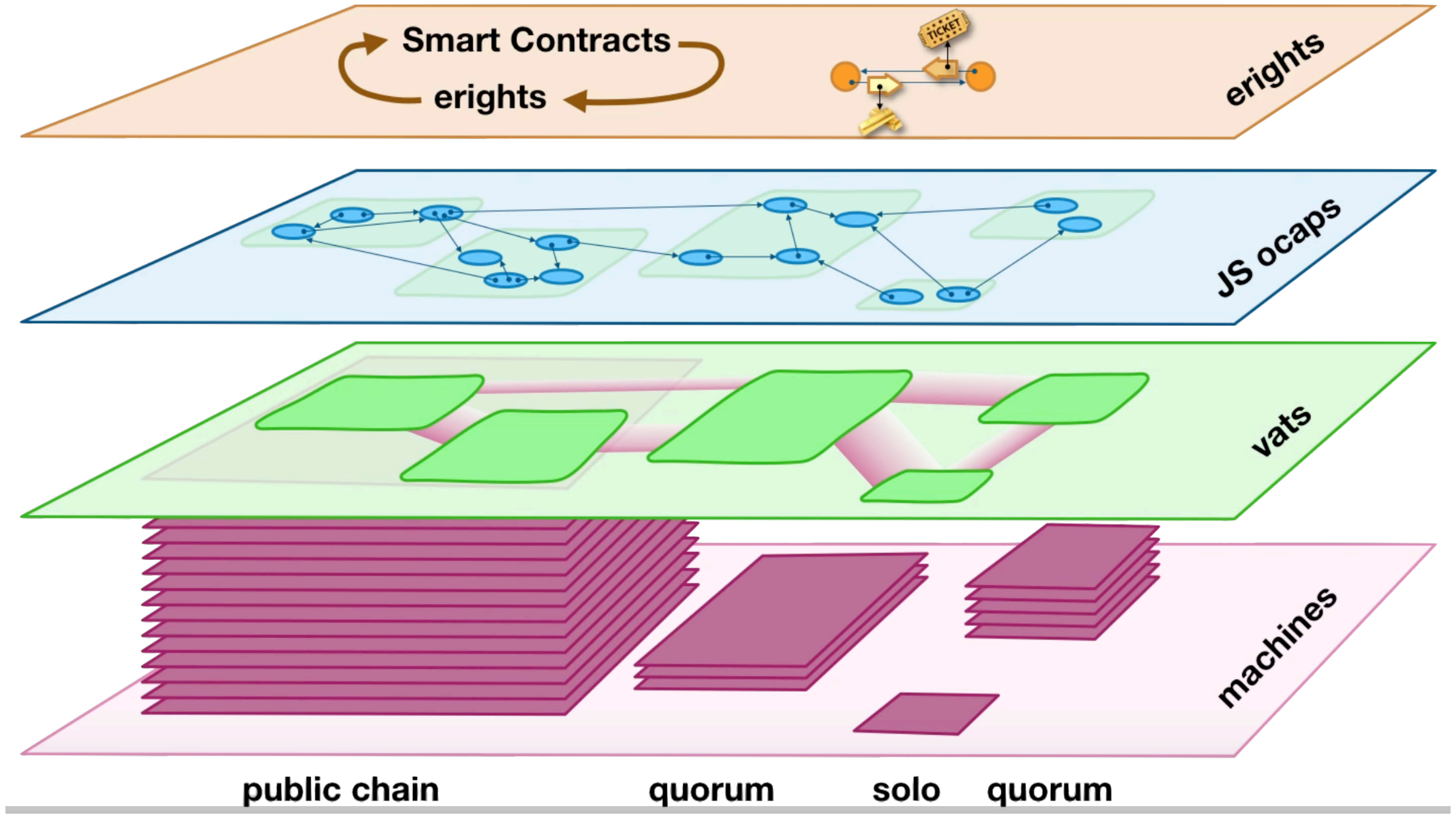
JavaScript & Web3: Agoric's DeFi platform

AGORIC

Digital assets (tokens)

"Hardened" JavaScript

Cosmos Blockchain

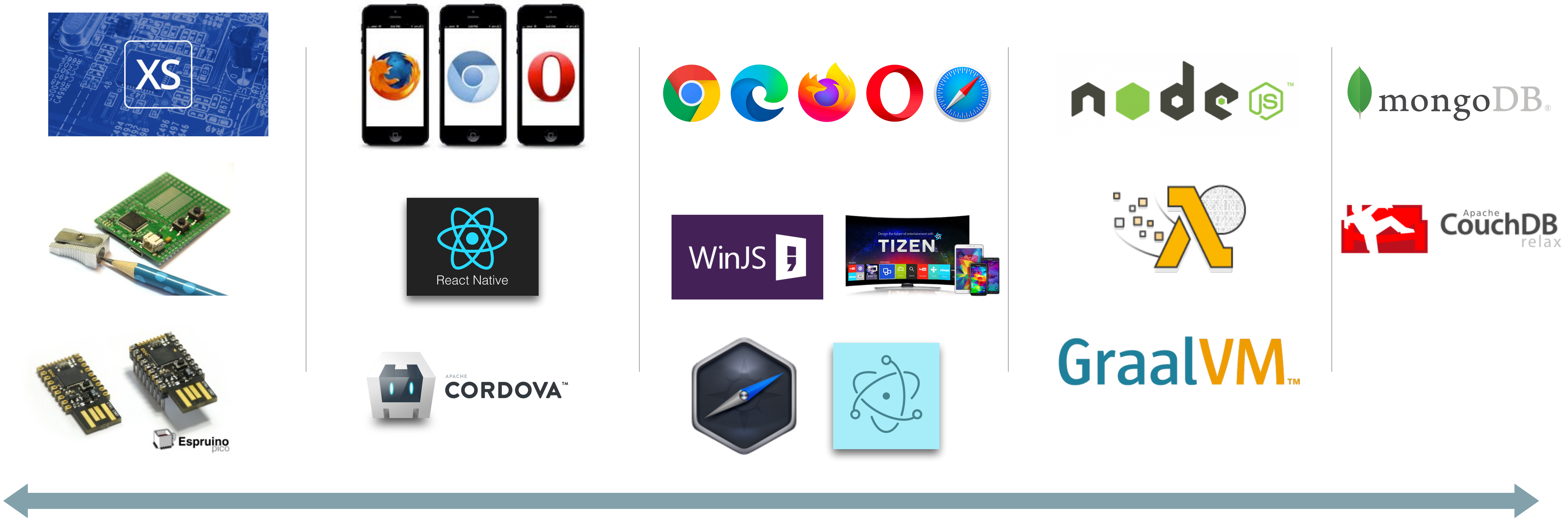


This Lecture

- Part I: why application security is critical to JavaScript applications
- Part II: the Principle of Least Authority, by example
- Part III: the object-capability model of access control
- Part IV: object-capability patterns

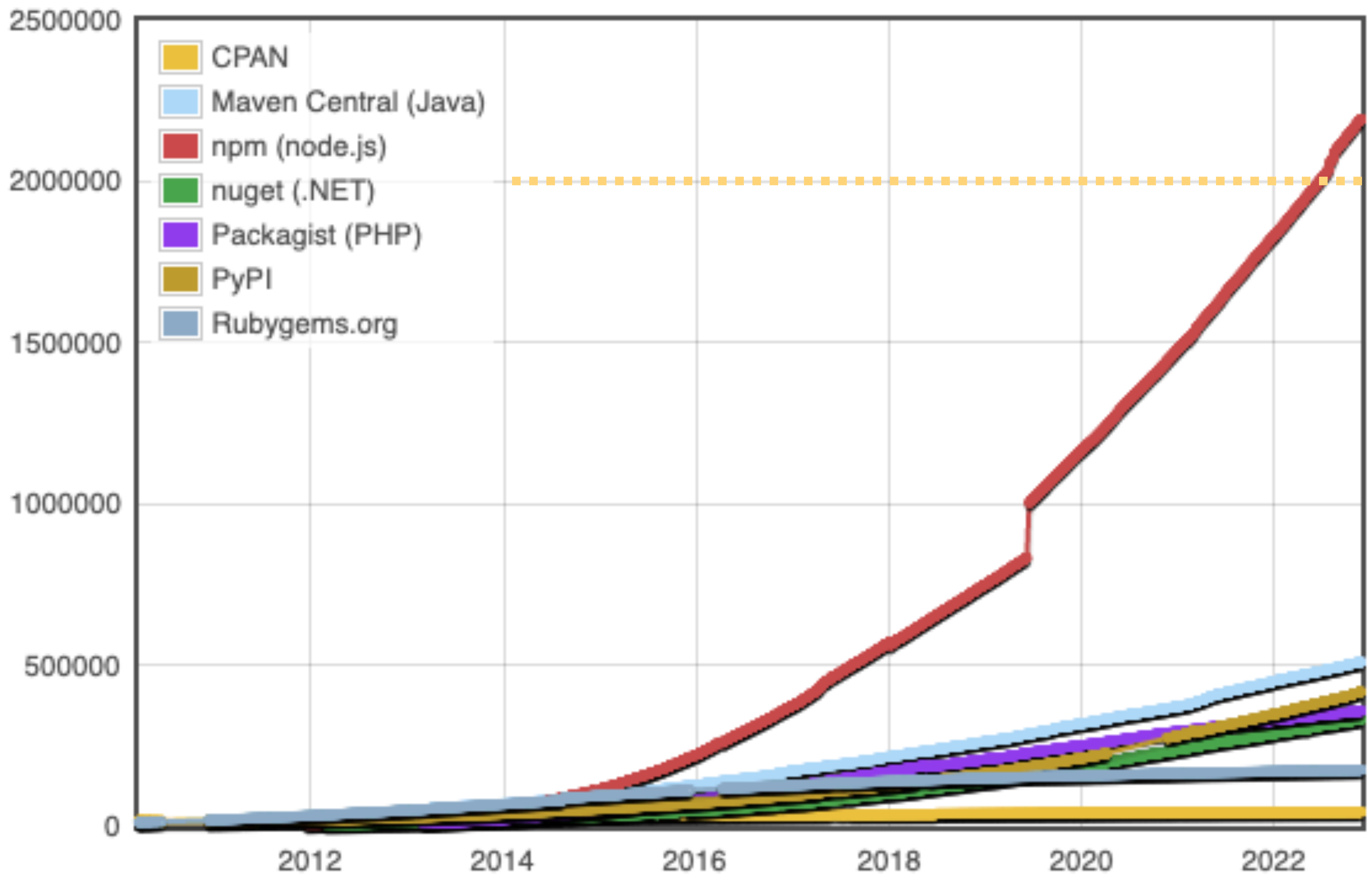
Part I
Why application security is critical to JavaScript applications

It’s no longer just about the Web. JavaScript is used widely across tiers



Embedded Mobile Desktop/Native Server Database

JavaScript applications are now built from thousands of modules



(source: modulecounts.com, Nov 2022)

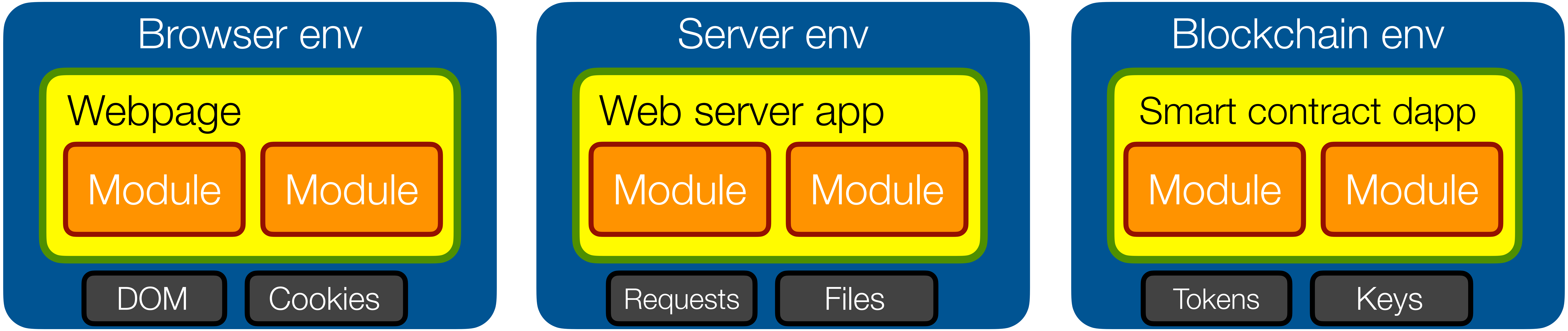
2,000,000 modules on NPM

“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)

Composable code: 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

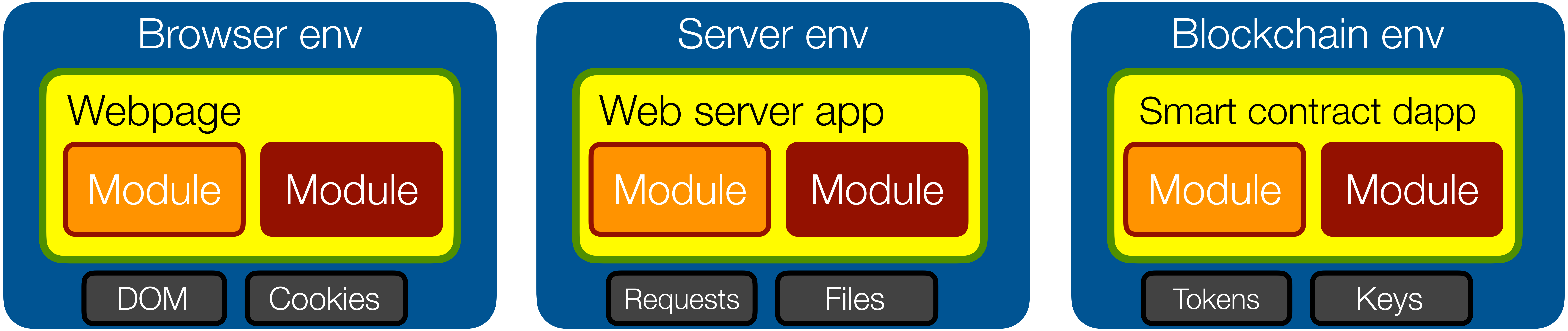
Blockchain env

Smart contract dapp

Module Module

Tokens Keys

What can happen when code goes **rogue**?



Browser env

Webpage

Module

Module

DOM

Cookies

Server env

Web server app

Module

Module

Requests

Files

Blockchain env

Smart contract dapp

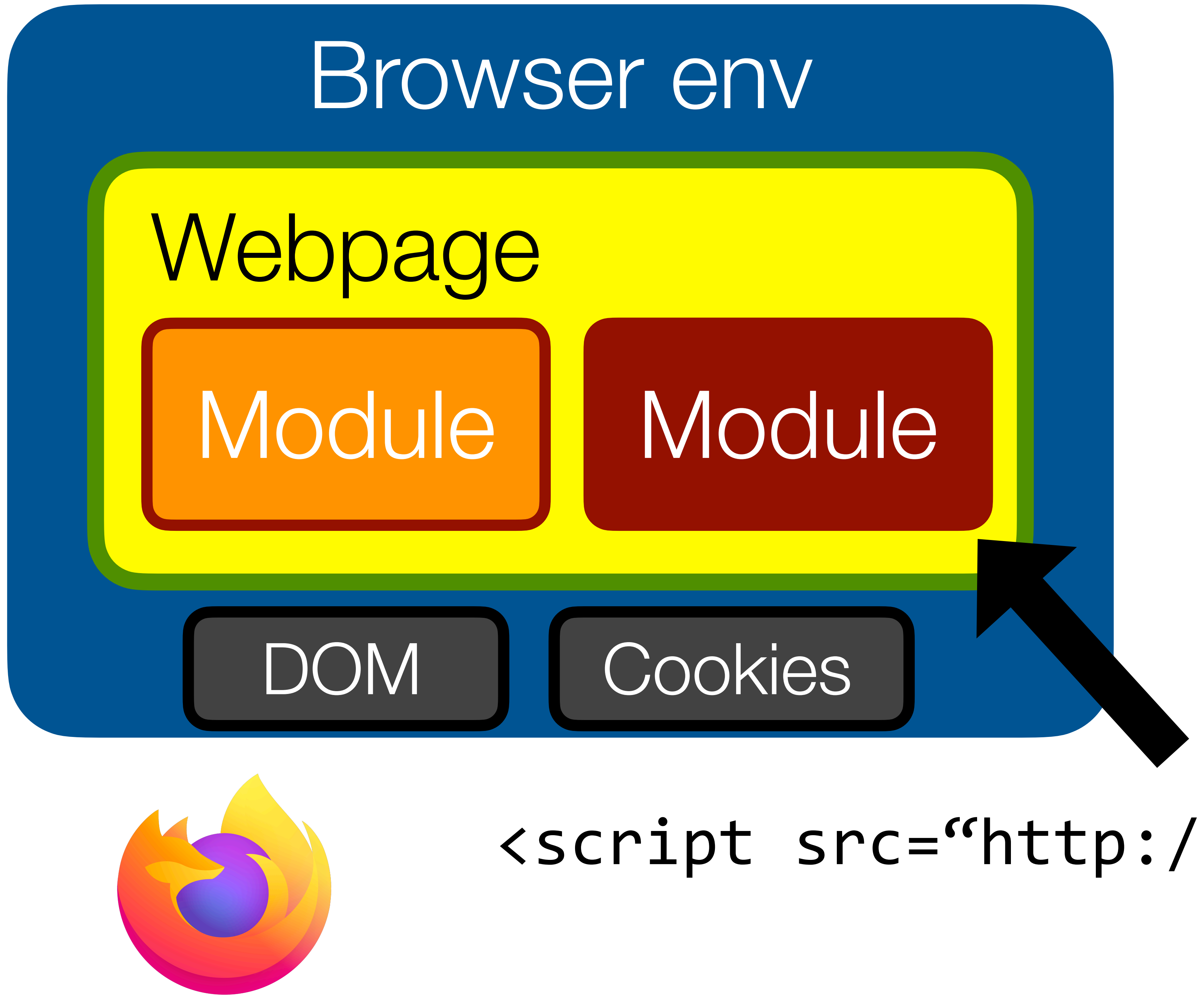
Module

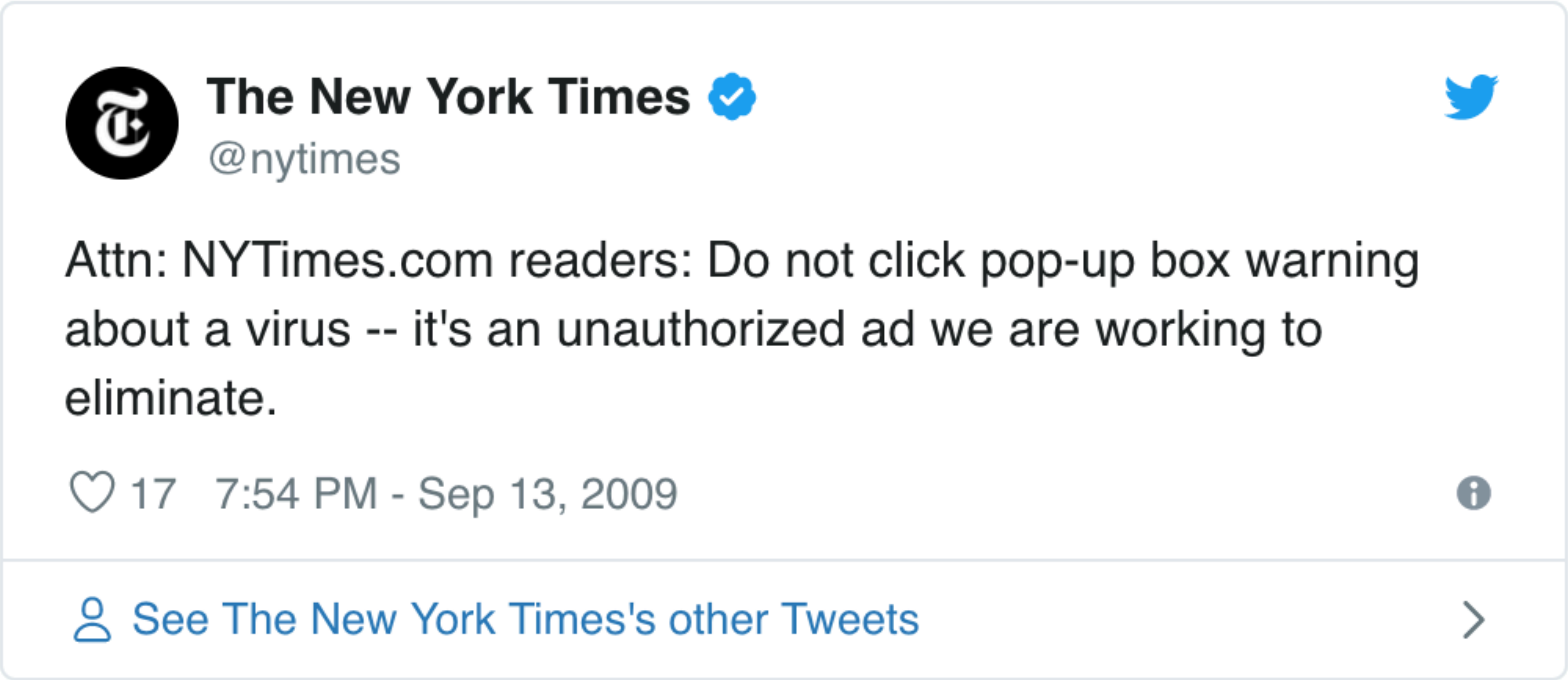
Module

Tokens

Keys

What can happen when code goes **rogue**?




The New York Times @nytimes

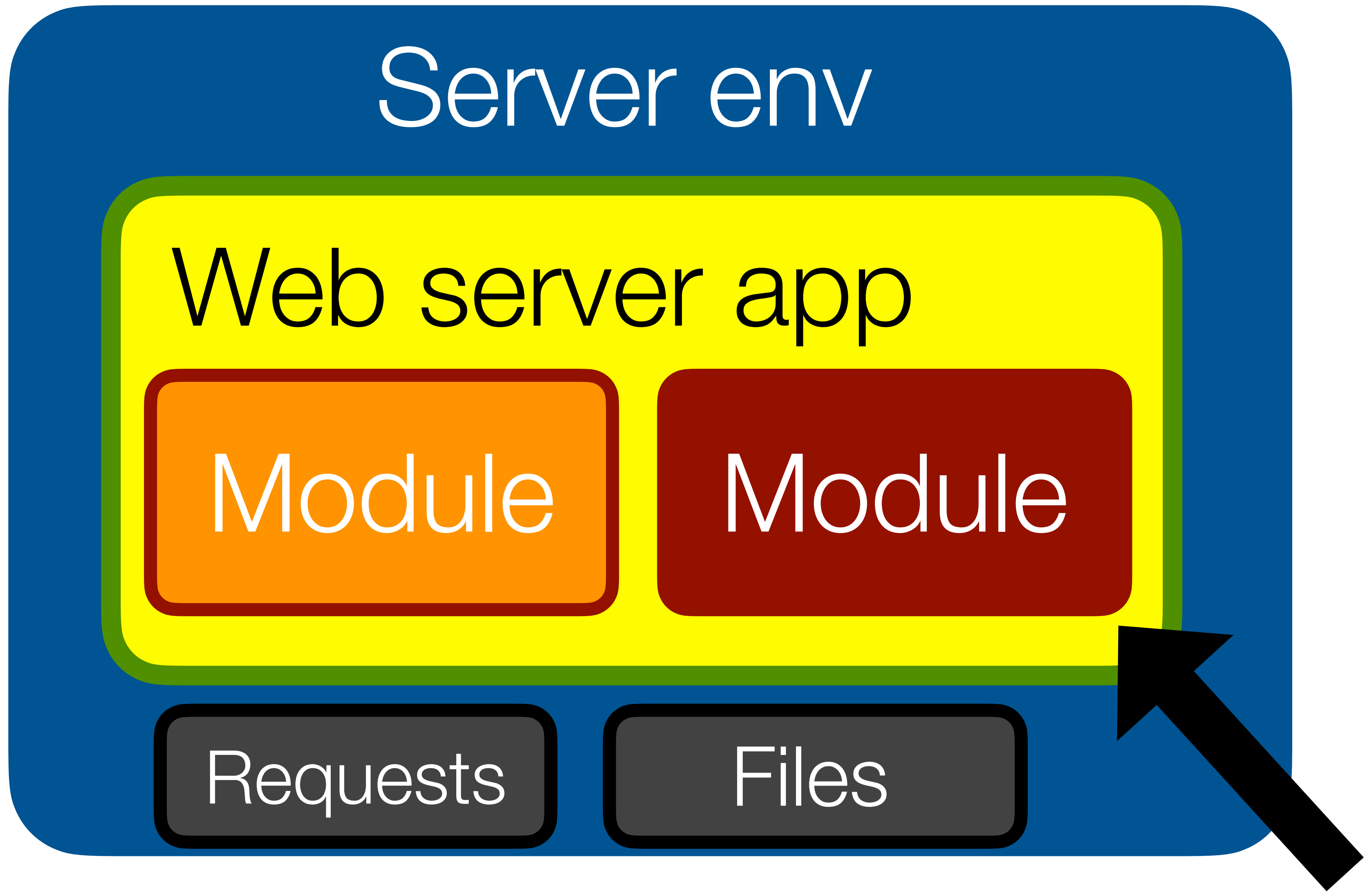
Attn: NYTimes.com readers: Do not click pop-up box warning about a virus -- it's an unauthorized ad we are working to eliminate.

♡ 17 7:54 PM - Sep 13, 2009

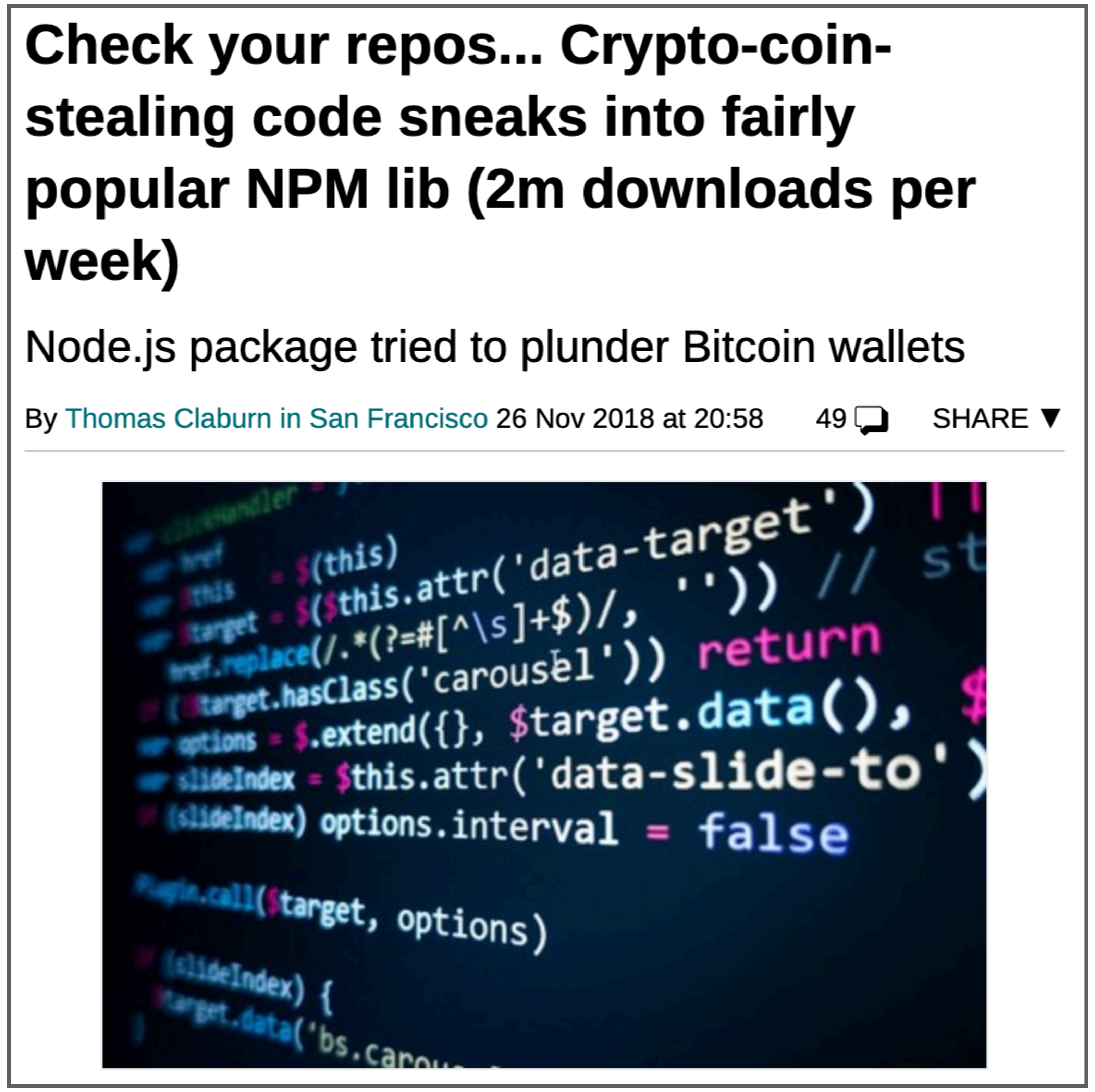
See The New York Times's other Tweets

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

What can happen when code goes **rogue**?



```
npm install event-stream
```



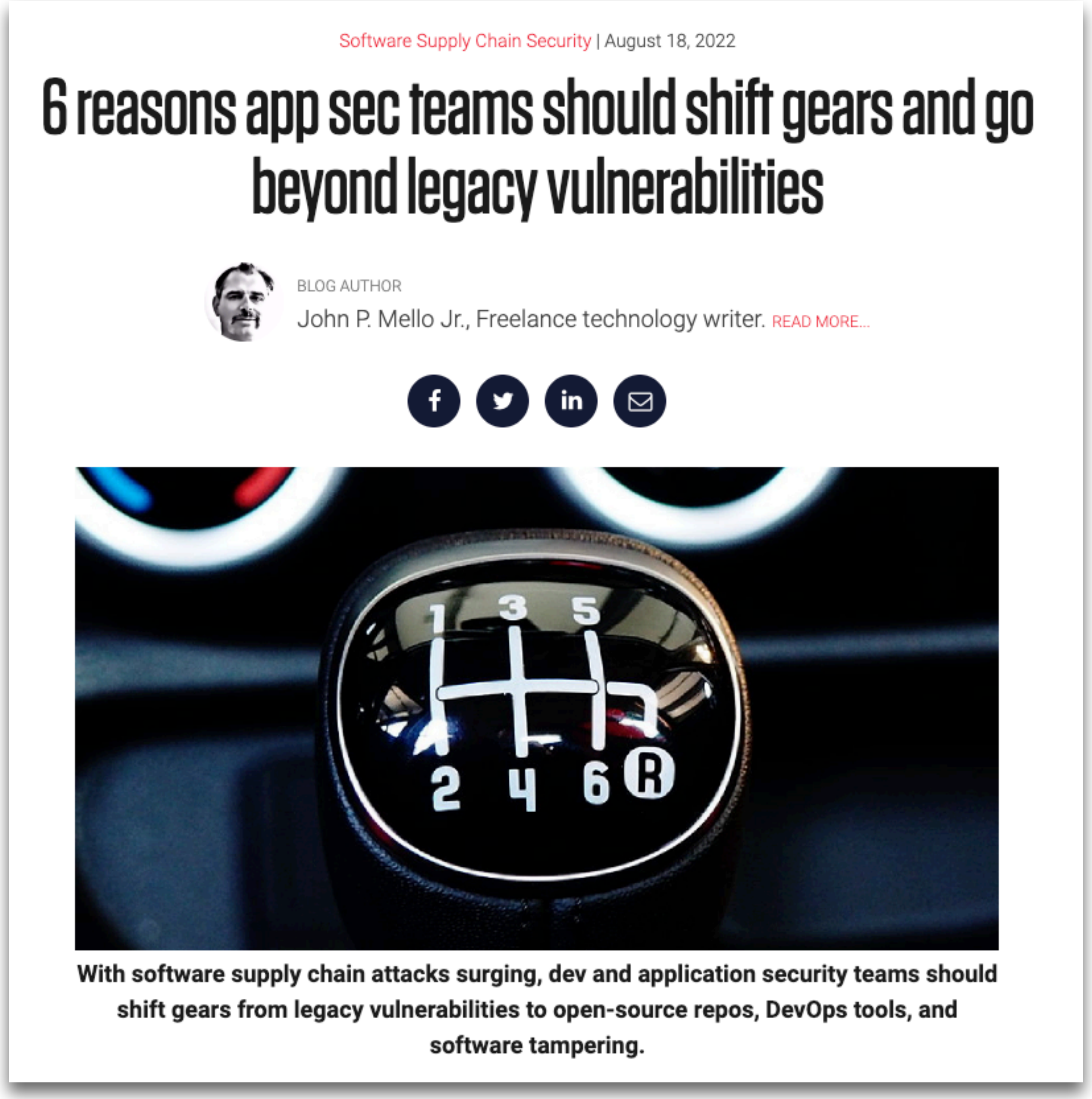
Check your repos... Crypto-coin-stealing code sneaks into fairly popular NPM lib (2m downloads per week)

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



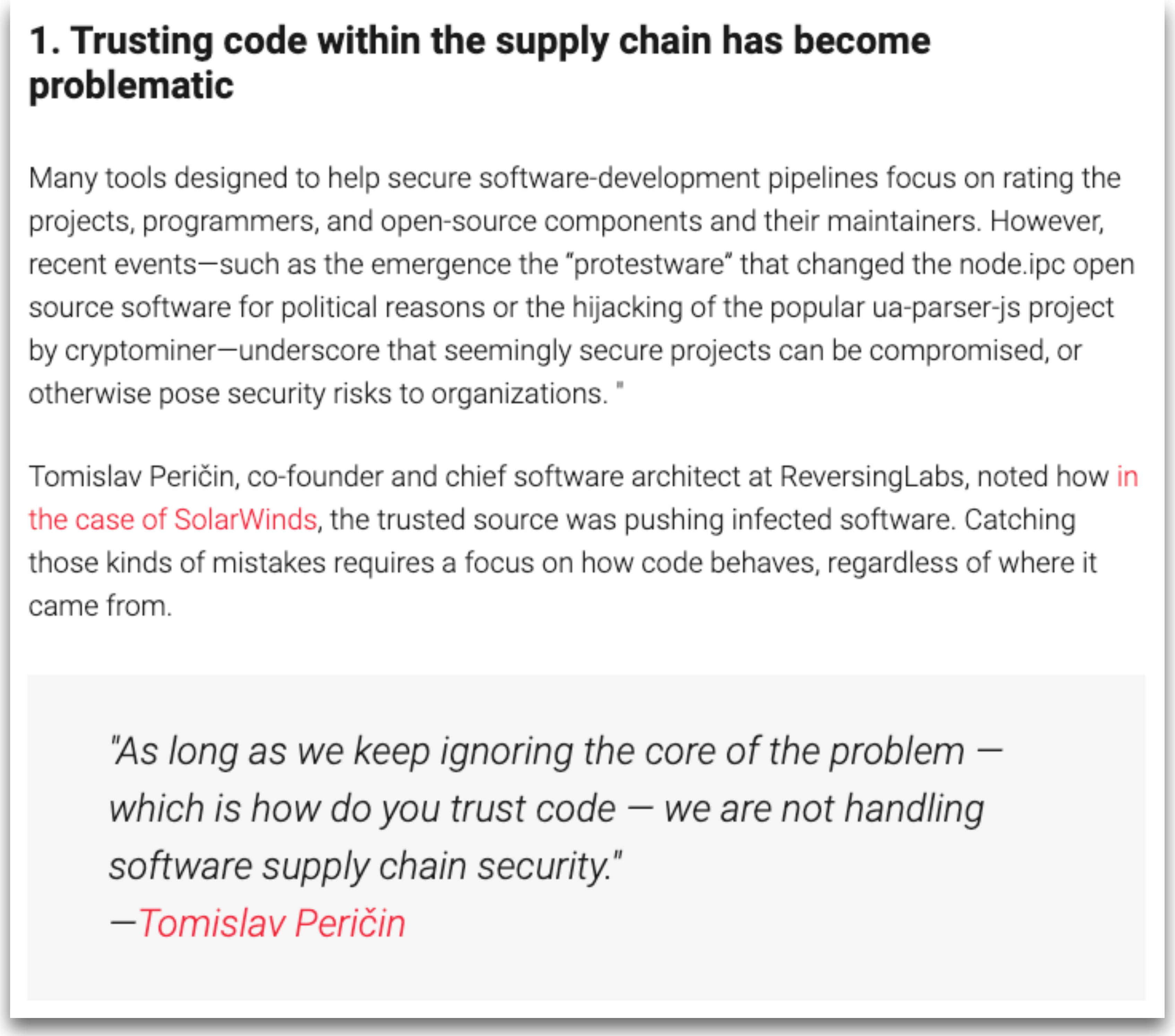
Software Supply Chain Security | August 18, 2022

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.

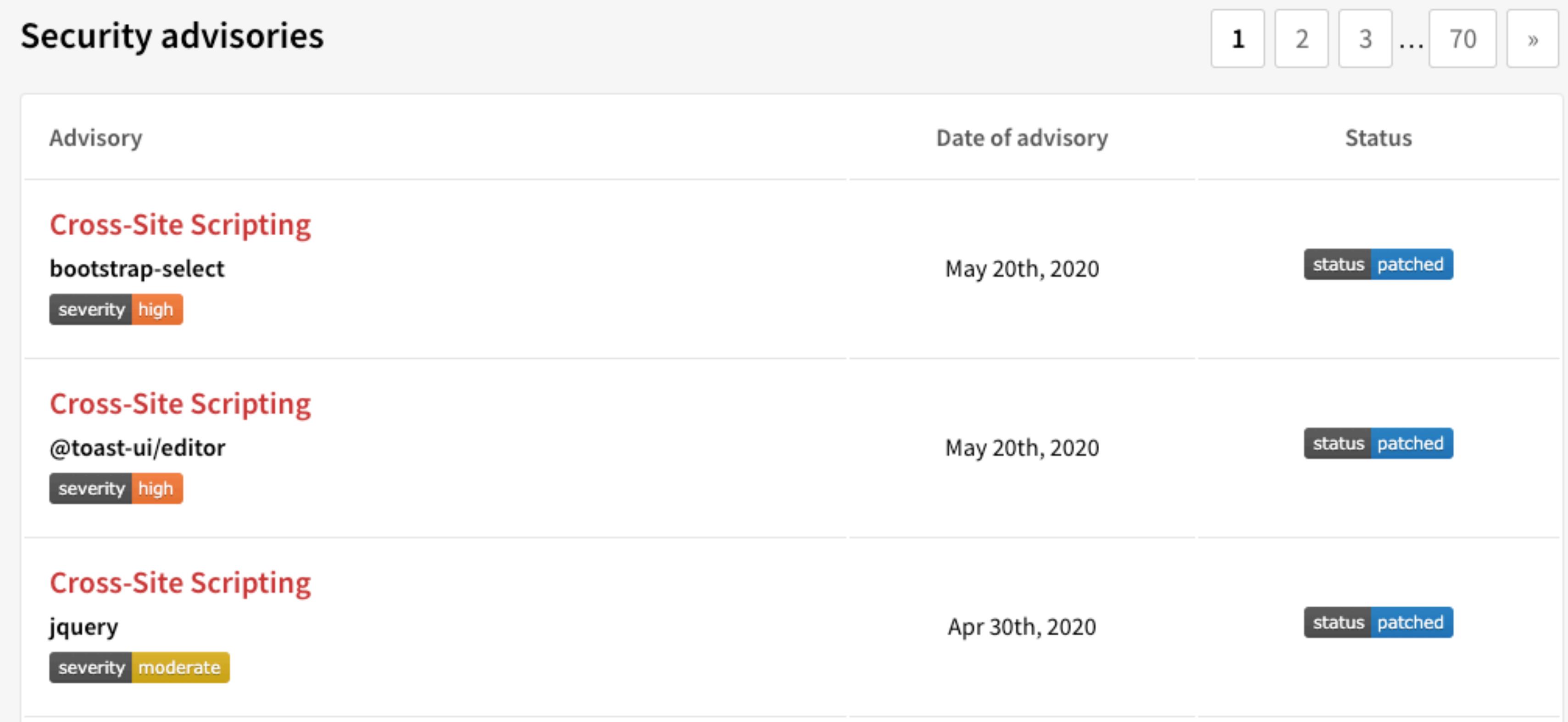
> *"As long as we keep ignoring the core of the problem — which is how do you trust code — we are not handling software supply chain security."*
> *—Tomislav Peričin*

(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

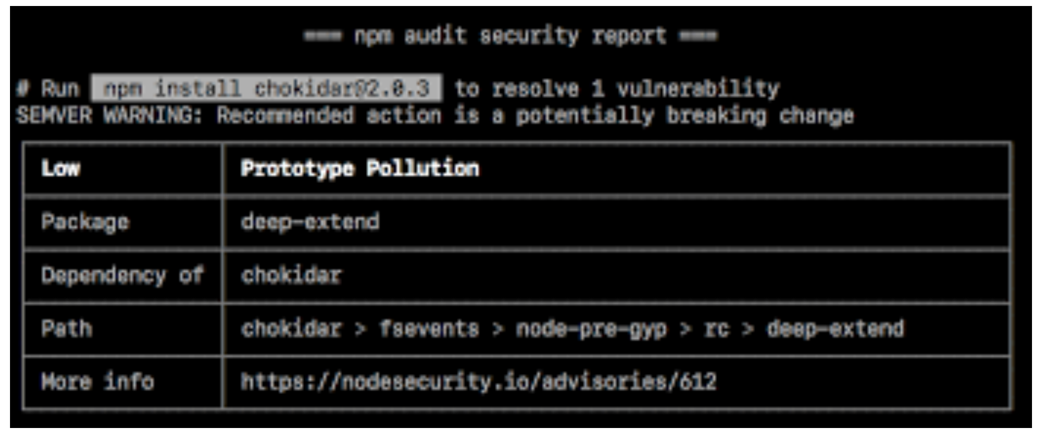
npm security advisories



Security advisories

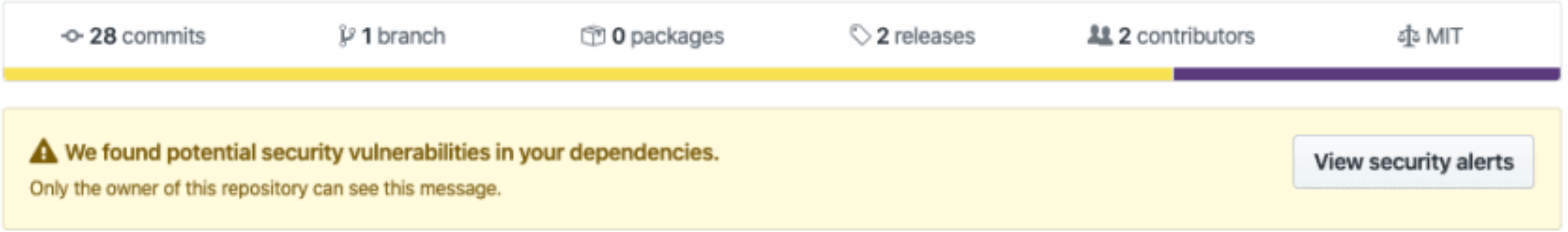
Advisory	Date of advisory	Status
Cross-Site Scripting bootstrap-select severity high	May 20th, 2020	status patched
Cross-Site Scripting @toast-ui/editor severity high	May 20th, 2020	status patched
Cross-Site Scripting jquery severity moderate	Apr 30th, 2020	status patched

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               │
└───────────────┴──────────────────────────────────────────────────────┘
```

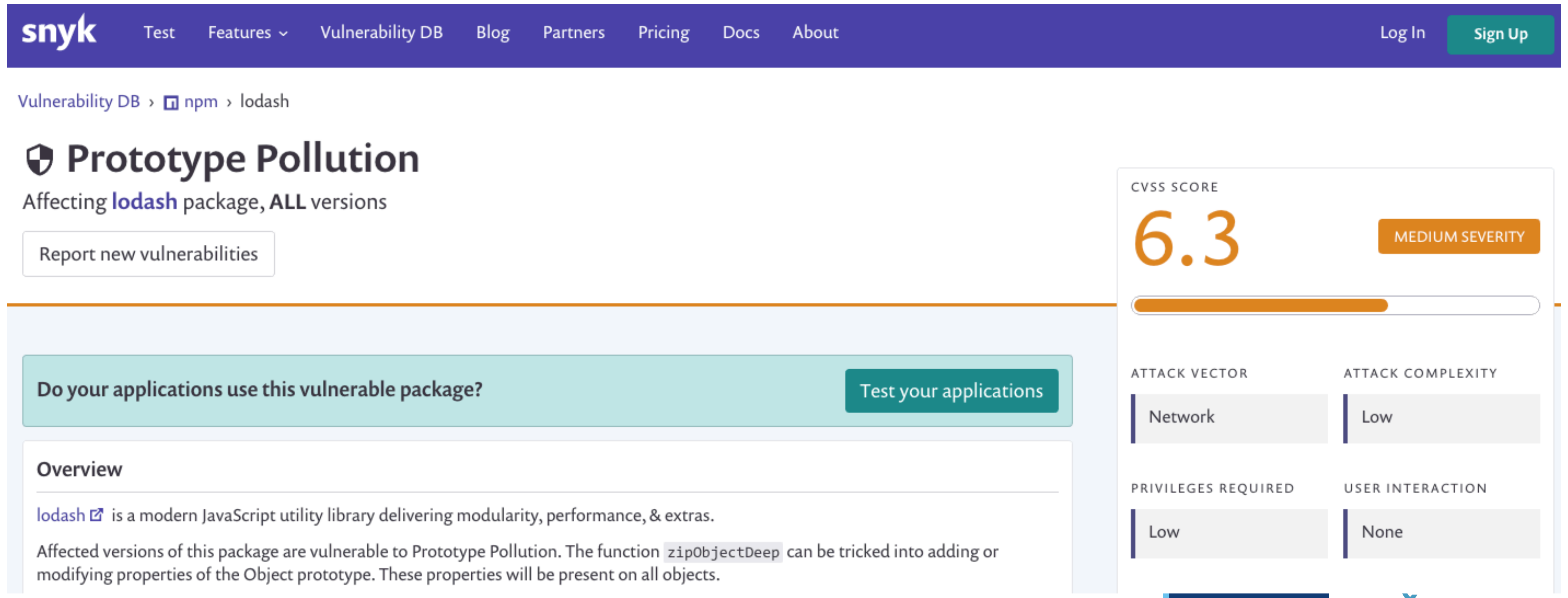
GitHub security alerts



⚠ 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	ATTACK COMPLEXITY
Network	Low

PRIVILEGES REQUIRED	USER INTERACTION
Low	None

Do your applications use this vulnerable package?

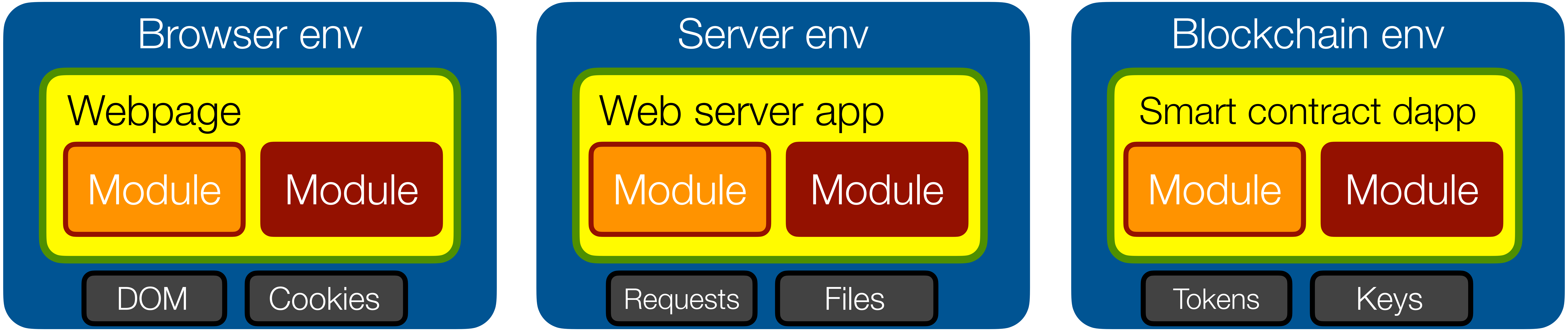
Overview

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



Browser env

Webpage

Module

Module

DOM

Cookies

Server env

Web server app

Module

Module

Requests

Files

Blockchain env

Smart contract dapp

Module

Module

Tokens

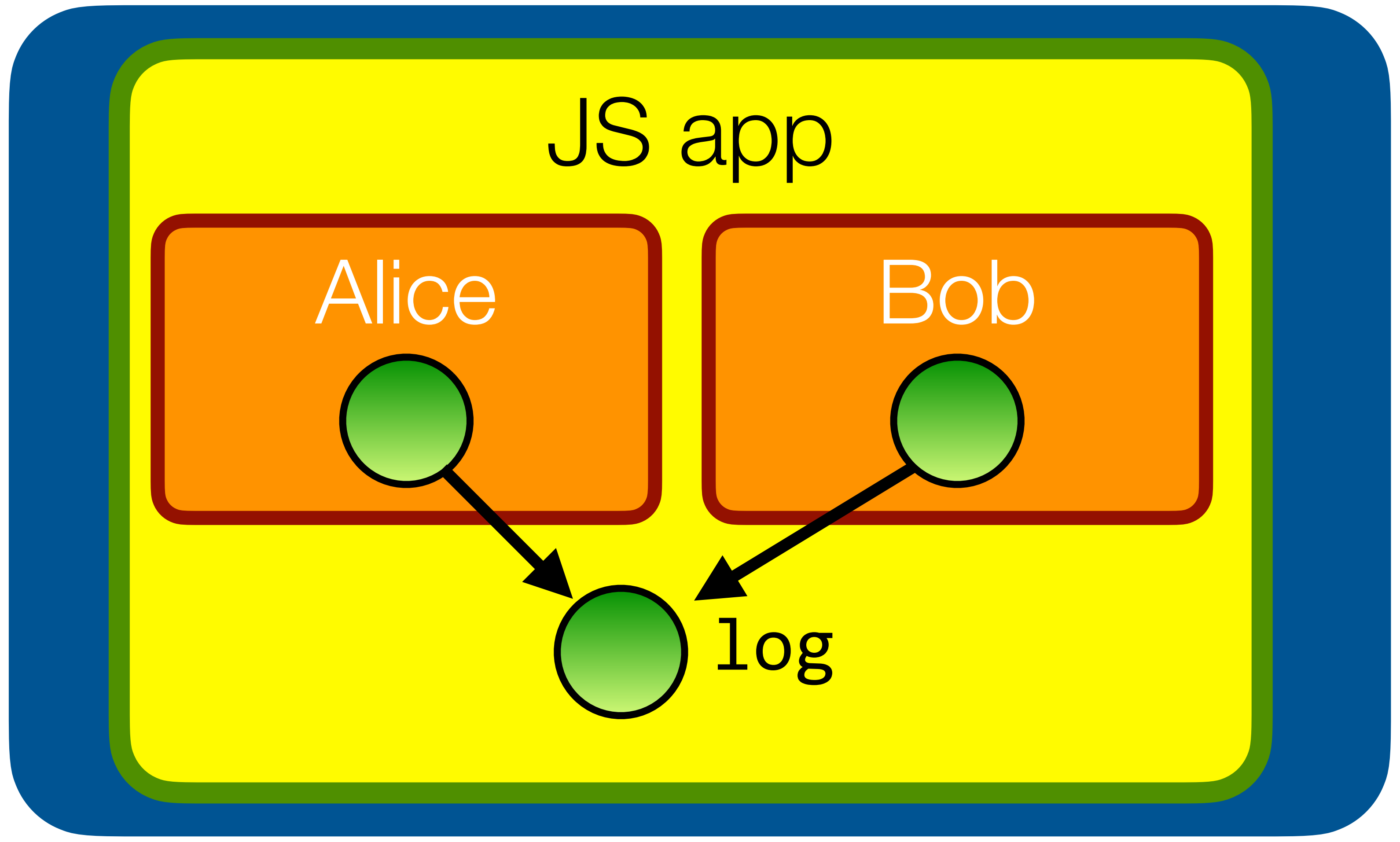
Keys

Part II
The Principle of Least Authority, by example

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.

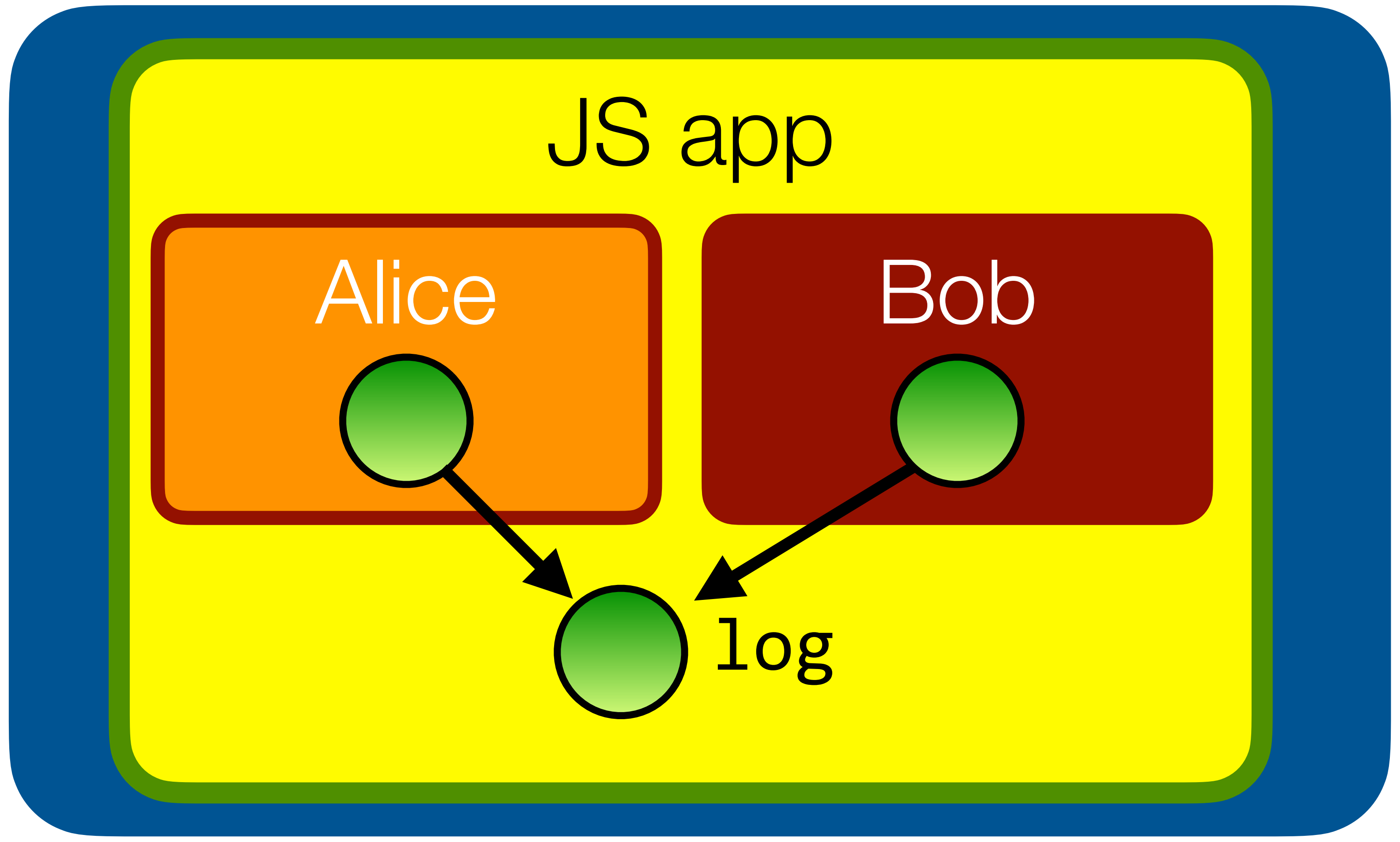
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

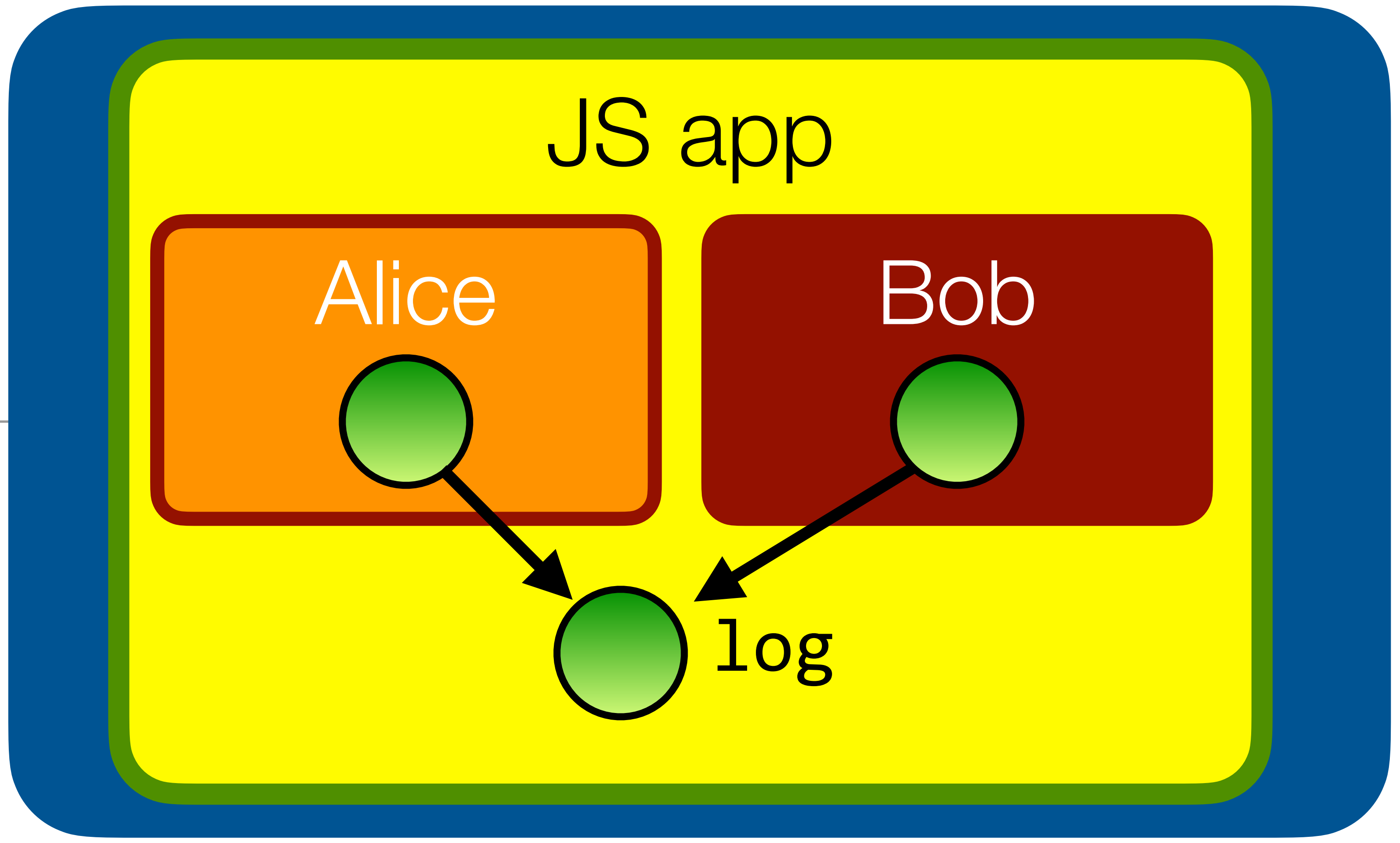
If Bob goes rogue, what could go wrong?

```
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!

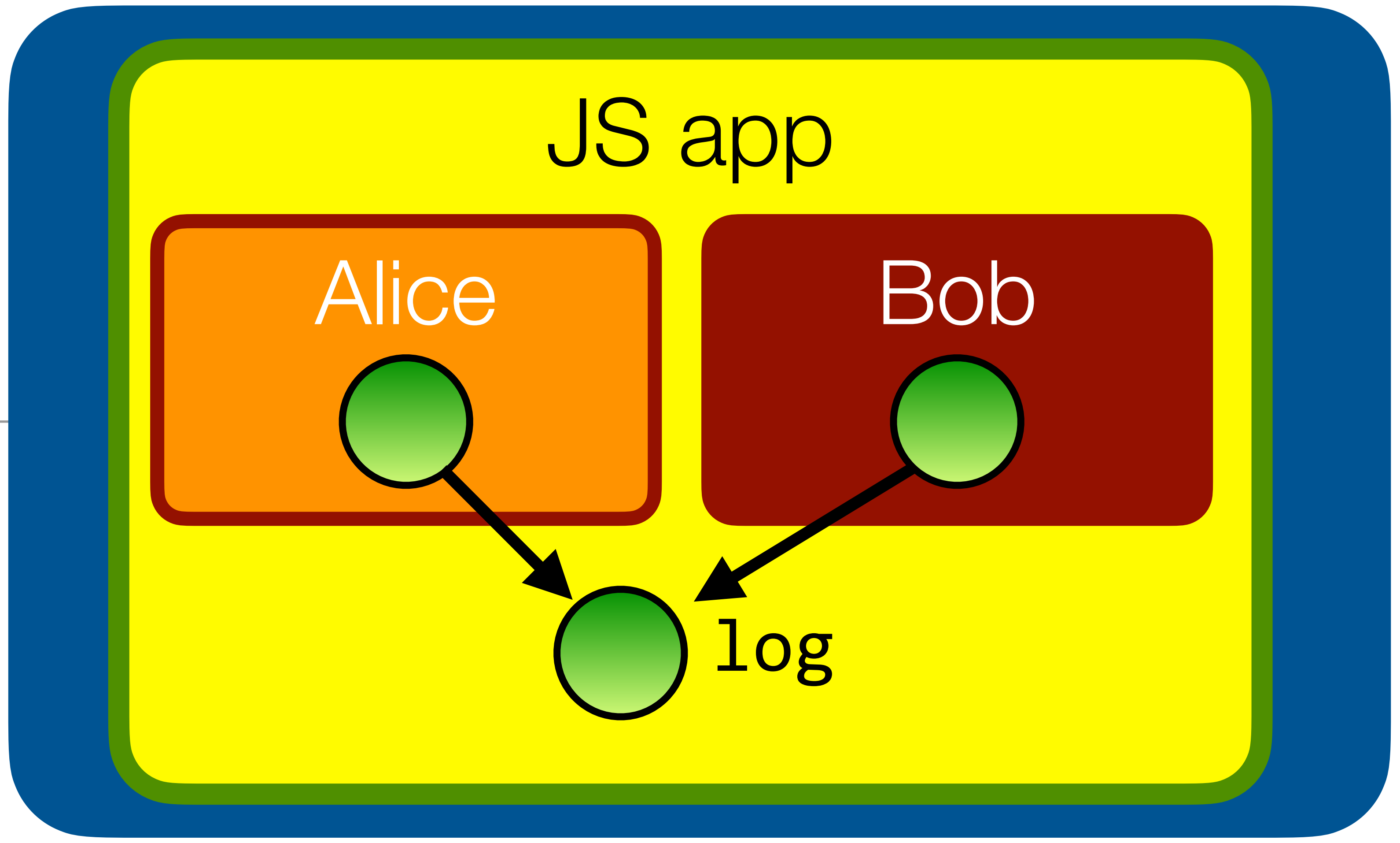
If Bob goes rogue, what could go wrong?



```
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

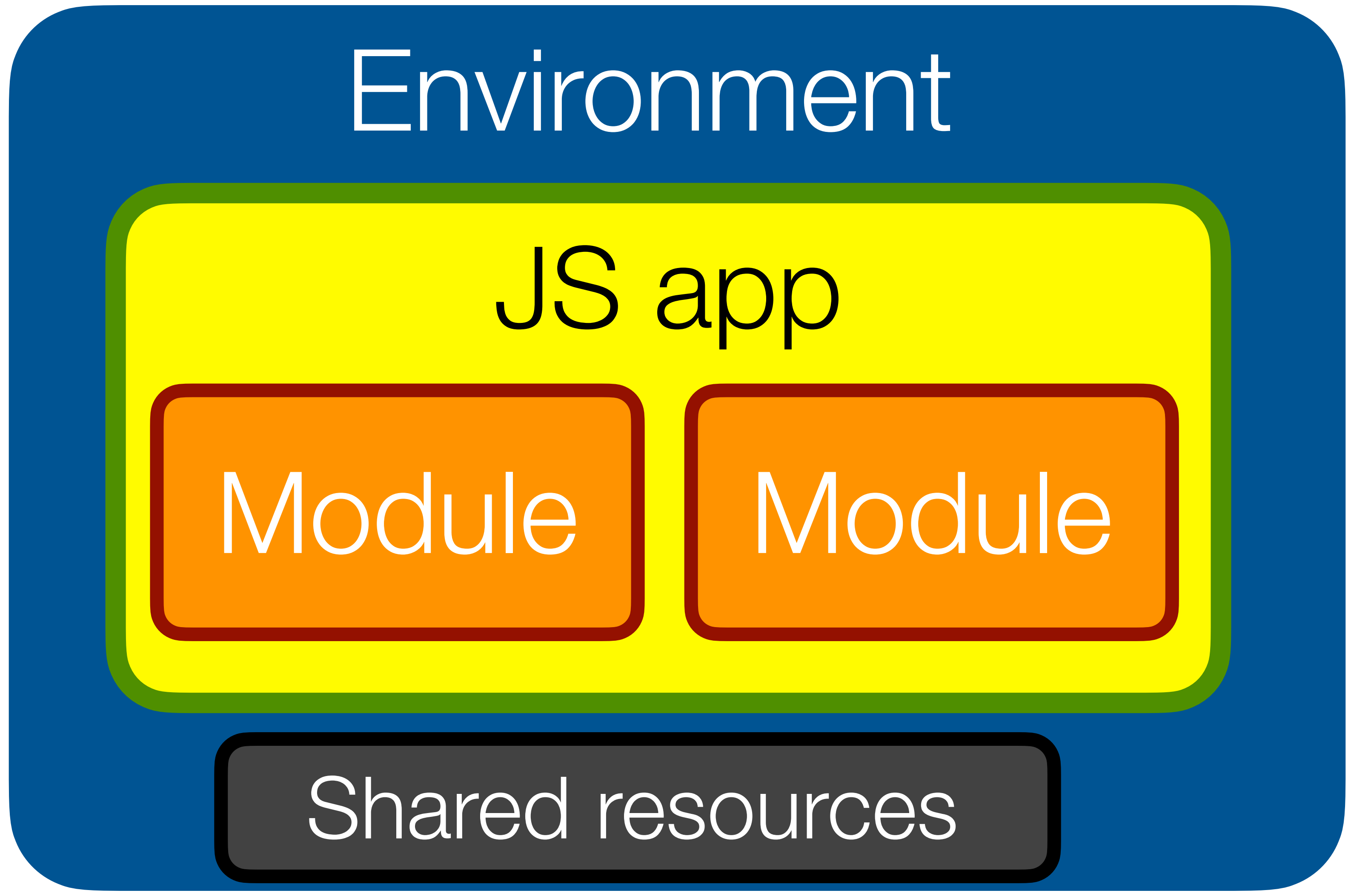
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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”);
}
```

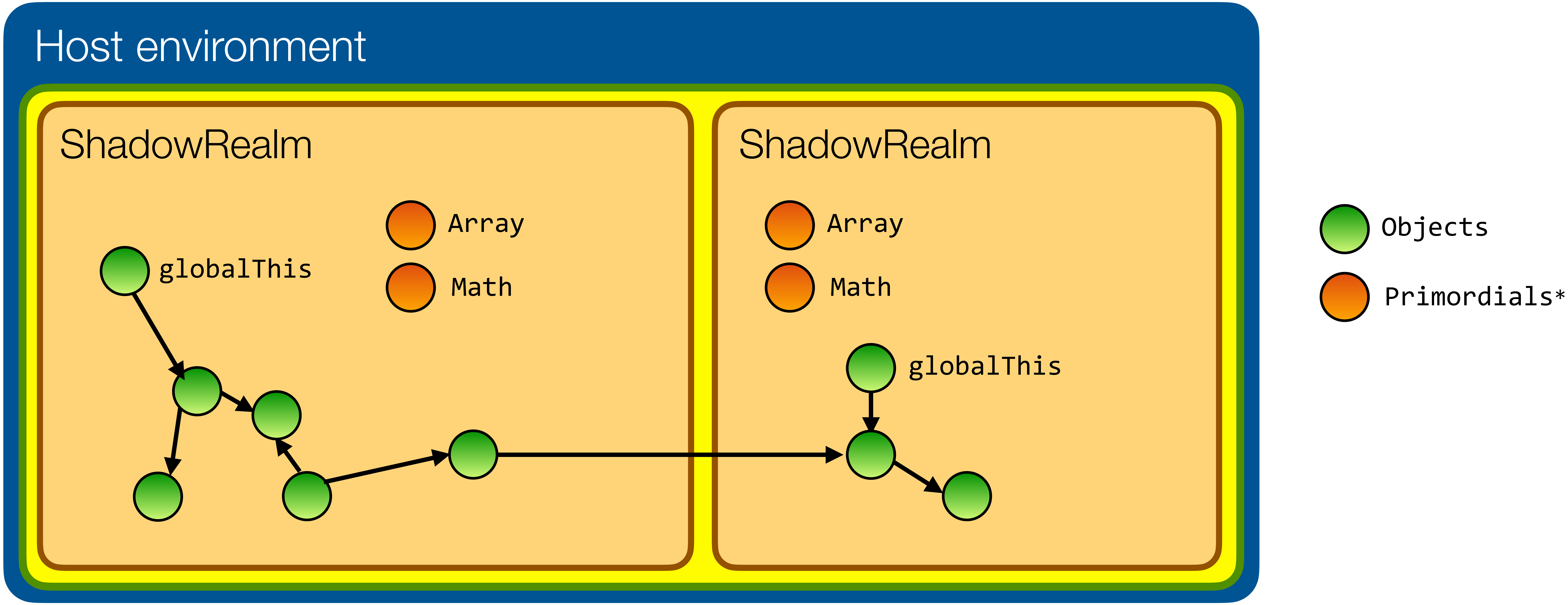
Prerequisite: isolating JavaScript modules

- Today: JavaScript offers no standardized isolation mechanisms
- Lots of environment-specific isolation mechanisms, but non-portable and ill-defined:
  - **Web Workers**: forced async communication, no shared memory
  - **iframes**: mutable primordials, “identity discontinuity”
  - **nodejs** `vm` **module**: same issues



ShadowRealms (TC39 Stage 3 proposal)

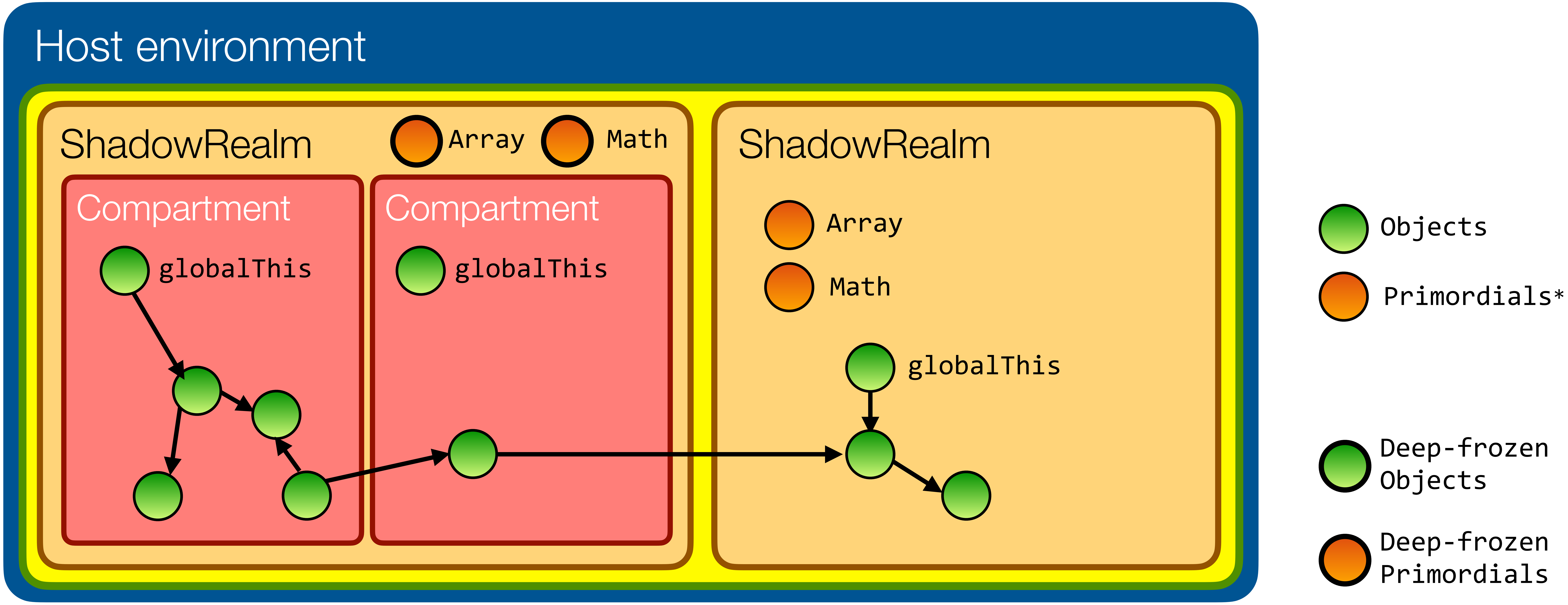
Intuitions: “iframe without DOM”, “principled version of node’s `vm` module”



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

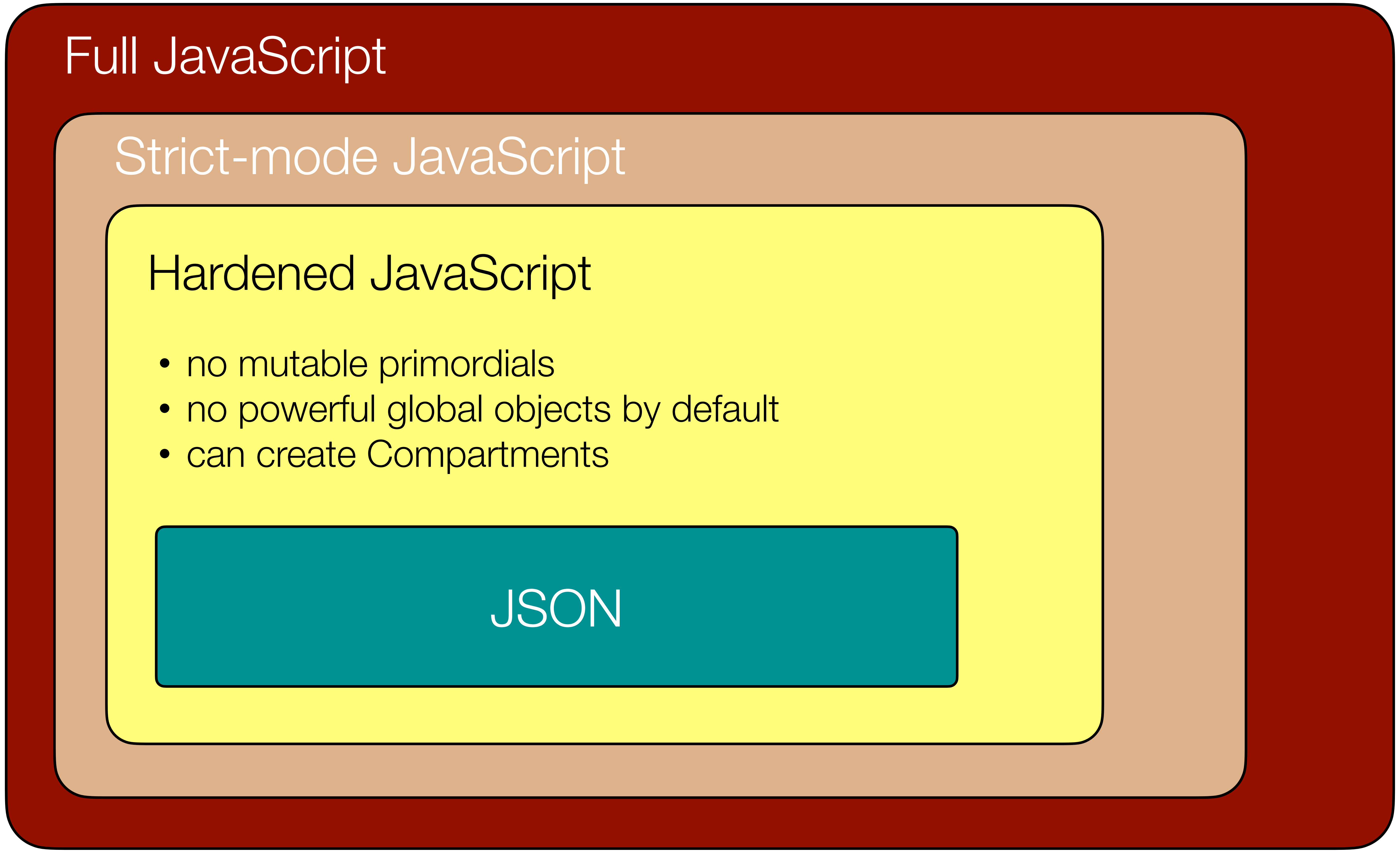
Compartments (TC39 Stage 1 proposal)

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



Full JavaScript

Strict-mode JavaScript

Hardened JavaScript

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

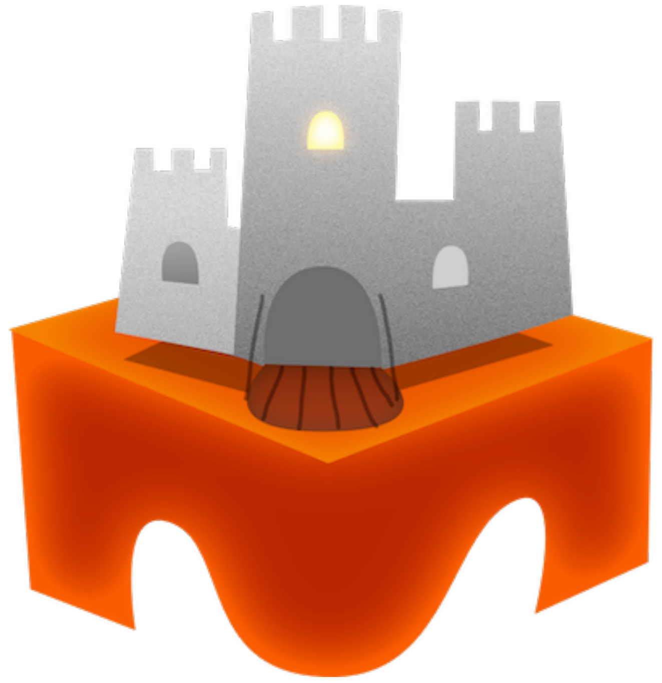
JSON

(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.

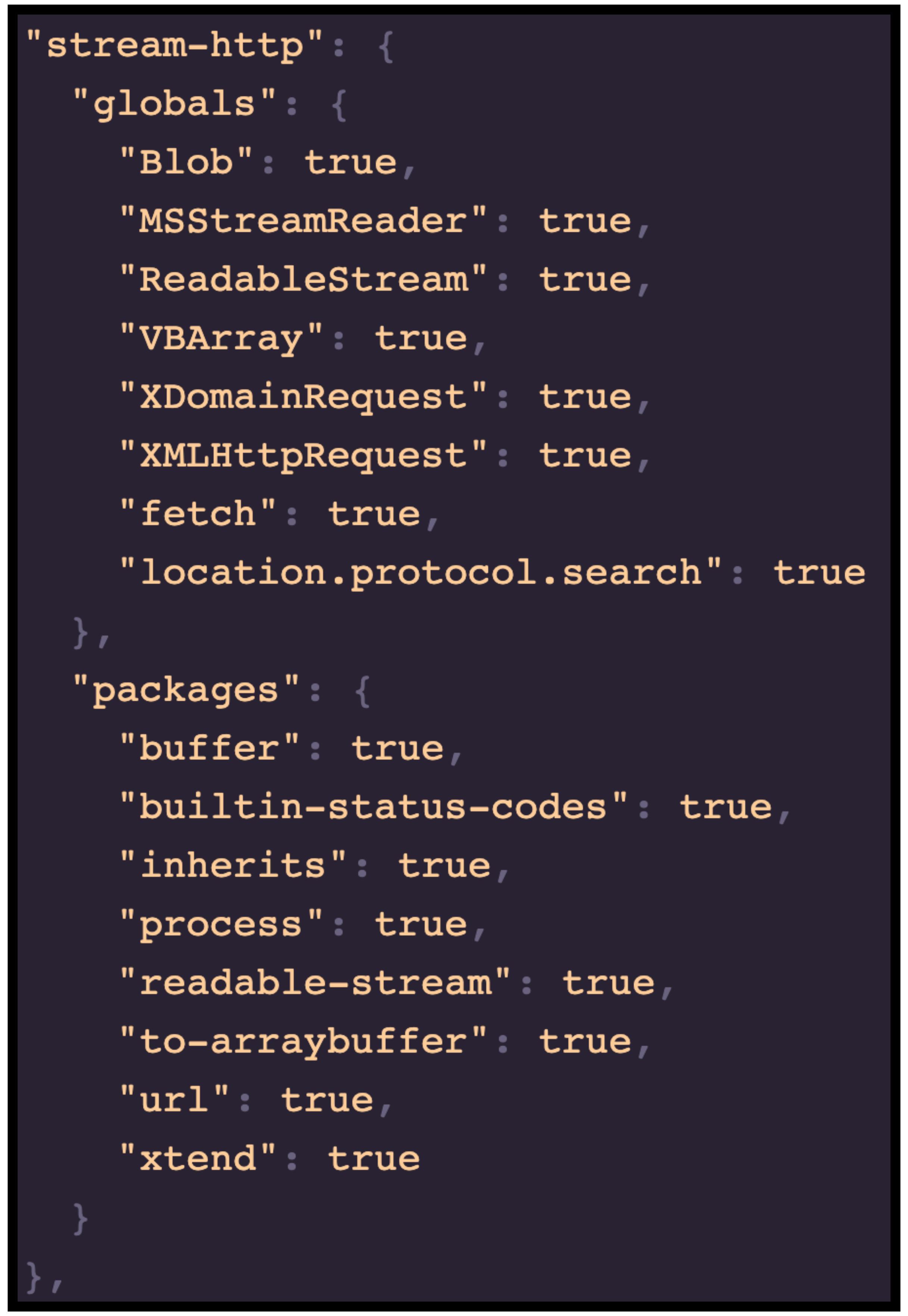
LavaMoat

- 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



https://github.com/LavaMoat/lavamoat

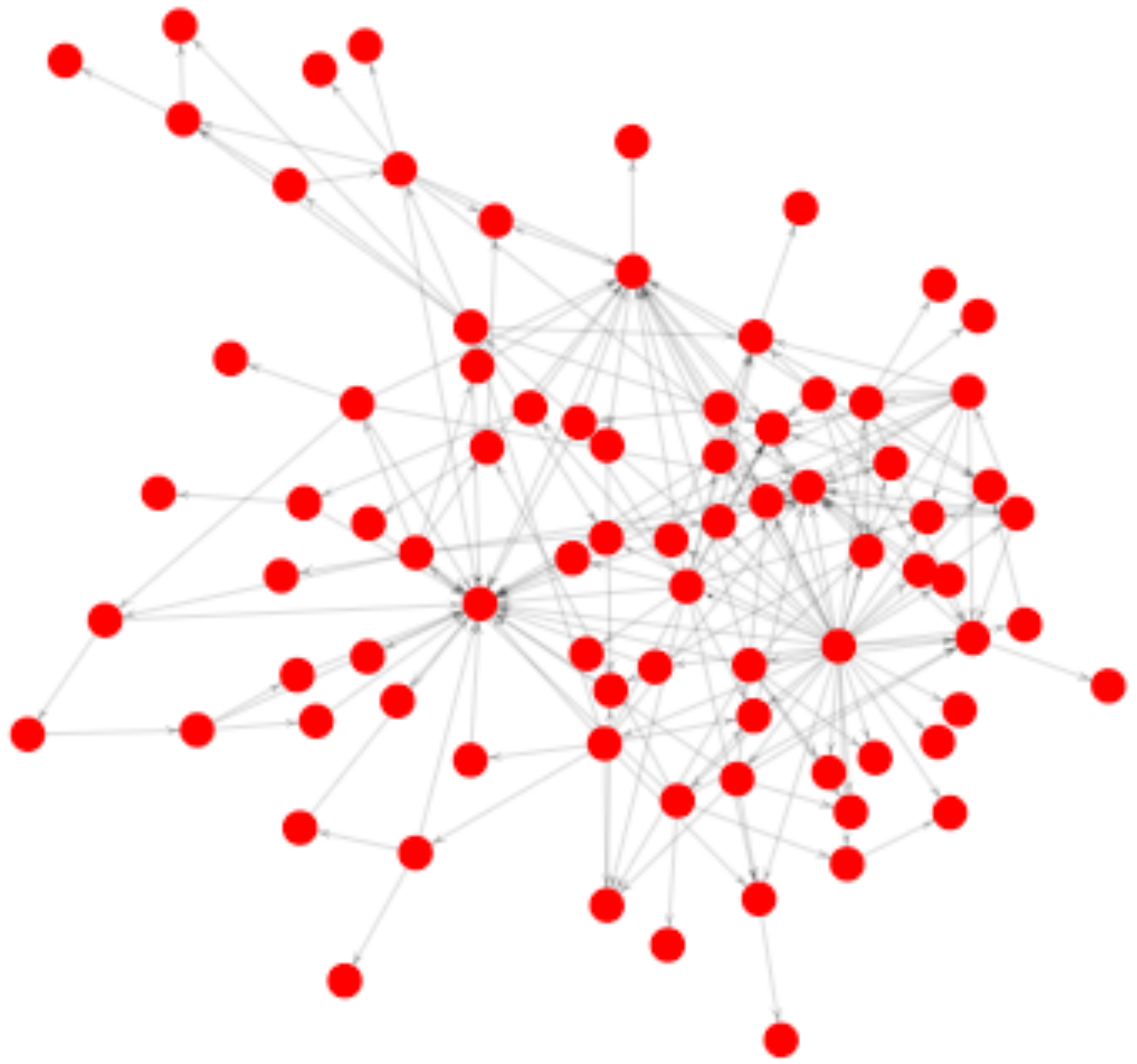
METAMASK



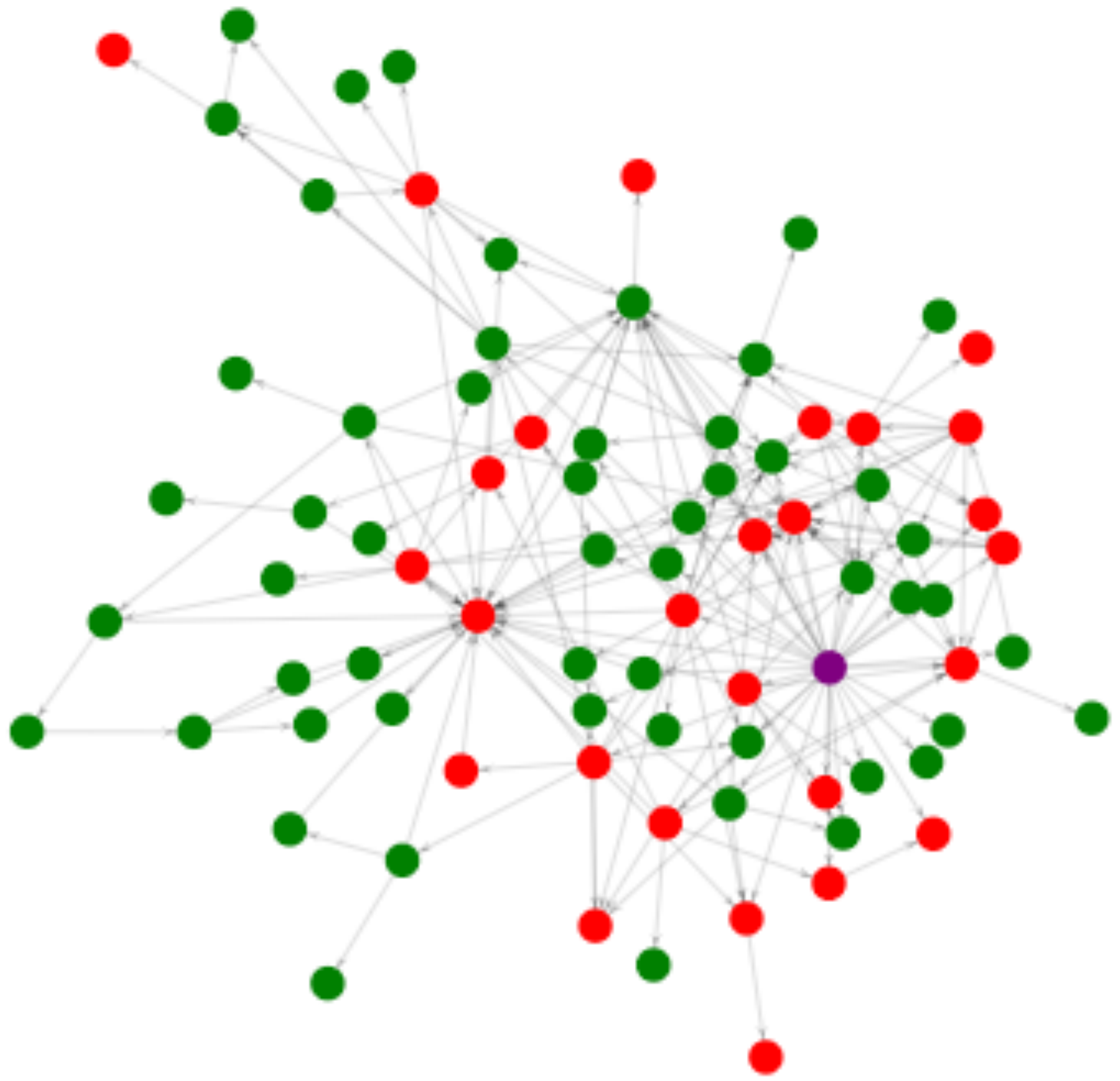
```
"stream-http": {
  "globals": {
    "Blob": true,
    "MSStreamReader": true,
    "ReadableStream": true,
    "VBArray": true,
    "XDomainRequest": true,
    "XMLHttpRequest": true,
    "fetch": true,
    "location.protocol.search": true
  },
  "packages": {
    "buffer": true,
    "builtin-status-codes": 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



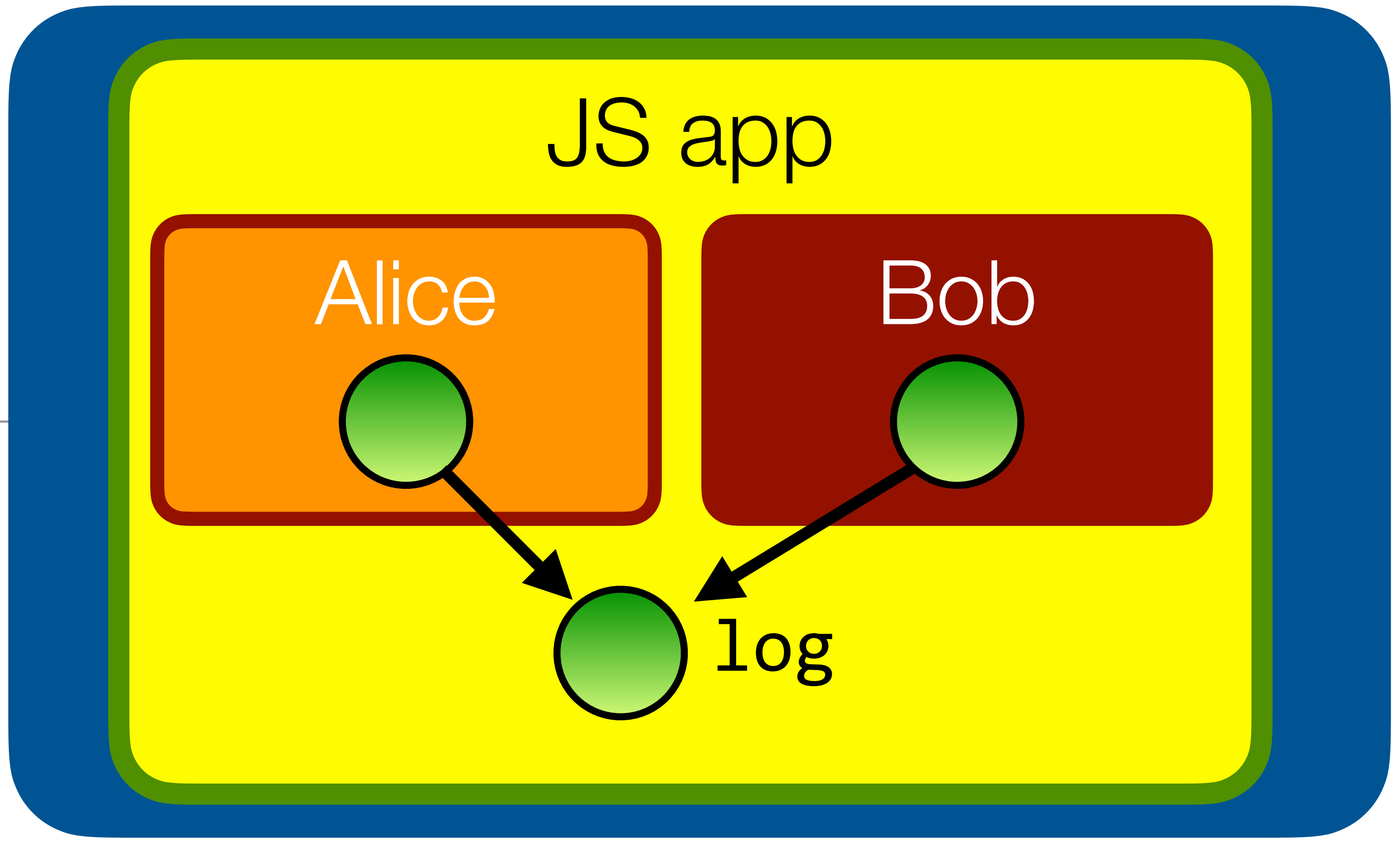
Exposure to package dependencies
with LavaMoat sandboxing



https://github.com/LavaMoat/lavamoat

Back to our example

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



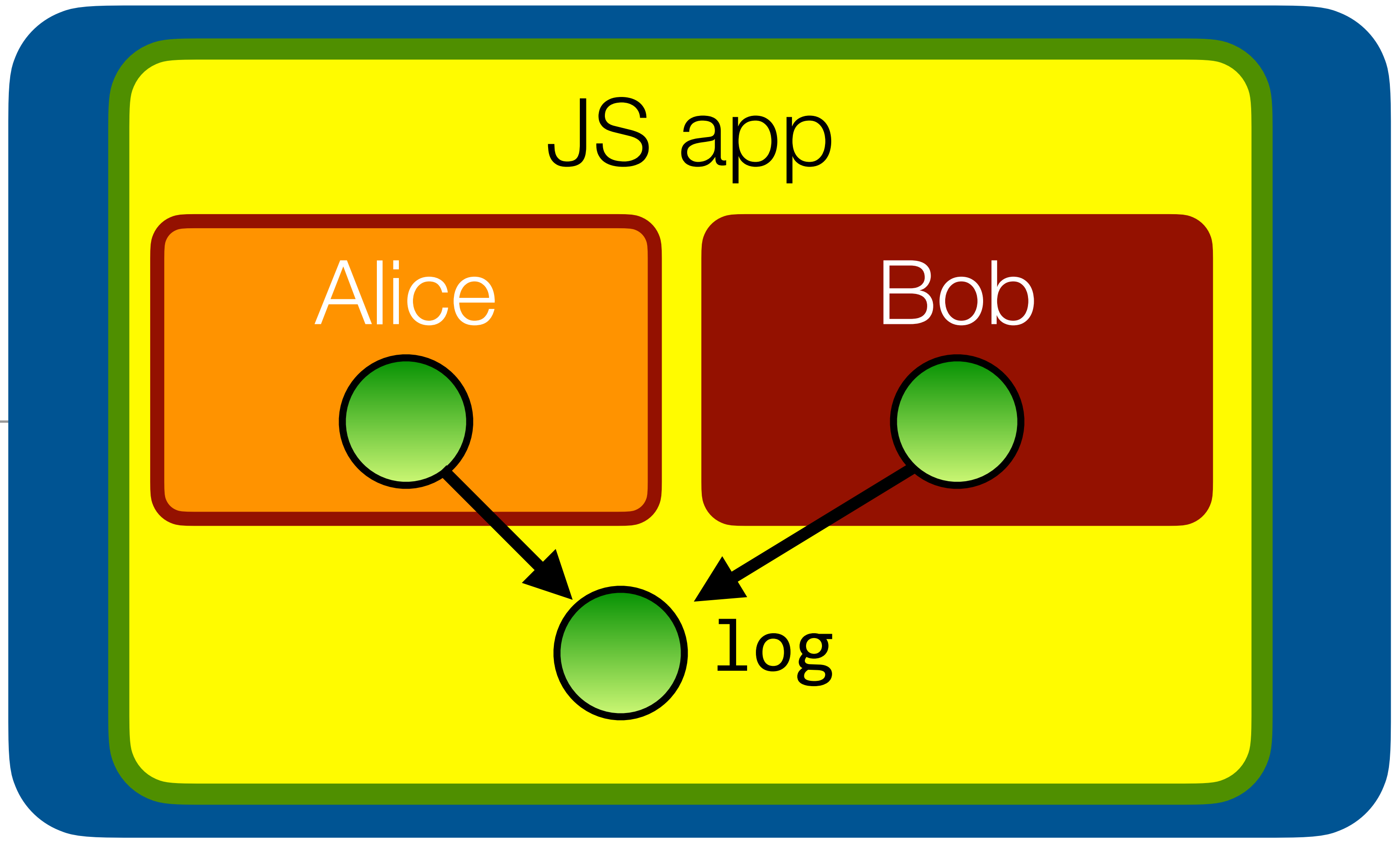
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.

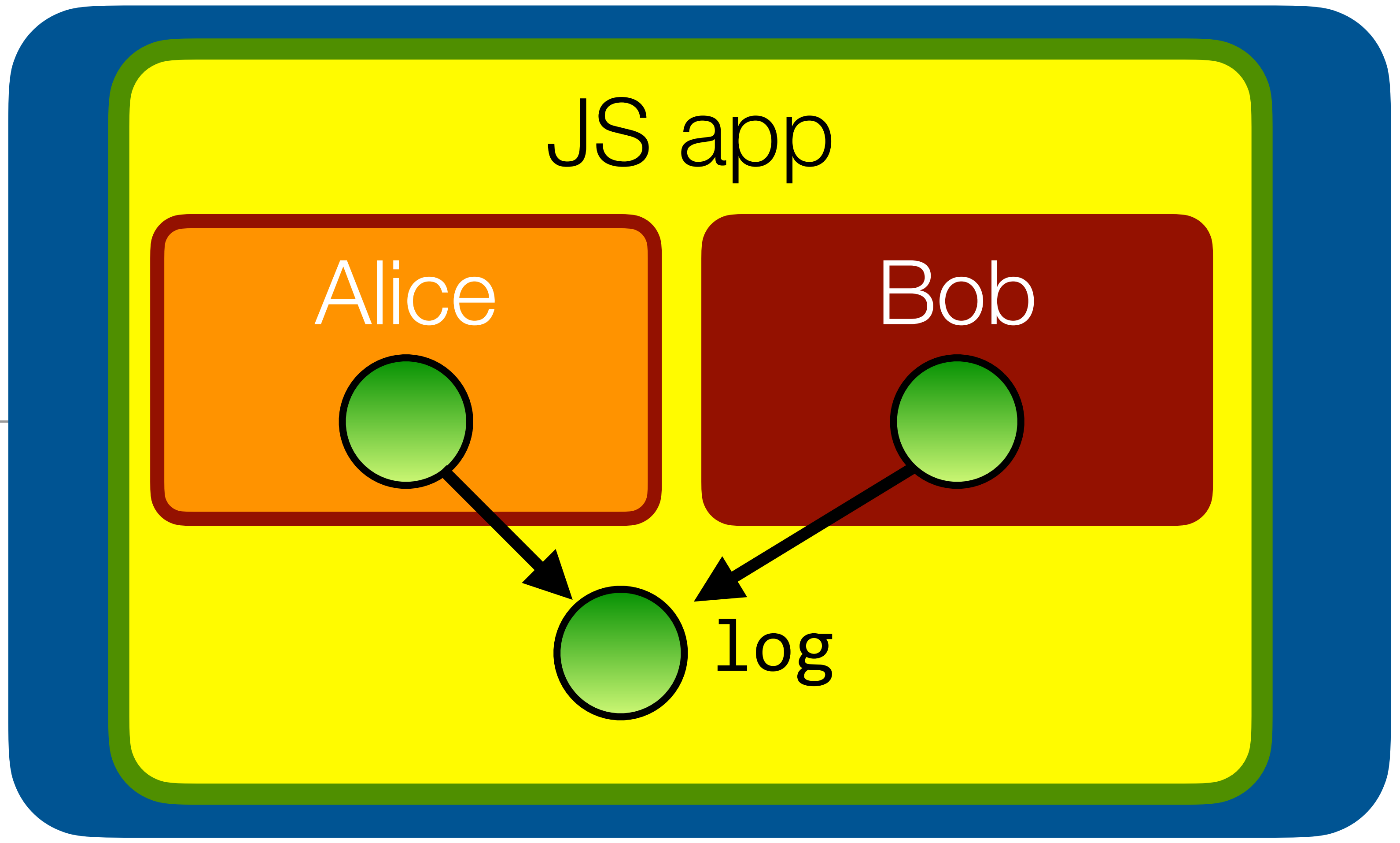


```
import * as alice from "alice.js";
import * as bob from “bob.js";

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**

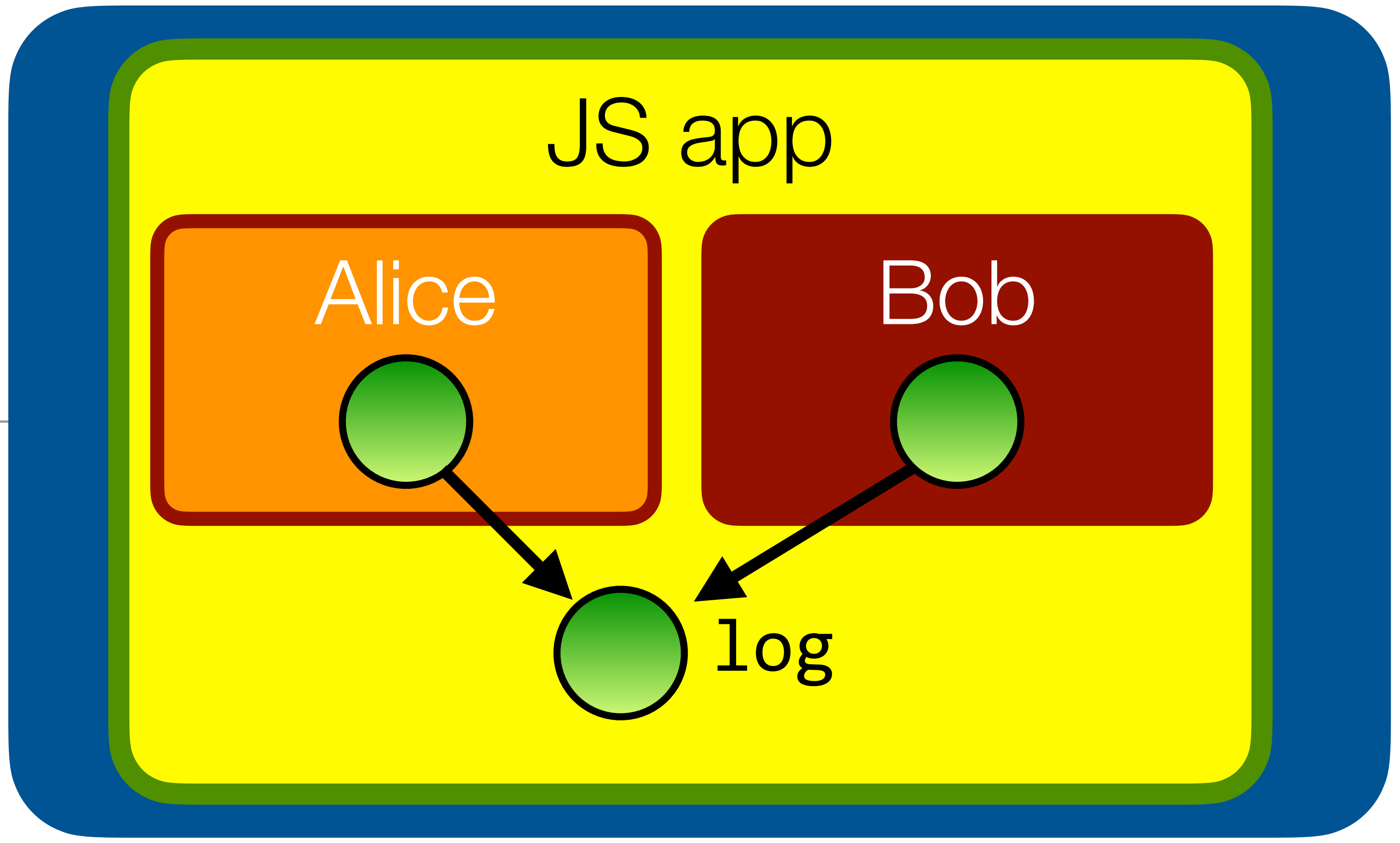


Object.freeze makes property bindings (not their values) immutable

```
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.

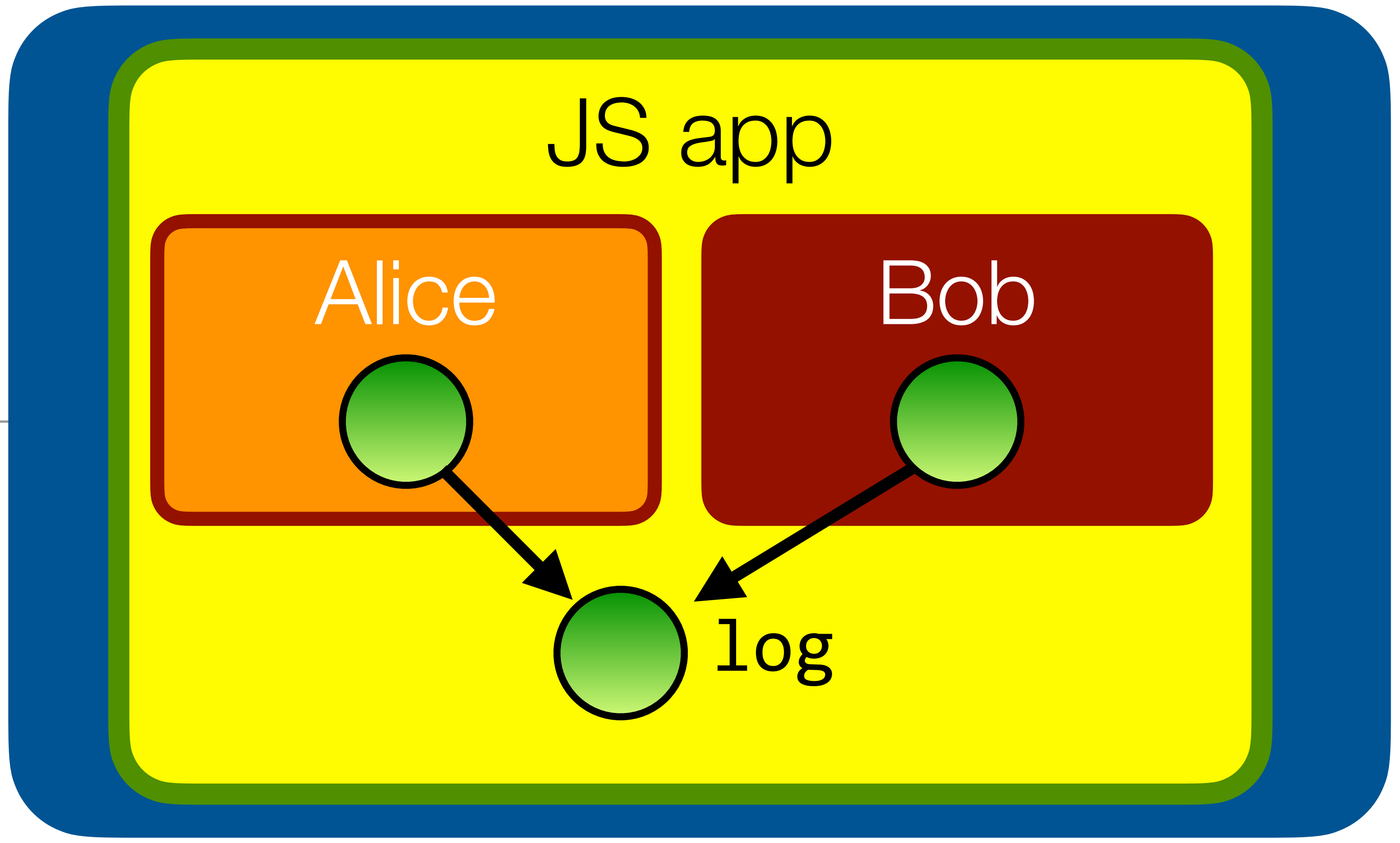


Functions are mutable too. Freeze doesn’t recursively freeze the object’s functions.

```
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



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

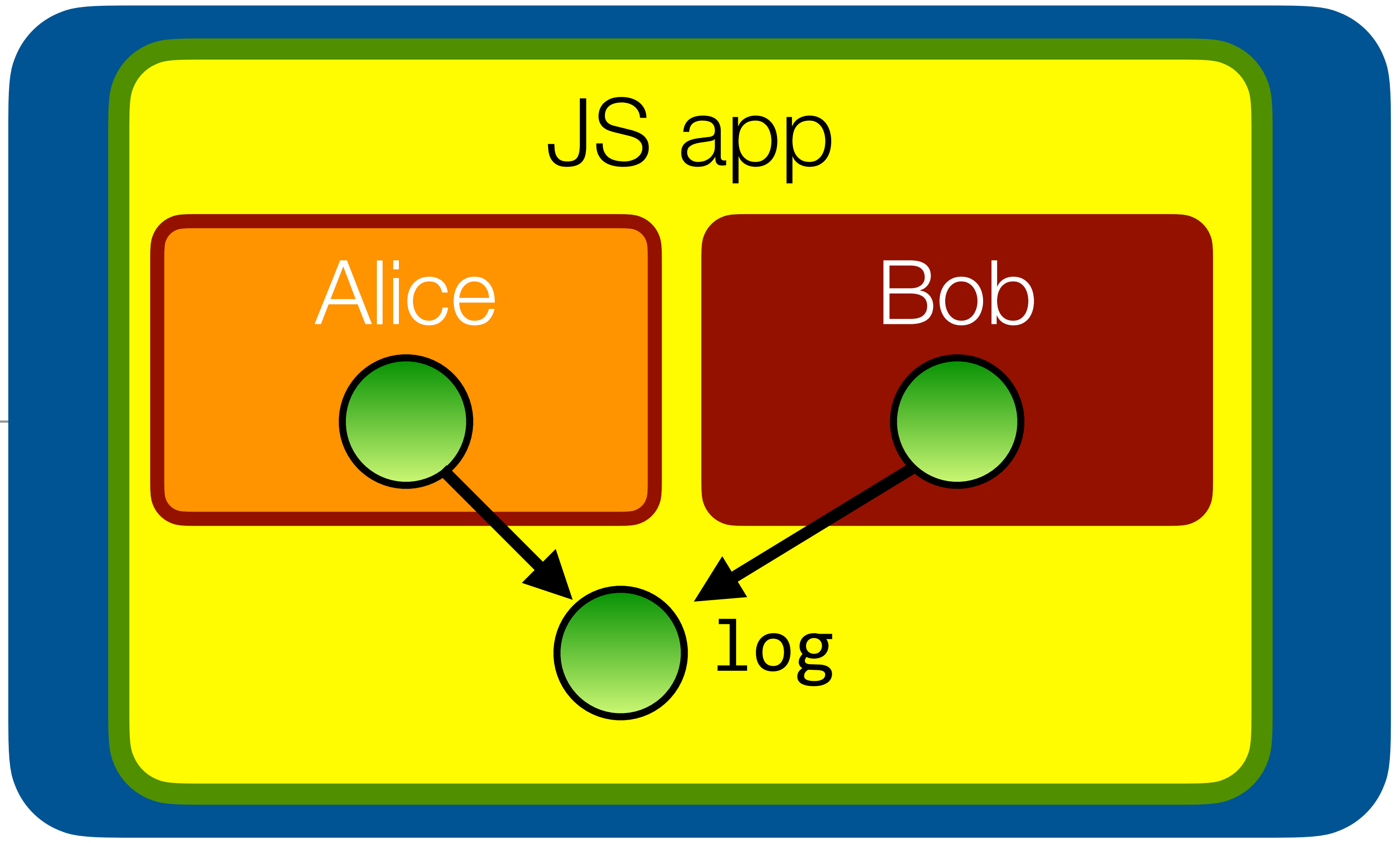
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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” };
```

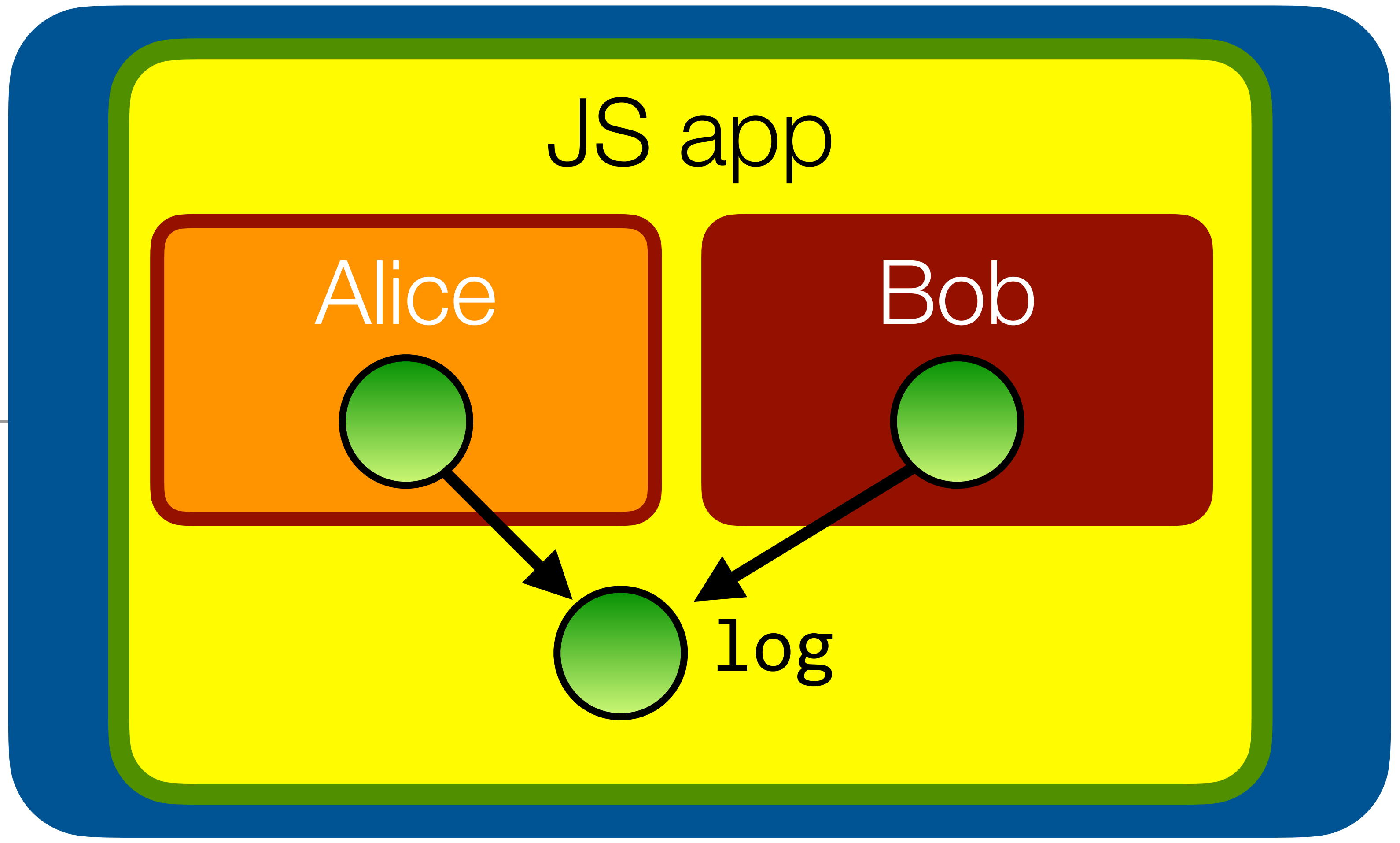
Two down, two to go



```
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);
```

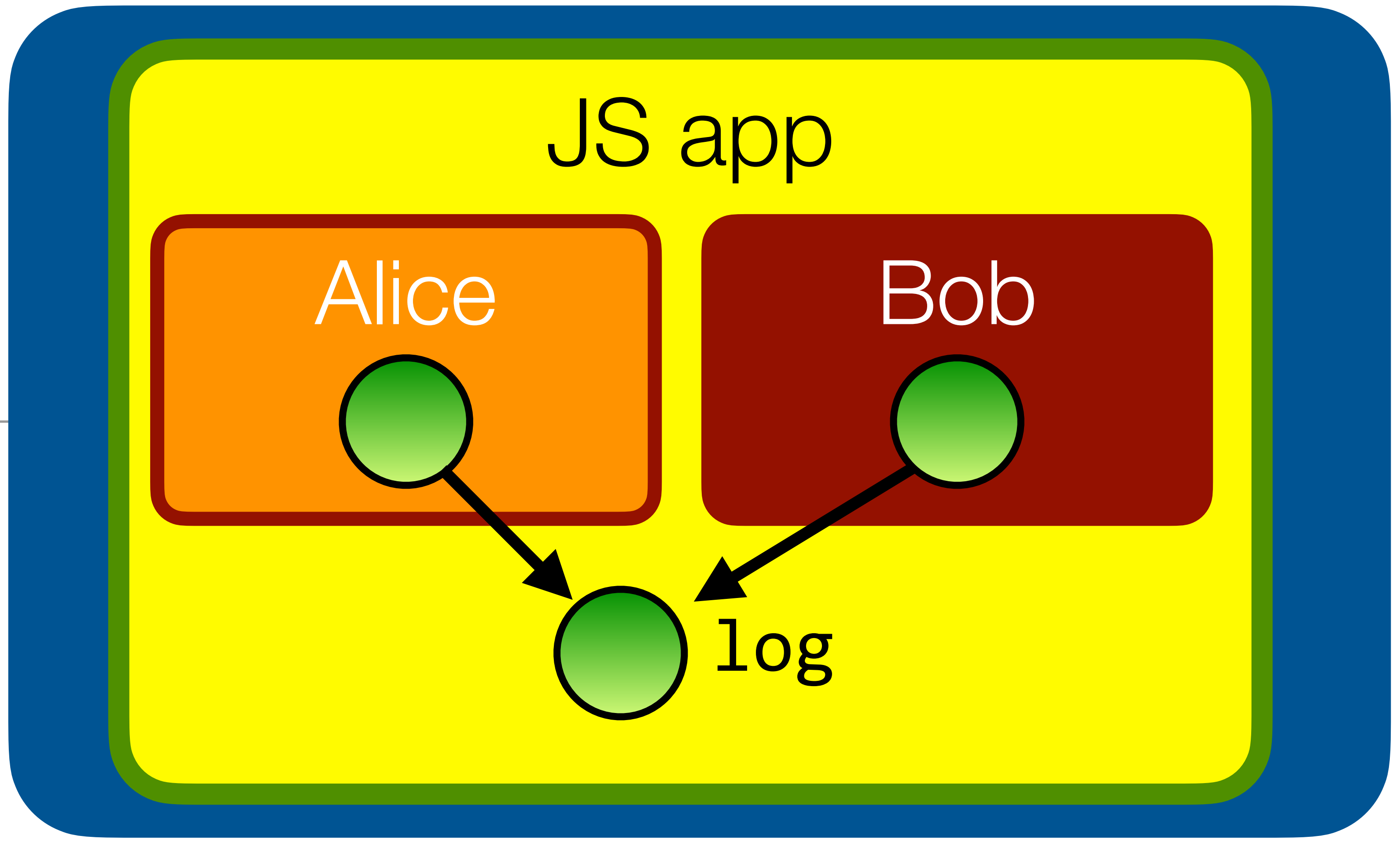
```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)
```

```
// 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 “persistent” data structure (see immutable-js.com)

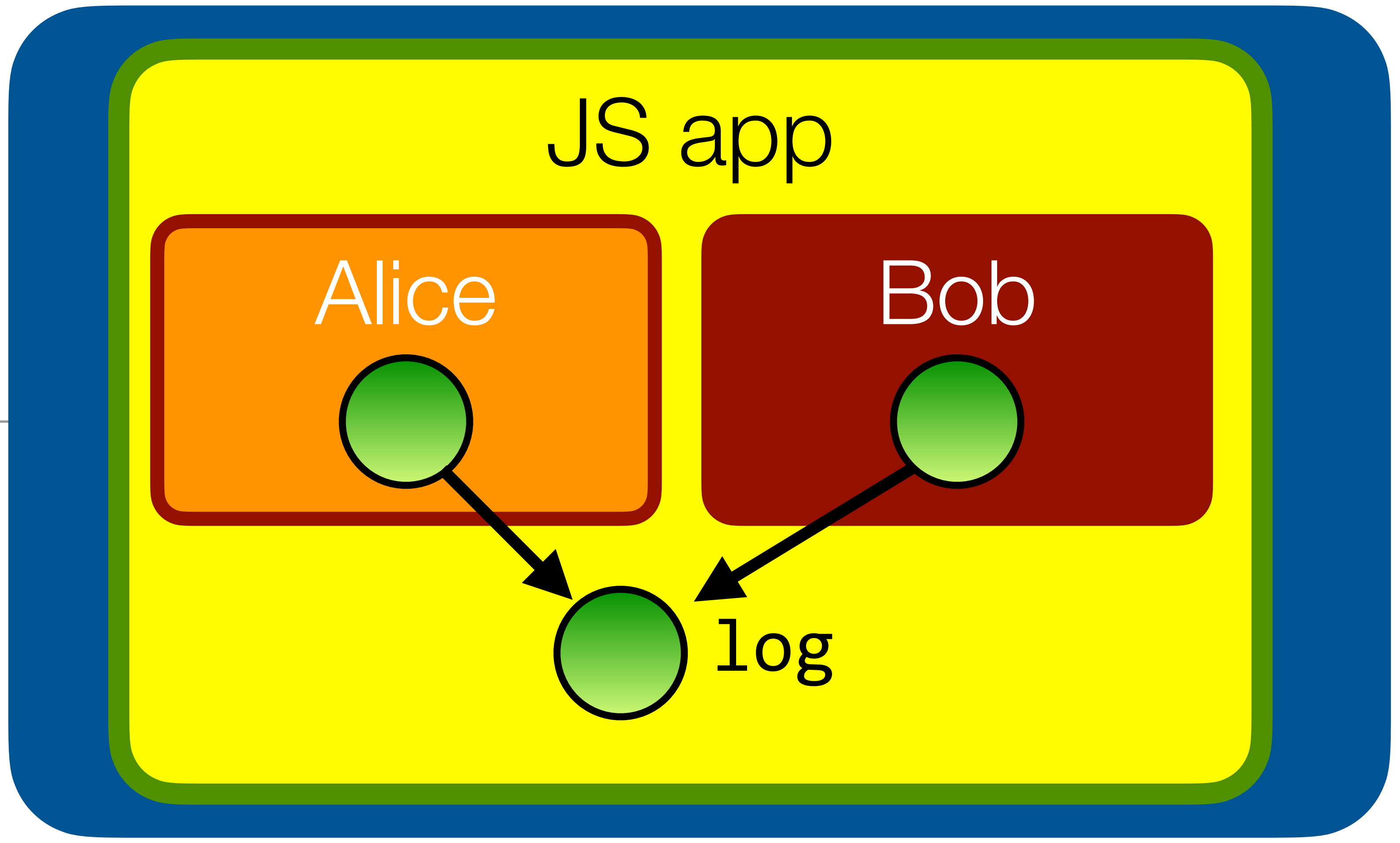
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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” };~~

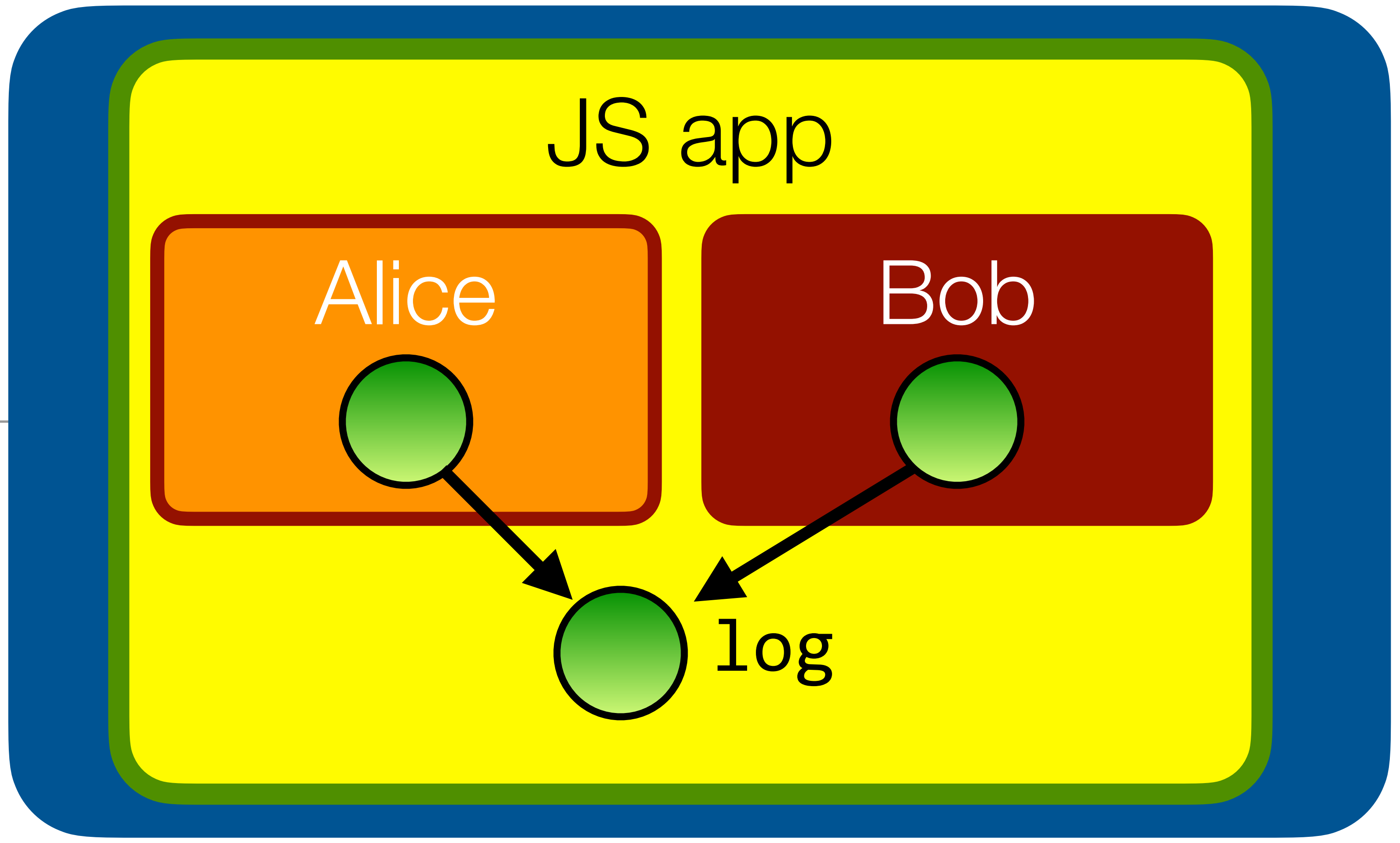
Three down, one to go



```
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



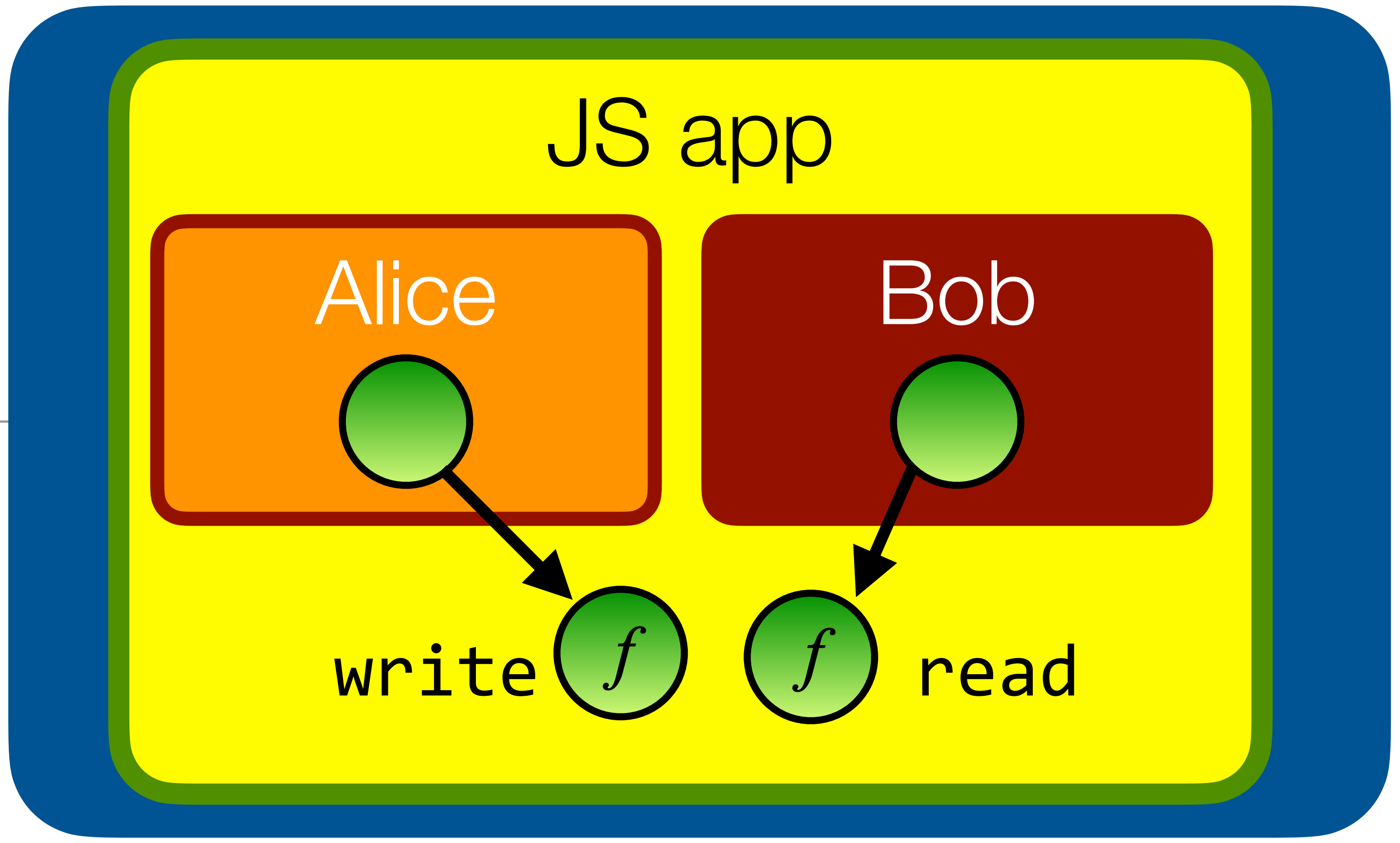
- Recall: we would like Alice to only write to the log, and Bob to only read from the log.
- Bob receives too much authority. How to limit?

```
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. Can you spot the bug?

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

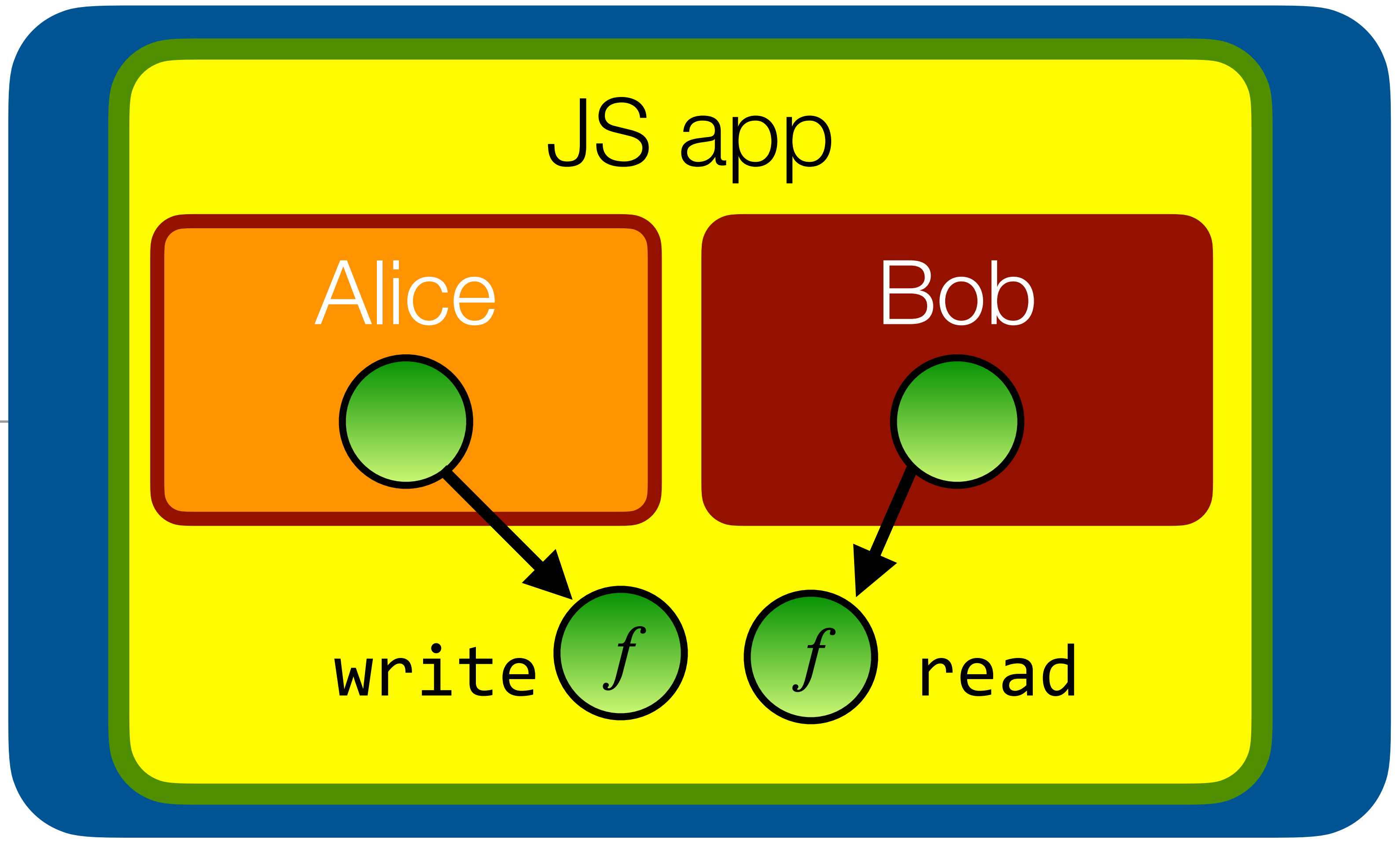
```
// in bob.js
// Bob can just write to the log
log.write(“I’m polluting the log”)
```

~~// 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” };~~

Pass only the authority that Bob needs.

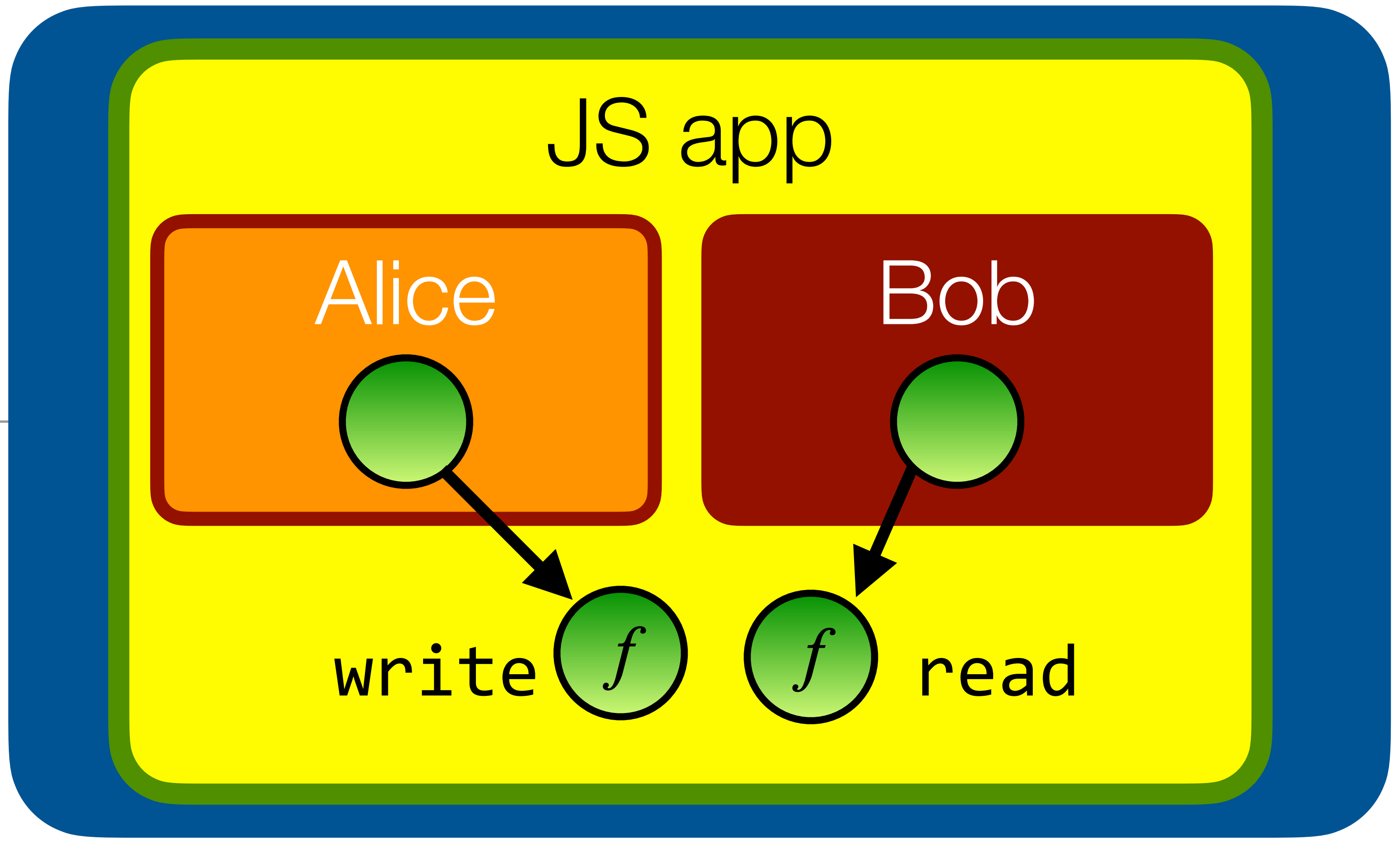


To avoid, must pass “bound” functions

```
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 = harden(new Log());
alice(log.write.bind(log));
bob(log.read.bind(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” };~~
```

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

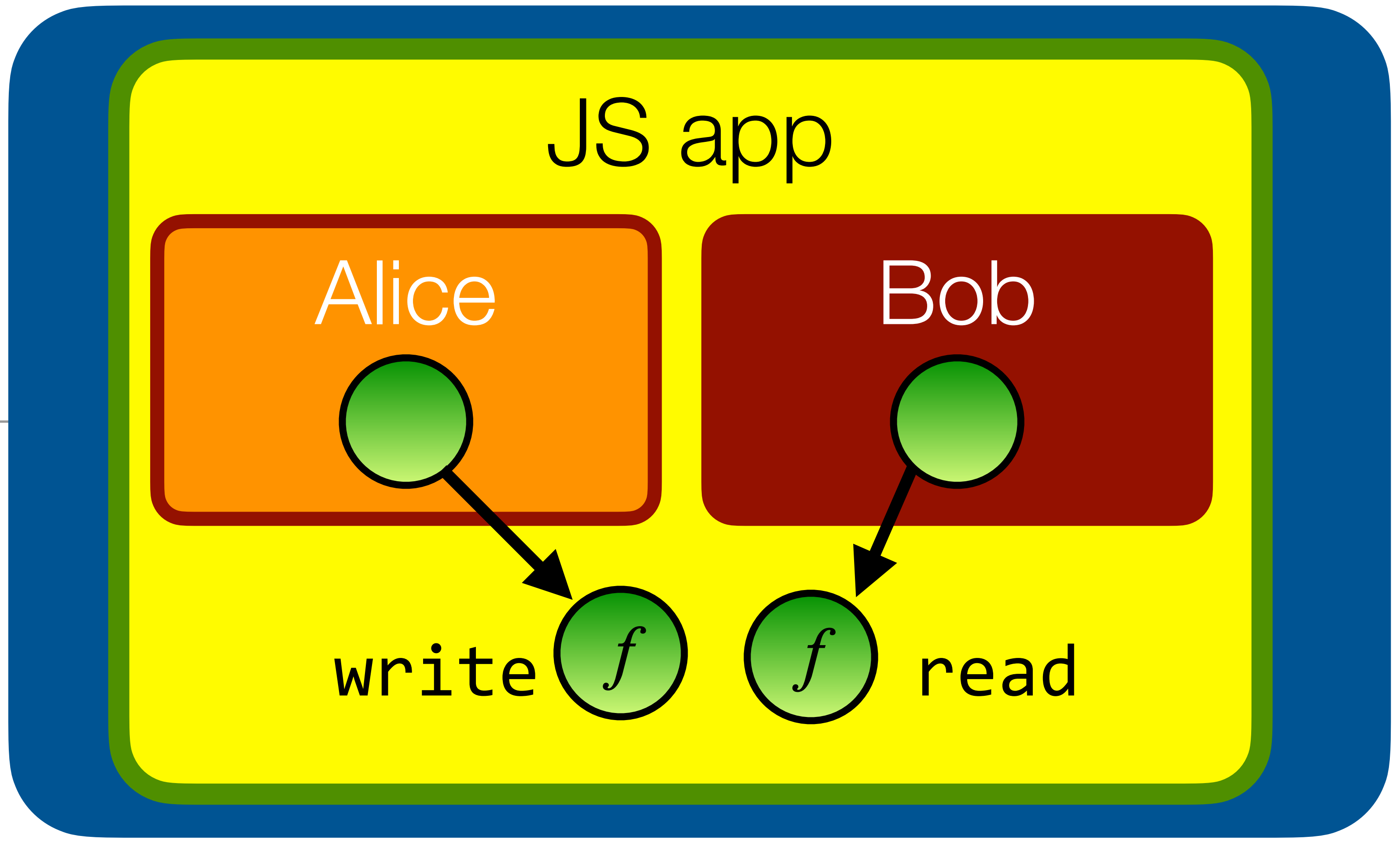


```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

let log = harden(new Log());
alice(log.write.bind(log));
bob(log.read.bind(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” };~~
```

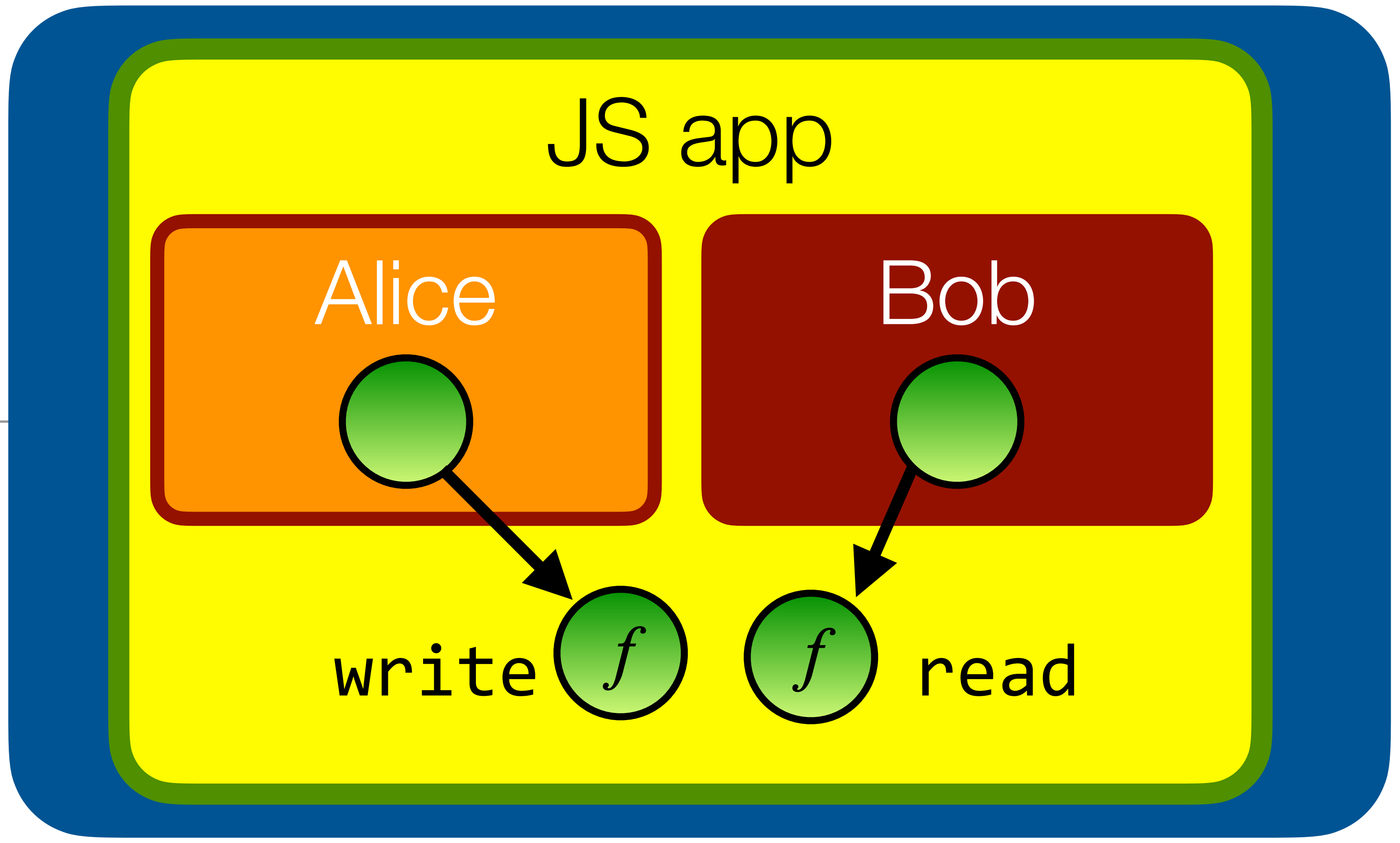
Is there a better way to write this code?

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 = harden(new Log());
alice(log.write.bind(log));
bob(log.read.bind(log));
```

Use the **Function as Object** pattern



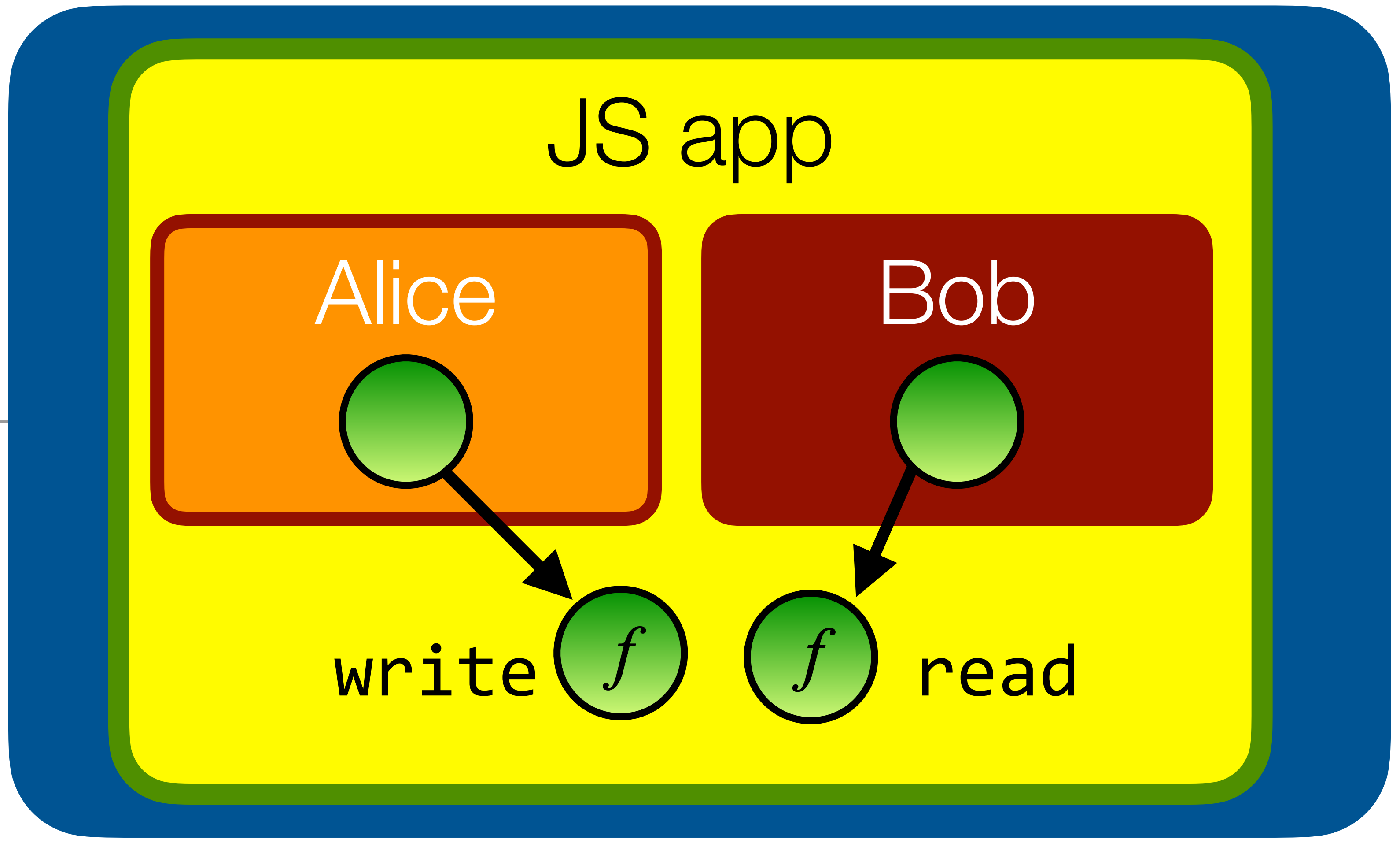
- A record of closures hiding state is a fine representation of an object of methods hiding instance vars
- Pattern long advocated by Doug Crockford instead of using classes or prototypes

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

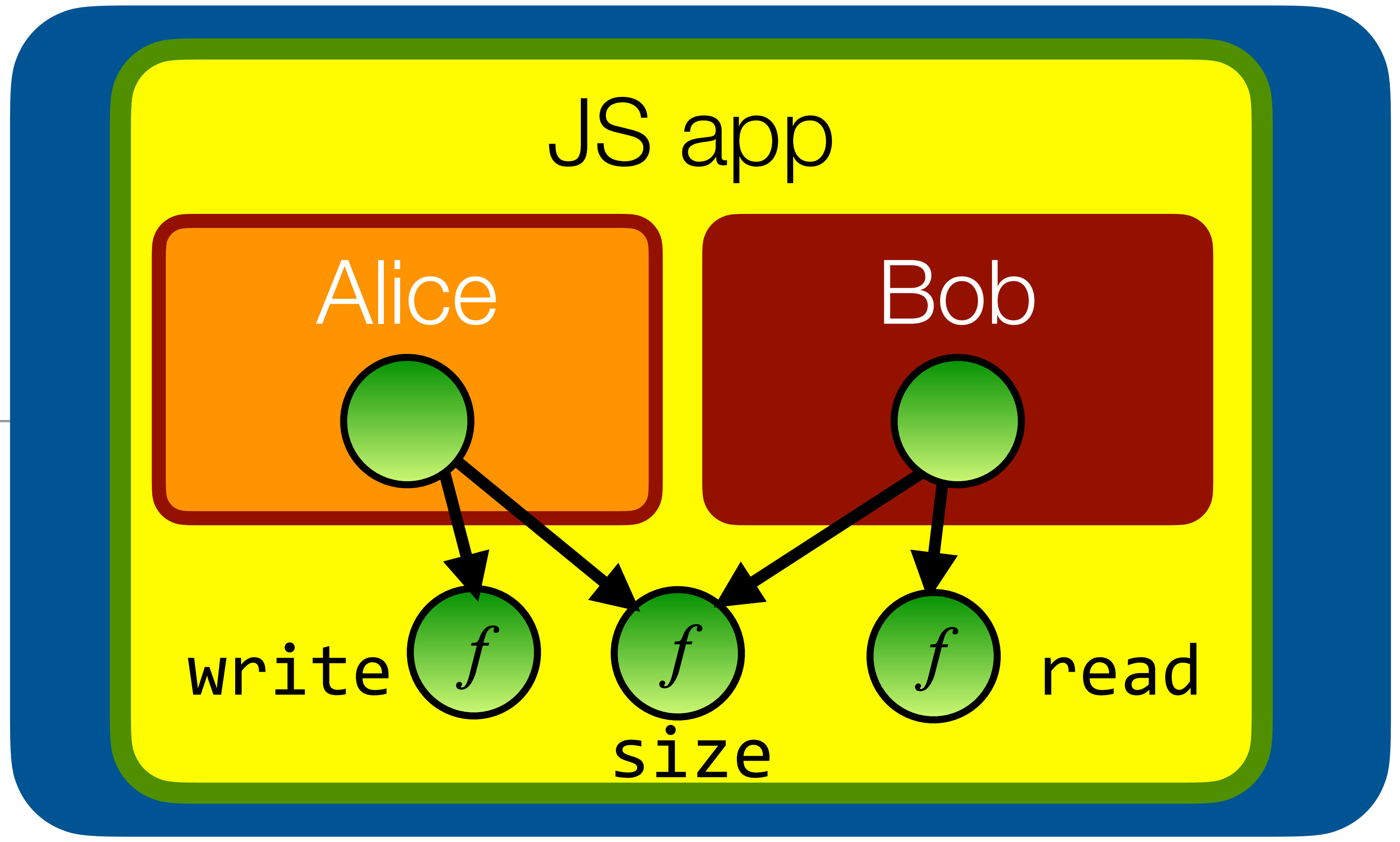
(See also https://martinfowler.com/bliki/FunctionAsObject.html)

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?

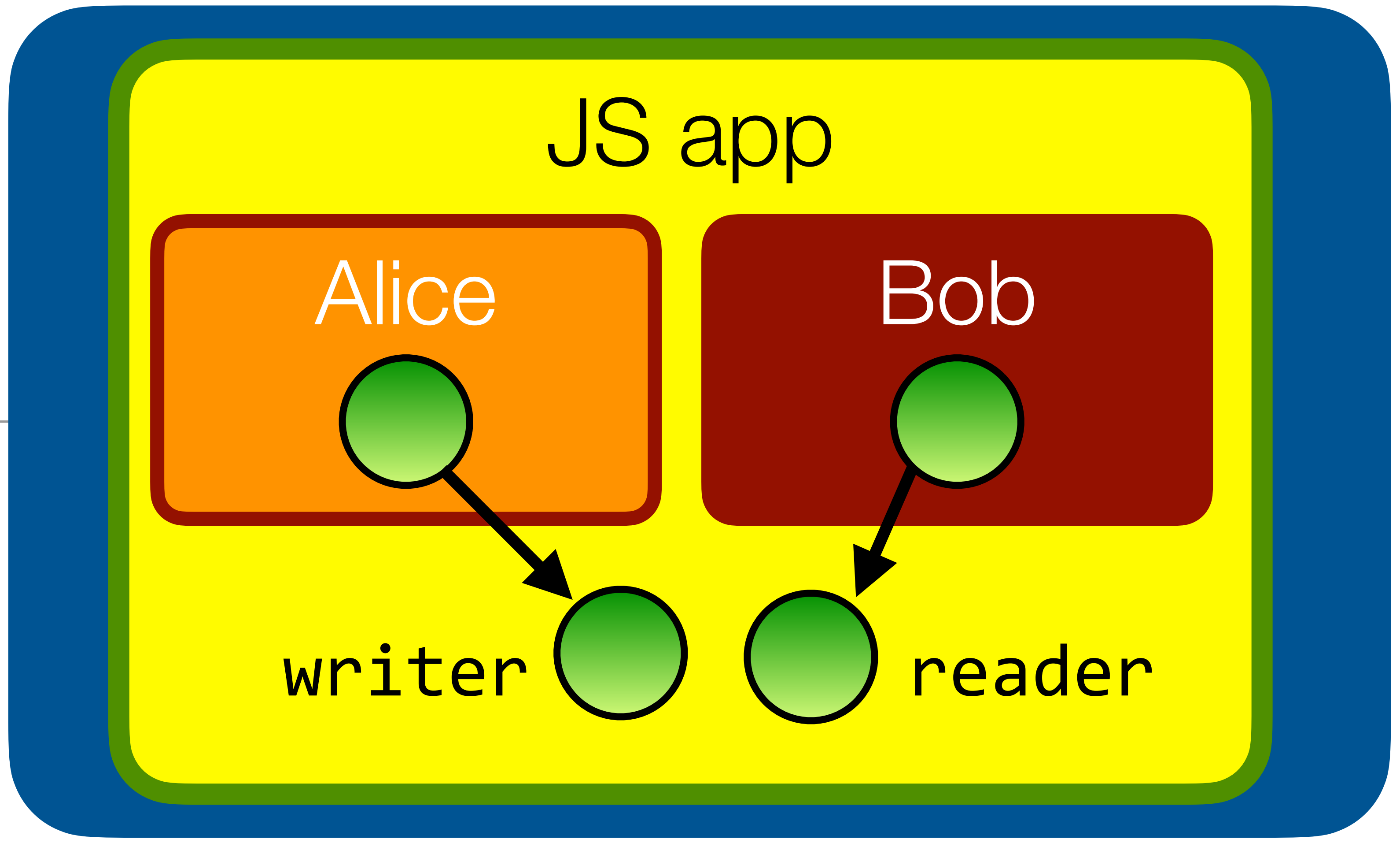


If over time we want to expose more functionality to Alice and Bob, we need to refactor all of our code.

```
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);
```

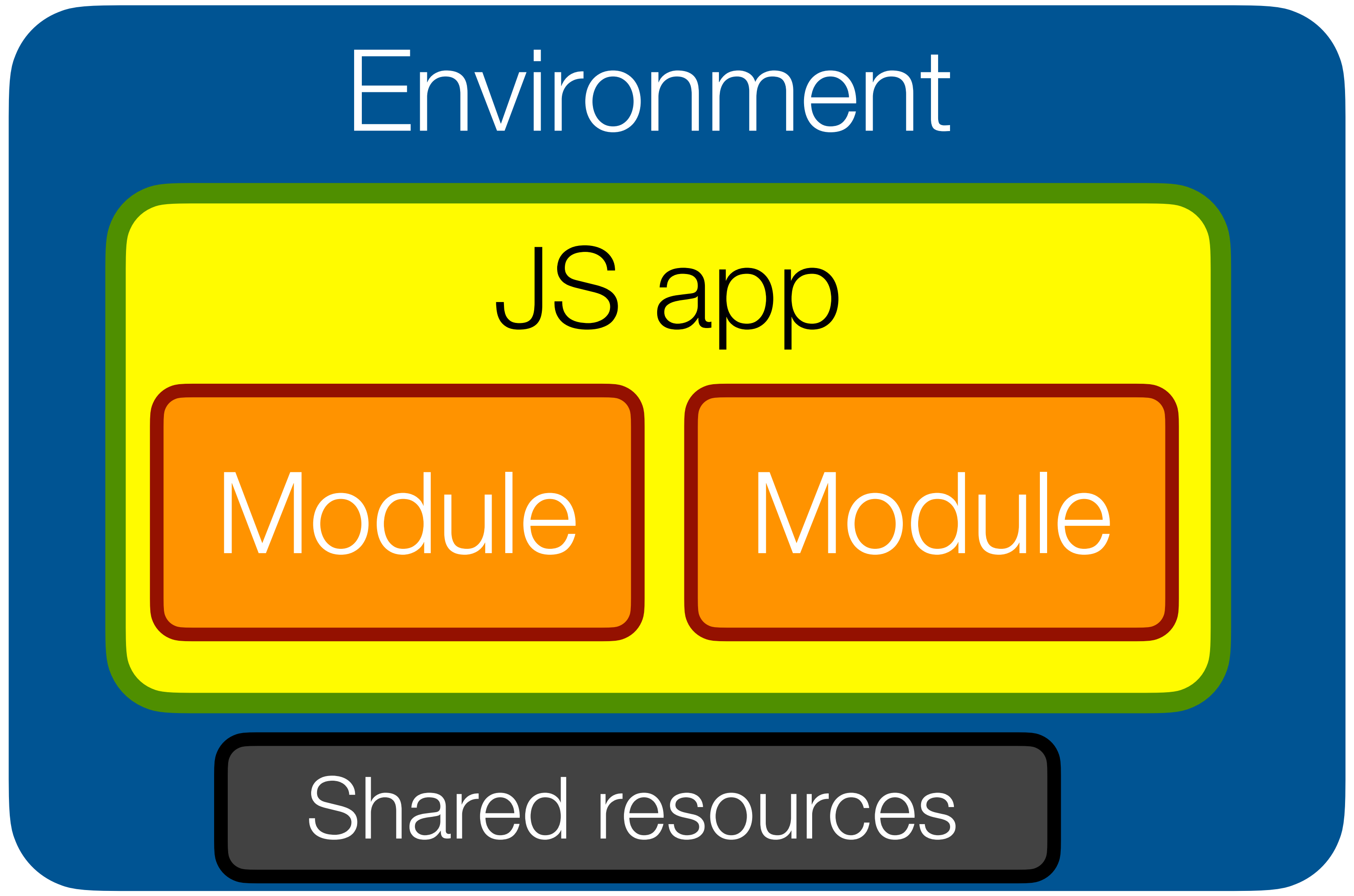
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

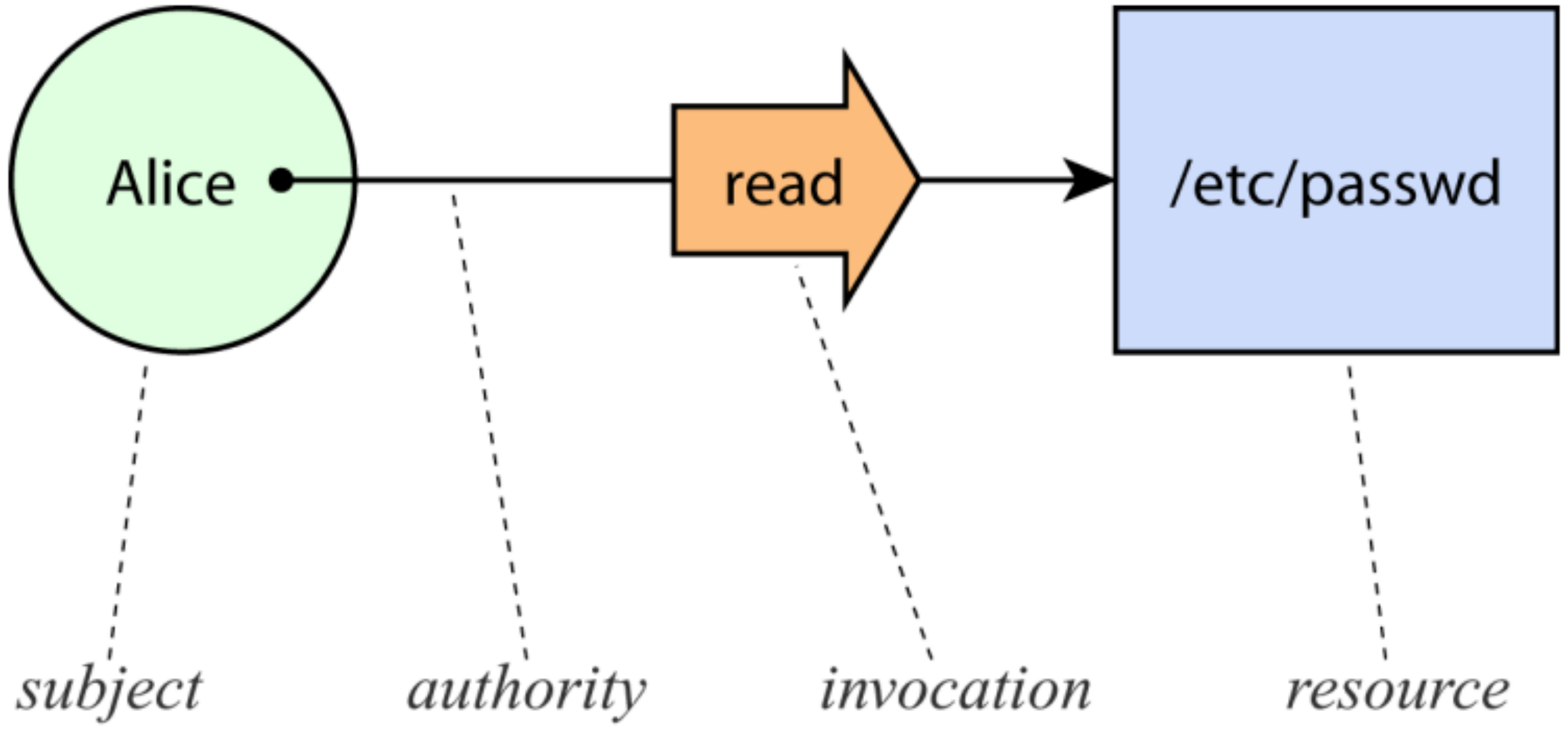
End of Part I: recap

- 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!**
- Fine-grained **access control** needed to **compose** functionality from untrusted modules in a least-authority manner

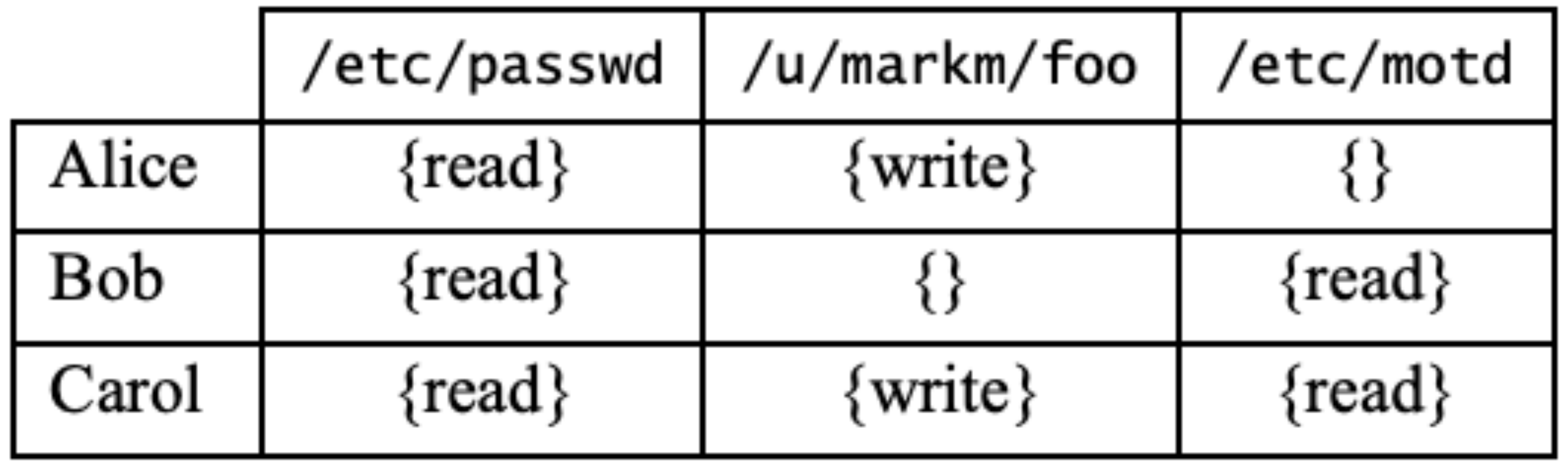


Part III
The object-capability model of access control

Access control: basic terminology



Access Matrix



	/etc/passwd	/u/markm/foo	/etc/motd
Alice	{read}	{write}	{}
Bob	{read}	{}	{read}
Carol	{read}	{write}	{read}

Who has what **authority** over which **resources**?

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

Principle of Least Authority (POLA): a tale of two copies

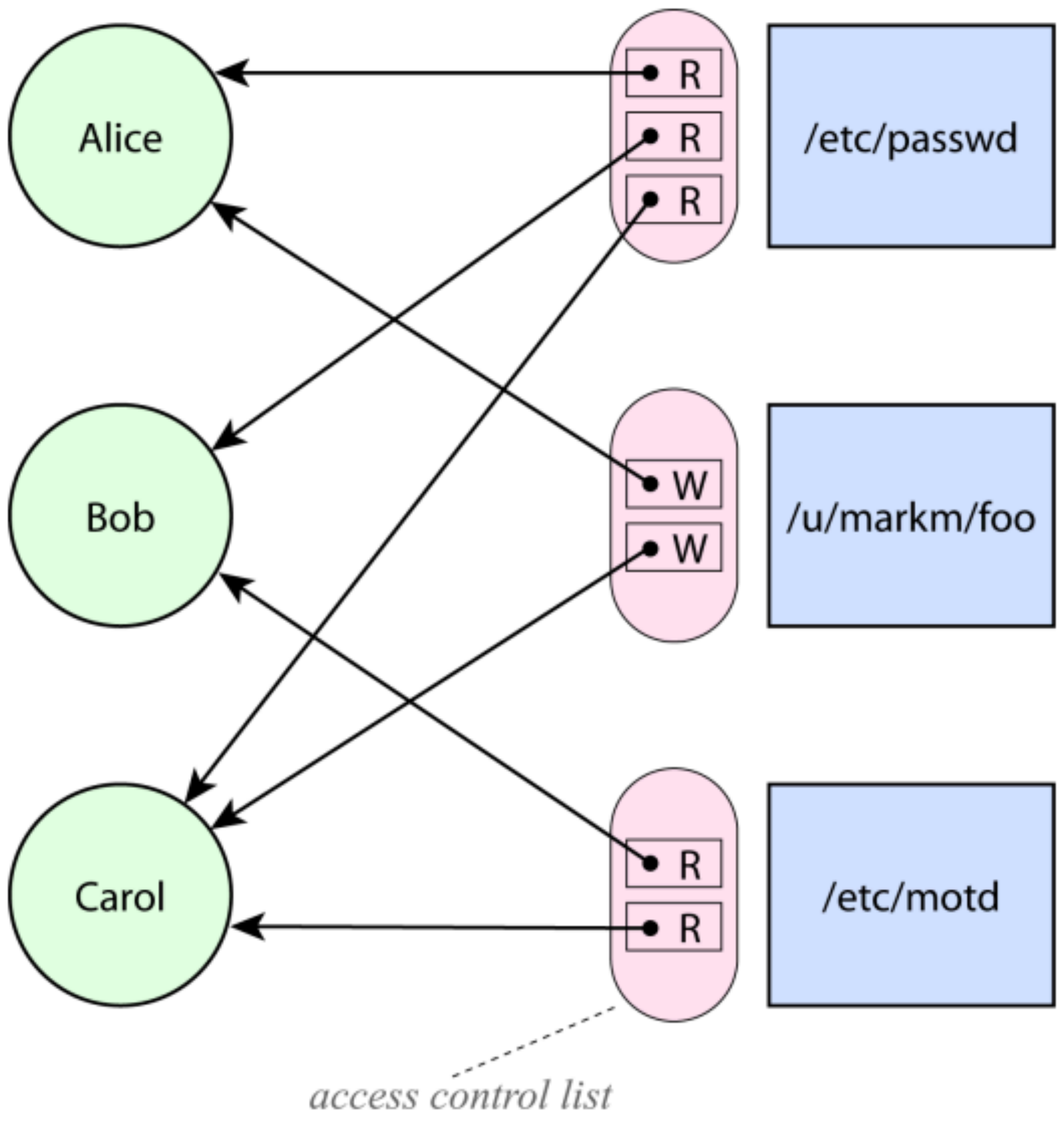
```
cp /home/tom/in.txt /home/tom/out.txt
```

```
cat < /home/tom/in.txt > /home/tom/out.txt
```

Access control: two alternative views

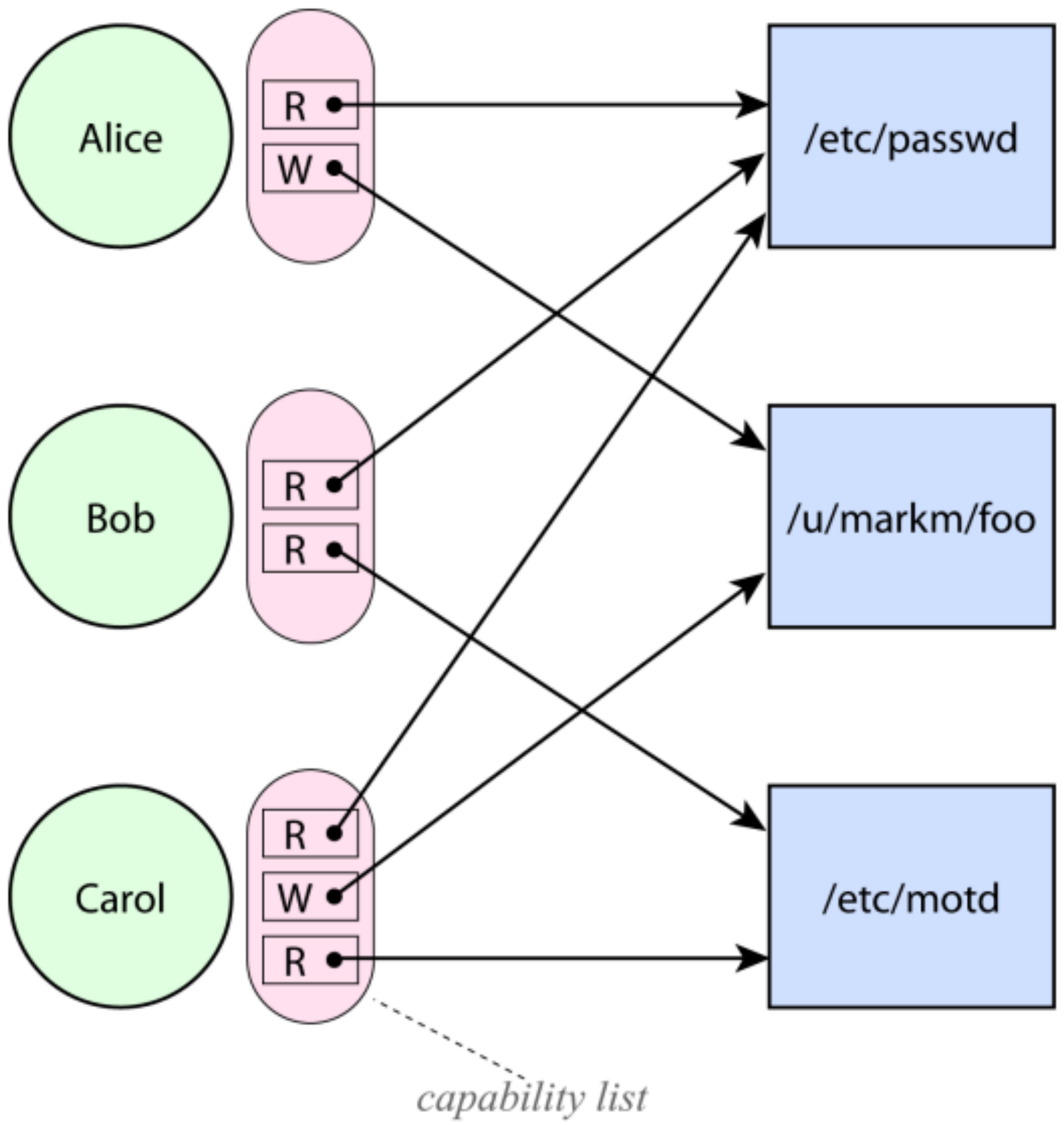
Access control lists (ACLs)

Access control organised around **identity.**



Capabilities (caps)

Access control organised around specific acts of **authorization.**

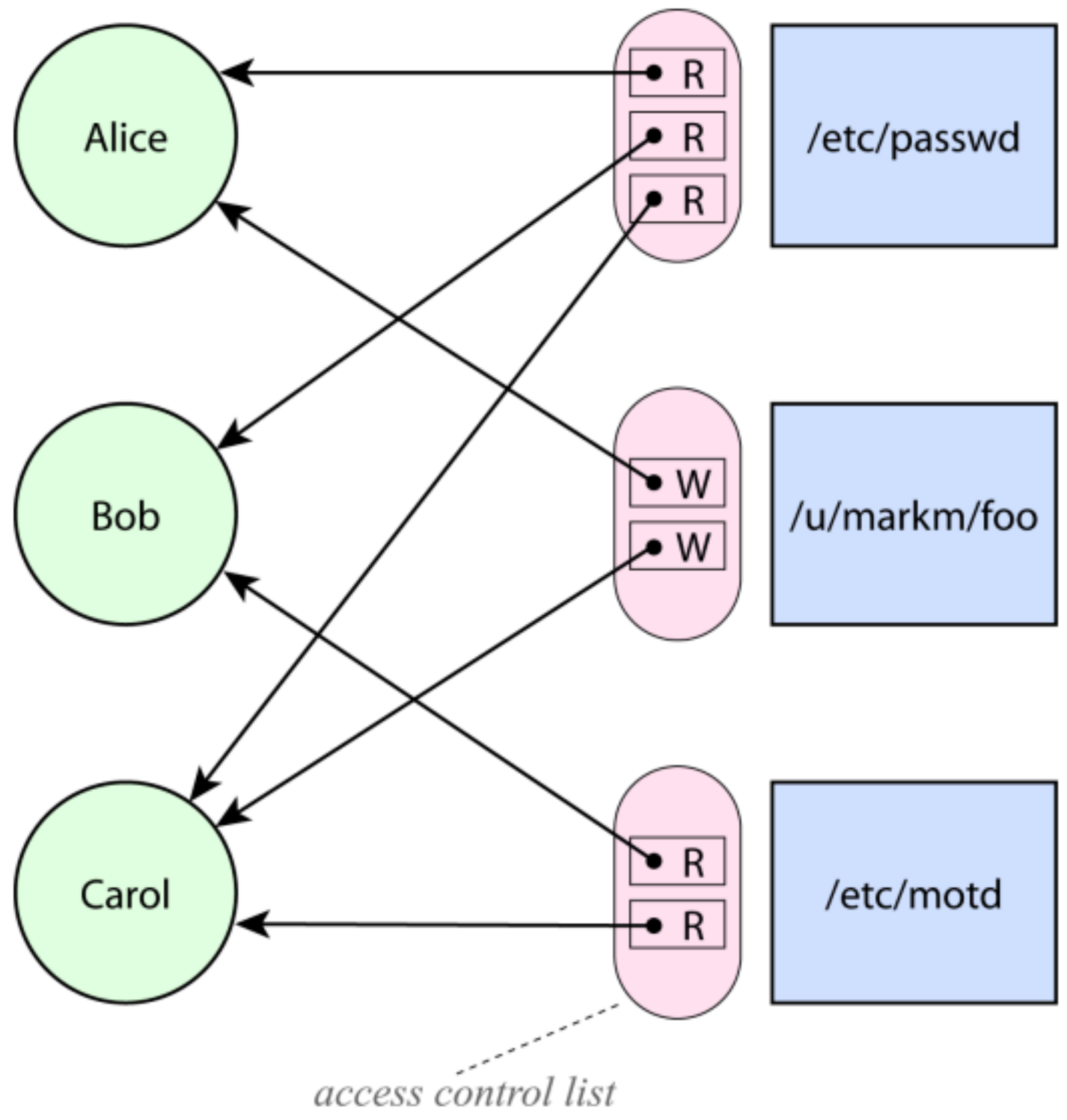


(source: Miller *et al.* “Capability myths demolished”, 2003)

Access control: two alternative views

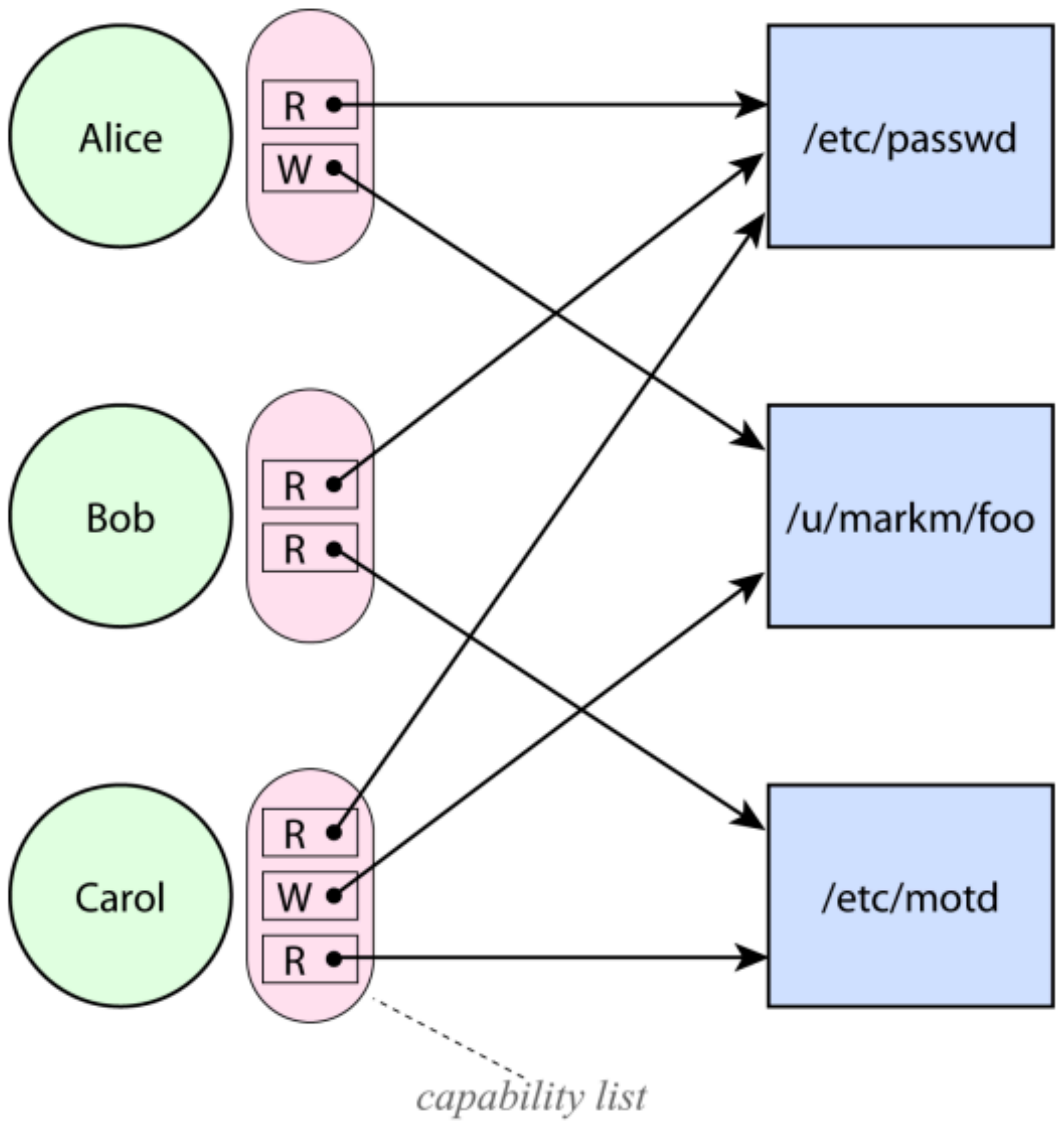
Access control lists (ACLs)

Access control organised around **identity.**



Capabilities (caps)

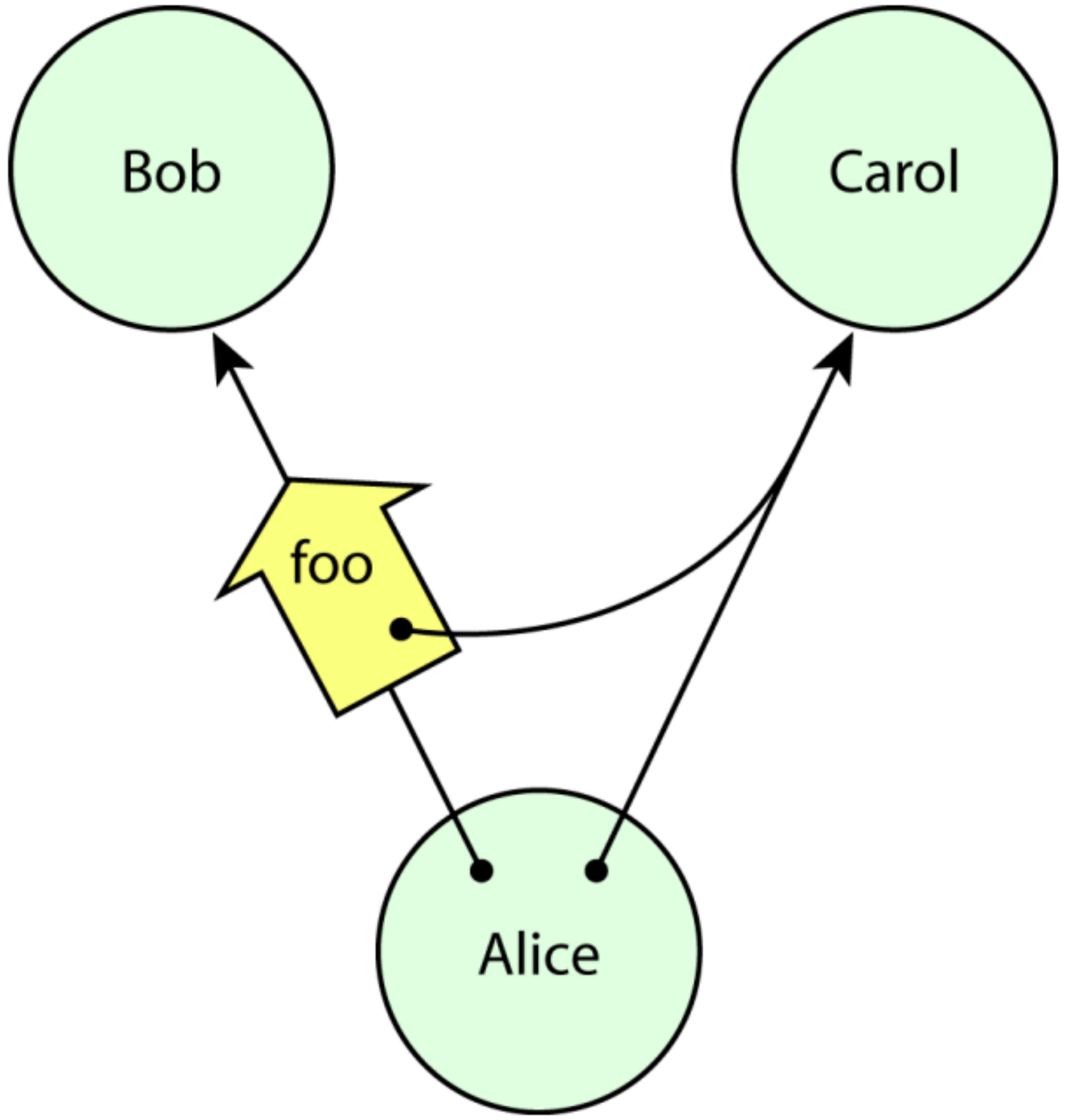
Access control organised around specific acts of **authorization.**



Despite the apparent symmetry, these models are not equivalent!

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

Capability systems excel at delegating authority



Granovetter Diagram

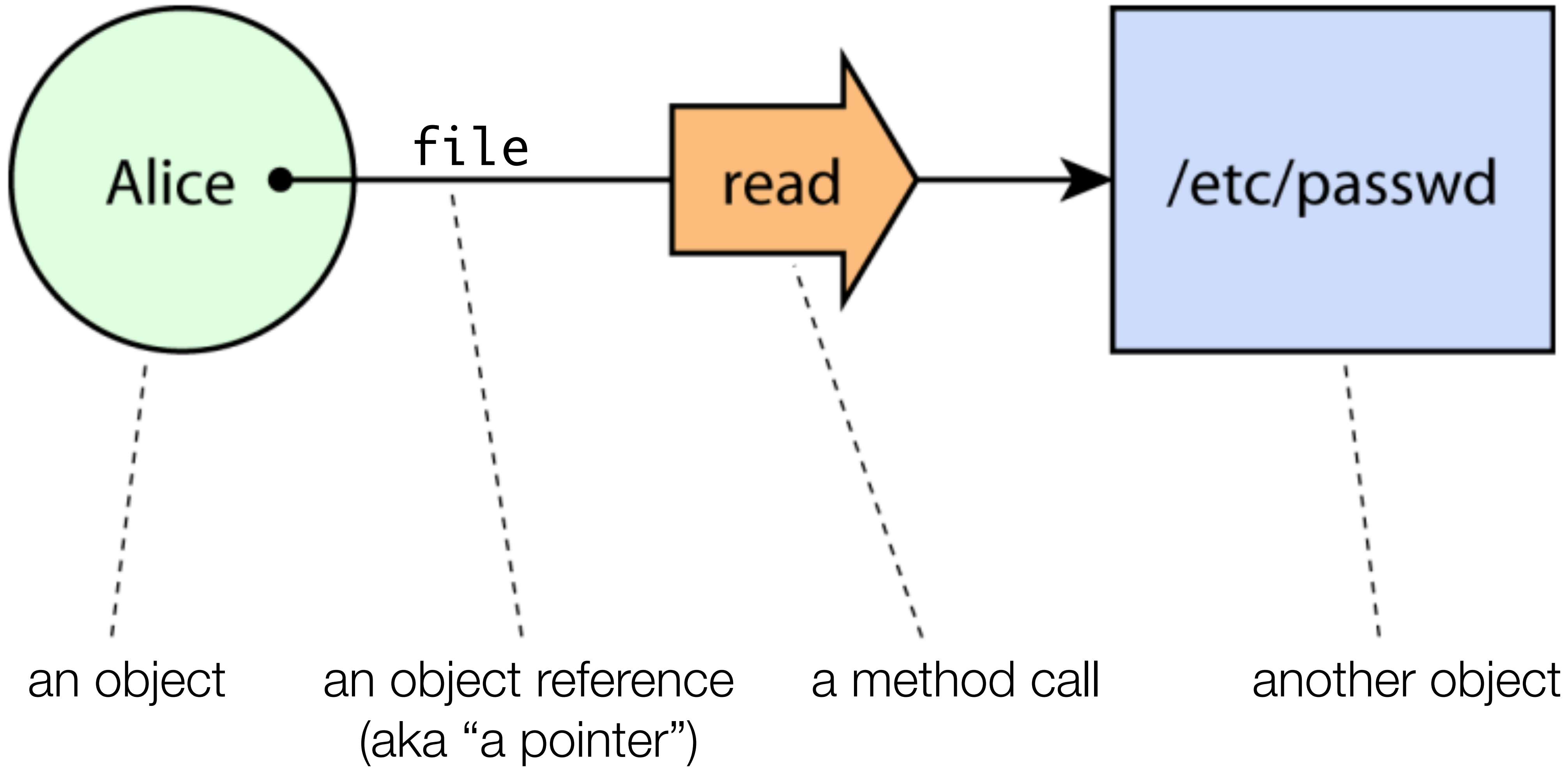
A capability both designates a resource *and* authorises some kind of access to it.

The two are inseparable.

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

What are **object**-capabilities?

- In a memory-safe programming language, an object-capability is simply **an unforgeable reference (a pointer)** to an object (or a function)
  - The designated resource = the object being pointed to
  - Exercising authority = invoking one of the designated object's public methods



Alice

file

read

/etc/passwd

an object

an object reference (aka "a pointer")

a method call

another object

```
// alice executes:
file.read()
```

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

When is a language an **object-capability language**?

1. The language must be **memory-safe**: object pointers are unforegeable
    - Cannot typecast an int to a pointer, cannot randomly access heap memory, …
2. The language must offer strong **encapsulation**
    - Objects need a way to privately store pointers to other objects
3. The language must **not** provide access to **undeniable** (ambient) **authority**
    - Examples of undeniable authority: the ability to import arbitrary modules, the ability to update mutable global variables
4. The only way to **delegate authority** is by sharing a pointer to an object
    - “Only connectivity begets connectivity”

“Only connectivity begets connectivity”

Three simple rules that describe how authority can be acquired in a capability-secure system:

Creation: e.g. alice creates carol herself

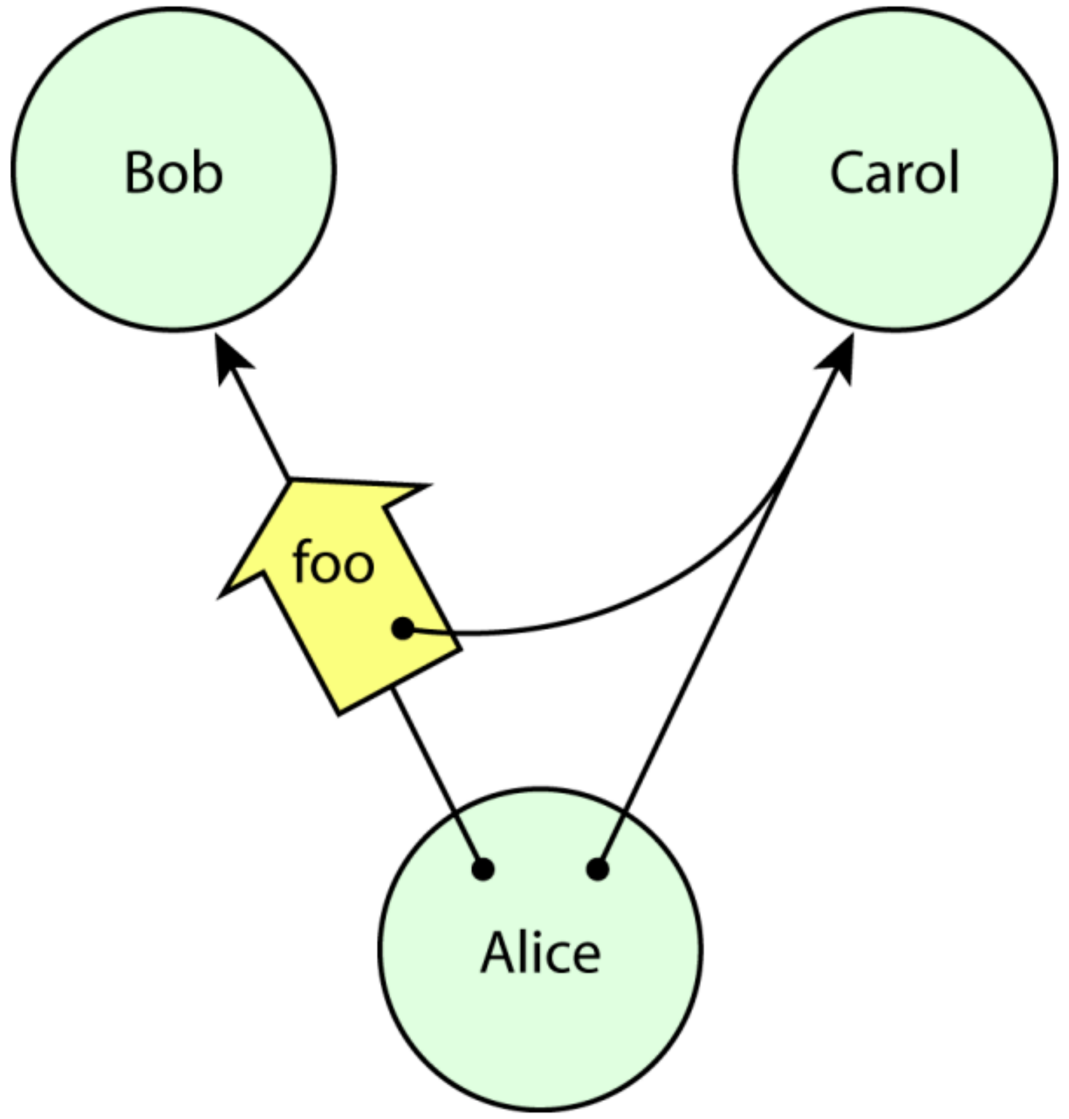
```
// alice executes:
let carol = makeCarol()
```

Endowment: e.g. at creation, alice is endowed with authority to access carol

```
// alice’s constructor:
function makeAlice(carol) {…}
```

Transfer: e.g. alice transfers carol to bob

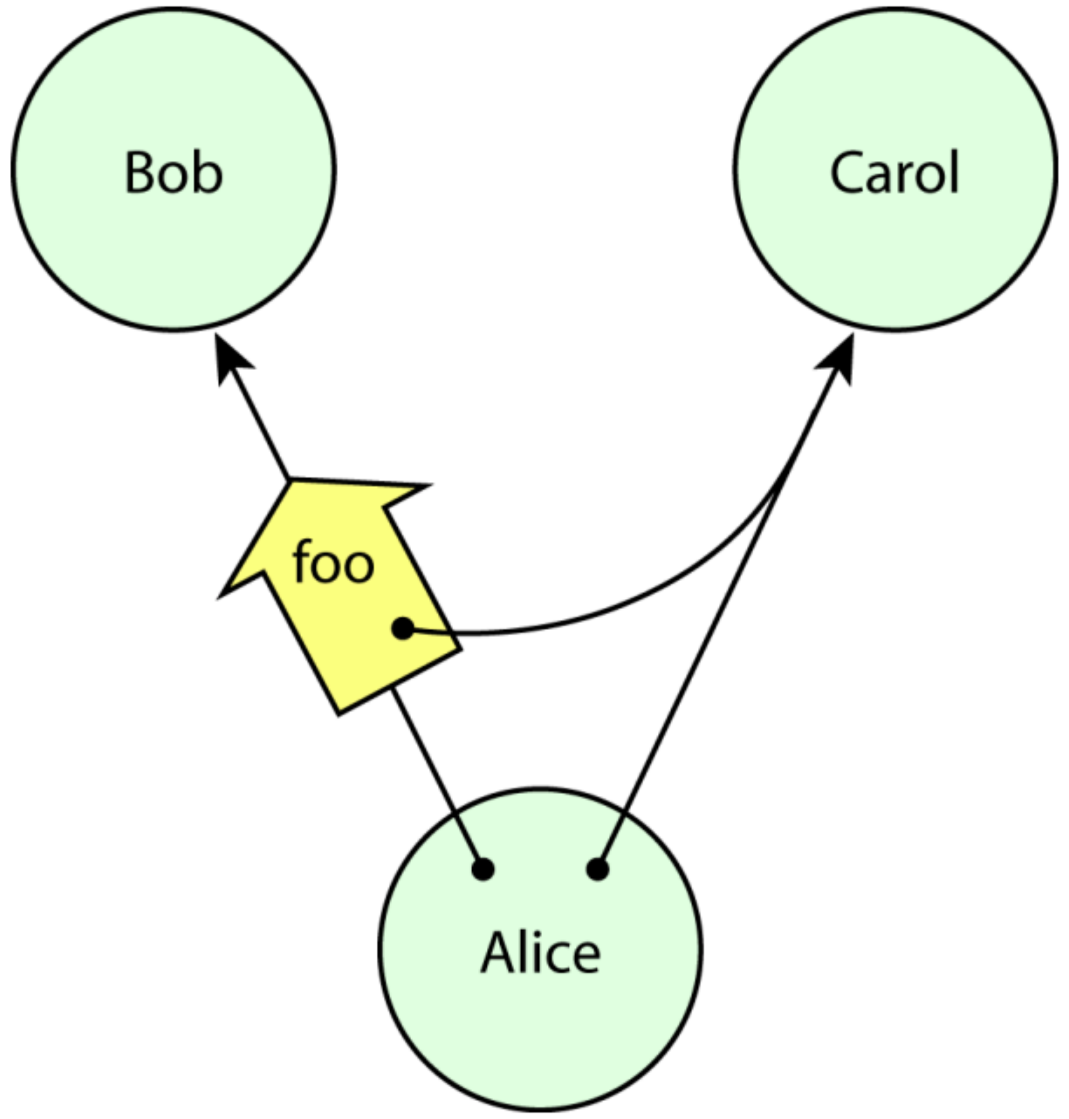
```
// alice executes:
bob.foo(carol)
```



Considerations when delegating authority using capabilities

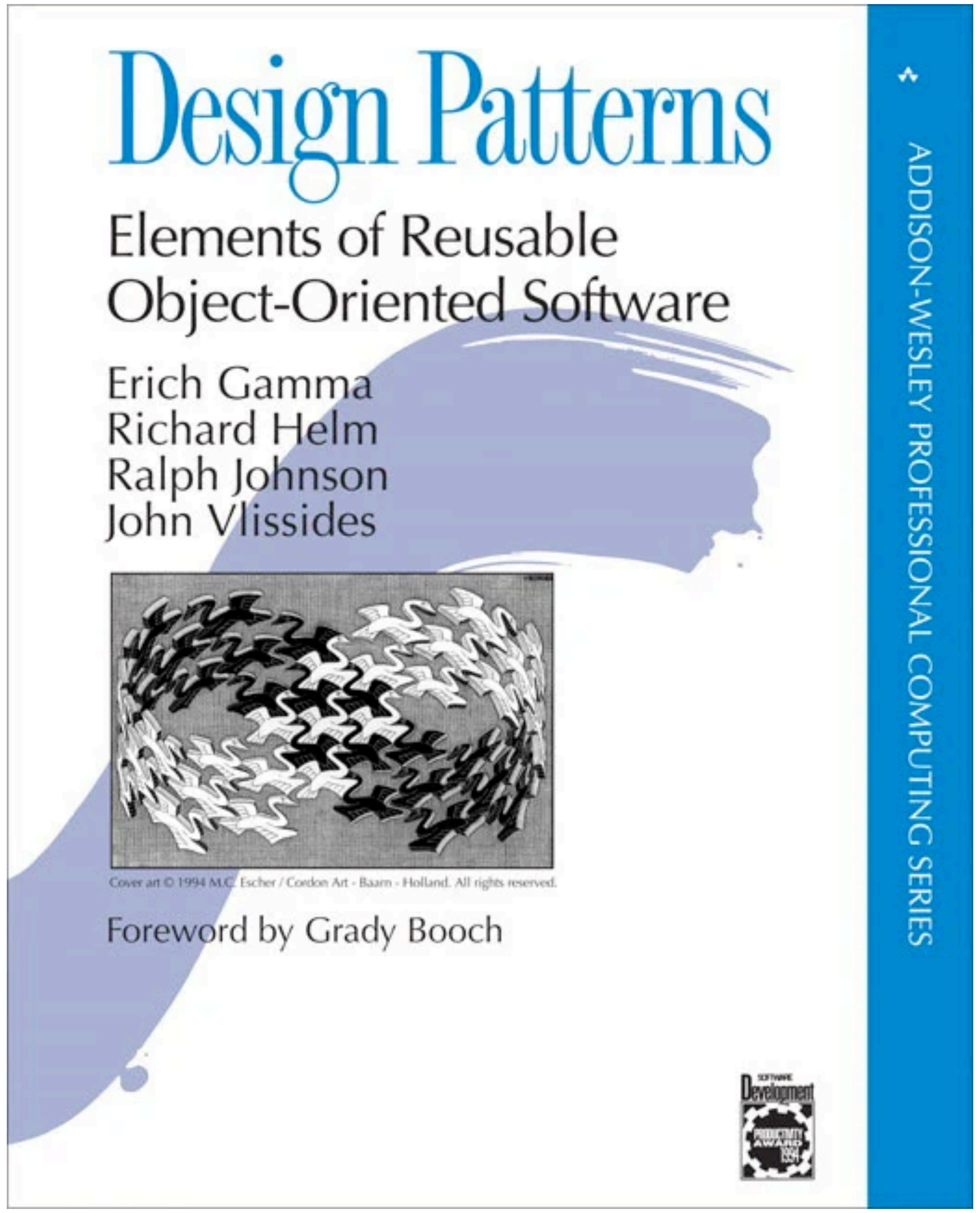
When Alice delegates authority to Bob, she may want to **limit** the authority given to Bob **(attenuation)**

Bob may also want to combine the authority given to him with his other authorities to gain **additional** authorities **(rights amplification)**



Part IV
Object-capability Patterns

Design Patterns (“Gang of Four”, 1994)



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

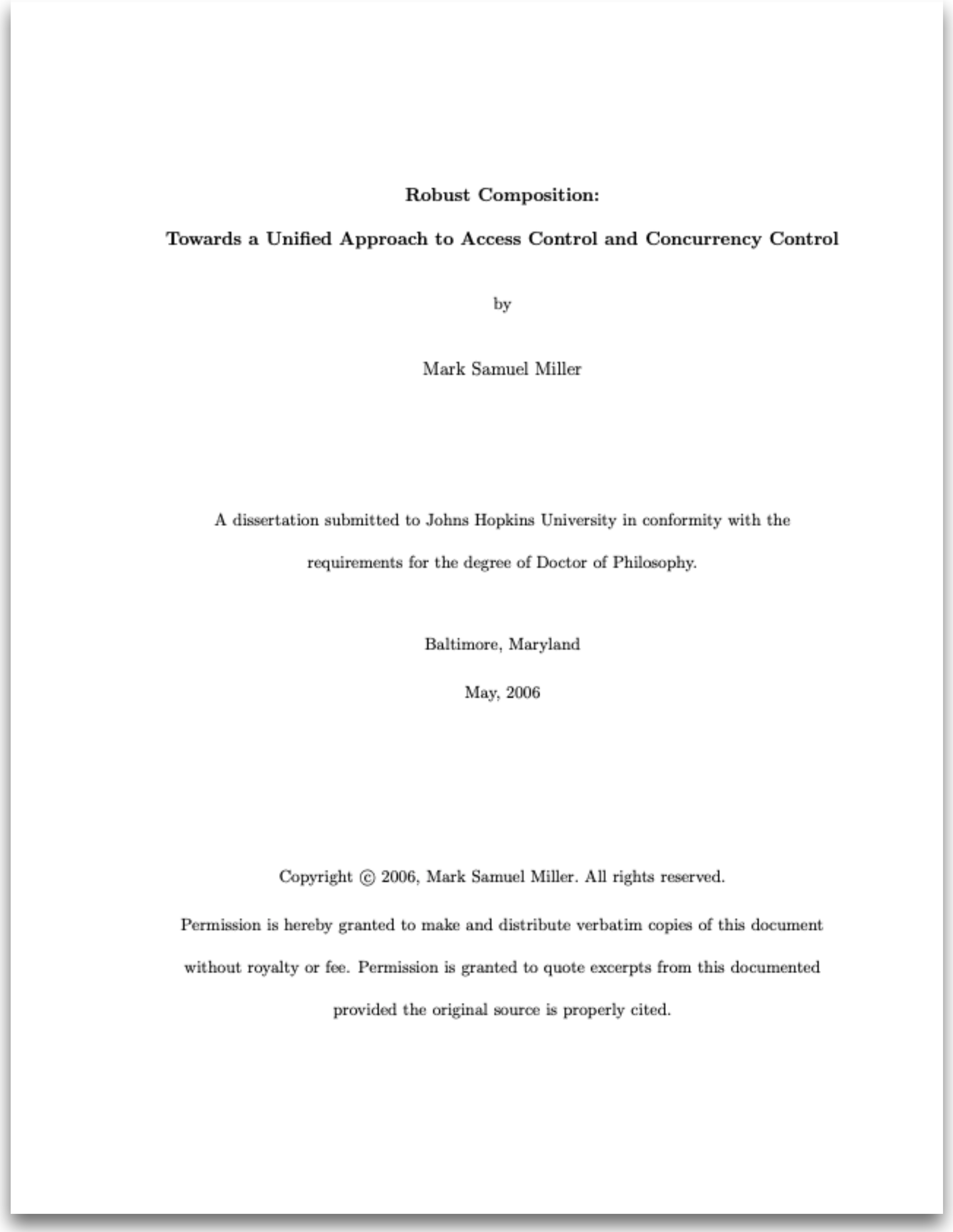
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

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

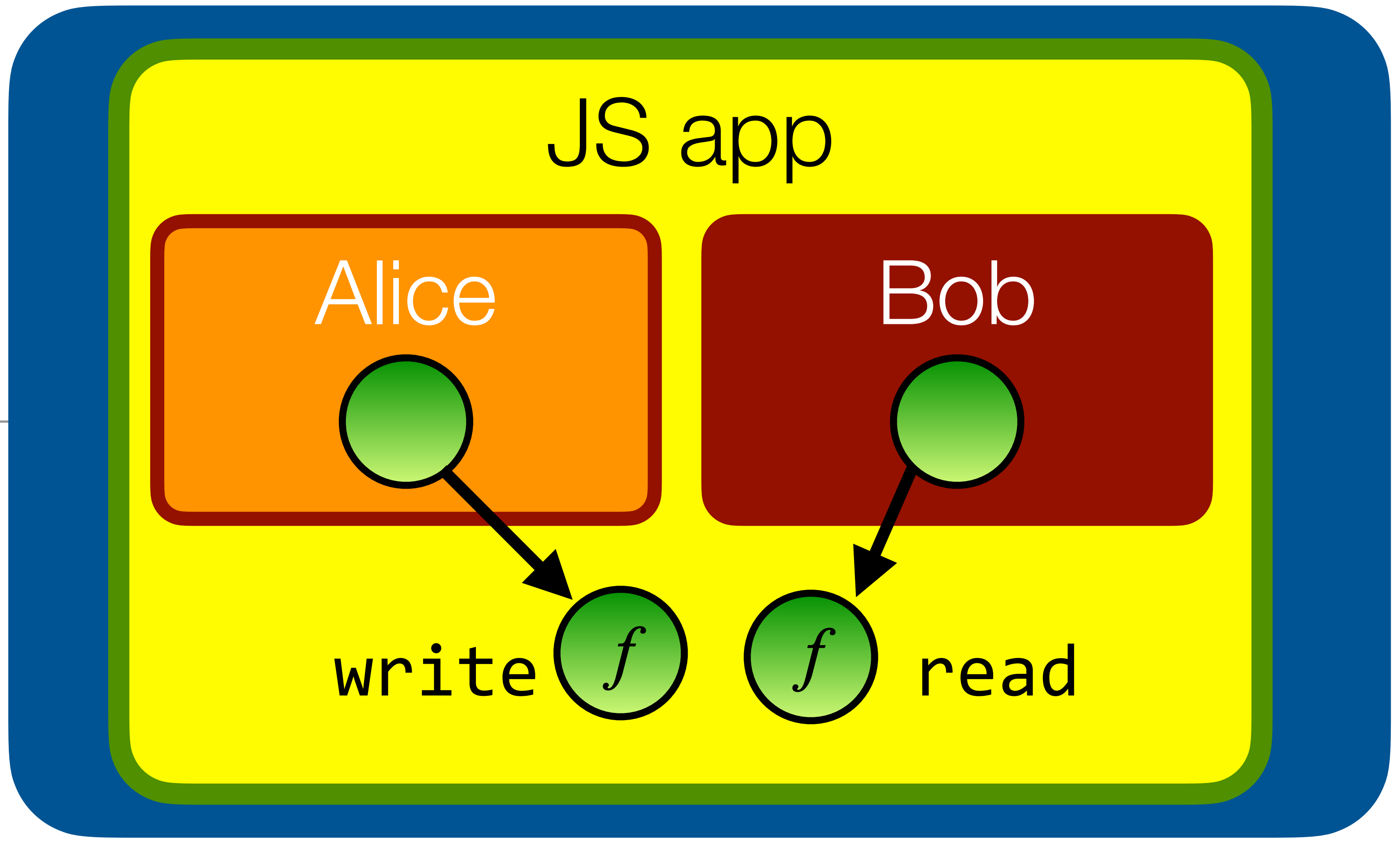
Copyright © 2006, Mark Samuel Miller. All rights reserved.

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

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

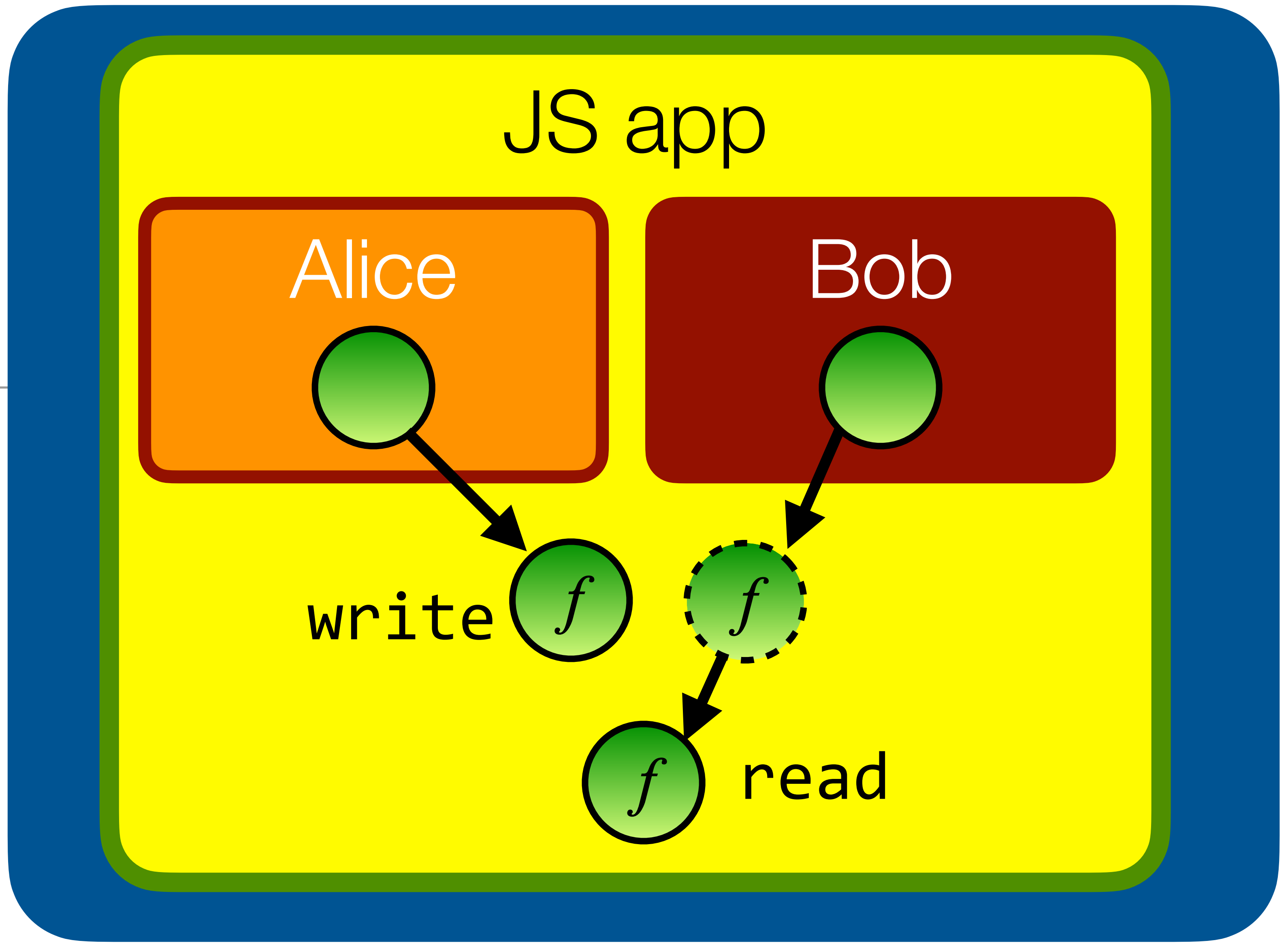
Further limiting Bob’s authority



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);
```

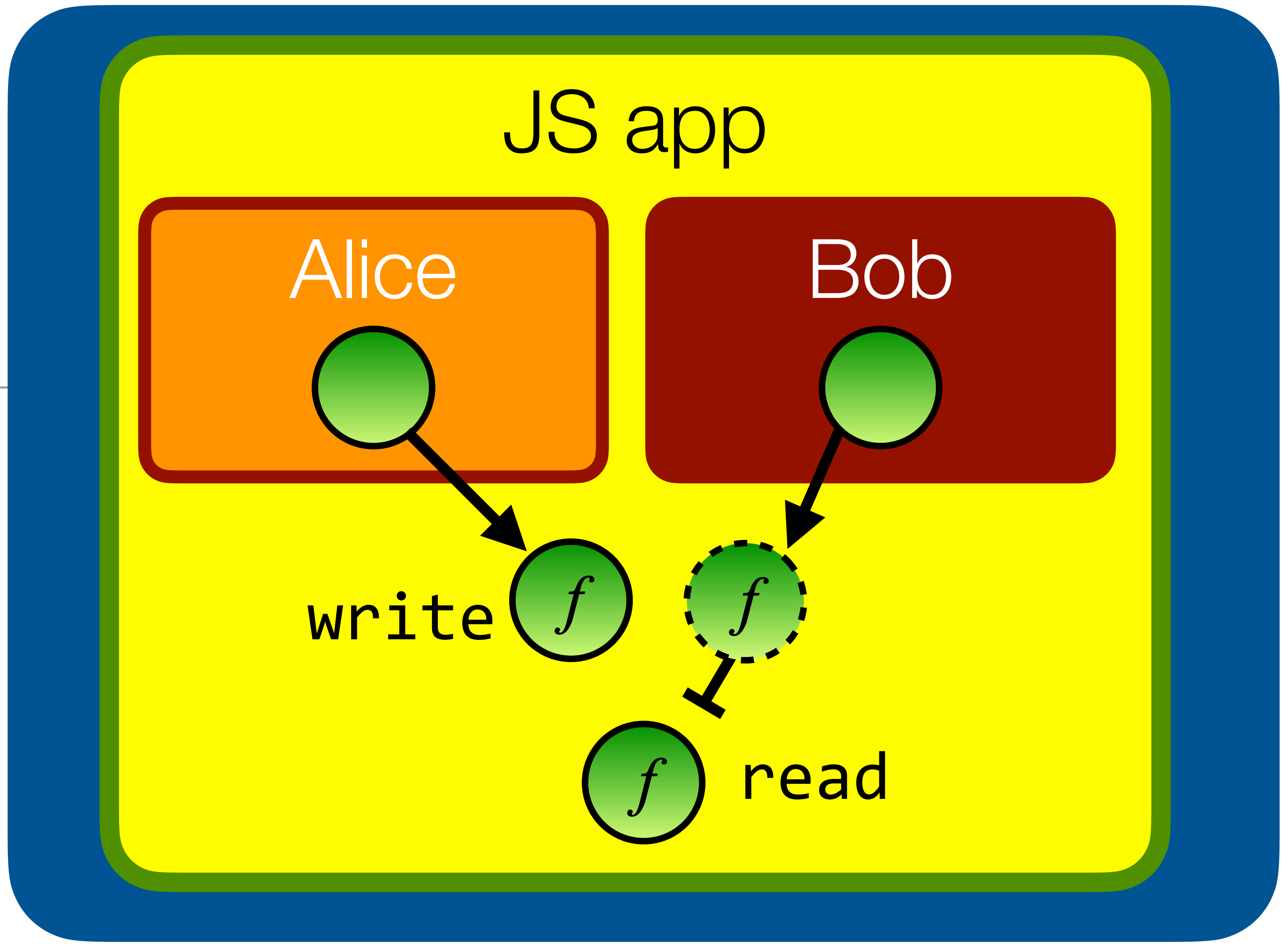
Use **caretaker** to insert access control logic



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);
```

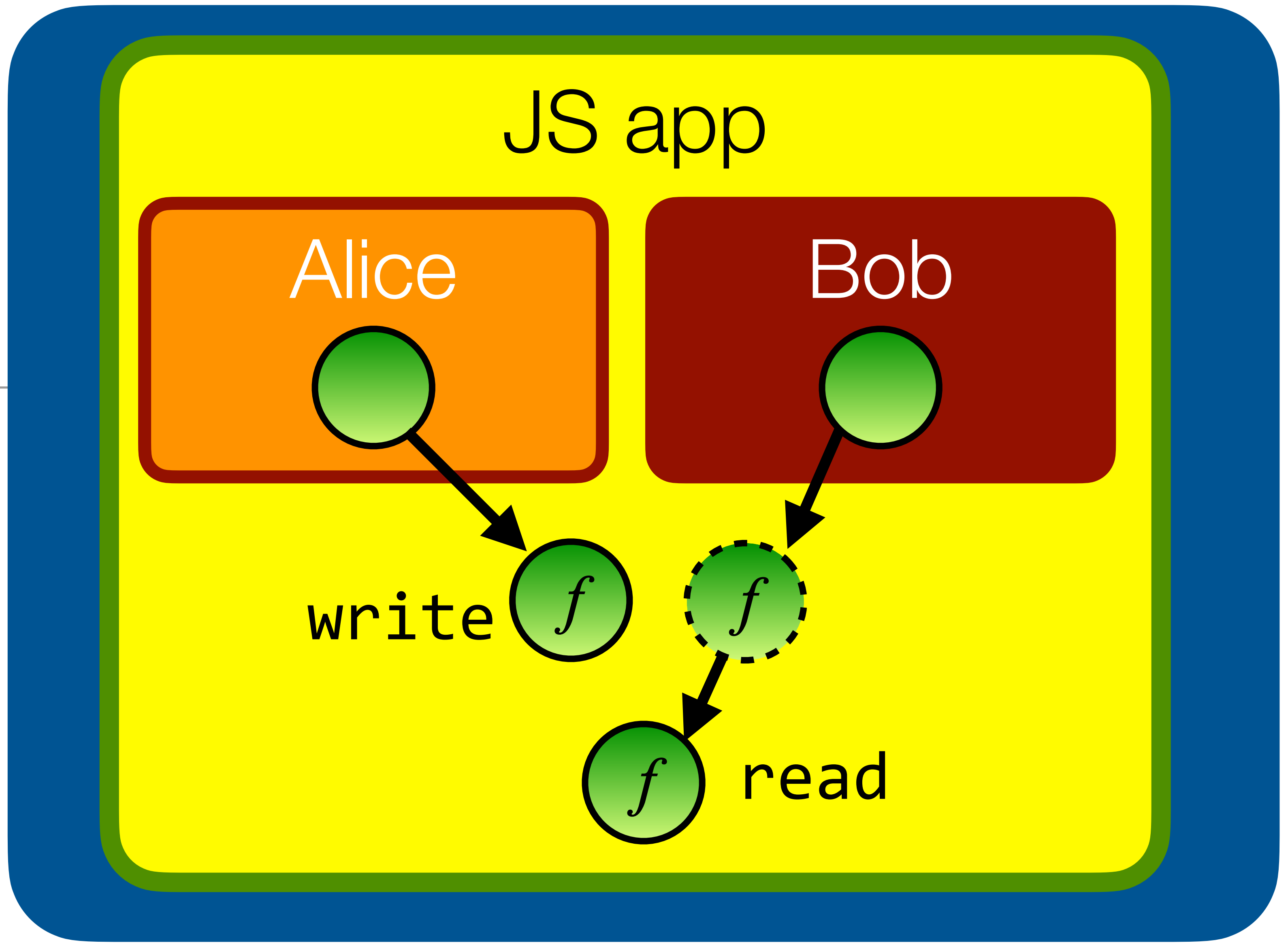
Use **caretaker** to insert access control logic



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();
```

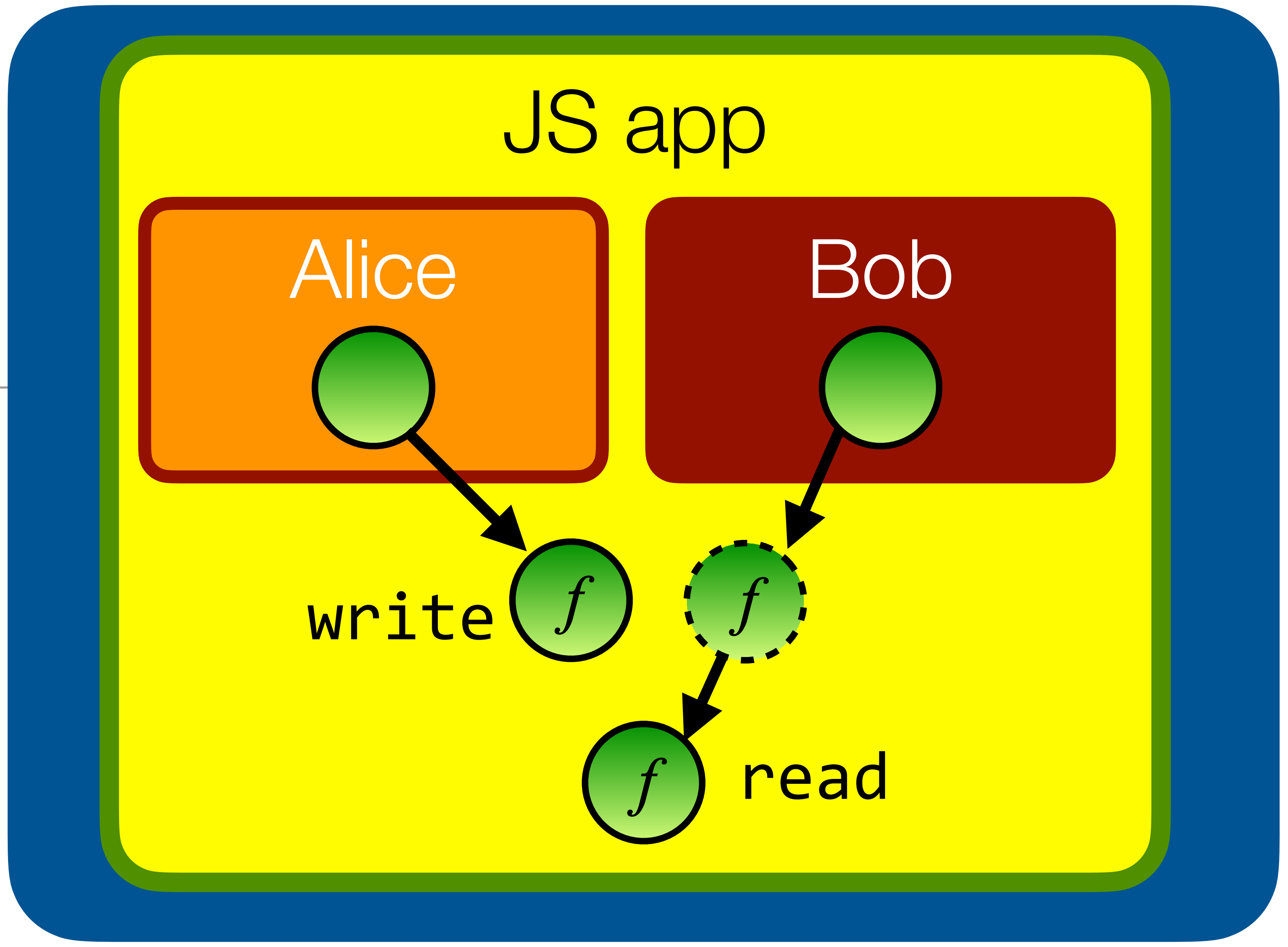
Use **caretaker** to insert access control logic



```
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]);
}
```

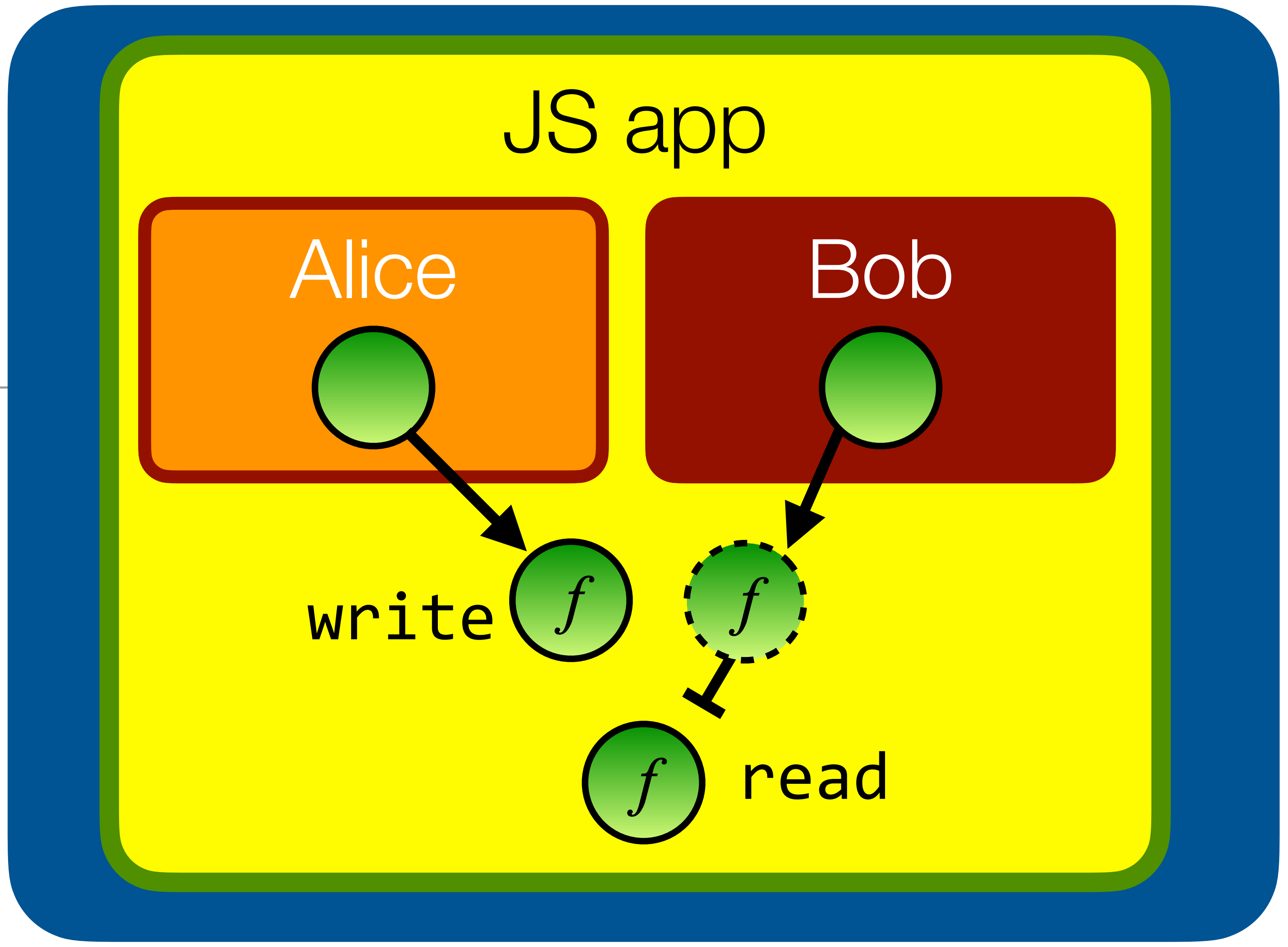
A caretaker is just a proxy object



```
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

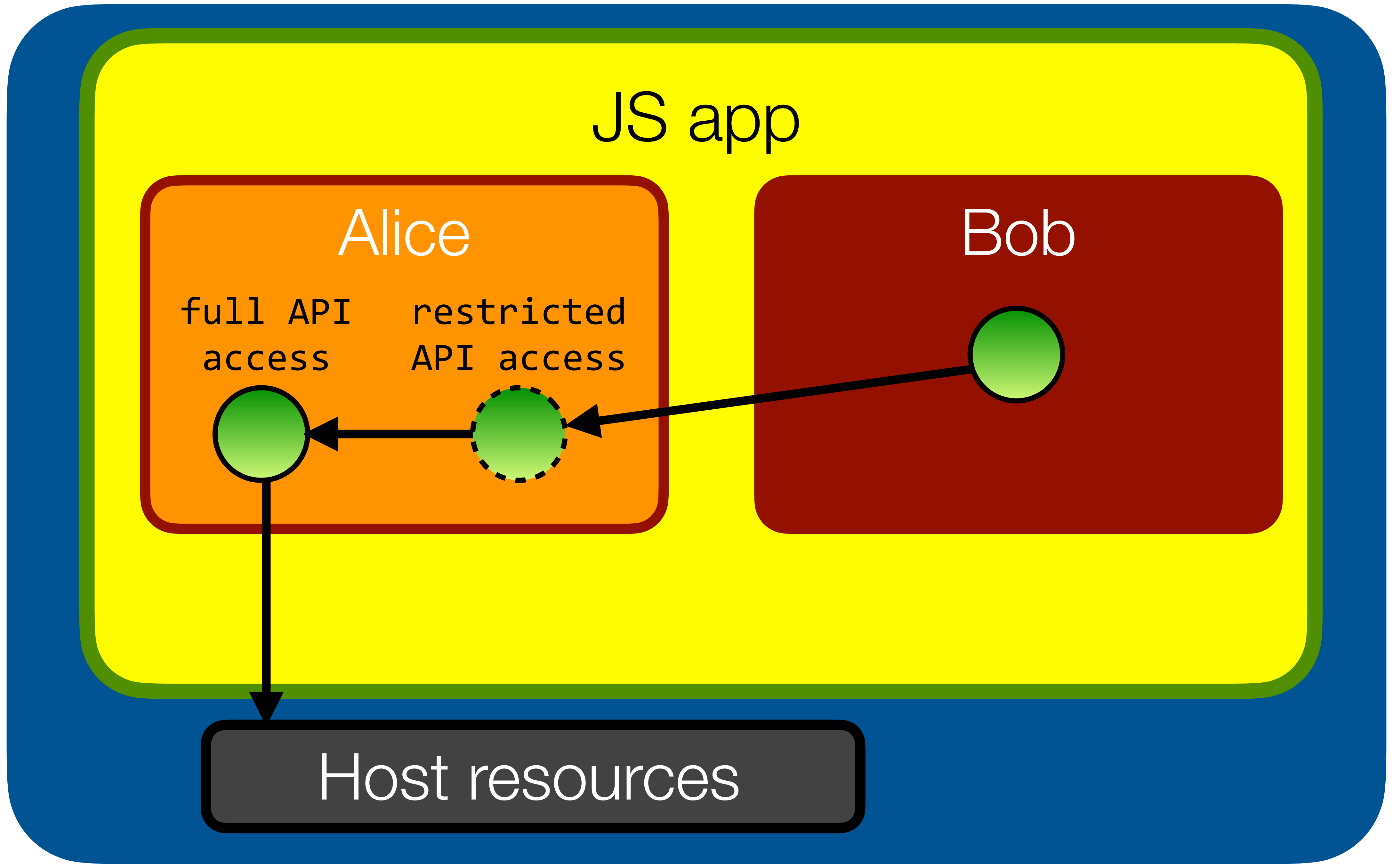


```
import * as alice from "alice.js";
import * as bob from “bob.js";

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]);
}
```

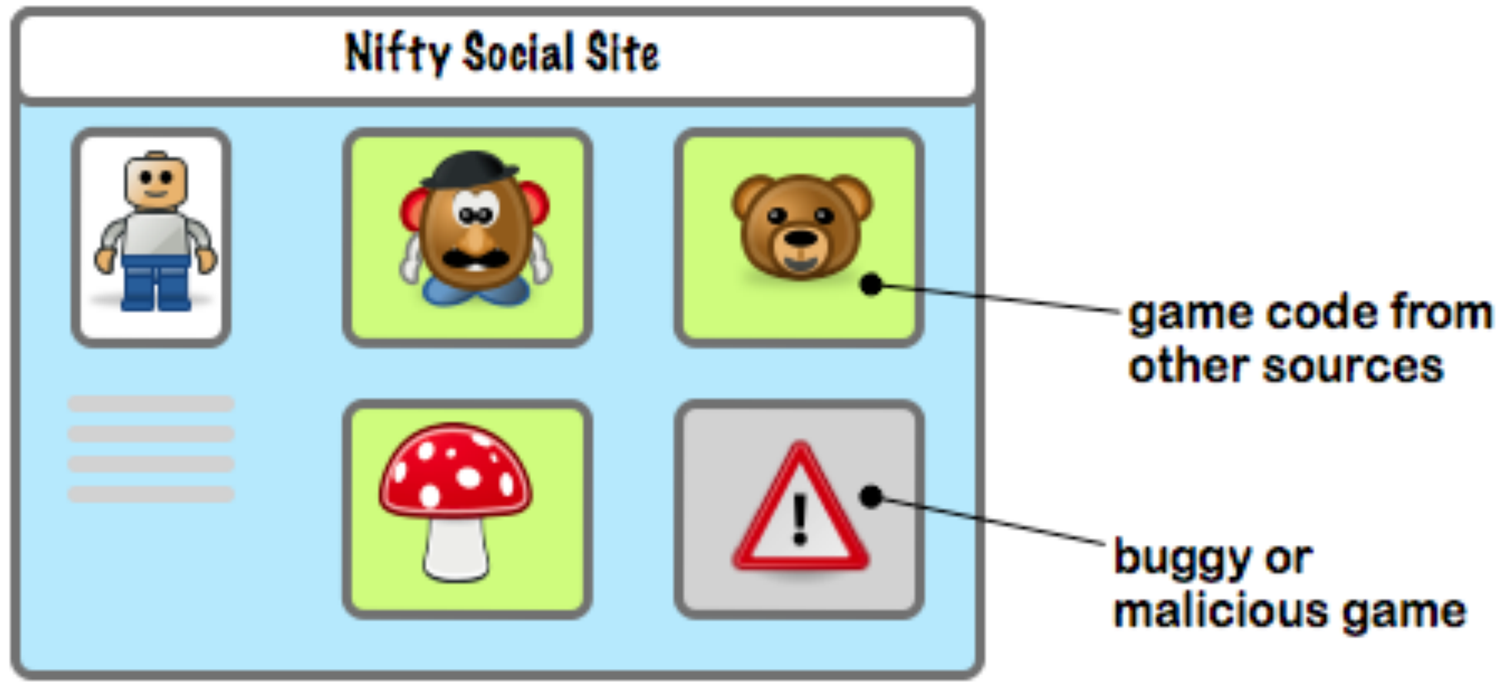
Taming is the process of restricting access to powerful APIs

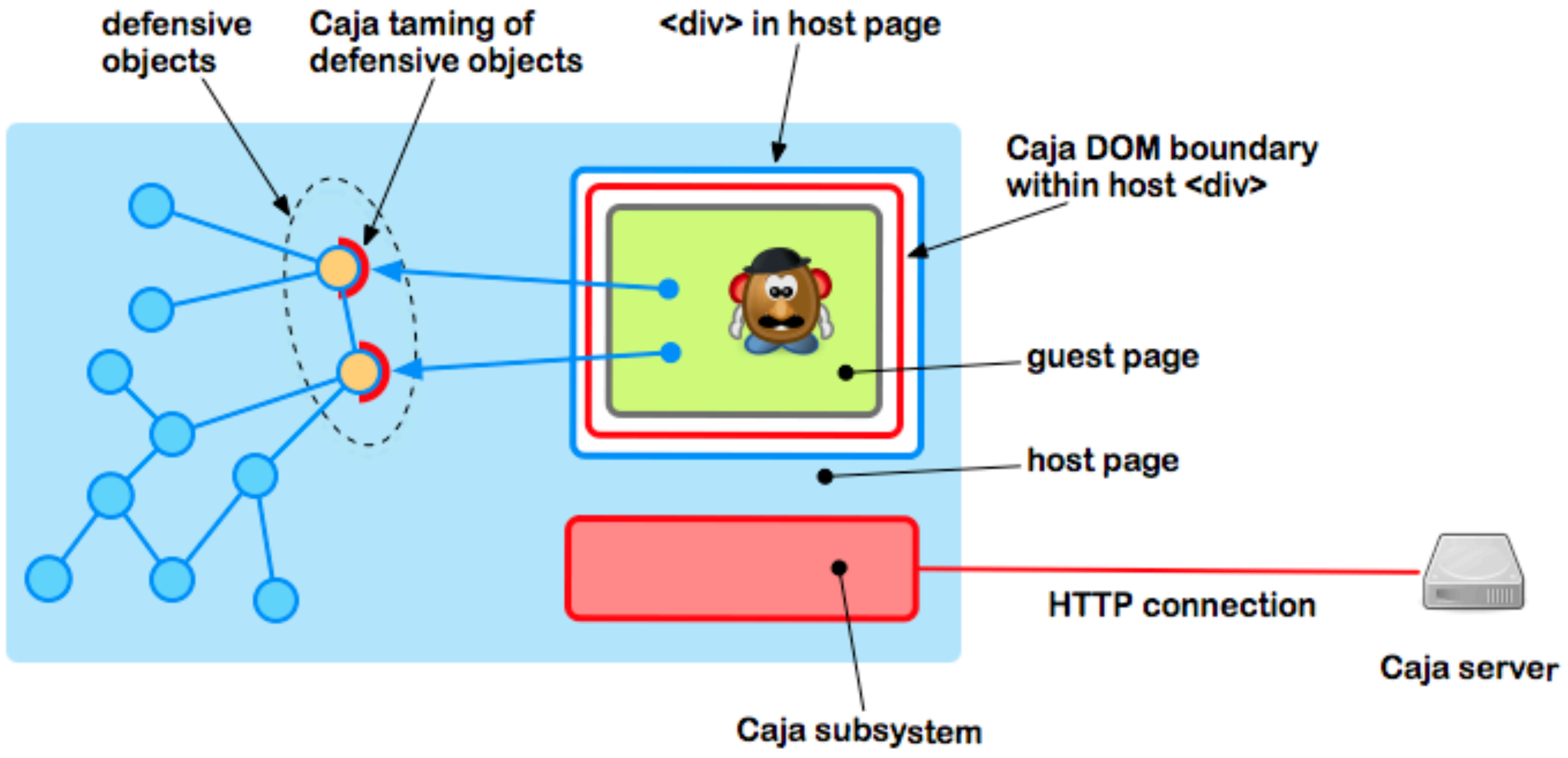
- Expose powerful objects through restrictive proxies to third-party code
- E.g. Alice might give Bob read-only access to a specific subdirectory of her file system



Taming is the process of restricting access to powerful APIs

- Example: how Google Caja limits access to the browser DOM

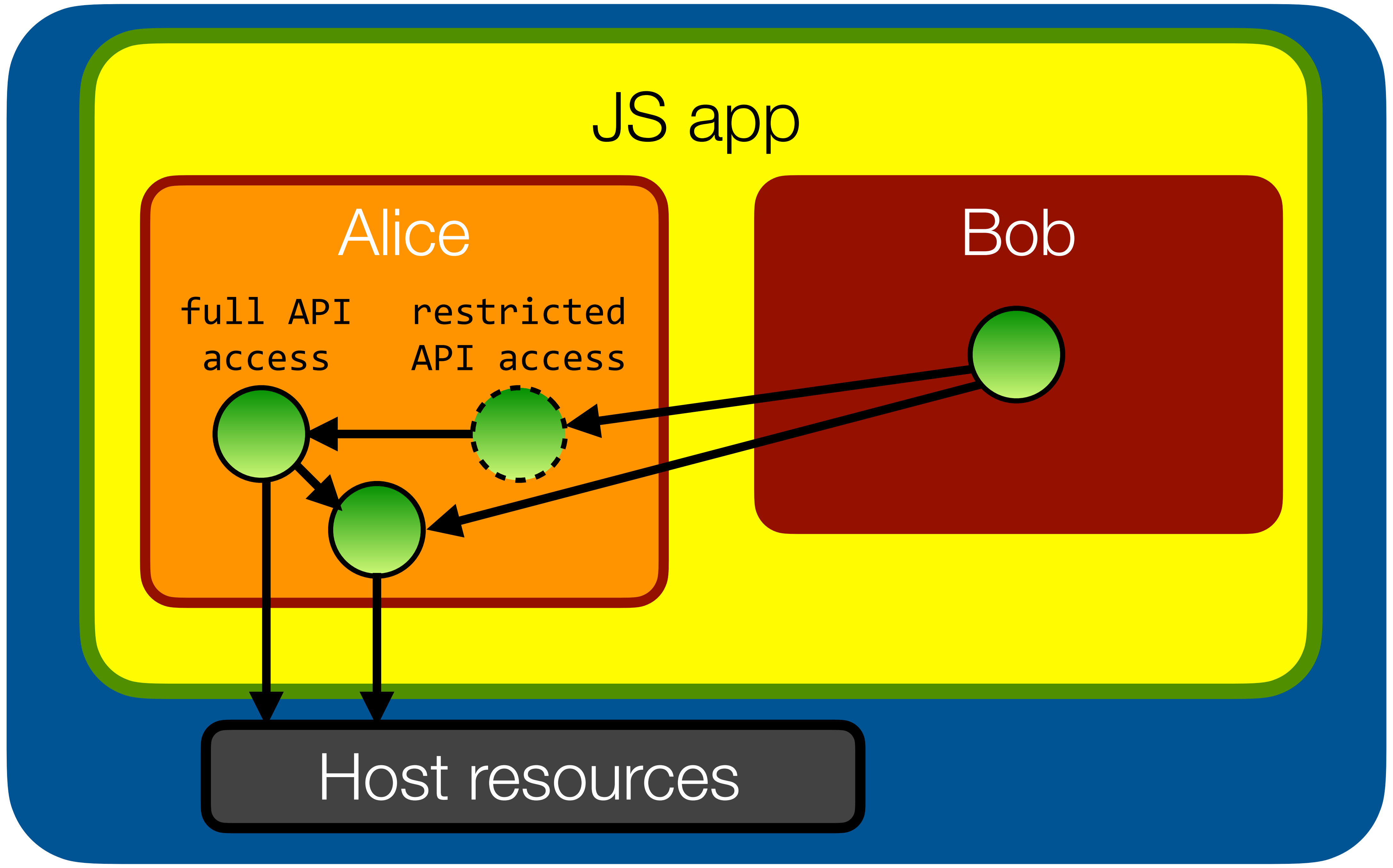




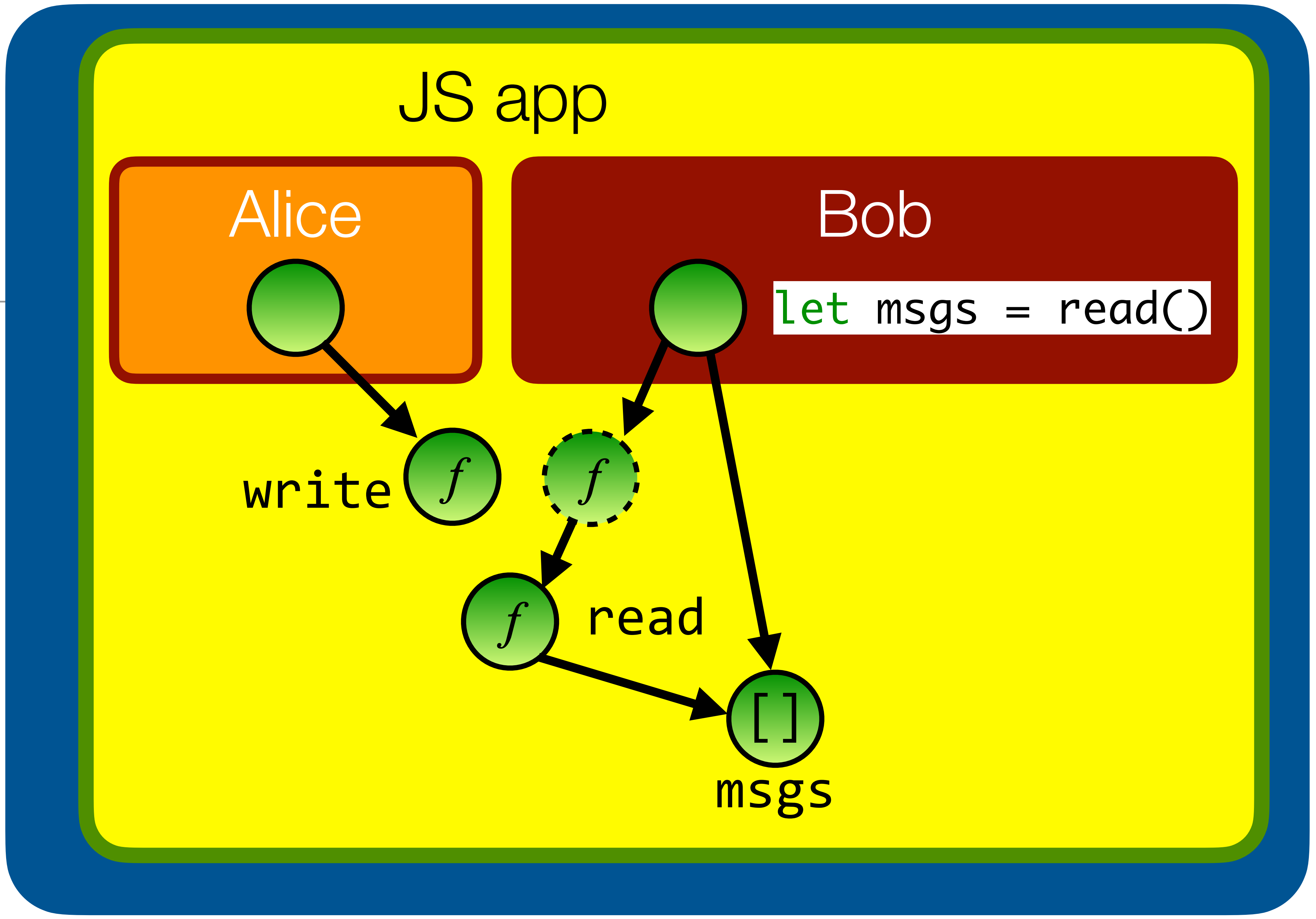
(source: Google Caja documentation: https://developers.google.com/caja/docs/about)

Taming is the process of restricting access to powerful APIs

Potential **hazard:** the taming proxy must ensure it does not “leak” any host resources via its restricted API.



Bob may still access the log’s messages

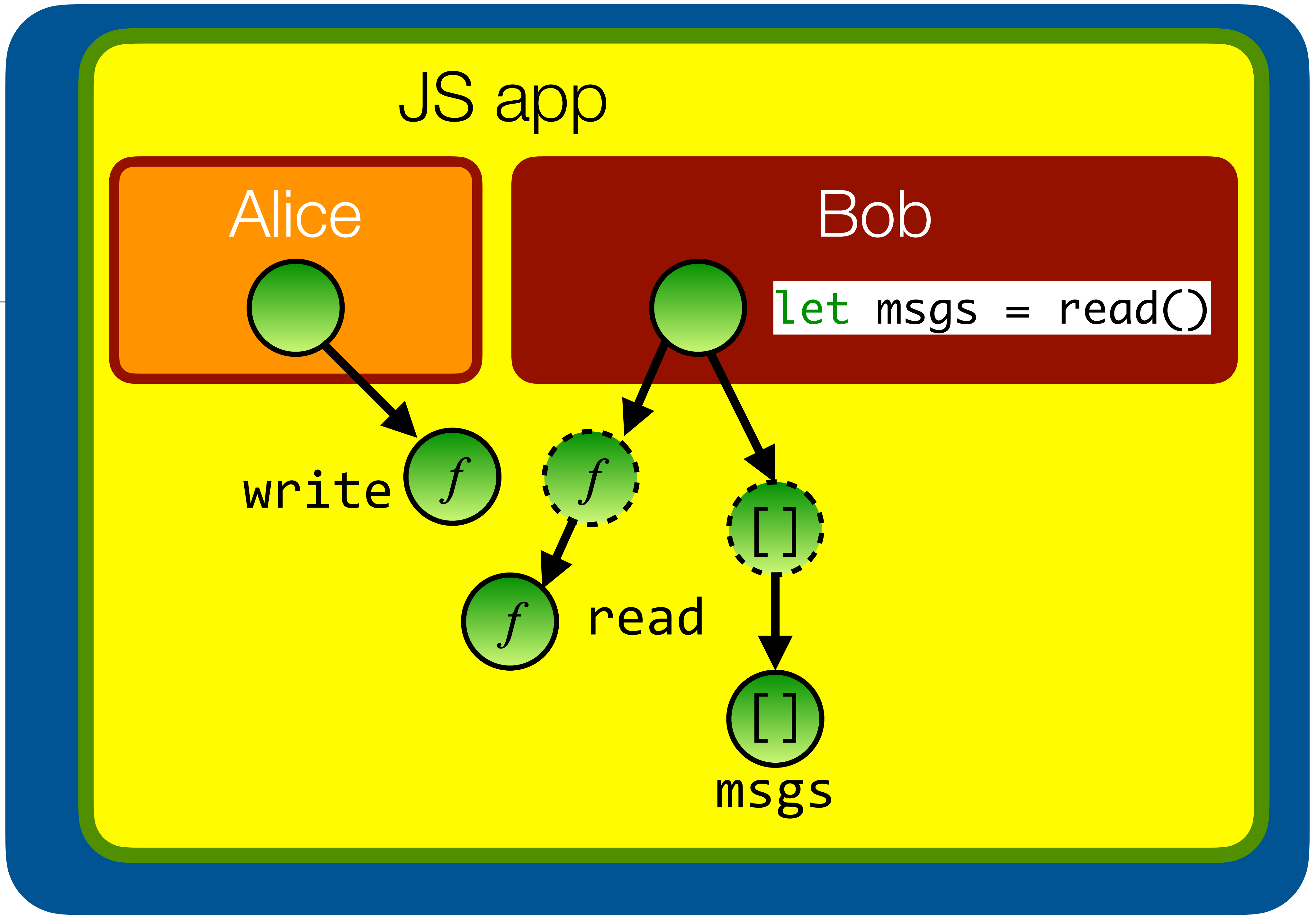


```
import * as alice from "alice.js";
import * as bob from “bob.js";

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 bob(log) {
  let msgs = log.read();
  …
}
```

Membranes are generalized caretakers

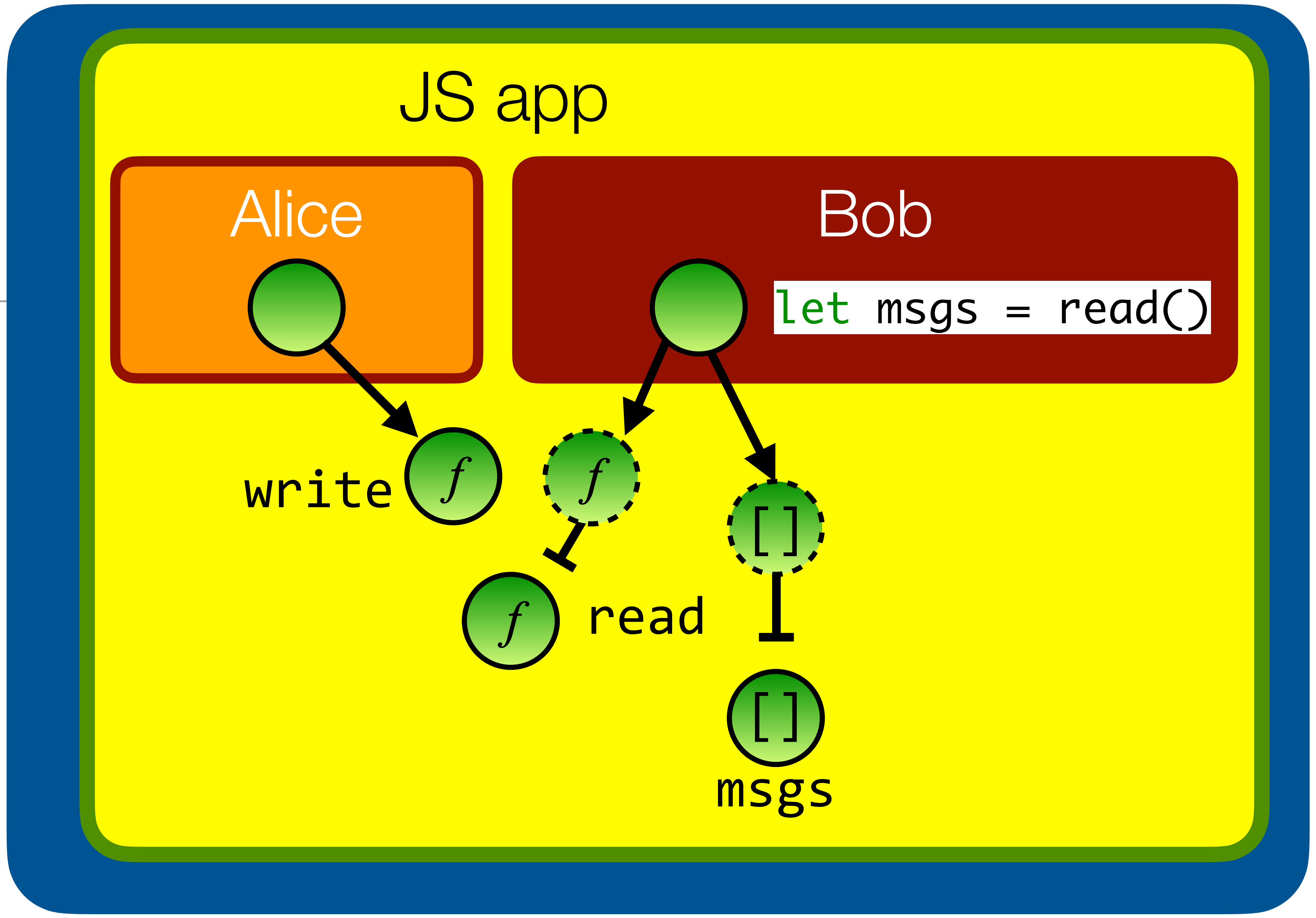
Proxy *any* object reachable from the log



```
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] = makeRevokableMembrane(log);
alice(log.write);
bob(rlog.read);
```

```
function bob(log) {
  let msgs = log.read();
  …
}
```

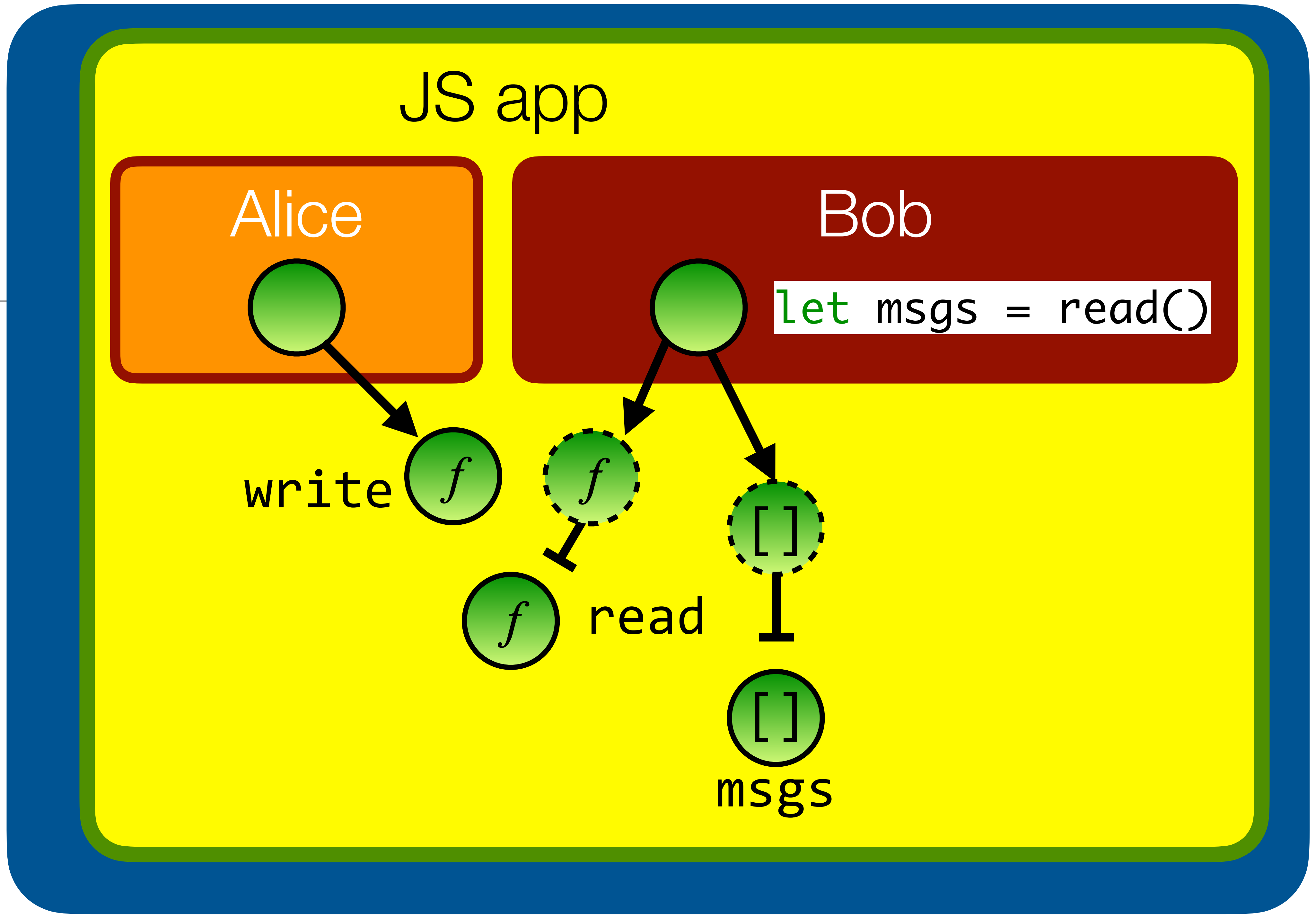
Membranes are generalized caretakers



```
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] = makeRevokableMembrane(log);
alice(log.write);
bob(rlog.read);

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

```
function bob(log) {
  let msgs = log.read();
  …
}
```

Membranes are generalized caretakers



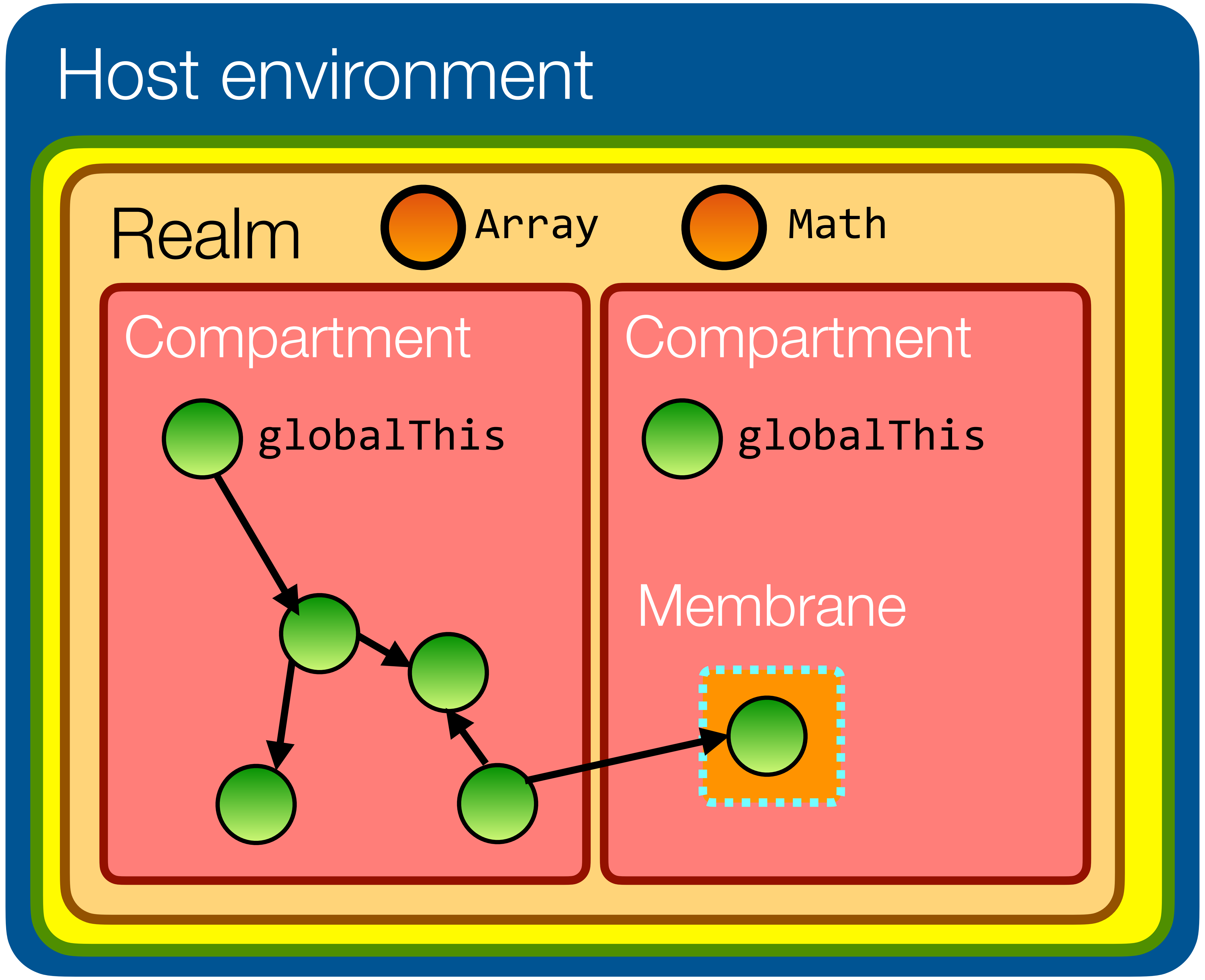
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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

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

```
function bob(log) {
  let msgs = log.read();
  …
}
```

Deep dive article at tvcutsem.github.io/membranes

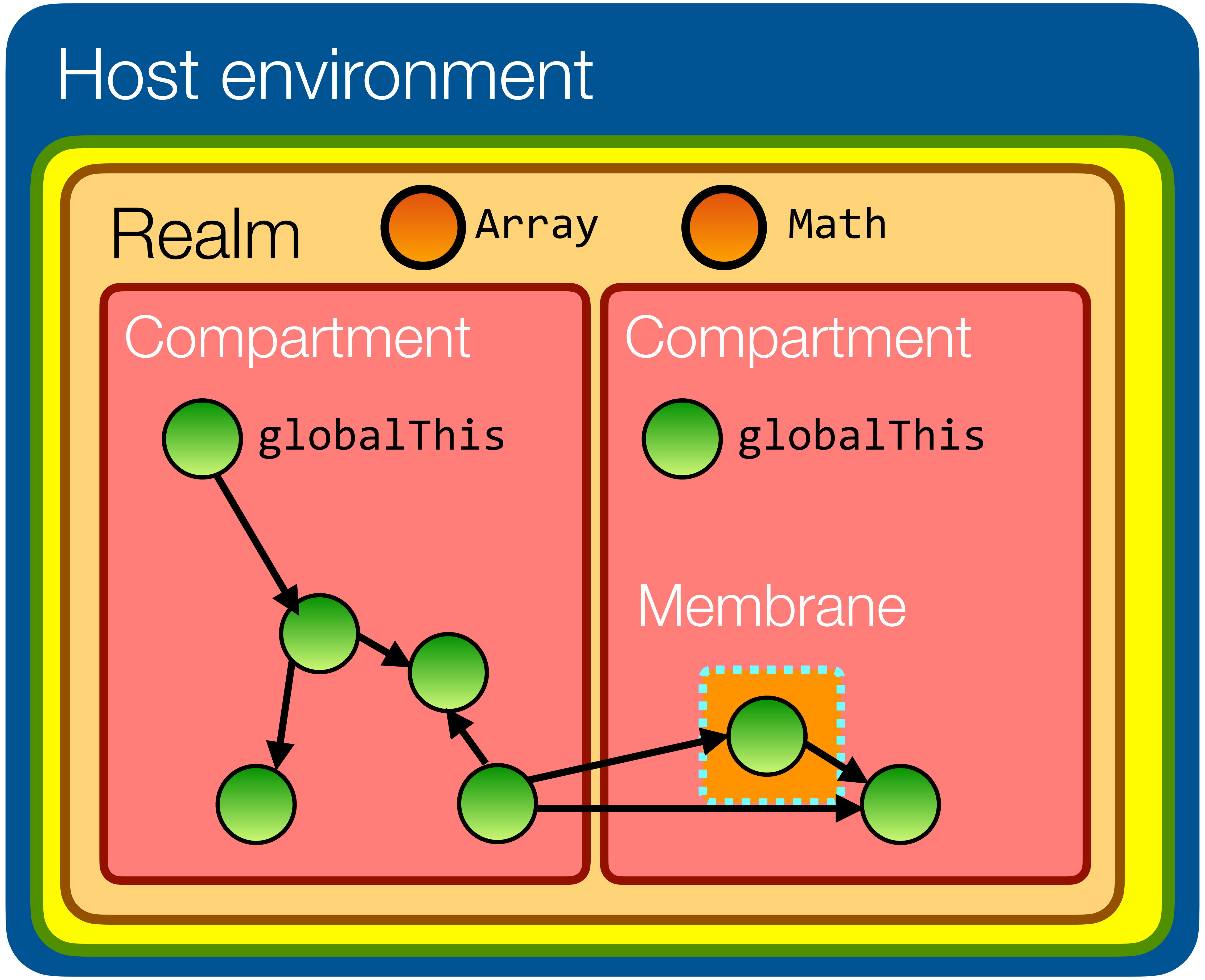
Compartments vs Membranes

- Compartments manage initial authority. Membranes manage subsequent interactions.



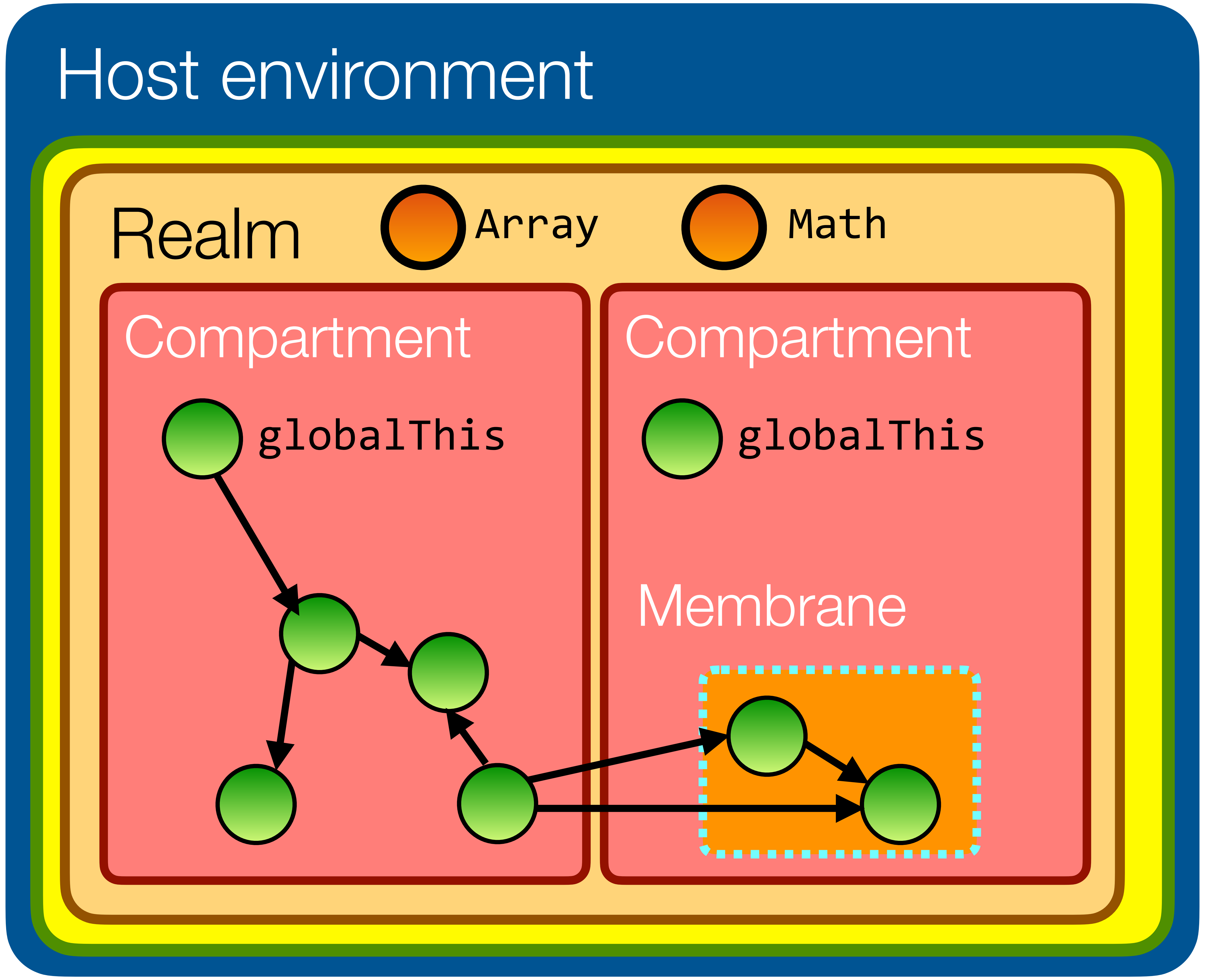
Compartments vs Membranes

- Compartments manage initial authority. Membranes manage subsequent interactions.



Compartments vs Membranes

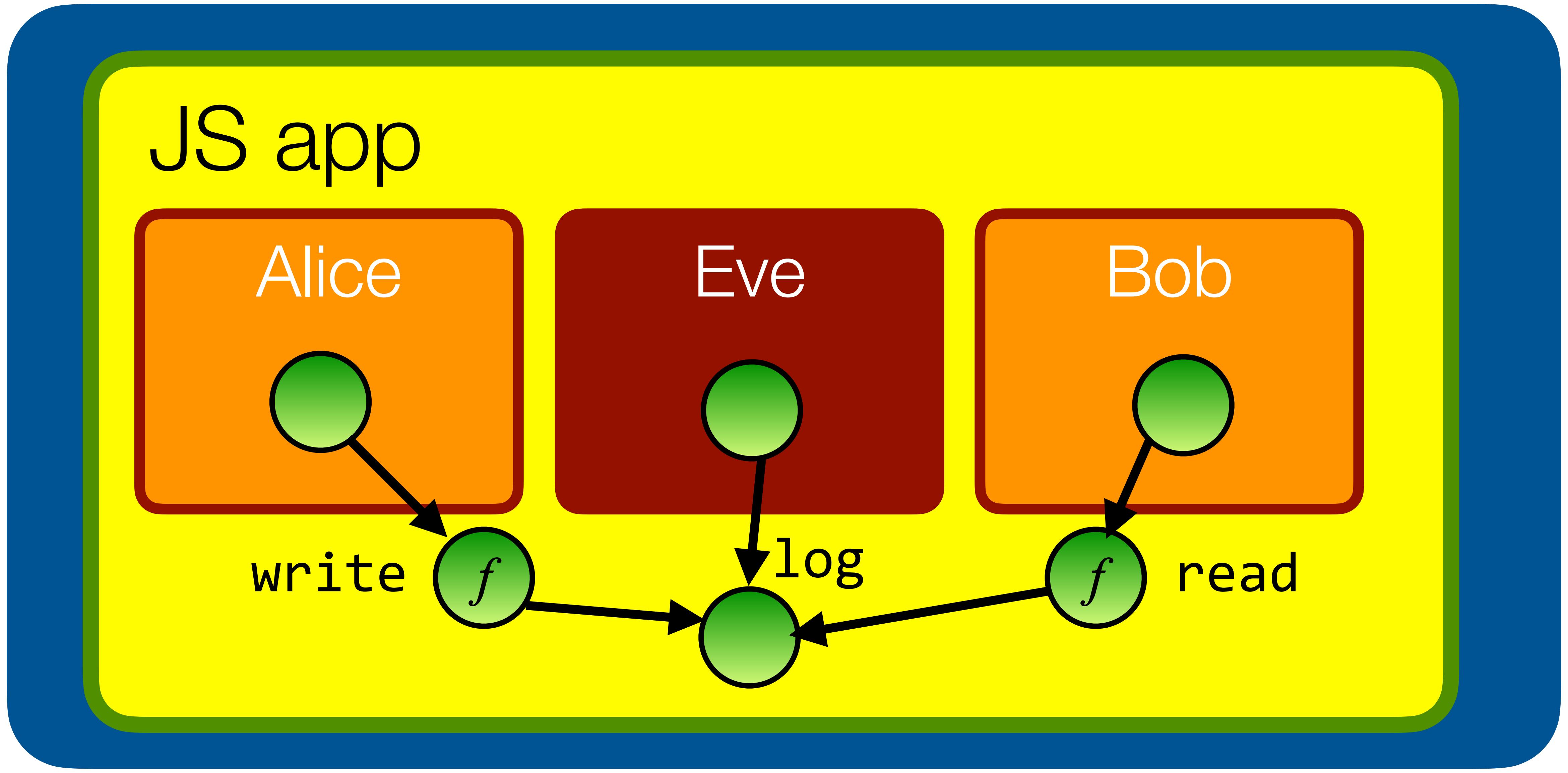
- Compartments manage initial authority. Membranes manage subsequent interactions.



Another exercise in POLA

- Eve needs access to the log as a whole, but we don’t want her to read or modify the *content* of the log

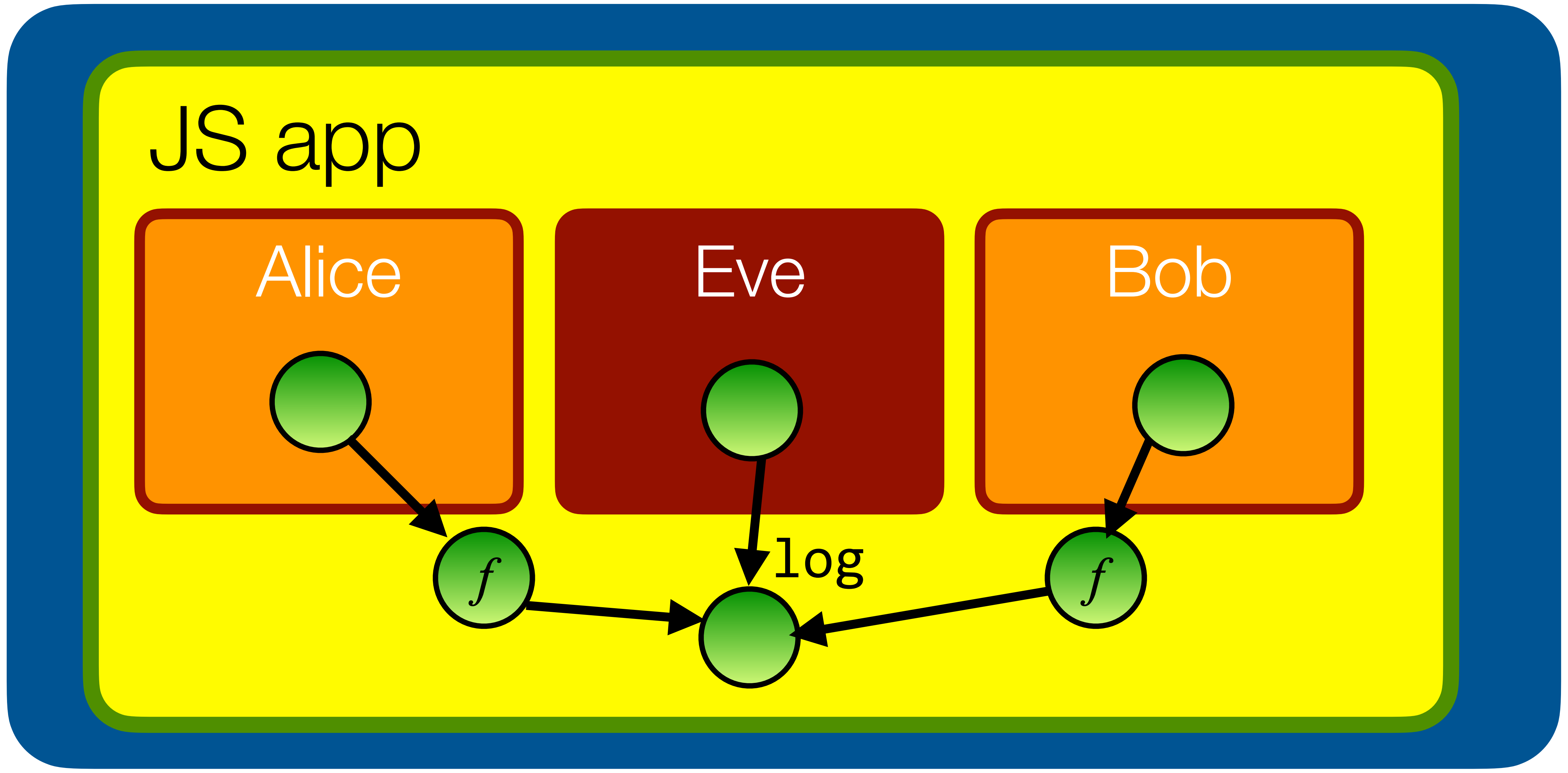
```
import * as alice from "alice.js";
import * as bob from “bob.js";
import * as eve from “eve.js";

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);
eve(log);
```



Sealer/unsealer pairs

- A sealer/unsealer pair enables the **confidentiality** and **integrity** properties of encryption, but in-process and without any actual cryptography

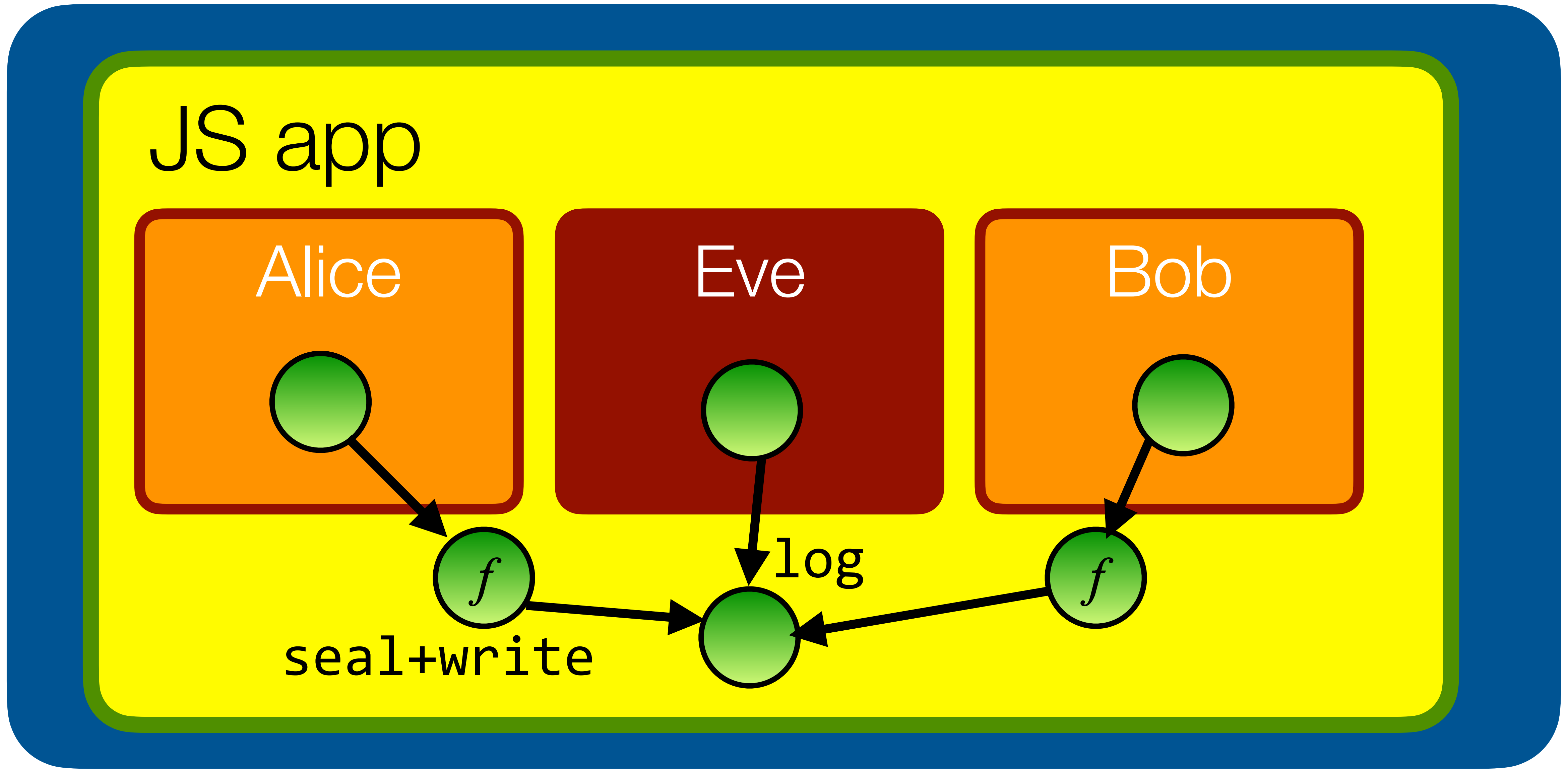
```
import * as alice from "alice.js";
import * as bob from “bob.js";
import * as eve from “eve.js";

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

let log = makeLog();
let [seal, unseal] = makeSealerUnsealerPair();
alice((msg) => log.write(seal(msg)));
bob(() => log.read().map(msg => unseal(msg)));
eve(log);
```



Sealer/unsealer pairs

- seal “encrypts” objects, unseal “decrypts” objects

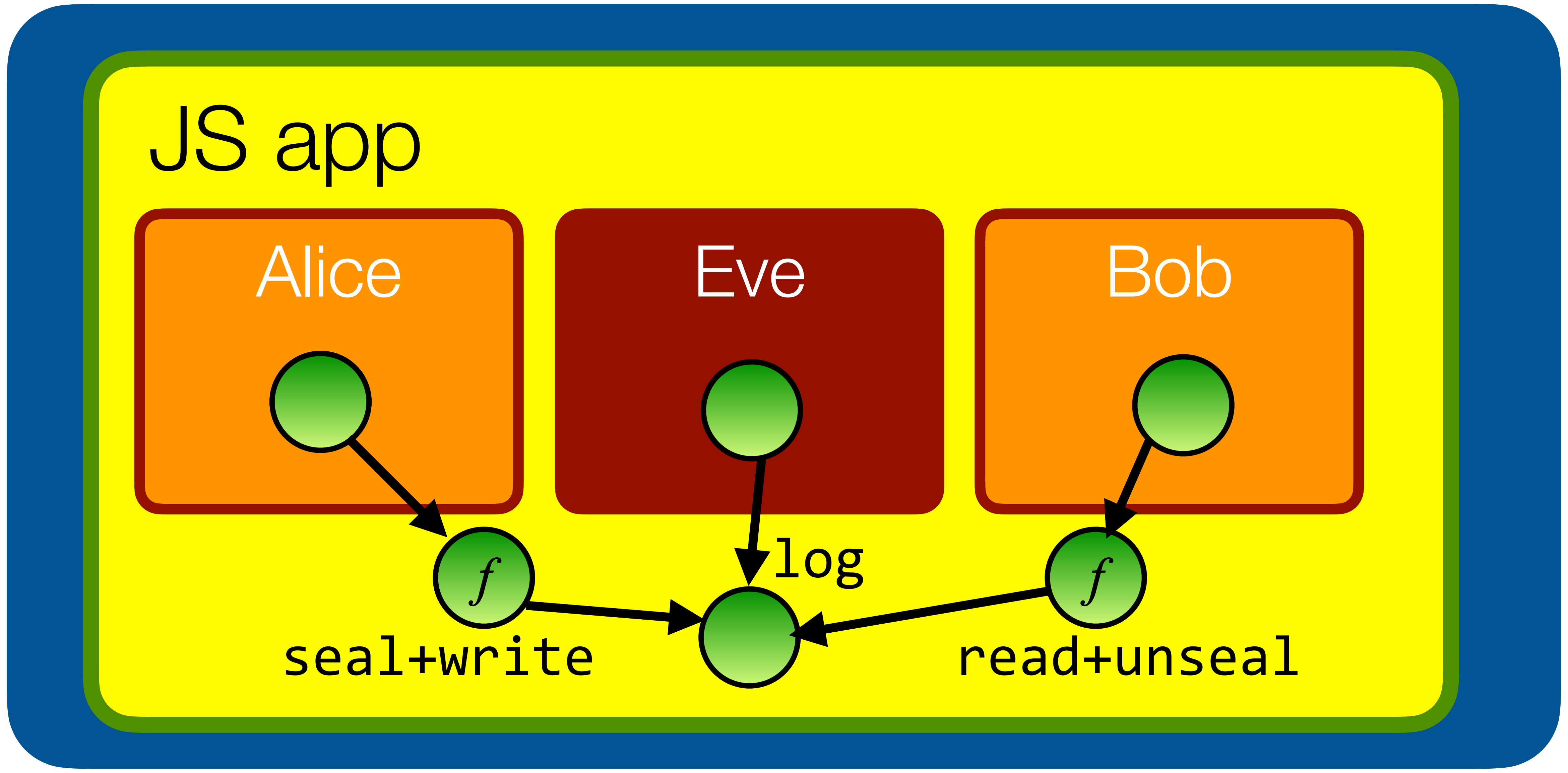
```
import * as alice from "alice.js";
import * as bob from “bob.js";
import * as eve from “eve.js";

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

let log = makeLog();
let [seal, unseal] = makeSealerUnsealerPair();
alice((msg) => log.write(seal(msg)));
bob(() => log.read().map(msg => unseal(msg));
eve(log);
```



Sealer/unsealer pairs

- seal “encrypts” objects, unseal “decrypts” objects

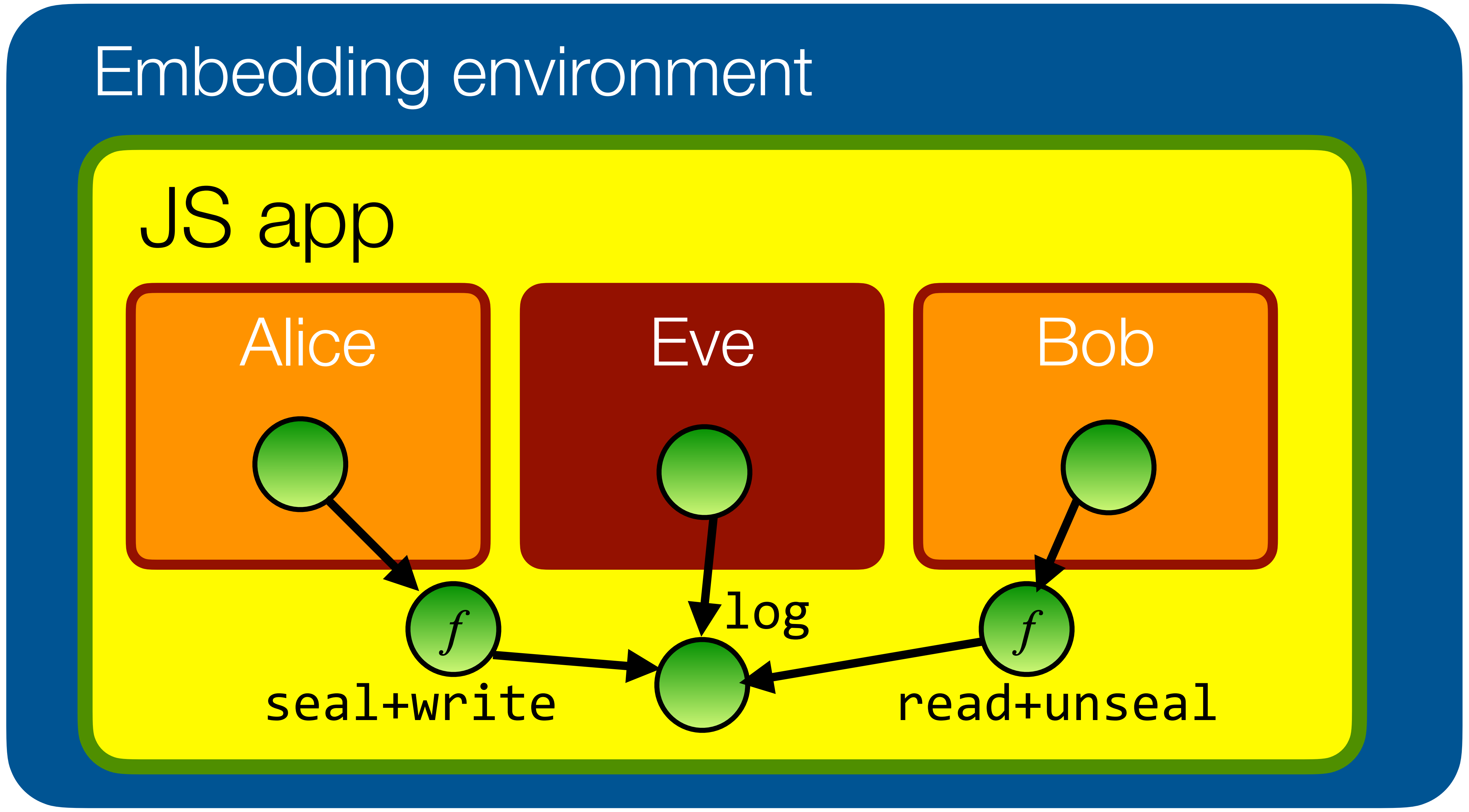
```
import * as alice from "alice.js";
import * as bob from “bob.js";
import * as eve from “eve.js";

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

let log = makeLog();
let [seal, unseal] = makeSealerUnsealerPair();
alice((msg) => log.write(seal(msg));
bob(() => log.read().map(msg => unseal(msg));
eve(log);
```



This is called “rights amplification”. It’s a useful POLA building block.

- Only code that has access to **both** the unseal function **and** the original object can access the sealed value

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

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

let log = makeLogger();
let [seal, unseal] = makeSealerUnsealerPair();
alice((msg) => log.write(seal(msg));
bob(() => log.read().map(msg => unseal(msg));
eve(log);
```



Object-capability 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** 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

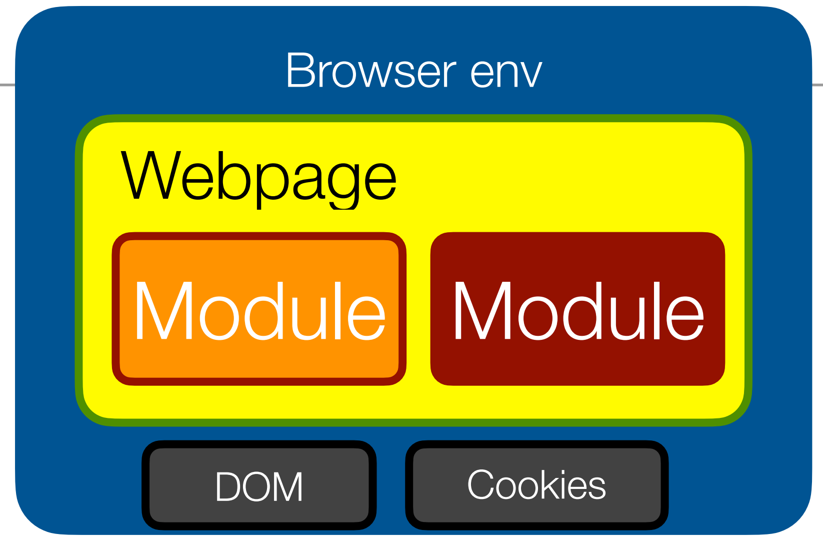
Salesforce Lightning

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

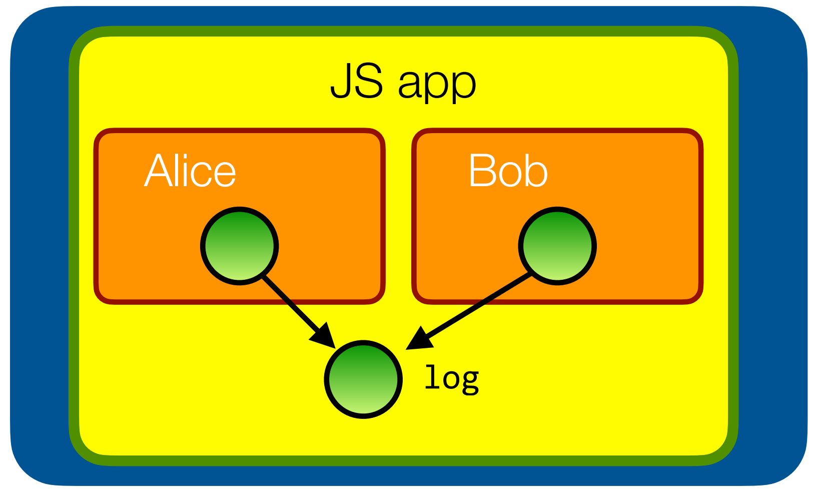
Summary

This Lecture

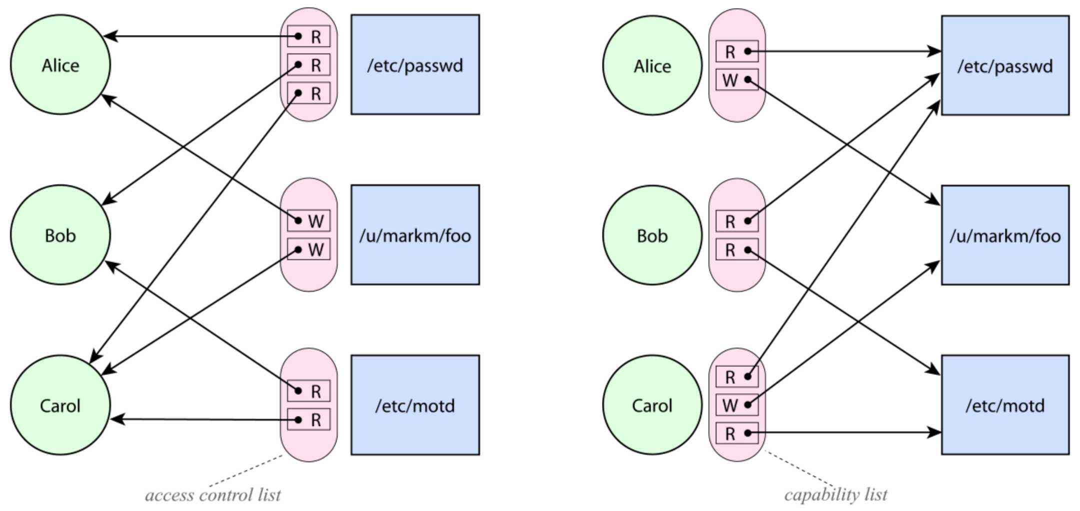
- Part I: why application security is critical to JavaScript applications



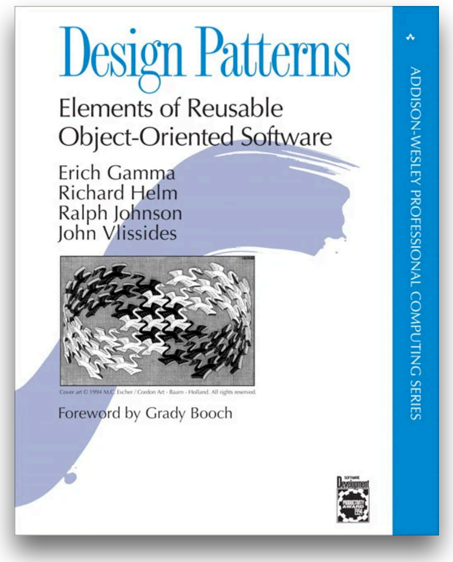
- Part II: the Principle of Least Authority, by example



- Part III: the object-capability model of access control



- Part IV: object-capability 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

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

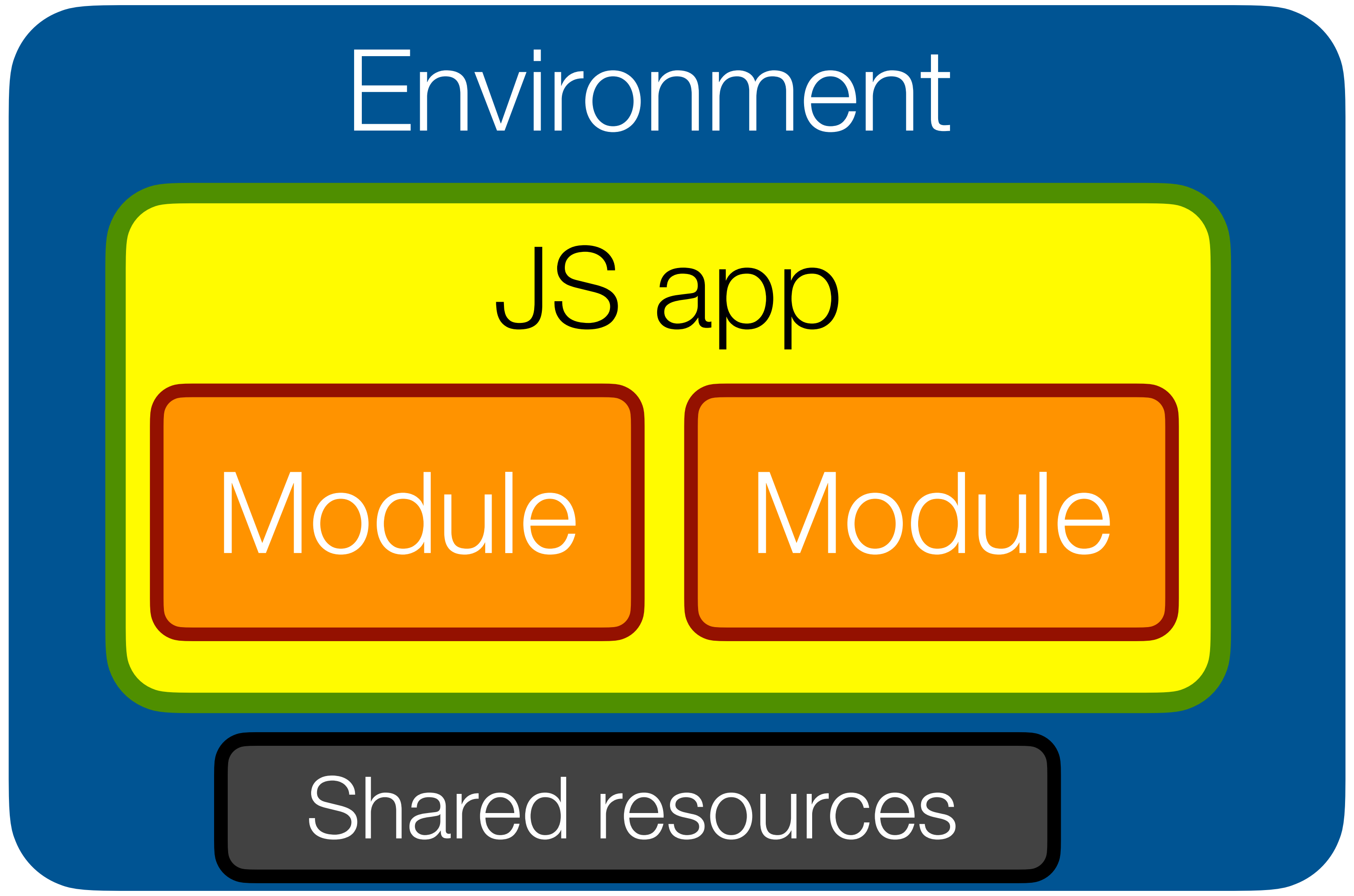
May, 2006

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.

The take-away messages

- Modern applications are **composed from many modules**. You can't trust them all.
- Apply the "principle of least authority" to **limit trust**.
  - **Isolate modules** (Hardened JS & Lavamoat)
  - Let modules **interact safely** using patterns such as facets, caretaker, membranes, taming, …
- Understanding these patterns is **important in a world of > 2,000,000 NPM modules.**
- Even more critical **in a Web3 world** where code starts to directly interact with digital assets



Appendix

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=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
- Chip Morningstar, “What are capabilities”: http://habitatchronicles.com/2017/05/what-are-capabilities/
- Why KeyKOS is fascinating: https://github.com/void4/notes/issues/41

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