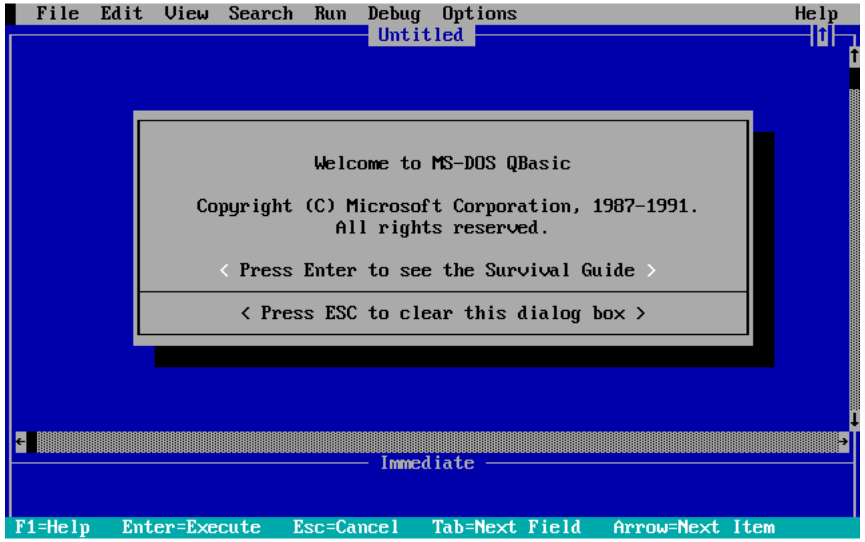
High Integrity JavaScript

Nathan Wall

A Few Facts About Me

- I'm part Creek (Native American).
- I made an unassisted triple play in baseball when I was 8.
- That same year I started programming in this language:



```
Welcome to MS-DOS QBasic

Copyright (C) Microsoft Corporation, 1987-1991.
All rights reserved.

< Press Enter to see the Survival Guide >

< Press ESC to clear this dialog box >
```

The Next 7 Slides

- Approaches to coping with JavaScript's extensibility
- Why should you care about high integrity?
- Getting up to speed on ECMAScript 5

Then...

- Achieving High Integrity

JavaScript is highly malleable

3 approaches:

1. **Don't worry about it.**
2. **Lock the environment.** Prevent things from being done that you don't like.
3. **Write code that always works.** ← *High Integrity*

Why should you care about high integrity?

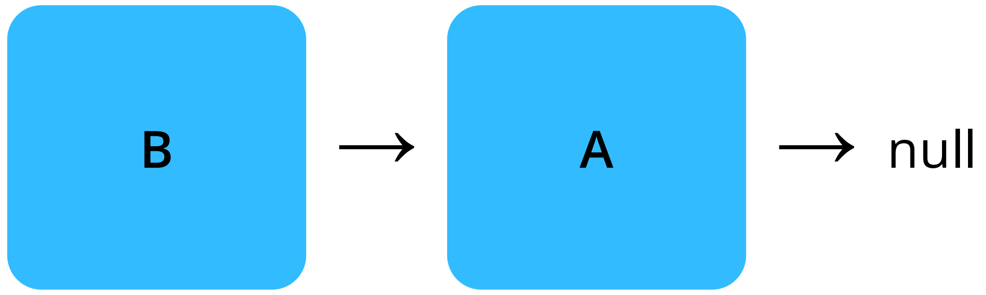
- Reusable code should be reusable anywhere.
- Don't unnecessarily limit the potential for creativity.
- JavaScript has matured enough to use in the writing of secure applications.

ECMAScript 5

`Object.create(`*`proto`*`)`

Creates an object with *proto* as its prototype.

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



ECMAScript 5

`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'
```

ECMAScript 5

`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);
});
```

Strict Mode

Strict mode fixes many security problems.

```
(function() {

    'use strict';

    // ...

})();
```

Achieving High-Integrity

- Writing General Purpose Code
- Private Variables
- Guarding Internal State

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) {

    'use strict';

    // Store built-in functions for later usage.
    var create = Object.create,
        keys = Object.keys,
        getOwnPropertyNames = Object.getOwnPropertyNames;

    // ...

})(Object, String);
```

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);
    }
}

{
  "object": "Object.prototype",
  "methods": {
    "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 protototype."
  }
}
```

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];
    });
}

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

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) {
    var rest = slice(arguments, 1);
    return _map.apply(arrayLike, rest);
}
var _slice = Array.prototype.slice
function slice(arrayLike, begin, end) {
    return _slice.call(arrayLike, begin, end);
}
```

Abstracting the process of turning a method into a function

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

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

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

Lazy Bind (uncurryThis)

Converts a *method* into a *function*.

```
var slice = lazyBind(Array.prototype.slice),
    forEach = lazyBind(Array.prototype.forEach),
    isPrototypeOf = lazyBind(Object.prototype.isPrototypeOf);
```

Example Uses

```
var toUpperCase = lazyBind(String.prototype.toUpperCase);
[ 'a', 'b', 'c' ].map(toUpperCase);
// => [ 'A', 'B', 'C' ]

var trim = lazyBind(String.prototype.trim);
var trimmedLines = linesOfText.split('\n').map(trim);
```

Lazy Bind (uncurryThis)

```
function lazyBind(f) {
    return _call.bind(f);
}

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.defineProperty(Object.prototype, 'value', {
    value: 'gotcha!'
});

var A = { }, foo;
Object.defineProperty(A, 'foo', {
    get: function() {
        return foo;
    },
    set: function(value) {
        foo = value;
    }
});
// => TypeError: A property cannot have both accessors and a value.
```

Be Aware of the Prototype Chain

```
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.

The Underscore Pattern

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

Problems:

- Name collisions
- No true encapsulation

The Module Pattern

```
function Foo(bar) {
    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 Module Pattern

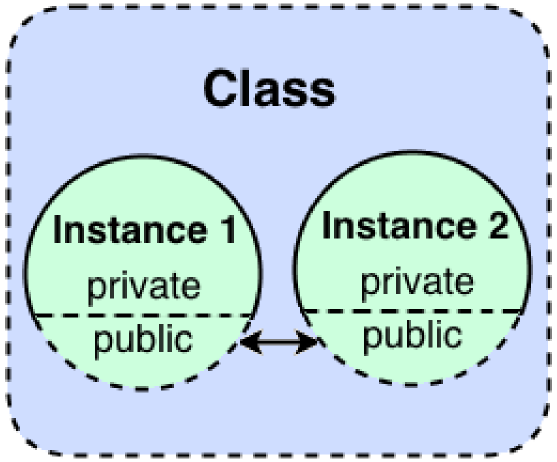
```
function BankAccount(balance) {
    this.getBalance = function() {
        return balance;
    };
    this.deposit = function(from, amount) {
        // Add to mike's balance.
        balance += amount;
        // How do we securely subtract an amount
        // from jane's account?
    };
}
```

Privates guarded by instance closures cannot be accessed across instances.

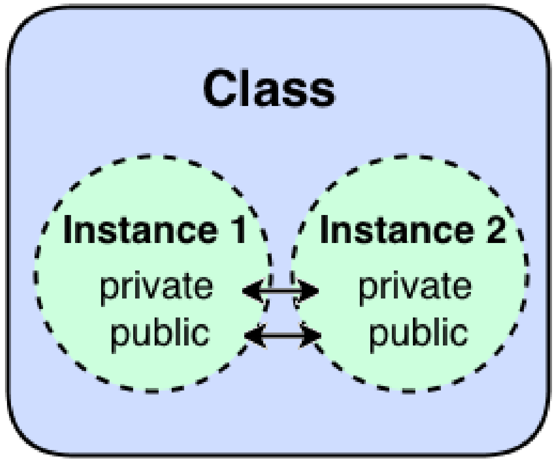
How can privileged changes across instances be made securely?

What you really want are *class-private* variables.

Instance-Private

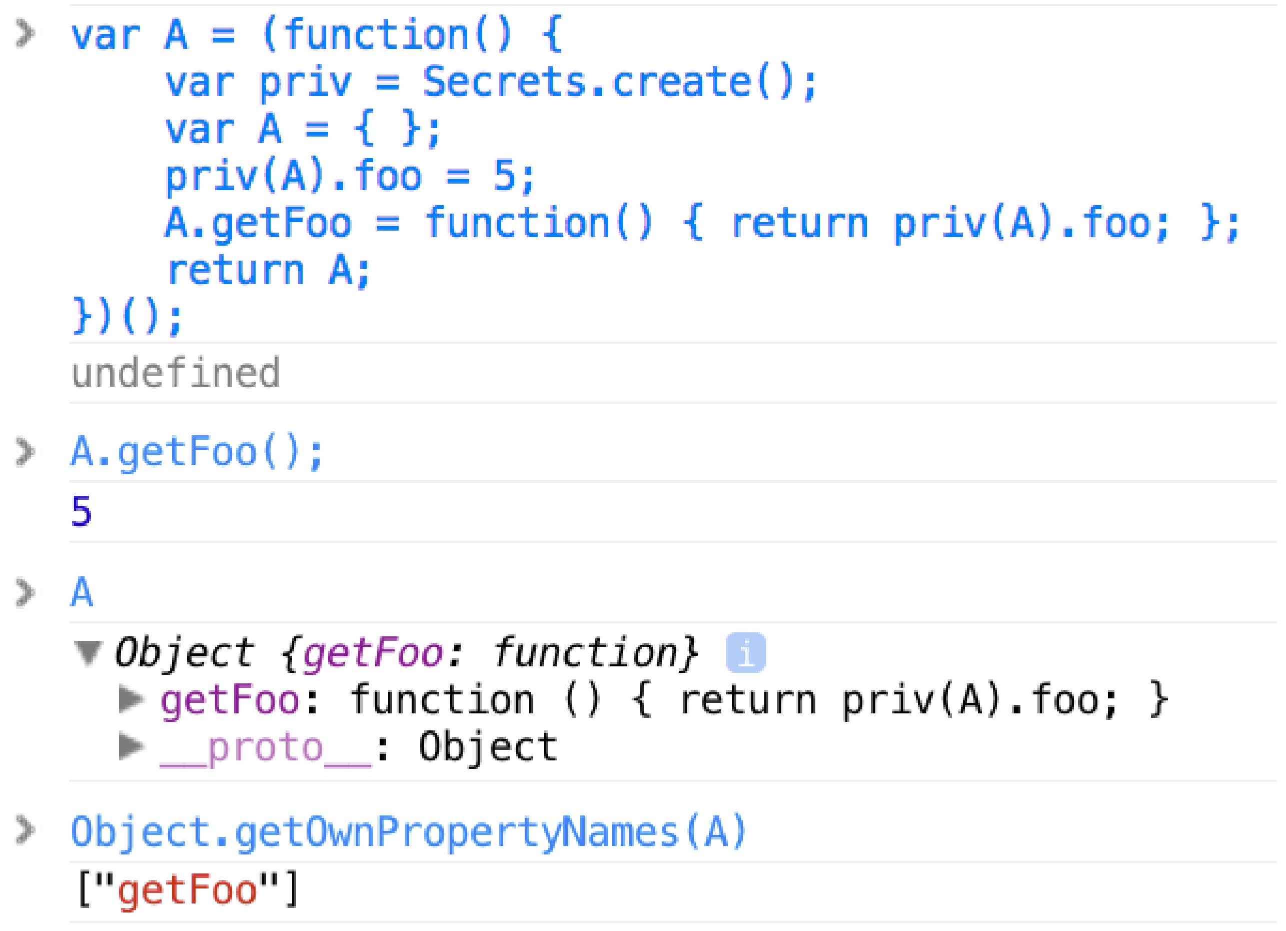


Class-Private



Secrets

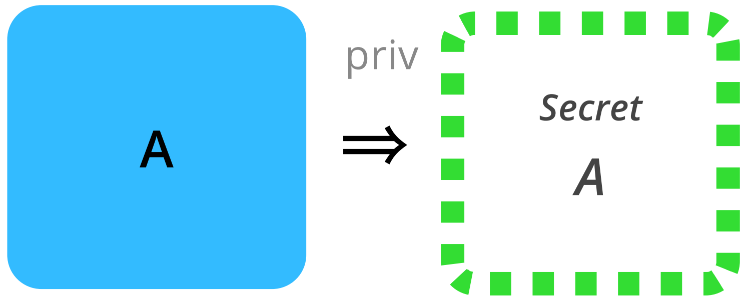
github.com/Nathan-Wall/Secrets



```
> var A = (function() {
      var priv = Secrets.create();
      var A = { };
      priv(A).foo = 5;
      A.getFoo = function() { return priv(A).foo; };
      return A;
  })();
  undefined
> A.getFoo();
  5
> A
  ▼ Object {getFoo: function}
    ▶ getFoo: function () { return priv(A).foo; }
    ▶ __proto__: Object
> Object.getOwnPropertyNames(A)
  ["getFoo"]
```

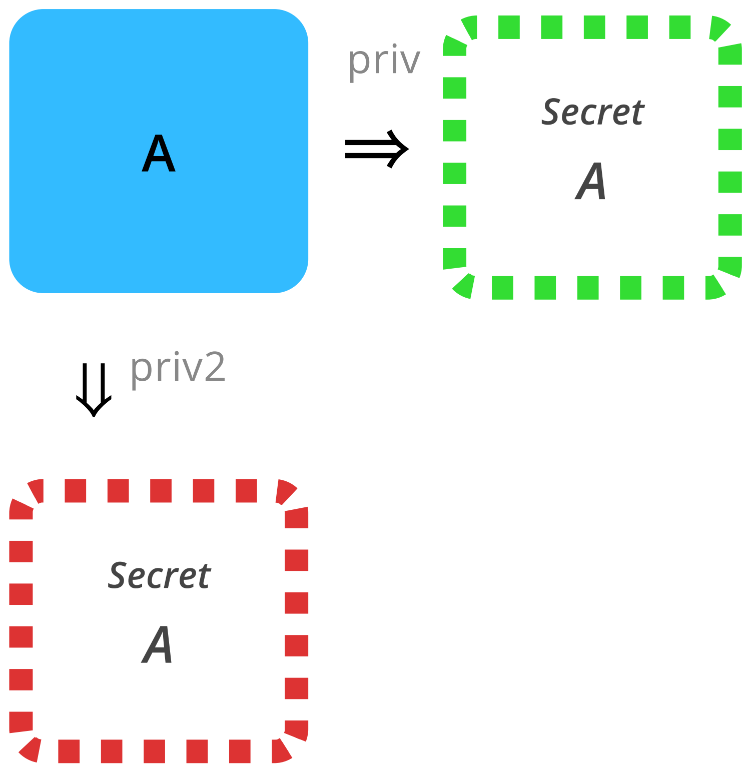
A secret is an object which is paired to a target object and used to store private information about the target.

```
var priv = Secrets.create();
var A = { };
priv(A).foo = 5;
```



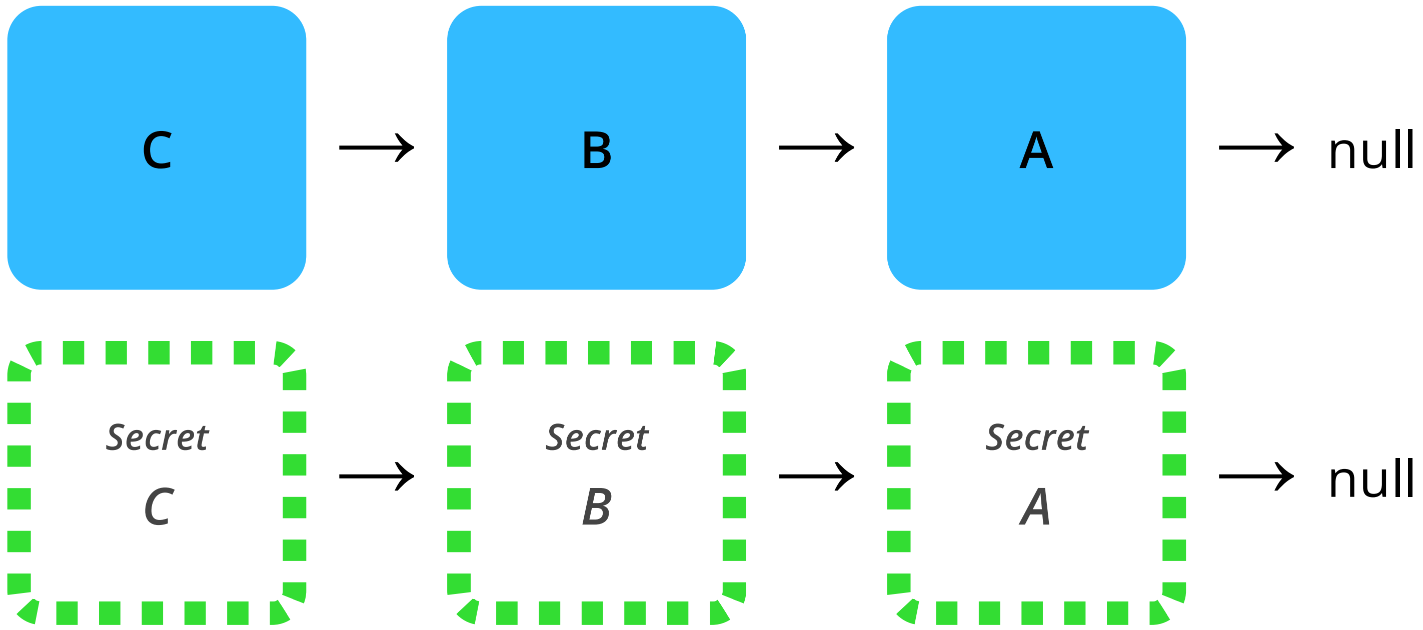
An object can have multiple secrets.

```
var priv = Secrets.create();
var A = { };
priv(A).foo = 5;
var priv2 = Secrets.create();
priv2(A).bar = 7;
```



Secrets have parallel prototype chains.

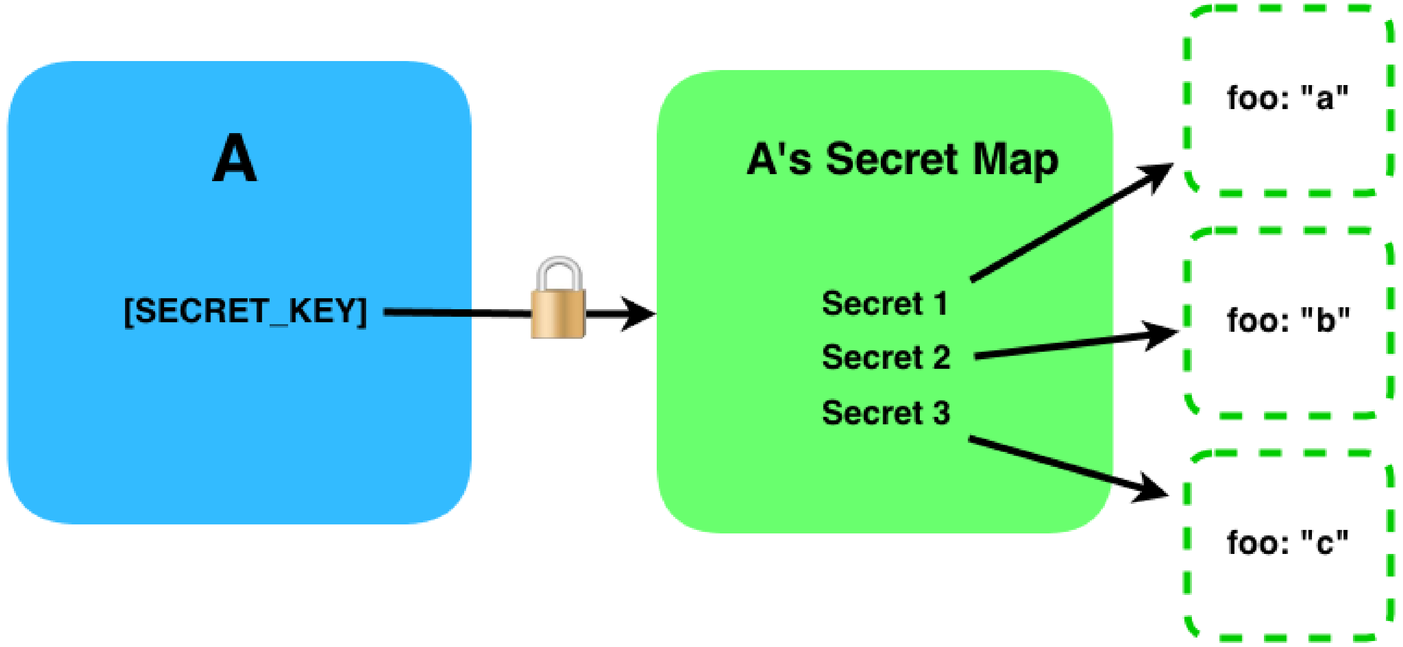
```
var A = Object.create(null),
    B = Object.create(A),    priv(A).foo = 8;
    C = Object.create(B);    priv(C).foo; // => 8
var SecretC = priv(C);
```



Secrets

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

Secrets in ES5



Steps *priv* takes:

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

Protected Variables

```
var Nameable, Renameable;
(function() {
    var prot = Secrets.create();
    Nameable = function(name) {
        prot(this).name = name;
    };
    Nameable.prototype.getName = function() {
        return prot(this).name;
    };
    Renameable = function(name) {
        Nameable.call(this, name);
    };
    Renameable.prototype = Object.create(Nameable.prototype);
    Renameable.prototype.setName = function(name) {
        prot(this).name = name;
    };
})();
```

Possibilities for ECMAScript 6

Private in Class Syntax?

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

Possibilities for ECMAScript 6

WeakMaps

(Don't work with prototypal inheritance)

```
let Nameable = (function() {
    let _name = new WeakMap();
    let Nameable = function(name) {
        _name.set(this, name);
    };
    Nameable.prototype.getName = function() {
        return _name.get(this);
    };
    return Nameable;
})();
```

Possibilities for ECMAScript 6

Private Symbols? (unlikely)

```
let Nameable = (function() {
    let _name = new Symbol(true);
    let Nameable = function(name) {
        this[_name] = name;
    };
    Nameable.prototype.getName = function() {
        return this[_name];
    };
    return Nameable;
})();
```

Possibilities for ECMAScript 6

Object.getPrivate?

(Prototypal inheritance?)

```
let Nameable = (function() {
    let _name = new Symbol();
    let Nameable = function(name) {
        Object.setPrivate(this, _name, name);
    };
    Nameable.prototype.getName = function() {
        return Object.getPrivate(this, _name);
    };
    return Nameable;
})();
```

Guarding Internal State

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

Guarding Internal State

```
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);
```

Guarding Internal State

```
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);
  };
```

Guarding Internal State

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);
}
```

Guarding Internal State

```
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);
  };
```

Guarding Internal State

Don't trust prototypes.

```
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);
}
```

Guarding Internal State

```
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);
    array.push(value);
  };
```

Guarding Internal State

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

Nathan Wall
nwall@appnexus.com

Questions?