JavaScript编码安全指南 V1.0

JavaScript页面类

I. 代码实现

1.1 原生DOM API的安全操作

1.1.1【必须】HTML标签操作，限定/过滤传入变量值

- 使用`innerHTML=`、`outerHTML=`、`document.write()`、`document.writeln()`时，如变量值外部可控，应对特殊字符（`&`，`<`，`>`，`"`，`'`）做编码转义，或使用安全的DOM API替代，包括：`innerText=`

```
// 假设 params 为用户输入， text 为 DOM 节点
// bad：将不可信内容带入HTML标签操作
const { user } = params;
// ...
text.innerHTML = `Follow @${user}`;
```

```
// good: innerHTML操作前，对特殊字符编码转义
function htmlEncode(iStr) {
    let sStr = iStr;
    sStr = sStr.replace(/&/g, "&amp;");
    sStr = sStr.replace(/>/g, "&gt;");
    sStr = sStr.replace(/</g, "&lt;");
    sStr = sStr.replace(/"/g, "&quot;");
    sStr = sStr.replace(/'/g, "&#39;");
    return sStr;
}

const { user } = params;
user = htmlEncode(user);
// ...
text.innerHTML = `Follow @${user}`;

// good: 使用安全的DOM API替代innerHTML
const { user } = params;
// ...
text.innerText = `Follow @${user}`;
```

1.1.2【必须】HTML属性操作，限定/过滤传入变量值

- 使用`element.setAttribute(name, value);`时，如第一个参数值`name`外部可控，应用白名单限定允许操作的属性范围。
- 使用`element.setAttribute(name, value);`时，操作`a.href`、`ifame.src`、`form.action`、`embed.src`、`object.data`、`link.href`属性时，如第二个参数值`value`外部可控，应参考*JavaScript页面类规范1.3.1*部分，限定页面重定向或引入资源的目标地址。

```
// good: setAttribute操作前，限定引入资源的目标地址
function addExternalCss(e) {
    const t = document.createElement('link');
    t.setAttribute('href', e),
    t.setAttribute('rel', 'stylesheet'),
    t.setAttribute('type', 'text/css'),
    document.head.appendChild(t)
}

function vaildURL(sUrl) {
return !!((/^(https?:\/\/)?[\w\-.]+\.(qq|tencent)\.com($|\/|\\)/i).test(sUrl) ||
(/^[\w][\w/.\-_%]+$/i).test(sUrl) || (/^[/\\][^/\\]/i).test(sUrl));
}

let sUrl = "https://evil.com/1.css"
if (vaildURL(sUrl)) {
    addExternalCss(sUrl);
}
```

1.2 流行框架/库的安全操作

1.2.1【必须】限定/过滤传入jQuery不安全函数的变量值

- 使用`.html()`、`.append()`、`.prepend()`、`.wrap()`、`$()`、`.replaceWith()`、`.wrapAll()`、`.wrapInner()`、`.after()`、`.before()`时，如变

量值外部可控，应对特殊字符（`&`, `<`, `>`, `"`, `'`）做编码转义。

```
// bad：将不可信内容，带入jQuery不安全函数.after()操作
const { user } = params;
// ...
$("p").after(user);

// good: jQuery不安全函数.html()操作前，对特殊字符编码转义
function htmlEncode(iStr) {
    let sStr = iStr;
    sStr = sStr.replace(/&/g, "&amp;");
    sStr = sStr.replace(/>/g, "&gt;");
    sStr = sStr.replace(/</g, "&lt;");
    sStr = sStr.replace(/"/g, "&quot;");
    sStr = sStr.replace(/'/g, "&#39;");
    return sStr;
}

// const user = params.user;
user = htmlEncode(user);
// ...
$("p").html(user);
```

- 使用`.attr()`操作`a.href`、`iframe.src`、`form.action`、`embed.src`、`object.data`、`link.href`属性时，应参考*JavaScript页面类规范1.3.1*部分，限定重定向的资源目标地址。
- 使用`.attr(attributeName, value)`时，如第一个参数值`attributeName`外部可控，应用白名单限定允许操作的属性范围。
- 使用`$.getScript(url [, success ])`时，如第一个参数值`url`外部可控（如：从URL取值拼接，请求jsonp接口），应限定可控变量值的字符集范围为：`[a-zA-Z0-9_-]+`。

1.2.2【必须】限定/过滤传入Vue.js不安全函数的变量值

- 使用`v-html`时，不允许对用户提供的内容使用HTML插值。如业务需要，应先对不可信内容做富文本过滤。
- 使用`v-bind`操作`a.href`、`iframe.src`、`form.action`、`embed.src`、`object.data`、`link.href`时，应确保后端已参考*JavaScript页面类规范1.3.1*部分，限定了供前端调用的重定向目标地址。
- 使用`v-bind`操作`style`属性时，应只允许外部控制特定、可控的CSS属性值

```
// bad：v-bind允许外部可控值，自定义CSS属性及数值
<a v-bind:href="sanitizedUrl" v-bind:style="userProvidedStyles">
click me
</a>

// good：v-bind只允许外部提供特性、可控的CSS属性值
<a v-bind:href="sanitizedUrl" v-bind:style="{
color: userProvidedColor,
background: userProvidedBackground
}" >
click me
</a>
```

1.3 页面重定向

1.3.1【必须】限定跳转目标地址

- 使用白名单，限定重定向地址的协议前缀（默认只允许HTTP、HTTPS）、域名（默认只允许公司根域），或指定为固定值；
- 适用场景包括，使用函数方法：`location.href`、`window.open()`、`location.assign()`、`location.replace()`；赋值或更新HTML属性：`iframe.src`、`form.action`、`a.href`、`embed.src`、`object.data`；

```
// bad: 跳转至外部可控的不可信地址
const sTargetUrl = getURLParam("target");
location.replace(sTargetUrl);

// good: 白名单限定重定向地址
function vaildURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.
(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) ||
(/^[/\\][^/\\]/i).test(sUrl));
}

const sTargetUrl = getURLParam("target");
if (vaildURL(sTargetUrl)) {
    location.replace(sTargetUrl);
}

// good: 制定重定向地址为固定值
const sTargetUrl = "http://www.qq.com";
location.replace(sTargetUrl);
```

1.4 JSON解析/动态执行

1.4.1【必须】使用安全的JSON解析方式

- 应使用`JSON.parse()`解析JSON字符串。低版本浏览器，应使用安全的Polyfill封装

```
// bad: 直接调用eval解析json
const sUserInput = getURLParam("json_val");
const jsonstr1 = `{"name":"a","company":"b","value":"${sUserInput}"}`;
const json1 = eval(`(${jsonstr1})`);

// good: 使用JSON.parse解析
const sUserInput = getURLParam("json_val");
JSON.parse(sUserInput, (k, v) => {
    if (k === "") return v;
    return v * 2;
});

// good: 低版本浏览器，使用安全的Polyfill封装（基于eval）
<script src="https://github.com/douglascrockford/JSON-js/blob/master/json2.js">
</script>;
const sUserInput = getURLParam("json_val");
JSON.parse(sUserInput);
```

1.5 跨域通讯

1.5.1【必须】使用安全的前端跨域通信方式

- 禁止通过`document.domain`降域，前端跨域通讯，应使用postMessage替代。

1.5.2【必须】使用postMessage应限定Origin

- 在message事件监听回调中，应先使用`event.origin`校验来源，再执行具体操作。
- 校验来源时，应使用`===`判断，禁止使用`indexOf()`

```
// bad: 使用indexOf校验Origin值
window.addEventListener("message", (e) => {
    if (~e.origin.indexOf("https://a.qq.com")) {
    // ...
    } else {
    // ...
    }
});

// good: 使用postMessage时，限定Origin，且使用===判断
window.addEventListener("message", (e) => {
    if (e.origin === "https://a.qq.com") {
    // ...
    }
});
```

II. 配置&环境

2.1 敏感/配置信息

2.1.1【必须】禁止明文硬编码AK/SK

- 禁止前端页面的JS明文硬编码AK/SK类密钥，应封装成后台接口，AK/SK保存在后端配置中心或密钥管理系统

2.2 第三方组件/资源

2.2.1【必须】使用可信范围内的统计组件

2.2.2 【必须】禁止引入非可信来源的第三方JS

- 应检查页面内引入第三方JS资源是否可控。

2.3 纵深安全防护

2.3.1【推荐】部署CSP，并启用严格模式

- 应部署CSP，并在规则中应启用最新的严格模式特性`nonce-`

Node.js后台类

I. 代码实现

1.1 输入验证

1.1.1【必须】按类型进行数据校验

- 所有程序外部输入的参数值，应进行数据校验。校验内容包括但不限于：数据长度、数据范围、数据类型与格式。校验不通过，应拒绝。
- 推荐使用组件：validator

```
// bad：未进行输入验证
Router.get("/vulxss", (req, res) => {
    const { txt } = req.query;
    res.set("Content-Type", "text/html");
    res.send({
        data: txt,
    });
});

// good：按数据类型，进行输入验证
const Router = require("express").Router();
const validator = require("validator");

Router.get("/email_with_validator", (req, res) => {
    const txt = req.query.txt || "";
    if (validator.isEmail(txt)) {
        res.send({
            data: txt,
        });
    } else {
        res.send({ err: 1 });
    }
});
```

1.2 执行命令

1.2.1 【必须】使用child_process执行系统命令，应限定或校验命令和参数的内容

- 适用场景包括：`child_process.exec`，`child_process.execSync`，`child_process.spawn`，`child_process.spawnSync`，`child_process.execFile`，`child_process.execFileSync`
- 调用上述函数，应首先考虑限定范围，供用户选择。
- 使用`child_process.exec`或`child_process.execSync`时，如果可枚举输入的参数内容或者格式，则应限定白名单。如果无法枚举命令或参数，则必须过滤或者转义指定符号，包括：`|;&$()><`!`
- 使用`child_process.spawn`或`child_process.execFile`时，应校验传入的命令和参数在可控列表内。

```
const Router = require("express").Router();
const validator = require("validator");
const { exec } = require('child_process');
```

```
// bad：未限定或过滤，直接执行命令
Router.get("/vul_cmd_inject", (req, res) => {
    const txt = req.query.txt || "echo 1";
    exec(txt, (err, stdout, stderr) => {
        if (err) { res.send({ err: 1 }) }
        res.send({stdout, stderr});
    });
});

// good：通过白名单，限定外部可执行命令范围
Router.get("/not_vul_cmd_inject", (req, res) => {
    const txt = req.query.txt || "echo 1";
  const phone = req.query.phone || "";
    const cmdList = {
        sendmsg: "./sendmsg "
    };
    if (txt in cmdList && validator.isMobilePhone(phone)) {
        exec(cmdList[txt] + phone, (err, stdout, stderr) => {
          if (err) { res.send({ err: 1 }) };
          res.send({stdout, stderr});
        });
    } else {
        res.send({
            err: 1,
            tips: `you can use '${Object.keys(cmdList)}'`,
        });
    }
});

// good：执行命令前，过滤/转义指定符号
Router.get("/not_vul_cmd_inject", (req, res) => {
    const txt = req.query.txt || "echo 1";
  let phone = req.query.phone || "";
    const cmdList = {
        sendmsg: "./sendmsg "
    };
    phone = phone.replace(/(\||;|&|\$\(|\(|\)|>|<|\`|!)/gi,"");
    if (txt in cmdList) {
        exec(cmdList[txt] + phone, (err, stdout, stderr) => {
          if (err) { res.send({ err: 1 }) };
          res.send({stdout, stderr});
        });
    } else {
        res.send({
            err: 1,
            tips: `you can use '${Object.keys(cmdList)}'`,
        });
    }
});
```

1.3 文件操作

1.3.1 【必须】限定文件操作的后缀范围

- 按业务需求，使用白名单限定后缀范围。

1.3.2 【必须】校验并限定文件路径范围

- 应固定上传、访问文件的路径。若需要拼接外部可控变量值，检查是否包含`..`、`.`路径穿越字符。如存在，应拒绝。

```
const fs = require("fs");
const path = require("path");
let { filename } = req.query;

// bad：未检查文件名/路径
fs.readFile(filename, (err, data) => {
    if (err) {
        return console.error(err);
    }
    console.log(`异步读取: ${data.toString()}`);
});

// good：检查了文件名/路径，是否包含路径穿越字符
filename = path.normalize(filename);
if (filename.indexOf("..") < 0) {
    fs.readFile(filename, (err, data) => {
        if (err) {
            return console.error(err);
        }
        console.log(data.toString());
    });
};
```

1.3.3 【必须】安全地处理上传文件名

- 将上传文件重命名为16位以上的随机字符串保存。
- 如需原样保留文件名，应检查是否包含`..`、`.`路径穿越字符。如存在，应拒绝。

1.3.4 【必须】敏感资源文件，应有加密、鉴权和水印等加固措施

- 用户上传的`身份证`、`银行卡`等图片，属敏感资源文件，应采取安全加固。
- 指向此类文件的URL，应保证不可预测性；同时，确保无接口会批量展示此类资源的URL。
- 访问敏感资源文件时，应进行权限控制。默认情况下，仅用户可查看、操作自身敏感资源文件。
- 图片类文件应添加业务水印，表明该图片仅可用于当前业务使用。

1.4 网络请求

1.4.1 【必须】限定访问网络资源地址范围

- 应固定程序访问网络资源地址的`协议`、`域名`、`路径`范围。
- 若业务需要，外部可指定访问网络资源地址，应禁止访问内网私有地址段及域名。

1.4.2 【推荐】请求网络资源，应加密传输

- 应优先选用https协议请求网络资源

1.5 数据输出

1.5.1 【必须】高敏感信息禁止存储、展示

- 口令、密保答案、生理标识等鉴权信息禁止展示

- 非金融类业务，信用卡cvv码及日志禁止存储

1.5.2【必须】一般敏感信息脱敏展示

- 身份证只显示第一位和最后一位字符，如：`3******************1`
- 移动电话号码隐藏中间6位字符，如：`134**************48`
- 工作地址/家庭地址最多显示到`区`一级
- 银行卡号仅显示最后4位字符，如：`********************8639`

1.5.3 【推荐】返回的字段按业务需要输出

- 按需输出，避免不必要的用户信息泄露
- 用户敏感数据应在服务器后台处理后输出，不可以先输出到客户端，再通过客户端代码来处理展示

1.6 响应输出

1.6.1 【必须】设置正确的HTTP响应包类型

- 响应头Content-Type与实际响应内容，应保持一致。如：API响应数据类型是json，则响应头使用`application/json`；若为xml，则设置为`text/xml`。

1.6.2 【必须】添加安全响应头

- 所有接口、页面，添加响应头`X-Content-Type-Options: nosniff`。
- 所有接口、页面，添加响应头`X-Frame-Options`。按需合理设置其允许范围，包括：`DENY`、`SAMEORIGIN`、`ALLOW-FROM origin`。用法参考：MDN文档
- 推荐使用组件： helmet

1.6.3 【必须】外部输入拼接到响应页面前，进行编码处理

场景	编码规则
输出点在HTML标签之间	需要对以下6个特殊字符进行HTML实体编码(&, <, >, ", ',/)。 示例： & --> & < --> < >--> > " --> " ' --> ' / --> /
输出点在HTML标签普通属性内（如href、src、style等，on事件除外）	要对数据进行HTML属性编码。 编码规则：除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为&#xHH;(以&#x开头，HH则是指该字符对应的十六进制数字，分号作为结束符)
输出点在JS内的数据中	需要进行js编码 编码规则： 除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为 \xHH （以 \x 开头，HH则是指该字符对应的十六进制数字） Tips：这种场景仅限于外部数据拼接在js里被引号括起来的变量值中。除此之外禁止直接将代码拼接在js代码中。
输出点在CSS中（Style属性）	需要进行CSS编码 编码规则： 除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为 \HH （以 \ 开头，HH则是指该字符对应的十六进制数字）
输出点在URL属性中	对这些数据进行URL编码 Tips：除此之外，所有链接类属性应该校验其协议。禁止JavaScript、data和Vb伪协议。

1.6.4 【必须】响应禁止展示物理资源、程序内部代码逻辑等敏感信息

- 业务生产（正式）环境，应用异常时，响应内容禁止展示敏感信息。包括但不限于：`物理路径`、`程序内部源代码`、`调试日志`、`内部账号名`、`内网ip地址`等。

```
// bad
Access denied for user 'xxx'@'xx.xxx.xxx.162' (using password: NO)"
```

1.6.5 【推荐】添加安全纵深防御措施

- 应部署CSP，并在规则中启用最新的严格模式特性`nonce-`

```
// good：使用helmet组件安全地配置响应头
const express = require("express");
const helmet = require("helmet");
const app = express();
app.use(helmet());

// good：正确配置Content-Type、添加了安全响应头，引入了CSP
Router.get("/", (req, res) => {
    res.header("Content-Type", "application/json");
    res.header("X-Content-Type-Options", "nosniff");
    res.header("X-Frame-Options", "SAMEORIGIN");
    res.header("Content-Security-Policy", "script-src 'self'");
});
```

1.7 执行代码

1.7.1 【必须】安全的代码执行方式

- 禁止使用 `eval` 函数
- 禁止使用 `new Function("input")()` 来创建函数
- 使用 `setInteval`，`setTimeout`，应校验传入的参数

1.8 Web跨域

1.8.1 【必须】限定JSONP接口的callback字符集范围

- JSONP接口的callback函数名为固定白名单。如callback函数名可用户自定义，应限制函数名仅包含 字母、数字和下划线。如：`[a-zA-Z0-9_-]+`

1.8.2 【必须】安全的CORS配置

- 使用CORS，应对请求头Origin值做严格过滤、校验。具体来说，可以使用“全等于”判断，或使用严格的正则进行判断。如：`^https://domain\.qq\.com$`

```
// good：使用全等于，校验请求的Origin
if (req.headers.origin === 'https://domain.qq.com') {
    res.setHeader('Access-Control-Allow-Origin', req.headers.origin);
    res.setHeader('Access-Control-Allow-Credentials', true);
}
```

1.9 SQL操作

1.9.1 【必须】SQL语句默认使用预编译并绑定变量

- 应使用预编译绑定变量的形式编写sql语句，保持查询语句和数据相分离

```
// bad：拼接SQL语句查询，存在安全风险
const mysql  = require("mysql");
const connection = mysql.createConnection(options);
connection.connect();
```

```
const sql = util.format("SELECT * from some_table WHERE Id = %s and Name =
%s", req.body.id, req.body.name);
connection.query(sql, (err, result) => {
    // handle err..
});

// good：使用预编译绑定变量构造SQL语句
const mysql  = require("mysql");
const connection = mysql.createConnection(options);
connection.connect();

const sql = "SELECT * from some_table WHERE Id = ? and Name = ?";
const sqlParams = [req.body.id, req.body.name];
connection.query(sql, sqlParams, (err, result) => {
    // handle err..
});
```

- 对于表名、列名等无法进行预编译的场景，如：`__user_input__` 拼接到比如 `limit`, `order by`, `group by` , `from tablename` 语句中。请使用以下方法：

  方案1：使用白名单校验表名/列名

```
// good
const tableSuffix = req.body.type;
if (!["expected1", "expected2"].indexOf(tableSuffix)) {
    // 不在表名白名单中，拒绝请求
    return ;
}
const sql = `SELECT * from t_business_${tableSuffix}`;
connection.query(sql, (err, result) => {
    // handle err..
});
```

  方案2：使用反引号包裹表名/列名，并过滤 `__user_input__` 中的反引号

```
// good
let { orderType } = req.body;
// 过滤掉__user_input__中的反引号
orderType = orderType.replace("`", "");
const sql = util.format("SELECT * from t_business_feeds order by `%s`",
orderType);
connection.query(sql, (err, result) => {
    // handle err..
});
```

  方案3：将 `__user_input__` 转换为整数

```
// good
let { orderType } = req.body;
// 强制转换为整数
orderType = parseInt(orderType, 10);
const sql = `SELECT * from t_business_feeds order by ${orderType}`;
connection.query(sql, (err, result) => {
    // handle err..
});
```

1.9.2 【必须】安全的ORM操作

- 使用安全的ORM组件进行数据库操作。如 `sequelize` 等
- 禁止`__user_input__`以拼接的方式直接传入ORM的各类raw方法

```
// bad: adonisjs ORM
// 参考：https://adonisjs.com/docs/3.2/security-introduction#_sql_injection
const username = request.param("username");
const users = yield Database
  .table("users")
  .where(Database.raw(`username = ${username}`));

// good: adonisjs ORM
const username = request.param("username");
const users = yield Database
  .table('users')
  .where(Database.raw("username = ?", [username]));
```

- 使用ORM进行Update/Insert操作时，应限制操作字段范围

```
/*
good
假设该api用于插入用户的基本信息，使用传入的req.body通过Sequelize的create方法实现
假设User包含字段：username,email,isAdmin,
其中,isAdmin将会用于是否系统管理员的鉴权，默认值为false
*/
// Sequelize: 只允许变更username、email字段值
User.create(req.body, { fields: ["username", "email"] }).then((user) => {
    // handle the rest..
});
```

> **为什么要这么做?**
> 在上述案例中，若不限定fields值，攻击者将可传入`{"username":"boo","email":"foo@boo.com","isAdmin":true}`将自己变为`Admin`，产生垂直越权漏洞。

1.10 NoSQL操作

1.10.1 【必须】校验参数值类型

- 将HTTP参数值代入NoSQL操作前，应校验类型。如非功能需要，禁止对象（Object）类型传入。

```
// bad：执行NOSQL操作前，未作任何判断
app.post("/", (req, res) => {
    db.users.find({ username: req.body.username, password: req.body.password },
(err, users) => {
    // **TODO:** handle the rest
    });
});

// good：在进入nosql前先判断`__USER_INPUT__`是否为字符串。
app.post("/", (req, res) => {
    if (req.body.username && typeof req.body.username !== "string") {
        return new Error("username must be a string");
```

```
    }
    if (req.body.password && typeof req.body.password !== "string") {
        return new Error("password must be a string");
    }
    db.users.find({ username: req.body.username, password: req.body.password },
(err, users) => {
        // **TODO:** handle the rest
    });
});
```

> **为什么要这么做?**
>
> JavaScript中，从http或socket接收的数据可能不是单纯的字符串，而是被黑客精心构造的对象(Object)。在本例中：
>
> - 期望接收的POST数据：`username=foo&password=bar`
> - 期望的等价条件查询sql语句：`select * from users where username = 'foo' and password = 'bar'`
> - 黑客的精心构造的攻击POST数据：`username[$ne]=null&password[$ne]=null`或JSON格式：`{"username": {"$ne": null},"password": {"$ne": null}}`
> - 黑客篡改后的等价条件查询sql语句：`select * from users where username != null & password != null`
> - 黑客攻击结果：绕过正常逻辑，在不知道他人的username/password的情况登录他人账号。

1.10.2 【必须】NoSQL操作前，应校验权限/角色

- 执行NoSQL增、删、改、查逻辑前，应校验权限

```
// 使用express、mongodb(mongoose)实现的删除文章demo
// bad：在删除文章前未做权限校验
app.post("/deleteArticle", (req, res) => {
    db.articles.deleteOne({ article_id: req.body.article_id }, (err, users) => {
        // TODO: handle the rest
    });
});

// good：进入nosql语句前先进行权限校验
app.post("/deleteArticle", (req, res) => {
    checkPriviledge(ctx.uin, req.body.article_id);
    db.articles.deleteOne({ article_id: req.body.article_id }, (err, users) => {
        // TODO: handle the rest
    });
});
```

1.11 服务器端渲染（SSR）

1.11.1 【必须】安全的Vue服务器端渲染(Vue SSR)

- 禁止直接将不受信的外部内容传入`{{{ data }}}`表达式中
- 模板内容禁止被污染

```
// bad: 将用户输入替换进模板
const app = new Vue({
    template: appTemplate.replace("word", __USER_INPUT__),
});
renderer.renderToString(app);
```

- 对已渲染的HTML文本内容（renderToString后的html内容）。如需再拼不受信的外部输入，应先进行安全过滤，具体请参考**1.6.3**

```
// bad: 渲染后的html再拼接不受信的外部输入
return new Promise(((resolve) => {
    renderer.renderToString(component, (err, html) => {
        let htmlOutput = html;
        htmlOutput += `${__USER_INPUT__}`;
        resolve(htmlOutput);
    });
}));
```

1.11.2 【必须】安全地使用EJS、LoDash、UnderScore进行服务器端渲染

- 使用render函数时，模板内容禁止被污染

  lodash.Template:

  ```
  // bad: 将用户输入送进模板
  const compiled = _.template(`<b>${__USER_INPUT__}<%- value %></b>`);
  compiled({ value: "hello" });
  ```

  ejs:

  ```
  // bad: 将用户输入送进模板
  const ejs = require("ejs");
  const people = ["geddy", "neil", "alex"];
  const html = ejs.render(`<%= people.join(", "); %>${__USER_INPUT__}`, {
  people });
  ```

- Ejs、LoDash、UnderScore提供的HTML插值模板默认形似`<%= data %>`，尽管在默认情况下`<%= data %>`存在过滤，在编写HTML插值模板时需注意:

  1. 用户输入流入html属性值时，必须使用双引号包裹：`<base data-id = "<%= __USER_INPUT__ %>">`
  2. 用户输入流入`<script></script>`标签或on*的html属性中时，如`<script>var id = <%= __USER_INPUT__ %></script>`，须按照1.6.3中的做法或白名单方法进行过滤，框架/组件的过滤在此处不起作用

1.11.3 【必须】在自行实现状态存储容器并将其JSON.Stringify序列化后注入到HTML时，必须进行安全过滤

1.12 URL跳转

1.12.1【必须】限定跳转目标地址

- 适用场景包括:

1. 使用30x返回码并在Header中设置Location进行跳转

2. 在返回页面中打印`<script>location.href=__Redirection_URL__</script>`

- 使用白名单，限定重定向地址的协议前缀（默认只允许HTTP、HTTPS）、域名（默认只允许公司根域），或指定为固定值；

```
// 使用express实现的登录成功后的回调跳转页面

// bad: 未校验页面重定向地址
app.get("/login", (req, res) => {
    // 若未登录用户访问其他页面，则让用户导向到该处理函数进行登录
  // 使用参数loginCallbackUrl记录先前尝试访问的url，在登录成功后跳转回loginCallbackUrl:
    const { loginCallbackUrl } = req.query;
    if (loginCallbackUrl) {
        res.redirect(loginCallbackUrl);
    }
});

// good: 白名单限定重定向地址
function isVaildURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.
(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) ||
(/^[/\\][^/\\]/i).test(sUrl));
}
app.get("/login", (req, res) => {
    // 若未登录用户访问其他页面，则让用户导向到该处理函数进行登录
  // 使用参数loginCallbackUrl记录先前尝试访问的url，在登录成功后跳转回loginCallbackUrl:
    const { loginCallbackUrl } = req.query;
    if (loginCallbackUrl && isValidUrl(loginCallbackUrl)) {
        res.redirect(loginCallbackUrl);
    }
});

// good: 白名单限定重定向地址，通过返回html实现
function isVaildURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.
(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) ||
(/^[/\\][^/\\]/i).test(sUrl));
}
app.get("/login", (req, res) => {
    // 若未登录用户访问其他页面，则让用户导向到该处理函数进行登录
  // 使用参数loginCallbackUrl记录先前尝试访问的url，在登录成功后跳转回loginCallbackUrl:
    const { loginCallbackUrl } = req.query;
    if (loginCallbackUrl && isValidUrl(loginCallbackUrl)) {
        // 使用encodeURI，过滤左右尖括号与双引号，防止逃逸出包裹的双引号
        const redirectHtml = `<script>location.href =
"${encodeURI(loginCallbackUrl)}";</script>`;
        res.end(redirectHtml);
    }
});
```

1.13 Cookie与登录态

1.13.1【推荐】为Cookies中存储的关键登录态信息添加http-only保护

II. 配置&环境

2.1 依赖库

2.1.1【必须】使用安全的依赖库

- 使用自动工具，检查依赖库是否存在后门/漏洞，保持最新版本

2.2 运行环境

2.2.1 【必须】使用非root用户运行Node.js

2.3 配置信息

2.3.1【必须】禁止硬编码认证凭证

- 禁止在源码中硬编码 `AK/SK`、`数据库账密`、`私钥证书` 等配置信息
- 应使用配置系统或KMS密钥管理系统。

2.3.2【必须】禁止硬编码IP配置

- 禁止在源码中硬编码 `IP` 信息

> **为什么要这么做?**
>
> 硬编码IP可能会导致后续机器裁撤或变更时产生额外的工作量，影响系统的可靠性。

2.3.3【必须】禁止硬编码员工敏感信息

- 禁止在源代码中含员工敏感信息，包括但不限于：`员工ID`、`手机号`、`微信/QQ号` 等。

© 腾讯安全应急响应中心

版本	贡献者
V1.0 (最后修订：2020年10月21日)	martin、monsoon、l0u1s