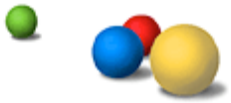

# Remaining Hazards and Mitigating Patterns for Secure Mashups in EcmaScript 5

Mark S. Miller and the Cajadores



# Overview

- The Mashup Problem
  - The Offensive and Defensive Code Problems
- JavaScript (EcmaScript) gets simpler
  - ES3, ES5, ES5/strict, SES-on-ES5
- Secure EcmaScript (SES) defenses
  - Confinement and Tamper Proofing
- Remaining SES Security Hazards
  - Riddles: Attack these example
- Mitigating Patterns for Attack Resistant Code

# New Skills open up New Worlds

## Remember learning

"Avoid goto"

"Beware pointer arithmetic"

"Beware threads and locks"

"Zero index origin likes closed-open intervals"

"Manual encoding is better than string append"

"Auto-escaping is better than manual encoding"

and various oo patterns and their hazards?

## Co-evolution of skills and tools

Student drivers think hard to avoid accidents.

Experts avoid traps, but think about destination.

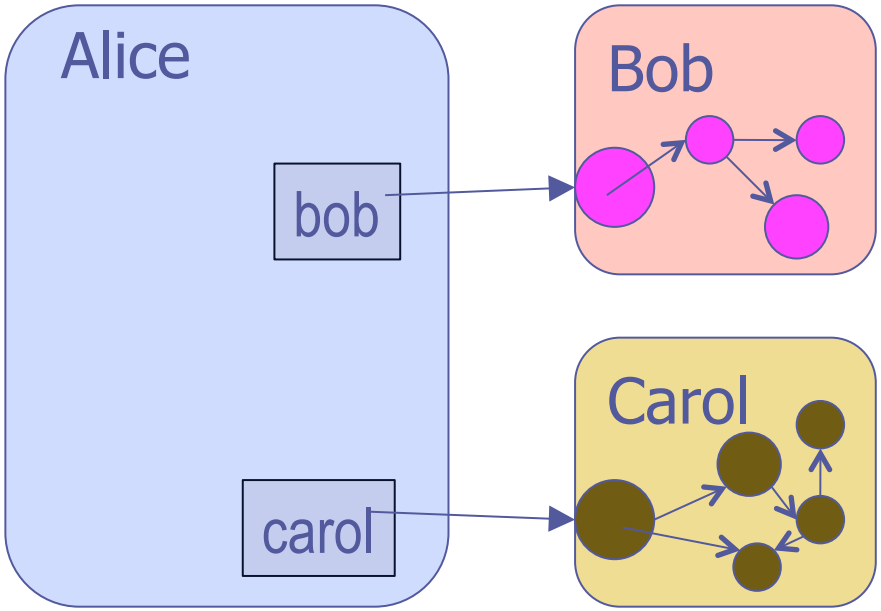
Cars learn to help.

# Mashups are Everywhere

```
…
<script src="https://evil.com/matrix.js">
<script>
  var prod = matMult(matrixA, matrixB);
</script>
```

Why can matMult hijack my account?

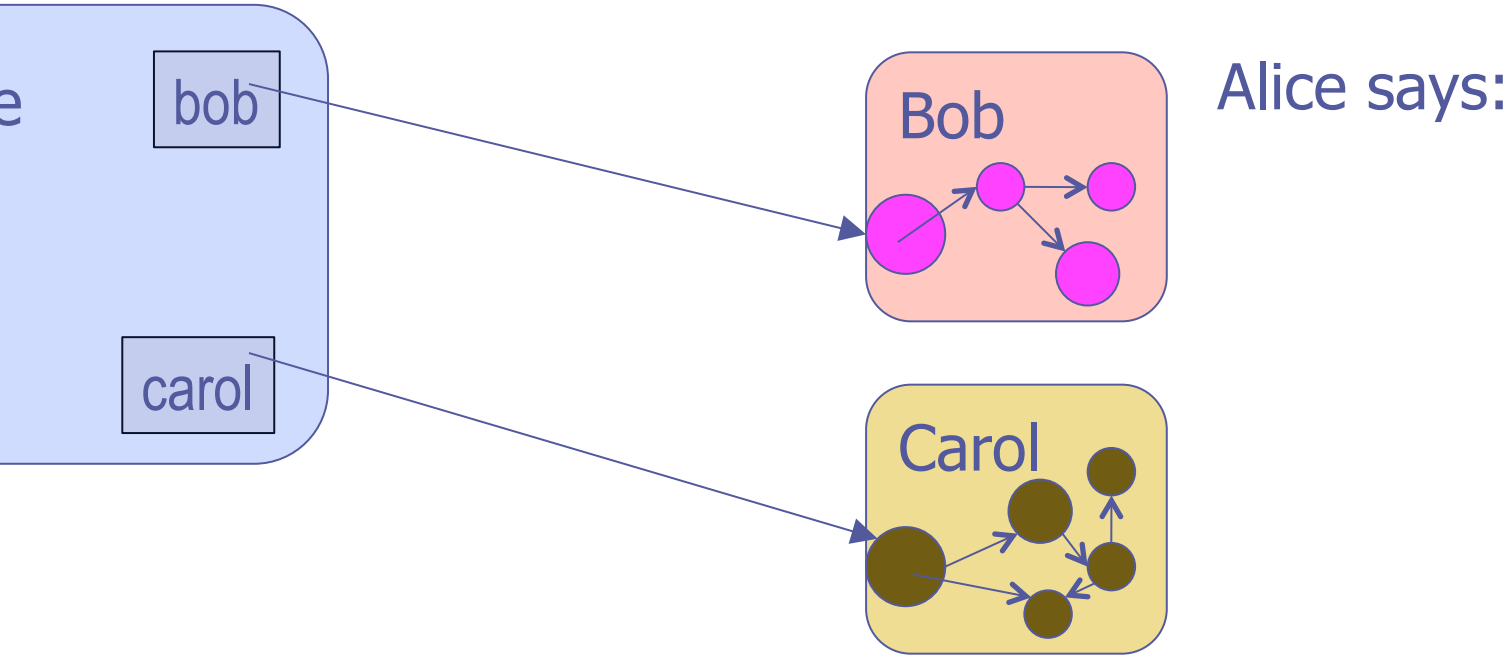
# A Trivial Mashup Scenario




Alice says:

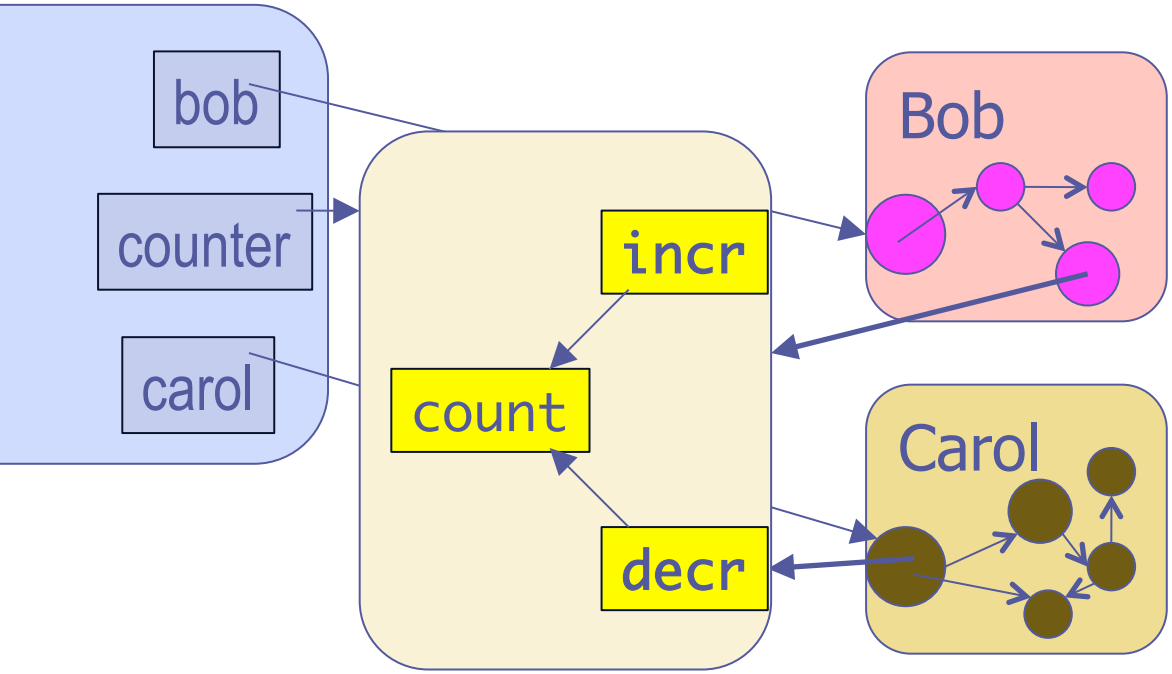
```
var bobSrc = //site B
var carolSrc = //site C
var bob = eval(bobSrc);
var carol = eval(carolSrc);
```

# A Trivial Mashup Scenario




Alice says:

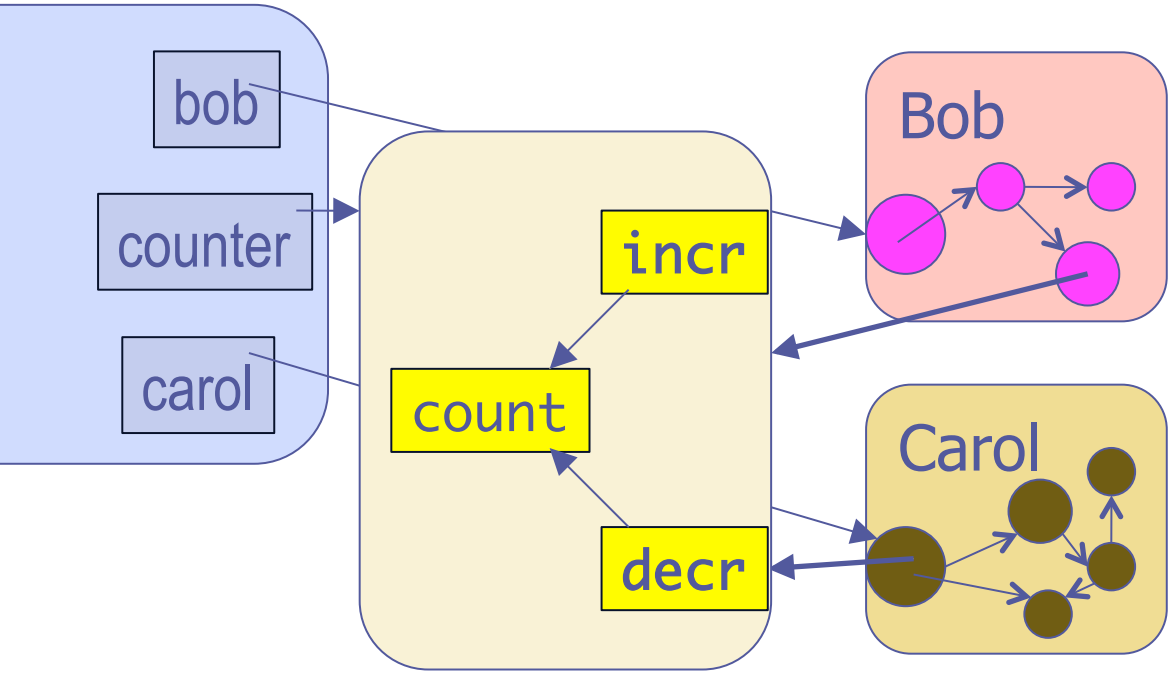
# A Trivial Mashup Scenario




Alice says:

```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

# A Trivial Mashup Scenario

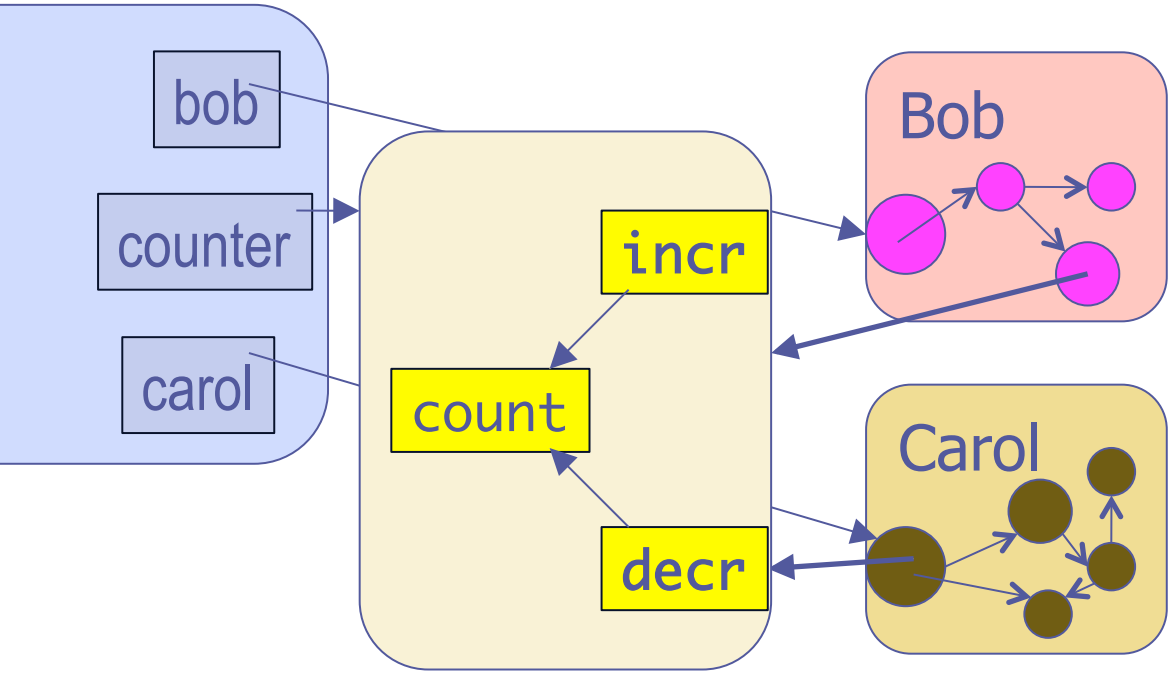



Alice says:

```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

Bob can count up and down and see result.
Carol can count down and see the result.

# A Trivial Mashup Scenario




Alice says:

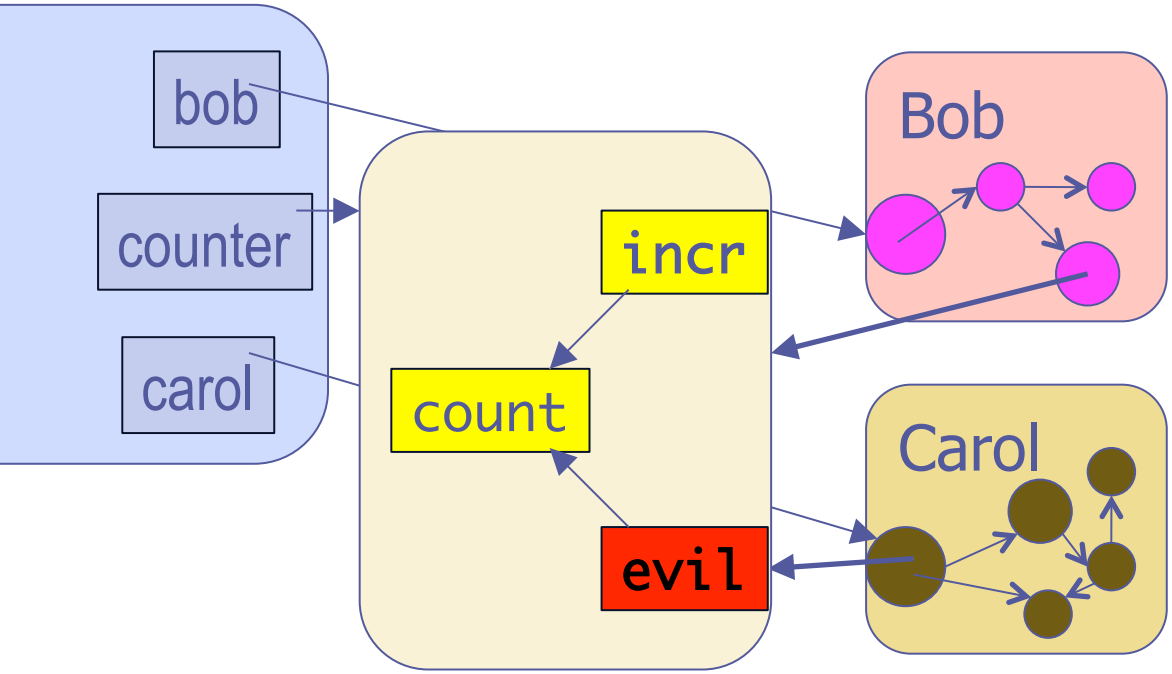
```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

Principle of least authority:

Bob can ***only*** count up and down and see result.

Carol can ***only*** count down and see the result.

# A Trivial Mashup Attack Scenario




Alice says:

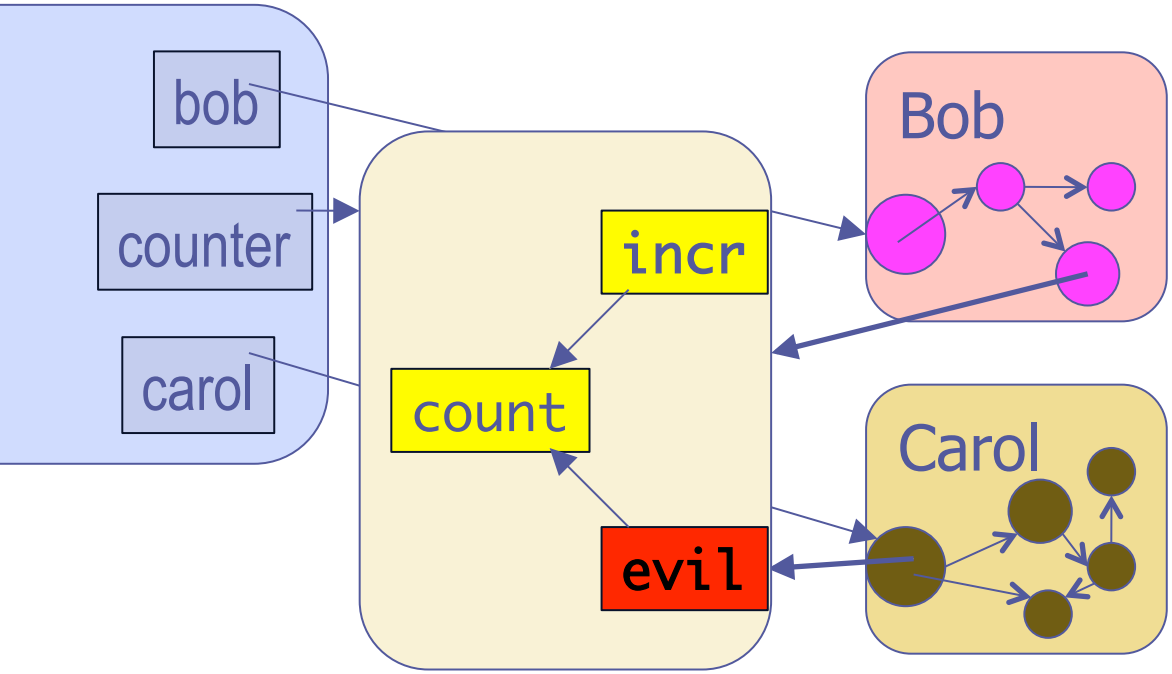
```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

Bob says:

```
counter.decr = evil;
```

When Alice or Carol try to count down,
they call Bob's evil function instead.

# A Trivial Mashup Attack Scenario




Alice says:

```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

Bob says:

```
counter.decr = evil;
…window…document.cookie…
```

Bob can do much worse damage!

# The Mashup Problem

"A Mashup is a Self Inflicted Cross-Site Script"

—Douglas Crockford

The Offensive Code Problem

- Solved by SES

The Defensive Code Problem

- Mitigated by patterns made possible by SES
- Still Hard! A puzzle solving skill to learn.

# The Offensive Code Problem

## Abuse of Global Authority

Phishing, Redirection, Cookies

## Prototype Poisoning

```
Object.prototype.toString = evilFunc;
```

## Global Scope Poisoning

```
JSON = {parse: eval};
```

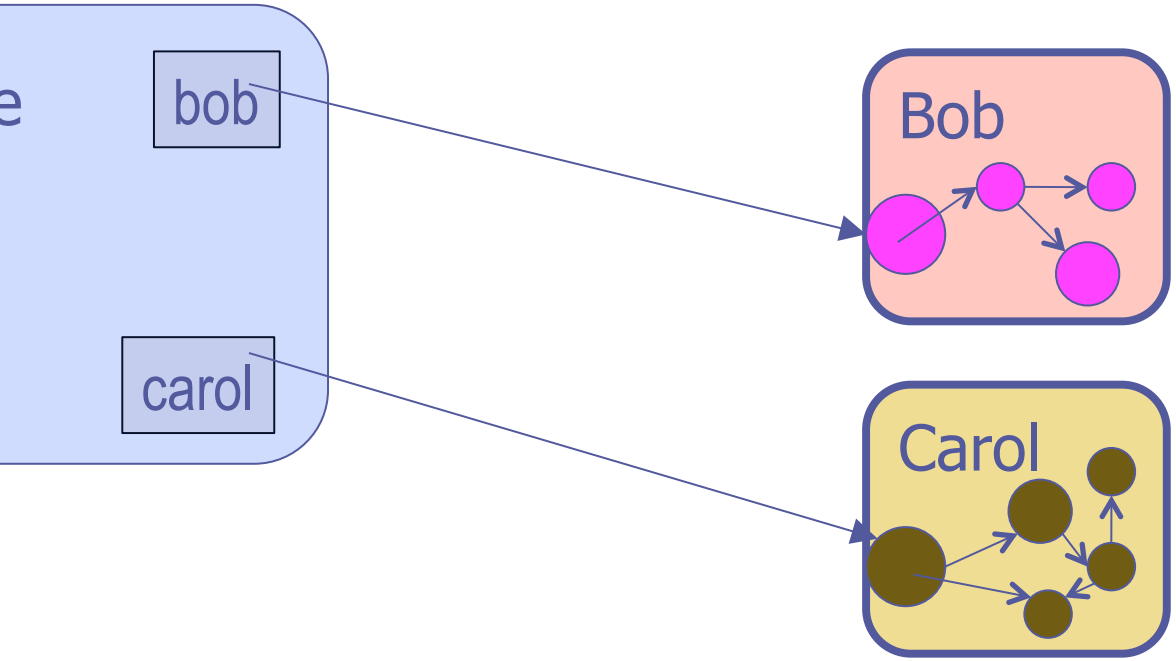
# Turning EcmaScript 5 into SES

```
<script src="initSES.js"></script>
```

Monkey patch away bad non-std behaviors
Remove non-whitelisted primordials
Install leaky WeakMap emulation
Make virtual global root
Freeze whitelisted global variables

- Replace **eval** & Function with safe alternatives
  Freeze accessible primordials

# SES eval ➔ Confinement

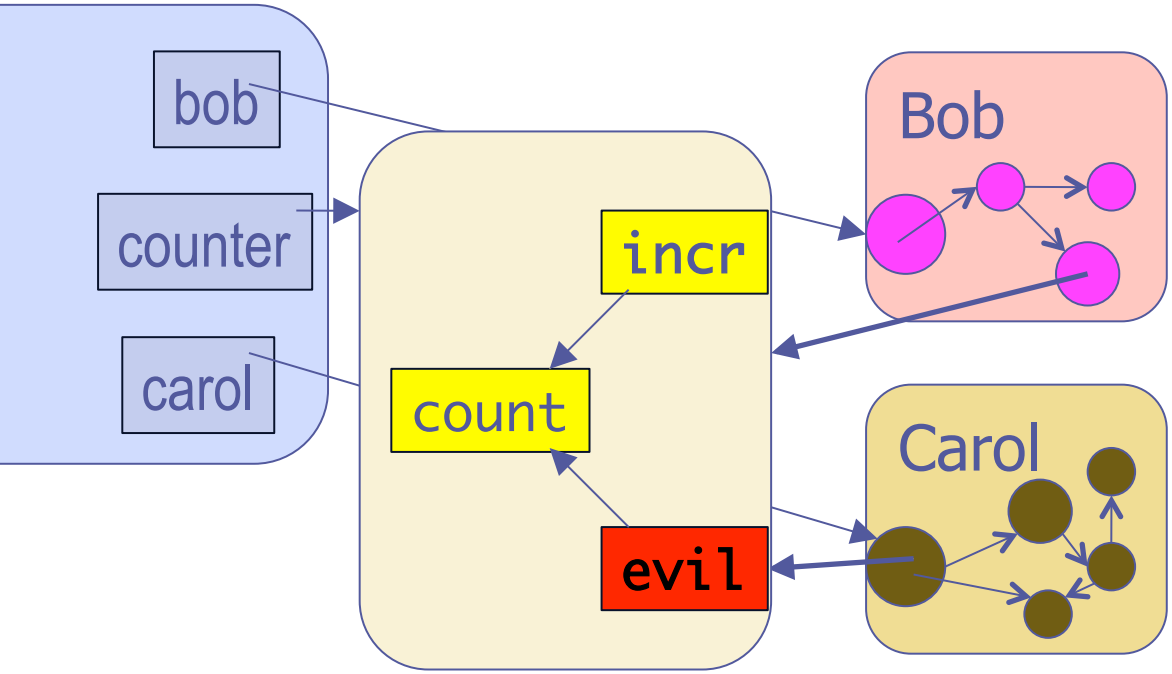



Alice says:

```
var bobSrc = //site B
var carolSrc = //site C
var bob = eval(bobSrc);
var carol = eval(carolSrc);
```

Bob cannot yet cause any effects outside himself!

# Need Bullet-proof Defensive Objects




Alice says:

```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

Bob says:

```
counter.decr = evil;
```

~~…window…document.cookie…~~

Bob can still subvert a non-defensive counter

# The Defensive Code Problem

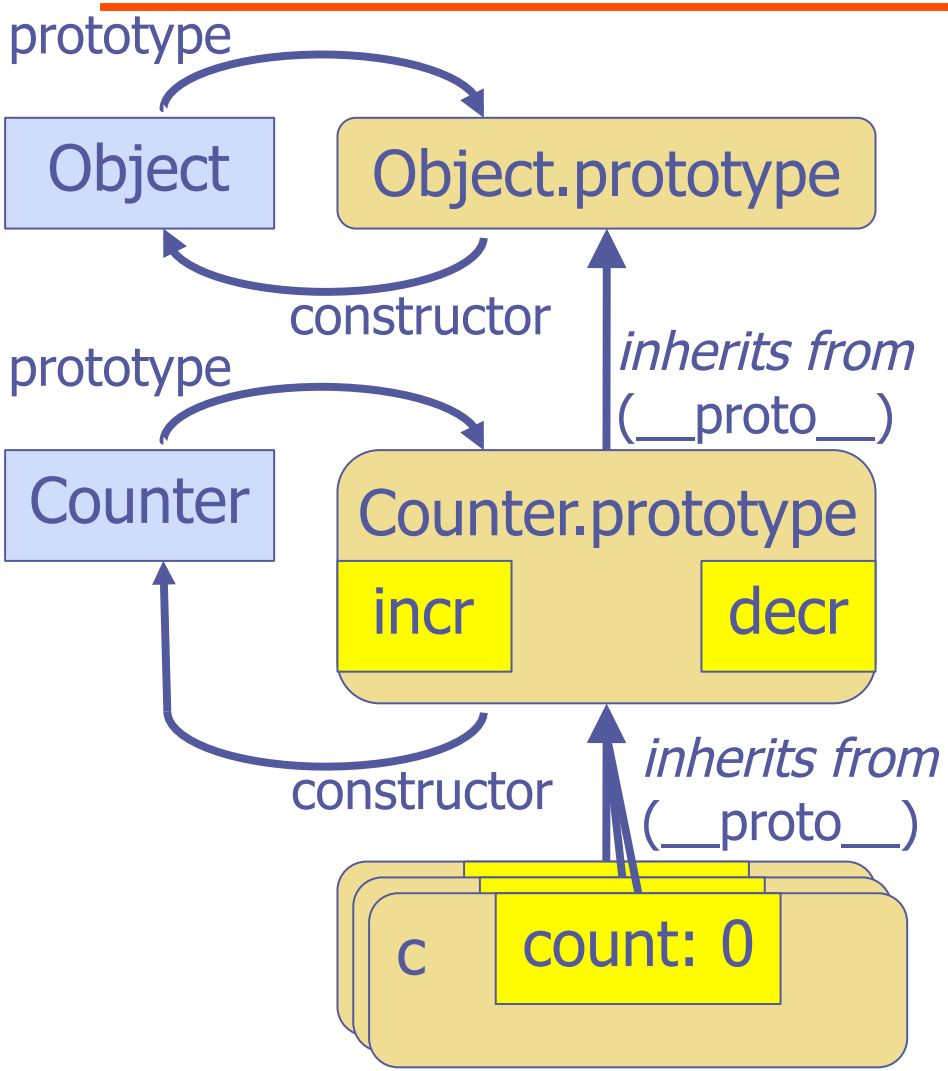
Violating Encapsulation

Tampering with API surface

Violating Assumptions ➔ Loss of Integrity

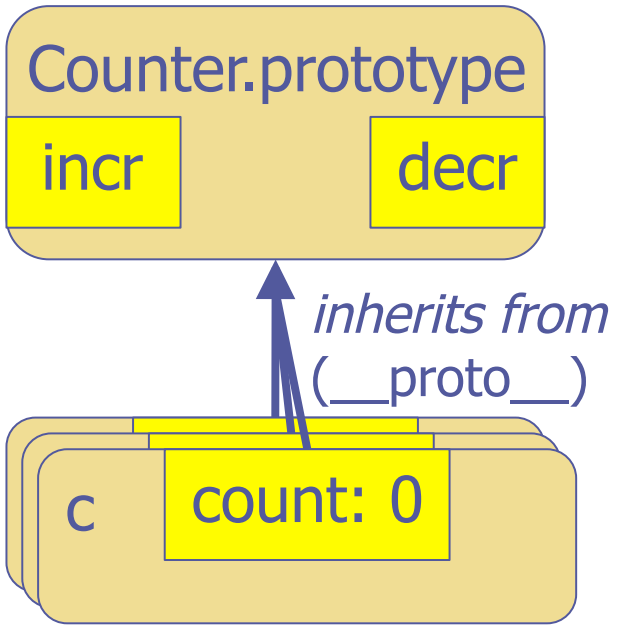
Contagious Corruption

# Classic JS Prototypal Objects



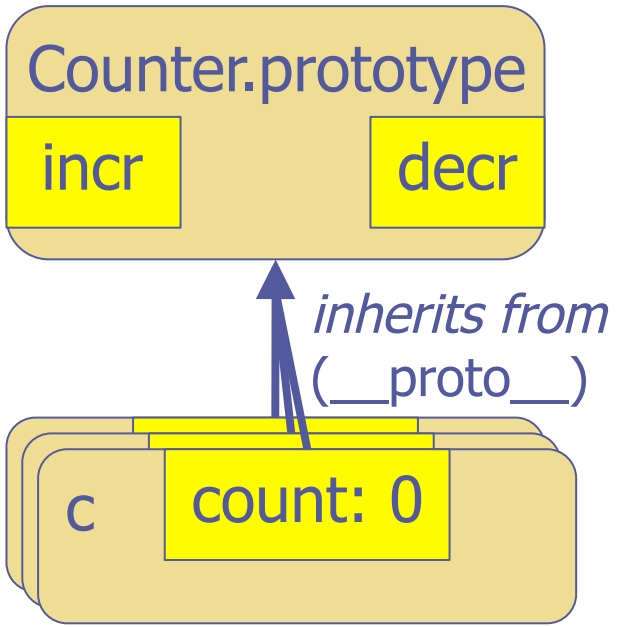

```
function Counter() { this.count = 0; }

Counter.prototype = {
    incr:  function() { return ++this.count; },
    decr: function() { return --this.count; },
    constructor: Counter
};
var c = new Counter();
```

# Classic JS Prototypal Objects




```
function Counter() { this.count = 0; }

Counter.prototype = {
    incr:  function() { return ++this.count; },
    decr: function() { return --this.count; },
    constructor: Counter
};
var c = new Counter();
```

# Classic JS Prototypal Objects

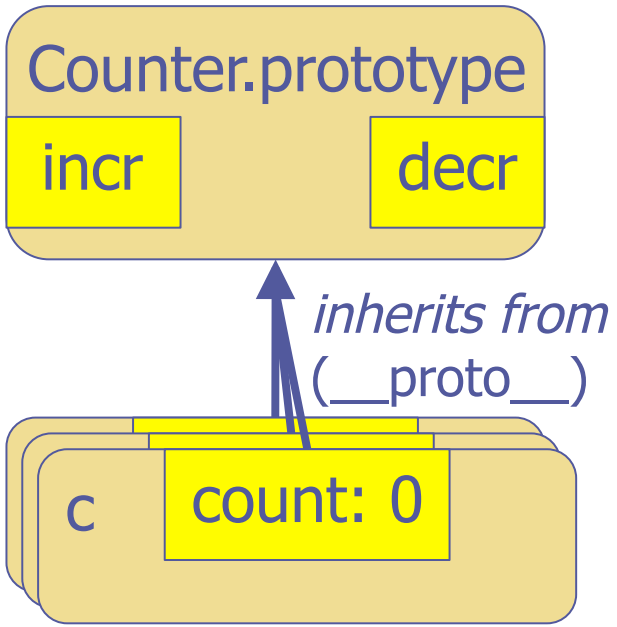


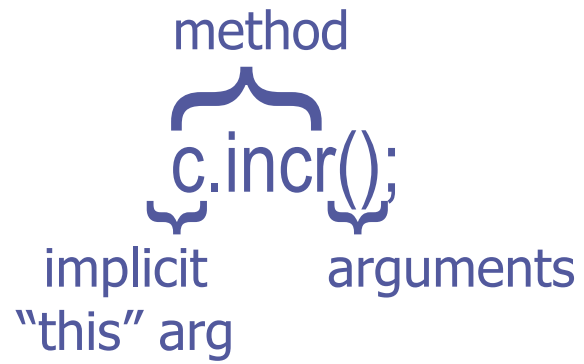

```
function Counter() { this.count = 0; }

Counter.prototype = {
    incr:  function() { return ++this.count; },
    decr: function() { return --this.count; },
    constructor: Counter
};
var c = new Counter();
```

```
c.incr();
```

# Classic JS Prototypal Objects

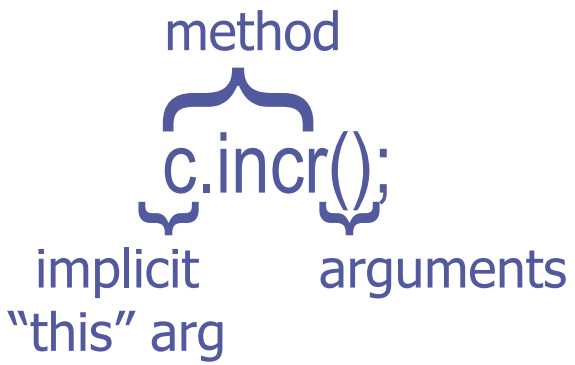
```
function Counter() { this.count = 0; }

Counter.prototype = {
    incr:  function() { return ++this.count; },
    decr: function() { return --this.count; },
    constructor: Counter
};
var c = new Counter();
```



# Function Call, Method Call, Reflection

| | | |
|---|---|---|
| c.incr() | Method call | "this" is c |
| c['incr']() | Method call | "this" is c |
| (c.incr)() | Method call | "this" is c |
| (c['incr'])() | Method call | "this" is c |
| var *incr* = c.incr;<br>incr() | Function call | "this" is undefined |
| (1,c.incr)() | Function call | "this" is undefined |
| d.incr = c.incr;<br>d.incr() | Method call | "this" is d |
| c.incr.apply(d, []) | Reflective call | "this" is d |
| applyFn(c.incr, d, []) | Reflective call | "this" is d |

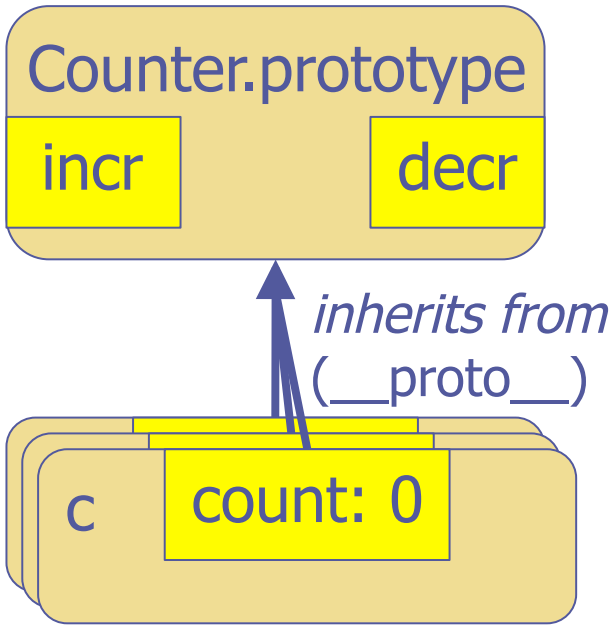


# Reflection Helper

```
var applyFn = Function.prototype.call.bind(Function.prototype.apply);
```

obj.name(…args) ➔ applyFn(obj.name, obj, args)
func(…args) ➔ applyFn(func, undefined, args)

# Classic JS Prototypal Objects



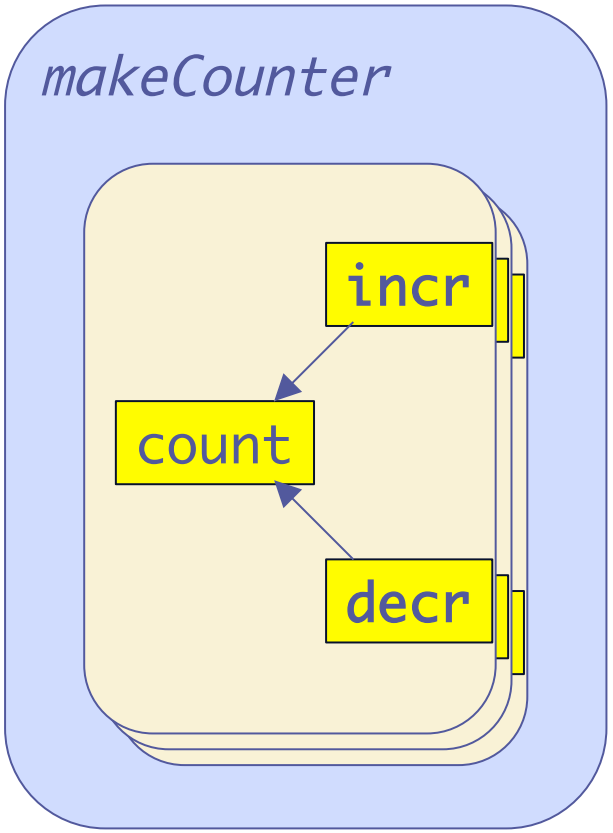

```
function Counter() { this.count = 0; }

Counter.prototype = {
    incr:  function() { return ++this.count; },
    decr: function() { return --this.count; },
    constructor: Counter
};
var c = new Counter();

// Confusion attacks:
applyFn(c.incr, nonCounter, []);

// Corruption attacks:
c.count = -Infinity;
```

# First Lesson

Classic JS Prototype Pattern is hazardous

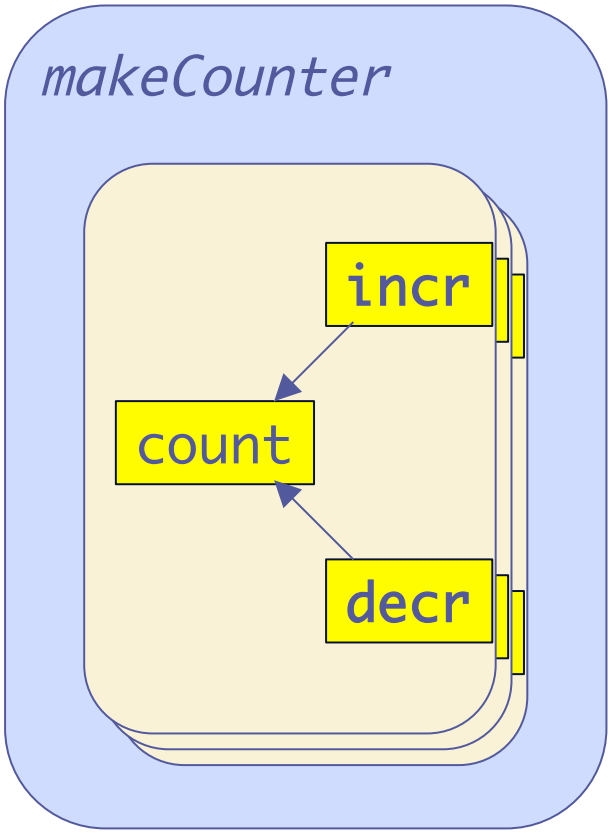
“this” is hazardous

# Objects as Closures in JavaScript




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

# Objects as Closures in JavaScript




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

A *record* of *closures* hiding *state*
is a fine representation of an
*object* of *methods* hiding *instance vars*

# Robustness impossible in ES3

Mandatory mutability (monkey patching)

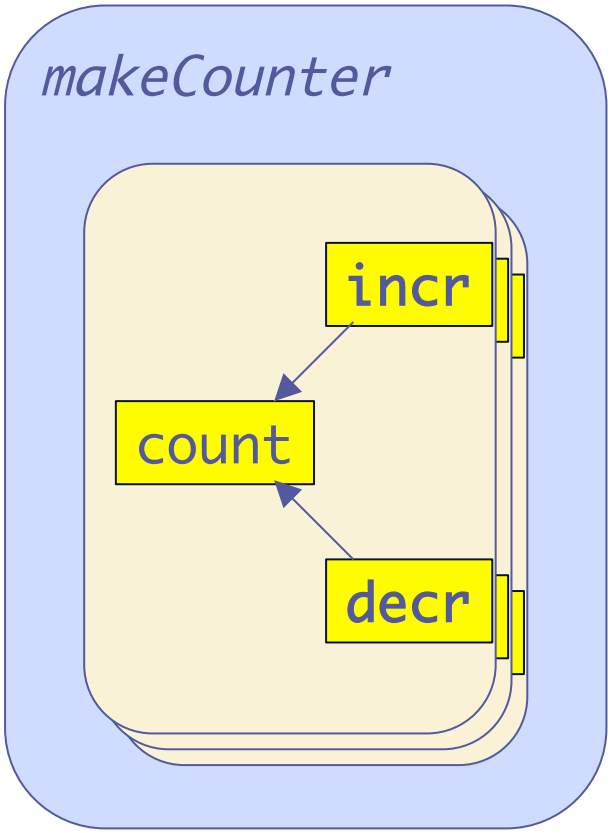
Not statically scoped – repaired by ES5

```
(function n() {…x…})                  // named function exprs
try{throw fn;}catch(f){f();…x…}       // thrown function
Object = Date; …{}…                   // "as if by"
```

Not statically scoped – repaired by ES5/strict

```
with (o) {…x…}                        // attractive but botched
delete x;                             // dynamic deletion
eval(str); …x…                        // eval exports binding
```

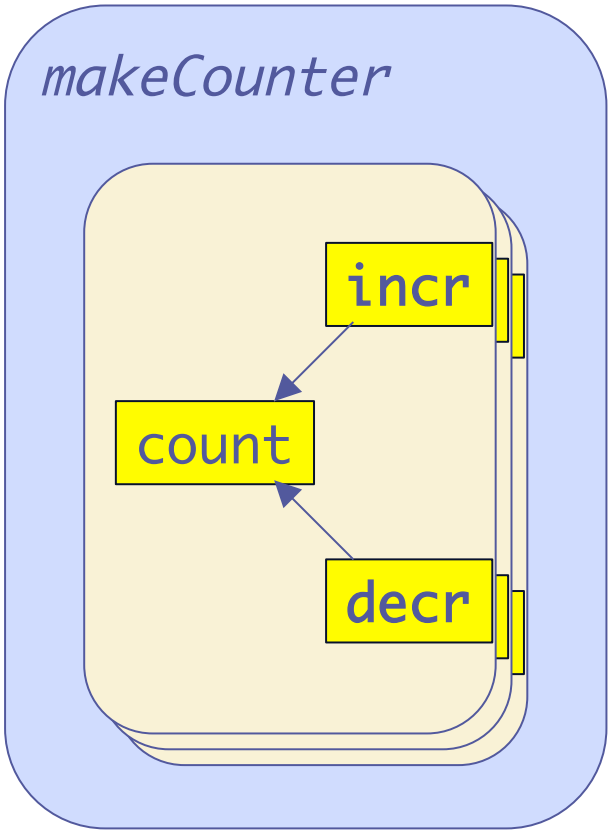
# Objects as Closures in EcmaScript 3




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

- Mandatory mutability
- Scoping confusions
- Encapsulation leaks

# Using ES5 to Stop Bob's 1st Attack




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

- *Unexpressed* mutability
- Scoping confusions
- Encapsulation leaks

# Encapsulation Leaks in non-strict ES5

```
function doSomething(ifBobKnows, passwd) {
    if (ifBobKnows() === passwd) {
        //… do something with passwd
    }
}
```

# Encapsulation Leaks in non-strict ES5

```
function doSomething(ifBobKnows, passwd) {
    if (ifBobKnows() === passwd) {
        //… do something with passwd
    }
}
```

Bob says:

```
var stash;
function ifBobKnows() {
    stash = arguments.caller.arguments[1];
    return arguments.caller.arguments[1] = badPasswd;
}
```

# Encapsulation in ES5/strict

```
"use strict";
function doSomething(ifBobKnows, passwd) {
    if (ifBobKnows() === passwd) {
        //… do something with passwd
    }
}
```

Bob's attack fails:

```
return arguments.caller.arguments[1] = badPasswd;
```

Parameters not joined to arguments.

# Encapsulation in ES5/strict

```
"use strict";
function doSomething(ifBobKnows, passwd) {
    if (ifBobKnows() === passwd) {
        //… do something with passwd
    }
}
```

Bob's attack fails:

```
return arguments.caller.arguments[1] = badPasswd;
```

Poison pills.

# Encapsulation in ES5/strict

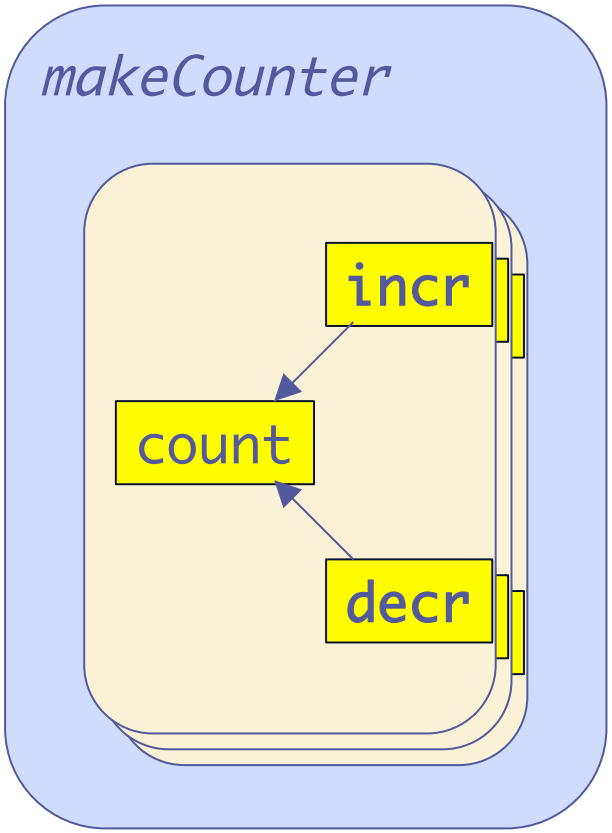
```
"use strict";
function doSomething(ifBobKnows, passwd) {
    if (ifBobKnows() === passwd) {
        //… do something with passwd
    }
}
```

Bob's attack fails:

```
return arguments.caller.arguments[1] = badPasswd;
```

Even non-strict "**.caller**" can't reveal a strict caller.

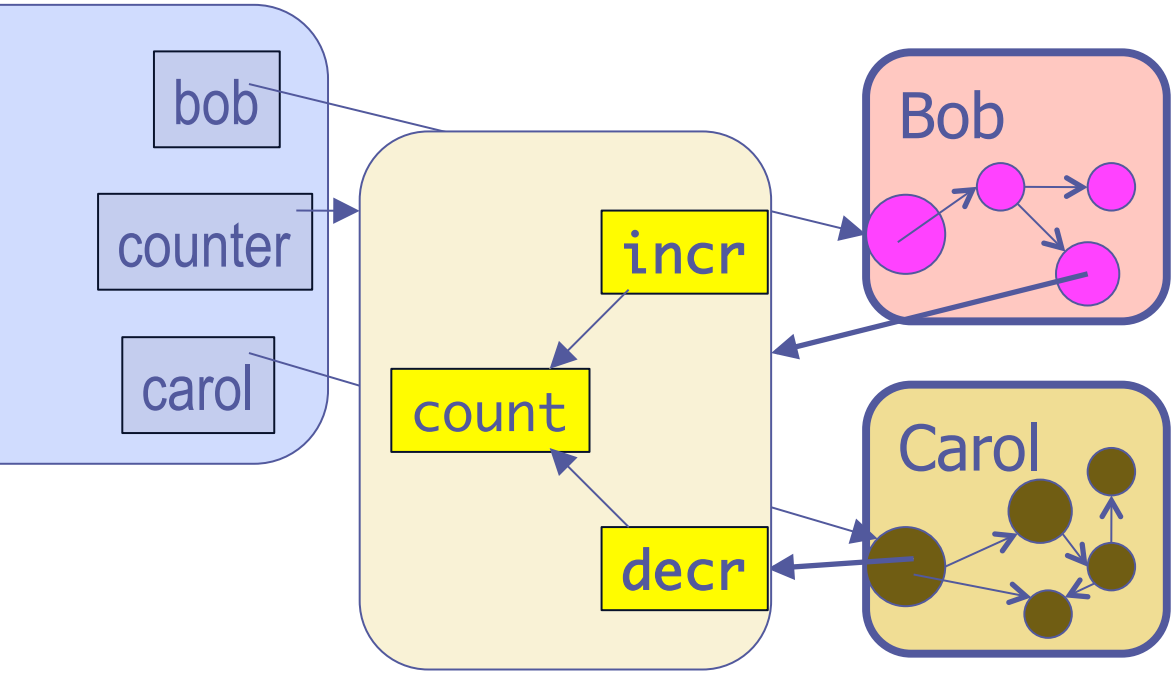
# Defensive Objects in SES on ES5




```
"use strict";
function makeCounter() {
    var count = 0;
    return def({
        incr: function() { return ++count; },
        decr: function() { return --count; }
    });
}
```

A *tamper-proof* record of
*lexical* closures *encapsulating* state
is a *defensible* object

# Goal Achieved!




Alice says:

```
var counter = makeCounter();
bob(counter);
carol(counter.decr);
```

Principle of least authority:

Bob can ***only*** count up and down and see result.

Carol can ***only*** count down and see the result.

# Lessons

- "this" rebinding
  - Avoid "this" and Prototypes
  - Use objects-as-closures or traits.js
  - (when security is worth extra allocations)
- Mutability Leakage
  - def
  - traits.js

# Encapsulation Riddle

```
function Table() {
    var array = [];
    return def({
        add:   function(v) { array.push(v); },
        store: function(i, v) { array[i] = v; },
        get:   function(i) { return array[i]; }
    });
}
```

Riddle: Steal array from table

# Encapsulation Riddle

```
function Table() {
    var array = [];
    return def({
        add:   function(v) { array.push(v); },
        store: function(i, v) { array[i] = v; },
        get:   function(i) { return array[i]; }
    });
}
```

Attack 1:

```
var stash;
table.store('push', function(v) { stash = this; });
table.add("doesn't matter");
```

# Encapsulation Riddle

```
function Table() {
    var array = [];
    return def({
        add:   function(v) { array.push(v); },
        store: function(i, v) { array[i] = v; },
        get:   function(i) { return array[i]; }
    });
}
```

Attack 2 by Jorge Chamorro on es-discuss:

```
var stash;
table.store('__proto__', { push: function(v) { stash = this; } });
table.add("doesn't matter");
```

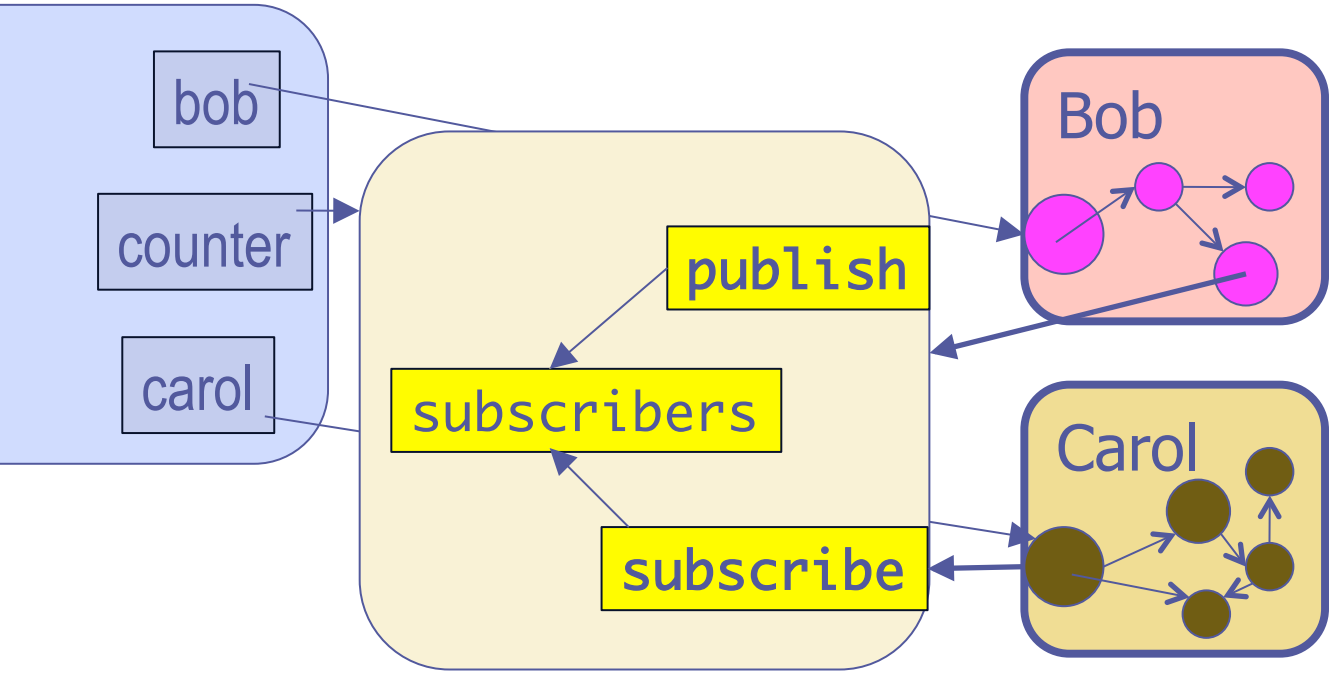
# Encapsulation Riddle, solved

```
function Table() {
    var array = [];
    return def({
        add:   function(v) { array.push(v); },
        store: function(i, v) { array[+i] = v; },
        get:   function(i) { return array[+i]; }
    });
}
```

## Both attacks foiled

```
var stash;
table.store('__proto__', { push: function(v) { stash = this; } });
table.add("doesn't matter");
```

# Publish or Perish




Bob can ***only*** publish or subscribe.
Carol can ***only*** subscribe.
All subscribers see all publications in order

# Publish or Perish

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                subscribers[+i](publication);
            }
        }
    });
}
```

Riddle: find three attacks

# Confusing Callbacks and Methods

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                subscribers[+i](publication);
            }
        }
    });
}
```

```
topic.subscribe(function evilSubscriber(publication) {
    this[+0] = evilSubscriber;
    this.length = 1;
});
```

# Confusing Callbacks and Methods

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                (1,subscribers[+i])(publication);
            }
        }
    });
}
```

```
topic.subscribe(function evilSubscriber(publication) {
    this[+0] = evilSubscriber;
    this.length = 1;
});
```

# Aborting the Wrong Plan

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                (1,subscribers[+i])(publication);
            }
        }
    });
}
```

```
topic.subscribe(function evilSubscriber(publication) {
    throw new Error("skip those losers");
});
```

# Nested Publication

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                (1,subscribers[+i])(publication);
            }
        }
    });
}
```

```
topic.subscribe(function evilSubscriber(publication) {
    topic.publish(outOfOrderPublication);
});
```

# Asynchronous Helpers

```
var applyFn = Function.prototype.call.bind(Function.prototype.apply);

function applyLater(func, self, args) {
    setTimeout(function() { applyFn(func, self, args); },
                    0);
}
```

obj.name(…args) ➔ applyFn(obj.name, obj, args)
func(…args) ➔ applyFn(func, undefined, args)

obj ! name(…args) ➔ applyLater(obj.name, obj, args)
func ! (…args) ➔ applyLater(func, undefined, args)

# Publish or Perish, solved

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                applyLater(subscribers[+i], undefined, [publication]);
            }
        }
    });
}
```

Thwarts all three attacks

# Publish or Perish, solved beautifully

```
function Topic() {
    var subscribers = [];
    return def({
        subscribe: function(subscriber) { subscribers.push(subscriber); },
        publish:    function(publication) {
            for (var i = 0; i < subscribers.length; i++) {
                subscribers[+i] ! (publication);
            }
        }
    });
}
```

Thwarts all three attacks

# New Skills open up New Worlds

## Remember learning

Avoid "this". Use closures rather than prototypes

Freeze everything by default: def, traits

Use bare "[" only for reflection: a[+i], map.get(k)

Deny callbacks inappropriate "this":

(1,x.foo)(args…), applyFn(x.foo, that, args)

Beware Synchronous Callbacks: Use applyLater or infix "**!**"

and various JS oo patterns and their hazards?

## Co-evolution of skills and tools

Student SES programmers think to avoid vulnerabilities.

Experts avoid enough traps to think about composition.

Libraries and IDEs need to learn to help.