Architecting Robust JavaScript Applications

Tom Van Cutsem

@tvcutsem

JSCONF.BE 2020

About me

- Computer scientist with broad experience in academia and industry
- Past TC39 member and active contributor to ECMAScript standards
- Author of Proxy and Reflect APIs
- Author of Traits.js
- Passionate user and advocate of JavaScript

tvcutsem.github.io


@tvcutsem


tvcutsem

A software architecture view of security

~~same-origin policy~~

~~iframe sandbox~~

~~principals~~

~~OAuth~~

~~cookies~~

~~content security policy~~

~~CORS~~

~~html sanitization~~

modules

objects

functions

visibility

dependencies

mutation

dataflow

A software architecture view of security

“Security is just the extreme of Modularity”

- Mark S. Miller

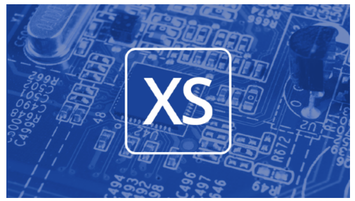
Modularity: avoid needless dependencies (to prevent bugs)
Security: avoid needless vulnerabilities (to prevent exploits)
Vulnerability is a form of dependency!

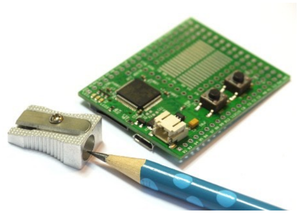
This Talk

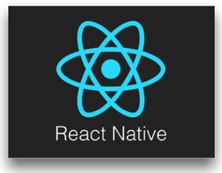
- Part I: why it’s becoming important to write more robust applications
- Part II: patterns that let you write more robust applications

Part I
The need for more robust JavaScript apps

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







Embedded Mobile Desktop/Native Server Database

ECMAScript: “Standard” JavaScript

V8

V8

Spidermonkey

GraalVM

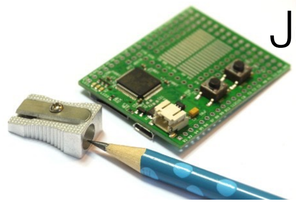
Nitro

Moddable XS

JerryScript

Espruino

Duktape



A Tale of Two Standards Bodies

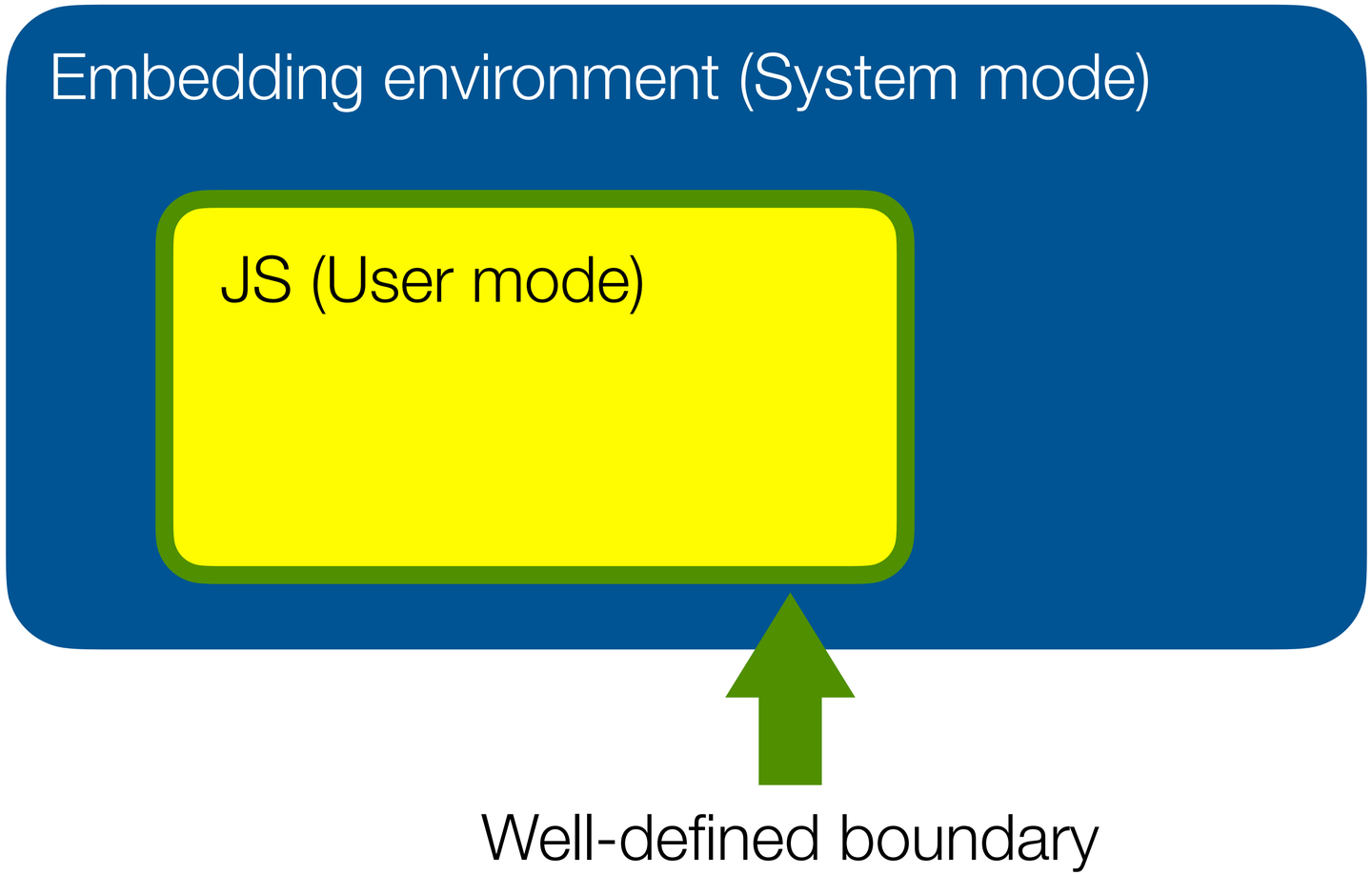
"Any organization that designs a system […] will produce a design whose structure is a copy of the organization's communication structure."

-- Melvyn Conway, 1967

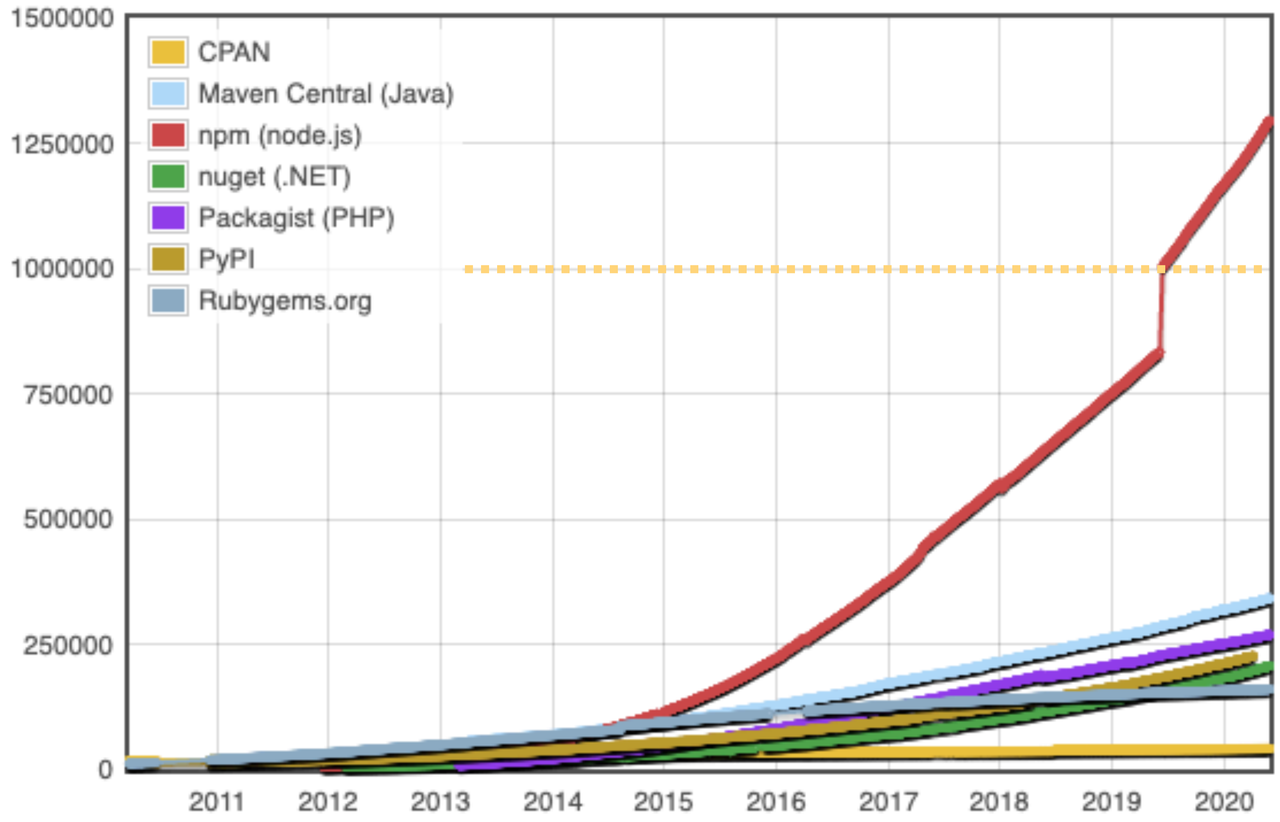
- Standardizes JavaScript
- Core language + small standard library
- Math, JSON, String, RegExp, Array, …
- **“User mode”**

- Standardizes browser APIs
- Large set of system APIs
- DOM, LocalStorage, XHR, Media Capture, …
- **“System mode”**

“User mode” separation makes JS an embeddable compute engine



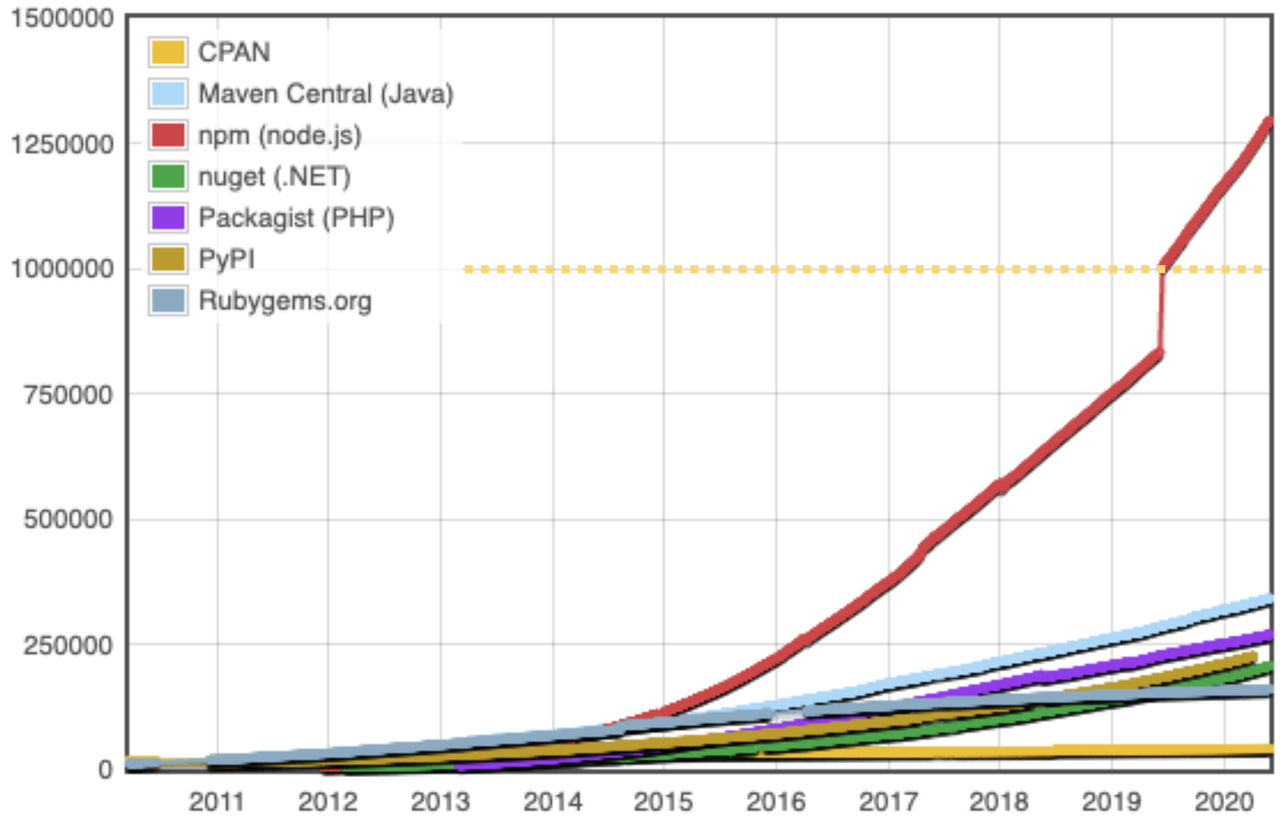
JavaScript applications are now built from thousands of modules



1,000,000 modules on NPM

(source: modulecounts.com, May 2020)

JavaScript applications are now built from thousands of modules



1,000,000 modules on NPM

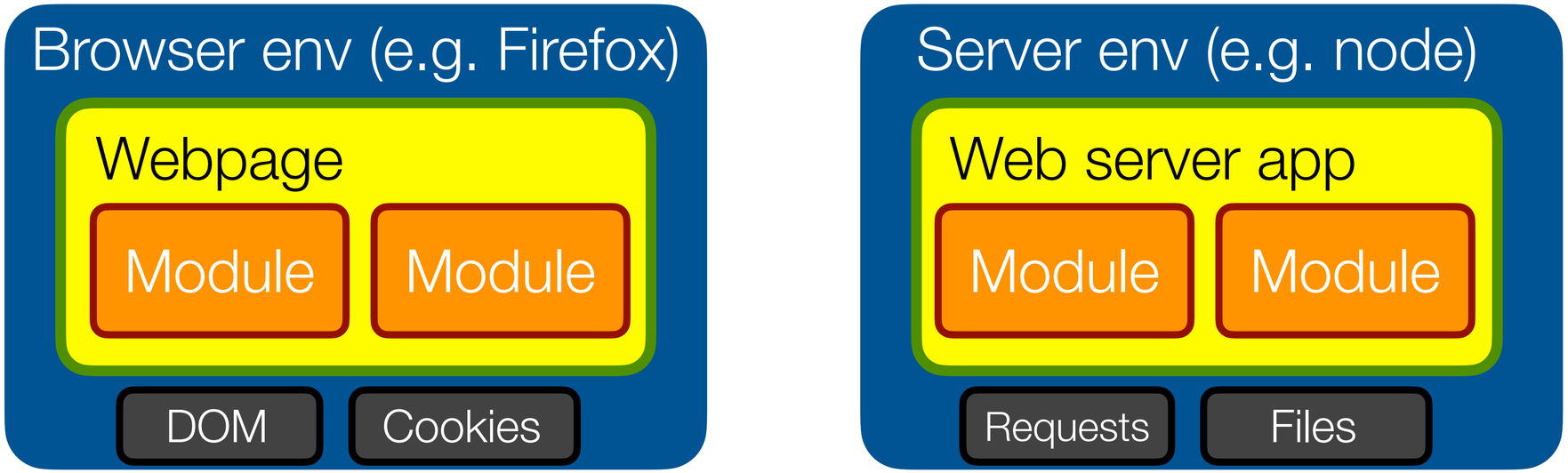
"The average modern web application has over 1000 modules […] ***97% of the code in a modern web application comes from npm***. An individual developer is responsible only for the final 3% that makes their application unique and useful."

(source: npm blog, December 2018)

(source: modulecounts.com, May 2020)

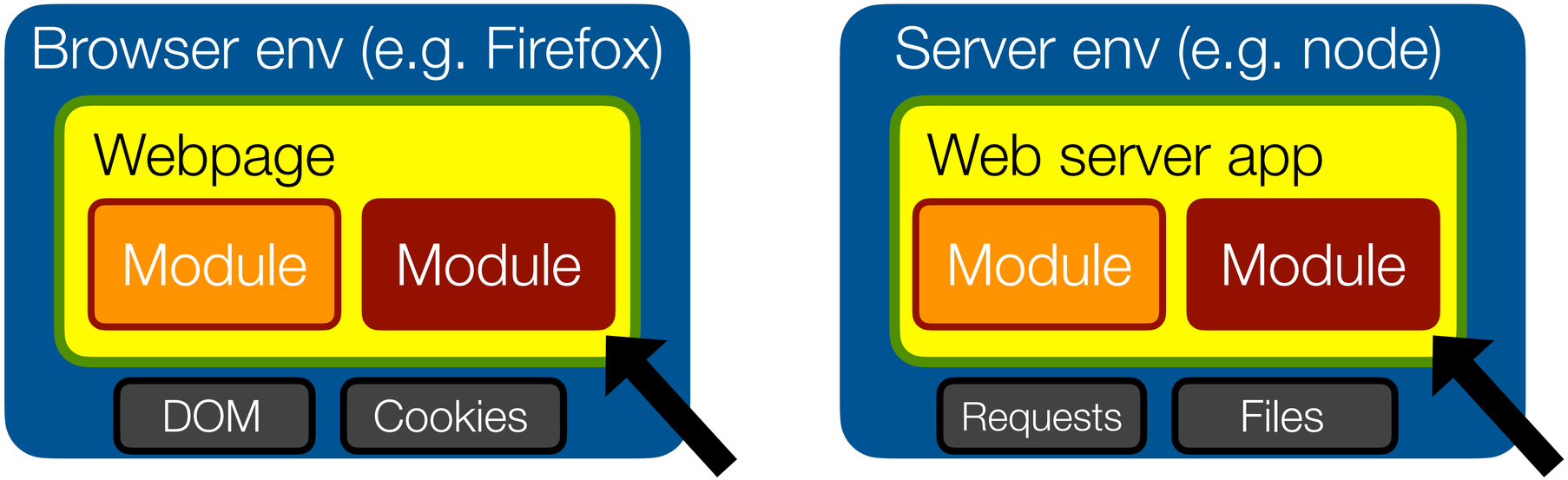
It’s all about **trust**

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



It’s all about **trust**

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

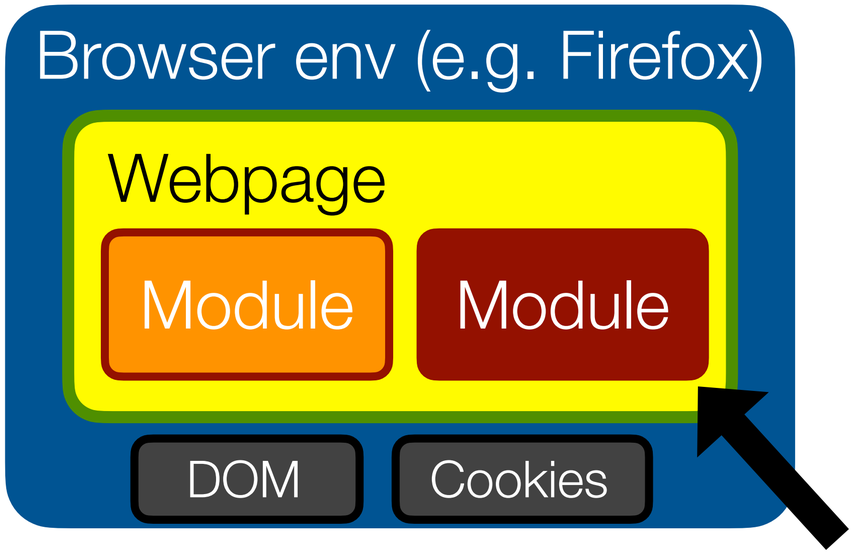



```
<script src=“http://evil.com/ad.js”>
```

```
npm install evil-logger
```

It’s all about **trust**

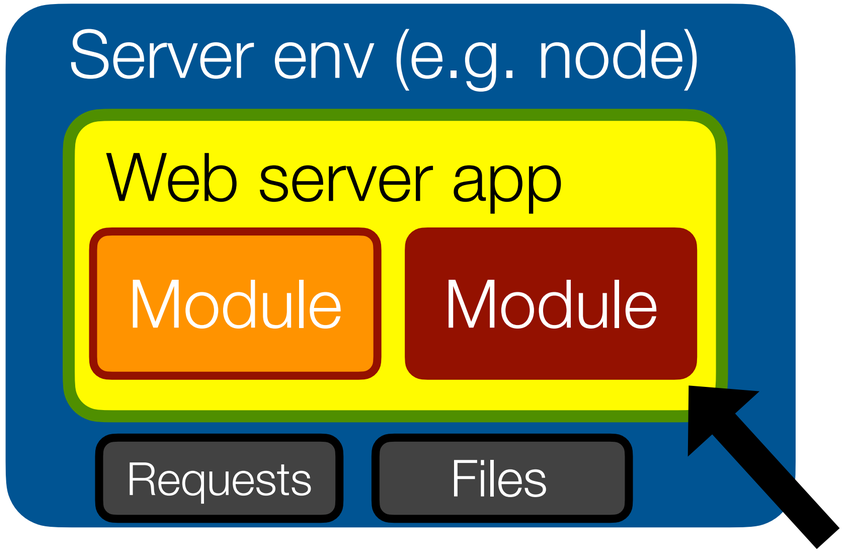
- It is exceedingly common to run code you don’t know/trust in a common environment




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

It’s all about **trust**

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




```
npm install evil-logger
```

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

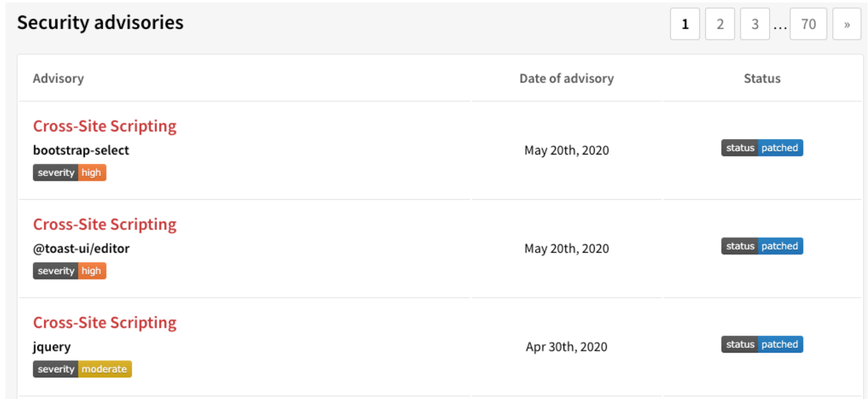
By Thomas Claburn in San Francisco 26 Nov 2018 at 20:58 49 SHARE ▼

(source: theregister.co.uk)

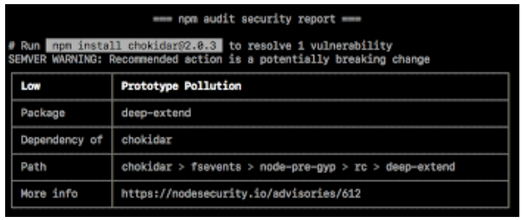
Increasing awareness

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

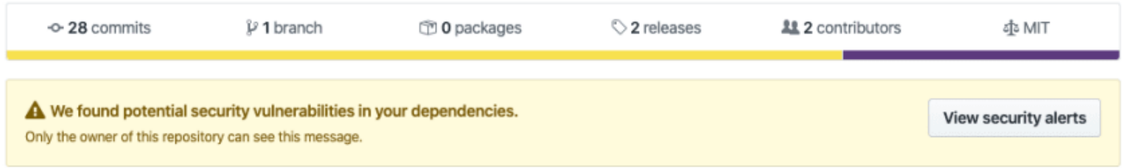
npm security advisories



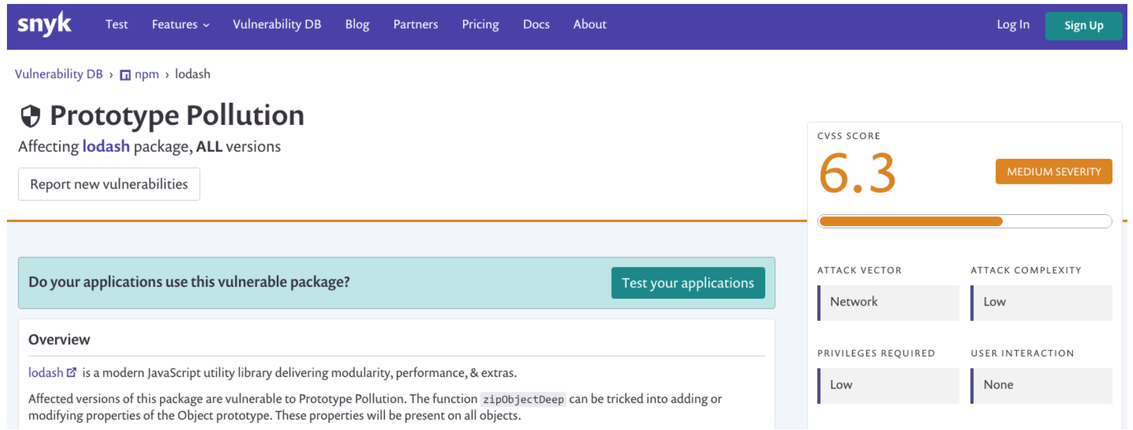
npm audit



GitHub security alerts

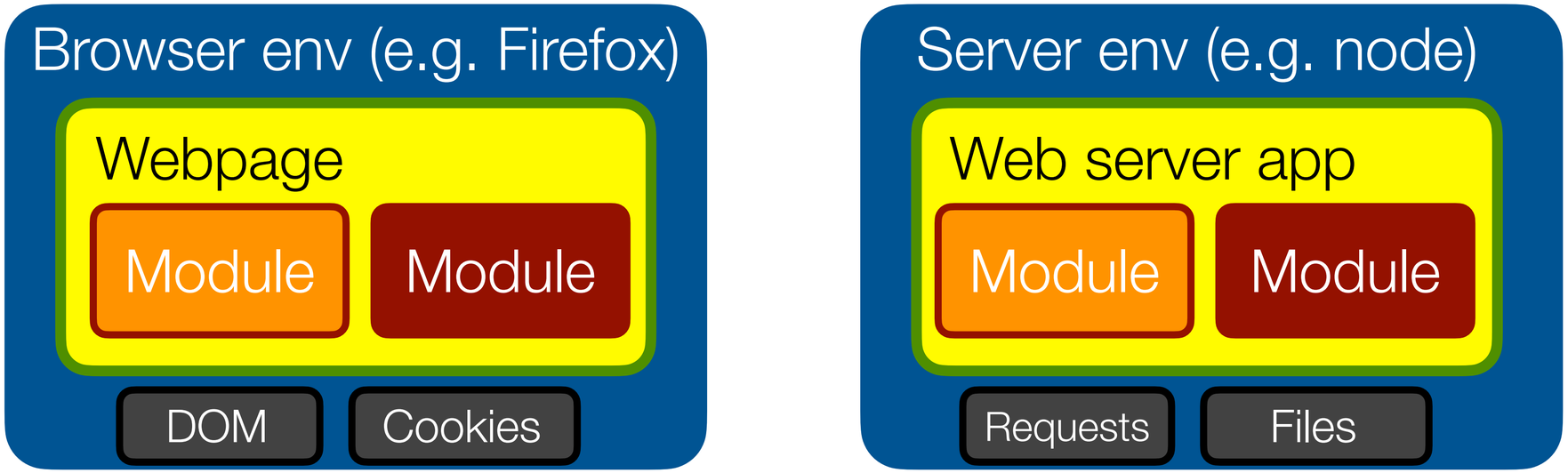


Snyk vulnerability DB



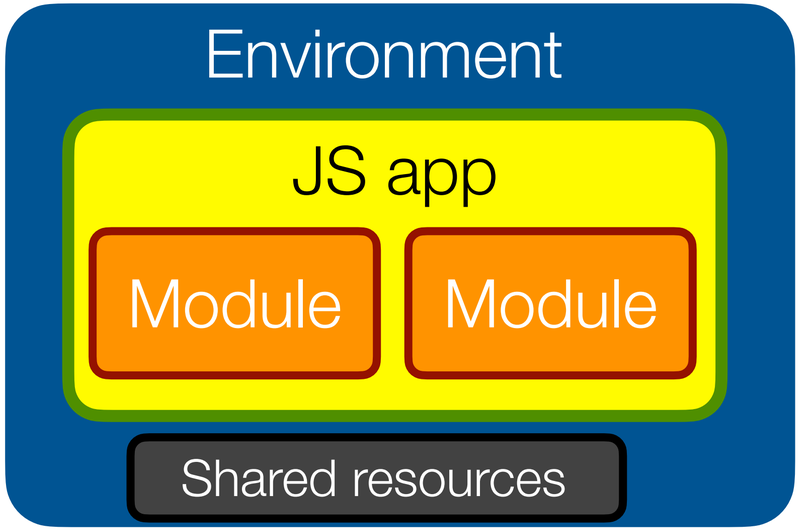
Avoiding interference is the name of the game

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



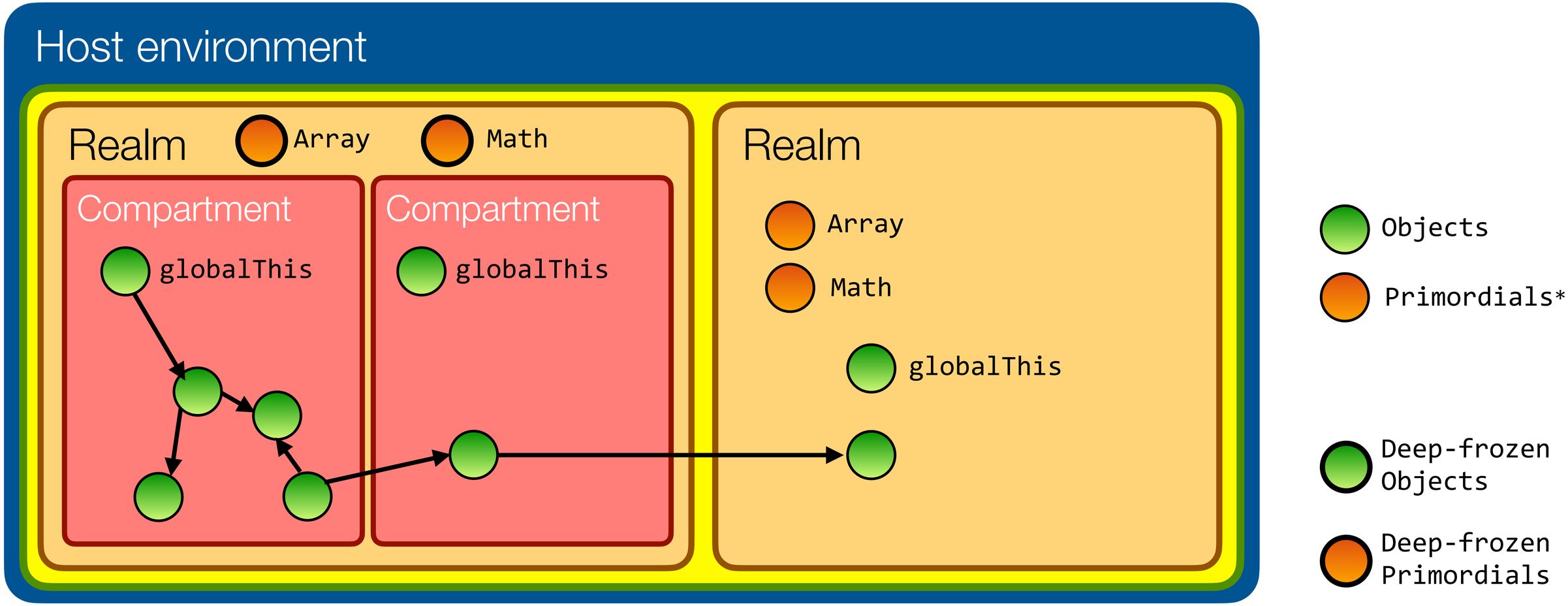
Prerequisite: isolating JavaScript modules

- Today: JavaScript offers no “User mode” isolation mechanisms
- Lots of “System mode” isolation mechanisms, but non-portable:
  - **Web Workers**: forced async communication, no shared memory
  - **iframes**: mutable primordials, “identity discontinuity”
  - **node** `vm` **module**: same issues (note: prefer `vm2` npm module)



Realms and Compartments: “User mode” isolation

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



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

Realms and Compartments: “User mode” isolation

Realms

```
let r = new Realm();
r.globalThis.x = 1;
let res = r.globalThis.eval(`x + 1`);


// fails, no non-standard globals
r.globalThis.eval(`process.exit(0)`);


// does not affect outer Realm’s Array
r.globalThis.eval(
  `Array.prototype.push = undefined`);


let arr = r.globalThis.eval(`[]`);
arr instanceof Array // => false
```

Compartments

```
let c = new Compartment({x: 1})
let res = c.evaluate(`x + 1`) // => 3


// fails, no non-standard globals
c.evaluate(`process.exit(0)`);


// fails, primordials are immutable
c.evaluate(
  `Array.prototype.push = undefined`);


let arr = c.evaluate(`[]`)
arr instanceof Array; // => true
```

Realms and Compartments: “User mode” isolation

Realms

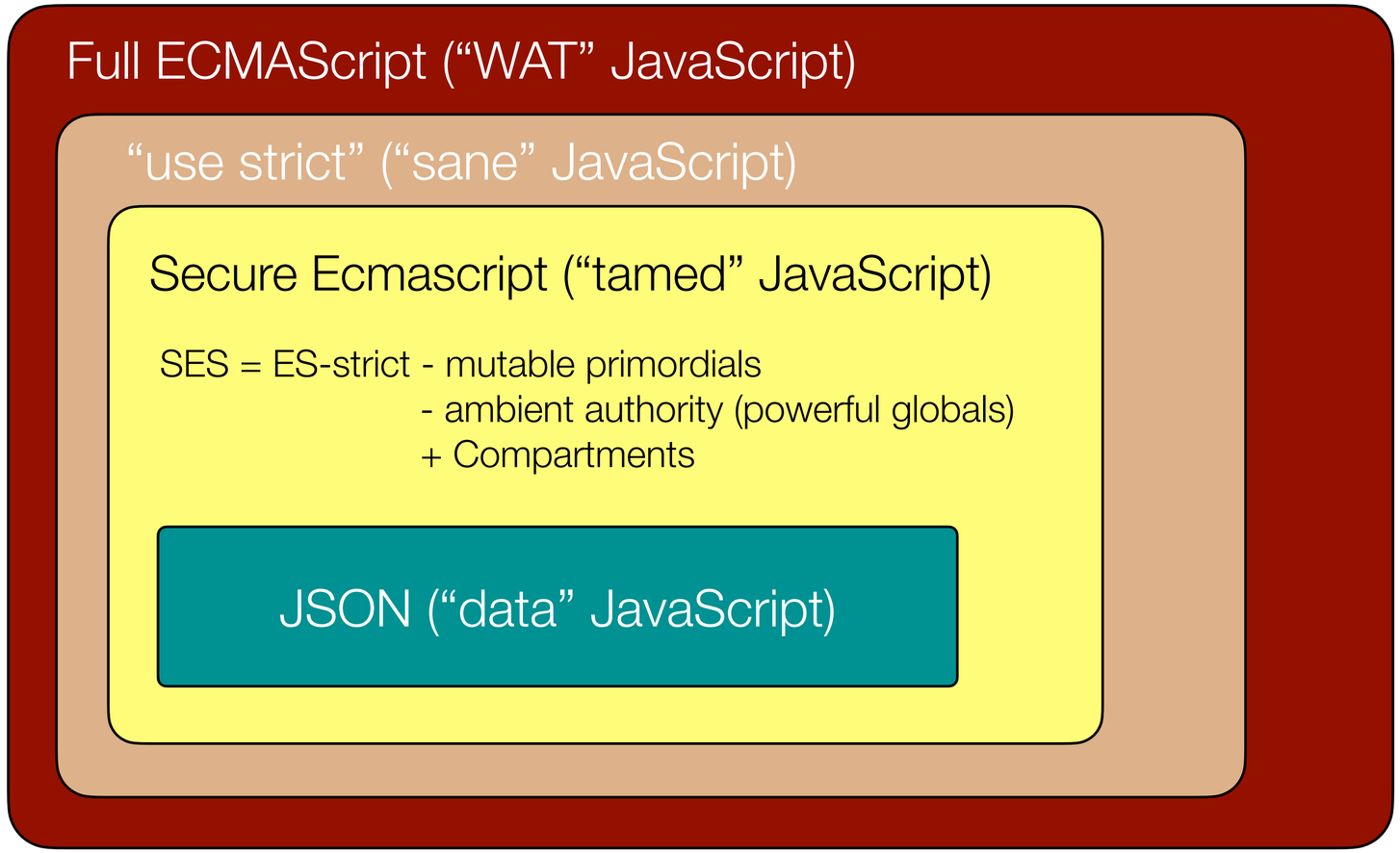
- Each realm has its own set of mutable primordials
- Useful for sandboxing “legacy” code that mutates primordials

- TC39 Stage 2: https://github.com/tc39/proposal-realms/
- Shim available at github.com/Agoric/realms-shim

Compartments

- Each compartment shares a set of immutable and powerless primordials
- Preferred for well-behaved code. More lightweight than Realms.
- No “identity discontinuity” between compartments.
- Compartments have “hooks” to customize module imports (e.g. load each module in own compartment)
- TC39 Stage 1: https://github.com/tc39/proposal-compartments
- Shim available at https://github.com/Agoric/ses-shim

Secure ECMAScript (SES)



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

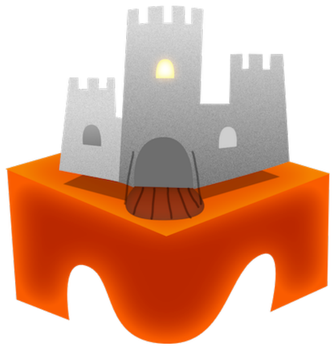
- A subset of JavaScript, building on Compartments
- Key idea: no powerful objects by default. SES code can only affect the outside world through objects (capabilities) explicitly granted to it (**POLA**)

```
import ‘ses’;
lockdown();
```

- TC39 Stage 1: https://github.com/tc39/proposal-ses

LavaMoat

- Build tool that puts each of your app's package dependencies into its own SES sandbox
- Auto-generates config file indicating authority needed by each package
- Plugs into Webpack and Browserify

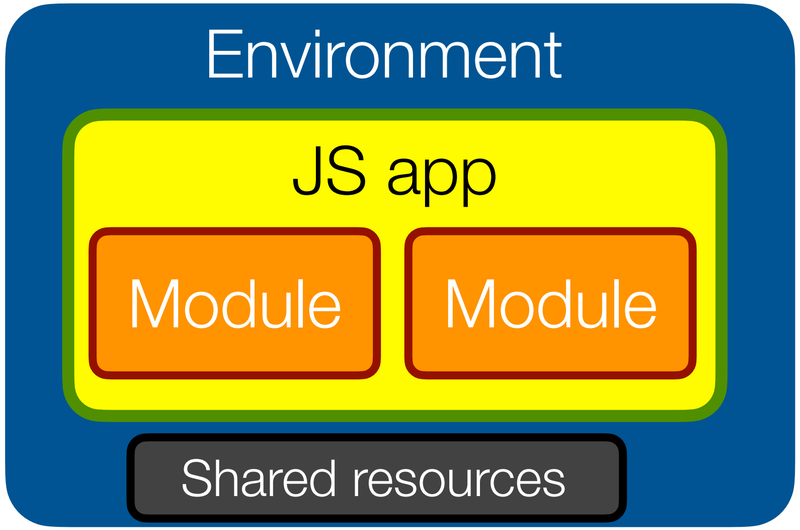


https://github.com/LavaMoat/lavamoat


```
"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
  }
},
```

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. SES adds basic isolation.
- **Isolated modules must still interact!**
- Design patterns exist to **compose modules** in ways that minimize unwanted interactions.
- Going forward: assume all code running in Secure ECMAScript environment



Part II
Robust Application Design Patterns

Design Patterns

Visitor

Factory

Observer

Singleton

State

Design Patterns for secure cooperation

Defensible object

Sealer/unsealer pair

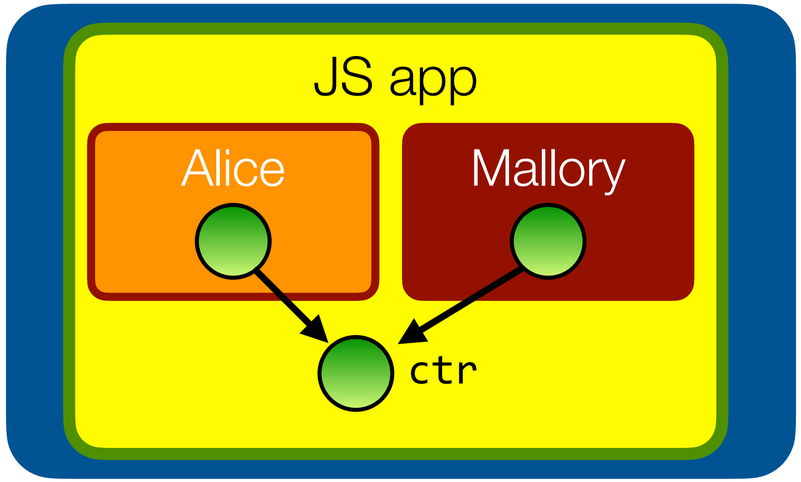
Reliable branding

API Taming

Membrane

#1: make private state truly private

```
class Counter {
  constructor() {
    this.count_ = 0;
  }
  incr() { return ++this.count_; }
  decr() { return --this.count_; }
}

let ctr = new Counter();
ctr.count_ // 0
```

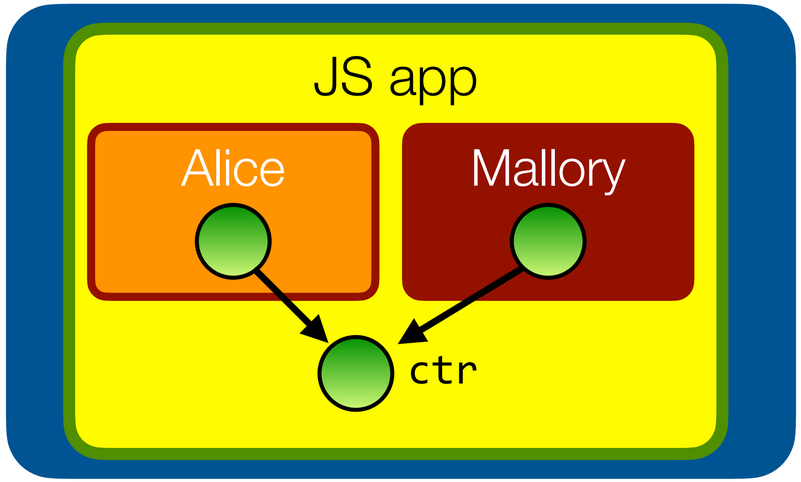



```
let aliceMod = /* load alice's code */
let malloryMod = /* load mallory's code */
aliceMod(ctr);
malloryMod(ctr);
```

#1: make private state truly private

- Private fields (TC39 Stage 3 proposal)

```
class Counter {
  #count = 0;

  incr() { return ++this.#count; }
  decr() { return --this.#count; }
}

let ctr = new Counter();
ctr.#count // error
```

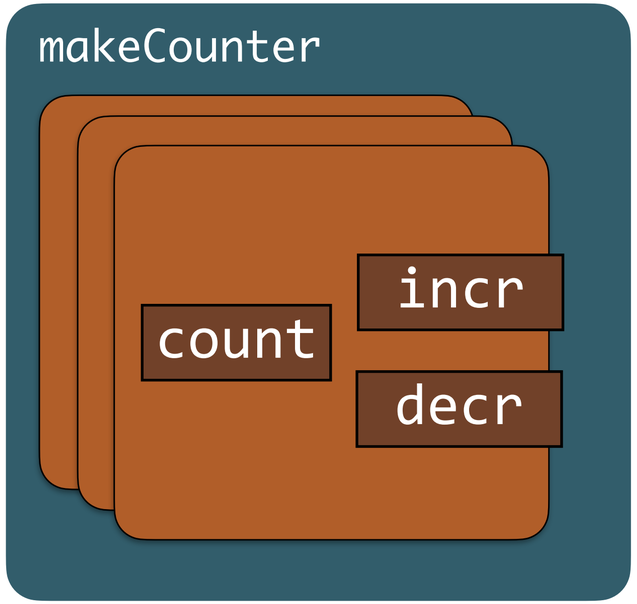



```
let aliceMod = /* load alice's code */
let malloryMod = /* load mallory's code */
aliceMod(ctr);
malloryMod(ctr);
```

(https://github.com/tc39/proposal-class-fields)

#1: hide mutable state through closure

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

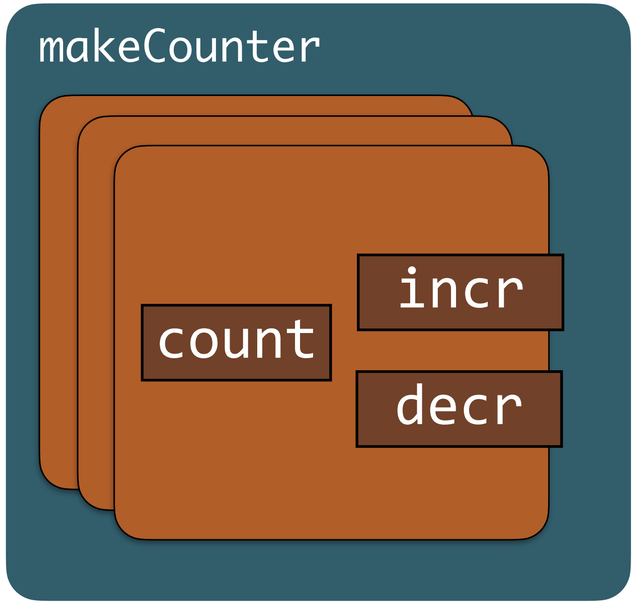



```
function makeCounter() {
  let count = 0;
  return {
    incr() { return ++count; },
    decr() { return --count; }
  }
}

let ctr = makeCounter();
ctr.count // undefined
```

(source: Mark S. Miller, "bringing object-orientation to security programming")

#2: make objects tamper-proof by freezing them

- Javascript objects are mutable records: any field can be overwritten by any of its clients (intentionally or unintentionally)

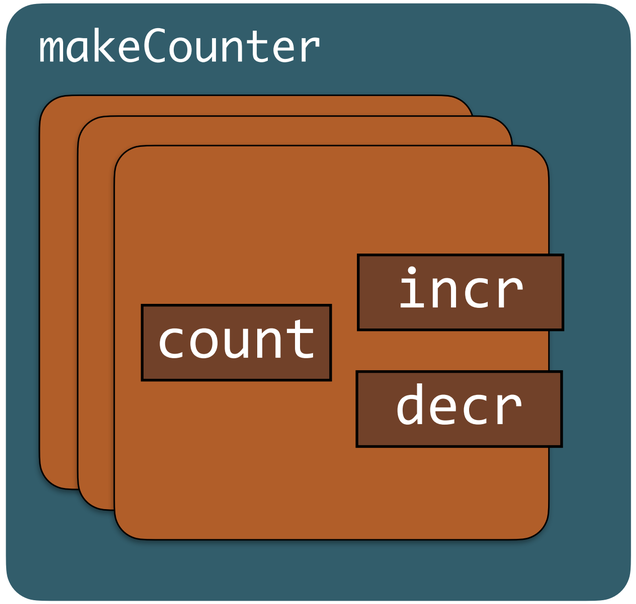



```
function makeCounter() {
  let count = 0;
  return Object.freeze({
    incr() { return ++count; },
    decr() { return --count; }
  })
}

let ctr = makeCounter();
ctr.incr = ctr.decr; // error
```

(source: Mark S. Miller, "bringing object-orientation to security programming")

#2: make objects tamper-proof by freezing them

- Note: freezing an object does not transitively freeze any objects/functions reachable from the object. Full tamper-proofing requires a ‘deep-freeze’
- SES provides such a ‘deep-freeze’ function called “harden”

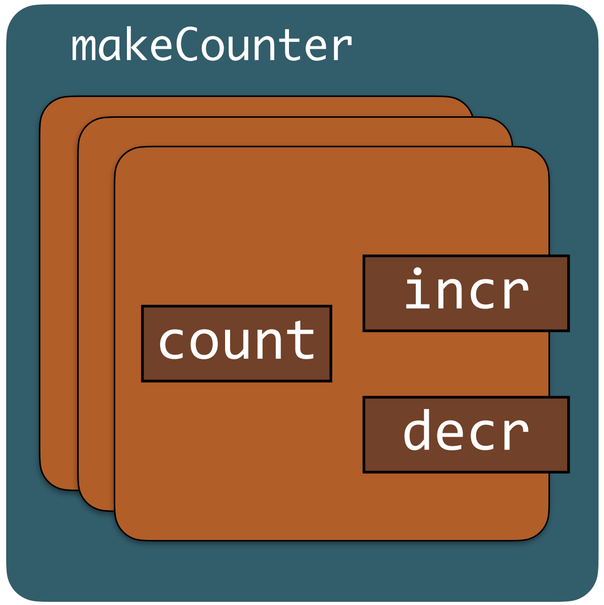



```
import ‘ses’;
lockdown()

function makeCounter() {
  let count = 0;
  return harden({
    incr() { return ++count; },
    decr() { return --count; }
  })
}

let ctr = makeCounter();
ctr.incr.apply = function() {…}; // error
```

(source: Mark S. Miller, “bringing object-orientation to security programming”)

#3: safe monkey-patching

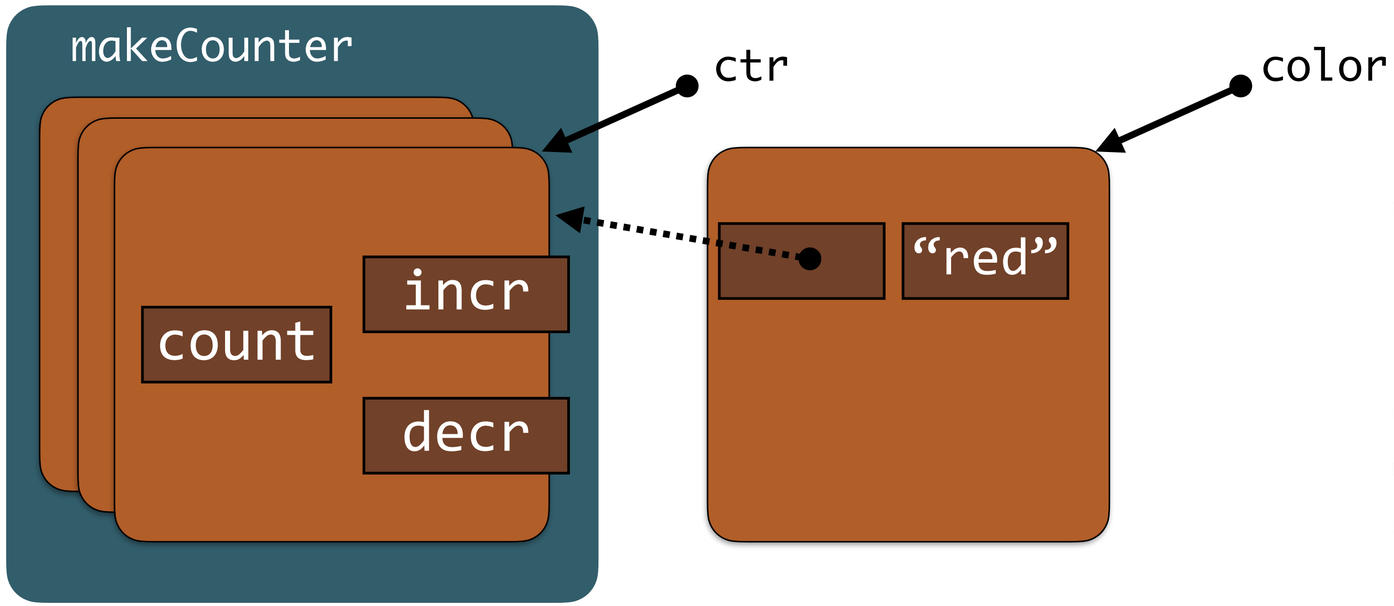
- It is common for one module to want to “expand” the objects of another module with new properties.
- Common practice today: **monkey-patching**




```
ctr.color = “red”;
```

#3: safe monkey-patching using WeakMaps

- WeakMaps can store new properties **without mutating** the original objects
- Unlike traditional monkey-patching, also works for frozen objects

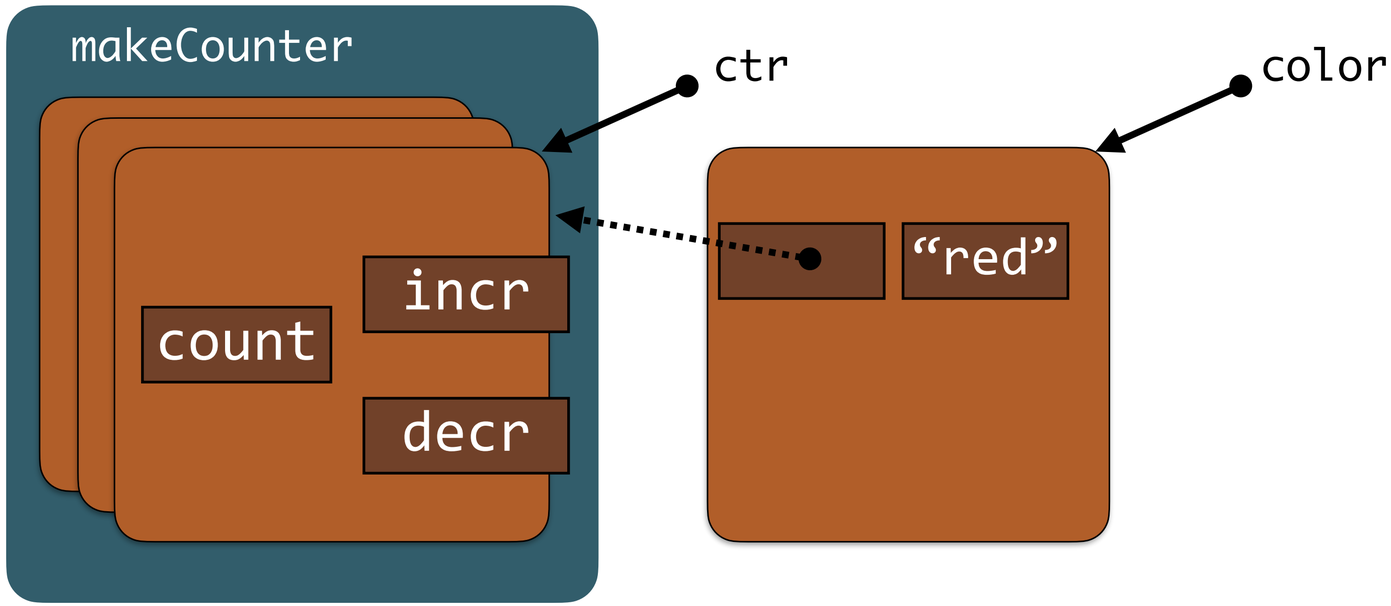



```
ctr.color = “red”;
```

```
const ctr = makeCounter();
const color = new WeakMap();
color.set(ctr, “red”);
color.get(ctr); // “red”
```

#3: safe monkey-patching using WeakMaps

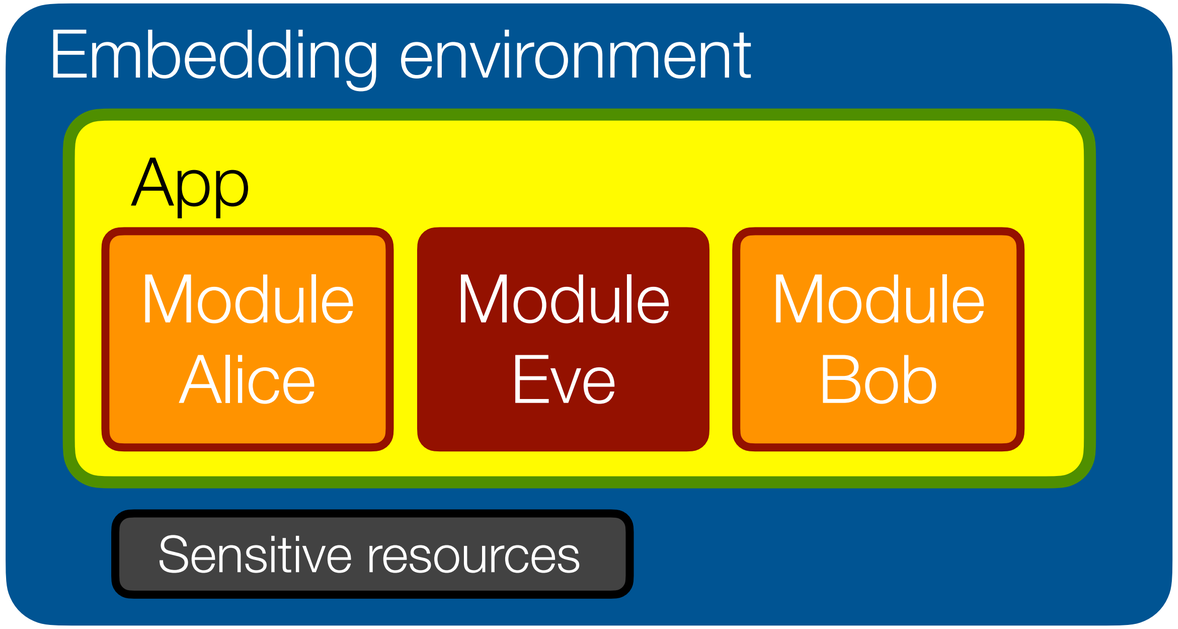
- **Bonus**: only code that has access to **both** the WeakMap and the original object can access the value
- “rights amplification”




```
ctr.color = “red”;

const ctr = makeCounter();
const color = new WeakMap();
color.set(ctr, “red”);
color.get(ctr); // “red”
```

#4: use sealer/unsealer pairs to “encrypt” objects with no crypto

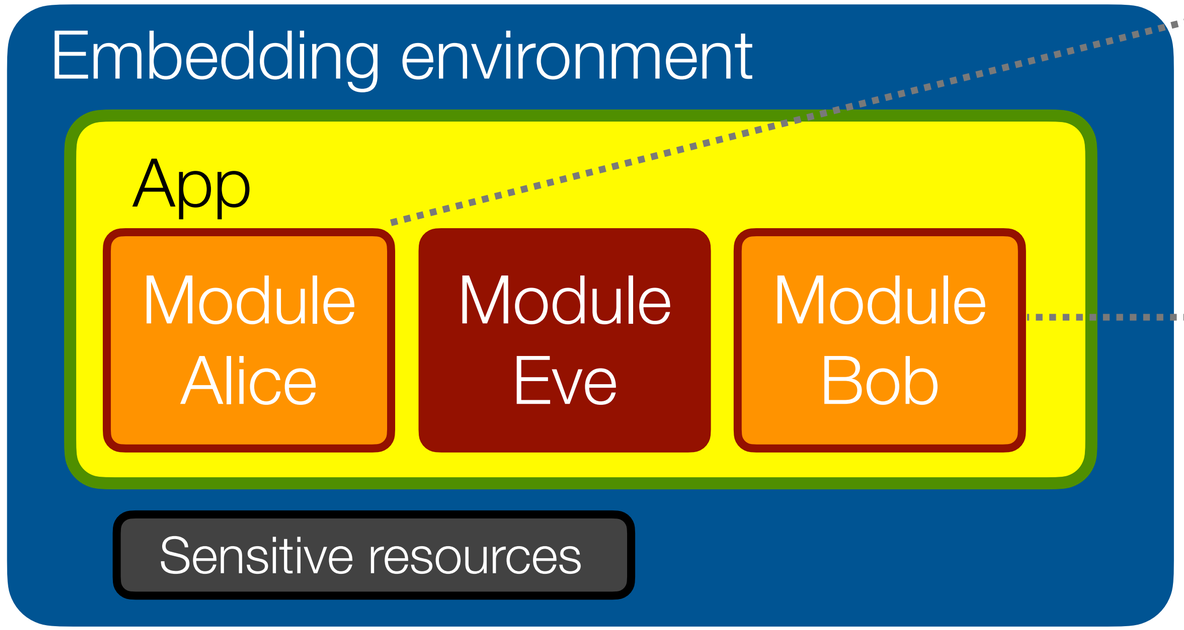
- Consider the following (common) setup:



- How can code inside Alice safely pass objects to Bob *through* Eve while preventing Eve from inspecting or tampering with her objects?
- How can code inside Bob verify that the objects passed to it from Eve originated from Alice?

#4: use sealer/unsealer pairs to “encrypt” objects with no crypto

- Alice creates sealer/unsealer pair and gives unsealer to Bob
- Alice seals her objects using sealer before exposing to Eve
- Bob unseals the objects received from Eve using unsealer




```
// Alice says:
const [seal, unseal] =
    makeSealerUnsealerPair();
bob.setup(unseal);

const box = seal(value);
eve.give(box);
```

```
// Bob says:
function setup(unseal) {
  eve.register((box) => {
    const value = unseal(box);
    // use value from Alice
  })
}
```

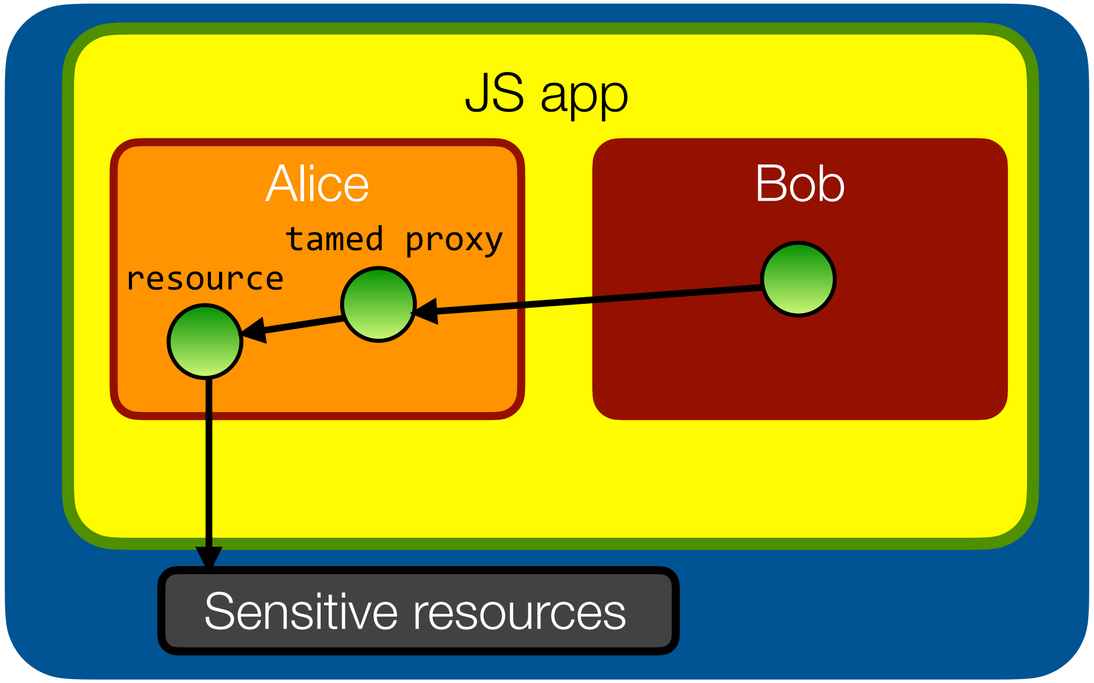
#4: use sealer/unsealer pairs to “encrypt” objects with no crypto

```
function makeSealerUnsealerPair() {
  const boxes = new WeakMap();
  function seal(value) {
    const box = Object.freeze({});
    boxes.set(box, value);
    return box;
  }
  function unseal(box) {
    if (boxes.has(box)) {
      return boxes.get(box);
    } else {
      throw new Error("invalid box");
    }
  }
  return harden([seal, unseal]);
}
```

(code adapted from Google Caja reference implementation. Based on ideas from James Morris, 1973)

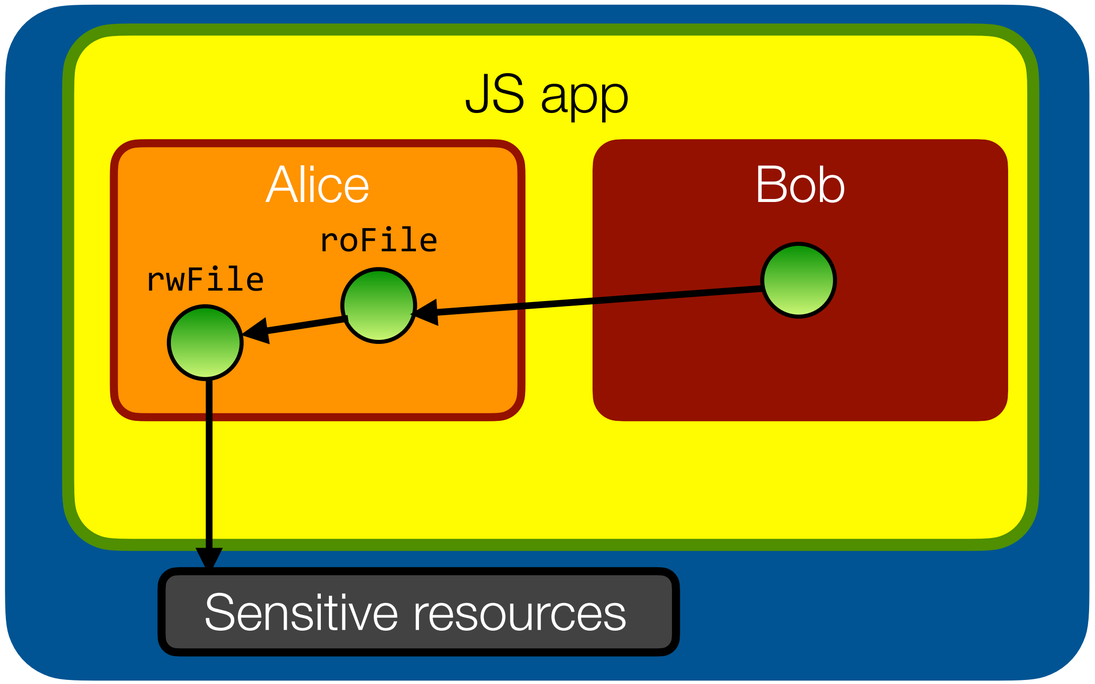
#5: use the Proxy pattern to attenuate APIs (taming)

- Expose powerful objects through restrictive proxies to third-party code
- For example, a proxy object may expose only a subset of the API



#5: use the Proxy pattern to attenuate APIs (taming)

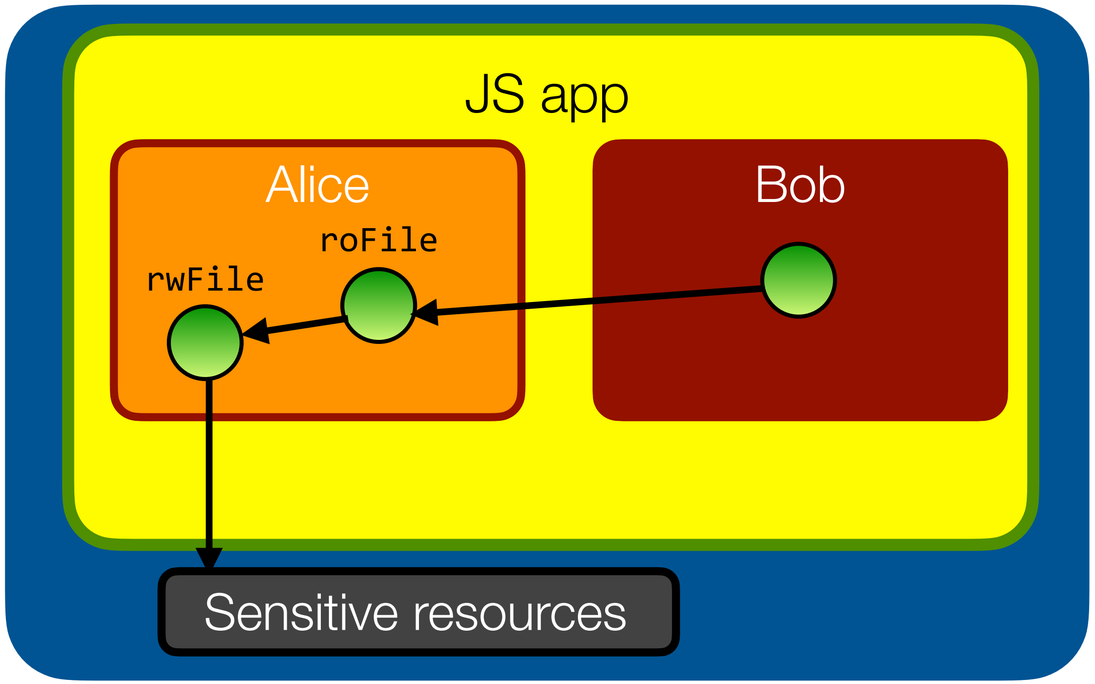
- Implement whatever access control policy is relevant to your app
- Example: attenuating read-write access to read-only access:




```
interface File {
  read(): string[]
  write(string[] s): void
  numLines(): number
}
```

#5: use the Proxy pattern to attenuate APIs (taming)

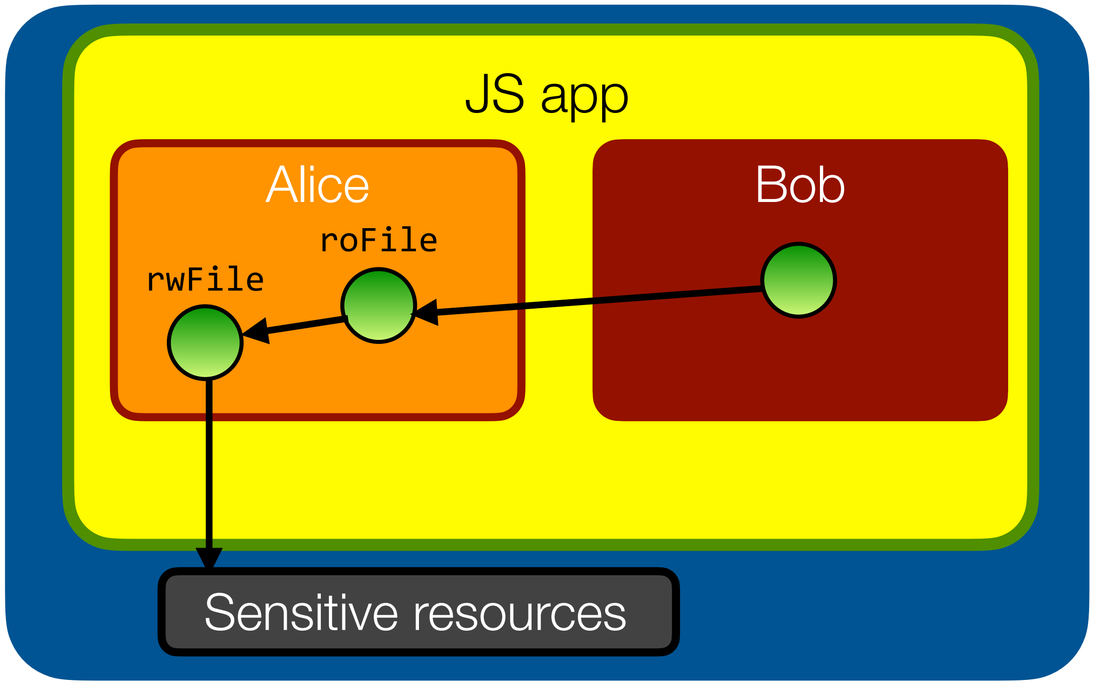
- Implement whatever access control policy is relevant to your app
- Example: attenuating read-write access to read-only access:




```
function makeReadOnly(file) {
  return harden({
    read() { return file.read(); }
    write(s) { throw `readonly`; }
    numLines() { return file.numLines(); }
  });
}

// Alice says:
const roFile = makeReadOnly(rwFile);
bob.give(roFile);
```

#5: use the Proxy pattern to attenuate APIs (taming)

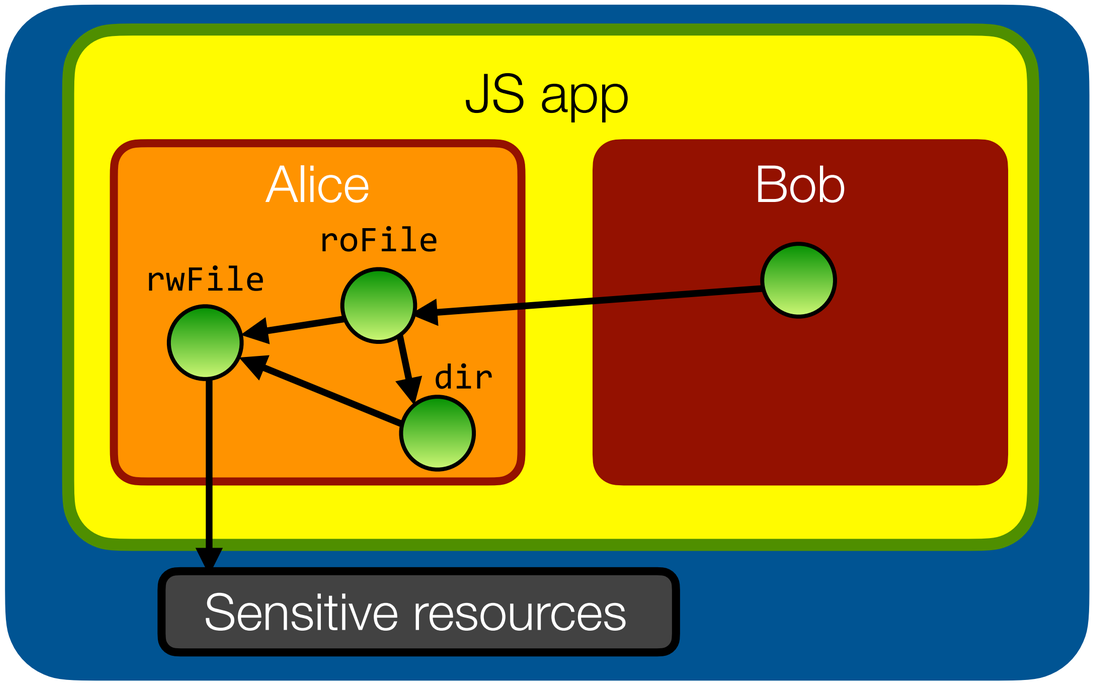
- Implement whatever access control policy is relevant to your app
- Example: attenuating read-write access to read-only access:




```
interface File {
  read(): string[]
  write(string[] s): void
  numLines(): number
  getParent(): Directory
}

interface Directory {
  listFiles(): File[]
}
```

#5: use the Proxy pattern to attenuate APIs (taming)

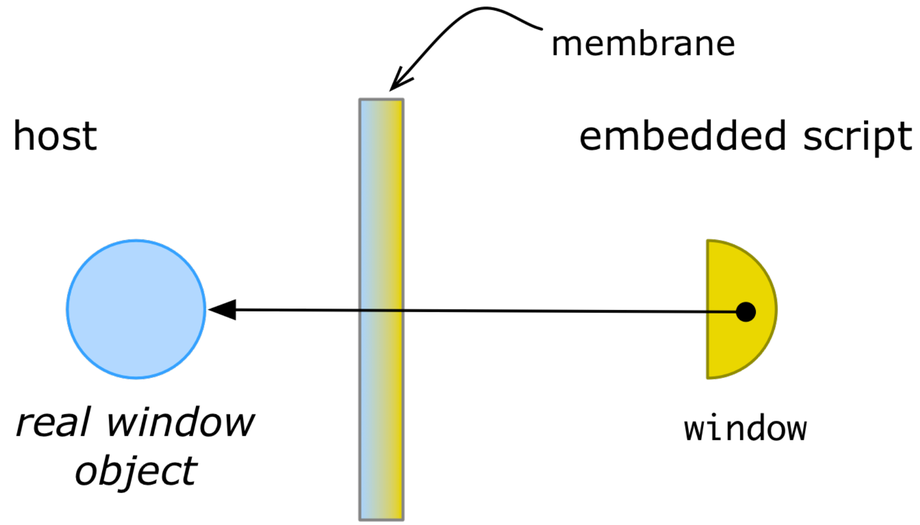
- Pitfall: intercepting transitive access to the underlying resource




```
function makeReadOnly(file) {
  return harden({
    read() { return file.read(); }
    write(s) { throw `readonly`; }
    numLines() { return file.numLines(); }
    getParent() { return file.getParent(); }
  });
}

// Alice says:
const roFile = makeReadOnly(rwFile);
bob.give(roFile);

// Bob says:
const dir = roFile.getParent();
dir.listFiles()[0].write(`gotcha`);
```

#6: use the Membrane pattern to isolate entire groups of objects

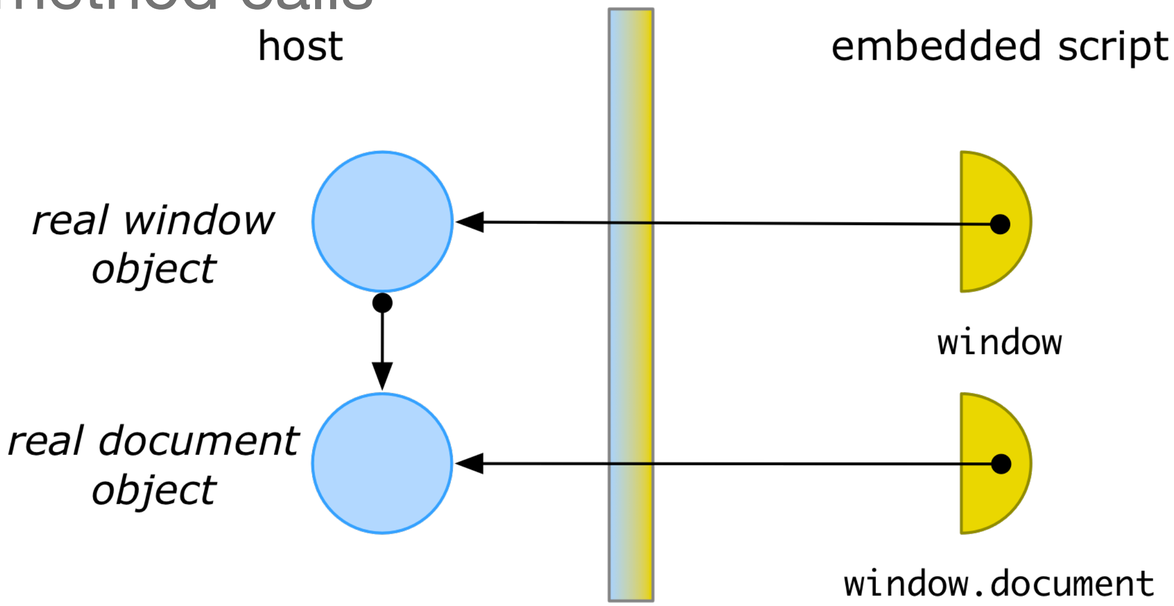
- **Membranes** generalize the Proxy pattern: wrap groups of objects (object graphs) rather than one single object
- The trick is to dynamically inject new proxy objects by intercepting all property access / method calls



Full article at tvcutsem.github.io/membranes

#6: use the Membrane pattern to isolate entire groups of objects

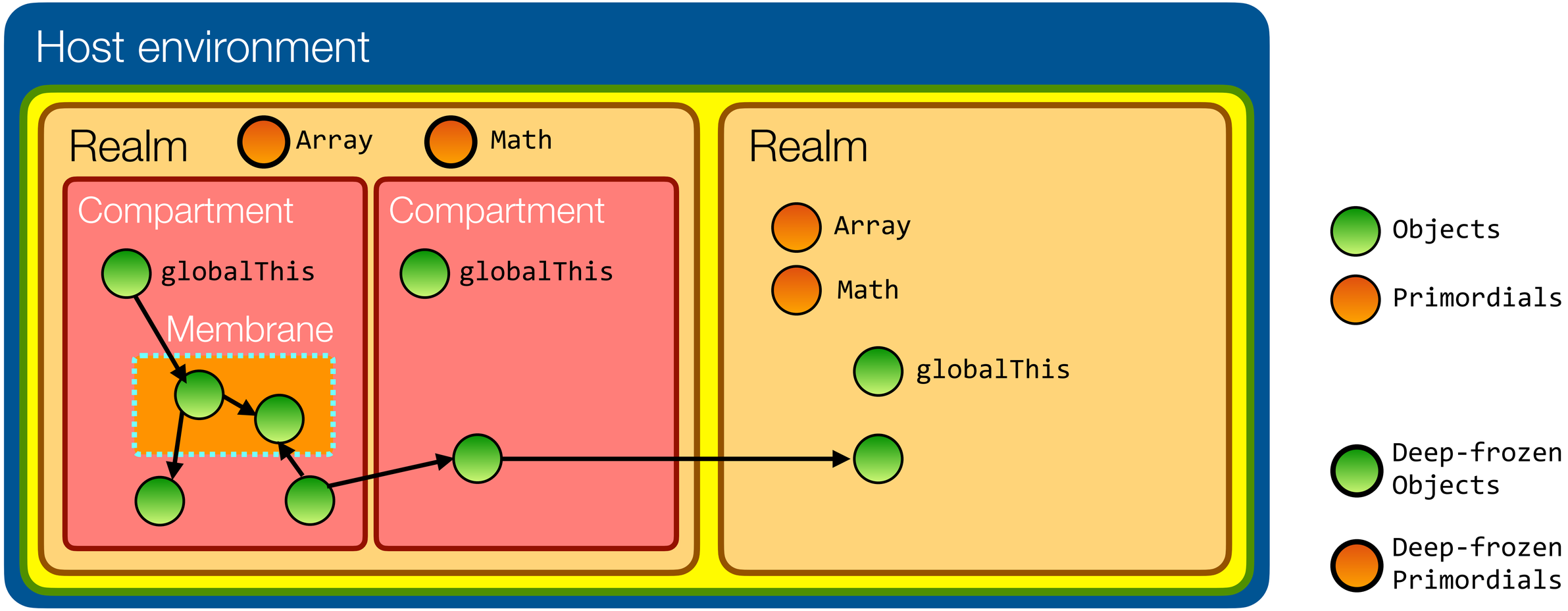
- **Membranes** generalize the Proxy pattern: wrap groups of objects (object graphs) rather than one single object
- The trick is to dynamically inject new proxy objects by intercepting all property access / method calls



Full article at tvcutsem.github.io/membranes

Membranes, Compartments, Realms

- Realms & Compartments manage initial authority. Membranes manage subsequent interactions.



These patterns are used in industry

Google Caja

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

Mozilla Firefox

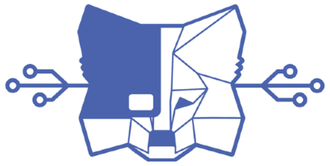
Uses **membranes** to isolate site origins from privileged JS code

Salesforce Lightning

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

Moddable XS

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



MetaMask Snaps

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

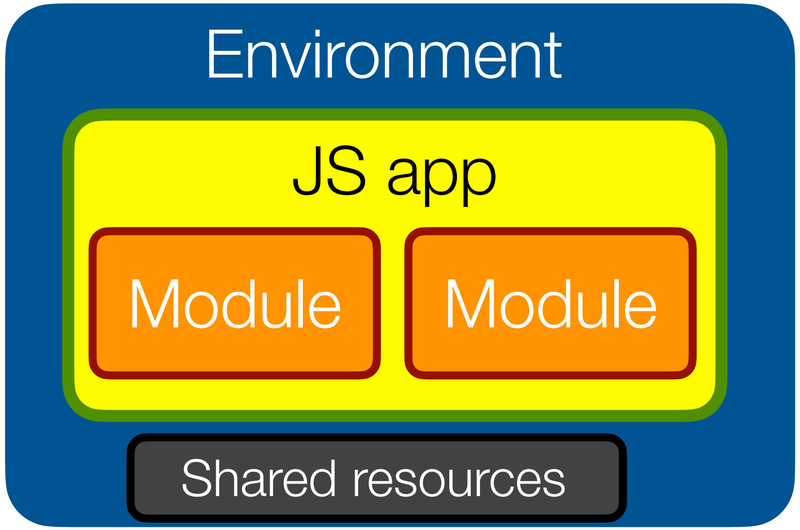
Agoric Zoe

Uses **SES** for writing smart contracts executed on a blockchain

Conclusion

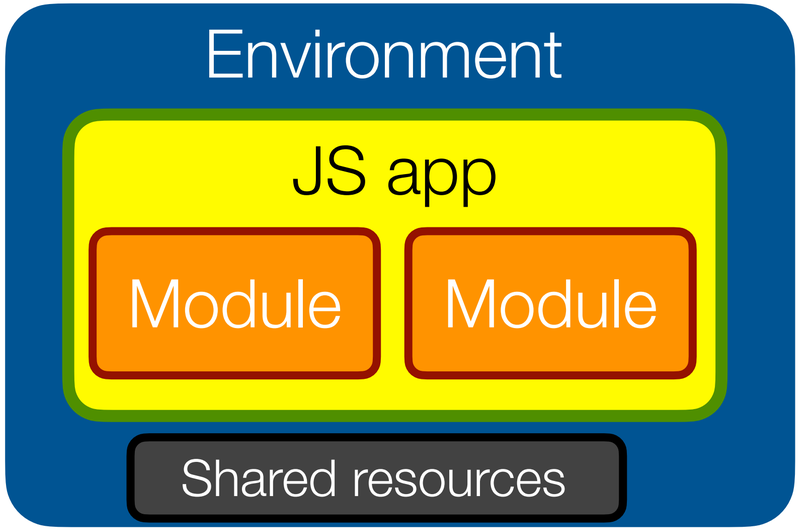
Summary

- Security as the extreme of modularity.
- Modern JS apps are **composed from many modules**. You can't trust them all.
- Traditional **security boundaries don't exist between modules**. SES adds basic isolation.
- Isolated **modules must** still **interact**.
- Design patterns exist to **compose modules** in ways that minimize unwanted interactions.
- Understanding these patterns is **important in a world of > 1,000,000 NPM modules**



Summary

- Security as the extreme of modularity.
- Modern JS apps are **composed from many modules**. You can't trust them all.
- Traditional **security boundaries don't exist between modules**. SES adds basic isolation.
- Isolated **modules must** still **interact**.
- Design patterns exist to **compose modules** in ways that minimize unwanted interactions.
- Understanding these patterns is **important in a world of > 1,000,000 NPM modules**



Thank You!

@tvcutsem

Acknowledgements

- Mark S. Miller (for the inspiring work on Object-capabilities, Robust Composition, E, Caja, JavaScript and Secure ECMAScript)
- Marc Stiegler's "PictureBook of secure cooperation" (2004) was a great source of inspiration for this talk
- Doug Crockford's Good Parts and How JS Works books were an eye-opener and provide a highly opinionated take on how to write clean, good, robust JavaScript code
- 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

References

- Compartments: https://github.com/tc39/proposal-compartments
- Realms: https://github.com/tc39/proposal-realms
- SES: https://github.com/tc39/proposal-ses and https://github.com/Agoric/SES (ancestral version at https://github.com/google/caja/wiki/SES)
- Subsetting ECMAScript: https://github.com/Agoric/Jessie
- Caja: https://developers.google.com/caja
- Sealer/Unsealer pairs: <http://erights.org/elib/capability/ode/ode-capabilities.html> and <http://www.erights.org/history/morris73.pdf>
- 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