Architecting Robust JavaScript Applications

Tom Van Cutsem


@tvcutsem

About me

- Computer scientist with broad experience in academia and industry
- Past TC39 member and active contributor to ECMAScript standards
- Passionate user and advocate of JavaScript

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!

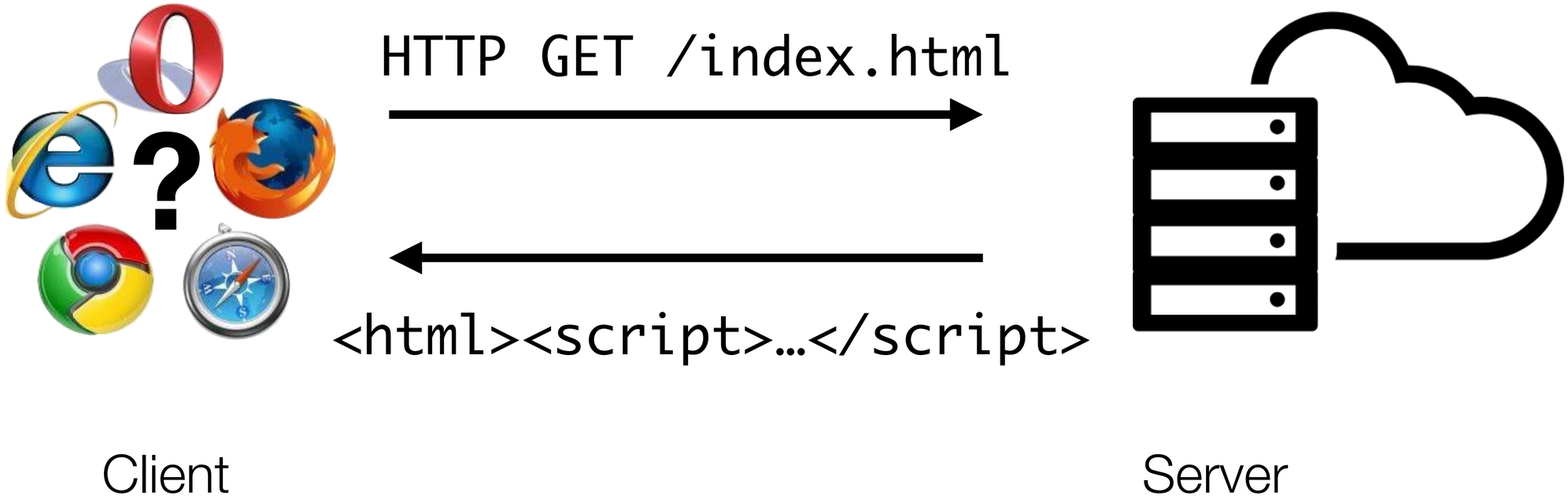
This Talk

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

Part I
The need for more robust JavaScript apps

JavaScript & the importance of standards

- As a website author, you don’t get to choose the execution platform!
- Remember the Browser Wars of the early 1990s



ECMAScript: “Standard” JavaScript

~~Carakan~~

V8

ChakraCore

Spidermonkey

V8

JavaScriptCore

V8

Java

GraalVM

A Tale of Two Standards Bodies

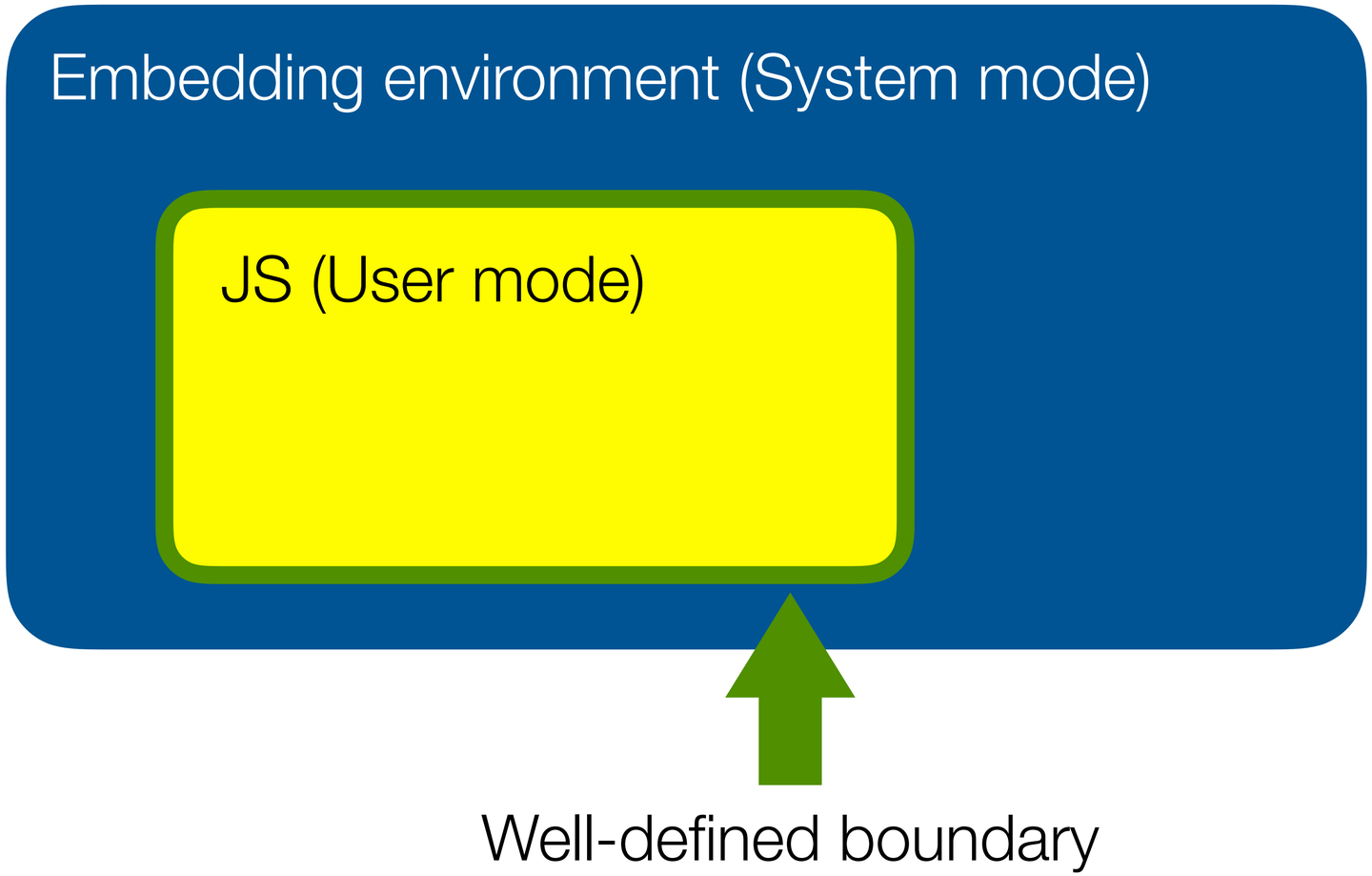
"Any organization that designs a system […] will produce a design whose structure is a copy of the organization's communication structure."

-- Melvyn Conway, 1967

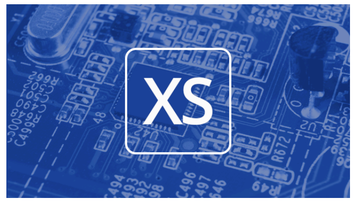
- Standardizes JavaScript
- Core language + small standard library
- Math, JSON, String, RegExp, Array, …
- **“User mode”**

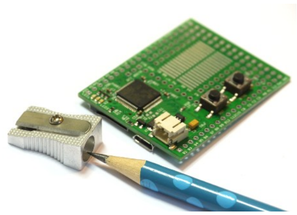
- Standardizes browser APIs
- Large set of system APIs
- DOM, LocalStorage, XHR, Media Capture, …
- **“System mode”**

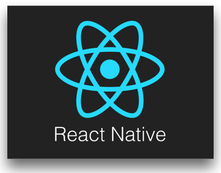
“User mode” separation makes JS an embeddable compute engine

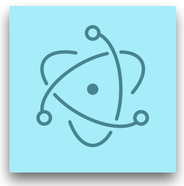


As a result, JavaScript used widely across tiers





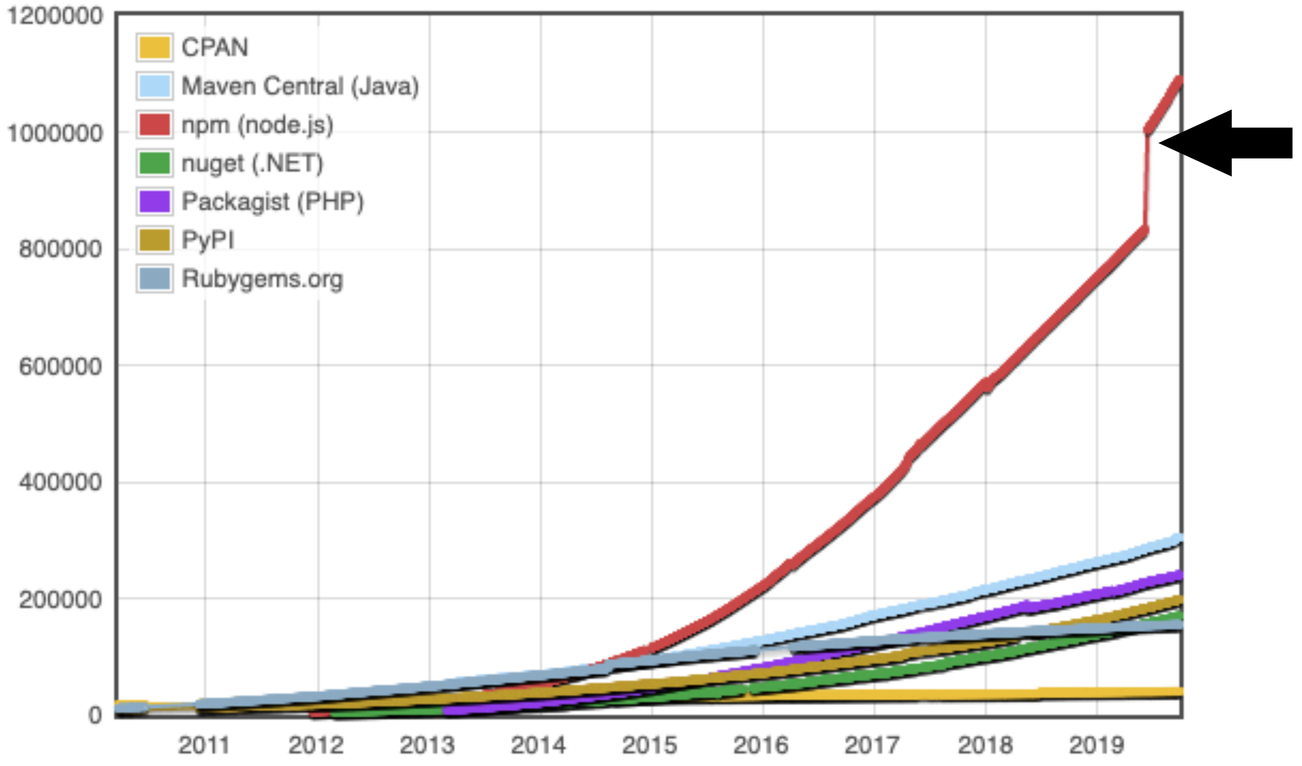




Embedded Mobile Desktop/Native Server Database

JavaScript applications are now built from thousands of modules

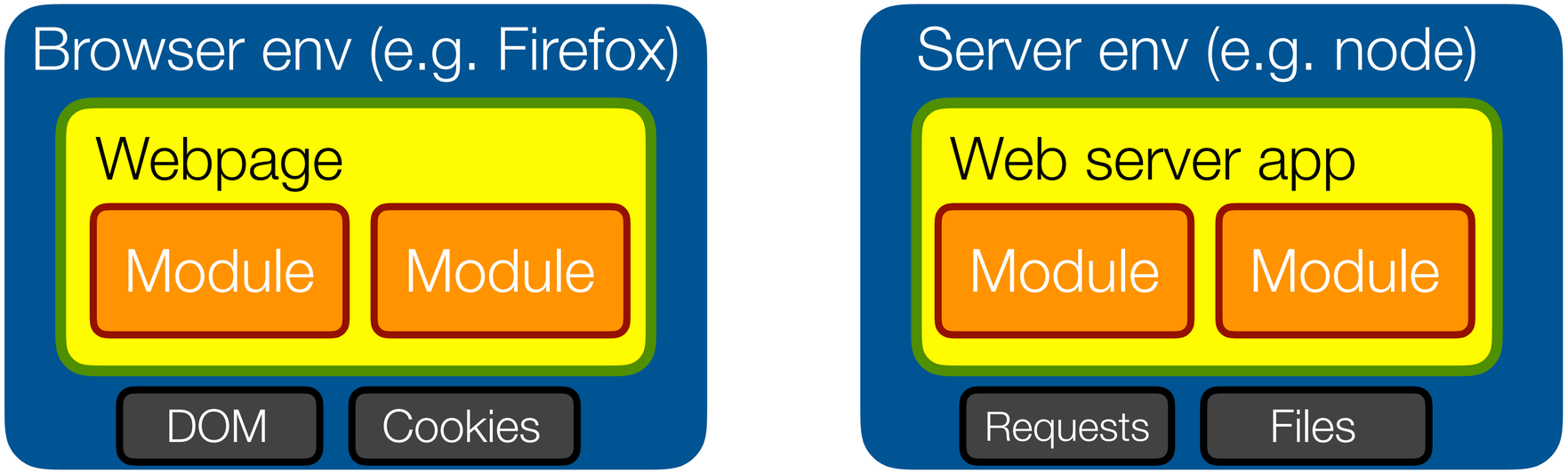
- Node package manager (NPM) is the world's largest package manager



(source: modulecounts.com, Sept. 2019)

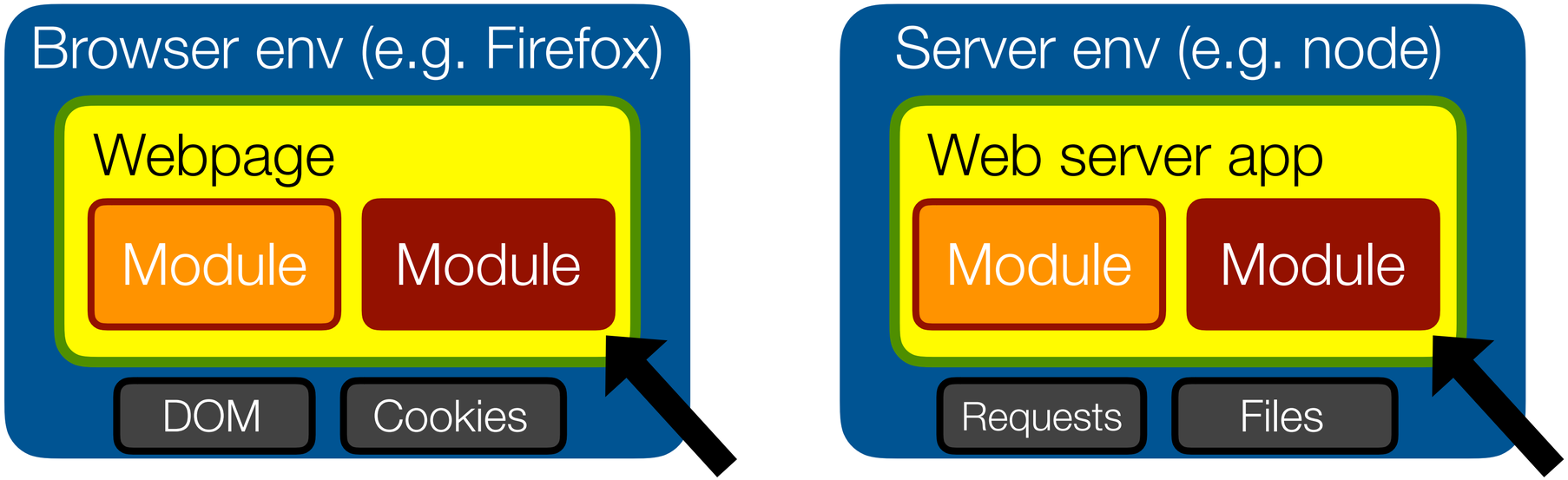
It’s all about **trust**

- 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

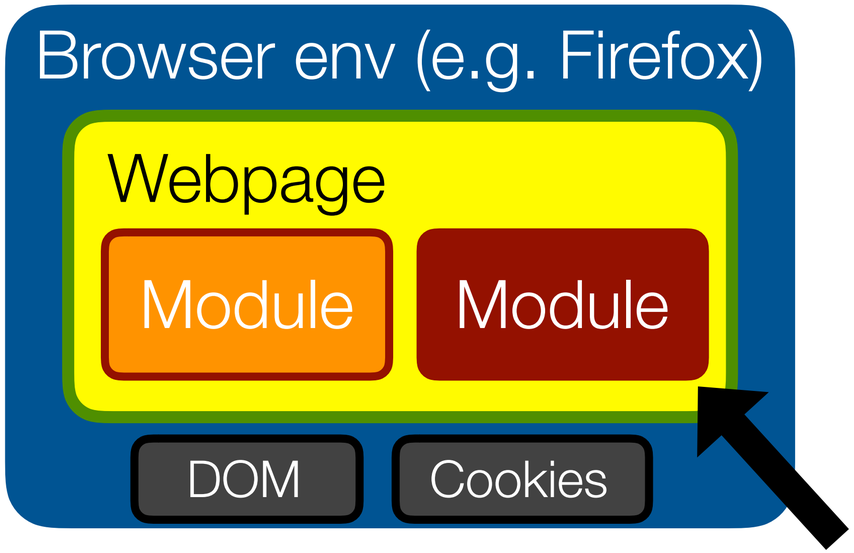



```
<script src=“http://evil.com/ad.js”>
```

```
npm install evil-logger
```

It’s all about **trust**

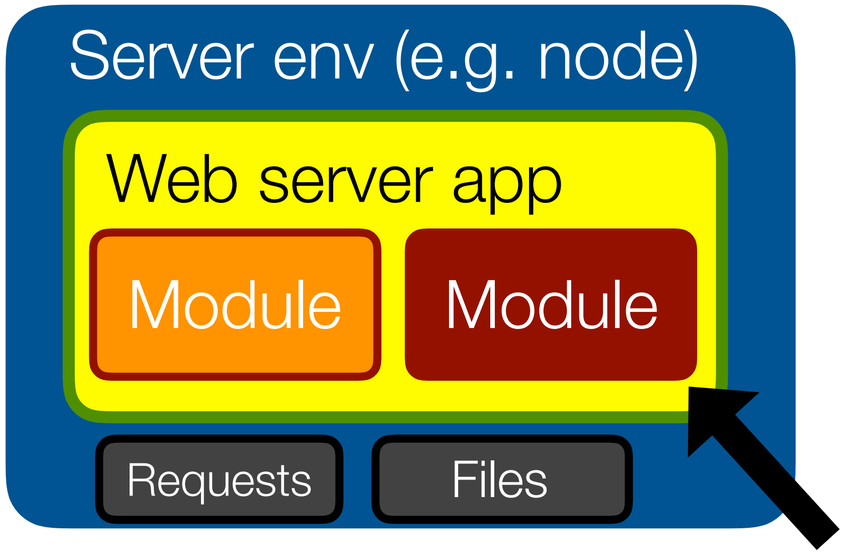
- It is exceedingly common to run code you don’t know/trust in a common environment




```
<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)

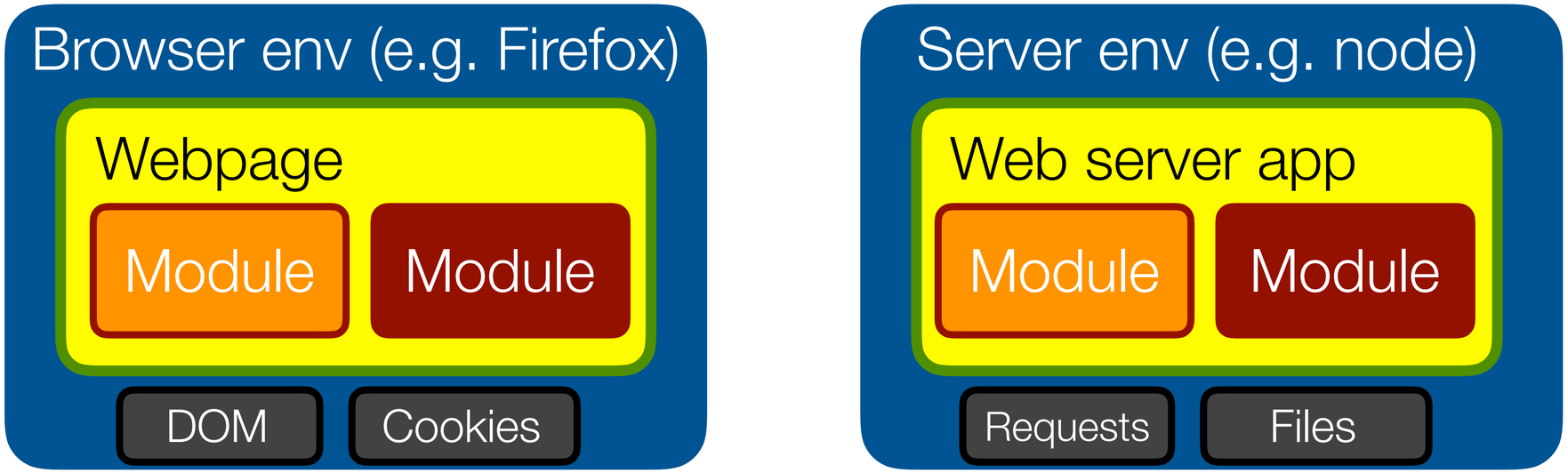
Node.js package tried to plunder Bitcoin wallets

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

(source: theregister.co.uk)

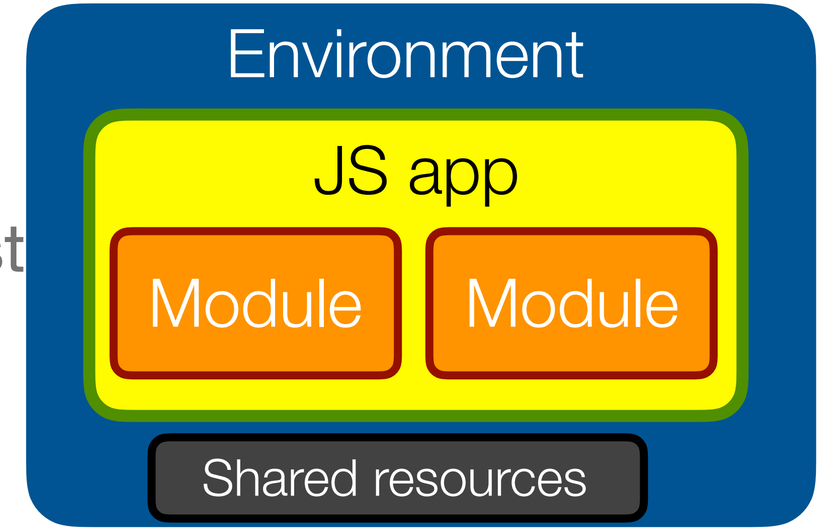
Avoiding interference is the name of the game

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



Prerequisite: isolating JavaScript modules

- Up to today, JavaScript offers no “User mode” way of isolating code into its own environment
- Lots of “System mode” isolation mechanisms exist but non-portable. Examples:
  - **Web Workers**: forced async communication, no shared memory
  - **iframes**: mutable primordials (*), “identity discontinuity”
  - **node** `vm` **module**: easy to break isolation. Use `vm2` module instead <npmjs.com/package/vm2>



(*) primordials = built-in objects like `Object`, `Array`, `Function`, `Math`, `JSON`, etc.

Realms: “User mode” isolation

- Realms are a TC39 Stage 2 proposal
- Intuitions: “iframe without DOM”, “principled version of node’s `vm` module”

```
let g = window; // outer global
let r = new Realm(); // root realm

let f = r.evaluate("(function() { return 17 })");

f() === 17 // true

Reflect.getPrototypeOf(f) === g.Function.prototype // false
Reflect.getPrototypeOf(f) === r.global.Function.prototype // true
```

(source: https://github.com/tc39/proposal-realms/)

- Shim library available at github.com/Agoric/realms-shim

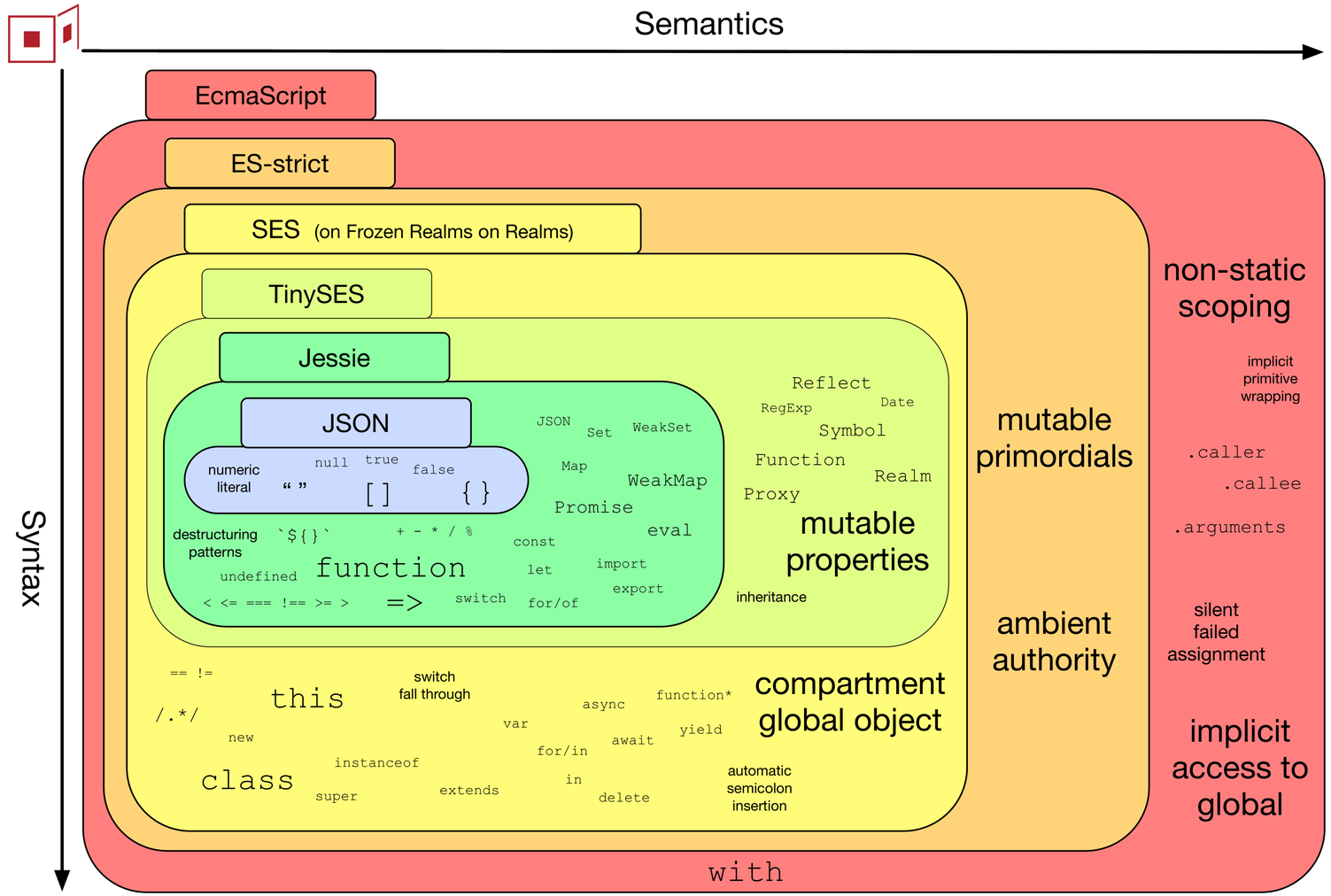
Secure ECMAScript (SES) (aka “Frozen Realms”)

- Another TC39 Proposal (stage 1)
- Adds “frozen realm”: realm whose primordials are all immutable. Immutable primordials can be efficiently shared across child realms.
- Code can be evaluated in a frozen child realm with its own global environment:

```
let val = SES.confine("x + y", {x:1,y:2}); // returns 3
```

- Shim library available at https://github.com/Agoric/SES

Secure ECMAScript is a subset of ES-strict

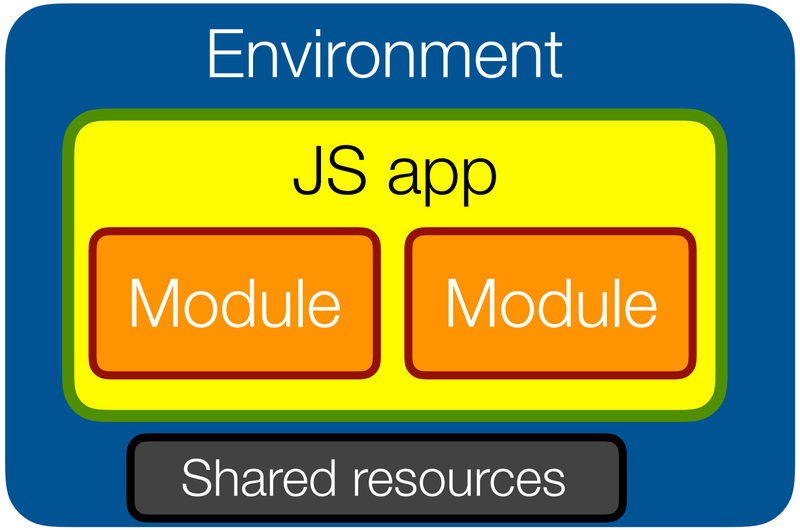


(source: Agoric, https://github.com/Agoric/Jessie)

- All code in strict mode (“sane” JavaScript)
- Immutable primordials
- Own whitelisted global environment
- No “powerful” non-standard globals (e.g. `process`, `window`, …) by default

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

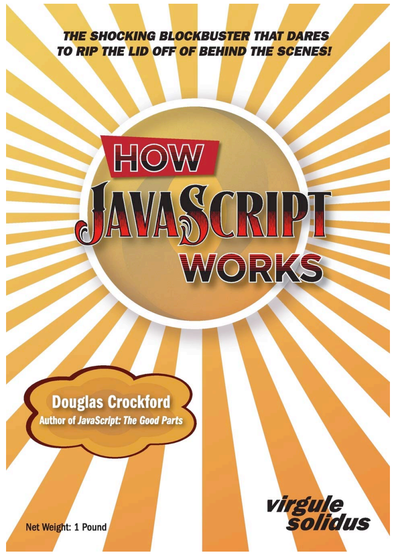
#1: make private state truly private

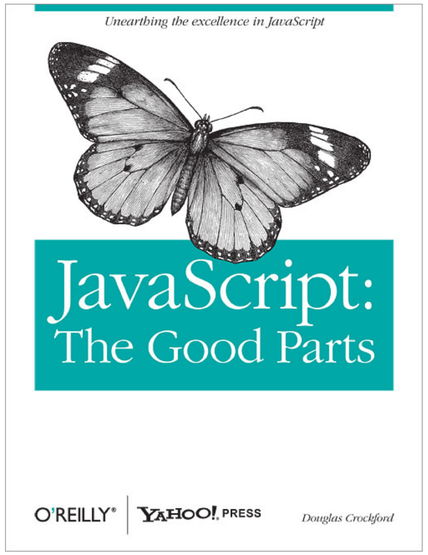
```
class Counter {
  constructor() {
    this.count_ = 0;
  }
  incr() { return ++this.count_; }
  decr() { return --this.count_; }
}

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

What Crockford has to say about this

“a beginning or ending _ is sometimes intended to indicate a public property […] that would have been private if the program had been written correctly. So, a dangling _ is a flag indicating that the coder is incompetent”
(How JavaScript Works, Chapter 1)

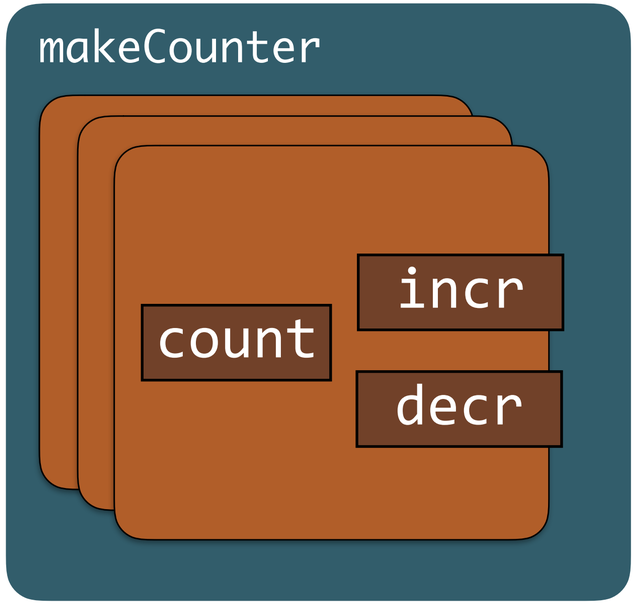
Douglas Crockford, Inventor of JSON





#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

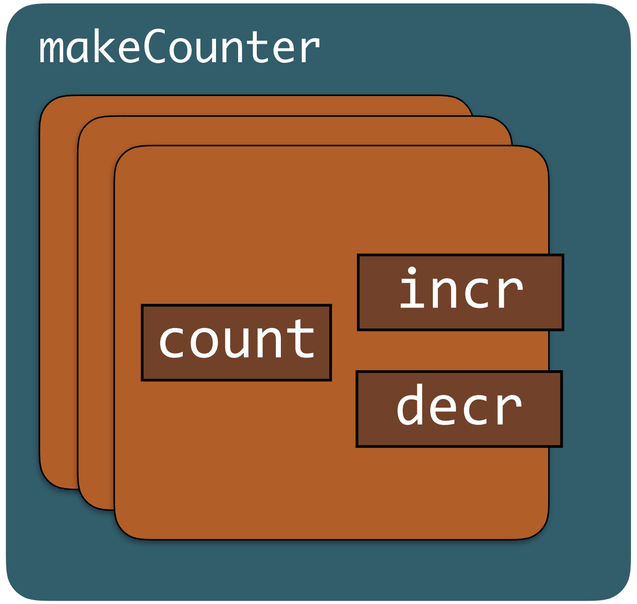



```
function makeCounter() {
  let count = 0;
  return {
    incr() { return ++count; },
    decr() { return --count; }
  }
}

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)
- Note: freezing an object does not transitively freeze any objects/functions reachable from the object. Full immutability requires a ‘deep-freeze’

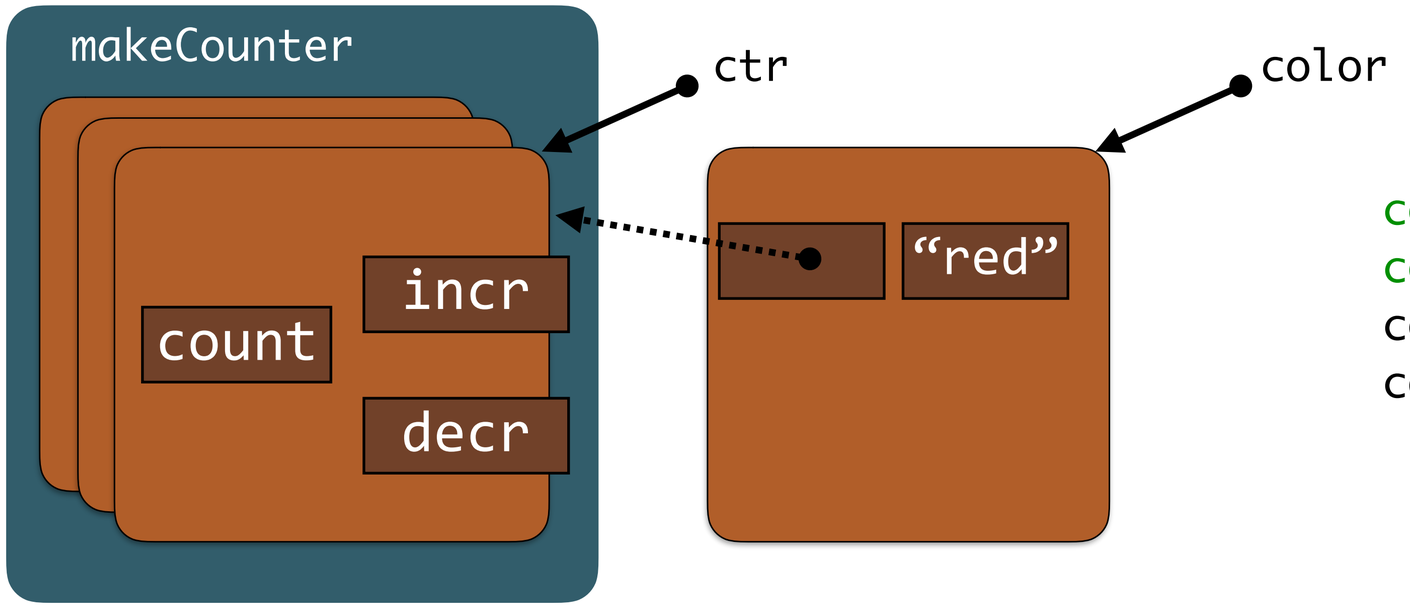



```
function makeCounter() {
  let count = 0;
  return Object.freeze({
    incr() { return ++count; },
    decr() { return --count; }
  })
}

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

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

#3: safely extend objects with new properties using WeakMaps

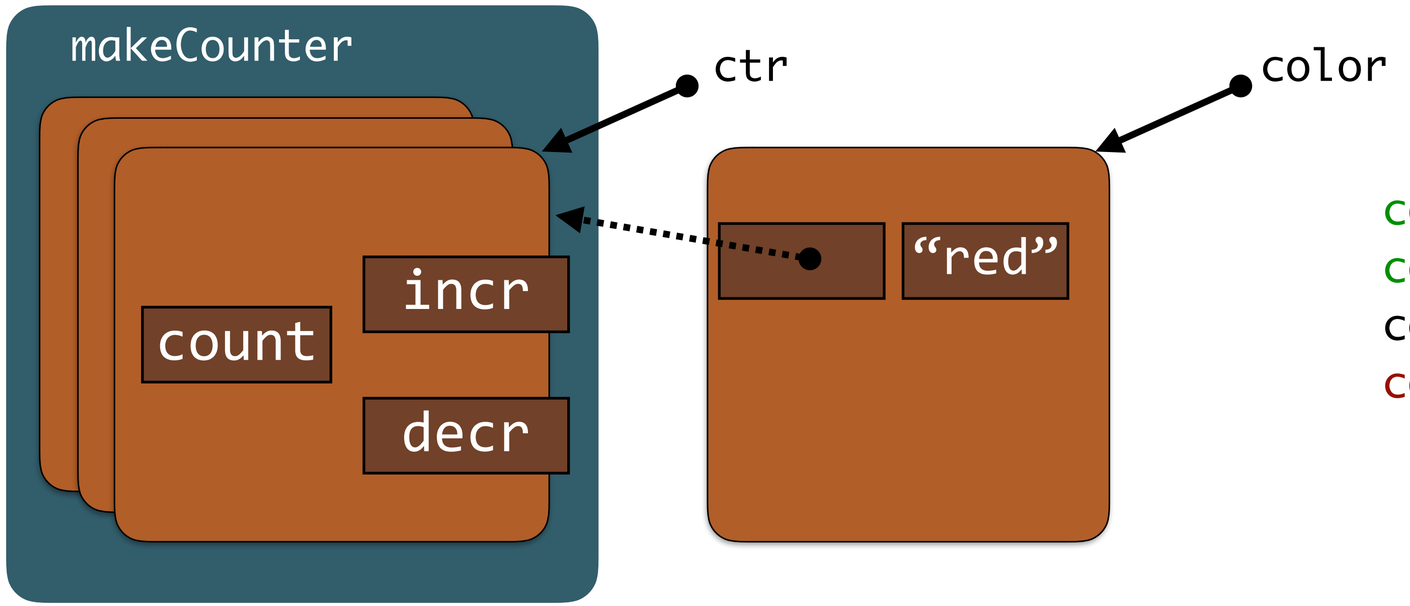
- It is common for one module to want to “expand” the objects of another module with new properties. Common practice today: **monkey-patching**
- WeakMaps can store new properties **without mutating** the original objects
  - Also works for frozen objects




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

#3: safely extend objects with new properties using WeakMaps

- It is common for one module to want to “expand” the objects of another module with new properties. Common practice today: **monkey-patching**
- WeakMaps can store new properties **without mutating** the original objects
- **Bonus**: only code that has access to **both** the WeakMap and the original object can access the value




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

#4: use WeakSets to do reliable “instance of” tests (“brands checks”)

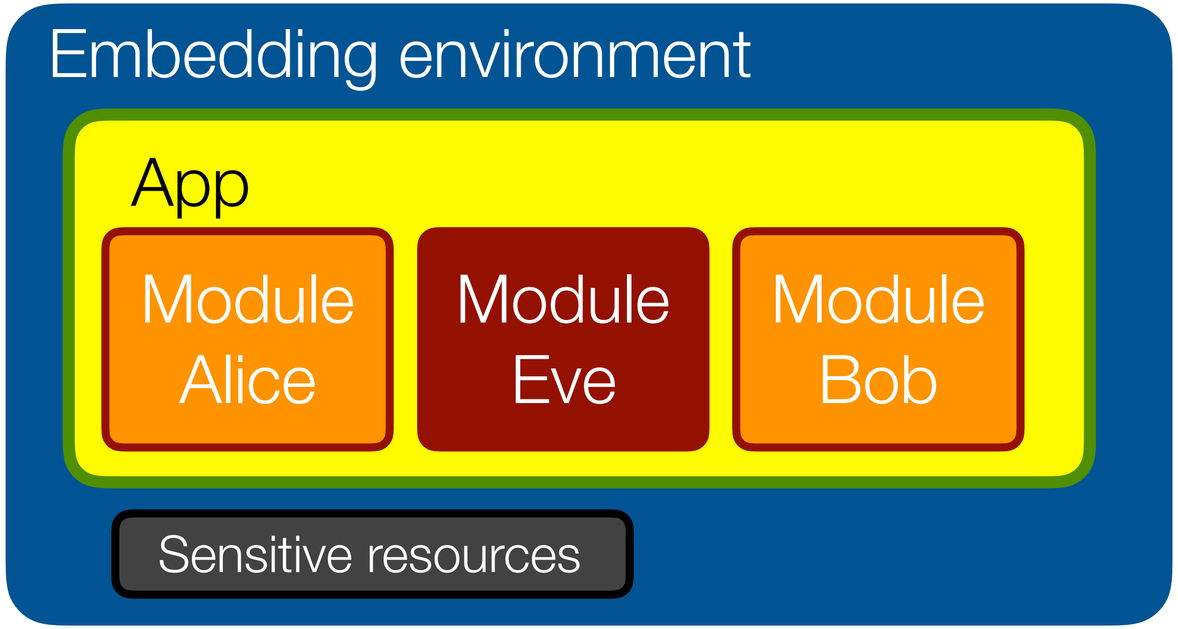
- It is common for functions to want to verify whether the arguments they receive are “genuine” objects of a certain type
- Common practice today: **duck-testing**

```
class Duck {
  constructor() {
    this.__isADuck__ = true;
  }
  quack() { … }
}

function f(arg) {
  if (arg.__isADuck__) {
    arg.quack();
  }
}
```

```
const isADuck = new WeakSet();
function makeDuck() {
  const duck = Object.freeze({
    quack() { … }
  });
  isADuck.add(duck);
  return duck;
}

function f(arg) {
  if (isADuck.has(arg)) {
    arg.quack();
  }
}
```

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

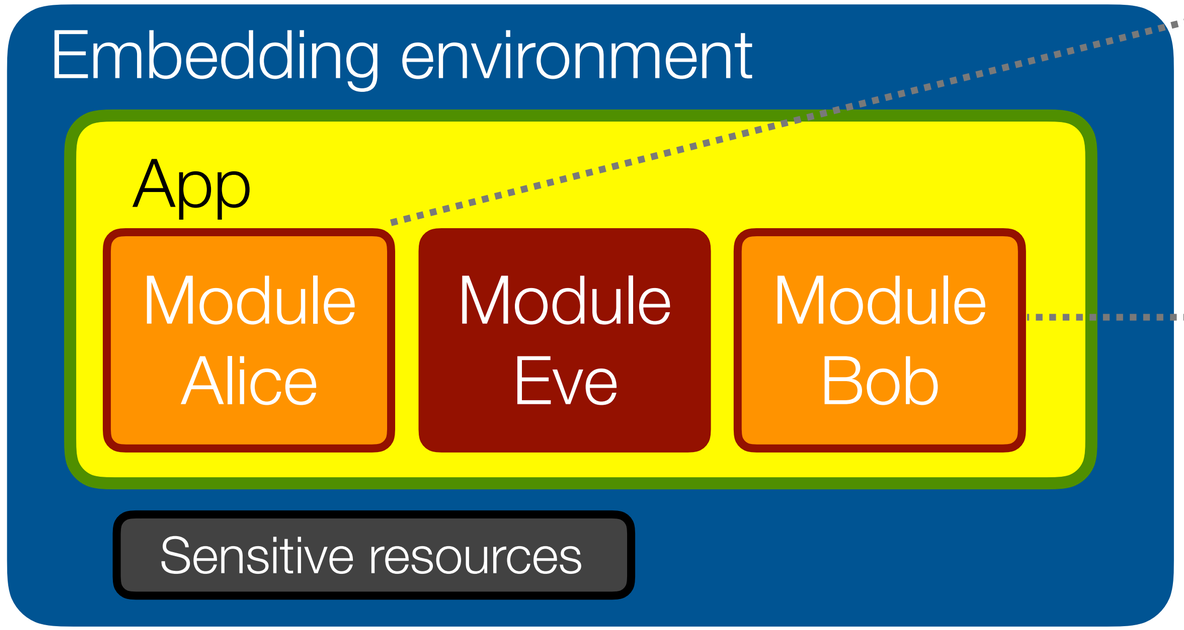
- Consider the following (common) setup:



- 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?

#5: 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
  })
}
```

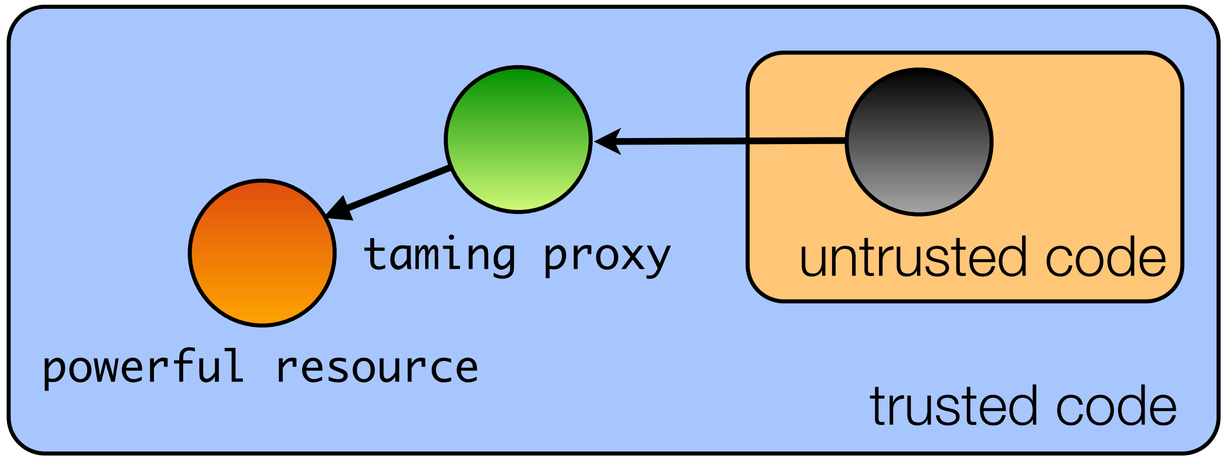
#5: use sealer/unsealer pairs to “encrypt” objects with no crypto

```
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 [seal, unseal];
}
```

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

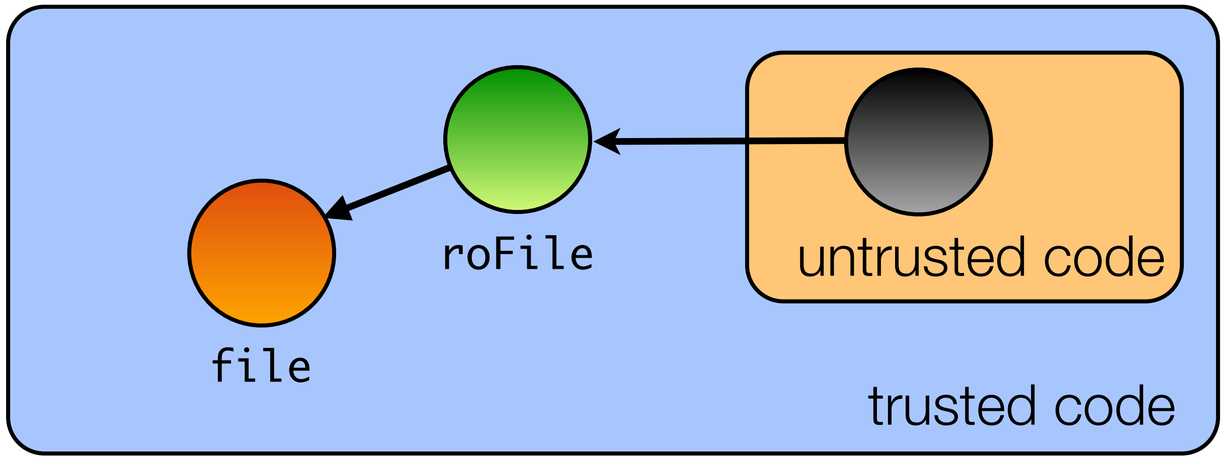
#6: use the Proxy pattern to attenuate APIs (taming)

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



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

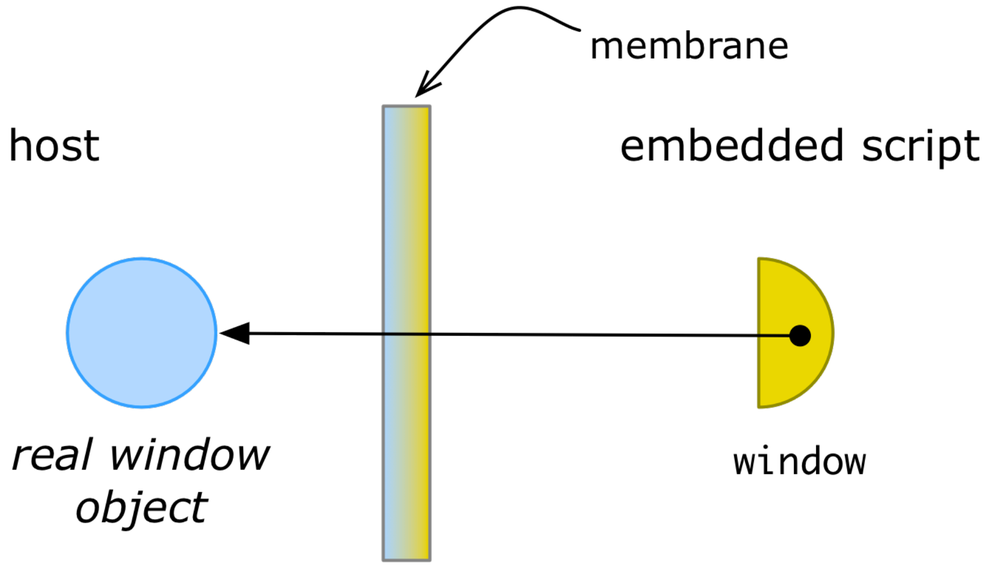
- Example: attenuating read-write access to read-only access:




```
function makeReadOnly(file) {
  return Object.freeze({
    read() { return file.read(); }
    getLength() { return file.getLength(); }
  });
}

// Alice says:
const roFile = makeReadOnly(file);
eve.give(roFile);
```

#7: generalizing the Proxy pattern to isolate object graphs

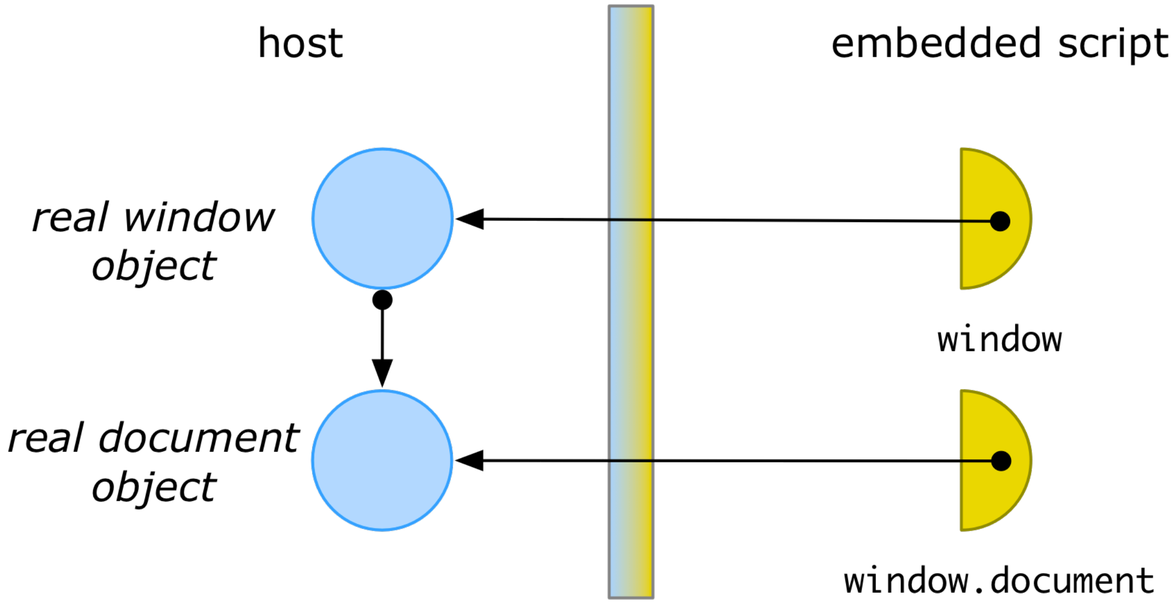
- A membrane injects a layer of proxy objects between two or more object graphs, which can be used to intercept all communication
- Membrane grows/shrinks as needed based on dynamic interaction patterns



Full article at tvcutsem.github.io/membranes

#7: generalizing the Proxy pattern to isolate object graphs

- A membrane injects a layer of proxy objects between two or more object graphs, which can be used to intercept all communication
- Membrane grows/shrinks as needed based on dynamic interaction patterns

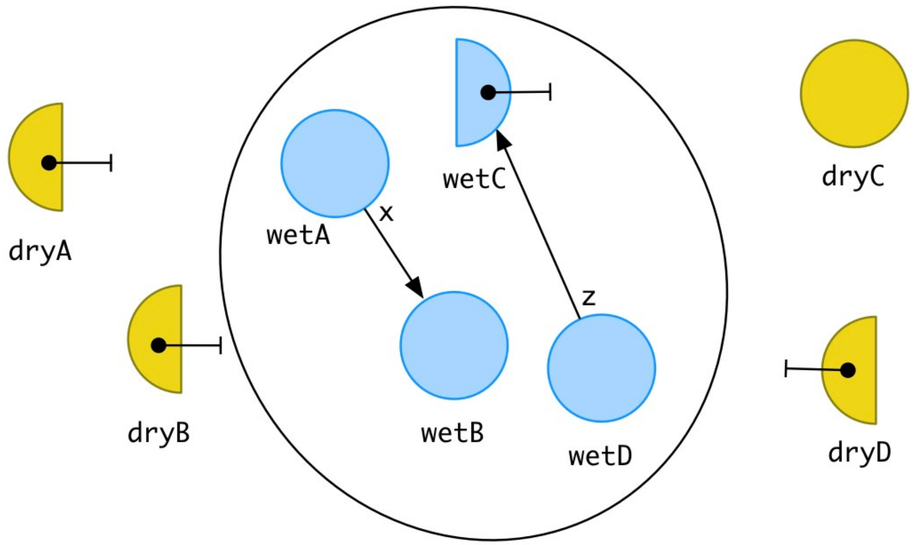


Full article at tvcutsem.github.io/membranes

#7: generalizing the Proxy pattern to isolate object graphs

- Membranes can be built from Proxy objects and WeakMaps
- The proxies of a membrane can share state

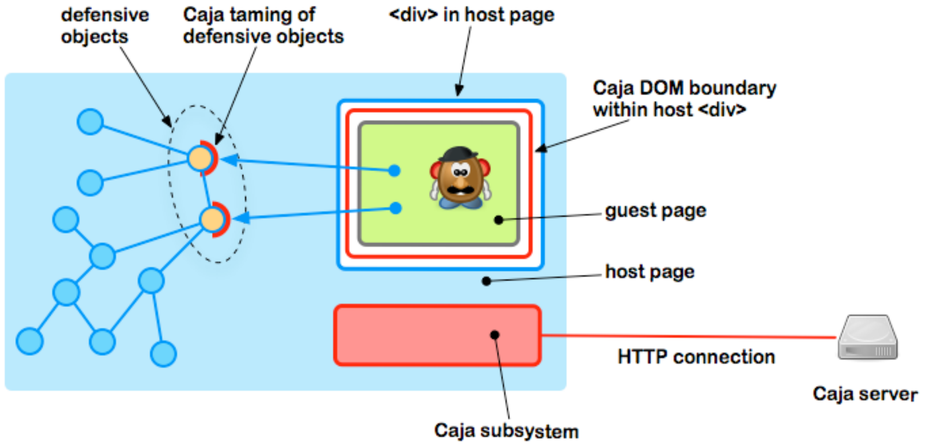
```
function makeMembrane(initDryTarget) {
  let enabled = true;
  let wetProxies = new WeakMap();
  let dryProxies = new WeakMap();
  ...
  function wet2dry(wetTarget) { … }
  function dry2wet(dryTarget) { … }
  ...
  return {
    proxy: dry2wet(initDryTarget),
    revoke: function() { enabled = false; }
  };
}
```



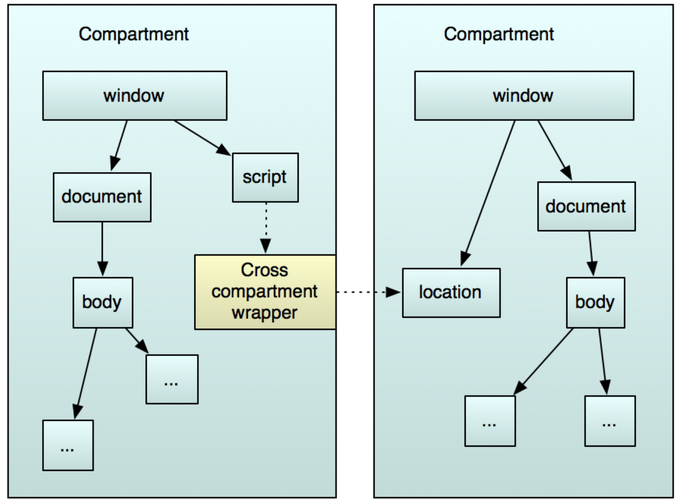
Full article at tvcutsem.github.io/js-membranes

These patterns are used in industry

- Embedding third-party content on web properties: **Google** Caja uses taming.
- Application components / plug-ins:
  - **Mozilla** uses membranes in Firefox to implement security boundaries between different site origins and privileged JS code
  - **Salesforce** uses Secure ECMAScript and membranes in its Lightning UI platform for mobile and desktop
- Smart contracts: **Cosmos** blockchain project builds on Secure ECMAScript



(source: Google, developers.google.com/caja)

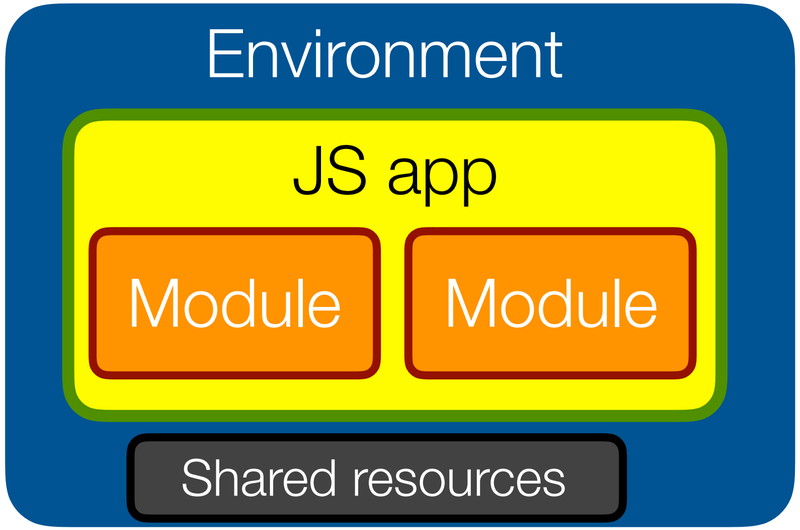


(source: Mozilla, developer.mozilla.org)

Conclusion

Summary

- View security as extreme 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**



@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

- 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>
- SES: https://github.com/tc39/proposal-ses and https://github.com/Agoric/SES (past incarnation at https://github.com/google/caja/wiki/SES)
- Realms: https://github.com/tc39/proposal-realms (original at https://github.com/FUDCo/ses-realm)
- Subsetting ECMAScript: https://github.com/Agoric/Jessie