

High Integrity JavaScript (HIJS)

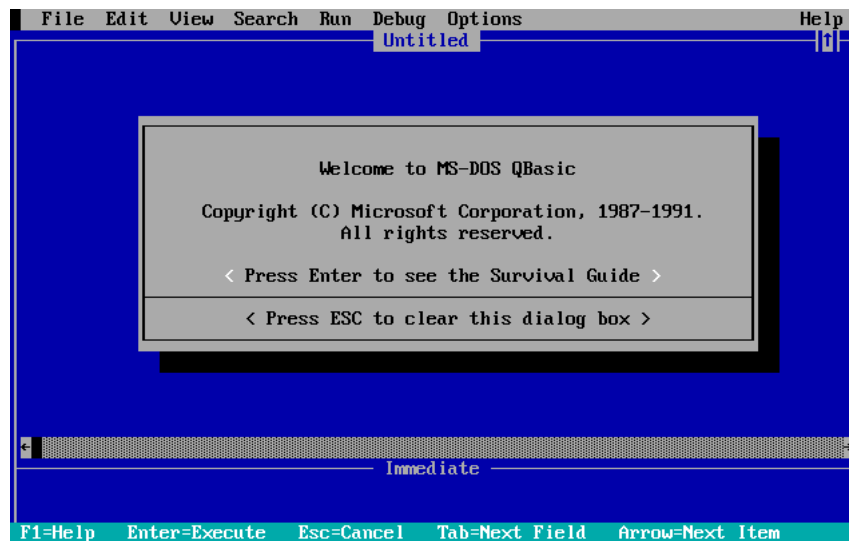
How to use and write 3rd party code so that it always works.

Nathan Wall

Nathan-Wall.github.io/hijs

A Couple Facts About Me

- I made an unassisted triple play in baseball when I was 8.
- That same year I started programming in this language:



JavaScript is highly malleable

```
function setDate(dateStr) {  
    var timestamp = Date.parse(date);  
    if (isNaN(timestamp)) {  
        // some code  
    } else {  
        // other code  
    }  
}
```

Built-In

DateJS

> Date.parse(null);	> Date.parse(null);
NaN	null
> Date.parse('');	> Date.parse('');
NaN	null
> Date.parse(2005);	> Date.parse(2005);
1104537600000	null

3 approaches:

1. **Don't worry about it.** Assume a safe environment.
2. **Lock the environment.** Prevent things from being done that you don't like (SES).
3. **Write code that always works.** ←

When should you care about high integrity?

- **Using 3rd party code**
 - Libraries and plugins
 - Analytics
 - Browser extensions
- **Writing 3rd party code**
- **Applications requiring security**
 - Banks
 - Social Networks
 - Email Clients

HIJS Goals

- Design scripts that run predictably, reliably, and securely given a consistent initialization environment.
- Leave the environment in a state that functions observably identically to the way it did when we got access to it.

Achieving High-Integrity

- Getting up to Speed on ECMAScript 5
- Writing General Purpose Code
- Private Variables
- Guarding Internal State

ECMAScript 5

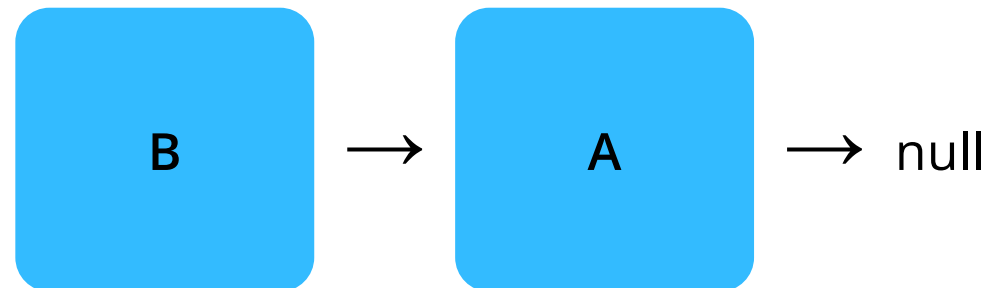
Provides a more hardened, securable language.

create

`Object.create(proto)`

Creates an object with as its prototype.

```
var A = Object.create(null),  
    B = Object.create(A);
```



defineProperty

`Object.defineProperty(obj, propName, desc)`

Can be used to define getters and setters on an object.

```
var A = { }, foo;  
Object.defineProperty(A, 'foo', {  
  get: function() {  
    return foo + '_extra';  
  },  
  set: function(value) {  
    foo = value;  
  }  
});
```

```
A.foo = 'bar';  
A.foo; // => 'bar_extra'
```

defineProperty

Data Descriptor	Accessor Descriptor
-----------------	---------------------

value	get
-------	-----

writable	set
----------	-----

enumerable	enumerable
------------	------------

configurable	configurable
--------------	--------------

freeze

`Object.freeze(obj)`

Locks an object's properties so that they can't be changed.

```
var A = { x: 1 };  
Object.freeze(A);  
A.x = 5;  
A.x; // => 1  
A.y = 2;  
A.y; // => undefined
```

bind

```
var _forEach = Array.prototype.forEach;
```

```
var foo = [ 'a', 'b', 'c', 'd', 'e' ],  
    forEachFoo = _forEach.bind(foo);
```

```
forEachFoo(function(item) {  
    console.log(item);  
});
```

// same as:

```
foo.forEach(function(item) {  
    console.log(item);  
});
```

General Purpose Code

- Store built-ins for later usage
- Evade naming collisions
- Support generic objects
- Be aware of the prototype chain

Store Built-Ins

Built-in functions can be overridden, so store the existing ones when your script initializes.

```
(function(Object, String, Date, undefined) {  
  
    'use strict';  
  
    // Store built-in functions for later usage.  
    var parse = Date.parse,  
        keys = Object.keys,  
        getOwnPropertyNames = Object.getOwnPropertyNames;  
  
    // ...  
  
    var array = [ ];  
    array.push('foo');  
  
})(Object, String, Date);
```

Naming Collisions

```
function eachKey(obj, callback) {  
    var key, value, isOwn;  
    for (key in obj) {  
        value = obj[key];  
        isOwn = obj.hasOwnProperty(key);  
        callback(key, value, isOwn);  
    }  
}
```

Calls a callback function for each property in an object, passing in the property name, value, and whether it is an `own` property.

```
eachKey({  
    "toString": "Converts an object to a string representation.",  
    "valueOf": "Converts an object to a value representation.",  
    "hasOwnProperty": "Determines if an object has an own property.",  
    "isPrototypeOf": "Determines if an object is another's prototype."  
}, display);  
, display);
```


Solution to Naming Collisions: Write Functionally

Don't depend on `Object.prototype`.

```
function eachKey(obj, callback) {  
    var key, value, isOwn;  
    for (key in obj) {  
        value = obj[key];  
        isOwn = hasOwn(obj, key);  
        callback(key, obj[key], isOwn);  
    }  
}  
  
var _hasOwnProperty = Object.prototype.hasOwnProperty;  
function hasOwn(obj, key) {  
    return _hasOwnProperty.call(obj, key);  
}
```

Supporting Generic Objects

```
function pluck(array, propertyName) {  
    return array.map(function(u) {  
        return u[propertyName];  
    });  
}
```

Name	
First:	<input type="text" value="William"/>
Middle:	<input type="text" value="Howard"/>
Last:	<input type="text" value="Taft"/>

```
var values = pluck([  
    document.getElementById('first_name'),  
    document.getElementById('middle_name'),  
    document.getElementById('last_name'),  
], 'value');  
  
values; // => [ "William", "Howard", "Taft" ]
```

Supporting Generic Objects

```
function pluck(array, propertyName) {  
  return array.map(function(u) {  
    return u[propertyName];  
  });  
}
```

```
var values = pluck(document.getElementsByTagName('input'), 'value');  
// => TypeError: Object #<NodeList> has no method 'map'
```

Solution to Supporting Generic Objects: Write Functionally

Don't depend on prototype methods.

```
function pluck(array, propertyName) {  
    return map(array, function(u) {  
        return u[propertyName];  
    });  
}  
  
var _map = Array.prototype.map;  
function map(arrayLike, callback) {  
    return _map.call(arrayLike, callback);  
}
```

Abstracting the process of turning a method into a function

Good

```
var _hasOwnProperty = Object.prototype.hasOwnProperty;
function hasOwn(obj, key) {
    return _hasOwnProperty.call(obj, key);
}
```

Better

```
var _hasOwnProperty = Object.prototype.hasOwnProperty,
    _call = Function.prototype.call,
    hasOwn = _call.bind(_hasOwnProperty);

var slice = _call.bind(Array.prototype.slice),
    map = _call.bind(Array.prototype.map),
    isPrototypeOf = _call.bind(Object.prototype.isPrototypeOf);
```

Lazy Bind (uncurryThis)

Converts a `function` into a `function` with **this** as its first argument.

```
var slice = lazyBind(Array.prototype.slice),  
    map = lazyBind(Array.prototype.map),  
    isPrototypeOf = lazyBind(Object.prototype.isPrototypeOf);  
  
function lazyBind(f) {  
    return _call.bind(f);  
}
```

Better

```
var _call = Function.prototype.call,  
    _bind = Function.prototype.bind,  
    lazyBind = _bind.bind(_call);  
  
var lazyBind = Function.prototype.bind.bind(Function.prototype.call);
```

Be Aware of the Prototype Chain

Do you really want that to inherit from `Object.prototype`?

```
var A = { }, foo;
Object.defineProperty(A, 'foo', {
  get: function() {
    return foo;
  },
  set: function(value) {
    foo = value;
  }
});
```

Be Aware of the Prototype Chain

Do you really want that to inherit from `Object.prototype`?

```
Object.prototype.value = 'gotcha!';
```

```
var A = { }, foo;
```

```
Object.defineProperty(A, 'foo', {  
  get: function() {  
    return foo;  
  },  
  set: function(value) {  
    foo = value;  
  }  
});
```

```
// value: 'gotcha!' (inherited)
```

```
});
```

```
=> TypeError: A property cannot have both accessors and a value.
```


Be Aware of the Prototype Chain

Solution: Use an object which inherits from `null`.

```
var create = Object.create;  
    defineProperty = Object.defineProperty,  
    keys = Object.keys,  
    forEach = lazyBind(Array.prototype.forEach);  
  
function define(obj, propName, desc) {  
    var D = create(null);  
    forEach(keys(desc), function(key) {  
        D[key] = desc[key];  
    });  
    defineProperty(obj, propName, D);  
}
```

Private Variables

- Separate interface from implementation.
- Only permit what is legitimately necessary.

Private in Class Syntax for ES6?

```
class Nameable {  
    constructor(private name) { }  
    getName() { return this@name; }  
}
```

The Underscore Pattern

```
function Foo() {  
    this._bar = Math.random();  
}  
Foo.prototype.getBar = function() {  
    return this._bar;  
};
```

Problems:

- Naming collisions
- No true encapsulation

The Neo-Classical Pattern

```
function Foo() {  
    var bar = Math.random();  
    this.getBar = function() {  
        return bar;  
    };  
}
```

Problems:

- Not compatible with prototypal inheritance
- No class-private variables

The BankAccount Example

```
var jane = new BankAccount(1000);  
var mike = new BankAccount(400);
```

```
mike.deposit(jane, 200);
```

```
jane.getBalance(); // => 800  
mike.getBalance(); // => 600
```

The Underscore Pattern

```
function BankAccount(balance) {  
    this._balance = balance;  
}  
BankAccount.prototype.getBalance = function() {  
    return this._balance;  
};  
BankAccount.prototype.deposit = function(from, amount) {  
    this._balance += amount;  
    from._balance -= amount;  
};
```

No True Encapsulation

The Neo-Classical Pattern

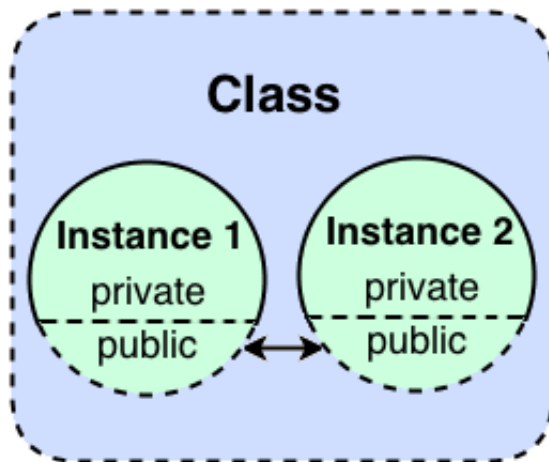
```
function BankAccount(balance) {  
  this.getBalance = function() {  
    return balance;  
  };  
  this.deposit = function(from, amount) {  
    // Add to this balance.  
    balance += amount;  
    // Can't access from.balance!  
  };  
}
```

Privates guarded by instance closures
cannot be accessed across instances.

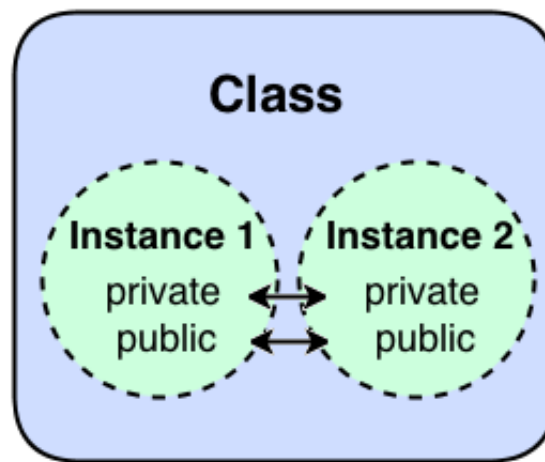
How can privileged changes across instances be made securely?

What you really want are variables.

Instance-Private



Class-Private

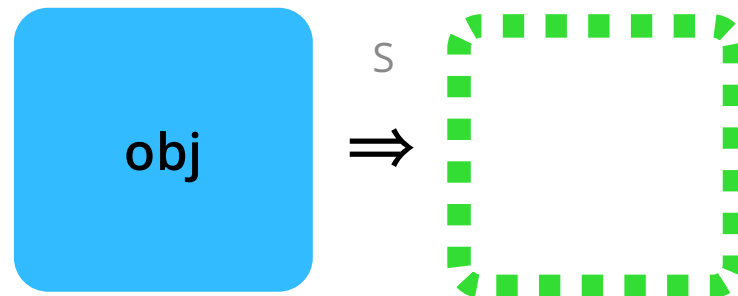


Secrets

github.com/Nathan-Wall/Secrets

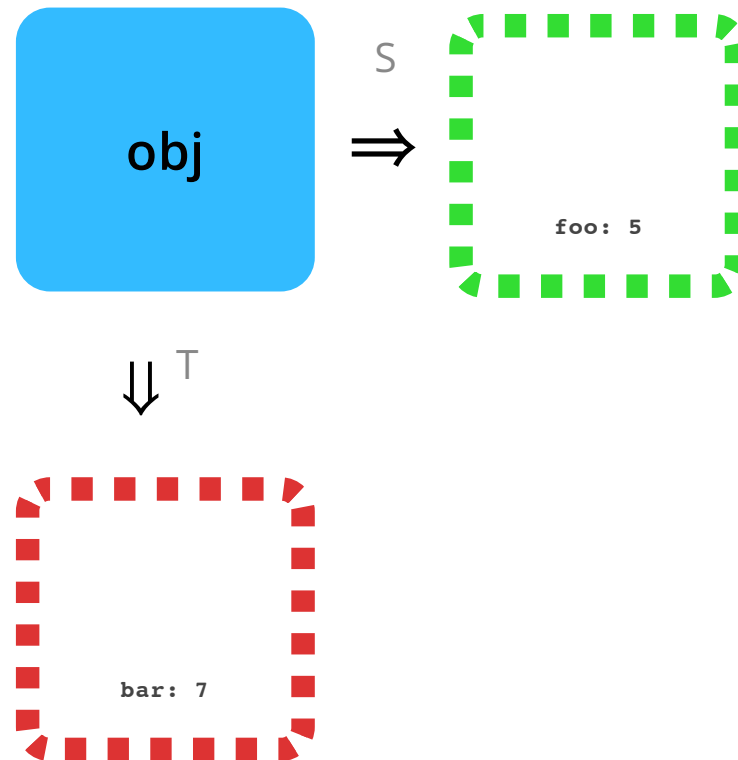
A secret is an object which is paired to a target object and used to store private information about the target.

```
var S = Secrets.create();  
var obj = { };  
S(obj).foo = 5;
```



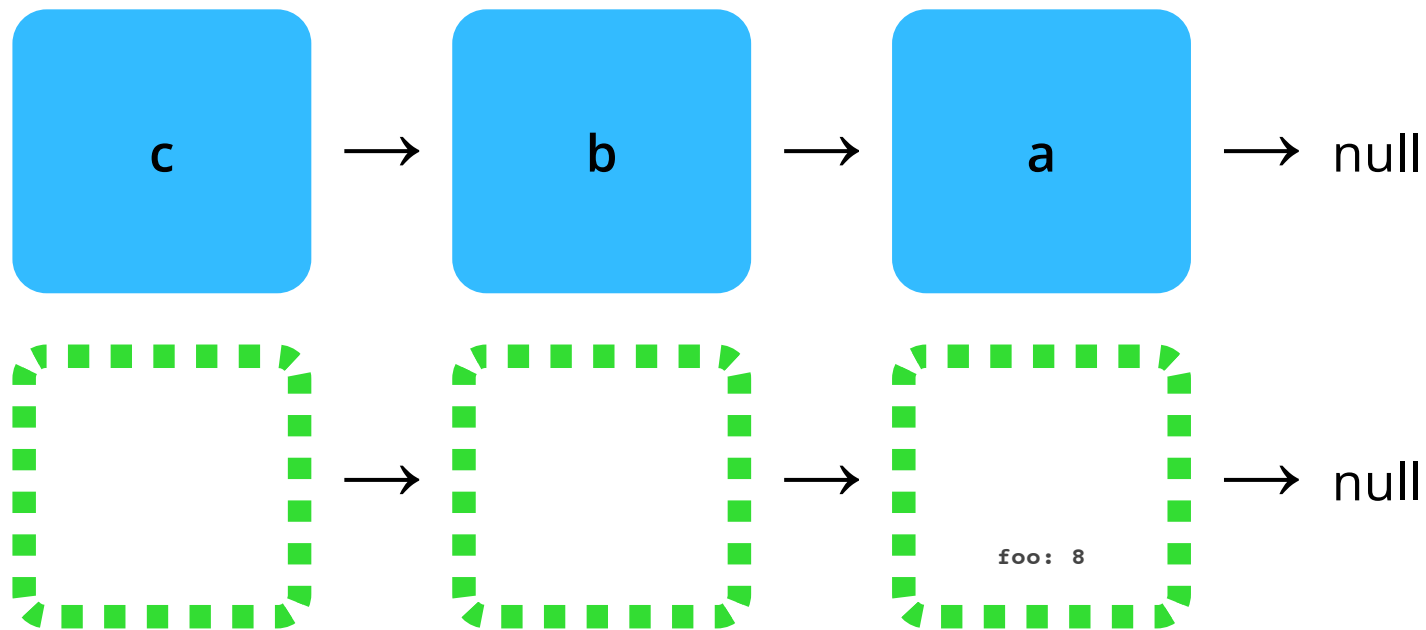
An object can have multiple secrets.

```
var S = Secrets.create();  
var obj = { };  
S(obj).foo = 5;  
var T = Secrets.create();  
T(obj).bar = 7;
```



Secrets have parallel prototype chains.

```
var a = Object.create(null),  
    b = Object.create(a),  
    c = Object.create(b);  
S(a).foo = 8;  
S(c).foo; // => 8
```



Secrets

```
var BankAccount = (function() {  
    var S = Secrets.create();  
    function BankAccount(balance) {  
        S(this).balance = balance;  
    }  
    BankAccount.prototype.getBalance = function() {  
        return S(this).balance;  
    };  
    BankAccount.prototype.deposit =  
        function(from, amount) {  
            S(this).balance += amount;  
            S(from).balance -= amount;  
        };  
    return BankAccount;  
})();
```

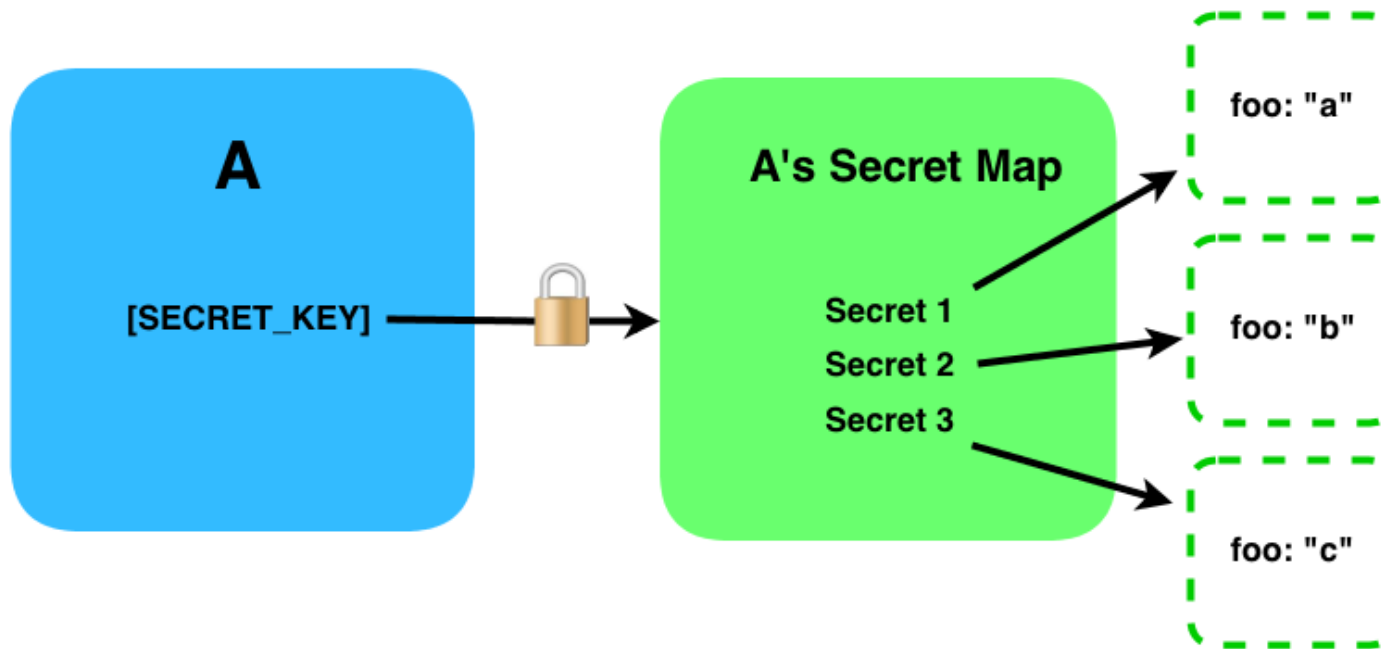
Protected Variables

```
var Nameable, Renameable;
(function() {
    var S = Secrets.create();

    Nameable = function(name) {
        S(this).name = name;
    };
    Nameable.prototype.getName = function() {
        return S(this).name;
    };

    Renameable = function(name) {
        Nameable.call(this, name);
    };
    Renameable.prototype = Object.create(Nameable.prototype);
    Renameable.prototype.setName = function(name) {
        S(this).name = name;
    };
})();
```

Under the Hood: Secrets in ES5



Steps `take`s:

1. Unlocks the lock.
2. Requests Secret Map from `A[SECRET_KEY]`.
3. Locks the lock.
4. Returns `secretMap[S_key]`.

Under the Hood: Secrets with WeakMaps

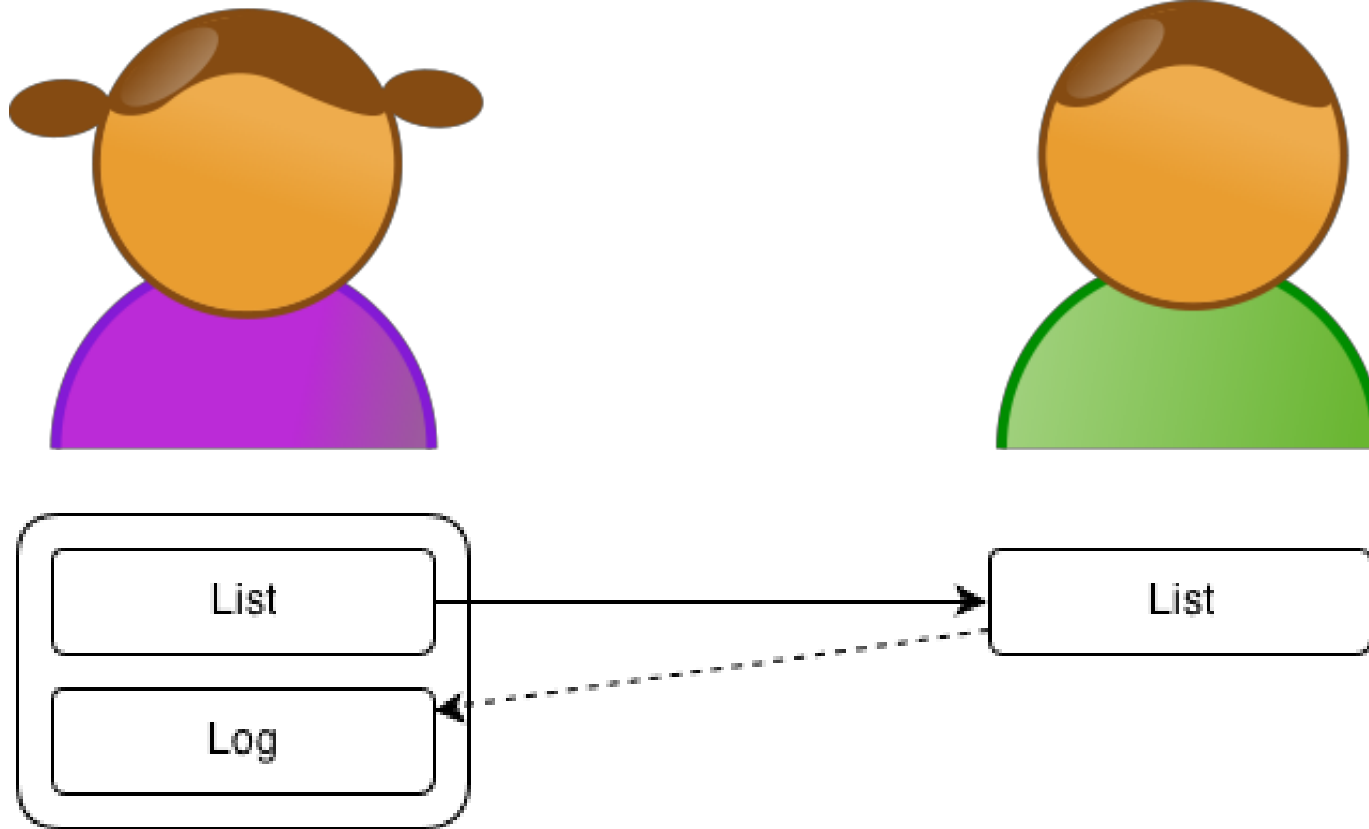
```
Secrets.create = function() {  
  var wm = new WeakMap();  
  return function(obj) {  
    var secret = wm.get(obj);  
    if (!secret) {  
      secret = createSecret(obj);  
      wm.set(obj, secret);  
    }  
    return secret;  
  };  
}
```


Guarding Internal State

It turns out to be more difficult to keep privates private than you might think!

Guarding Internal State

Making a Logged List



```
function LoggedList() {
  var array = [ ];
  this.add = function(value) {
    console.log('add', value);
    array.push(value);
  };
  this.get = function(index) {
    return array[index];
  };
  this.set = function(index, value) {
    console.log('set', index, value);
    array[index] = value;
  };
  Object.freeze(this);
}

var list = new LoggedList();
sendToBob(list);
```



Attack: Take advantage of insecure methods.

```
function sendToBob(list) {  
  var stolen;  
  list.set('push', function() {  
    stolen = this;  
  });  
  list.add('steal');  
  list.set('push', Array.prototype.push);  
  stolen.push('unlogged item');  
}
```



```
this.set = function(index, value) {  
  console.log('set', index, value);  
  array[index] = value;  
};  
this.add = function(value) {  
  console.log('add', value);  
  array.push(value);  
};
```



Solution: Neutralize arguments from external code.

```
function LoggedList() {
  var array = [ ];
  this.add = function(value) {
    console.log('add', value);
    array.push(value);
  };
  this.get = function(index) {
    return array[index];
  };
  this.set = function(index, value) {
    console.log('set', index, value);
    // `+index` coerces to number
    array[+index] = value;
  };
  Object.freeze(this);
}
```



Attack: Modify built-in prototypes.

```
function sendToBob(list) {  
    var push = Array.prototype.push;  
    var stolen;  
    Array.prototype.push = function(v) {  
        stolen = this;  
    };  
    list.add('steal');  
    Array.prototype.push = push;  
    stolen.push('unlogged item');  
}  
  
this.add = function(value) {  
    console.log('add', value);  
    array.push(value);  
};
```



Solution: Don't trust methods.

```
var push = lazyBind(Array.prototype.push);
```

```
function LoggedList() {  
  var array = [ ];  
  this.add = function(value) {  
    console.log('add', value);  
    push(array, value);  
  };  
  this.get = function(index) {  
    return array[index];  
  };  
  this.set = function(index, value) {  
    console.log('set', index, value);  
    array[+index] = value;  
  };  
  Object.freeze(this);  
}
```



Attack: Take advantage of built-in behavior.

```
function sendToBob(list) {  
  var stolen;  
  Object.defineProperty(Object.prototype, '0',  
    { set: function() { stolen = this; } }  
  );  
  list.add('steal');  
  stolen.push('unlogged item');  
}  
  
this.add = function(value) {  
  console.log('add', value);  
  push(array, value);  
};
```



Solution: Be cautious with built-ins.

```
var create = Object.create,  
    push = lazyBind(Array.prototype.push);
```

```
function LoggedList() {  
  var array = create(null);  
  this.add = function(value) {  
    console.log('add', value);  
    push(array, value);  
  };  
  this.get = function(index) {  
    return array[index];  
  };  
  this.set = function(index, value) {  
    console.log('set', index, value);  
    array[+index] = value;  
  };  
  Object.freeze(this);  
}
```



High Integrity

- Writing general purpose code
- Achieving private variables
- Guarding internal state

Questions?

Nathan Wall

nwall@appnexus.com

github.com/Nathan-Wall

