Writing robust JavaScript code
or
JavaScript: the Good, the Bad, the Strict and the Secure Parts

Tom Van Cutsem
Talk Outline

• Part I: JavaScript, the Good and the Bad parts

• Part II: ECMAScript 5 and Strict Mode

• Part III: a glance at upcoming ECMAScript 6 features

• Part IV: Caja and Secure ECMAScript (SES)
Talk Outline

• This talk is about:
  • The JavaScript language proper
  • Language dialects and features to enable or improve security

• This talk is not about:
  • Security exploits involving JavaScript, or how to avoid specific exploits (e.g. XSS attacks)
About Me

• Professor of Computer Science at Vrije Universiteit Brussel, Belgium
  • Focus on Programming Languages & Distributed Systems

• ECMA TC39 (Javascript standardization committee)

• Visiting Faculty at the Google Caja team (2010)
Part I: Javascript, the Good and the Bad parts
What developers think about JavaScript

• Lightning talk Gary Bernhardt at CodeMash 2012

• https://www.destroyallsoftware.com/talks/wat
The world’s most misunderstood language

Good Parts: Functions

• Functions are first-class, may capture lexical variables (closures)

```javascript
var add = function(a,b) {
    return a+b;
}

add(2,3); // 5

function accumulator(s) {
    return function(n) {
        return s += n;
    }
}

var a = accumulator(0);
a(1); // 1
a(2); // 3

button.addEventListener('click', function (event) { ... });
```
Good Parts: Objects

- No class declaration needed, literal syntax, arbitrary nesting

```javascript
var bob = {
    name: "Bob",
    dob: {
        day: 15,
        month: 03,
        year: 1980
    },
    address: {
        street: "Main St.",
        number: 5,
        zip: 94040,
        country: "USA"
    }
};
```
Good Parts: combining objects and functions

- Functions can act as object constructors and methods

```javascript
function makePoint(i,j) {
  return {
    x: i,
    y: j,
    toString: function() {
      return '(' + this.x + ',' + this.y + ')';
    }
  };
}

var p = makePoint(2,3);
var x = p.x;
var s = p.toString();
```
A dynamic language...

// computed property access and assignment
p.x
p.x = 42;

// dynamic method invocation
p.toString();

// add new properties to an object at runtime
p.z = 0;

// delete properties from an object at runtime
delete p.x;
Bad Parts: global variables

- Scripts depend on global variables for linkage

Bad

```html
<script>
var x = 0; // global
var myLib = {
    inc: function() {
        return ++x;
    }
};
</script>

<script>
var res = myLib.inc();
</script>
```

Better

```html
<script>
var myLib = (function(){
    var x = 0; // local
    return {
        inc: function() {
            return ++x;
        }
    };
}());

<script>
var res = myLib.inc();
</script>
```
Bad Parts: with statement

• with-statement breaks static scoping

```javascript
var x = 42;
var obj = {};
with (obj) {
    print(x); // 42
    obj.x = 24;
    print(x); // 24
}
```
More Bad Parts

- Implicit type coercions
- No integers (all numbers are IEEE 754 floating point)
- "var hoisting": variables are not block-scoped but function-scoped
- Automatic semicolon insertion
- ...

Delving Deeper

• Some finer points about JavaScript functions and objects
Functions

- Functions are objects

```javascript
function add(x,y) { return x + y; }
add(1,2) // 3

add.doc = "returns the sum of two numbers";
```
Objects

- No classes.
- Instead, functions may be used as object constructors.
- All objects have a “prototype” link
  - Lookup of a property on an object traverses the prototype links
  - Similar to inheritance between classes
  - In some implementations, the prototype is an explicit property of the object named `__proto__`
function Point(x, y) {
    this.x = x;
    this.y = y;
}

Point.prototype = {
    toString: function() {
        return "[Point " +this.x+"," +this.y+"]";
    }
}

var p = new Point(1,2);
function Point(x, y) {
    this.x = x;
    this.y = y;
}

Point.prototype = {
    toString: function() {
        return "[Point " + this.x + "," + this.y + "]";
    }
}

var p = new Point(1, 2);
p.x;
p.toString();
function Point(x, y) {
    this.x = x;
    this.y = y;
}

Point.prototype = {
    toString: function() {
        return "[Point " + this.x + "," + this.y + "]";
    }
}

var p1 = new Point(1,2);
var p2 = new Point(3,4);
var p3 = new Point(5,6);
Summary so far

- Javascript: “a Lisp in C’s clothing” (D. Crockford)

- Good parts: functions, object literals

- Bad parts: global vars, lack of static scoping, ...

- Functions and objects work well together
Part II: ECMAScript 5 and Strict Mode
ECMAScript

- “Standard” Javascript
  - 1st ed. 1997
  - 2nd ed. 1998
  - 3rd ed. 1999
  - 4th ed.
  - 5th ed. 2009
  - 6th ed. end of 2014 (tentative) (draft already available)
ECMAScript 5 Themes

• New APIs, including JSON

• Support for more robust programming
  • Tamper-proof objects
  • Strict mode
ECMAScript 5 Themes

• **New APIs, including JSON**

• Support for more robust programming
  
  • Tamper-proof objects

  • Strict mode
JSON

- **JavaScript Object Notation**

- A subset of Javascript to describe *data* (numbers, strings, arrays and objects without methods)


```json
{   "name" : "Bob",
    "age" : 42,
    "address" : {   
        "street" : "Main st."
        
    
    
} }
ECMAScript 5 and JSON

• Before ES5, could either parse quickly or safely

• Unsafe: `eval(jsonString)`

• In ES5: use `JSON.parse`, `JSON.stringify`

```
{“a”:1, “b”:[1,2], “c”: “hello”}
```

```
[“a”:1, “b”:[1,2], “c”: “hello”]

‘{“a”:1,
 “b”:[1,2],
 “c”: “hello”}’
```
ECMAScript 5 Themes

• New APIs, including JSON

• **Support for more robust programming**
  
  • Tamper-proof objects
  
  • Strict mode
Tamper-proof Objects: motivation

- Objects are *mutable* bags of properties
- No way to protect an object from modifications by its clients
- *Unsafe* to share objects across trust boundaries on a single page
- Problematic when embedding third-party scripts on your page
- Necessary first step: protect objects from third-party modifications
Tamper-proof Objects

```javascript
var point = {
  x: 0,
  y: 0
};

Object.preventExtensions(point);
point.z = 0; // error: can't add new properties

Object.seal(point);
delete point.x; // error: can't delete properties

Object.freeze(point);
point.x = 7; // error: can't assign properties
```
EcmaScript 5 Strict mode

• Safer, more robust, subset of the language

• Why?
  • No silent errors
  • True static scoping rules
  • No global object leakage
EcmaScript 5 Strict mode

• Explicit opt-in to avoid backwards compatibility constraints

• How to opt-in

  • Per “program” (file, script tag, ...)

  • Per function

• Strict and non-strict mode code can interact (e.g. on the same web page)

<script>
  "use strict";
  ...
</script>

function f() {
  "use strict";
  ...
}
Strict-mode opt-in: gotcha’s

- Beware: minification and deployment tools may concatenate scripts

```html
<script>
"use strict";
// in strict mode
</script>

<!-- not in strict mode -->

<script>
// not in strict mode
function f(){...}
</script>

<!-- accidentally strict! -->

<script>
"use strict";
// accidentally strict!
function f(){...}
</script>
Strict-mode opt-in: gotcha’s

- Suggested refactoring is to wrap script blocks in function bodies

```html
<script>
(function(){
  "use strict";
  // in strict mode
}())
</script>

<script>
// not in strict mode
function f(){...}
</script>

<script>
(function(){
  "use strict";
  // in strict mode
}())

// not in strict mode
function f(){...}
</script>
```
Static scoping in ES5

- ECMAScript 5 non-strict is not statically scoped

- Four violations:
  - `with (obj) { x }` statement
  - `delete x;` // may delete a statically visible var
  - `eval('var x = 8');` // may add a statically visible var
  - Assigning to a non-existent variable creates a new global variable
    
    ```javascript
    function f() { var xfoo; xFoo = 1; }
    ```
Ecmascript 5 Strict: syntactic restrictions

- The following are forbidden in strict mode (signaled as syntax errors):

```javascript
with (expr) {
  ...x...
}

{ a: 1,
  b: 2,
  b: 3 } // duplicate property

function f(a, b, b) {
  // repeated param name
}

delete x; // deleting a variable

if (a < b) {
  // declaring functions in blocks
  function f() {}[]
}

var n = 023; // octal literal

function f(eval) {
  // eval as variable name
}
```
EcmaScript 5 Strict

- Runtime changes (fail silently outside of strict mode, throw an exception in strict mode)

```javascript
function f() {
    "use strict";
    var xfoo;
    xFoo = 1; // error: assigning to an undeclared variable
}

"use strict";
var p = Object.freeze({x:0,y:0});
delete p.x; // error: deleting a property from a frozen object
```
EcmaScript 5 Strict: avoid global object leakage

- Runtime changes: default this bound to undefined instead of the global object

```javascript
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p = new Point(1,2);
var p = Point(1,2);
// window.x = 1;
// window.y = 2;
print(x) // 1 (bad!)
```

```javascript
"use strict";
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p = new Point(1,2);
var p = Point(1,2);
// undefined.x = 1;
// error (good!)
```
Direct versus Indirect Eval

- ES5 runtime changes to eval (both in strict and non-strict mode)
- eval “operator” versus eval “function”

**Direct Eval**

```javascript
function f() {
    var x = 0;
    eval("x = 5");
    return x;
}
f() // returns 5
```

**Indirect Eval**

```javascript
function f(g) {
    var x = 0;
    g("x = 5");
    return x;
}
f(eval) // returns 0
```
ECMAScript 5 Themes: summary

• New APIs, including JSON

• Support for more robust programming
  • Tamper-proof objects
  • Strict mode
Part III: ECMAScript 6
ECMAScript 6

• Many new additions (too many to list here *)

• Classes

• Modules

• String Templates

• Proxies

* see http://kangax.github.io/es5-compat-table/es6/ for an overview of ES6 features
ECMAScript 6 classes

- All code inside a class is implicitly opted into strict mode!

```javascript
function Point(x, y) {
    this.x = x;
    this.y = y;
}

Point.prototype = {
    toString: function() {
        return "[Point...]";
    }
}

var p = new Point(1,2);
p.x;
p.toString();
```

```javascript
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    toString() {
        return "[Point...]";
    }
}

var p = new Point(1,2);
p.x;
p.toString();
```
ECMAScript 6 modules

- All code inside a module is implicitly opted into strict mode!

```html
<script>
var x = 0; // global
var myLib = {
    inc: function() {
        return ++x;
    }
};
</script>

```html
<script type="module"
name="myLib">
var x = 0; // local!
export function inc() {
    return ++x;
}
</script>

```html
<script>
var res = myLib.inc();
</script>

```html
<script type="module">
import { inc } from 'myLib';
var res = inc();
</script>
```
ECMAScript 6 string templates

- String interpolation (e.g. for templating) is very common in JS
- Vulnerable to injection attacks

```javascript
function createDiv(input) {
    return "<div>" + input + "</div>";
}

createDiv("</div><script>...");
// "<div></div><script>...</div>"
```
ECMAScript 6 string templates

- String templates combine convenient syntax for interpolation with a way of automatically building the string

```javascript
function createDiv(input) {
    return html`<div>${input}</div>`;
}

createDiv("</div><script>...");
// "<div>&lt;/div&gt;&lt;script&gt;..."</div>"
ECMAScript 6 string templates

- User-extensible: just sugar for a call to a template function
- Expectation that browser will provide html, css template functions

```javascript
function createDiv(input) {
    return html(['<div>', '</div>'], input);
}

createDiv('</div><script>…');
// "<div>"<script>…"
```
ECMAScript 6 proxies

- Dynamic proxy objects: objects whose behavior can be controlled in JavaScript itself

- Goals:
  - Create generic object wrappers
  - Emulate host objects
Host objects

- Objects provided by the host platform
- E.g. the **DOM**: a tree representation of the HTML document
- Appear to be Javascript objects, but not implemented in Javascript
- Can have odd behavior that regular JavaScript objects cannot emulate
Proxy example: log all property accesses

```javascript
function makePoint(x, y) {
    return {
        x: x,
        y: y
    };
}
var p = makePoint(2,2);
var lp = makeLogger(p);
lp.x
// log: get x
// returns 2
lp.y = 3
// log: set y 3
```
ECMAScript 6 proxies

```javascript
var proxy = new Proxy(target, handler);

handler.get(target, 'foo')

handler.set(target, 'foo', 42)

proxy.foo

proxy.foo = 42
```
Example: logging all property accesses

```javascript
function makeLogger(target) {
    var proxy = new Proxy(target, {
        get: function(target, name) {
            console.log('get', name);
            return target[name];
        },
        set: function(target, name, val) {
            console.log('set', name, val);
            return target[name] = val;
        },
    });
    return proxy;
}
```
Example: logging all property accesses

```javascript
function makeLogger(target) {
    var proxy = new Proxy(target, {
        get: function(target, name) {
            console.log('get', name);
            return target[name];
        },
        set: function(target, name, val) {
            console.log('set', name, val);
            return target[name] = val;
        }
    });
    return proxy;
}
```
Example: logging all property accesses

```javascript
function makeLogger(target) {
    var proxy = new Proxy(target, {
        get: function(target, name) {
            console.log('get', name);
            return target[name];
        },
        set: function(target, name, val) {
            console.log('set', name, val);
            return target[name] = val;
        }
    });
    return proxy;
}
```
Another Proxy example: a revocable reference

- revocable reference: limit the lifetime of an object reference

```javascript
var {proxy, revoke} = makeRevocable(resource);
plugin.run(proxy);
// later
revoke();
```
Another Proxy example: a revocable reference

- revocable reference: limit the lifetime of an object reference

```javascript
var {proxy, revoke} = makeRevocable(resource);
plugin.run(proxy);
// later
revoke();
```
Another Proxy example: a revocable reference

- revocable reference: limit the lifetime of an object reference

```javascript
var {proxy, revoke} = makeRevocable(resource);
plugin.run(proxy);
// later
revoke();
```
Another Proxy example: a revocable reference

- revocable reference: limit the lifetime of an object reference

```javascript
var {proxy, revoke} = makeRevocable(resource);
plugin.run(proxy);
// later
revoke();
```
Another example: a revocable reference

```javascript
function makeRevocable(resource) {
    var enabled = true;
    return {
        proxy: new Proxy(resource, {
            get: function(target, name) {
                if (enabled) {
                    return target[name];
                }
                throw new Error("revoked");
            }
        }),
        revoke: function() { enabled = false; }
    }
}
```
Proxies: availability

- Firefox
- `node --harmony`
- Chrome (enable experimental JS flag in chrome://flags)
- Library that implements the latest ES6 specification

```html
<script src="reflect.js"></script>
```

- Available on Github: [https://github.com/tvcutsem/harmony-reflect](https://github.com/tvcutsem/harmony-reflect)
Part IV: Caja and Secure ECMAScript (SES)
Caja enables the safe embedding of third-party active content inside your website

- Secures Google Sites
- Secures Google Apps Scripts

More generally: Gadgets, Mashups:

https://developers.google.com/caja/docs/about/
Caja

- Not a traditional sandbox. Caja-compiled code is safe to inline directly in a webpage `<div>`. No iframes. No web workers.

- Can put multiple third-party apps into the same page and allow them to directly exchange JavaScript objects
  
  - Great for writing mash-ups

- The host page is protected from the embedded apps
  
  - E.g. embedded app can’t redirect the host page to phishing sites, or steal cookies from the host page
Caja : Taming

- Caja proxies the DOM. Untrusted content interacts with a virtual DOM, never with the real DOM.

https://developers.google.com/caja/docs/about/
• Example: Caja Playground

• http://caja.appspot.com
Caja consists of:

- A capability-secure JavaScript subset (SES)
- A safe DOM wrapper (Domado)
- A HTML and CSS sanitizer (sandbox scripts embedded in HTML/CSS)
- SES is the portion of Caja responsible for securing JavaScript
Secure ECMAScript

- SES adds confinement
- ES5/strict adds proper static scoping
- ES5 adds tamper-proof objects
- ES3
Secure ECMAScript

- Implemented as a library on top of ES5/strict
- Include as first script, before any other JavaScript code runs:

  <script src="startSES.js"></script>
Secure ECMAScript

<script src="startSES.js"></script>

- Deep-frozen global environment (incl. frozen global object)
  - Can’t update properties of Object, Array, Function, Math, JSON, etc.

- Whitelisted global environment
  - No “powerful” non-standard globals (e.g. document, window, XMLHttpRequest, ...)
  - Code that spawns an SES environment may provide selective access to these

- Patches eval and Function to accept only ES5/strict code that can only name global variables on the whitelist
Proxies again

• Caja uses object capabilities to express security policies

• In the object-capability paradigm, an object is powerless unless given a reference to other (more) powerful objects

• Common to wrap objects with proxies that define a security policy
  • E.g. revocable reference: limit the lifetime of an object reference
Wrap-up
Wrap-up

ES3

ES5

ES5/strict

SES

JavaScript: the Good, the Bad, the Strict, and the Secure parts.
Take-home messages

• Strict mode: a saner JavaScript (opt-in in ES5)

• ES6 builds on strict mode (classes and modules)

• Secure ECMAScript (SES) builds on strict mode

• If you want your code to be securable, opt into strict mode

• Proxies are a power-tool to express fine-grained security policies
References

• Warmly recommended: Doug Crockford on JavaScript
  http://goo.gl/FGxmM (YouTube playlist)
References

• ECMAScript 5:
  
  • “Changes to JavaScript Part 1: EcmaScript 5” (Mark S. Miller, Waldemar Horwat, Mike Samuel), Google Tech Talk (May 2009)
  

• Caja: https://developers.google.com/caja

• SES: http://code.google.com/p/google-caja/wiki/SES

• Proxies: http://soft.vub.ac.be/~tvcutsem/invokedynamic/proxies_tutorial

• ES6 latest developments: http://wiki.ecmascript.org and the es-discuss@mozilla.org mailing list.
  ES6 Modules: http://www.2ality.com/2013/07/es6-modules.html