Validator

发表于 2021-01-15 分类于 Challenge， 2020， 安洵杯， Web
Challenge | 2020 | 安洵杯 | Web | Validator

点击此处获得更好的阅读体验

WriteUp来源

https://xz.aliyun.com/t/8581

题目考点

- *Nodejs代码审计*
- *原型链污染分析*

解题思路

出题的思路来自于XNUCA2020的一道原型链污染题，原题的正解是污染原型链value值为空，但是0ops的师傅在解题的过程中做到了任意原型链污染，这题就是以这个任意原型链污染为基础的。(师傅们在做题的时候应该是可以直接搜到这个payload的)参考原比赛的wp：ooooooldjs

针对任意原型链污染这个点，深入的分析在后面。

题目部分源码：

```
if (req.body.password == "D0g3_Yes!!!"){
    console.log(info.system_open)
    if (info.system_open == "yes"){
        const flag = readFile("/flag")
        return res.status(200).send(flag)
    }else{
        return res.status(400).send("The login is successful, but the system is under test and not open...")
    }
}else{
    return res.status(400).send("Login Fail, Password Wrong!")
}
```

这里只有一个简单的info.system_open的判断，所以我们只需要构造出能够污染info.system_open的payload即可。
最终构造出的payload如下：

```
{"password":"D0g3_Yes!!!", "a": {"__proto__": {"system_open": "yes"}}, "a\"].__proto__[\"system_open": "yes" }
```

express-validator 6.6.0 原型链污染详细分析

测试用例

测试例子:

```
const express = require('express')
const app = express()

const port = 9000

app.use(express.json())
app.use(express.urlencoded({
    extended: true
}))

const {
    body,
    validationResult
} = require('express-validator')

middlewares = [
    body('*').trim() // 对所以键值进行trim处理
]

app.use(middlewares)

app.post("/user", (req, res) => {
    const foo = "hellowrold"
    return res.status(200).send(foo)
})


app.listen(port, () => {
    console.log(`server listening on ${port}`)
})
```

依赖包版本：

```
npm init
npm install lodash@4.17.16
npm install express-validator@6.6.0
npm install express
```

express-validator参考：https://express-validator.github.io/docs/

在分析这个原型链污染漏洞之前，我们先对express-validator的过滤器(sanitizer)的实现流程进行一个分析。

过滤器(sanitizer)实现流程

在src/middlewares/validation-chain-builders.js文件中找到body的实现

传递到了check_1.check方法中，跟入check.js文件

location传递进来后传递到setLocations方法里创建了一个builder对象，并传入到chain_1.SanitizersImpl方法中。对于return，在题目的Wirteup中有以下的描述：

> 先看return的地方，check函数里的middleware就是express-validator最终对接express的中间件。utils_1.bindAll函数做的事情就是把对象原型链上的函数绑定成了对象的一个属性，因为Object.assign只做浅拷贝，utils.bindAll之后Object.assign就可以把sanitizers和validators上面的方法都拷贝到middleware上面了，这样就能通过这个middleware调用所有的验证和过滤函数。

针对bindAll，我个人的理解是：bindAll函数就是把需要调用的方法都绑定到middleware上进而实现链式调用。

- 什么是链式调用：https://juejin.im/post/6844904030221631495
- bindAll方法: https://my.oschina.net/cangy/blog/301038

传入bindAll的参数值是通过Chain_1.SanitizersImpl返回的，可以通过chain.js确定到这个函数的定义位置为src/chain/sanitizers-impl.js。

在这个类中存在很多的过滤器(sanitizer)，过滤器实现的方法都调用了this.addStandardSanitization()将过滤器传入到sanitization_1.Sanitization()方法中，得到的结果最终传递给this.builder.addItem()。

先来看sanitization_1.Sanitization()方法，位置在：src/context-items/sanitization.js：

这个Sanitization类中的run方法最终通过调用sanitizer方法设置了context的值。(context后面的处理过程在漏洞分析部分)

再来看this.builder.addItem()做了什么，位置在src/context-builder.js

就是把传入进来的值压入this.stack栈中。

回到Sanitization类中的run方法，这个run方法是在哪调用的呢？再看到check.js，这里创建了一个runner对象，并在middleware里调用了run方法：

同样可以从chain/index.js中找到实现runner.run方法的具体位置为：

这里可以看到是从context.stack里面循环遍历了contextItem，并调用了其run方法。在这条循环语句处下断点查看一下context的内容：

在stack里面就是包含了我们所调用的过滤器，而这个context.stack也就是this.builder.addItem()所设置的值。

这就是完整的express-validator的过滤器(sanitizer)的实现流程，wp中对这个过程有一个总结：

> express-validator的做法是把各种validator和sanitizers的方法绑定到check函数返回的middleware上，这些validator和sanitizer的方法通过往context.stack属性里面push context-items，最终在ContextRunnerImpl.run()方法里遍历context.stack上面的context-items，逐一调用run方法实现validation或者是sanitization

我这里画了一个流程图来梳理这一过程：

(这个流程图画的比较复杂，如果你尝试跟过一遍的话再来看这个流程图就会比较容易理解一些

在题目环境中`npm install`的时候就会有提示，express-validator库中的所依赖的lodash库存在原型链污染漏洞。

这是因为express-validator的依赖包中，lodash的安装版本最低为4.17.15的，所以在一定条件下会存在原型链污染漏洞。(这里的测试环境我们安装的是4.17.16版本，lodash在4.17.17以下存在原型链污染漏洞)

继续分析：

跟着上面过滤器(sanitizer)实现流程的最后几步，runner.run方法在context.stack里面循环遍历了contextItem，并调用了其run方法。

我们先来看看这个值的传入过程是怎么样的。

请求中值的传入过程

测试数据包：

```
1 {"__proto__[test]": "123 "}
```

在调用run方法时传入了一个instance.value的变量，这个变量的值是我们传入json数据当中的值，run方法在调用过滤器处理后给其赋予了一个新的值。
我们下断点来查看一下：

经过过滤器处理后(也就是经过了一个`trim()`处理)：

可以看到，newvalue是instance.value经过run方法处理后得到的值，一直往上推可以得知instance的实现方法是this.selectFields，位置是在select-fields.js文件中：

select-fields.js:

这个文件的处理过程中我们需要了解到的就是在segments.reduce函数中对输入的值进行了一些判断和替换。重要的点就是当传入的键中存在.，则会在字符两边加上[" "]，并且最终返回的是一个字符串形式的结果。(对于这些语句更为详细的原因可以参考writeup中对这一段的描述)

接着之前的过程，在经过了过滤器的处理之后，会通过lodash.set对指定的path设置新值，也就是如图中的_.set(req[location], path, newValue)过程。

现在可以尝试一下能不能通过lodash.set原型链污染来污染指定的值：

尝试污染proto[test]，结果发现是污染并没有成功：

原因是因为，当lodash.set中第一个参数存在一个与第二个参数同名的键时，污染就会失败，测试如下：

所以，我们就要尝试去绕过这个点。
我们来看一下这个语句：

```
path !== '' ? _.set(req[location], path, newValue) : _.set(req, location, newValue);
```

这里的第一个参数是从请求中直接取出来的，path是经过先前处理后的出来的值。所以能不能通过这个处理来进行绕过呢？当然是可以的。
当我们传入：

```
{""].proto["test":"123 "}
```

这里的键为"].__proto__["test，由于字符里面存在.，所以在segments.reduce函数处理时会对其左右加双引号和中括号，最终变成：[""].__proto__["test"]。这时在调用set函数时，值的情况就为：

这时就不存在同名的键了，于是查看污染的后的值发现：

我们设置的值并没有传递进去，而是污染为了一个空值。为什么传递进来的newValue为空值呢？

从select-fields.js中可以看到，是因为取值时，使用的是lodash.get方法从req['body']中取被处理后的键值，处理后的键是不存在的，所以取出来的值就为undefined。

当undefined传递到Sanitization.run方法中后，经过了一个.toString()的处理变成了"空字符串。

lodash.get方法中读取键值的妙用

那我们还有没有办法污染呢？结果肯定是有的，我们跟入这个lodash.get方法，这个方法的具体实现位置位于：lodash/get.js

继续跟踪到lodash/_baseGet.js

从中我们可以看到这个函数取值的一些逻辑，首先，path经过了castPath处理将字符串形式的路径转为了列表，如下面的内容所示。转换完后通过一个while循环将值循环取出，并在object这个字典里去取出对应的值。

```
// 初始值
['a'].__proto__.['b']

// 转换完后的值
["a","__proto__","b",]
```

那这个地方能不能利用呢？当然也是可以的，我们来看下最终的payload：

```
1 {"a": {"__proto__": {"test": "testvalue"}}, "a\"].__proto__[\"test": 222}
```

这个时候我们在这个函数处下断点就可以看到，`a\"].__proto__[\"test`经过castPath处理变成了["a", "**proto**", "test"]，在Object循环取值最终取到的是`"a": {"__proto__": {"test": 'testvalue'}`中的test键的值，这样就达到了控制value的目的。

还未遍历前：

最后一次遍历：

最终污染成功：

- **本文作者：** CTFHub
- **本文链接：** https://writeup.ctfhub.com/Challenge/2020/安洵杯/Web/9svHGs6PFkEBneTx4XaZ6r.html
- **版权声明：** 本博客所有文章除特别声明外，均采用 BY-NC-SA 许可协议。转载请注明出处！

Challenge # 2020 # Web # 安洵杯
XSS
Web-Bash-Vino0o0o