

前端安全相关面试题（20题）

1. 什么是XSS攻击？有哪些类型？如何防范？

参考答案：

XSS（Cross-Site Scripting）跨站脚本攻击是指攻击者在网页中注入恶意脚本代码，当用户浏览该网页时，脚本会在用户浏览器中执行。

类型：

- **存储型XSS**：恶意脚本被存储在服务器数据库中
- **反射型XSS**：恶意脚本通过URL参数等方式反射给用户
- **DOM型XSS**：通过修改DOM结构来执行恶意脚本

防范措施：

- 输入验证和输出编码
- 使用CSP（Content Security Policy）
- 设置HttpOnly Cookie
- 使用安全的DOM操作方法

2. 什么是CSRF攻击？如何防范？

参考答案：

CSRF（Cross-Site Request Forgery）跨站请求伪造是指攻击者诱导用户在已登录的网站上执行非本意的操作。

防范措施：

- 使用CSRF Token验证
- 检查Referer头
- 使用SameSite Cookie属性

- 重要操作添加验证码
- 使用双重Cookie验证

3. 什么是CSP？如何配置和使用？

参考答案：

CSP（Content Security Policy）内容安全策略是一种安全机制，用于检测和减轻XSS攻击。

配置方式：

代码块

```
1  <!-- HTTP头配置 -->
2  Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'
3
4  <!-- Meta标签配置 -->
5  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-
   src 'self'">
```

常用指令：

- `default-src`：默认策略
- `script-src`：脚本来源
- `style-src`：样式来源
- `img-src`：图片来源

4. 什么是点击劫持攻击？如何防范？

参考答案：

点击劫持（Clickjacking）是指攻击者使用透明或不可见的iframe覆盖在正常网页上，诱导用户点击。

防范措施：

- 设置X-Frame-Options头：`DENY`、`SAMEORIGIN`

- 使用CSP的frame-ancestors指令
- JavaScript框架检测：

代码块

```
1  if (top !== window) {  
2      top.location = window.location;  
3  }
```

5. Cookie有哪些安全属性？ 分别有什么作用？

参考答案：

安全属性：

- **HttpOnly**：防止JavaScript访问Cookie，减少XSS攻击
- **Secure**：只在HTTPS连接中传输Cookie
- **SameSite**：控制跨站请求时是否发送Cookie
 - **Strict**：完全禁止跨站发送
 - **Lax**：部分跨站请求可发送
 - **None**：允许跨站发送（需配合Secure）

代码块

```
1  document.cookie = "sessionId=abc123; HttpOnly; Secure; SameSite=Strict";
```

6. 什么是SQL注入？ 前端如何防范？

参考答案：

SQL注入是指攻击者通过在输入中插入恶意SQL代码来操作数据库。

前端防范：

- 输入验证和过滤
- 使用参数化查询（后端配合）
- 对特殊字符进行转义
- 限制输入长度和格式
- 使用白名单验证

代码块

```
1 // 输入验证示例
2 function validateInput(input) {
3     const sqlKeywords = /(\b(SELECT|INSERT|UPDATE|DELETE|DROP|UNION)\b)/i;
4     return !sqlKeywords.test(input);
5 }
```

7. 什么是同源策略？如何安全地进行跨域请求？

参考答案：

同源策略要求协议、域名、端口完全相同才能访问资源。

安全跨域方案：

- **CORS**：服务器设置Access-Control-Allow-Origin
- **JSONP**：仅支持GET请求，存在安全风险
- **代理服务器**：后端代理请求
- **PostMessage**：用于iframe通信

代码块

```
1 // CORS示例
2 fetch('https://api.example.com/data', {
3     method: 'POST',
4     credentials: 'include', // 发送Cookie
5     headers: {
6         'Content-Type': 'application/json'
7     }
8 })
```

```
8  });
```

8. 如何安全地存储敏感信息？

参考答案：

存储方案对比：

- **LocalStorage/SessionStorage**：明文存储，XSS可访问
- **Cookie**：可设置HttpOnly，但仍有CSRF风险
- **IndexedDB**：XSS可访问

安全措施：

- 敏感信息加密存储
- 使用短期Token代替长期凭证
- 定期清理存储数据
- 避免在前端存储密码等敏感信息

代码块

```
1  // 加密存储示例
2  const encryptData = (data, key) => {
3      return CryptoJS.AES.encrypt(JSON.stringify(data), key).toString();
4  };
```

9. 什么是中间人攻击？如何防范？

参考答案：

中间人攻击（MITM）是指攻击者在通信双方之间拦截和修改数据。

防范措施：

- 使用HTTPS加密传输
- 验证SSL证书
- 使用HSTS（HTTP Strict Transport Security）
- 证书固定（Certificate Pinning）
- 避免使用公共WiFi进行敏感操作

代码块

```
1 // HSTS头设置
2 Strict-Transport-Security: max-age=31536000; includeSubDomains
```

10. 如何防范恶意文件上传？

参考答案：

前端防范：

- 文件类型验证（MIME类型检查）
- 文件大小限制
- 文件名过滤
- 文件内容检测

代码块

```
1 function validateFile(file) {
2     const allowedTypes = ['image/jpeg', 'image/png', 'image/gif'];
3     const maxSize = 5 * 1024 * 1024; // 5MB
4
5     if (!allowedTypes.includes(file.type)) {
6         throw new Error('不支持的文件类型');
7     }
8
9     if (file.size > maxSize) {
10        throw new Error('文件过大');
11    }
12
13    return true;
```

11. 什么是DOM Clobbering攻击？如何防范？

参考答案：

DOM Clobbering是指通过HTML注入来覆盖全局变量或DOM属性，影响JavaScript代码执行。

攻击示例：

代码块

```
1 
2 <script>
3 // 原本期望的对象被覆盖
4 console.log(window.userConfig); // 输出img元素而非配置对象
5 </script>
```

防范措施：

- 避免使用全局变量
- 使用命名空间
- 严格的HTML过滤
- 使用 `hasOwnProperty` 检查属性

12. 什么是原型污染攻击？如何防范？

参考答案：

原型污染是指攻击者修改JavaScript对象的原型，影响所有实例对象。

攻击示例：

代码块

```
1 // 恶意输入
2 const maliciousInput = '{"__proto__": {"isAdmin": true}}';
3 const userData = JSON.parse(maliciousInput);
4 // 所有对象都被污染
5 console.log({}.isAdmin); // true
```

防范措施:

- 使用 `Object.create(null)` 创建纯净对象
- 使用 `Map` 代替普通对象
- 输入验证, 过滤 `__proto__` 等危险属性
- 使用 `Object.freeze()` 冻结原型

13. 如何安全地处理用户输入?

参考答案:

处理原则:

- 永远不信任用户输入
- 输入验证: 白名单验证优于黑名单
- 输出编码: 根据上下文选择编码方式
- 长度限制: 防止DoS攻击

代码块

```
1 // 输入清理函数
2 function sanitizeInput(input) {
3     return input
4         .replace(/[<>\"']/g, '') // 移除危险字符
5         .trim()
6         .substring(0, 1000); // 长度限制
7 }
8
9 // HTML编码
10 function escapeHtml(text) {
11     const div = document.createElement('div');
12     div.textContent = text;
```

```
13     return div.innerHTML;
14 }
```

14. 什么是依赖混淆攻击？如何防范？

参考答案：

依赖混淆攻击是指攻击者上传恶意包到公共仓库，利用包管理器的解析机制让应用下载恶意包。

防范措施：

- 使用私有npm仓库
- 锁定依赖版本（package-lock.json）
- 定期审计依赖包： `npm audit`
- 使用 `.npmrc` 配置可信源
- 监控依赖包的异常行为

代码块

```
1 // .npmrc配置示例
2 @company:registry=https://private-registry.company.com/
3 registry=https://registry.npmjs.org/
```

15. 如何防范ReDoS（正则表达式拒绝服务）攻击？

参考答案：

ReDoS攻击利用正则表达式的回溯特性，通过特殊输入导致CPU占用过高。

危险模式：

代码块

```
1 // 危险的正则表达式
```

```
2  const badRegex = /^(a+)+$/;
3  const input = 'a'.repeat(25) + 'X'; // 触发灾难性回溯
```

防范措施：

- 避免嵌套量词： `(a+)+`、`(a*)*`
- 使用具体量词： `{1,10}` 代替 `+`
- 设置超时限制
- 使用正则表达式安全检测工具
- 考虑使用有限状态自动机

16. 什么是Tabnabbing攻击？如何防范？

参考答案：

Tabnabbing攻击是指恶意网站通过 `window.opener` 修改原页面，进行钓鱼攻击。

攻击场景：

代码块

```
1  // 恶意页面代码
2  if (window.opener) {
3      window.opener.location = 'https://fake-bank.com/login';
4  }
```

防范措施：

代码块

```
1  // 使用rel="noopener noreferrer"
2  <a href="https://external-site.com" target="_blank" rel="noopener noreferrer">
    链接</a>
3
4  // JavaScript打开窗口时清除opener
5  const newWindow = window.open('https://external-site.com');
6  newWindow.opener = null;
```

17. 如何安全地使用eval()和Function构造函数？

参考答案：

安全原则：

- **避免使用：** 优先使用JSON.parse()、模板字符串等替代方案
- **输入验证：** 严格验证和过滤输入
- **沙箱环境：** 在受限环境中执行

代码块

```
1  // 危险用法
2  eval(userInput); // 永远不要这样做
3
4  // 安全替代方案
5  // 1. JSON解析
6  const data = JSON.parse(jsonString);
7
8  // 2. 模板字符串
9  const template = `Hello ${name}`;
10
11 // 3. 使用安全的解析库
12 const safeEval = require('safe-eval');
13 safeEval(code, context);
```

18. 什么是子资源完整性（SRI）？如何使用？

参考答案：

SRI（Subresource Integrity）用于验证外部资源的完整性，防止CDN劫持等攻击。

使用方法：

代码块

```
1 <!-- 为外部脚本添加integrity属性 -->
2 <script src="https://cdn.example.com/library.js"
3     integrity="sha384-
4     oqVuAfXRRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
5     crossorigin="anonymous"></script>
6 <link rel="stylesheet"
7     href="https://cdn.example.com/style.css"
8     integrity="sha384-
9     BVYiISIFeK1dGmJRAkycuHAHRg320mUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
```

生成哈希值：

代码块

```
1 # 使用openssl生成SHA384哈希
2 openssl dgst -sha384 -binary file.js | openssl base64 -A
```

19. 如何防范WebSocket安全风险？

参考答案：

主要风险：

- 跨站WebSocket劫持
- 缺乏同源策略保护
- 数据传输安全

防范措施：

代码块

```
1 // 1. 使用WSS加密连接
2 const socket = new WebSocket('wss://secure-server.com/socket');
3
4 // 2. 验证Origin头 (服务端)
5 if (request.headers.origin !== 'https://trusted-domain.com') {
6     return reject();
7 }
```

```

7   }
8
9   // 3. 使用Token认证
10  socket.onopen = function() {
11      socket.send(JSON.stringify({
12          type: 'auth',
13          token: getAuthToken()
14      }));
15  };
16
17  // 4. 输入验证
18  socket.onmessage = function(event) {
19      const data = JSON.parse(event.data);
20      if (validateMessage(data)) {
21          processMessage(data);
22      }
23  };

```

20. 如何实现安全的前端日志记录？

参考答案：

安全考虑：

- 避免记录敏感信息
- 防止日志注入攻击
- 控制日志级别

代码块

```

1   class SecurityLogger {
2       constructor() {
3           this.sensitiveFields = ['password', 'token', 'ssn', 'creditCard'];
4       }
5
6       // 清理敏感信息
7       sanitizeData(data) {
8           const cleaned = { ...data };
9           this.sensitiveFields.forEach(field => {
10               if (cleaned[field]) {
11                   cleaned[field] = '***';

```

```
12         }
13     });
14     return cleaned;
15 }
16
17 // 防止日志注入
18 escapeLogData(message) {
19     return message.replace(/\r\n/g, '');
20 }
21
22 log(level, message, data = {}) {
23     const cleanData = this.sanitizeData(data);
24     const safeMessage = this.escapeLogData(message);
25
26     console.log(`[${level}] ${safeMessage}`, cleanData);
27
28     // 发送到安全的日志服务
29     this.sendToLogService(level, safeMessage, cleanData);
30 }
31 }
```

这些面试题涵盖了前端安全的核心知识点，包括常见攻击手段、防范措施、安全最佳实践等。在实际面试中，面试官可能会深入询问具体的实现细节或结合实际项目场景进行讨论。