Node.js Web安全开发规范

背景

Web 应用系统是黑客或恶意攻击者的重要目标，通过一些缺陷，可能直接攻陷系统服务器，进而对内部网络造成进一步危害。所以针对web开发人员，制定了一个编码规范，帮助开发人员部分避免出现类似的安全缺陷。

网络安全漏洞分类

1、跨站脚本攻击(`xss` 攻击)

`xss` (Cross Site Scripting)，跨站脚本攻击。XSS是常见的Web攻击技术之一.所谓的跨站脚本攻击指得是:恶意攻击者往Web页面里注入恶意 `Script` 代码，用户浏览这些网页时，就会执行其中的恶意代码，可对用户进行盗取 `cookie` 信息、会话劫持等各种攻击。

```
xss攻击实例：
<input value='<SCRIPT>alert("xss")</SCRIPT>' />
```

解决办法：

- **对所有的输入、输出进行过滤**

```
npm install validator-js
// 使用 Validator 检验用户的输入信息
const validator = require('validator');
const number1 = '10.5';
const number2 = '10,5';
validator.isDecimal(number1); // true
validator.isDecimal(number2); // false

// 使用白名单校验
const allowedChars = '0123456789.';
const decimal1 = '10.5';
const decimal2 = '10,5';
validator.isWhitelisted(decimal1, allowedChars); // true
validator.isWhitelisted(decimal2, allowedChars); // false

// 长度校验
const addressInput = '123 Main St Anytown, USA';
validator.isLength(string, {min: 3, max: 255}); // true

// trim
const sanitize = require('validator');
sanitize.trim(' asdasd ');

// html转义
const sanitizer = require('validator');
const sampleSourceCode = '<script>alert("Hello World");</script>';
const sanitizedSampleSourceCode = sanitizer.escape(sampleSourceCode);
console.log(sanitizedSampleSourceCode);
// &lt;script&gt;alert(&quot;Hello World&quot;);&lt;&#x2F;script&gt;

// 也可以使用xss-filters
const xssFilters = require('xss-filters');
const unsafeFirstname = req.query.firstname;
xssFilters.inHTMLData(unsafeFirstname);
```

Validator	**Description**
contains(str, seed)	check if the string contains the seed.
equals(str, comparison)	check if the string matches the comparison.
isAfter(str [, date])	check if the string is a date that's after the specified date (defaults to now).
	check if the string contains only letters (a-zA-Z). Locale is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE',

isAlpha(str [, locale])	'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'el-GR', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']) and defaults to en-US. Locale list is validator.isAlphaLocales.
isAlphanumeric(str [, locale])	check if the string contains only letters and numbers. Locale is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'el-GR', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']) and defaults to en-US. Locale list is validator.isAlphanumericLocales.
isAscii(str)	check if the string contains ASCII chars only.
isBase64(str)	check if a string is base64 encoded.
isBoolean(str)	check if a string is a boolean.
isByteLength(str [, options])	check if the string's length (in UTF-8 bytes) falls in a range. options is an object which defaults to {min:0, max: undefined}.
isEmail(str [, options])	check if the string is an email.
isCreditCard(str)	check if the string is a credit card.
isEmail(str [, options])	check if the string is an email.
isFloat(str [, options])	check if the string is a float.
...	...

Sanitizer	Description
blacklist(input, chars)	remove characters that appear in the blacklist. The characters are used in a RegExp and so you will need to escape some chars, e.g. `blacklist(input, '\\[\\]')`.
escape(input)	replace `<`, `>`, `&`, `'`, `"` and `/` with HTML entities.
unescape(input)	replaces HTML encoded entities with `<`, `>`, `&`, `'`, `"` and `/`.
trim(input [, chars])	trim characters (whitespace by default) from both sides of the input.
whitelist(input, chars)	remove characters that do not appear in the whitelist. The characters are used in a RegExp and so you will need to escape some chars, e.g. `whitelist(input, '\\[\\]')`.
...	...

```
tips:
  Web应用安全中，用户的输入及关联的数据如果未做检查将导致安全风险。
  不能依赖于客户端校验，必须使用服务端代码对输入数据进行最终校验。
  对于不可信的数据，输出到客户端前必须先进行 HTML 编码。
```

2、跨站请求伪造(`CSRF` 攻击)

`CSRF` (Cross Site Request Forgery)，即跨站请求伪造，是一种常见的Web攻击。

`CSRF` 原理：CSRF攻击过程的受害者用户登录网站A，输入个人信息，在本地保存服务器生成的cookie。然后在A网站点击由攻击者构建一条恶意链接跳转到 B网站,然后B网站携带着的用户cookie信息去访问B网站.让A网站造成是用户自己访问的假相,从而来进行一些列的操作,常见的就是转账.

解决办法：

- **验证 HTTP `Referer` 字段**

HTTP 标题头是在 HTTP 请求和 HTTP 响应的开始阶段发送的，标题头中的 `referer` 字段包含来自请求源端的 Web 页面的 URL。不过，因为该字段是很容易被伪造的，所以不要根据 `referer` 字段的值做出任何安全决策。

```
禁止将 HTTP 标题头中的任何未加密信息作为安全决策依据。
```

- **在请求地址中添加 `token` 并验证**

`CSRF` 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 `cookie` 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 `cookie` 来通过安全验证。要抵御 `CSRF`，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 `cookie` 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 `token`，并在服务器端建立一个拦截器来验证这个 `token`，如果请求中没有 `token` 或者 `token` 内容不正确，则认为可能是 `CSRF` 攻击而拒绝该请求。

- **在 HTTP 头中自定义属性并验证**

虽然不支持HTTP 标题头中的任何未加密信息作为安全决策依据。但是我们可以在header中添加自定义属性，将加密的authorization放入自定义的header中，通过XMLHttpRequest，给所有的请求加上这个自定义头部信息。后端验证这个header信息判断是否有效。同时，通过 XMLHttpRequest 请求的地址不会被记录到浏览器的地址栏，也不用担心 token 会透过 Referer 泄露到其他网站中去

```
header:
{
    X-Custom-Heade: cookies.get('csrfToken'), // twSipje994j3jQj9QT6NFwe970MjmFkX
}
```

3、SQL注入攻击

SQL注入(SQL Injection)，应用程序在向后台数据库传递SQL(Structured Query Language，结构化查询语言)时，攻击者将SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令

```
1、通常所有的 SQL 语句应使用预编译操作，禁止拼接 SQL 语句。
2、在明确变量数据类型时，进行变量数量类型检查，确保是预定变量类型。
3、使用预编译模式访问数据库。
4、应用程序连接数据库服务器的数据库帐号，在满足业务需求的前提下，必须使用最低级别权限的数据库帐号
5、禁止在代码中存储如数据库连接字符串、口令和密钥之类的敏感数据，这样容易导致泄密。
```

错误示例：

```
const express = require('express');
const db = require('./db');
const router = express.Router();
router.get('/email', (req, res) => {
    db.query('SELECT email FROM users WHERE id = ' + req.query.id); .then((record)
 => {
    // do stuff res.send(record[0]); })
});
```

正确示例：

```
// 安全访问 Mysql 的代码示例
const express = require('express');
const db = require('./db');
const router = express.Router();
router.get('/email', (req, res) => {
    db.query('SELECT email FROM users WHERE id = $1', req.query.id).then((record)
=> {
        // do stuff
        res.send(record[0]);
    })
});

// 访问 postgresql 的代码示例
// evilUserData parameter is an input given by a user
// For the example it could be someting like that: or 1=1
const sql = 'SELECT * FROM users WHERE id = $1';
client.query(sql, evilUserData, (error, results, fields) => {
    if (error) {
        throw error;
    }
    // ...
});

// 安全访问 Mongodb 的代码示例

const dbQuery = {
    $where: 'this.UserID = new Number(' + req.query.id + ')'
};
db.Users.find(dbQuery);
```

使用配置文件存放数据库连接信息

```
{
    "db": {
        "user": "f00",
        "pass": "f00?bar#ItsP0ssible", "host": "localhost",
        "port": "3306"
    }
}
```

4、文件上传漏洞

该漏洞允许用户上传任意文件可能会让攻击者注入危险内容或恶意代码，并在服务器上运行。

原理：由于文件上传功能实现代码没有严格限制用户上传的文件后缀以及文件类型，导致允许攻击者向某

个可通过 Web 访问的目录上传任意PHP文件，并能够将这些文件传递给 PHP 解释器，就可以在远程服务器上执行任意PHP脚本。

- **文件上传规范：**

```
1、建议基于京东云存储(JFS)实现文件上传，确保文件存储的安全性
2、禁止将文件(含临时缓存文件)存储在 Web 应用程序目录
3、必须根据实际业务需求使用白名单策略限制上传文件后缀名，例如业务仅需上传图片时，只允许用户上传诸如 jpg、png、gif 等图片类型文件。
4、禁止上传 html、htm、swf 等可被浏览器解析的文件。
5、禁止将用户可控的内容上传至京东域名，避免用户上传的恶意文件，导致该京东域名被反病毒厂商拉黑，导致业 务不可用。
```

京东云对象存储文档: http://jpcloud.jd.com/pages/viewpage.action?pageId=10658955;

京东云图片存储文档: http://storage.jd.com/doc/jd-image.pdf;

- **文件下载规范：**

```
1、优先使用云储存下载功能
2、预先定义路径，禁止用户更改文件下载目录
3、对下载路径进行中含有"./"、"../"、".\"、"..\"等符号时，直接拒绝该请求
4、对下载文件后缀名进行白名单限制
```

京东云存储文档: http://jpcloud.jd.com/pages/viewpage.action?pageId=10658955;