Shape Sorter photo by Ella's Dad, CC BY 2.0

JavaScript Object Type Coercion

Thomas Hunter II
Aug 28, 2018 · 5 min read

There are optional, user-definable methods which will be called when performing various actions upon JavaScript objects, such as coercing them into a primitive value, serializing their contents, or even logging them. In this post we're going to look at these coercion situations and their corresponding methods.

At Intrinsic, we build a product to secure Node.js applications. We do so by locking down access to sensitive resources and only allowing whitelisted operations. This requires intimate knowledge of the internals of Node.js, as well as care when interacting with untrusted input from app code and third party dependencies. Untrusted code can be intentionally misleading when making use of these coercing methods.

First, an Attack

Suppose you're implementing a JavaScript library which accepts a URL, checks that it's valid, then adds auth data and makes a request. Perhaps your implementation would look like the following:

```
const VALID = 'https://api.example.org'

// Warning: Unsafe Validation
function validateAndAuth(url) {
  if (!String(url).startsWith(VALID)) {
    return false
  }

  return url + '?user=123&token=456'
}

validateAndAuth('https://api.example.org/v1/photos')
// Returns `https://api.example.org/v1/photos'
validateAndAuth('https://api.evil.org/evil/endpoint')
// Returns false
```

Unfortunately, an attacker using your library can easily bypass your seemingly acceptable security checks. For example, we can construct an object which would allow the following request to succeed and send information to an evil third party:

```
validateAndAuth(evilArgument)
// Returns 'https://api.evil.org/evil/endpoint'
```

Now, let's take a look at a bunch of different ways that user-provided code will get called when interacting with objects.

Object#toString()

The method `.toString()` is used when an object is coerced into a string. This is helpful when you want to get a nice string representation of an object vs. the not-so-useful `"[object Object]"`.

```
const s = {
  x: 7,
  y: 3,
  toString: function() {
    return `Square: ${this.x},${this.y}`
  }
}

console.log(s)
// { x: 7, y: 3, toString: [Function: toString] }
console.log(s + '')
// 'Square: 7,3'
console.log(+s)
// NaN
console.log(JSON.stringify(s)) // ignores
// '{"x":7,"y":3}'
```

Object#valueOf()

The method `.valueOf()` is used when an object is coerced into a primitive value such as a number. If you've ever added a number to an object and ended up with a value of `NaN`, then this method is your friend.

```
const box = {
  l: 7,
  w: 3,
  h: 4,
  valueOf: function() {
    return this.l * this.w * this.h
  }
}

console.log(box)
// { [Number: 84] l: 7, w: 3, h: 4, valueOf: [Function: valueOf] }
console.log(String(box))
// [object Object]
console.log(+box)
// 84
```

Object#[Symbol.toPrimitive]()

The most modern type coercion feature in JavaScript is the "well-known" symbol `Symbol.toPrimitive`, which can be used as an object method. This method provides a common interface for coercing an object into a primitive and supersedes the `.toString()` and `.valueOf()` methods.

When this method is run it will receive a single `hint` argument, which can have a value of either `number`, `string`, or `default`, depending on the situation in which the coercion occurs:

```
const p = {
  val: 'Thomas',
  [Symbol.toPrimitive]: function(hint) {
    if (hint === 'number') {
      return this.val.length
    } else if (hint === 'string') {
      return this.val
    }
    // hint === 'default'
    return undefined
  }
}

console.log(+p) // hint = 'number'
// 6
console.log(String(p)) // hint = 'string'
// 'Thomas'
console.log(!!p) // doesn't call the method
// true
console.log(p + '') // hint = 'default'
// 'undefined'
```

The `[Symbol.toPrimitive]()` method takes precedence over both the `.toString()` and `.valueOf()` methods, if both have been provided:

```
const poly = {
  valueOf() { return 99 }, // never called
  toString() { return 'toString' }, // never called
  [Symbol.toPrimitive](h) {
    return h === 'number' ? 42 :
      h === 'string' ? 'toPrimitive' :
      null
  }
}

console.log(Number(poly))
// 42
console.log(String(poly))
// 'toPrimitive'
console.log(poly + '')
// 'null'
```

Object#toJSON()

The `.toJSON()` method is designed to be used when serializing an object using `JSON.stringify()`.

Often, when serializing a complex object into a string-representation, there are attributes that we simply don't need or which otherwise cannot be represented using JSON. For example, functions cannot be represented using JSON. `Date`s will be represented as an ISO string and may contain more precision than is required. Redundant data provided as a convenience is also not needed (e.g. an `area` attribute on a square object when a `width` and `height` is already provided). Also, cyclical references in an object will throw an error when being serialized.

The `.toJSON()` method provides a final opportunity to cleanup the object before being converted into a JSON string representation.

```
const tom = {
  name: 'Thomas',
  dob: new Date('1986-04-01'),
  toJSON: function() {
    return {
      name: this.name,
      birthday: this.dob.toISOString().split('T')[0]
    }
  }
}

console.log(JSON.stringify(tom))
// '{"name":"Thomas","birthday":"1986-04-01"}'
```

The `JSON.stringify()` function doesn't take any of the aforementioned methods into consideration, only `toJSON()`.

Node.js: require('util').inspect.custom

Next, let's consider a special case used by Node.js. This method is a `Symbol` and is used when performing `util.inspect()` or `console.log()` (e.g. logging operations) on an object. The return value of this method is what will be used when when displaying the output in a terminal:

```
const util = require('util')

const flower = {
  name: 'Rose'
}

flower[util.inspect.custom] = function(depth, options) {
  return `@~}~~~ ${this.name} ~~~{~@`
}

console.log(flower)
// @~}~~~ Rose ~~~{~@
```

This method is useful for inserting escape sequences to colorize output or to otherwise format an object for readability.

Mitigating our Earlier Attack

Now that you're equipped with knowledge of object coercion, do you have an idea of how the original attack was crafted?

The security mechanism in the first version of our library will be bypassed when passed the following object:

```
const evilArgument = {
  calls: 0,
  toString() {
    this.calls++
    if (this.calls >= 2) {
      return 'https://api.evil.org/evil/endpoint'
    }
    return 'https://api.example.org'
  }
}
```

As you can see, instead of passing in a string, the attacker has passed in an object. This object has a `.toString()` method (though it could just as easily used the `Symbol.toPrimitive` approach). When we coerced the URL to a string the first time (this happened when we checked if the URL is valid), the result we received contained a whitelisted URL. However when we coerced the string a second time (this happened when we concatenated the authentication information) the result was an evil URL.

Mitigation for this is rather straightforward. If you're ever expecting to receive a string in public interface, first perform a `typeof` check and make sure what you're working with is a string. A more defensive version of our library code would look something like this:

```
function validateAndAuth(url) {
  if (typeof url !== 'string' || !url.startsWith(VALID)) {
    return false
  }

  return url + '?user=123&token=456'
}
```

At Intrinsic, we need to be very careful when application code interacts with the internal Node.js APIs. Many of the internal APIs available in Node.js will exhibit unexpected behavior if they expect a string or number but instead receive an object which can be coerced into one of those primitive values. The above mitigation is just one example of many things we address.

. . .

This article was written by me, Thomas Hunter II. I work at a company called Intrinsic (btw, we're hiring!) where we specialize in writing software for securing Node.js applications. We currently have a product which follows the Least Privilege model for securing applications. Our product proactively protects Node.js applications from attackers, and is surprisingly easy to implement. If you are looking for a way to secure your Node.js applications, give us a shout at hello@intrinsic.com.

Thanks to Intrinsic and Deian Stefan.

JavaScript Es2015 ES6 Nodejs