Fusion.js Security Features

https://fusionjs.com/docs/references/security 没什么信息

```
https://fusionjs.com/docs/
https://fusionjs.com/api/
```

Headers

```
X-Csrf-Token - CSRF令牌，用于防止跨站请求伪造攻击。默认值为"x"。
X-Uber-Edge-Botdefense - Uber自定义的机器人防护头，包含了一串加密哈希值，用于识别和防止自动化攻击。
X-Frame-Options: SAMEORIGIN - 防止点击劫持攻击，只允许同源页面进行框架嵌入。
Strict-Transport-Security: max-age=31536000 - HSTS策略，强制客户端在指定时间内(一年)只使用HTTPS连接。
X-Content-Type-Options: nosniff - 防止浏览器进行MIME类型嗅探，降低内容类型混淆攻击风险。
X-Xss-Protection: 1; mode=block - 启用浏览器内置的XSS防护机制，检测到攻击时阻止页面加载。
Access-Control-Allow-Origin: * - CORS策略配置，当前允许所有域名访问


MIME 类型嗅探

浏览器有一个默认行为叫"MIME 类型嗅探"：即使服务器声明了文件的 Content-Type，浏览器也会尝试通过分析文件内容来"猜测"文件的实际类型
例如：即使服务器说一个文件是 text/plain，但如果文件内容看起来像 HTML，浏览器可能会选择将其作为 HTML 来执行
潜在安全风险：

内容类型混淆攻击：
攻击者上传一个看似图片或文本的文件(例如 .jpg 或 .txt)
文件实际包含恶意的 JavaScript 代码
如果允许 MIME 嗅探，浏览器可能会将其识别为脚本并执行
XSS 攻击变种：
攻击者在图片文件中嵌入 HTML/JavaScript 代码
当浏览器嗅探到 HTML 内容时会执行这些代码
X-Content-Type-Options: nosniff 的作用：

明确告诉浏览器：必须严格遵守服务器声明的 Content-Type
```

```
禁止浏览器进行 MIME 类型嗅探
如果 Content-Type 与实际内容不匹配，浏览器会拒绝加载该资源
```

❗ JWT （fusion-plugin-jwt）

https://fusionjs.com/api/fusion-plugin-jwt 用法

```
/fusion-plugin-jwt/src/jwt-server.ts

JWT放在Cookie里了，可以通过 SessionCookieNameToken 自定义
export const DEFAULT_COOKIE_NAME = 'fusion-sess'; //默认名称



签名验证
  async loadToken() {
    if (this.token == null) {
      const verify = promisify(jwt.verify.bind(jwt));
      this.token = this.cookie
        ? await verify(this.cookie, this.config.secret).catch(() => ({}))
        : {};
    }
    return this.token;
  }


    middleware: (deps, service) => {
      const {secret, cookieName, expires = 86400} = deps; // 86400秒 = 24小时
      return async function jwtMiddleware(
        ctx: Context,
        next: () => Promise<void>
      ) {
        const sign = promisify(jwt.sign.bind(jwt)); // Promise化 jwt.sign
        const session = service.from(ctx);
        // $FlowFixMe
        const token = await session.loadToken(); // 加载并验证当前token
        await next();
        if (token) {
          delete token.exp; // Clear previous exp time and instead use `expiresIn` option below
          const time = Date.now(); // get time *before* async signing
          const signed = await sign(token, secret, {
```

```
            expiresIn: expires, // 设置新的过期时间
          });
          if (signed !== session.cookie) { // 只在 token 变化时更新 cookie
            const msExpires = new Date(time + expires * 1000);
            // TODO(#3) provide way to not set cookie if not needed yet
            ctx.cookies.set(cookieName, signed, {expires: msExpires});
          }
        }
      };
    },


    JWT的风险点主要是签名验证和密钥管理问题，参考：
    https://mp.weixin.qq.com/s/a_VDAhoTMQ54XntJ0bRV9Q
```

❗ Cookie (fusion--Koa--cookies)

```
fusion-core/src/types.ts 中，Cookie 相关的类型定义实际上继承的Koa ，
而Koa又是调用Cookies模块


fusion-core/src/types.ts
import type {Context as KoaContext} from 'koa';

// Context 继承自 Koa 的 Context
export type Context = KoaContext & {
  // ... 其他扩展
};
```

```
Koa 的代码在 node_modules/koa 目录下

koa/
├── lib/
│   ├── application.js     // Koa 应用核心
│   ├── context.js         // 上下文对象
│   ├── request.js         // 请求对象
│   └── response.js        // 响应对象
└── package.json

```

context.js

```
const delegate = require('delegates');
const Cookies = require('cookies');

const COOKIES = Symbol('context#cookies');

const proto = module.exports = {
  get cookies() {
    if (!this[COOKIES]) {
      this[COOKIES] = new Cookies(this.req, this.res, {
        keys: this.app.keys,
        secure: this.request.secure
      });
    }
    return this[COOKIES];
  },

  set cookies(_cookies) {
    this[COOKIES] = _cookies;
  }
};
```

Koa 的 Cookie 接口定义：

```
interface Cookies {
  get(name: string, options?: { signed: boolean }): string | undefined;
  set(name: string, value: string, options?: CookiesSetOptions): void;
}

interface CookiesSetOptions {
  maxAge?: number;
  signed?: boolean;
  expires?: Date;
  path?: string;
  domain?: string;
  secure?: boolean;
  httpOnly?: boolean;
  sameSite?: 'strict' | 'lax' | 'none';
  overwrite?: boolean;
}
```

cookies package相关代码

```
// 验证字段内容的正则表达式
var fieldContentRegExp = /^[\u0009\u0020-\u007e\u0080-\u00ff]+$/;

// 验证name和value
if (!fieldContentRegExp.test(name)) {
  throw new TypeError('argument name is invalid');
}

// 安全相关选项
Cookie.prototype.httpOnly = true;  // 防止XSS攻击
Cookie.prototype.secure = false;   // HTTPS传输
Cookie.prototype.sameSite = false; // 防止CSRF攻击
```

1. 风险点

```
默认配置不安全，需要看具体实现
```

❗ XSS 防御与风险 (react & fusion-core)

1. React的XSS防护 （使用React 组件的场景）

https://github.com/facebook/react/tree/main/packages/react-dom/src

```
fusionJS 使用 React 作为视图层，React 本身就提供了 XSS 防护

fusion-react 包
```

1. React 的内置 XSS 防护

```
import * as React from 'react';
import render from '../client';
```

```
3
4  // 使用 React.createElement 创建元素
5  render(React.createElement('span', null, 'hello'));
```

2. **服务端渲染时使用 ReactDOM.renderToString**

```
1  import {renderToString} from 'react-dom/server';
2
3  // 使用 renderToString 进行服务端渲染
4  expect(/Loading/.test(renderToString(app))).toBeTruthy();
```

3. **客户端渲染使用 ReactDOM.render**

```
1  // 在客户端使用 React 的标准渲染方法
2  render(React.createElement('span', null, 'hello'));
```

主要的 XSS 防护依赖于:

1. React 的自动转义机制 - React 会自动转义渲染的内容
2. React 的 JSX 编译 - 通过 Babel 配置确保正确处理
3. React 的服务端渲染安全机制

从 .babelrc 可以看到使用了标准的 React preset:

```
1  {
2    "presets": [
3      "@babel/preset-typescript",
4      [
5        "@babel/preset-react",
6        {
7          "runtime": "automatic"
8        }
9      ]
```

```
  ]
}
```

---

2. fusion-core/src/sanitization.ts （非React 组件的场景）

```
https://fusionjs.com/api/fusion-core#sanitization

 1. 中间件中的 HTML 处理
 2. 模板字符串的处理
 3. 服务端直接输出 HTML 的场景

 例如：
 - 服务端渲染    ############################################################

 // 在服务端渲染时，可能需要在 React 组件之外处理 HTML
// 比如：添加元数据、注入脚本等
app.middleware((ctx, next) => {
  // 这里的 HTML 处理就需要使用 fusion-core 的 sanitization
  ctx.template.head.push(html`<meta charset="utf-8">`);
  return next();
});

- 模版拼接       ############################################################

// React 组件中的内容：
<div>{userInput}</div>  // 使用 React 的转义

// 模板字符串中的内容：
html`<div>${userInput}</div>`  // 使用 fusion-core 的转义

##############################################################################

export {html, dangerouslySetHTML, consumeSanitizedHTML, escape, unescape};

五个函数，用于处理不安全的输入输出
```

```
##############################################################################
html 模板标签函数
```

```
3
4   html = (
5     [head, ...rest]: TemplateStringsArray,  // 静态字符串部分
6     ...values: Array<string>                // 动态插值部分
7   ): SanitizedHTMLWrapper => {
8     const obj = {};
9     // 定义 inspect 方法，用于调试时显示
10    Object.defineProperty(obj, inspect, {...});
11
12    // 将安全的HTML存储在不可枚举的属性中
13    Object.defineProperty(obj, key, {
14      enumerable: false,
15      configurable: false,
16      value: head + values.map((s, i) => escape(s) + rest[i]).join('')
17    });
18    return obj;
19  };
20
21  ##############################################################################
22  escape 函数 - 转义不安全字符
23
24  escape = (str: any): string => {
25    // 如果已经是安全的HTML，直接返回
26    if (str && str[key] !== undefined) return consumeSanitizedHTML(str);
27    // 转义危险字符
28    return String(str).replace(/[<>&"\u2028\u2029]/g, replaceForbidden);
29  };
30
31    const forbiddenChars = {
32      '<': '\\u003C',
33      '>': '\\u003E',
34      '"': '\\u0022',
35      '&': '\\u0026',
36      '\u2028': '\\u2028',
37      '\u2029': '\\u2029',
38    };
39    const replaceForbidden = (c) => forbiddenChars[c];
40
41  ##############################################################################
42  unescape 解码函数
43
44  const replaceEscaped = (c) => String.fromCodePoint(parseInt(c.slice(2), 16));
45  const unescape = (str: string): string => {
46    return str.replace(
47      /\\u003C|\\u003E|\\u0022|\\u002F|\\u2028|\\u2029|\\u0026/g,
48      replaceEscaped
49    );
```

```
};

################################################################################
consumeSanitizedHTML - 获取安全HTML

consumeSanitizedHTML = (h: SanitizedHTMLWrapper): string => {
  // 如果直接传入字符串，抛出错误提示使用html标签函数
  if (typeof h === 'string') {
    throw new Error(`Unsanitized html. Use html\`${h}\``);
  }
  return h[key];
};


################################################################################
dangerouslySetHTML - 不安全的HTML设置

dangerouslySetHTML = (str: string): any =>
  html([str]); // 将不安全的HTML包装成安全的形式

################################################################################

使用例子:

// 安全：静态内容不转义，动态内容自动转义
const safe = html`<div>${userInput}</div>`;

// 不安全：会抛出错误
const unsafe = '<div>' + userInput + '</div>';

// 必要时使用不安全方法（需谨慎）
const trusted = dangerouslySetHTML(verifiedHTML);
```

3. 风险点：

```
1. 用html模版标签的时候，内部调用escape处理，但是，forbiddenChars定义的被转义字符无法覆盖所有情况，

- forbiddenChars里没有单引号，单引号对的属性无法处理
- URL属性没有处理，例如src/href等
- 还有输出在javascript代码段中:
    html`<script> ${str}</script>`

```

```
不过如果有X-XSS-Protection保护，可以防止Reffected XSS
或结合CSP设置


2. dangerouslySetHTML 字符串内容会被直接渲染，不会进行任何转义

// 1. 只用于可信的、已处理过的 HTML
dangerouslySetHTML(trustedHtml);

// 2. 永远不要直接注入用户输入
// 错误示例 - 不要这样做!
dangerouslySetHTML(userInput);

```

❗CSRF protection （fusion-plugin-csrf-protection ）

https://fusionjs.com/api/fusion-plugin-csrf-protection 用法

```

**工作方式:**

获取csrf-token

if (ctx.path === '/csrf-token' && ctx.method === 'POST') {
  // TODO(#158): Remove this once clients have had the opportunity to upgrade
  ctx.set('x-csrf-token', 'x');
  ctx.status = 200;
  ctx.body = '';
}


浏览器端
// src/browser.ts
const enhancer = (fetch: Fetch) => {
  return (url, options) => {
    const isCsrfMethod = verifyMethod(options.method || 'GET');
    if (isCsrfMethod) {
      return fetch(prefix + String(url), {
        ...options,
        credentials: 'same-origin',  // 确保只有同源才可发送 cookies
        headers: {
          ...options.headers,
          'x-csrf-token': 'x',      // 添加 CSRF token, token 就是一个固定的 'x'
```

```
        },
      });
    }
    return fetch(prefix + String(url), options);
  };
};

服务器端验证
// src/server.ts
async function csrfMiddleware(ctx, next) {
  if (verifyMethod(ctx.method) && !ignoreSet.has(ctx.path)) {
    const token = ctx.headers['x-csrf-token'];
    if (!token) {    // 只是检查 token 是否存在
      ctx.throw(403, 'Missing csrf token...');
    }
  }
  return next();
}
```

- 风险点

```
1.  如果真的直接用这样的设计，那csrf根本没什么用：
    - Token是固定的('x'),攻击者可以轻易猜测或获取
    - 服务端验证过于简单,只验证token存在性而不验证其值的正确性

2.  ['POST', 'PUT', 'DELETE', 'PATCH'] 以外的Method不被保护


3. 支持配置豁免路径
import {CsrfIgnoreRoutesToken} from 'fusion-plugin-csrf-protection';
app.register(CsrfIgnoreRoutesToken, ['/api/public']);


4. /_events 路径默认被忽略，不要在该路径下处理敏感操作
// src/server.ts
const {ignored = []} = deps;
const ignoreSet = new Set(ignored);
ignoreSet.add('/_events'); // 内置忽略分析路径
```

❗序列化 & 原型链污染 风险 （fusion-plugin-rpc）

https://fusionjs.com/api/fusion-plugin-rpc 用法

```
RPC处理APi访问的过程:

客户端 (browser) ---call helper函数--- 发送序列化Javascript对象 ---> 服务器端API endpoint
服务器端API 先对其进行反序列化操作，再交给 "handler" 函数
handler 函数将对请求进行响应处理，并返回 以JSON 形式序列化的Javascript对象给客户端

响应将作为 Redux Action被发送，Reducer 会处理该动作，并在客户端更新 Redux 状态



fusion-plugin-rpc-redux-react插件会在浏览器端自动使用FetchToken来访问每个RPC处理程序的 API 端点

HOC 是一个函数，它接收一个组件作为输入，返回一个新的增强后的组件
类似于一个包装器，可以为组件添加额外的功能或数据
```

1. browser.ts

在浏览器端 (browser.ts), 发送**非FormData 请求**时有序列化操作:

```
    return fetch(
      `${apiPath}${rpcId}${queryParams}`,
      args instanceof FormData
        ? {
            ...options,
            method: 'POST',
            headers: {
              // Content-Type will be set automatically
              ...(headers || {}),
            },                  // FormData == multipart/form-data
            body: args,    // 直接发送 FormData 对象，不做序列化
          }
        : {
            ...options,
            method: 'POST',
            // $FlowFixMe
            headers: {
              'Content-Type': 'application/json',
              ...(headers || {}),
```

```
            },            // 非 FormData 才进行 JSON 序列化
            body: JSON.stringify(args || {}),
        }
    )


对响应进行反序列化:
    .then((r) => r.json())
```

2. server.ts

```
对于 multipart/form-data 类型的请求,使用 formidable 解析:

            if (
               ctx.req &&
               ctx.req.headers &&
               ctx.req.headers['content-type'] &&
               ctx.req.headers['content-type'].indexOf(
                 'multipart/form-data'
               ) !== -1
             ) {
               const form = new formidable.IncomingForm();
               body = await new Promise((resolve, reject) => {
                 form.parse(
                   ctx.req,
                   (
                     err,
                     fields: {
                       [x: string]: any;
                     },
                     files
                   ) => {
                     if (err) {
                       reject(err);
                     }

                     resolve({
                       ...fields,
                       ...files,
                     });
                   }
```

```
                  );
                });
              } else {

对于其他类型的请求(如 application/json),使用 koa-bodyparser 解析:

                // const parseBody = bodyparser(bodyParserOptions);

                await parseBody(ctx, () => Promise.resolve());
              }


 服务器响应时也会进行序列化,将响应包装成统一格式,
 Koa 会自动将 ctx.body 序列化为 JSON 并设置正确的 Content-Type header

            // 成功响应
            ctx.body = {
              status: 'success',
              data: result
            };

            // 失败响应
            ctx.body = {
              status: 'failure',
              data: {
                message: error.message,
                code: error.code,
                meta: error.meta,
              }
            };

```

❗序列化 & 原型链污染 风险 （fusion-plugin-react-redux）

https://fusionjs.com/api/fusion-plugin-react-redux 用法

1. 序列化 - 服务端

```
src/server.tsc

middleware(_, redux) {
  return async (ctx, next) => {
    // 序列化 Redux state
```

```
      const serialized = serialize(store.getState());
      const script = html`
        <script type="application/json" id="__REDUX_STATE__">
          ${serialized}
        </script>
      `;
    }
  }
```

2. 反序列化 - 客户端

```
src/browser.tsx

const getReduxState = () => {
  const stateElement = document.getElementById('__REDUX_STATE__');
  if (stateElement) {
    return deserialize(unescape(stateElement.textContent));
  }
};
```

3. 编解码实现

```
src/codec.ts

// 序列化
export function serialize(obj: any) {
  return encode(JSON.stringify(obj));
}

// 反序列化
export function deserialize(str: string) {
  return parseJSONWithUndefined(decode(str));
}

// 特殊字符编码
const encodeChars = {
  '\\': '%5C',
  '%': '%25',
};
```

4. 风险点

```
1. 反序列化风险

function parseJSONWithUndefined(str: string) {
  // 直接使用 JSON.parse，存在安全风险
  return JSON.parse(str, (key, value) => {
    if (value === '__UNDEFINED__') {
      return undefined;
    }
    return value;
  });
}

2. 序列化风险

// 当前实现
export function serialize(obj: any) {
  // 直接使用 JSON.stringify，没有任何安全检查
  return encode(JSON.stringify(obj));
}

// 编码特殊字符
const encodeChars = {
  '\\': '%5C',
  '%': '%25',
};


3. 反序列化/原型链污染风险

const getReduxState = () => {
  const stateElement = document.getElementById('__REDUX_STATE__');
  if (stateElement) {
    // 直接反序列化 DOM 内容，可能导致原型链污染
    return deserialize(unescape(stateElement.textContent));
  }
};


const serialized = JSON.stringify(state);
const script = html`
  <script type="application/json" id="__REDUX_STATE__">
    ${serialized}
  </script>
`;

```

❗ GraphQL 集成 (fusion-plugin-apollo)

https://fusionjs.com/api/fusion-plugin-apollo 用法

```
// src/plugin.tsx
export default (renderFn: Render) =>
  createPlugin<DepsType, ProvidesType>({
    // ...
    middleware({schema, endpoint = '/graphql'}) {
      // 创建 Apollo Server 实例
      const server = new ApolloServer({
        schema,
        context: ({ctx}) => ctx,
        // ...
      });
    }
  });
```

```
graphql的使用不当会造成安全风险
```

❗ 路由插件 (fusion-plugin-react-router)

用法

```
可以注册自定义静态上下文来处理服务器端重定向和设置状态代码
```

1. 风险点

```
关注Open Redirection漏洞，例如：

app.register(GetStaticContextToken, (ctx: Context) => {
  return {
    set status(code: string) {
      ctx.status = code;
    },
```

```
      set url(url: string) {
        ctx.status = 307;
        ctx.redirect(url);
      }
    }
});


这里的url变量可能从 客户端获取，可能是：

1-用户输入
2-HTTP request header
3-DOM中的 window.location.*     (hash, href, search, etc)
```

附录

1. React XSS防护

https://github.com/facebook/react/tree/main/packages/react-dom

React 的 JSX 转义主要通过以下几个步骤工作：

1. JSX 转换

首先，JSX 会被 Babel 或 TypeScript 转换为 `React.createElement` 调用：

```
// JSX 代码
const element = <h1 className="greeting">Hello, {name}!</h1>;

// 转换后的代码
const element = React.createElement(
  'h1',
  { className: 'greeting' },
  'Hello, ',
  name,
  '!'
);
```

2. createElement 处理

React 的 `createElement` 函数会处理这些参数：

```
export function createElement(type, config, children) {
  // ... 属性处理逻辑 ...

  // 处理 children
  const childrenLength = arguments.length - 2;
  if (childrenLength === 1) {
    props.children = children;
  } else if (childrenLength > 1) {
    const childArray = Array(childrenLength);
    for (let i = 0; i < childrenLength; i++) {
      childArray[i] = arguments[i + 2];
    }
    props.children = childArray;
  }

  return ReactElement(type, key, ref, props);
}
```

3. 字符串转义

React 在渲染时会自动转义内容：

```
function setTextContent(node, text) {
  if (text) {
    const firstChild = node.firstChild;
    if (
      firstChild &&
      firstChild === node.lastChild &&
      firstChild.nodeType === TEXT_NODE
    ) {
      firstChild.nodeValue = text;
      return;
    }
  }
```

```
    node.textContent = text;
  }
```

4. 属性处理

React 对不同类型的属性有不同的处理方式：

```
// 示例：属性处理
export const properties = {
  // 特殊属性处理
  accept: 0,
  acceptCharset: 1,
  // ...

  // 布尔属性
  checked: MUST_USE_PROPERTY | HAS_BOOLEAN_VALUE,

  // DOM 属性
  className: 0,

  // 危险属性需要特殊处理
  dangerouslySetInnerHTML: 0,
};
```

5. 安全特性

a. **自动转义**:

```
const userInput = '<script>alert("xss")</script>';
// 安全渲染，会被转义
return <div>{userInput}</div>;
// 输出: &lt;script&gt;alert(&quot;xss&quot;)&lt;/script&gt;
```

b. **属性值处理**:

```
const userAttr = '" onclick="alert(1)';
// 属性值会被安全处理
return <div id={userAttr}>Hello</div>;
// 输出: <div id="&amp;quot; onclick=&amp;quot;alert(1)">Hello</div>
```

c. **URL 属性特殊处理**:

```
// React 会检查 URL 协议
const userUrl = 'javascript:alert(1)';
return <a href={userUrl}>Click me</a>;
// 不安全的协议会被过滤
```

d. **style 对象处理**:

```
// style 必须是对象，不能是字符串
const styles = {
  color: userInput,
  '--custom-property': userInput // CSS 自定义属性也会被处理
};
return <div style={styles}>Hello</div>;
```

6. 特殊情况处理

a. **dangerouslySetInnerHTML**:

```
// 必须显式使用特定的API来插入HTML
const html = { __html: '<strong>Bold text</strong>' };
return <div dangerouslySetInnerHTML={html} />;
```

b. **事件处理**:

```
// 事件处理器必须是函数引用
```

```
  const handler = userInput; // 这不会工作
  return <button onClick={handler}>Click</button>;
```

7. **防御机制**:

```
// 1. 属性名规范化
<div className="test" /> // 而不是 class="test"

// 2. 禁止注入脚本
<img src="x" onerror="alert(1)" /> // onerror 会被忽略

// 3. 样式安全
<div style={{ color: 'red' }} /> // 必须使用对象语法

// 4. 内容类型检查
<script>alert(1)</script> // script 标签的内容不会被执行
```

这些机制共同确保了：

1. 所有输出到 DOM 的内容都经过适当转义
2. 属性值被正确处理和转义
3. 危险的 HTML/脚本 注入被阻止
4. URL 和样式值被安全处理
5. 事件处理器只能是受控的函数引用

这使得 React 的 XSS 防护非常强大，但开发者仍需注意：

1. 避免使用 `dangerouslySetInnerHTML`
2. 小心处理 URL 属性
3. 不要在 `style` 对象中使用用户输入
4. 注意服务端渲染时的内容处理