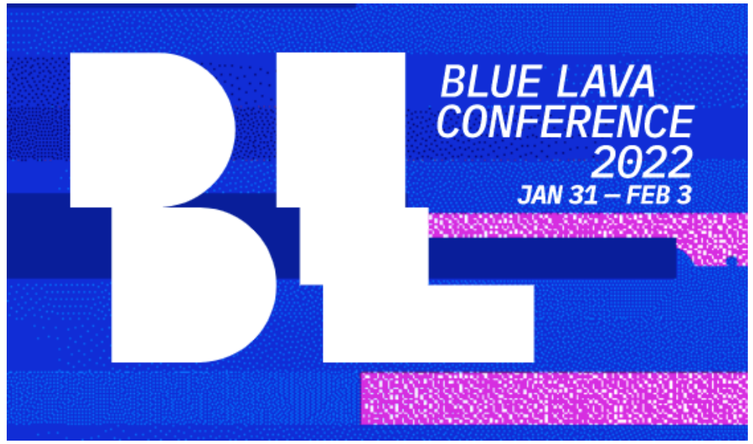
A Practitioner’s guide to Hardened JavaScript

Improving JavaScript (d)app security by practicing extreme modularity

Tom Van Cutsem


@tvcutsem



About me

- Computer scientist at Nokia Bell Labs
- Programming language designer
- In a past life: TC39 member and active contributor to ECMAScript standards
- Turning JS into a language for smart contracts before Ethereum was born

tvcutsem.github.io

@tvcutsem

A software architecture view of app security

“Security is just the extreme of Modularity”

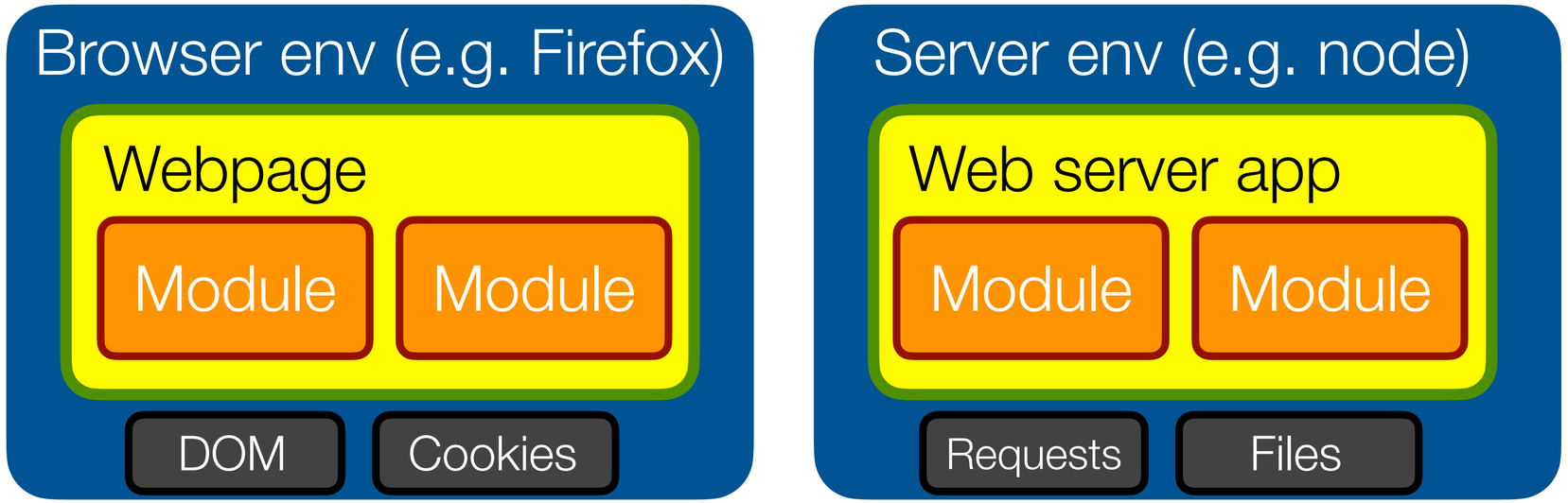
- Mark S. Miller
(Chief Scientist, Agoric)

Modularity: avoid needless dependencies (to prevent bugs)

Security: avoid needless vulnerabilities (to prevent exploits)

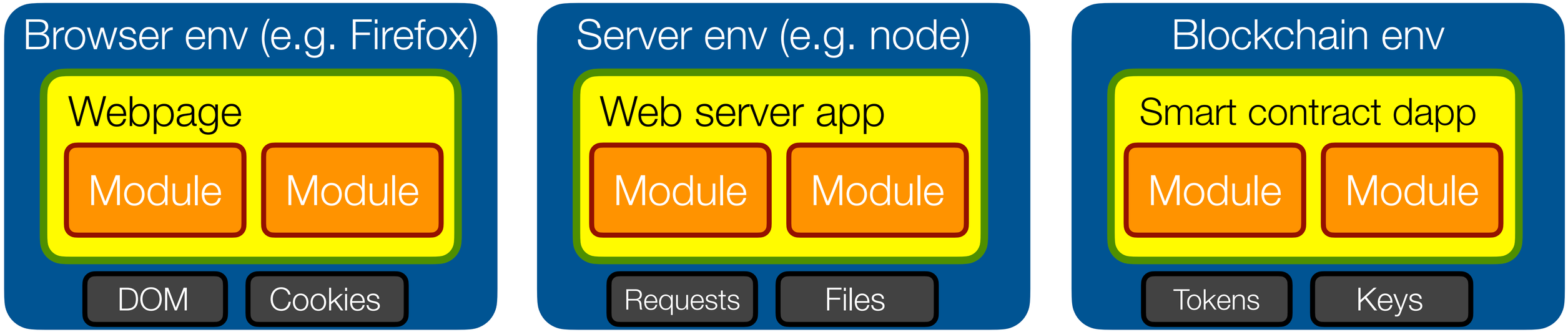
It’s all about **trust**

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

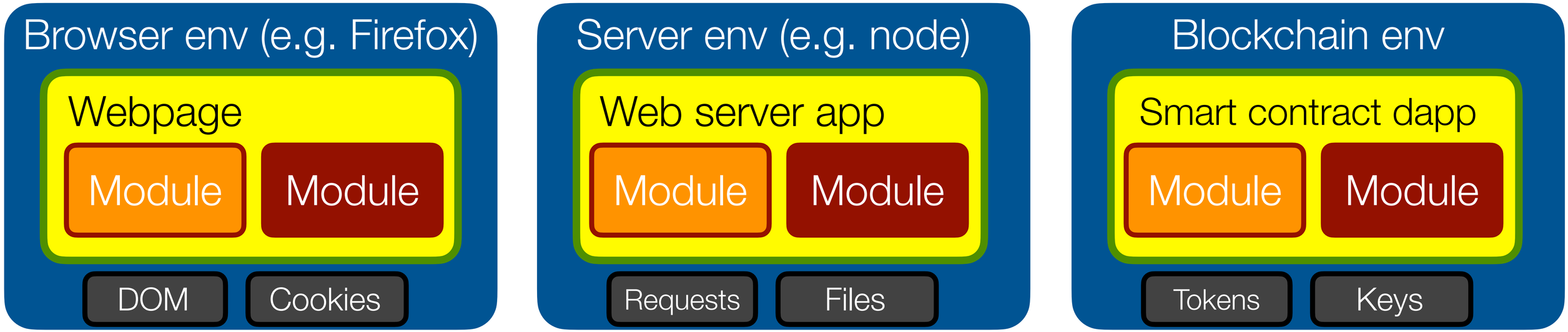


It’s all about **trust**

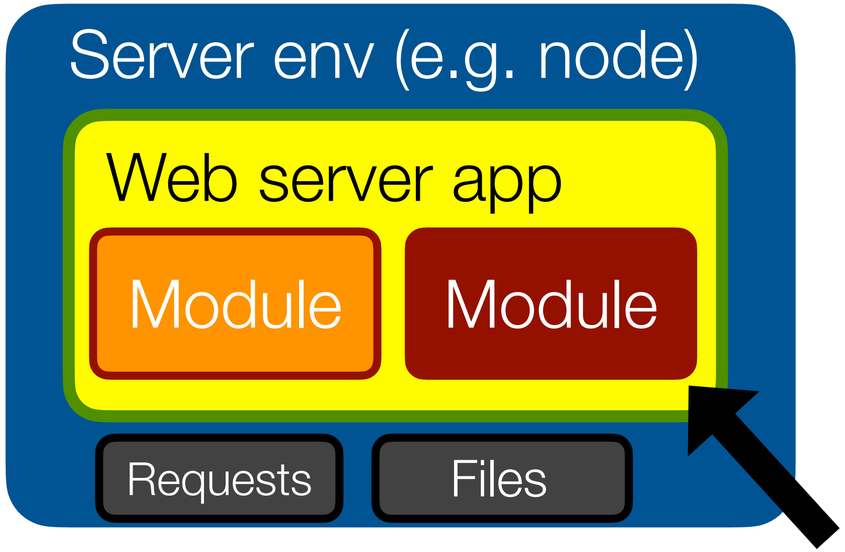
Even more critical in Web3 where code may access digital assets or wallets



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



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)

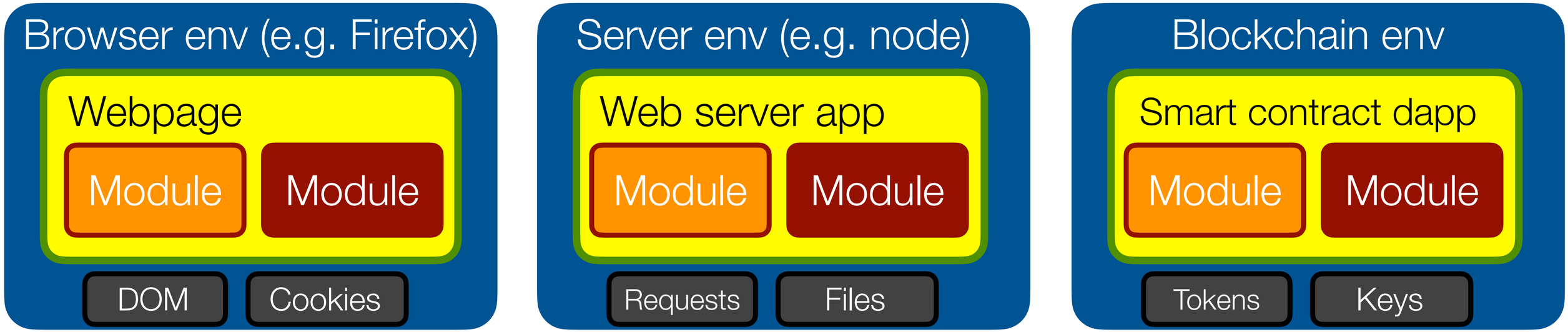
Node.js package tried to plunder Bitcoin wallets

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

(source: theregister.co.uk)

Avoiding interference is the name of the game

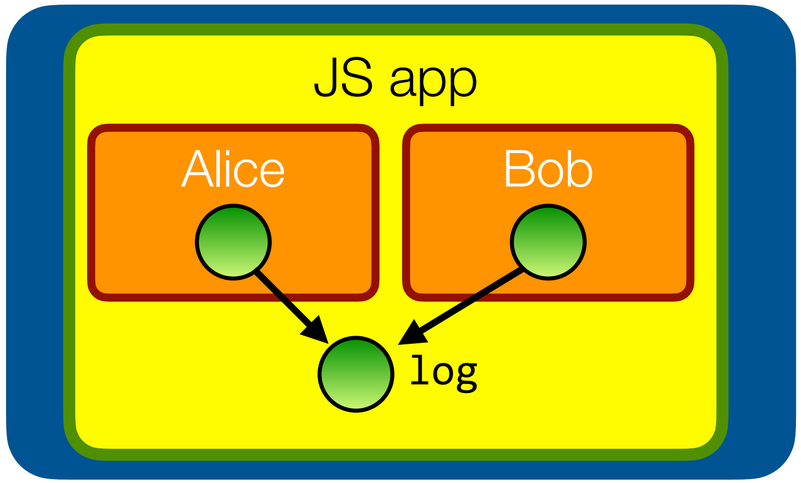
- Shield important resources/APIs from modules that don't need access
- Apply **Principle of Least Authority** (POLA) to application design



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.

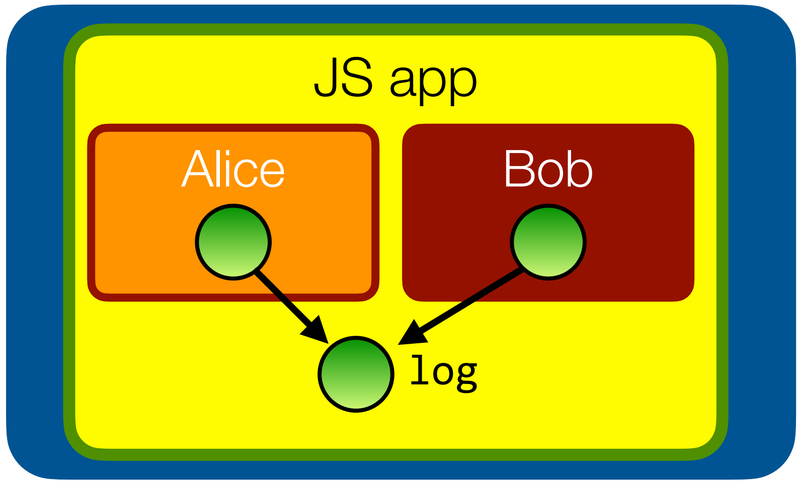
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

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



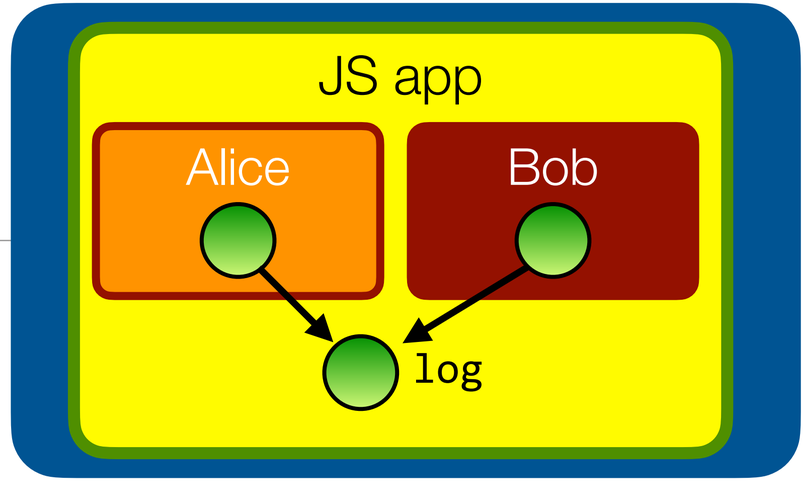
Running example: apply POLA to a basic shared log

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);
```



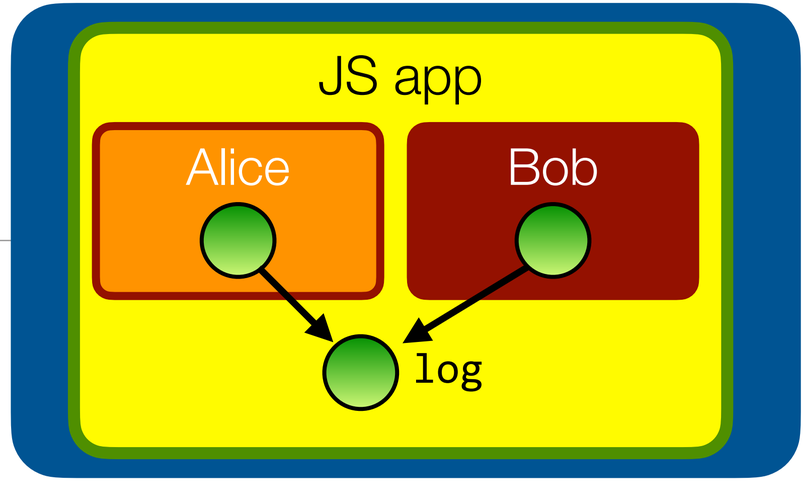
Bob has way too much authority!




```
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”);
}
```

How to solve “prototype poisoning” attacks?



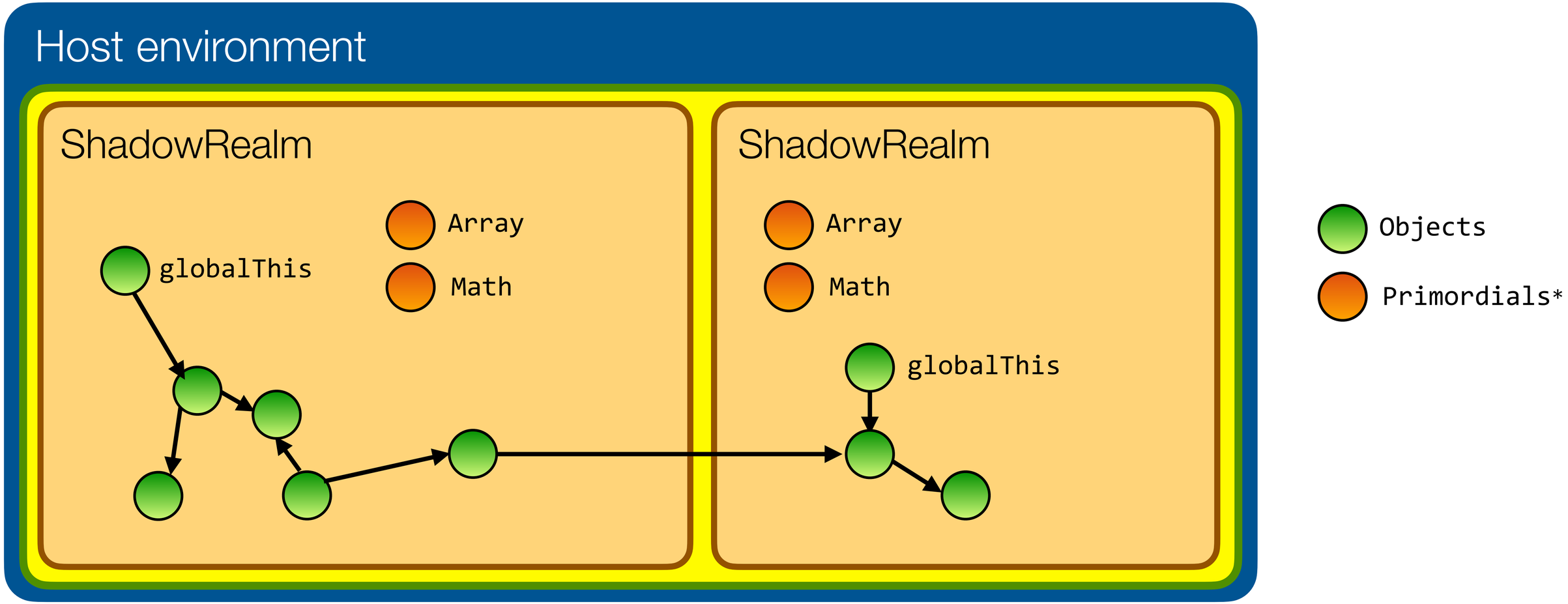
Load each module in its own environment, with its own set of “primordial” objects

```
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”);
}
```

ShadowRealms (TC39 Stage 3 proposal)

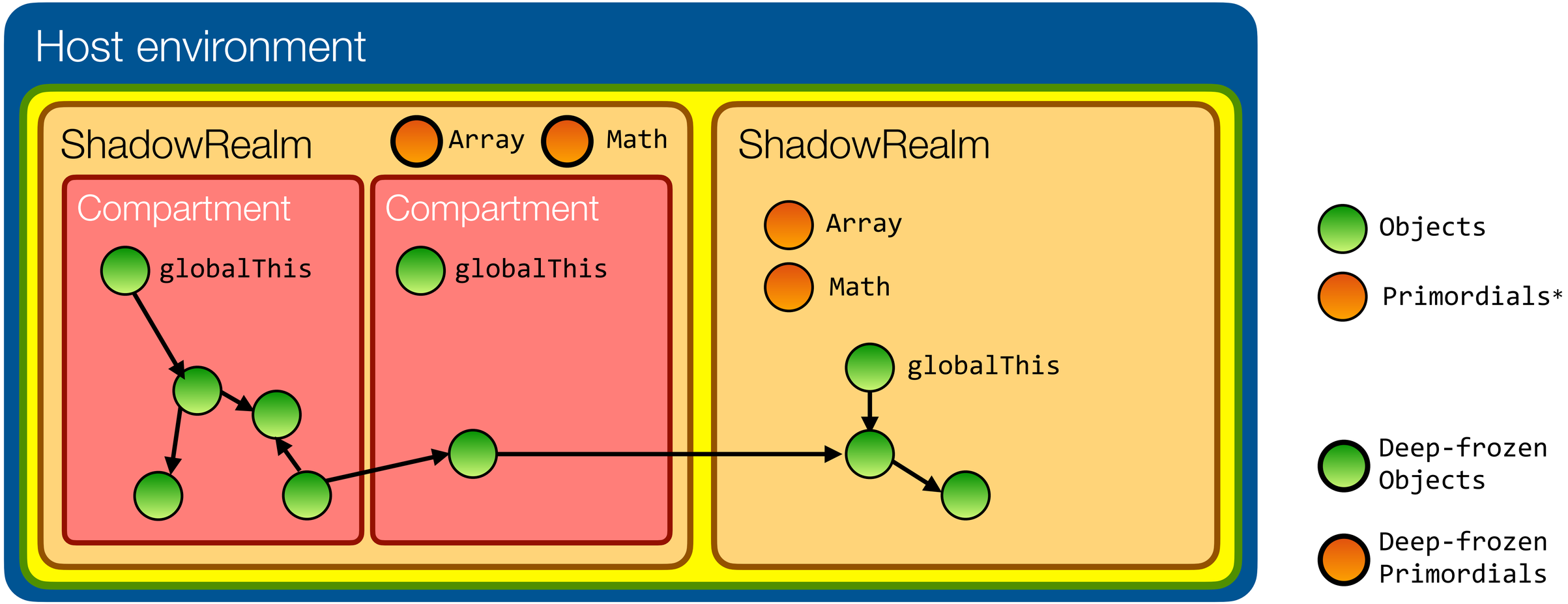
Intuitions: “iframe without DOM”, “principled version of node’s `vm` module”



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

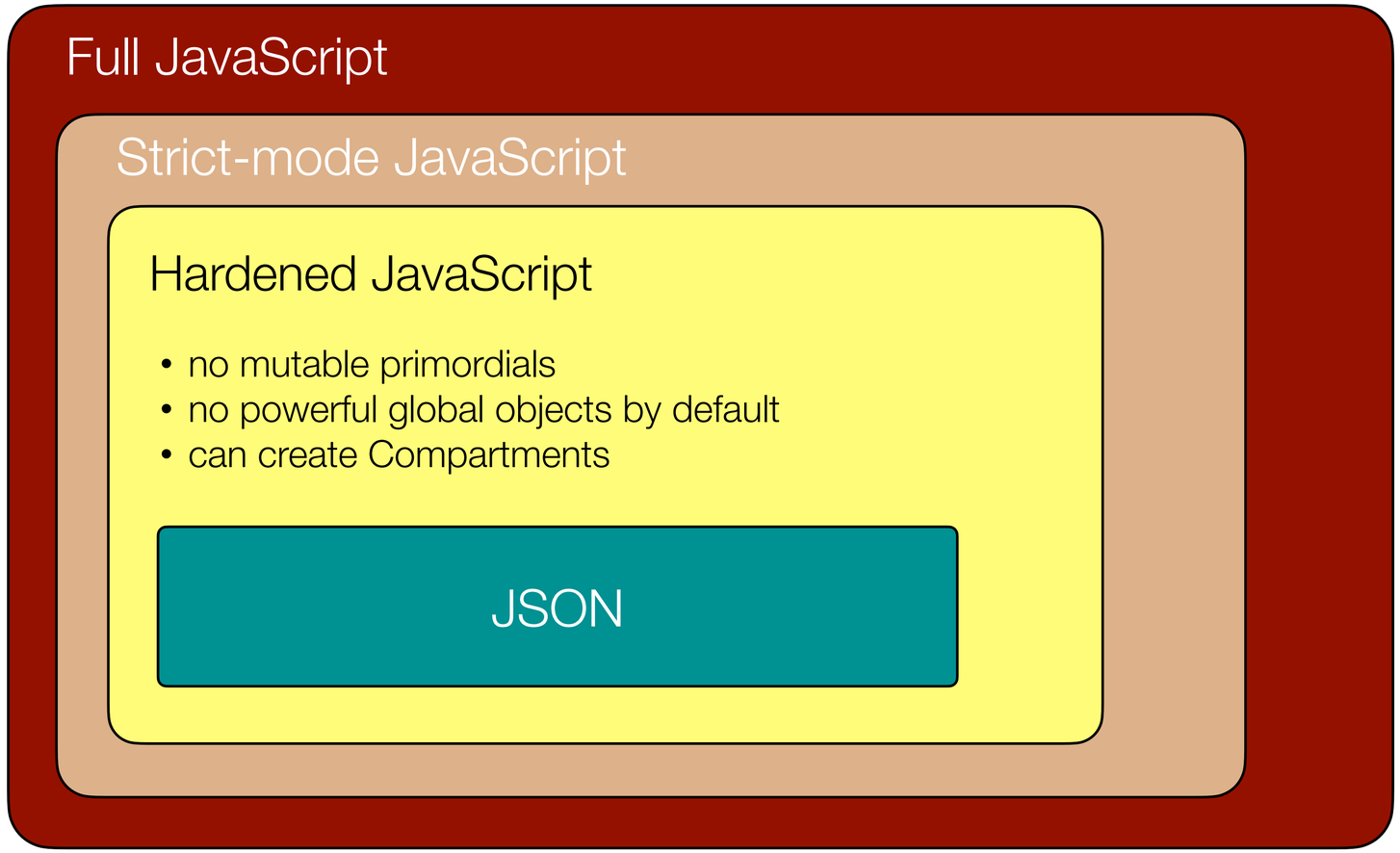
Compartments (TC39 Stage 1 proposal)

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



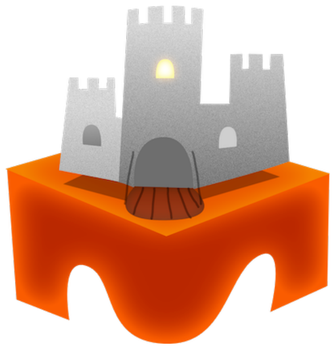
(inspired by the diagram at https://github.com/Agoric/Jessie)

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

```
import ‘ses’;
lockdown();
```

LavaMoat

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

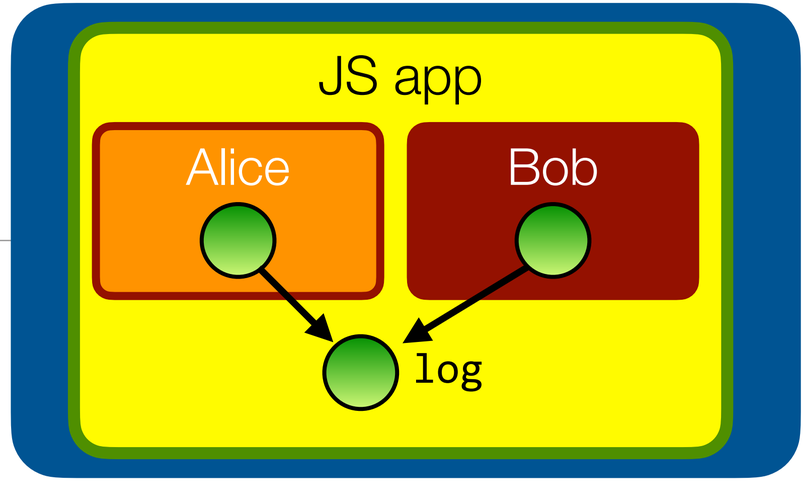


https://github.com/LavaMoat/lavamoat


```
"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
  }
},
```

Back to our example

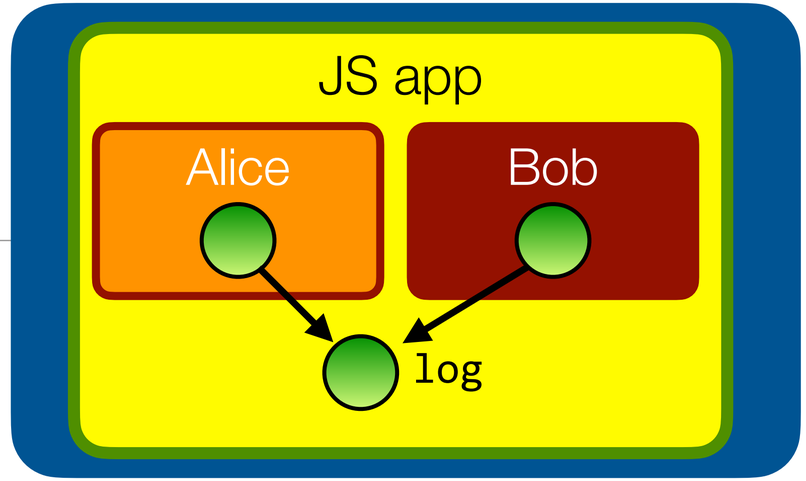
With Alice and Bob's code running in their own Compartment, we mitigate the poisoning attack




```
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

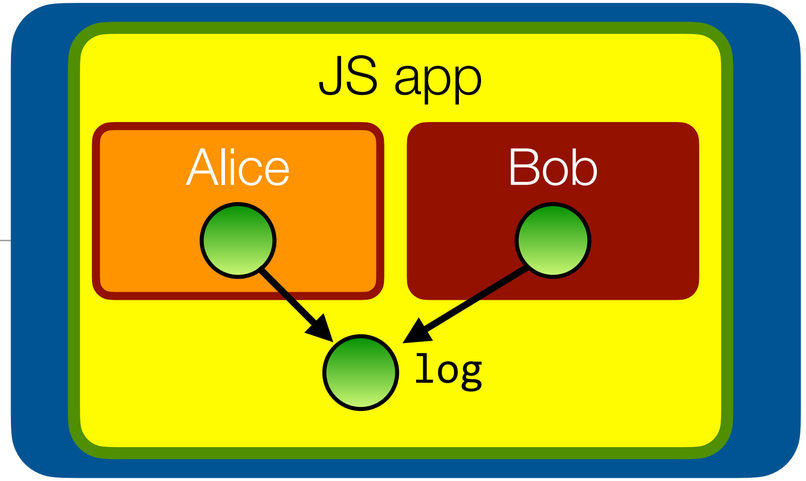



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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**

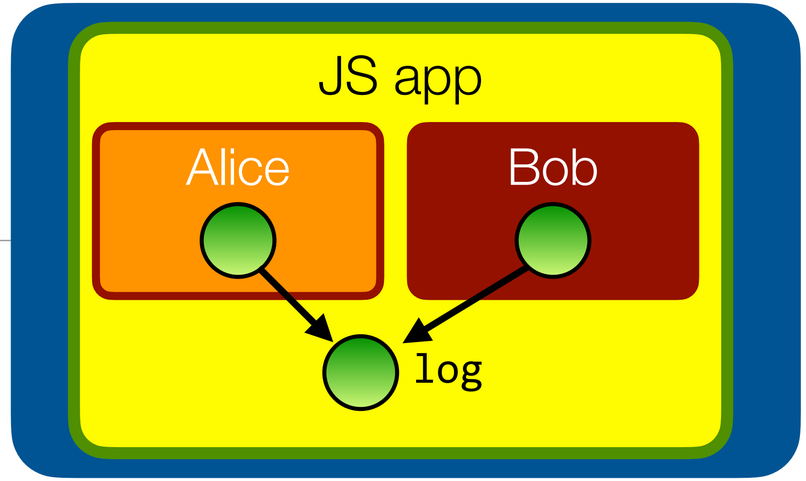


Object.freeze (ES5) makes property bindings (not their values) immutable

```
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.

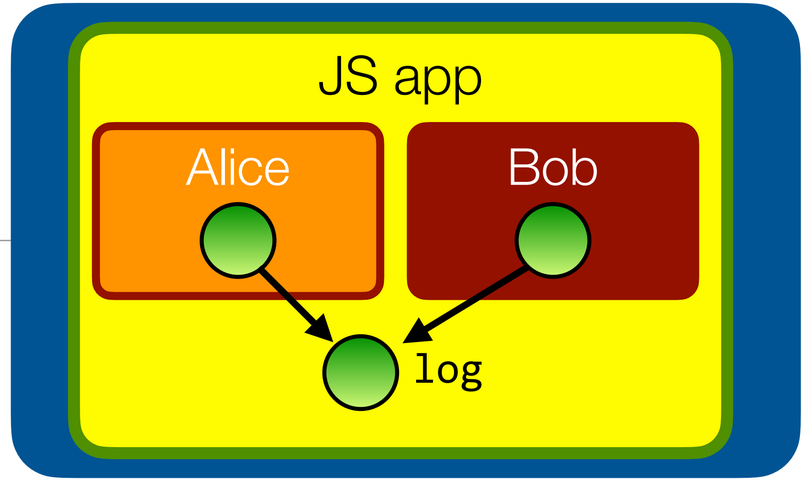


Functions are mutable too. Freeze doesn't recursively freeze the object's functions.

```
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

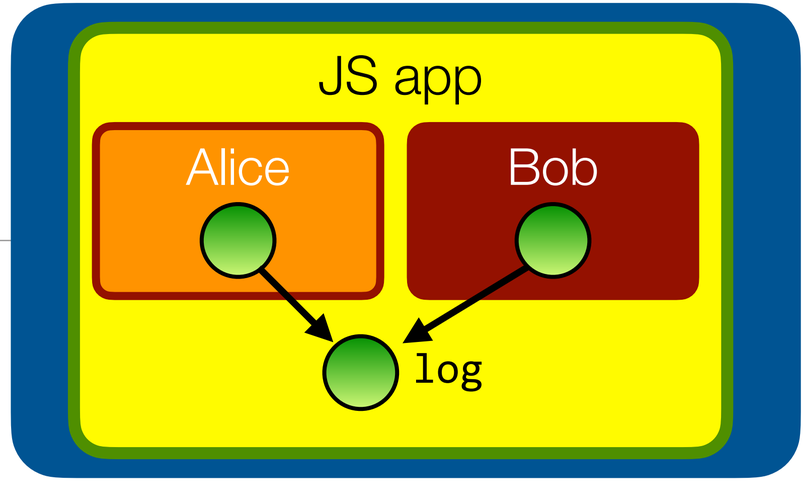


Hardened JS provides a `harden` function that "deep-freezes" an object

```
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" };
```

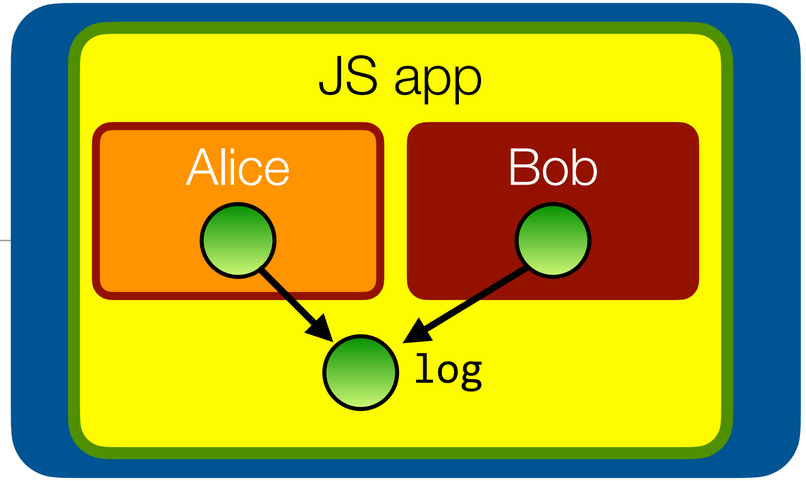
Two down, two to go




```
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” };
```

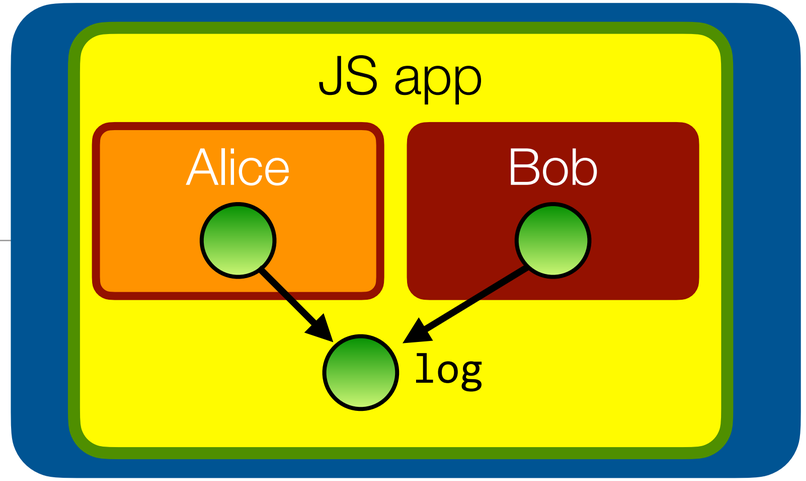
Two down, two to go




```
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

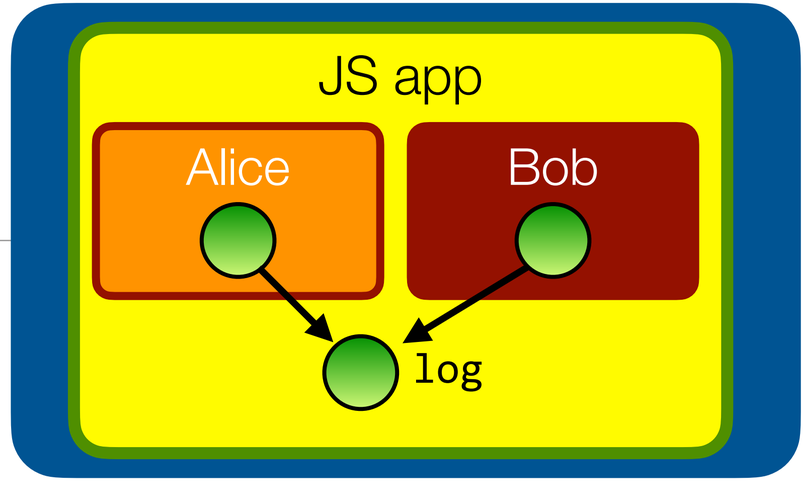


- Modify `read()` to return a copy of the mutable state.
- Even better would be to use a more efficient copy-on-write or “persistent” data structure (see immutable-js.com)

```
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” };
```

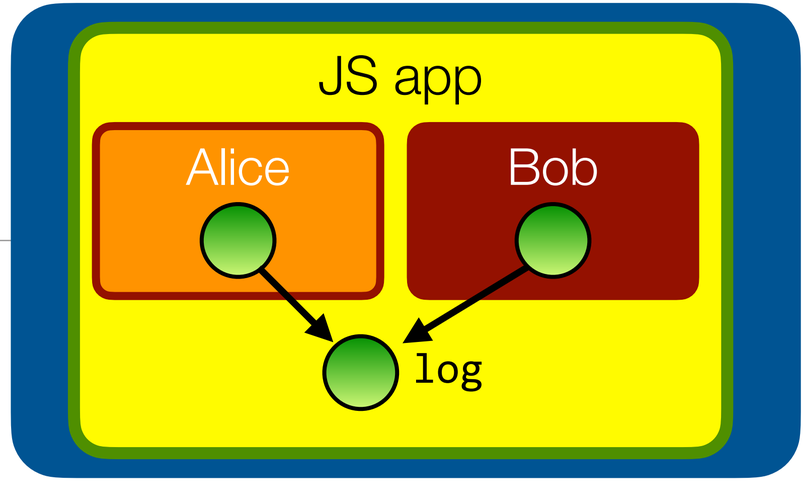
Three down, one to go




```
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?

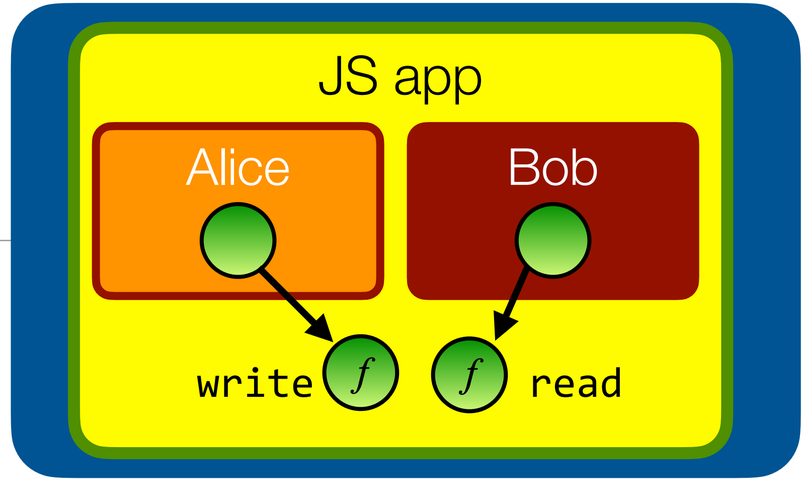
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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?

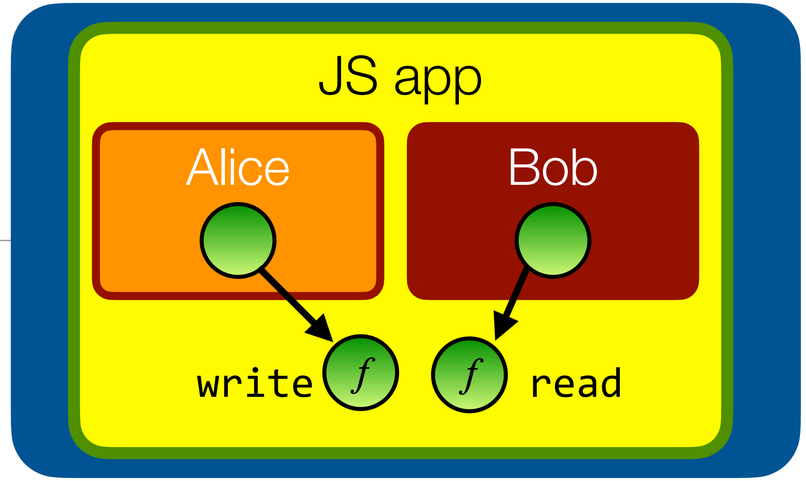



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.

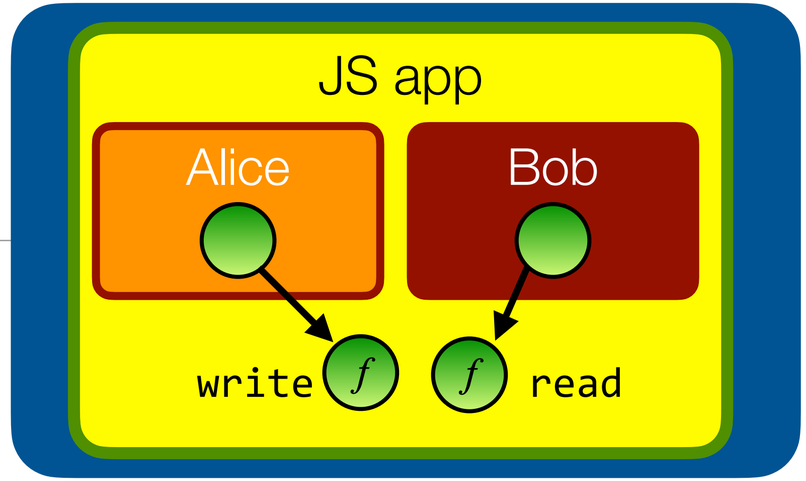
To avoid, only ever pass bound functions




```
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.

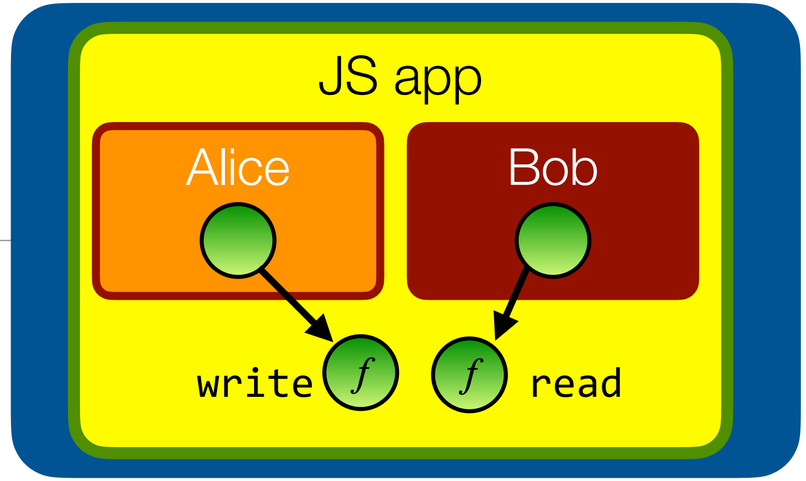



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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” };
```

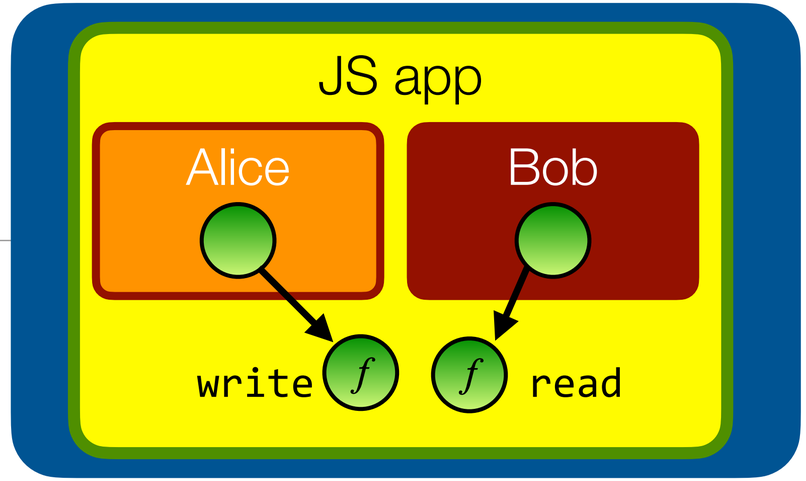
Is there a better way to write this code?

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



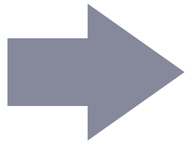

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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 instead of using classes or prototypes

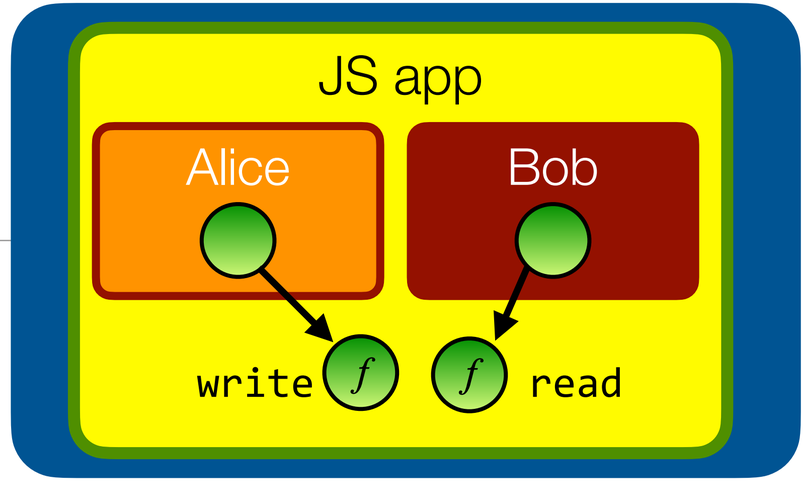
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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)

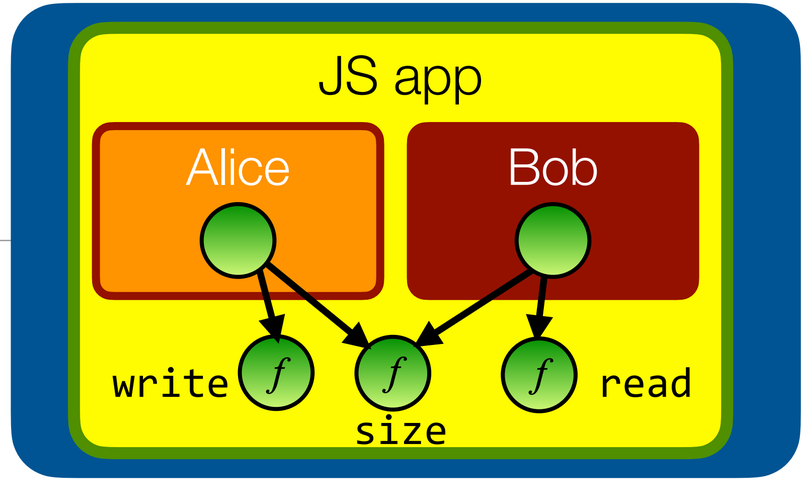
Use the Function as Object pattern



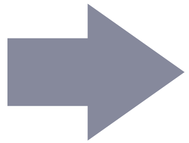

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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.



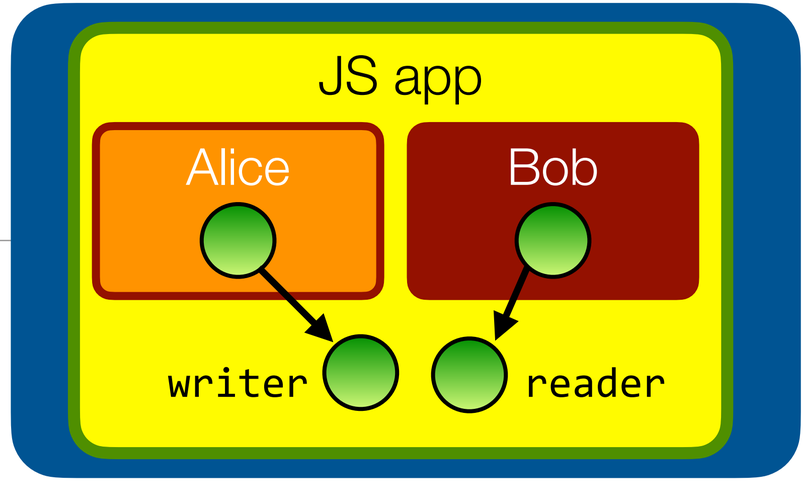

```
import * as alice from "alice.js";
import * as bob from “bob.js";

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);
```



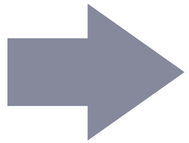
```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

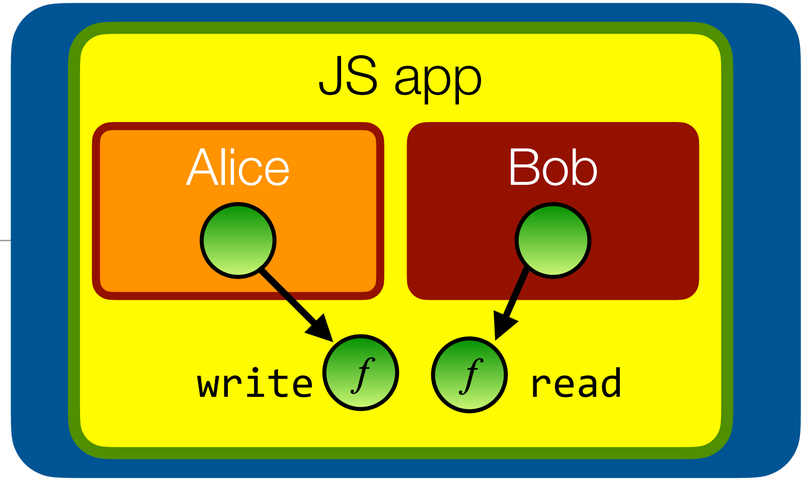



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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);
```

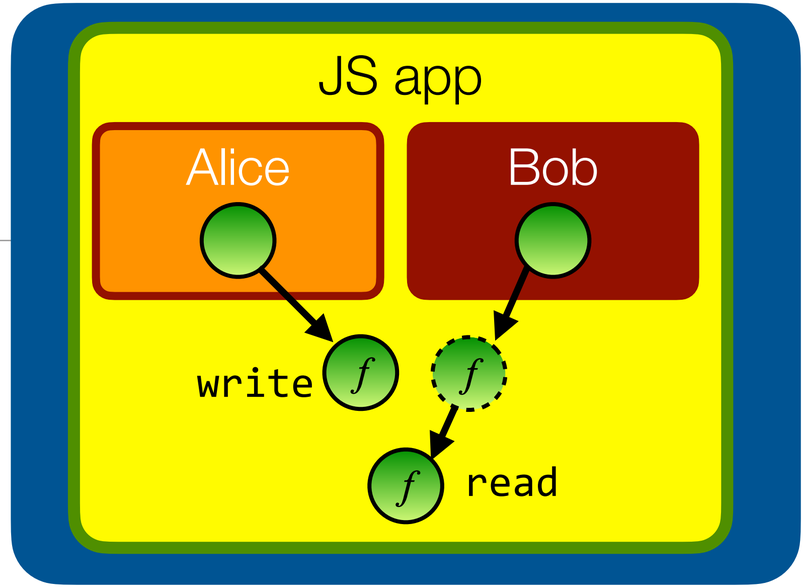
Further limiting Bob's authority



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);
```

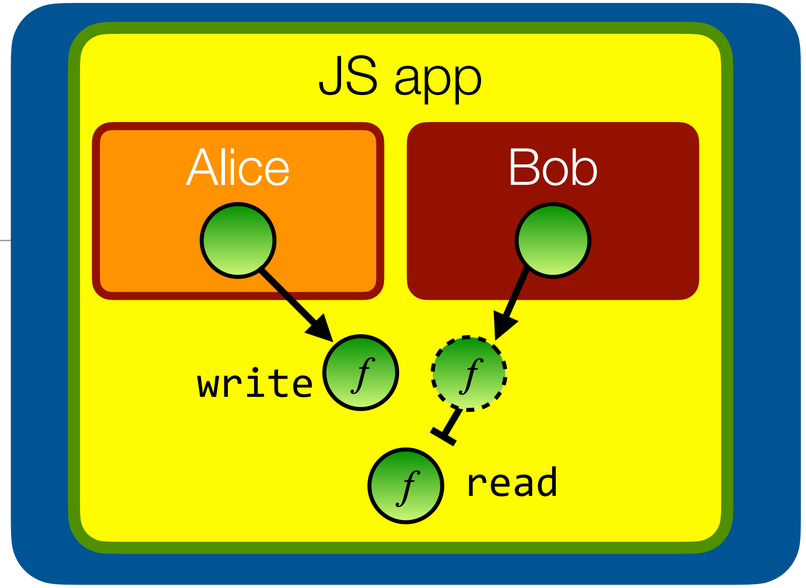
Use **caretaker** to insert access control logic



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);
```

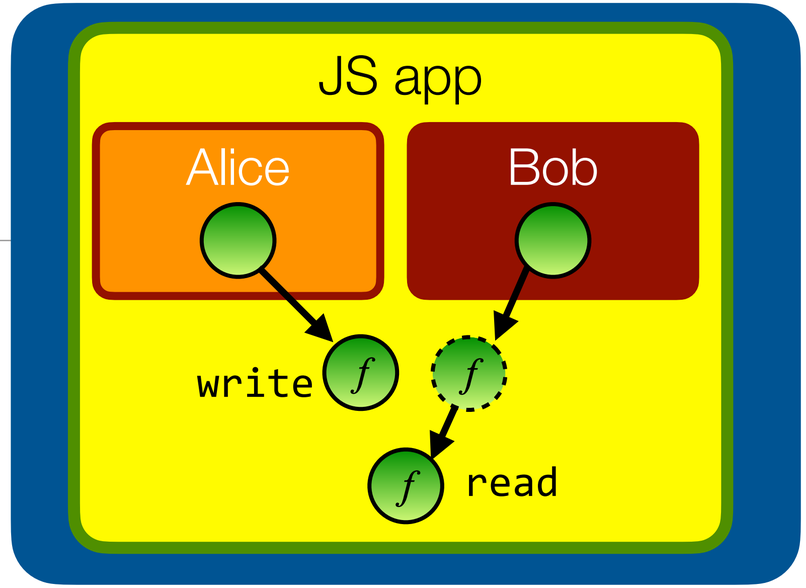
Use **caretaker** to insert access control logic



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();
```

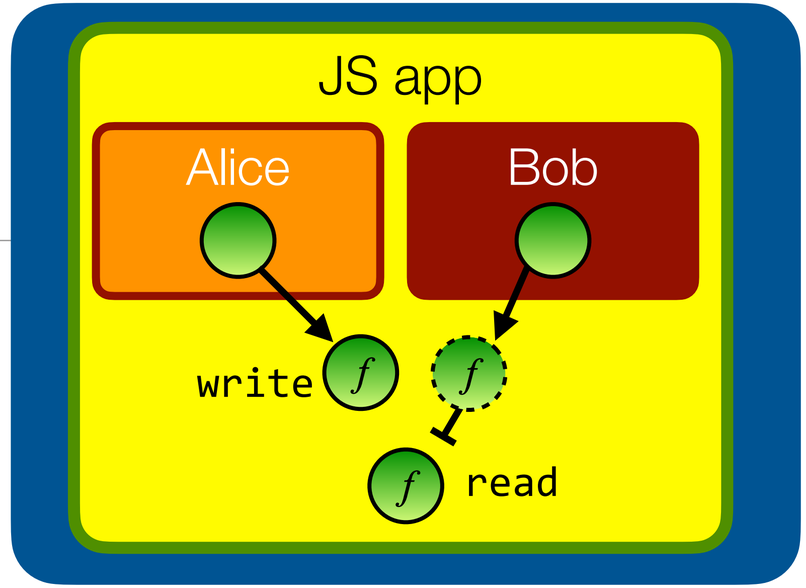
A caretaker is just a proxy object




```
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);

// 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

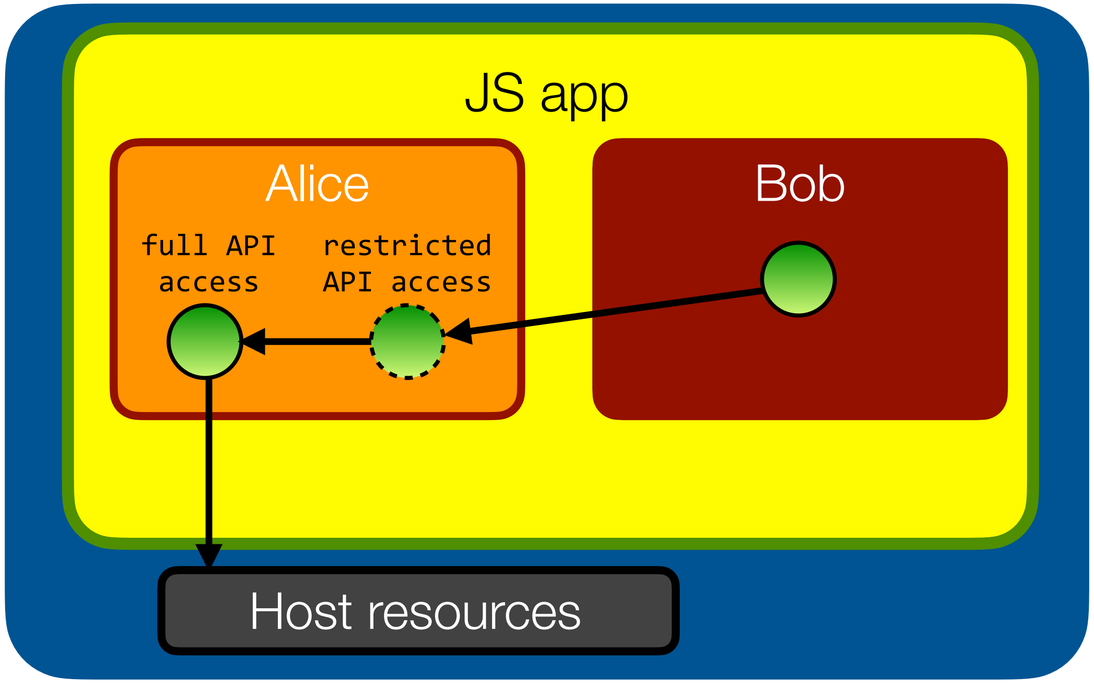



```
import * as alice from "alice.js";
import * as bob from “bob.js";

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

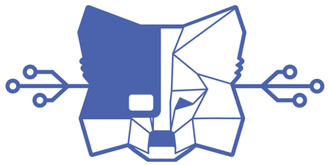
- Expose powerful objects through restrictive proxies to third-party code
- For example, Alice might give Bob read-only access to a specific subdirectory of her file system



These 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

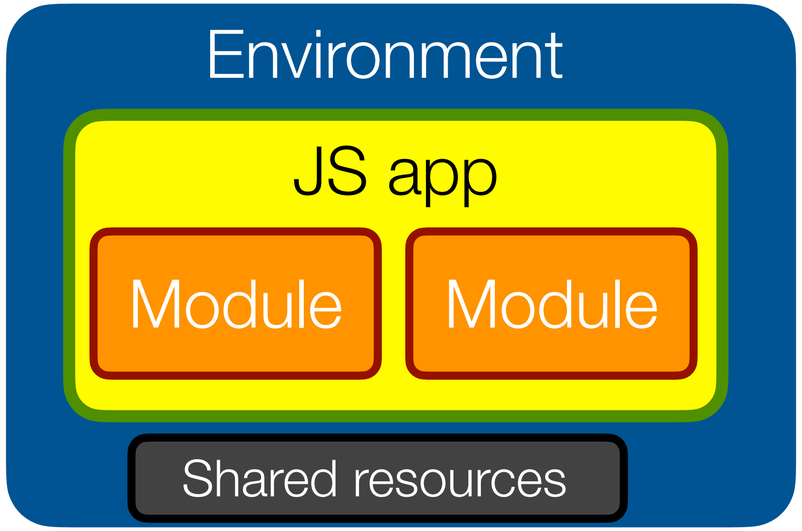
Uses **membranes** to isolate site origins from privileged JS code

Salesforce Lightning

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

The take-away messages

- Modern JS apps are **composed from many modules**. You can't trust them all.
- Apply the "principle of least authority" to **limit trust**.
  - **Isolate modules** using Hardened JS & Lavamoat
  - Let modules **interact safely** using patterns such as facets, caretaker, membranes, taming, …
- Understanding these patterns is **critical in Web3** where code directly interacts with digital assets



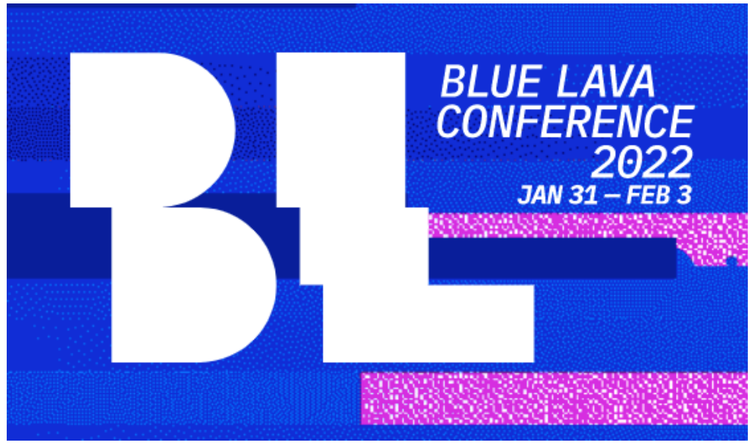
Improving JavaScript (d)app security by practicing extreme modularity

A Practitioner’s guide to Hardened JavaScript

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
- Kate Sills and Kris Kowal at Agoric for helpful comments on earlier versions of this talk
- 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
- 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