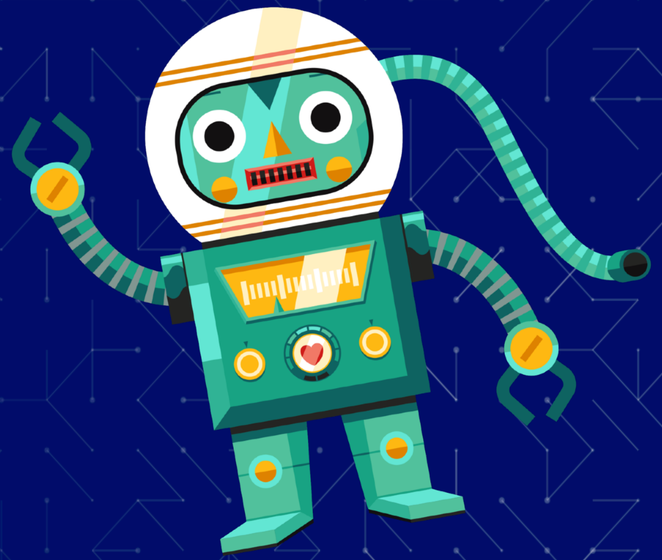


TECHORAMA

TECHORAMA

Architecting Robust JavaScript Applications

A practitioner's guide to Secure ECMAScript

Tom Van Cutsem

@tvcutsem

About me

- Computer scientist with one foot in academia and one foot in industry
- Past TC39 member and active contributor to ECMAScript standards
- Author of Proxy and Reflect APIs
- Author of Traits.js
- Passionate about programming languages and JavaScript in particular

 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

functions

information hiding

dependencies

immutability

dataflow

encapsulation

A software architecture view of security

“Security is just the extreme of Modularity”

- Mark S. Miller
(Chief Scientist, Agoric)

Modularity: avoid needless dependencies (to prevent bugs)

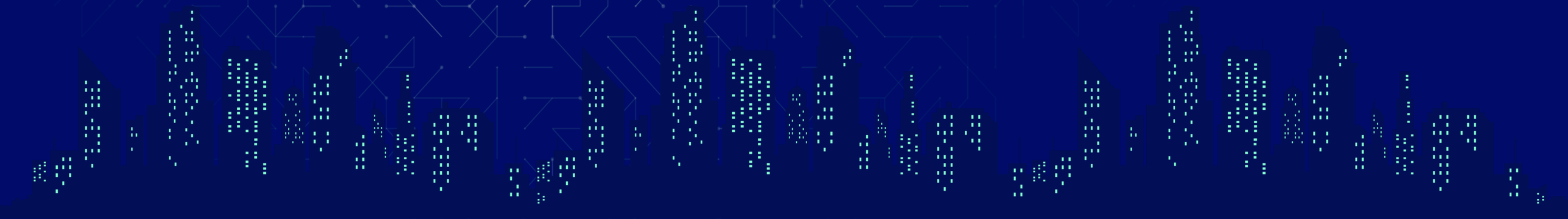
Security: avoid needless vulnerabilities (to prevent exploits)

This Talk

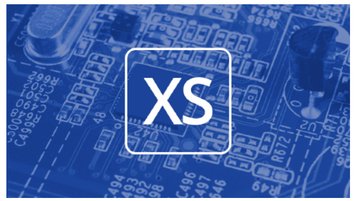
- Part I: why it’s becoming important to write more robust applications
- Part II: patterns that let you write more robust applications

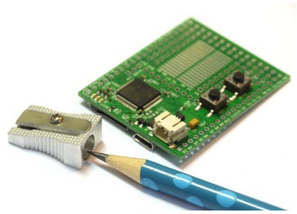
Part I

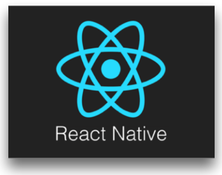
The need for more robust JavaScript apps



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

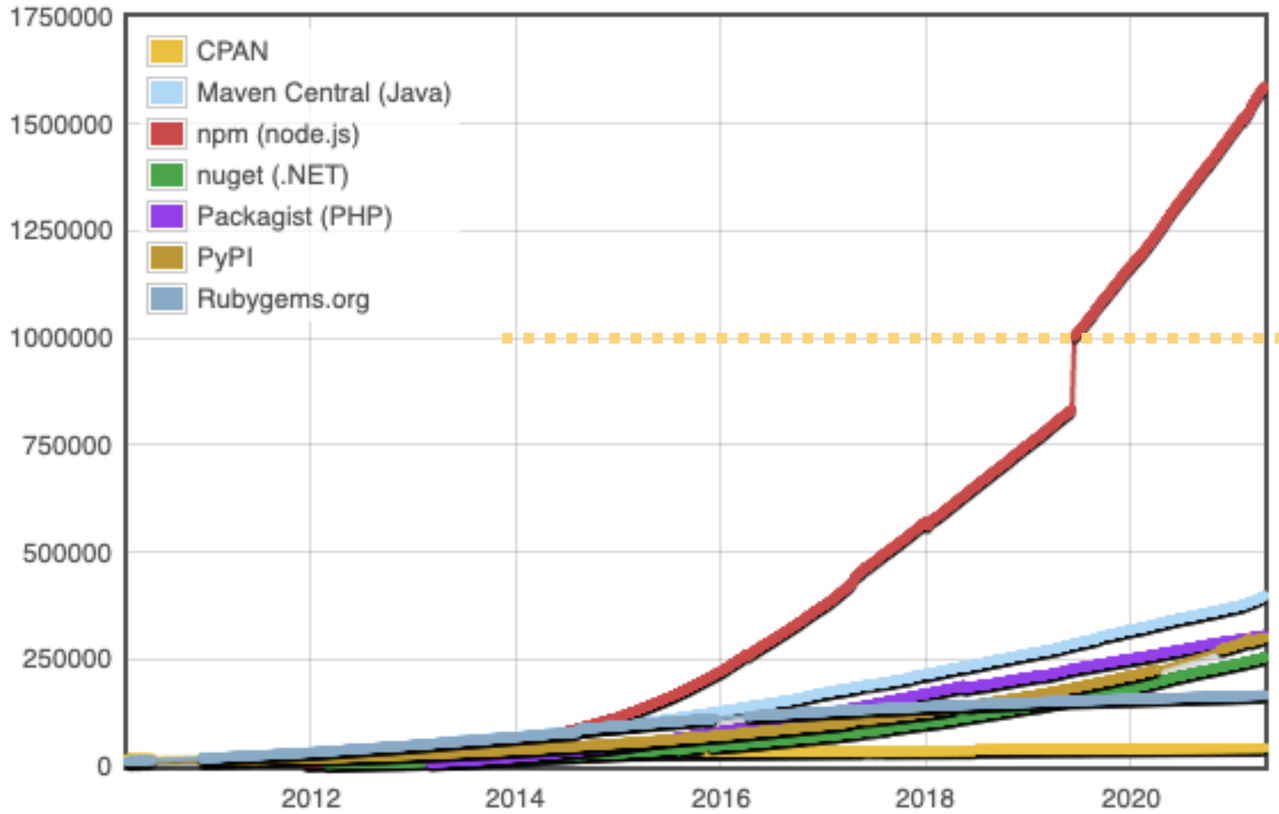






Embedded Mobile Desktop/Native Server Database

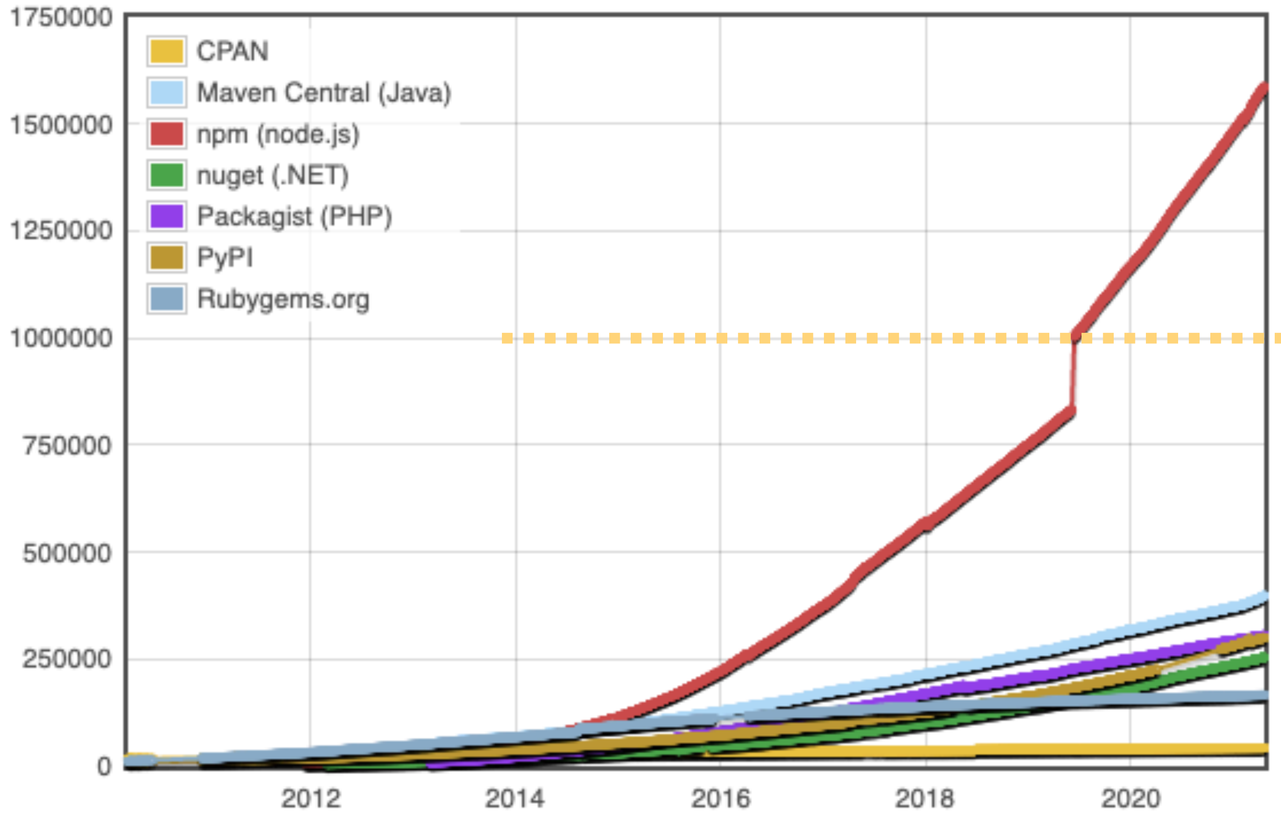
JavaScript applications are now built from thousands of modules



1,000,000 modules on NPM

(source: modulecounts.com, April 2021)

JavaScript applications are now built from thousands of modules



(source: modulecounts.com, April 2021)

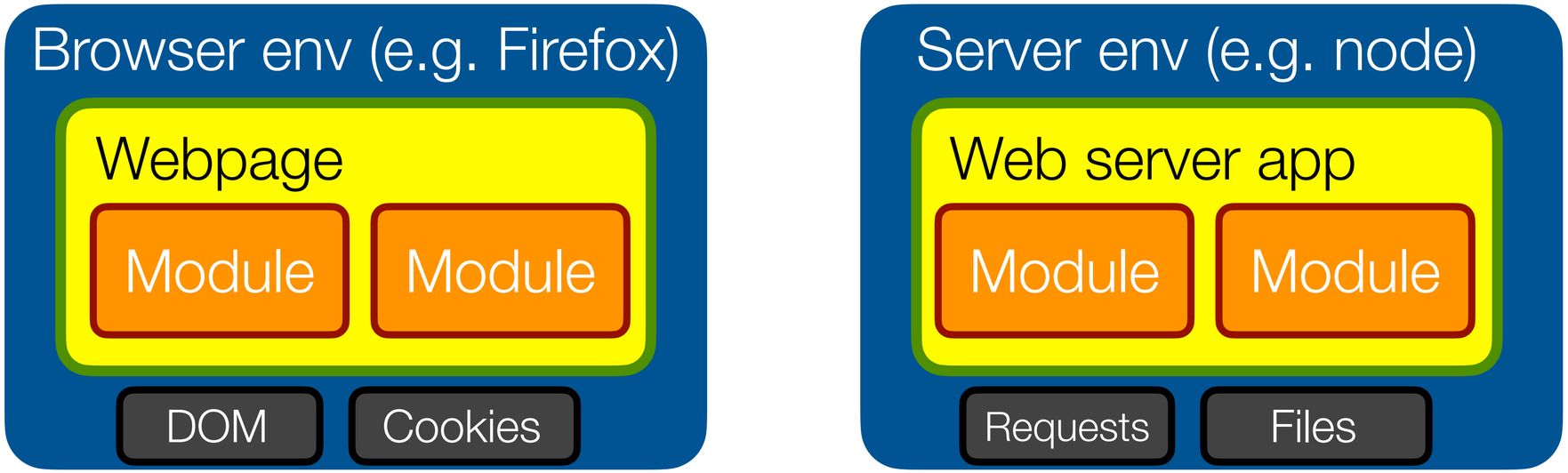
1,000,000 modules on NPM

"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)

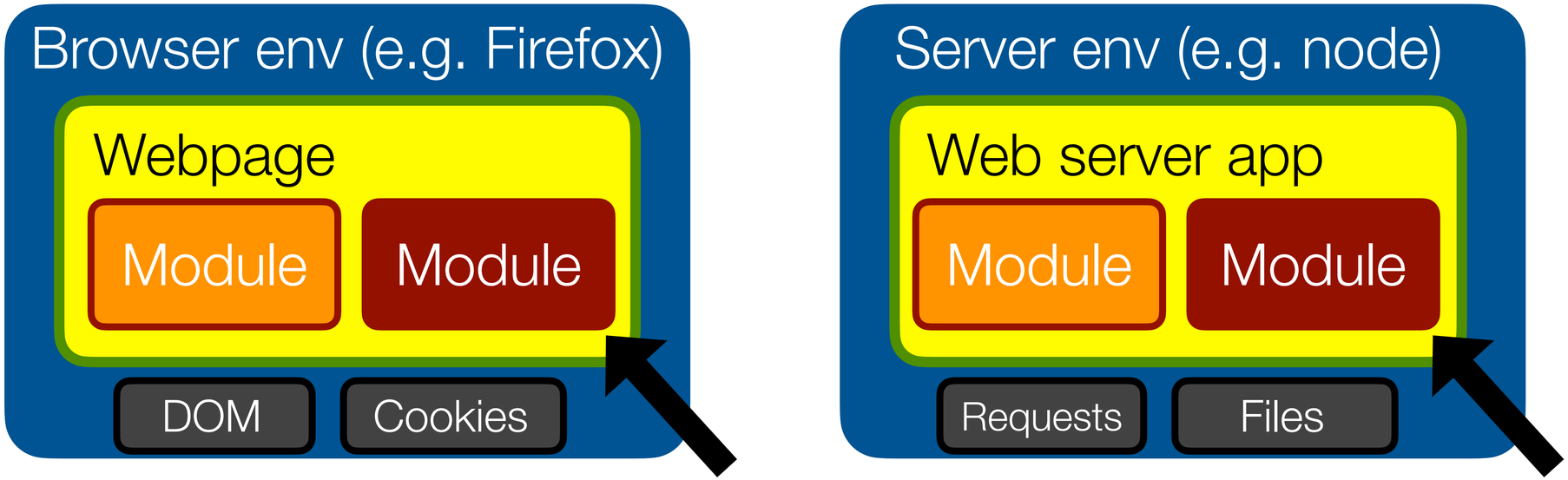
It’s all about **trust**

- 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

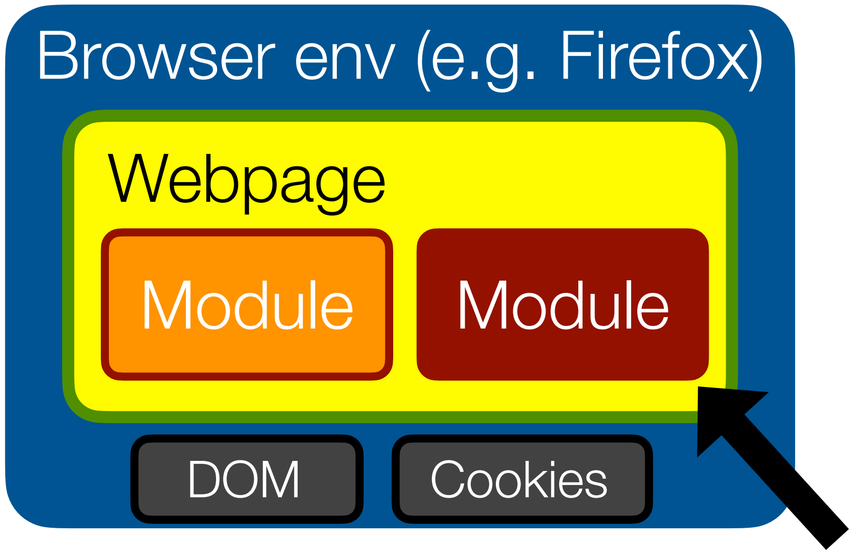



```
<script src=“http://evil.com/ad.js”>
```

```
npm install evil-logger
```

It’s all about **trust**

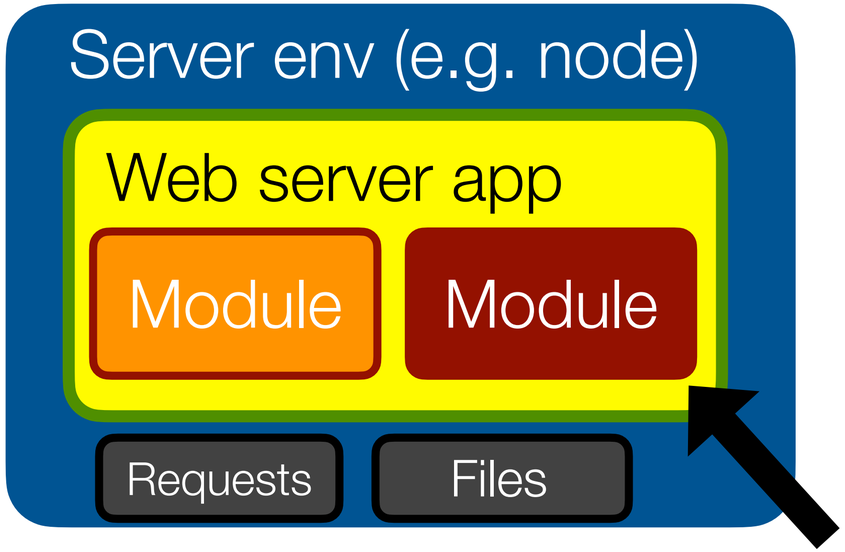
- It is exceedingly common to run code you don’t know/trust in a common environment




```
<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

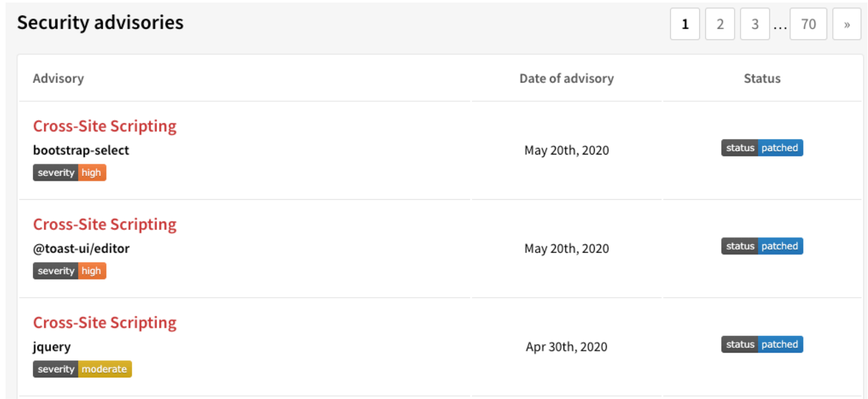
By Thomas Claburn in San Francisco 26 Nov 2018 at 20:58 49 SHARE ▼

(source: theregister.co.uk)

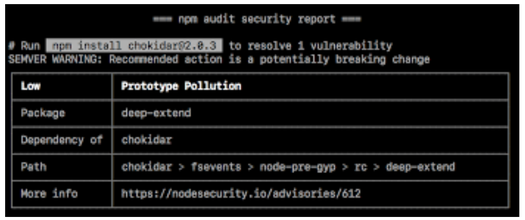
Vulnerabilities in dependencies: increasing awareness

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

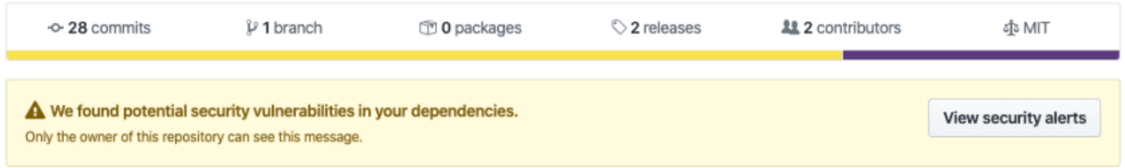
npm security advisories



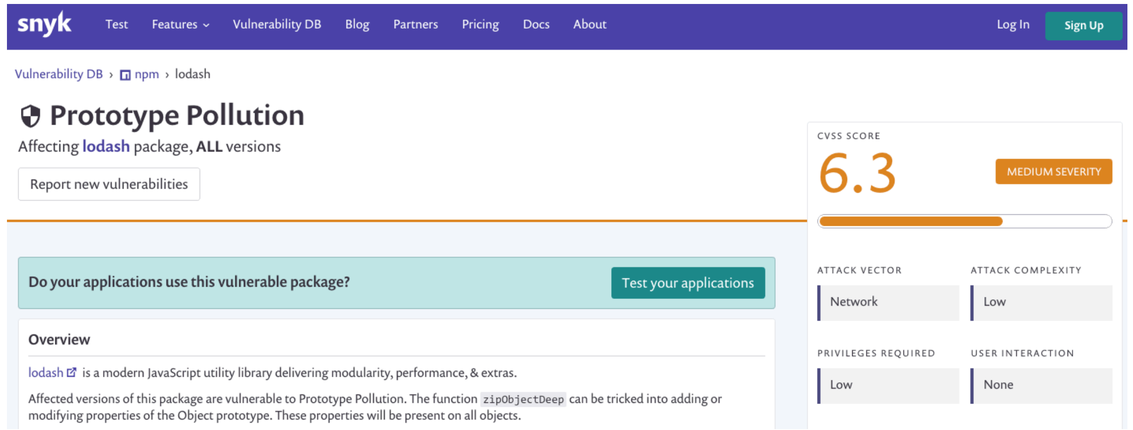
npm audit



GitHub security alerts

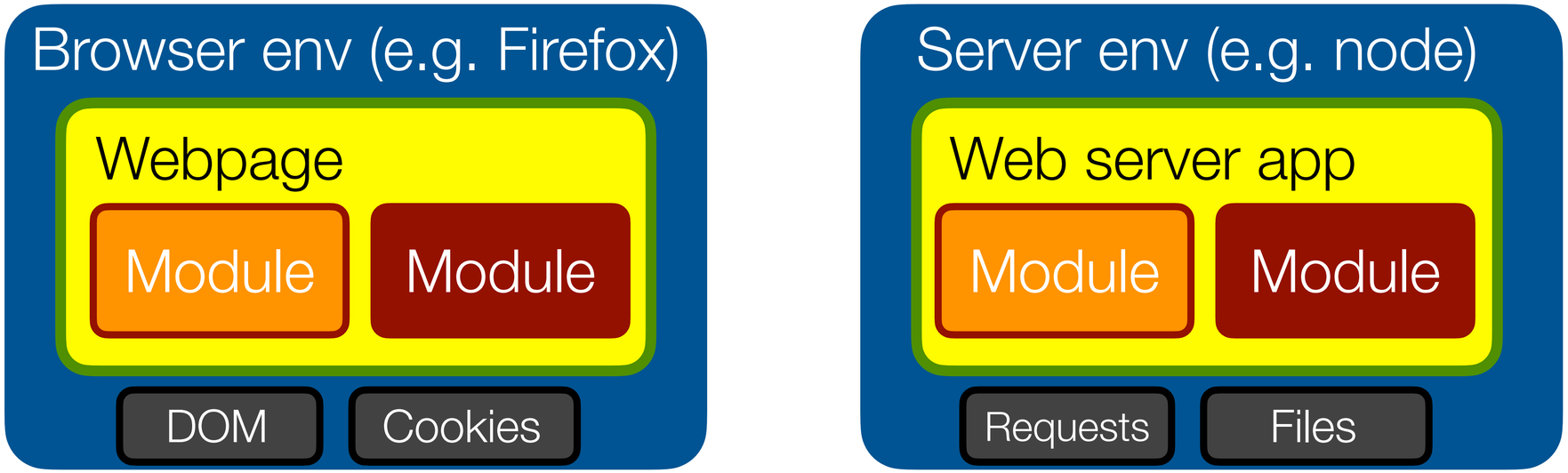


Snyk vulnerability DB



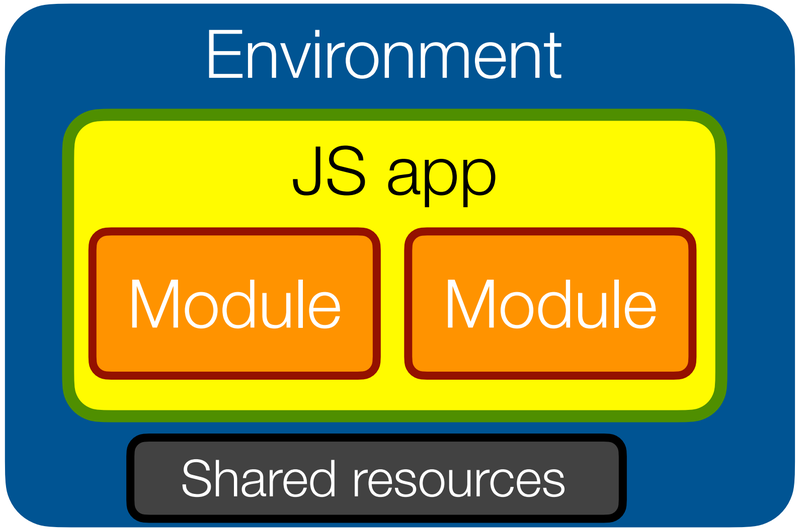
Avoiding interference is the name of the game

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



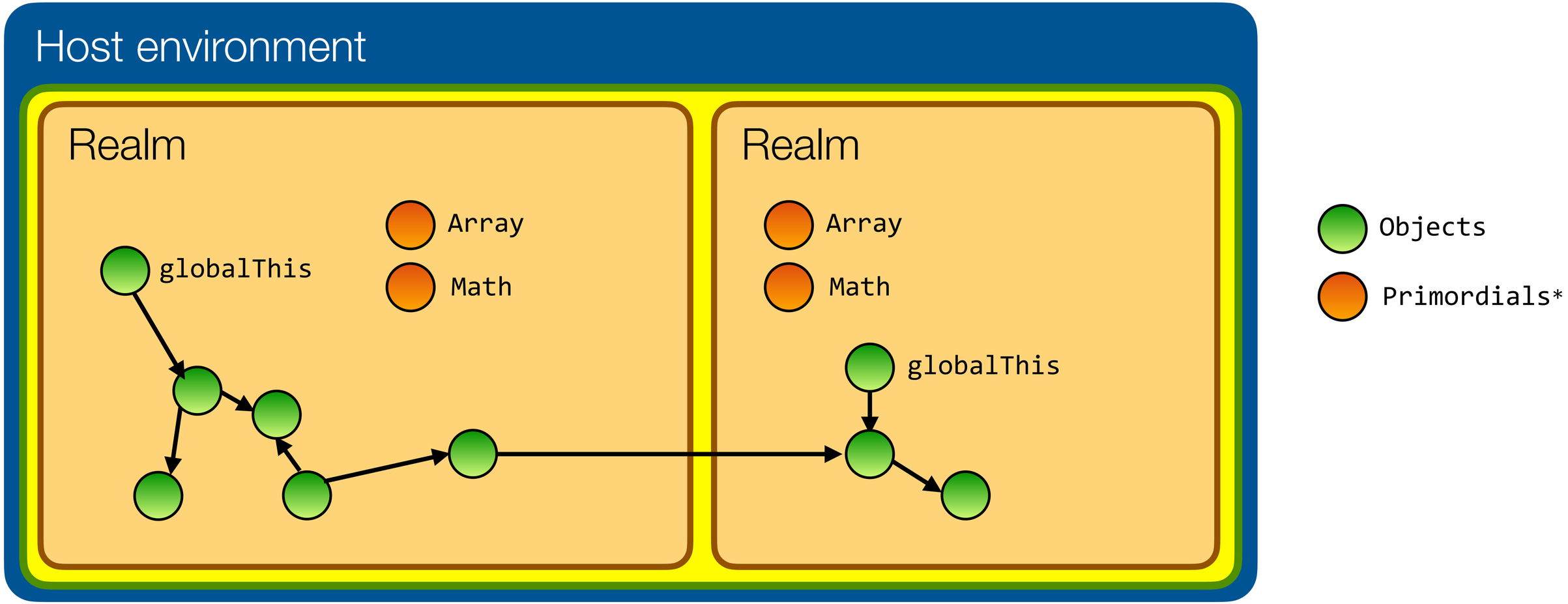
Prerequisite: isolating JavaScript modules

- Today: JavaScript offers no 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”
  - **node** `vm` **module**: same issues



Realms (TC39 Stage 3 proposal)

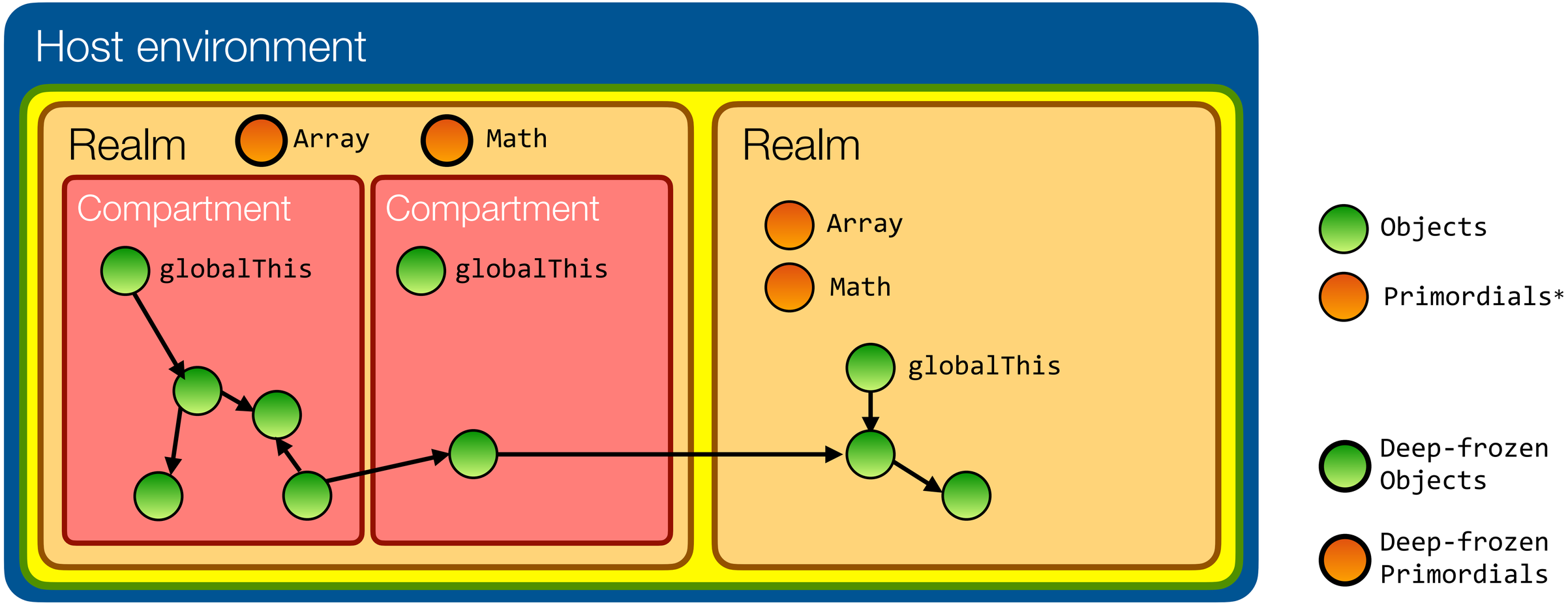
- Intuitions: “iframe without DOM”, “principled version of node’s `vm` module”



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

Compartments (TC39 Stage 1 proposal)

- Compartments: separate globals but shared (immutable) primordials.



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

Realms and Compartments: draft API

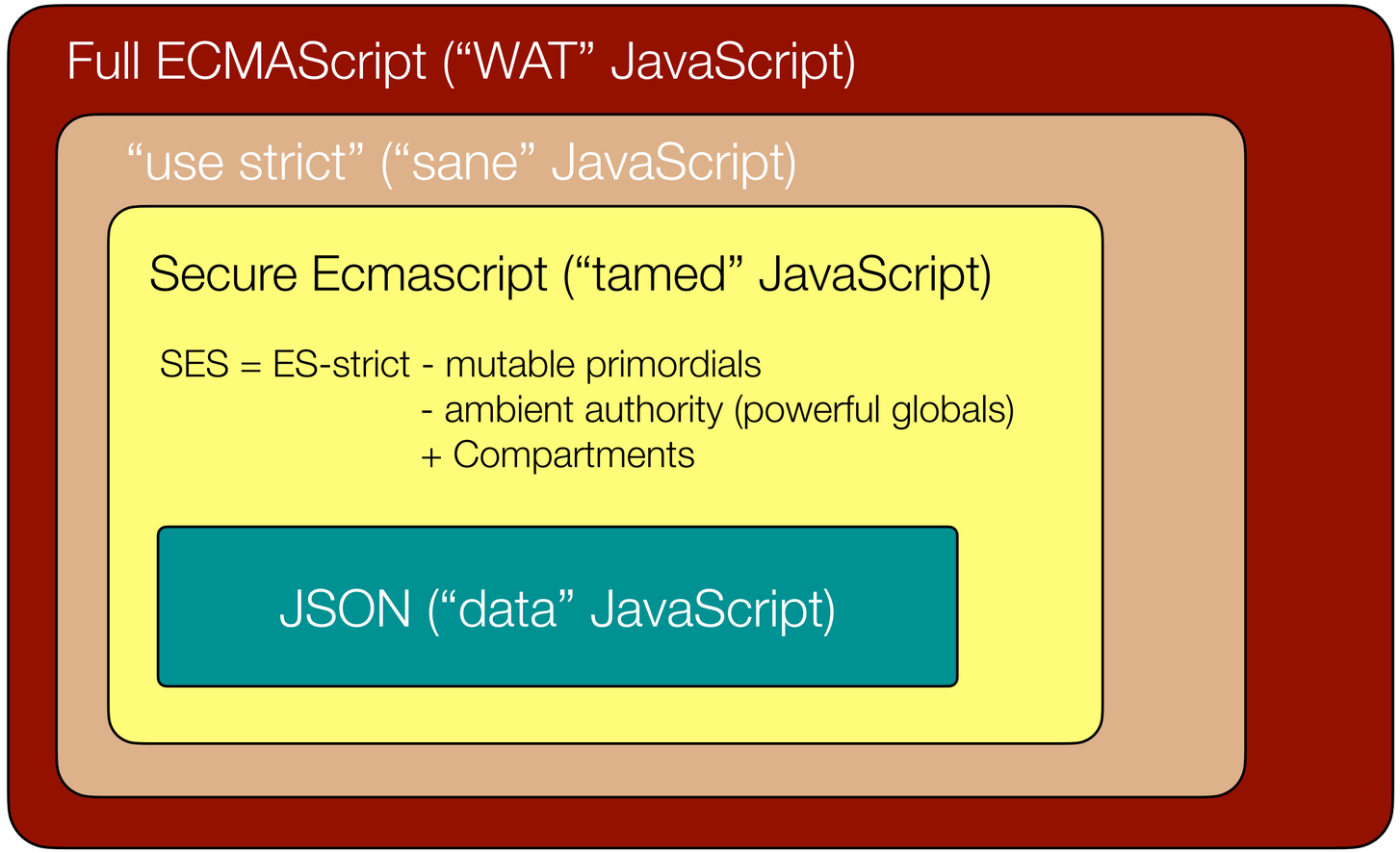
Realms

```
let r = new Realm();
let res = r.evaluate(`x = 1; x + 1`);
// here, x is still undefined


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


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


let arr = r.evaluate(`[]`);
// TypeError: no x-realm object access
allowed, only callable objects
```

Compartments

```
let c = new Compartment({x: 1})
let res = c.evaluate(`x + 1`) // => 2


// 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
```

Secure ECMAScript (SES)



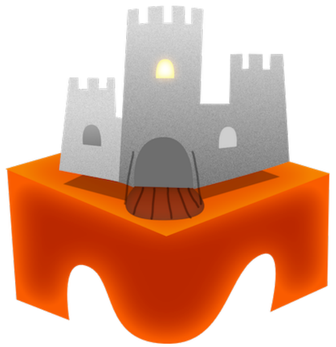
(inspired by the diagram at https://github.com/Agoric/Jessie)

- A subset of JavaScript, building on Compartments (TC39 Stage 1 proposal)
- Key idea: no powerful objects by default. SES code can only affect the outside world only through objects (capabilities) explicitly granted to it (**POLA**)

```
import ‘ses’;
lockdown();
```

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

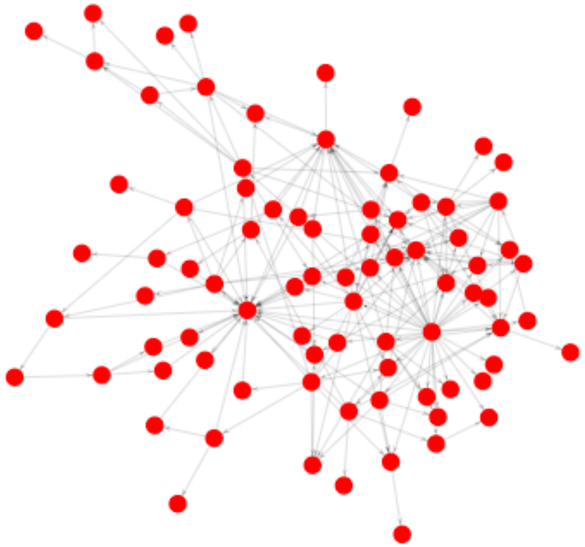


https://github.com/LavaMoat/lavamoat

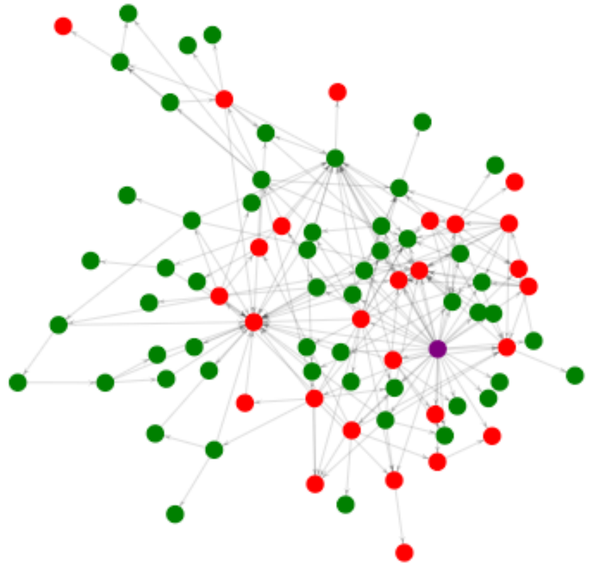

```
"stream-http": {
  "globals": {
    "Blob": true,
    "MSStreamReader": true,
    "ReadableStream": true,
    "VBArray": true,
    "XDomainRequest": true,
    "XMLHttpRequest": true,
    "fetch": true,
    "location.protocol.search": true
  },
  "packages": {
    "buffer": true,
    "builtin-status-codes":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

LavaMoat enables more focused security reviews

Exposure to package dependencies without LavaMoat sandboxing



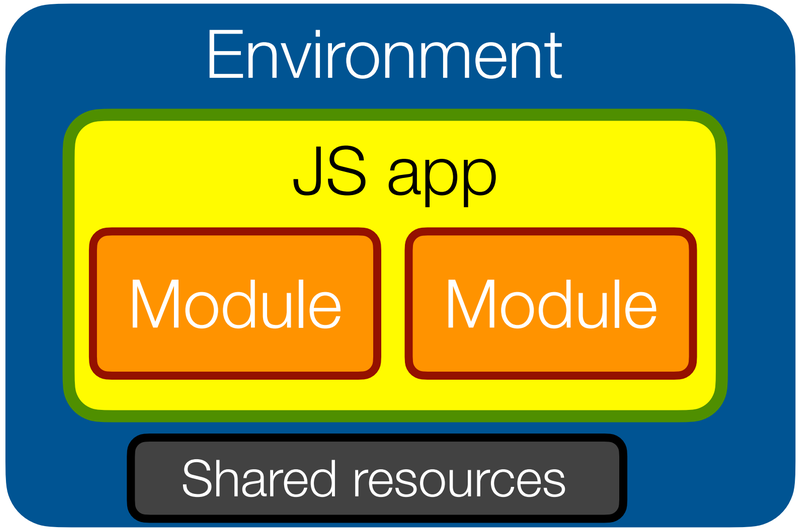
Exposure to package dependencies with LavaMoat sandboxing



https://github.com/LavaMoat/lavamoat

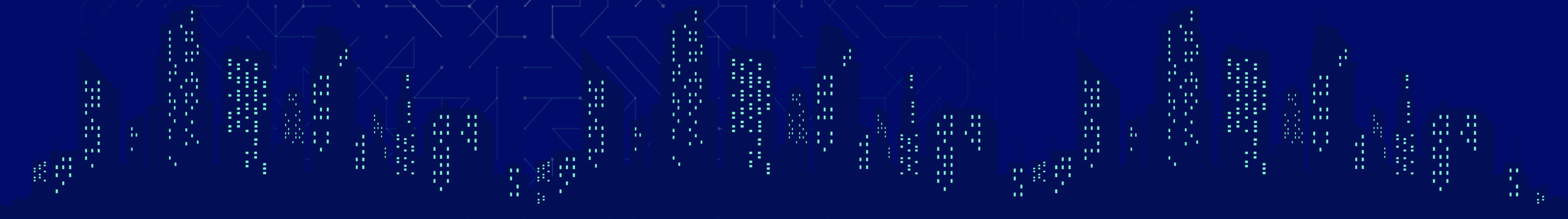
End of Part I: recap

- 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 SES



Part II

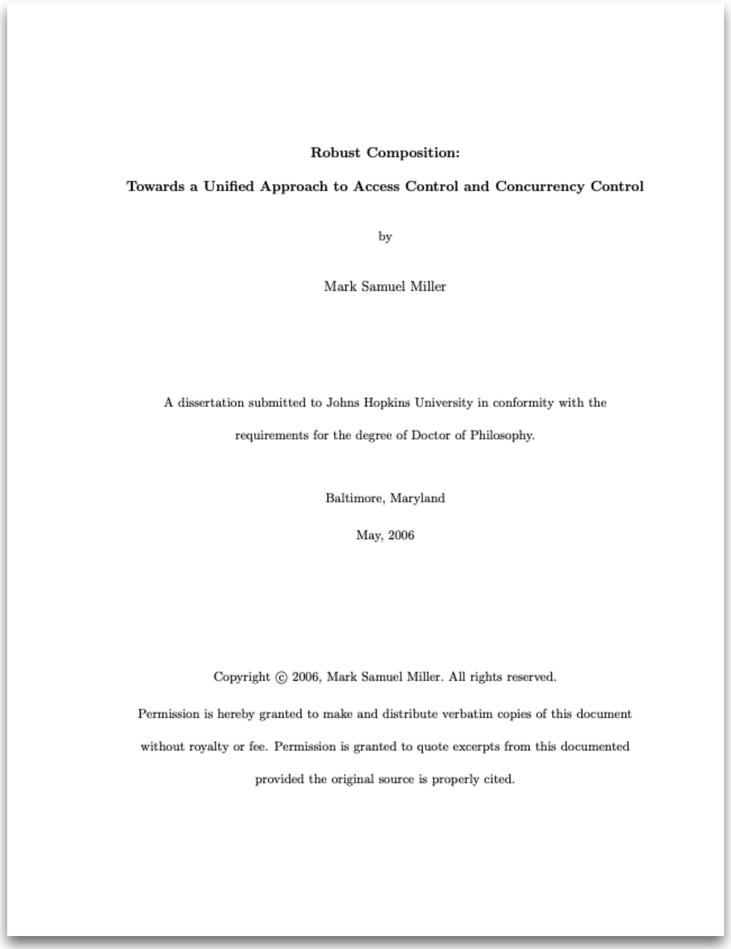
Robust Application Design Patterns



Design Patterns (“Gang of Four”, 1994)

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

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



Robust Composition:

Towards a Unified Approach to Access Control and Concurrency Control

by

Mark Samuel Miller

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

Baltimore, Maryland

May, 2006

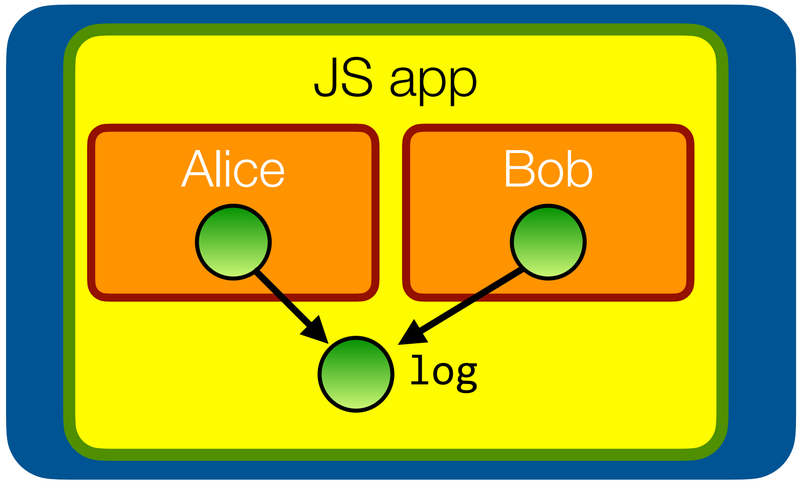
- Taming
- Facet
- Sealer/unsealer pair
- Caretaker
- Membrane
- …

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

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.

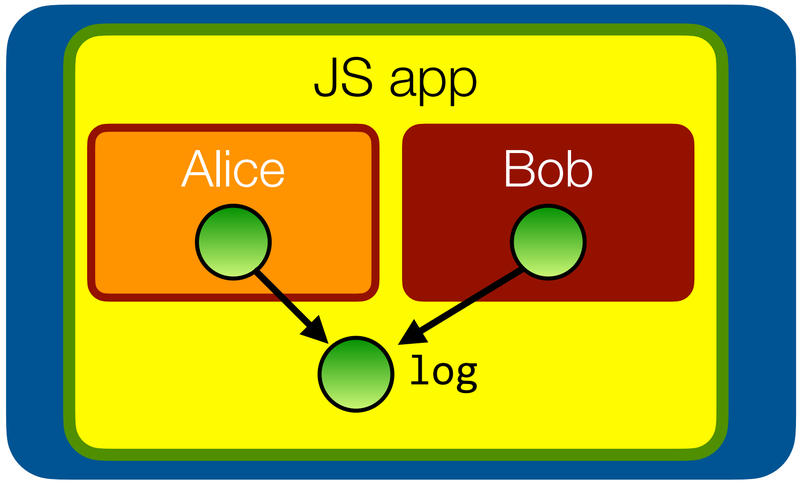
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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



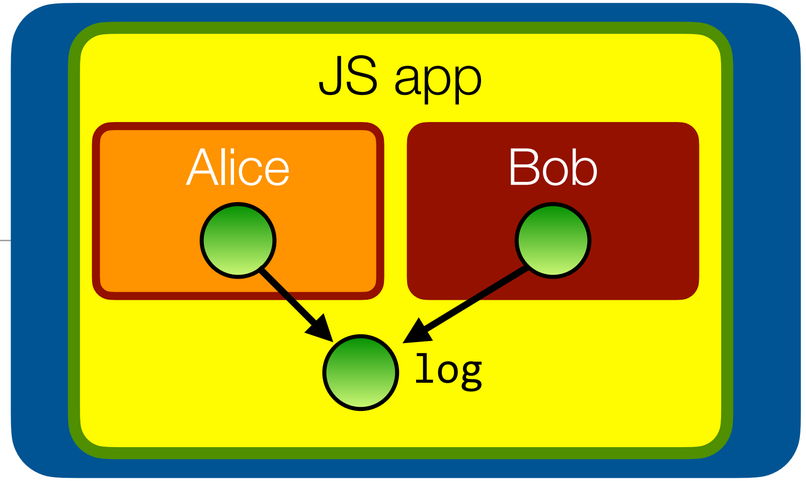
Running example: apply POLA to a basic shared log

- 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!

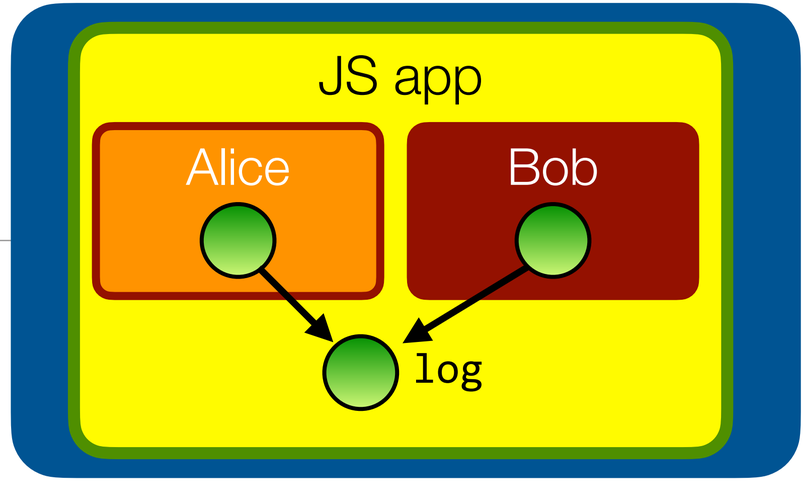



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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”);
}
```

Bob has way too much authority!

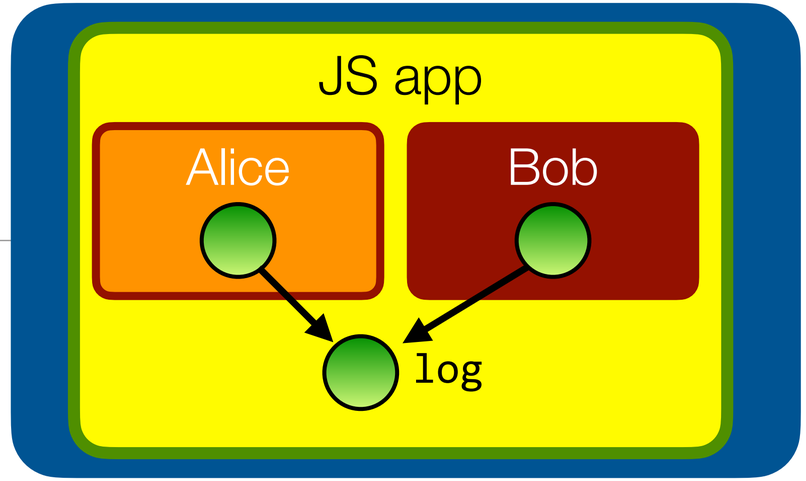
- SES mitigates the last attack. Immutable primordials.




```
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

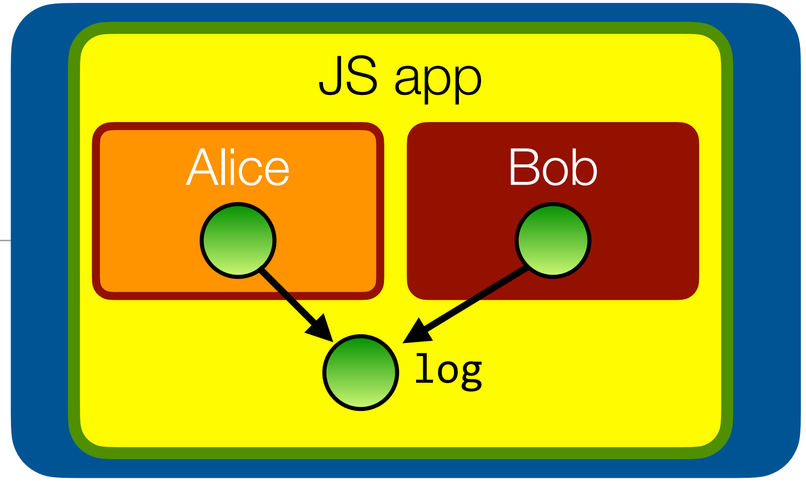



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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 (ES5) makes property *bindings* (not their *values*) immutable

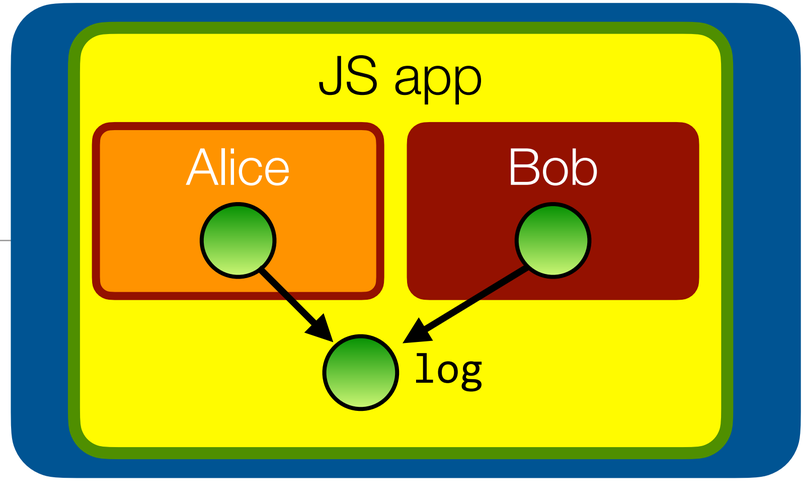



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.

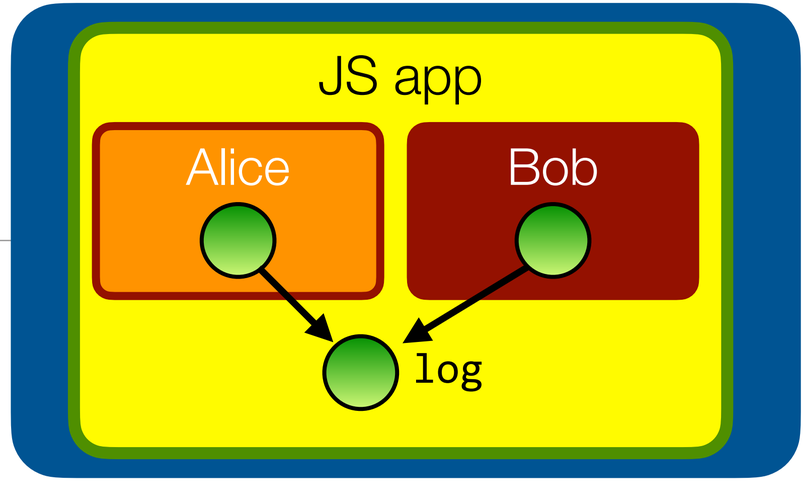



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

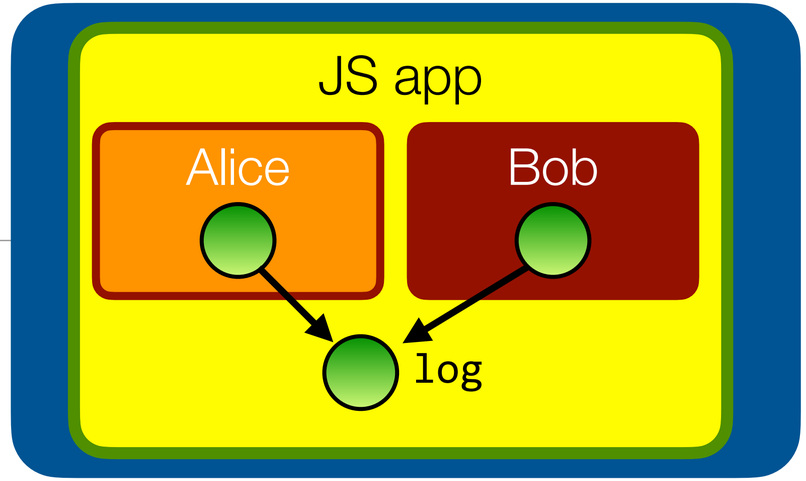
- SES provides a 'harden' function that "deep-freezes" an object




```
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" };
```

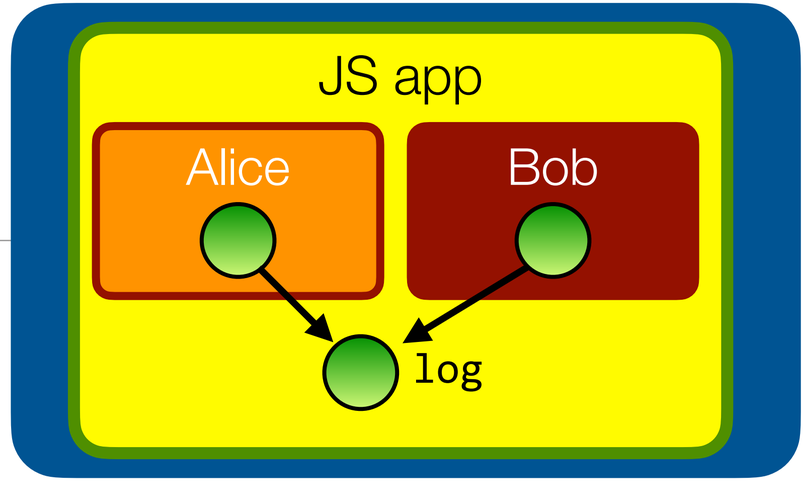
Two down, two to go




```
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” };
```

Don’t share access to mutable internals



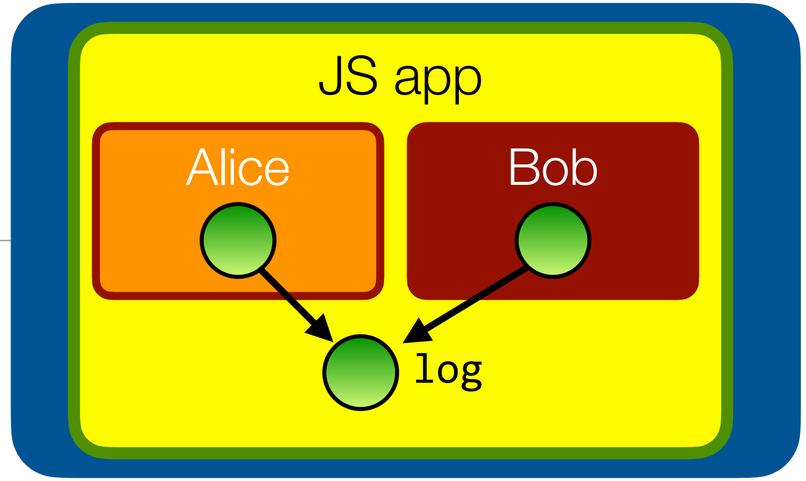
- Modify read() to return a copy of the mutable state.
- Even better would be to use a more efficient copy-on-write or “persistent” data structure (see immutable.js)

```
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

- Bob receives too much authority. How to limit?

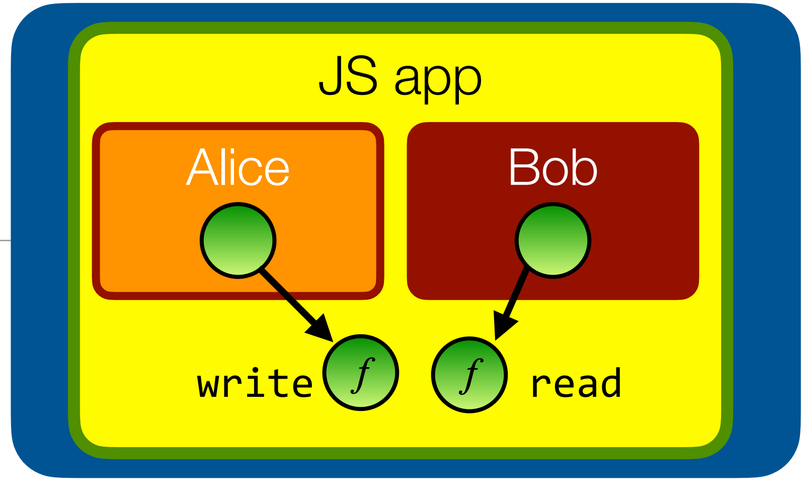



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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?

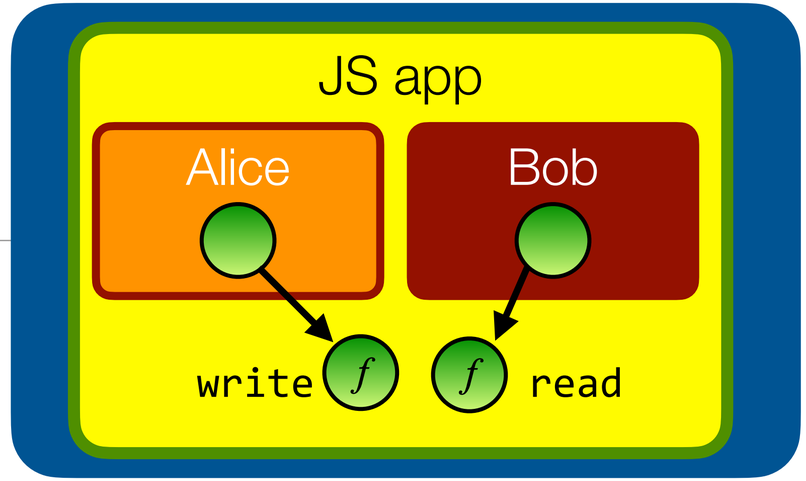



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.

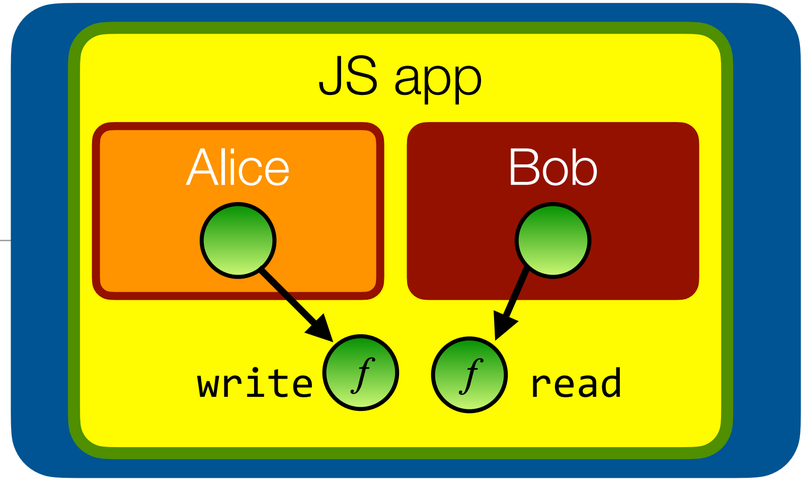
- To avoid, only ever pass bound functions




```
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.

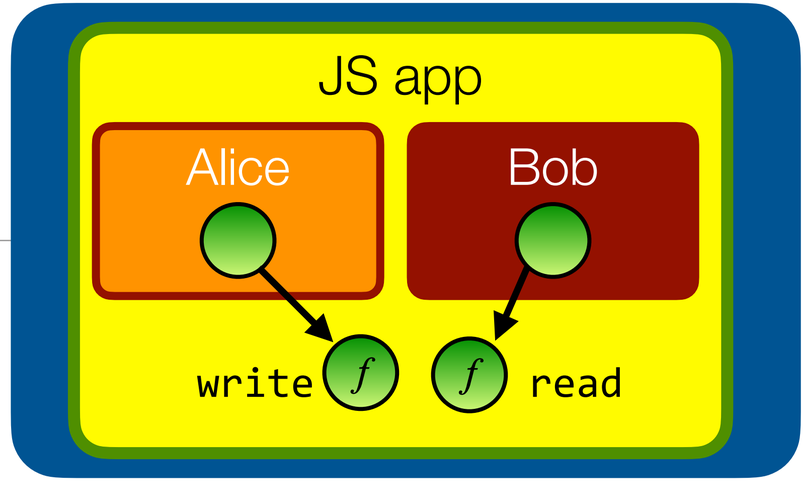



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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” };
```

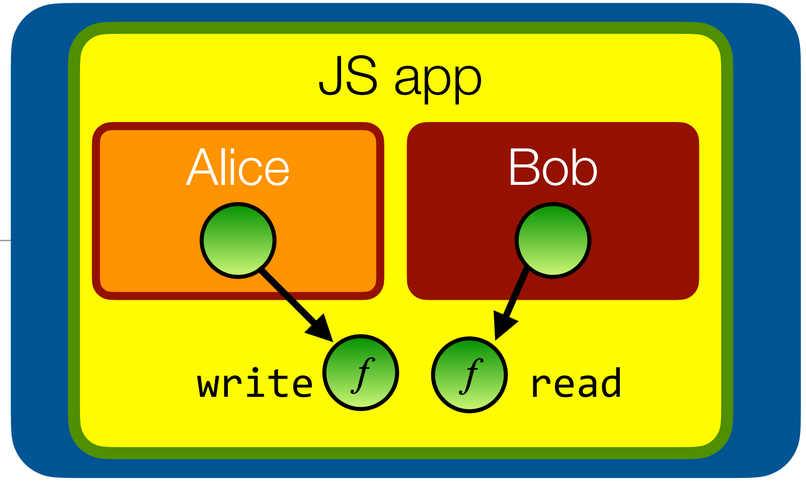
Is there a better way to write this code?

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



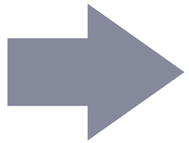

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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 in lieu of using classes or prototypes

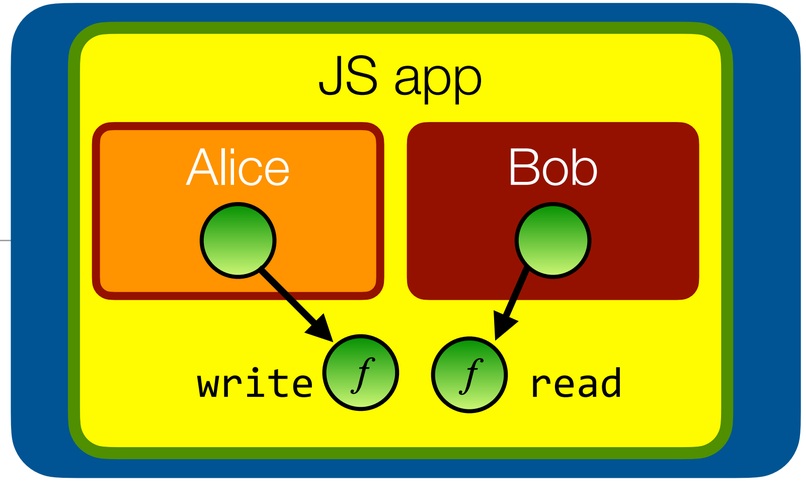
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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)

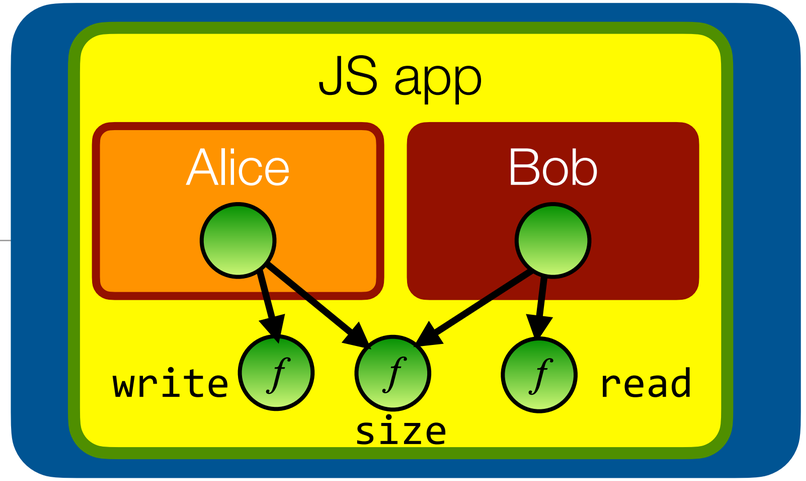
Use the Function as Object pattern



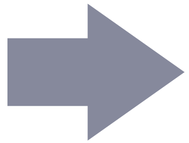

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.



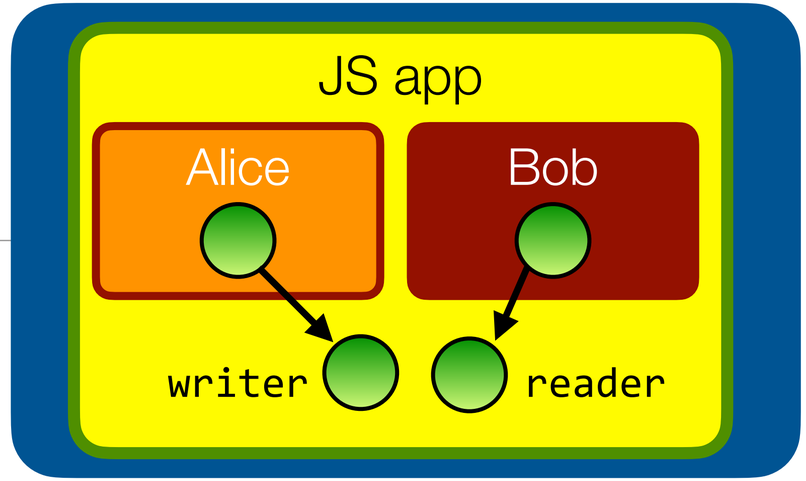

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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);
```



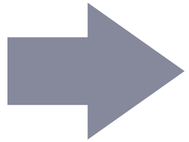
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

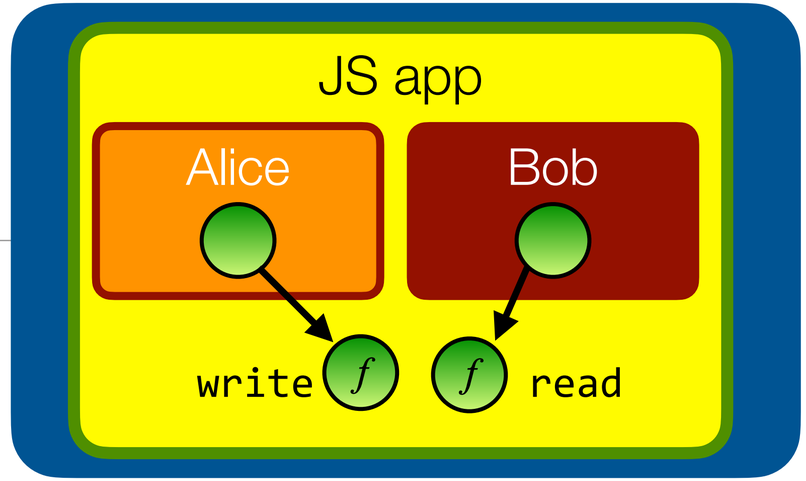



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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);
```

Further limiting Bob's authority

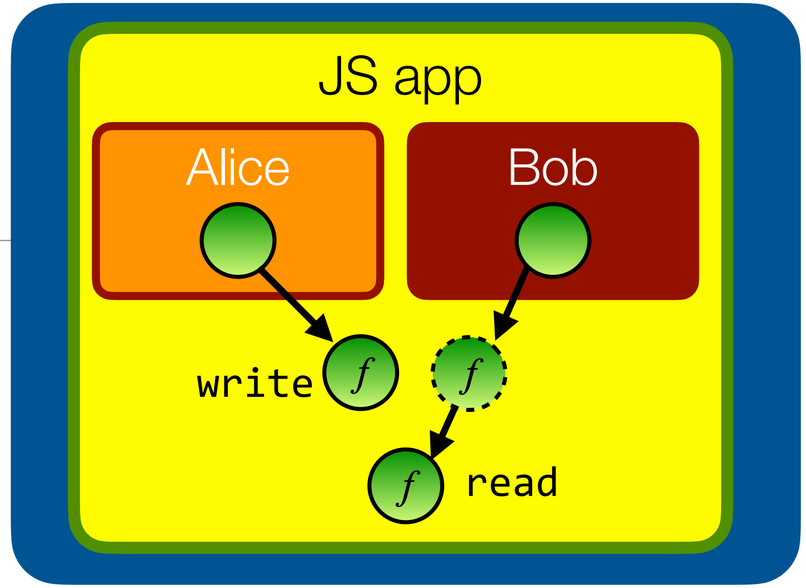


- We would like to give Bob only temporary read access to the log.

```
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

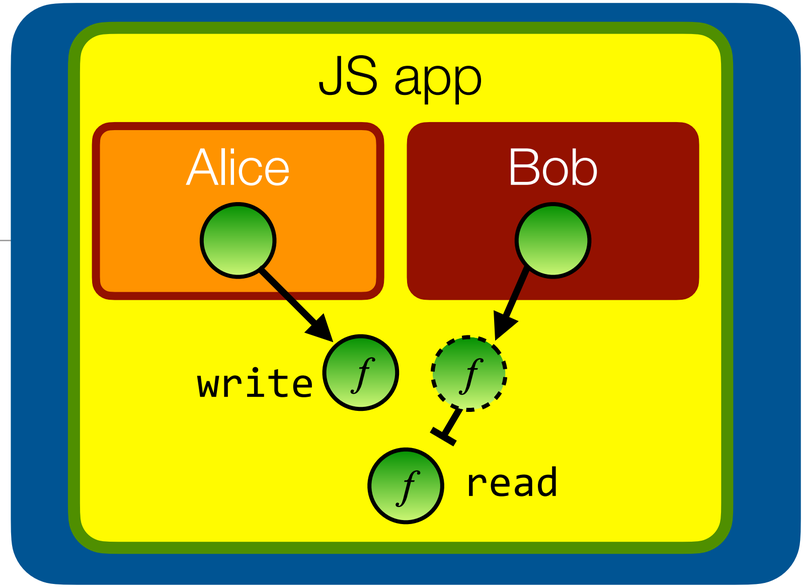
- We would like to give Bob only temporary read access to the log.




```
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);
```

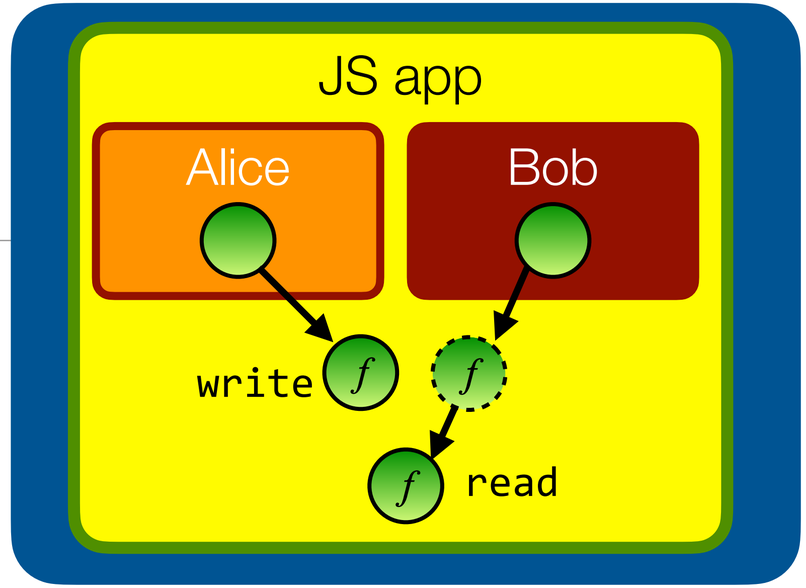
Use **caretaker** to insert access control logic

- 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();
```

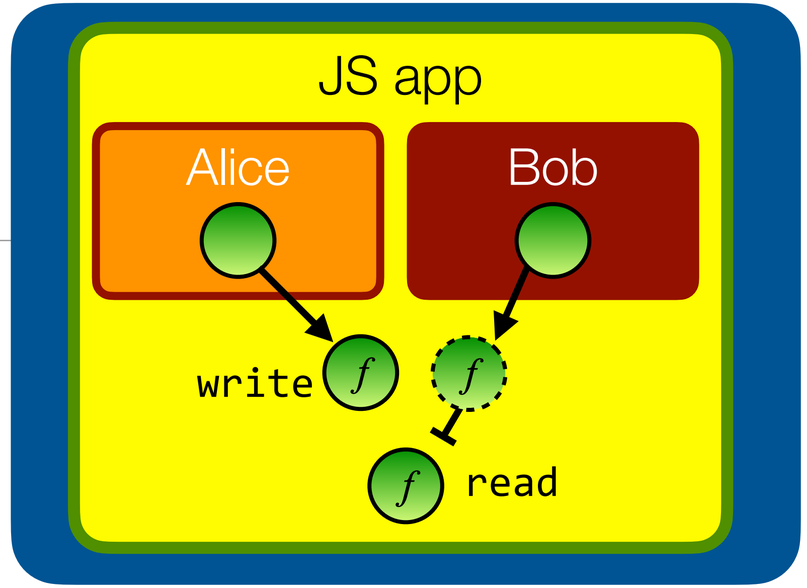
A caretaker is just a proxy object




```
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);
```

```
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



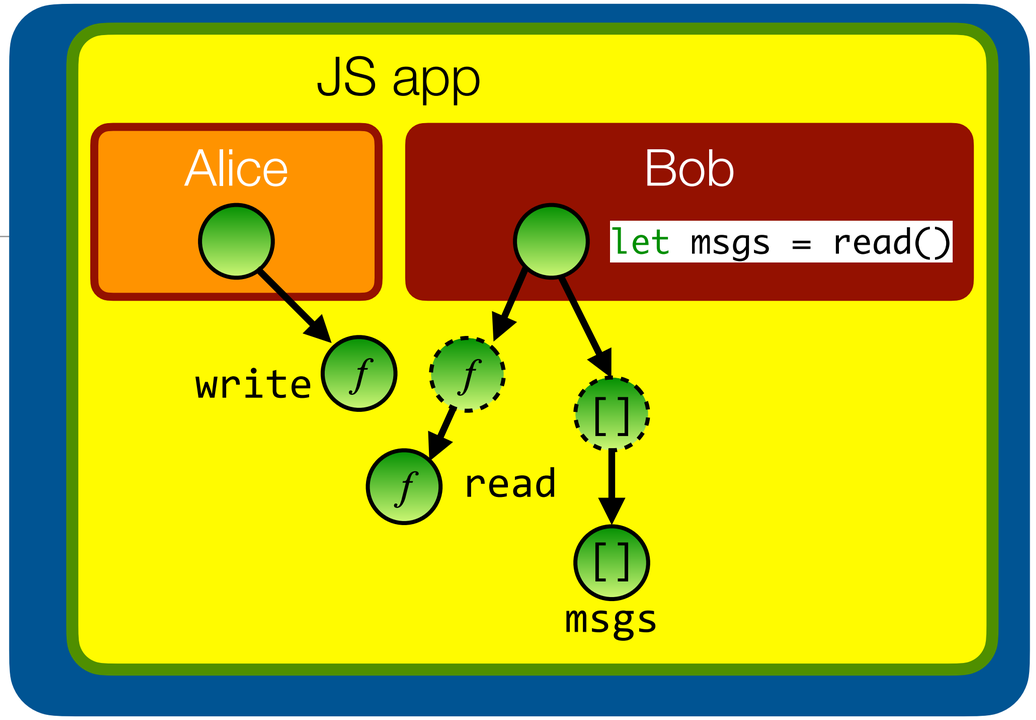

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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]);
}
```

Membranes are generalized caretakers

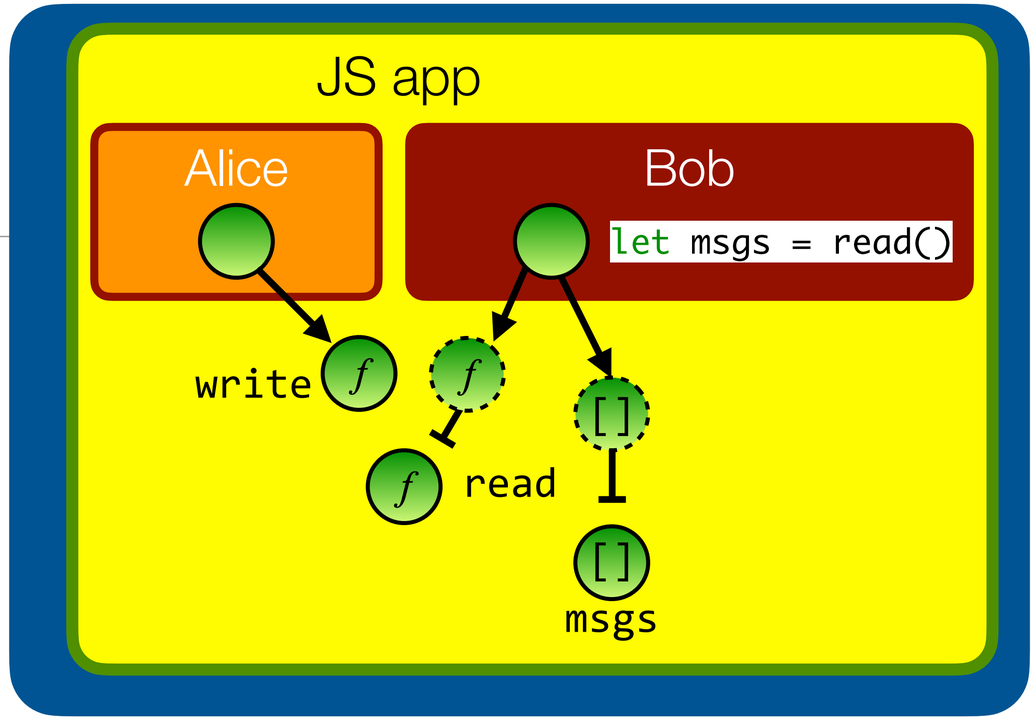
- Proxy *any* object reachable from the log

```
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);
```

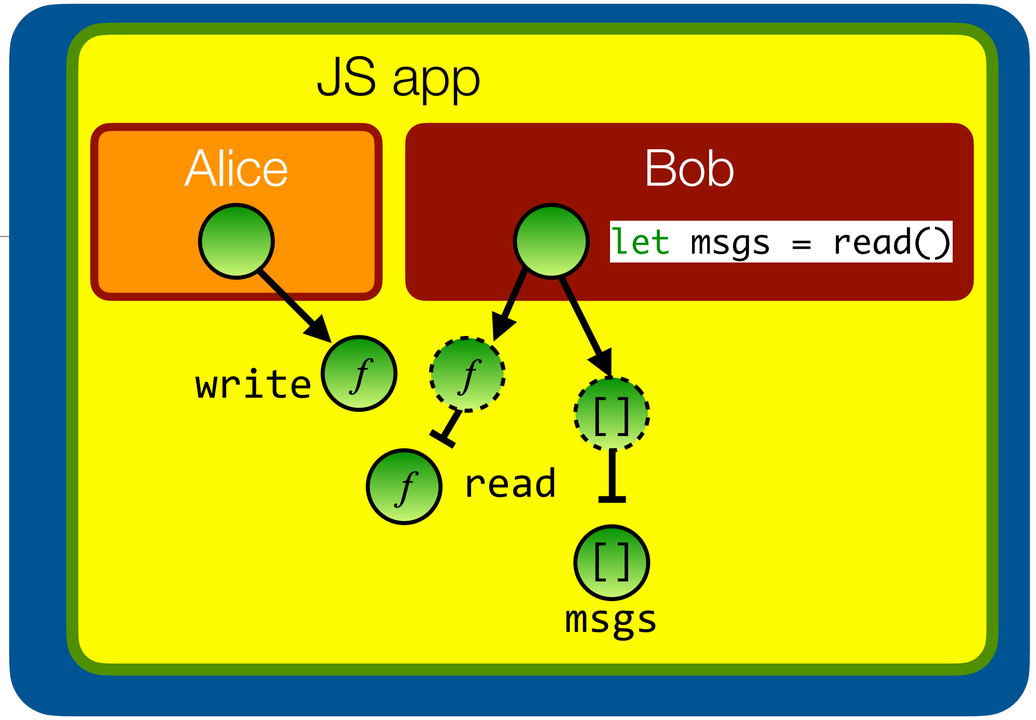


Membranes are generalized caretakers

```
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();
```



Membranes are generalized caretakers



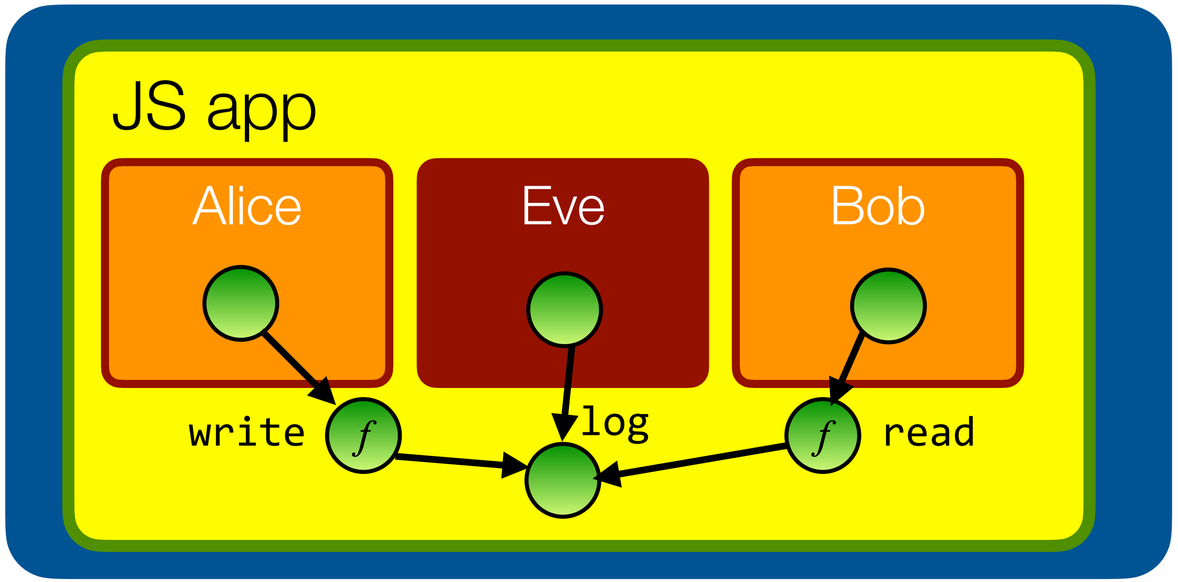

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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();
```

Deep dive at tvcutsem.github.io/membranes

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

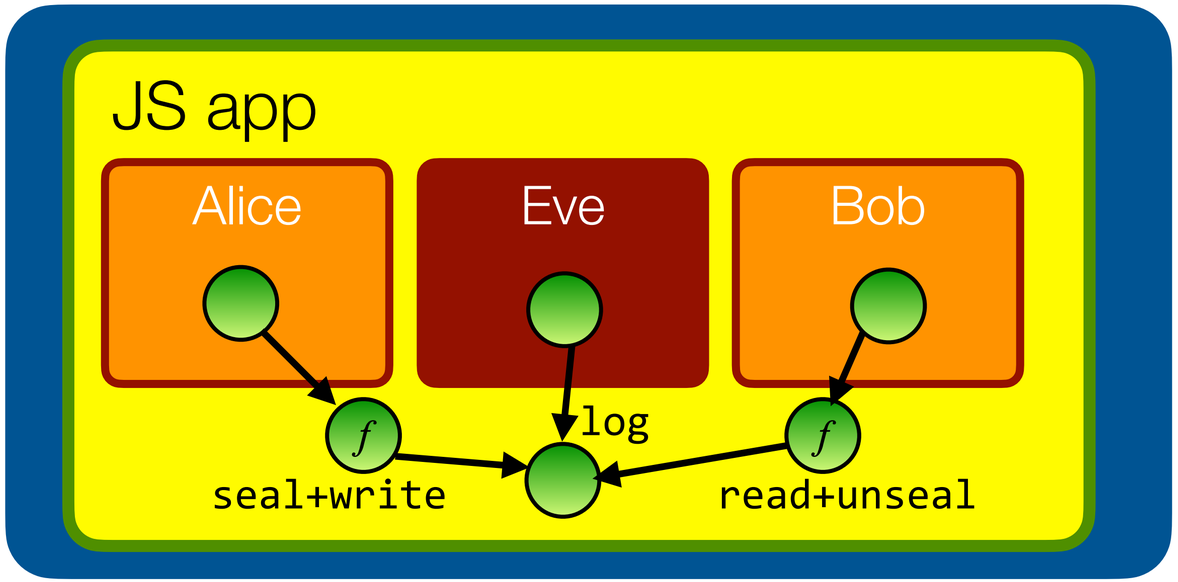
```
import * as alice from "alice.js";
import * as bob from “bob.js";
import * as eve from “eve.js";

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 of crypto, but in-process and without any actual crypto
- 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);
```



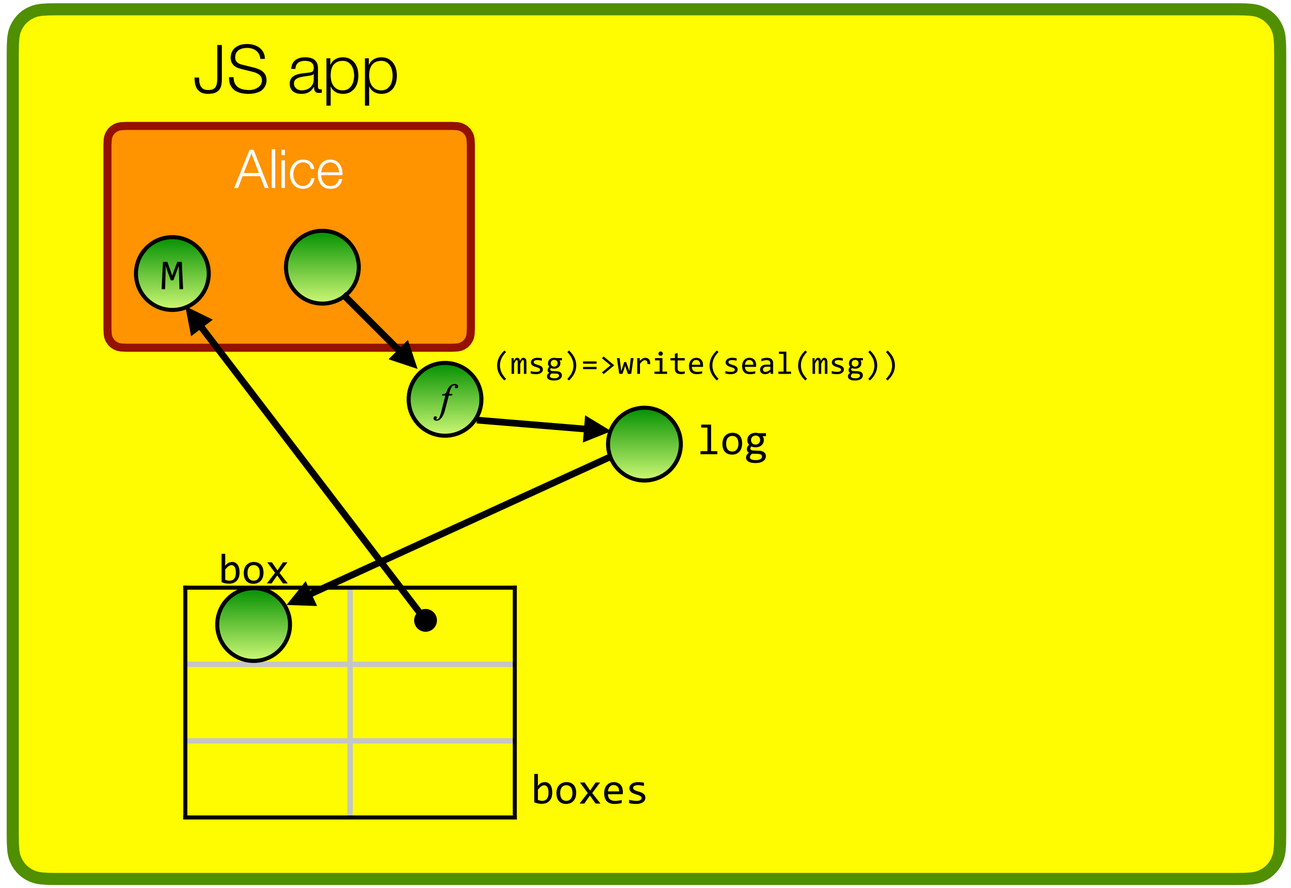
Sealer/unsealer pairs

```
function makeSealerUnsealerPair() {
  const boxes = new WeakMap();
  function seal(value) {
    const box = harden({});
    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)

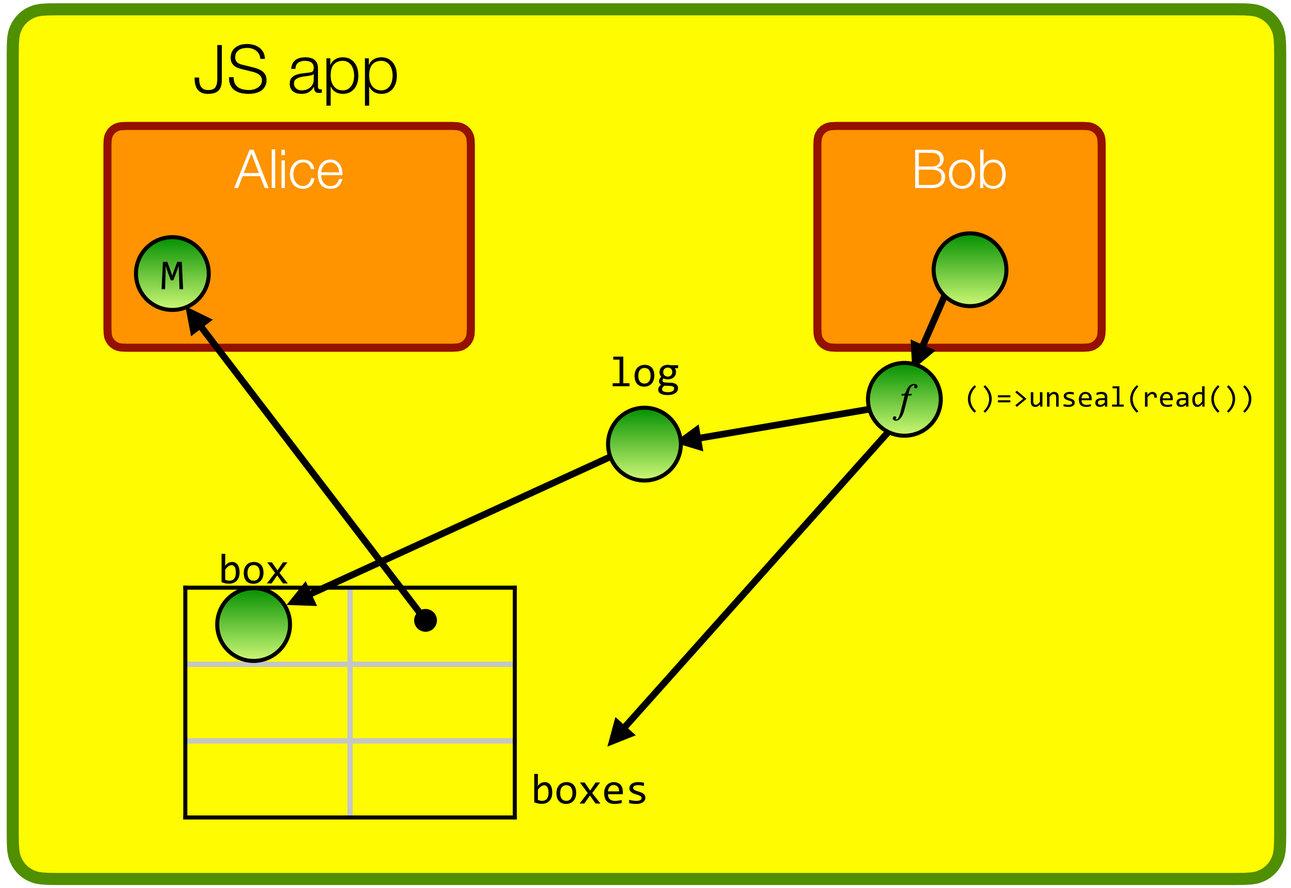
Alice stores only a meaningless ‘box’ object in the log

```
function makeSealerUnsealerPair() {
  const boxes = new WeakMap();
  function seal(value) {
    const box = harden({});
    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]);
}
```



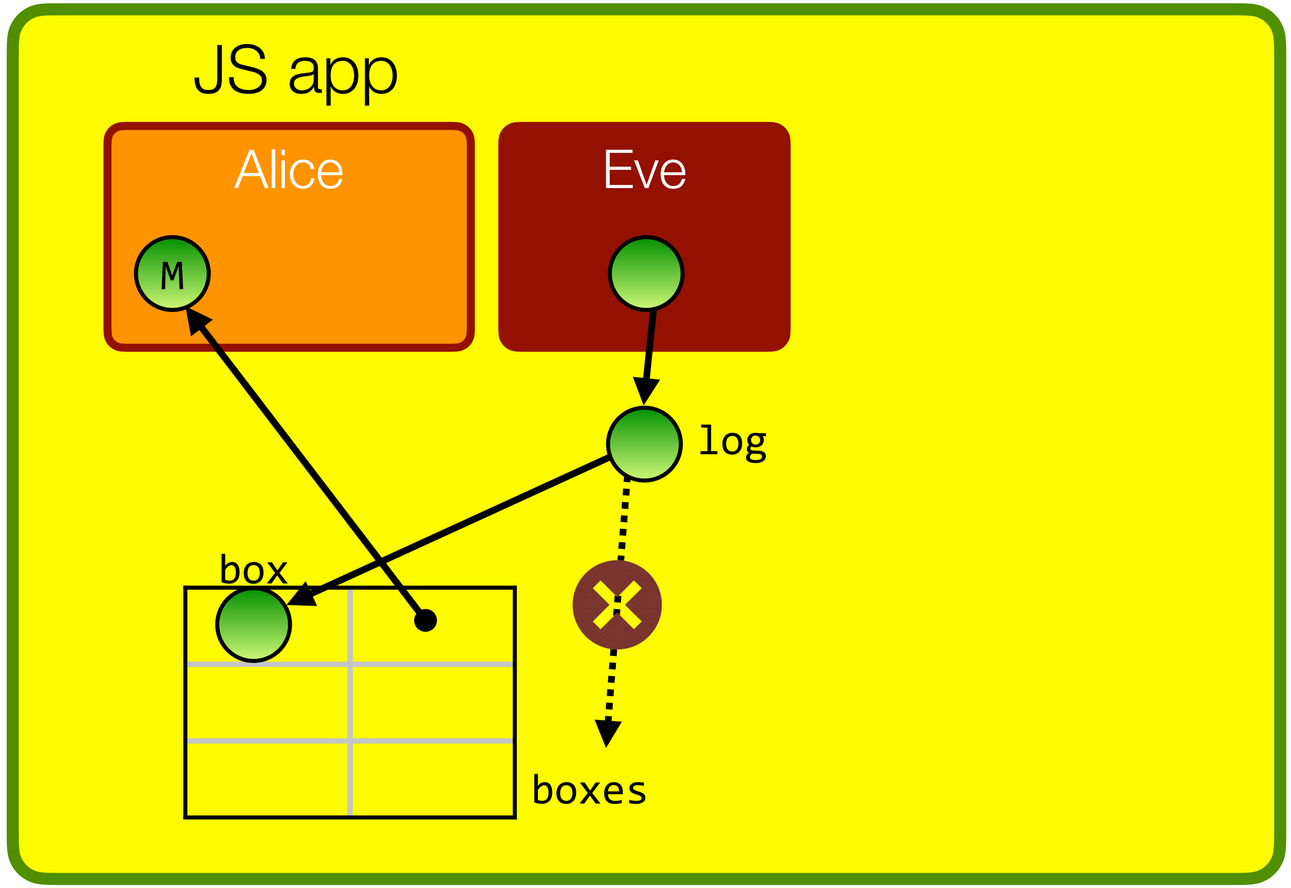
Bob can read the box from the log, and retrieve M via the unsealer

```
function makeSealerUnsealerPair() {
  const boxes = new WeakMap();
  function seal(value) {
    const box = harden({});
    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]);
}
```



Eve can access the box, but not the unsealer. She can’t unbox.

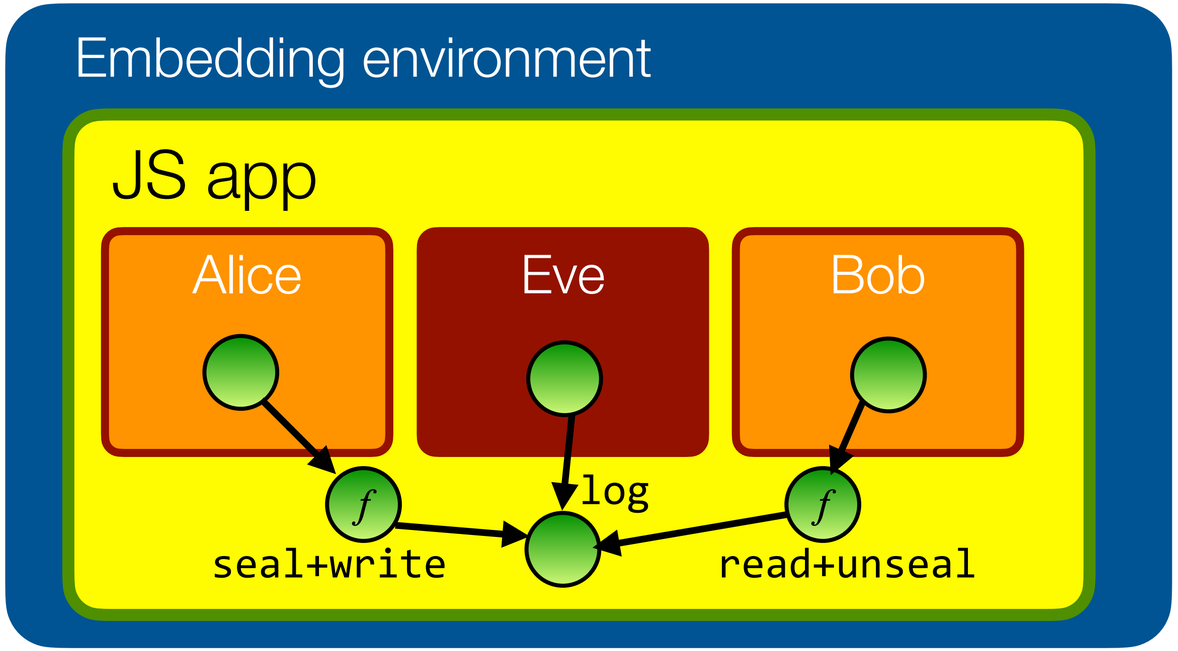
```
function makeSealerUnsealerPair() {
  const boxes = new WeakMap();
  function seal(value) {
    const box = harden({});
    boxes.set(box, value);
    return box;
  }
  function unseal(box) {
    if (boxes.has(box)) {
      return boxes.get(box);
    } else {
      throw new Error("invalid box");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



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);
```



These patterns are used in industry

Google Caja

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

Mozilla Firefox

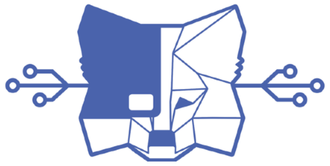
Uses **membranes** to isolate site origins from privileged JS code

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

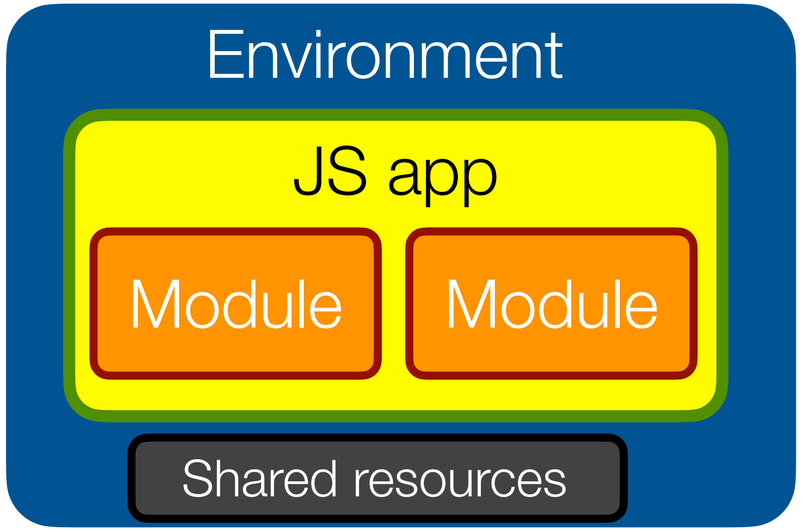
Agoric Zoe

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**



TECHORAMA

Architecting Robust JavaScript Applications

A practitioner's guide to Secure ECMAScript

Tom Van Cutsem

@tvcutsem

Thanks for listening!

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) 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
- The SES secure coding guide: https://github.com/endojs/endo/blob/master/packages/ses/docs/secure-coding-guide.md

Further Reading

- Compartments: https://github.com/tc39/proposal-compartments and https://github.com/Agoric/ses-shim
- Realms: https://github.com/tc39/proposal-realms and github.com/Agoric/realms-shim
- SES: https://github.com/tc39/proposal-ses and https://github.com/endojs/endo/tree/master/packages/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