56. 什么是 XSS、CSRF？如何从 JS 层面避免？

XSS 与 CSRF 防御 技术笔记

1. 核心概念 (Core Concept)

跨站脚本攻击 (Cross-Site Scripting, XSS) 和 **跨站请求伪造 (Cross-Site Request Forgery, CSRF)** 是两种常见的 Web 安全漏洞。XSS 旨在向用户浏览器注入恶意脚本，利用用户的身份（权限）执行恶意操作或窃取信息；CSRF 则旨在利用用户已有的会话（认证）信息，在用户不知情或未经授权的情况下，伪造用户的请求执行非法操作。

2. 为什么需要它？ (The "Why")

- **保护用户数据与隐私:** XSS 可用于窃取用户的 Cookie、会话信息、敏感输入等，导致用户账号被盗或隐私泄露。CSRF 可能导致用户的敏感操作（如修改密码、转账、删除数据）在用户未知情下执行。
- **维护网站声誉与数据完整性:** 受 XSS 或 CSRF 攻击的网站可能被用于传播恶意内容、进行钓鱼攻击，损害网站信誉，甚至破坏服务器数据。
- **符合安全合规性要求:** 许多行业的安全标准和法规要求 Web 应用具备基本的安全防护能力，防止这些常见攻击。

3. API 与用法 (API & Usage)

注意: XSS 和 CSRF 的防御主要依赖于后端、HTTP 协议特性以及浏览器安全策略。从 JavaScript 层面更多是配合实现防御策略，或者加固某些环节。以下将侧重于从 JS 层面可以配合或实现的常见防御措施。

3.1 XSS 防御 (从 JS 层面配合)

XSS 主要是由于把用户输入的内容不加过滤地作为 HTML/JavaScript 代码执行导致的。关键在于对不可信数据的处理。

- **输入过滤与输出转义 (Input Filtering & Output Escaping):** 这是最核心的防御，即便从 JS 层面输入数据，也要假设其不可信。
  - **输入过滤:** 在接收用户输入时，对数据进行验证和清洗，比如限制输入长度、字符类型等，但纯粹的过滤很难彻底防止 XSS。
  - **输出转义:** 在将不可信数据插入到 HTML 页面时，根据插入位置（HTML Body, HTML 属性, JavaScript 代码, URL 等）进行相应的转义。这是防止 XSS 的关键。从 JS 层面，即使使用 `innerHTML` 或其他 DOM manipulation 方法，也要确保传入的数据经过了适当的转义。现代前端框架（如 React, Vue, Angular）通常内置了自动转义机制，但在手动操作 DOM 时需特别주의。

```
// 错误示例：直接将用户输入插入到 DOM
// 假设 userInput = "<script>alert('XSS')</script>";
document.getElementById('unsafeDiv').innerHTML = userInput; // 存在 XSS 风险

// 安全示例：使用 textContent 或进行转义（框架通常自动完成）
document.getElementById('safeDiv').textContent = userInput; // 作为纯文本插入

// 如果必须插入 HTML，使用成熟的第三方库进行清洗和过滤
// 示例（使用 DOMPurify 库 - 需要引入）：
// import DOMPurify from 'dompurify';
// const cleanHTML = DOMPurify.sanitize(userInput);
// document.getElementById('safeHTMLDiv').innerHTML = cleanHTML;
```

- **设置 HttpOnly 属性的 Cookie:** 带有 `HttpOnly` 标志的 Cookie 不能通过 JavaScript 的 `document.cookie` 接口访问。这可以防止 XSS 攻击者窃取用户的会话 Cookie，但不能阻止所有类型的 XSS 攻击。
- **内容安全策略 (Content Security Policy, CSP):** CSP 是一个 HTTP 响应头部，但也可以通过 `<meta>` 标签在 HTML 中定义。它允许开发者限制浏览器加载资源的来源（脚本、样式、图片等），以及限制行内脚本和 `eval()` 的使用，显著降低 XSS 风险。从 JS 层面可以配合 CSP，例如避免使用行内事件处理器 (`onclick="..."`) 和动态执行代码 (`eval()`)。

```
<!-- 示例：通过 meta 标签设置 CSP -->
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' https://trusted.cdn.com; style-src 'self';">
```

3.2 CSRF 防御 (从 JS 层面配合)

CSRF 的核心在于利用用户的会话凭证（通常是 Cookie）来伪造请求。防御的关键是让服务器能够验证请求是否来自用户合法的操作。

- **同源策略 (Same-Origin Policy):** 浏览器内置的安全机制，限制了来自不同源的脚本对 DOM 的访问和某些类型的请求。但 CSRF 攻击利用的不是脚本的跨域能力，而是浏览器发送请求时**自动带上同源 Cookie** 的特性。
- **使用 Anti-CSRF Token:** 这是最常用的 CSRF 防御手段。
  - 服务器在渲染表单或页面时，生成一个唯一的、随用户会话变化的 Token，并将其嵌入到页面中（如隐藏字段或 JS 变量）。
  - 用户提交请求时，前端（通常通过 JS）从页面中读取这个 Token，并作为请求数据的一部分（如请求体参数或自定义请求头）发送给服务器。
  - 服务器接收请求后，验证携带的 Token 是否与服务器端存储的该用户会话对应的 Token 一致。如果一致，则认为是合法请求；否则，认为是伪造请求并拒绝。

```
// 示例：使用 Fetch API 发送带有 CSRF Token 的 POST 请求
// 假设在页面中有一个隐藏 input 或 JS 变量 csrfToken 存储了 Token
const csrfToken = document.querySelector('meta[name="csrf-
token"]').getAttribute('content'); // 或从 JS 变量获取

fetch('/api/update', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'X-CSRF-Token': csrfToken // 常见的自定义请求头
        // 或者将 token 放到 request body 中
    },
    body: JSON.stringify({
        data: 'some data'
        // csrf_token: csrfToken // 另一种方式，放到 body
    })
})
.then(response => {
    // 处理响应
});
```

- **SameSite Cookie 属性:** 这是现代浏览器提供的一种重要的 CSRF 防御机制。它可以限制浏览器在进行跨站请求时是否发送 Cookie。
  - `SameSite=Lax` (默认值): 跨站请求通常不携带 Cookie，但例外情况包括通过顶部导航（如 `<a href="...">` 或 `<form method="GET">` 简单请求）发起的 GET 请求会携带 Cookie。
  - `SameSite=Strict`: 任何跨站请求都不携带 Cookie。安全性最高，但可能影响用户体验（如从第三方网站链接到本站时无法保持登录状态）。
  - `SameSite=None; Secure`: 只有在 HTTPS 连接下，才能在跨站请求中携带 Cookie。需要确保网站使用 HTTPS。

  后端设置 Cookie 时添加此属性：
  `Set-Cookie: sessionid=abcdef; SameSite=Lax; Secure; HttpOnly;`

  从 JS 无法直接设置 `SameSite` 属性，这通常是服务器端或反向代理的配置。但作为前端开发者应了解其作用并配合测试。
- **Referer 检查 (作为辅助手段):** 服务器检查请求的 `Referer` 头部是否是同一个域。容易被绕过（用户或浏览器设置不发送 Referer，Referer 可能被伪造，或者在 HTTPS 到 HTTP 跳转时不发送 Referer）。不应作为主要的防御手段。

4. 关键注意事项 (Key Considerations)

- **XSS 防御核心在于输出转义:** 永远不要相信用户的输入，在将不可信数据呈现到页面之前，必须根据其上下文位置进行严格的转义。使用成熟、经过安全审计的模板引擎或前端框架通常会自动处理转义。

- **CSRF 防御核心在于验证请求来源:** 通过 Anti-CSRF Token 或 SameSite Cookie 等机制，确保敏感操作的请求确实是用户主动发起的，而不是被第三方网站伪造的。
- **防御是多层次的:** XSS 和 CSRF 的防御不是单一的，需要前端、后端、运维（如配置 CSP、WAF）等多方配合。
- **关注细节和边界情况:** 即使使用了防御措施，也需要注意实现细节，如 Token 的生成、存储和验证是否安全，SameSite Cookie 在各种浏览器和场景下的行为是否符合预期。
- **定期进行安全审计和测试:** 软件会不断更新，新的漏洞也可能出现。定期对代码进行安全审查和渗透测试是必要的。

5. 参考资料 (References)

- **OWASP Cheat Sheet Series - XSS Prevention Cheat Sheet:** https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html (业界公认的 XSS 防御指南)
- **OWASP Cheat Sheet Series - CSRF Prevention Cheat Sheet:** https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html (业界公认的 CSRF 防御指南)
- **MDN Web Docs - Content Security Policy (CSP):** https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- **MDN Web Docs - SameSite cookies:** https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite