Electron Security

CSP

NOP Team

CSP | Electron 安全

0x01 简介

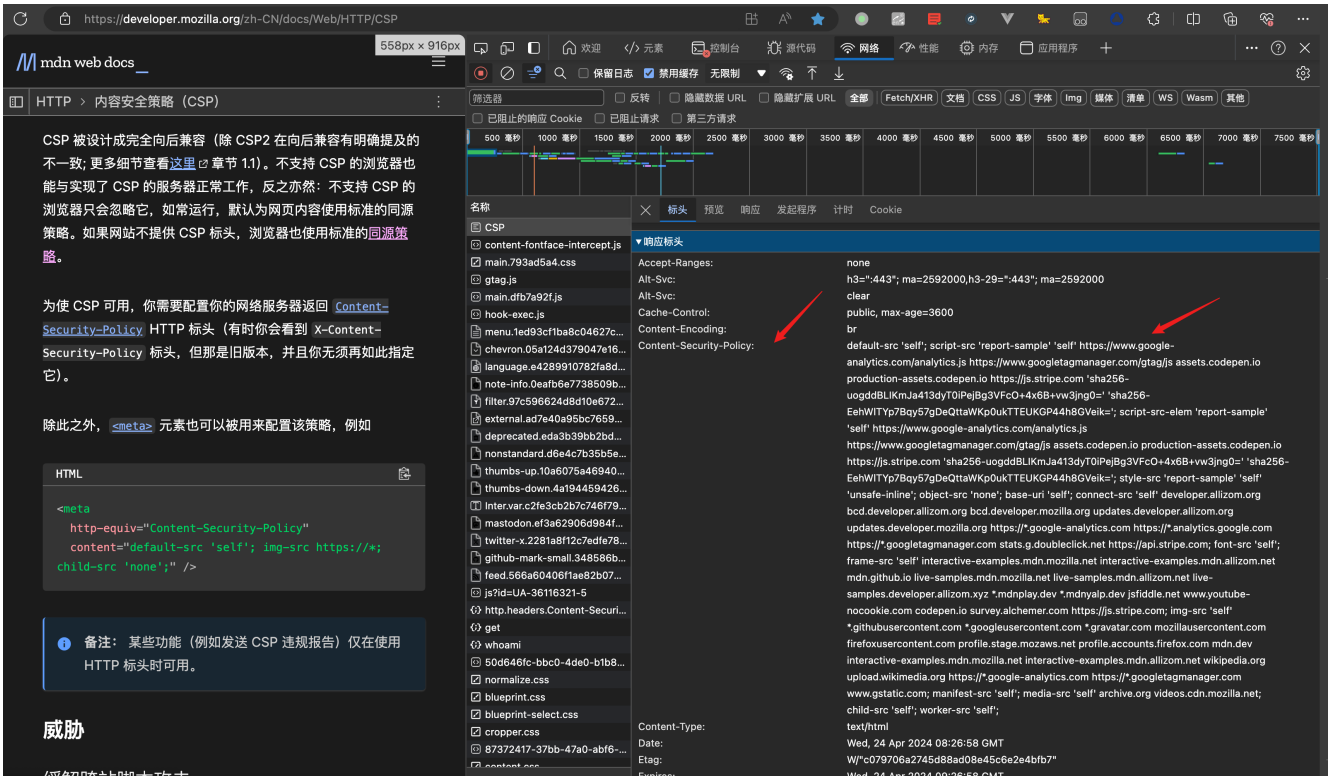
大家好，今天和大家讨论的是 `CSP` ，即内容安全策略。相信很多朋友在渗透测试的过程中已经了解过 `CSP` 了

内容安全策略（CSP）是一个额外的安全层，用于检测并削弱某些特定类型的攻击，包括跨站脚本（XSS）和数据注入攻击等。这些攻击可能造成数据盗取、网站内容污染、恶意软件分发等

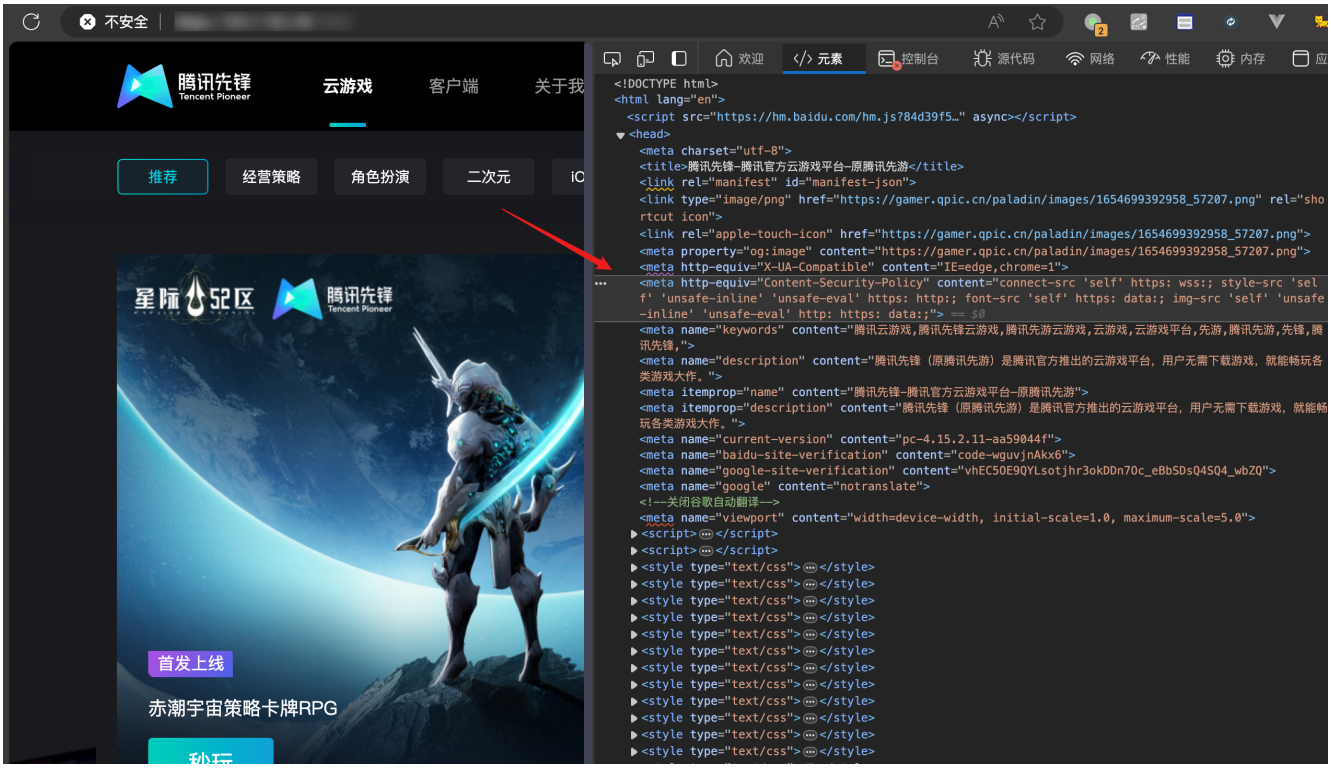
`CSP` 是一种类似**白名单机制**，它并不是局限于 `Electron` ，它是一项 `Web` 相关协议的策略，你可以通过 `CSP` 配置允许从哪些地方执行 `JavaScript` 、要求使用安全协议等

一般 `CSP` 有两种配置方式

- 通过返回包头 `Content-Security-Policy`



- 在网页中 `<meta>` 元素中进行配置




```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://*; child-src 'none';" />
```

> https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
>
> https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Security-Policy

0x02 CSP 基础

`CSP` 是以一段明文字符串的存在的，所以我们可以直接在网页中的 `<meta>` 标签中进行测试，比较方便

表现形式是下面这种形式

```
Content-Security-Policy: <policy-directive>; <policy-directive>
```

用分号分割多个策略，每个策略内部形式如下

```
指令名称 <source> <source>;
```

指令名称很好理解，例如 `img-src` 是指图片和图标的源地址限制； `script-src` 是指 `JavaScript` 的源地址

`<source>` 是允许的来源，例如 `'self'` 是限制资源允许来自同源，`www.trusted.com` 是指允许来自 `www.trusted.com` 的资源，这里是支持通配符 `*` 的

因此接下来的介绍，我们将整体分为**指令与值**

CSP 值的内容

主机名相关的值

- `https://example.com` 允许从特定源加载资源
- 端口限定：如 `https://example.com:443`，可以指定特定端口的资源
- 协议限定：如 `https:` 或 `wss:`，只允许使用特定协议的资源
- 特定路径：如 `https://example.com/path/to/resource`，可以限制到特定目录或文件
- 通配符：如 `*.example.com`，用于允许同一主域名下的所有子域名

协议相关的值

- `data:` 允许内嵌的 `data:` 形式的数据 `URI`
- `mediastream:` 允许内嵌的 `mediastream:` 形式的数据 `URI`
- `blob:` 允许内嵌的 `blob:` 形式的数据 `URI`
- `filesystem:` 允许内嵌的 `filesystem:` 形式的数据 `URI`

当然还有上面的 `http:` 和 `https:`

- `'self'` 允许加载同源资源
- `'none'` 禁止加载任何资源
- `'wasm-unsafe-eval'` 允许加载和执行WebAssembly模块，而无需通过 `"unsafe-eval"` 允许不安全的 `JavaScript` 执行
- `'unsafe-inline'` 允许内联脚本和样式（不推荐，除非必要）
- `'unsafe-eval'` 允许使用 `eval()`、`new Function()` 等动态代码执行（不推荐，除非必要）

- `'unsafe-hashes'` 允许启用特定的内联事件处理程序。如果您只需要允许内联事件处理程序，而不允许内联 `<script>` 元素或 `javascript:` ，则这是一种比使用 `unsafe-inline` 表达式更安全的方法
- `'nonce-<base64-value>'` 使用加密随机数（一次性使用的数字）的特定内联脚本的允许列表
- `'<hash-algorithm>-<base64-value>'` 脚本或样式的sha256、sha384或sha512散列
- `'strict-dynamic'` 与 `nonce` 值或 `hash` 一起使用时，允许动态生成的脚本，同时忽略其他源列表（除了 `'self'` 和 `'unsafe-inline'` ）
- `'report-sample'` 要求在违规报告中包含违规代码的示例
- `'inline-speculation-rules'` 允许在脚本中包含推测规则(试验性技术)

这里大部分比较容易理解，我挑选一些进行介绍

1) data

这个其实就是有些时候通过 `data:[<mediatype>][;base64],<data>` 这种格式加载资源相关的限制， `CSP` 是白名单，所以默认没有配置 `data` 的话，肯定是不允许的，因此假设我们允许通过 `data` 插入图片，我们需要进行如下 `CSP` 配置

```
Content-Security-Policy: img-src data:;
```

我们通过一个实验测试一下，一个 `Hello World` 程序如下

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
  margin-left: 200px;
  background: #5d9ab2 no-repeat top left;
}

.center_div {
  border: 1px solid gray;
  margin-left: auto;
  margin-right: auto;
  width: 90%;
  background-color: #d0f0f6;
```

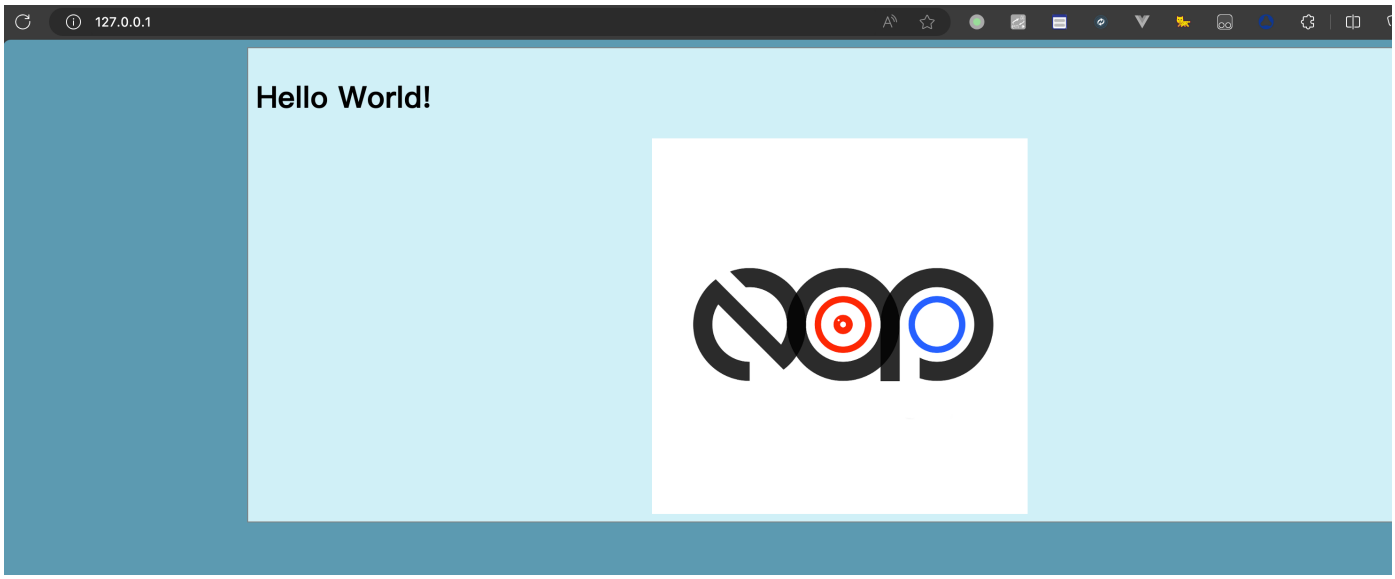
```
  text-align: left;
  padding: 8px;
}
</style>
</head>
<body>

<div class="center_div">
  <h1>Hello World!</h1>
</div>

</body>
</html>
```

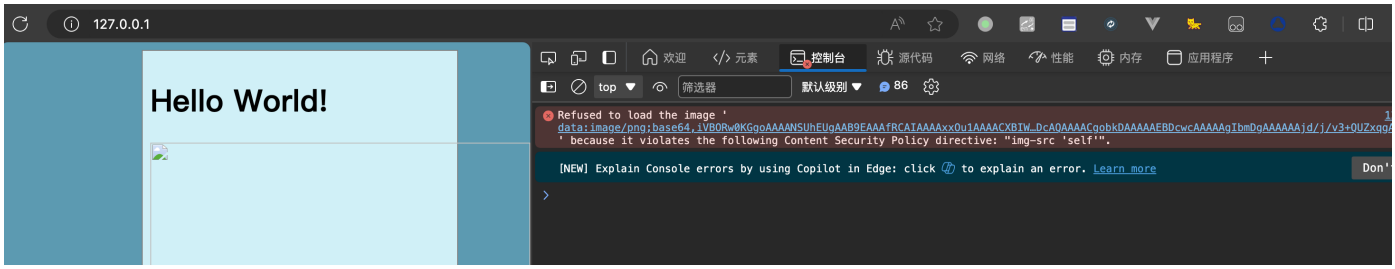
我们尝试在其中加入 `CSP` 策略，并通过 `data` 的形式添加一张图片

如果未设置 `CSP` ， `Edge` 浏览器默认是可以正常加载 `data` 格式的图片的



如果配置 `CSP` ， `img-src` 未设置 `data`

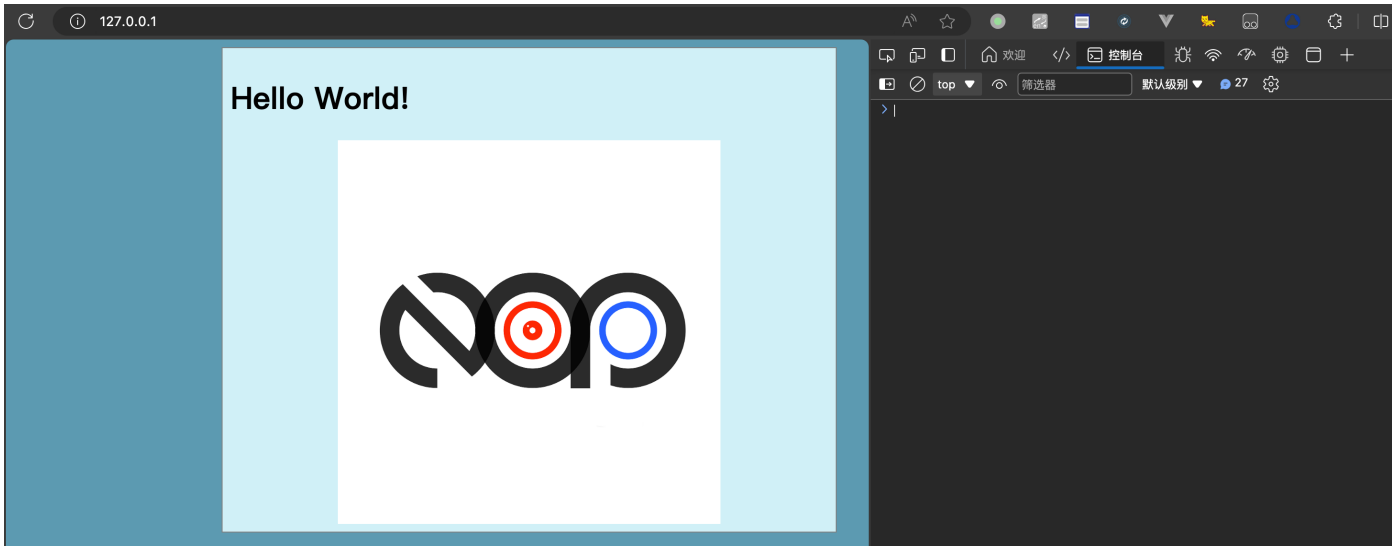
```
<meta http-equiv="Content-Security-Policy" content="img-src 'self';" />
```



加载失败

如果配置 `CSP` ， `img-src` 设置 `data`

```
<meta http-equiv="Content-Security-Policy" content="img-src 'self' data:"
/>
```



正常加载

2) unsafe-inline

`'unsafe-inline'` 用于指定允许使用内联样式（`inline styles`）和内联脚本（`inline scripts`）。在CSP中，内联样式指的是直接在HTML元素的 `style` 属性中编写的 `CSS` 代码，而内联脚本则是指在HTML文档中使用 `<script>` 标签直接编写或内嵌的 `JavaScript` 代码。

嗷，原来这个就是内联，似乎并不是一个好名字

如果禁止内联样式以及内联脚本，则会有效防止注入内联脚本式的 `XSS` 攻击

3) Nonce

`nonce` 这个词在加解密的内容中经常遇到，通常表示为一个随机值，是一个在特定上下文中仅使用一次的数字或字符串

在 `CSP` 中也差不多， `Nonce` 是一种在 `CSP` 中用于允许特定脚本或样式执行的临时凭证。

当启用 `Nonce-based CSP` 时，服务器会在生成 `HTML` 页面时为每个可信的内联脚本或样式标签分配一个随机生成的、一次性使用的值（ `Nonce` ）。这个 `Nonce` 随后被嵌入到相应的 `HTML` 标签中，并同时在 `CSP` 响应头中声明该 `Nonce` 可用于允许特定类型资源的加载。

`CSP` 内容

```
Content-Security-Policy: script-src 'nonce-randomlyGeneratedValueHere' ...
```

`html` 页面内容

```
<script nonce="randomlyGeneratedValueHere">
  // Your inline script code
</script>
```

通过 `Nonce` 这种方式，可以将要加载的内联 `JavaScript` 固定下来，也就是所谓的白名单，这是一个有趣的方法

4) Hash

`Hash` （在 `CSP` 中通常指的是 `Subresource Integrity, SRI` ）是一种基于资源内容散列值的安全机制，用于确保远程加载的脚本或样式文件在传输过程中没有被篡改

服务器为每个外部资源计算一个独特的散列值（通常使用 `SHA-256` 、 `SHA-384` 或 `SHA-512` 算法），并将该值以 `integrity` 属性的形式包含在 `HTML` 标签中。 `CSP` 则检查加载的资源是否与提供的散列值匹配。例如：

举例来说，假设你的网站使用了 `CSP` ，并且你希望确保加载的 `JavaScript` 文件没有被篡改。你可以在 `CSP` 策略中添加一个指令来验证 `JavaScript` 文件的完整性。

首先，你需要计算 `JavaScript` 文件的哈希值，并将其转换为 `Base64` 编码。例如，使用 `SHA256` 哈希算法计算文件的哈希值，并将其转换为 `Base64` 编码得到 `<hash-algorithm>-<base64-value>` 。

假设计算得到的哈希值为 `G0Ty2Cl5vV6z+1bC2Yk7F3V8G9QpN4R6S2Mx8Z0A3T1=`，那么你的 `CSP` 策略中的资源完整性校验指令可能如下所示：

```
script-src 'self' 'sha256-G0Ty2Cl5vV6z+1bC2Yk7F3V8G9QpN4R6S2Mx8Z0A3T1='
```

在页面中，引入方式如下

```
<script src="https://example.com/scripts/app.js" integrity="sha256-
G0Ty2Cl5vV6z+1bC2Yk7F3V8G9QpN4R6S2Mx8Z0A3T1=" nonce="random-nonce">
</script>
```

剩下的比较陌生的也确实不是很常见，如果遇到，可以参考下面的文档

> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/Sources

解析来的部分是关于指令的

---

0. default-src

`default-src` 指令用作其他 `CSP` 获取指令的 `callback`。对于以下缺少的每个指令，用户代理都会查找 `default-src` 指令并为其使用此值

简单来说就是部分指令的默认值

- `child-src`
- `connect-src`
- `font-src`
- `frame-src`
- `img-src`
- `manifest-src`

- `media-src`
- `object-src`
- `prefetch-src`
- `script-src`
- `script-src-elem`
- `script-src-attr`
- `style-src`
- `style-src-elem`
- `style-src-attr`
- `worker-src`

以下两种配置是等效的

```
Content-Security-Policy: default-src 'self'; script-src https://example.com
```

```
Content-Security-Policy: connect-src 'self';
                         font-src 'self';
                         frame-src 'self';
                         img-src 'self';
                         manifest-src 'self';
                         media-src 'self';
                         object-src 'self';
                         script-src https://example.com;
                         style-src 'self';
                         worker-src 'self'
```

1. base-uri

> `base-uri` 指令限制了可以应用于一个文档的 `<base>` 元素的 URL。假如指令值为空，那么任何 URL 都是允许的。如果指令不存在，那么用户代理会使用 `<base>` 元素中的值。
>
> https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Security-Policy/base-uri

注意，一些值对 `base-uri` 没有意义，例如关键字 `'unsafe-inline'` 和 `'strict-dynamic'`。

案例

```
<meta http-equiv="Content-Security-Policy" content="base-uri 'self'" />
```

允许页面基础URI（即页面自身所在的源）作为页面中所有相对URL的基础地址，仅允许加载与当前页面同源的资源

2. child-src

> `child-src` 指令定义了使用如 `<frame>` 和 `<iframe>` 等元素在加载 `web worker` 和嵌套浏览上下文时的有效来源。对于 `worker` 来说，不合规的请求会被用户代理当作致命的网络错误处理。
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/child-src

举例来说

```
Content-Security-Policy: child-src https://example.com/
```

这样 `<iframe>` 和 `Worker` 的地址如果不是 `https://example.com/` 就不会加载

3. connect-src

> `connect-src` 指令用于限制通过使用脚本接口加载的 URL
>
> https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Security-Policy/connect-src

- `<a> ping`
- `fetch()`
- `XMLHttpRequest`
- `WebSocket`
- `EventSource`
- `Navigator.sendBeacon()`

案例

```
Content-Security-Policy: connect-src https://example.com/
```

只允许以上标签和方法请求 `https://example.com/`

4. fenced-frame-src

> 这是一个实验性的 CSP
>
> fenced-frame-src 指令指定加载到 `<fencedframe>` 元素中的嵌套浏览上下文的有效源
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/fenced-frame-src

案例

```
Content-Security-Policy: fenced-frame-src https://example.com/
```

5. font-src

> 这个好理解，加载字体的源的限制
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/font-src

案例

```
Content-Security-Policy: font-src https://example.com/
```

此时以下代码就无法加载字体

```
<style>
  @font-face {
    font-family: "MyFont";
    src: url("https://not-example.com/font");
  }
  body {
    font-family: "MyFont";
  }
</style>
```

6. form-action

> **`form-action`** 指令能够限定当前页面中表单的提交地址
>
> https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Security-Policy/form-action

案例

```
<meta http-equiv="Content-Security-Policy" content="form-action 'none'" />
```

此时下面的代码就会被阻止

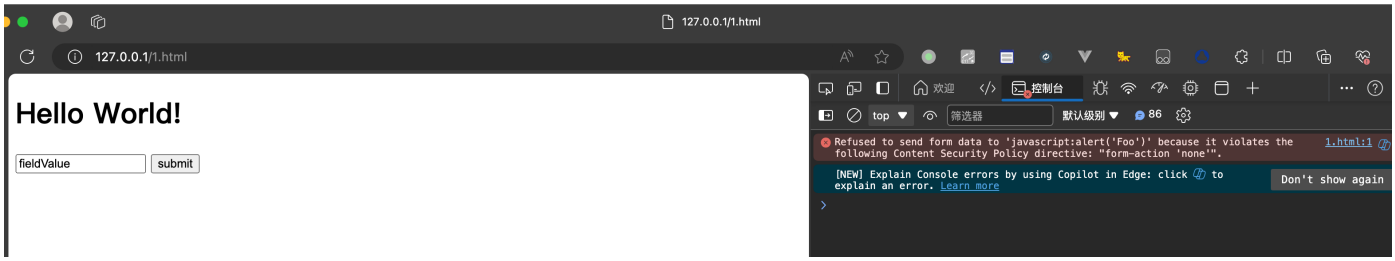
```
<meta http-equiv="Content-Security-Policy" content="form-action 'none'" />

<form action="javascript:alert('Foo')" id="form1" method="post">
  <input type="text" name="fieldName" value="fieldValue" />
  <input type="submit" id="submit" value="submit" />
</form>

<!-- Error: Refused to send form data because it violates the following
Content Security Policy directive: "form-action 'none'". -->
```

之前没有测试过 `action` 是否可以执行 `javascript` 代码，看起来像是可以的

测试发现还真的可以，设置了 `CSP` 策略后就无法执行了



7. frame-ancestors

> `frame-ancestors` 指令指定了一个可以包含 `<frame>`、`<iframe>`、`<object>`、`embed` 等元素的有效父级。
>
> https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors

这个 `CSP` 专门用于控制哪些网页（即祖先帧）可以嵌套当前页面。它的作用在于防止点击劫持（`Clickjacking`）等攻击手段，确保网页内容不会在未经授权的上下文中被显示。

8. frame-src

> `frame-src` 指定了可以被 `<frame>` 和 `<iframe>` 嵌套浏览上下文加载的有效 URL
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-src

案例

```
Content-Security-Policy: frame-src https://example.com/
```

此时，下面的 `<iframe>` 标签加载过程中就会被阻止

```
<iframe src="https://not-example.com/"></iframe>
```

9. img-src

> 没啥说的，图片和图标允许加载的地址
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/img-src

10. manifest-src

> `manifest-src` 是CSP中的一个特定指令，专门用来指定允许加载 Progressive Web App (PWA) 清单文件（即 `manifest.json` 文件）的来源。当一个网站希望将其Web应用转化为PWA时，会创建一个 `manifest.json` 文件，该文件包含了关于PWA的一些元数据，如名称、图标、启动画面、主题颜色、显示模式（全屏、最小化窗口等）、服务工作线程（Service Worker）地址等。浏览器通过读取这个清单文件，能够按照PWA的标准对网站进行配置，使其具有类似原生应用的体验，比如添加至主屏幕、离线访问能力等。
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/manifest-src

11. media-src

> 和 `img-src` 类似，可以被 `<audio>` 和 `<video>` 元素加载的有效源URL
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/media-src

12. object-src

> 指定了 `<object>` 和 `<embed>` 元素的有效源URL
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/object-src

13. report-to

> `report-to` 是Content Security Policy (CSP) 中的一个扩展指令，用于指定一个报告端点（reporting endpoint），该端点接收浏览器发送的CSP违规报告。当浏览器检测到页面上的内容加载或执行行为违反了当前设置的CSP策略时，通常会阻止这些不合规的操作以保护用户安全。同时，浏览器会生成一份详细的违规报告，并按照 `report-to` 指令指定的方式将其发送给指定的服务器端点。
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/report-to

很少见到开启这个操作的 `CSP`

14. upgrade-insecure-requests

> 这个选项用于将网页中 `HTTP` 协议的资源加载全部提升为 `HTTPS`，用于提升安全性
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/upgrade-insecure-requests

案例

```
Content-Security-Policy: upgrade-insecure-requests;
```

一旦将上述头部设置在计划从 `HTTP` 迁移到 `HTTPS` 的 `example.com` 域名上，非跳转（`non-navigational`）的不安全资源请求会自动升级到 `HTTPS`（包括第当前域名以及第三方请求）。

```
<img src="http://example.com/image.png" />
<img src="http://not-example.com/image.png" />
```

这些 `URL` 在请求发送之前都会被改写成 `HTTPS`，也就意味着不安全的请求都不会发送出去。注意，如果请求的资源在 `HTTPS` 情况下不可用，则该请求将失败，其也不能回退到 `HTTP`。

15. sandbox

> 这个和 `<iframe>` 的 `sandbox` 很像，主要是对于一些行为进行限制，它不支持 `<meta>` 在网页引入的方式，只能通过返回包的包头中指定
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/sandbox

```
Content-Security-Policy: sandbox;
Content-Security-Policy: sandbox <value>;
```

其中 `value` 可以取以下值

- `allow-forms` 允许嵌入式浏览上下文提交表单。如果未使用此关键字，则不允许此操作。
- `allow-modals` 允许嵌入式浏览上下文打开模态窗口。
- `allow-orientation-lock` 允许嵌入式浏览上下文禁用锁定屏幕方向的功能。
- `allow-pointer-lock` 允许嵌入式浏览上下文使用Pointer Lock API (en-US)。
- `allow-popups` 允许弹出窗口（像 `window.open`，`target="_blank"`，`showModalDialog`）。如果未使用此关键字，则该功能将无提示失败。
- `allow-popups-to-escape-sandbox` 允许沙盒文档打开新窗口而不强制沙盒标记。例如，这将允许安全地沙箱化第三方广告，而不会对登陆页面施加相同的限制。
- `allow-presentation` 允许嵌入器控制 `iframe` 是否可以启动演示会话。

- `allow-same-origin` 允许将内容视为来自其正常来源。如果未使用此关键字，则嵌入的内容将被视为来自唯一来源。
- `allow-scripts` 允许嵌入式浏览上下文运行脚本（但不创建弹出窗口）。如果未使用此关键字，则不允许此操作。
- `allow-top-navigation` 允许嵌入式浏览上下文将内容导航（加载）到顶级浏览上下文。如果未使用此关键字，则不允许此操作。

16. script-src

> script-src 指令指定 JavaScript 的有效源。
>
> 这不仅包括直接加载到 `<script>` 元素中的URL，还包括可以触发脚本执行的内联脚本事件处理程序（`onclick`）和 `XSLT stylesheets` 样式表。
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src

17. script-src-attr

> script-src-attr指令指定 JavaScript 内联事件处理程序的有效源
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src-attr

相对于 `script-src` ，这个指令只针对内联 `JavaScript` 执行，包括 `onclick` 这种

18. script-src-elem

> script-src-elem指令指定 `<script>` 元素的有效源
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src-elem

19. style-src

> 指定了样式表的有效源
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/style-src

20. style-src-attr

> 为 Dom 中的内联样式表指定了有效源
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/style-src-attr

21. style-src-elem

> 为 `<style>` 和 `<link>` 元素指定了有效源
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/style-src-elem

22. require-trusted-types-for

> 实验性的技术
>
> 这个主要是对抗 `Dom-Base XSS` 的，检查传递给 `Element.innerHTML` 的内容，阻止 `XSS`
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for

`require-trusted-types-for` 和 `trusted-types` 是 `Content Security Policy (CSP)` 中与 `Trusted Types` 相关的两个指令，旨在加强网页对 `DOM-based Cross-Site Scripting (XSS)` 攻击的防御。`Trusted Types` 是一种高级的防御机制，用于确保浏览器在处理可导致 `XSS` 漏洞的危险 `API`（如 `innerHTML`、`document.write()` 等）时，只能接受经过验证的、安全的值。

`require-trusted-types-for` 用于指定哪些 `DOM API` 必须使用 `Trusted Types`。

当你在 `CSP` 策略中声明此指令时，浏览器将强制在指定的上下文中使用 `Trusted Types`，否则相关的 `DOM` 操作将会失败。这可以防止未经验证的字符串直接插入到可能导致 `XSS` 的 `API` 中。

案例

```
Content-Security-Policy: require-trusted-types-for 'script';
```

23. trusted-types

> 实验性技术
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types

`trusted-types` 是CSP指令的另一部分，用于定义允许创建的 `Trusted Types` 策略以及可信赖的类型创建器 `(Trusted Type policy creators)` 。策略创建器是用于生成具体 `Trusted Types` 对象的函数或类，它们负责验证即将注入到 `DOM` 中的内容，确保其安全。

使用Trusted Types的基本流程如下：

1. **定义策略**：在 `JavaScript` 中编写策略创建器，它们包含对输入内容的安全检查逻辑，确保只有符合安全规范的值才能通过验证。
2. **在CSP中声明策略**：如上所述，在CSP响应头中使用 `trusted-types` 指令列出允许使用的策略创建器名称。
3. **应用策略**：在实际代码中，使用已声明的策略创建器生成 `Trusted Types` 对象（如 `TrustedHTML`、`TrustedScriptURL` 等），然后将这些对象赋值给相应的 `DOM` 属性或方法。

24. worker-src

> **`worker-src`** 指令指定了 `Worker`、`SharedWorker` 或 `ServiceWorker` 脚本的有效来源。

> https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Security-Policy/worker-src

案例

```
Content-Security-Policy: worker-src https://example.com/
```

以下 `Worker`、 `SharedWorker`、 `ServiceWorker` 被阻止，无法加载

```
<script>
  let blockedWorker = new Worker("data:application/javascript,…");
  blockedWorker = new SharedWorker("https://not-example.com/");
  navigator.serviceWorker.register("https://not-example.com/sw.js");
</script>
```

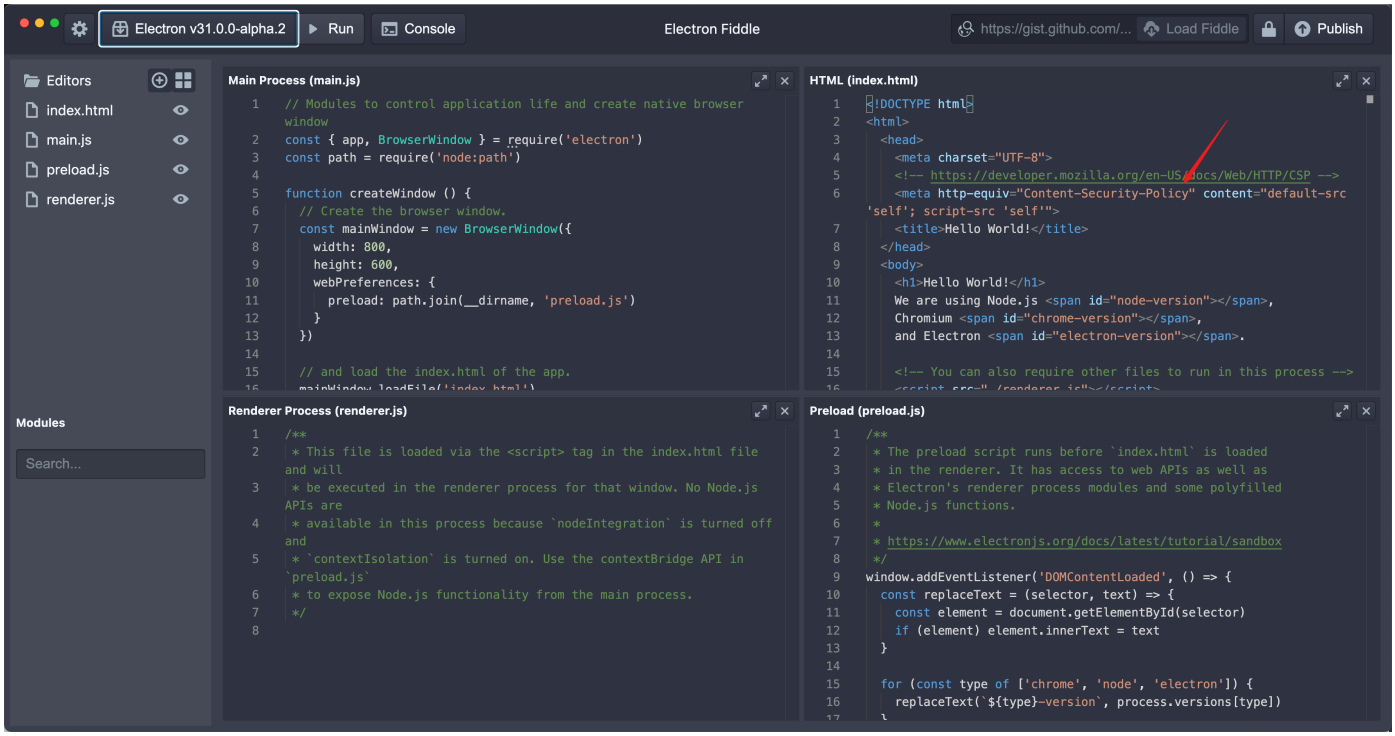
> 参考文章
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/Sources
>
> https://websec.readthedocs.io/zh/latest/vuln/xss/csp.html
>
> https://www.ruanyifeng.com/blog/2016/09/csp.html

0x03 Electron 中使用 CSP

> https://www.electronjs.org/zh/docs/latest/tutorial/security#7-content-security-policy%E5%86%85%E5%AE%B9%E5%AE%89%E5%85%A8%E7%AD%96%E7%95%A5

在 `Electron` 中也是差不多的方式，

本地加载文件创建窗口可以通过在页面中添加 `<meta>` 标签的形式设置 `CSP`



远程加载资源创建窗口还可以通过修改 `HTTP(s)` 返回头的方式设置 `CSP`

```
// main.js
const { session } = require('electron')

session.defaultSession.webRequest.onHeadersReceived((details, callback) =>
{
  callback({
    responseHeaders: {
      ...details.responseHeaders,
      'Content-Security-Policy': ['default-src \'none\'']
    }
  })
})
```

这种方法对于 `file://` 加载资源的时候无效

0x04 CSP 可以突破同源策略吗?

异想天开一下，如果设置了 `CSP` ，将非同源的网站设置为有效源，可以无视同源策略吗?

`index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="Content-Security-Policy" content="script-src
http://192.168.31.26/" />
  <title>Content-Security-Policy Test</title>
</head>
<body>
  <h1>Content-Security-Policy Test</h1>

  <!-- 跨源脚本 -->
  <img src="x" onerror="import(unescape('http://192.168.31.26/1.js'))" />
```

```
</body>
</html>
```

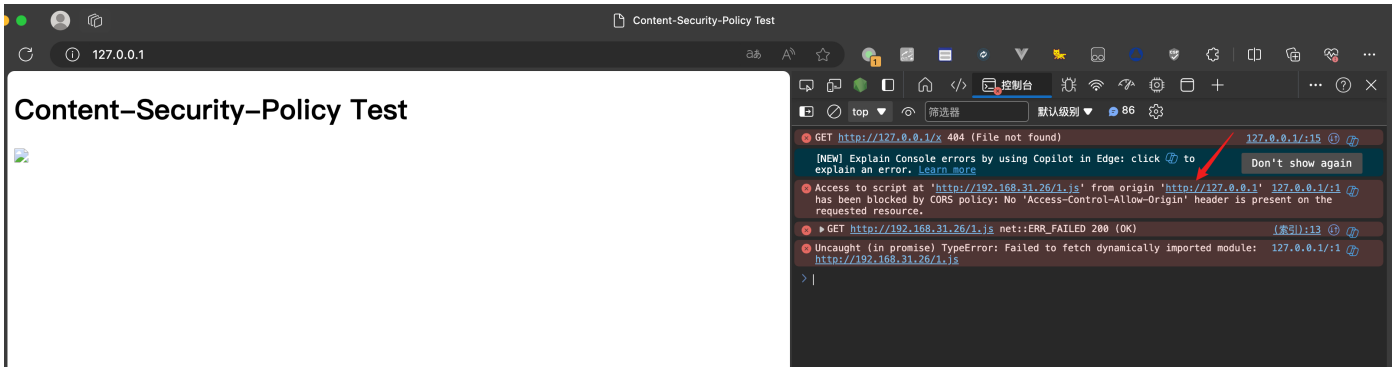
```
→  tmp cat 1.js
console.log("payload runs")
→  tmp ifconfig
enp0s5: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.31.26  netmask 255.255.255.0  broadcast 192.168.31.255
        inet6 fe80::40ea:fa59:998c:b4ac  prefixlen 64  scopeid 0x20<link>
        ether (                 txqueuelen 1000  (Ethernet)
        RX packets 1879  bytes 1383644 (1.3 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1296  bytes 230476 (225.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 397  bytes 33846 (33.0 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 397  bytes 33846 (33.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

→  tmp cat 1.js
console.log("payload runs")
→  tmp sudo python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

访问 `index.html`

```
→  tmp sudo python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.31.216 - - [24/Apr/2024 22:55:26] "GET /1.js HTTP/1.1" 200 -
192.168.31.216 - - [24/Apr/2024 22:55:33] "GET /1.js HTTP/1.1" 200 -
192.168.31.216 - - [24/Apr/2024 22:55:34] code 404, message File not found
192.168.31.216 - - [24/Apr/2024 22:55:34] "GET /favicon.ico HTTP/1.1" 404 -
```



很可惜，虽然收到了请求，但是没有加载成功

这也很容易理解， `CSP` 其实是另外一层的安全策略，它和同源策略独立的

0x05 CSP 绕过

其实没有什么绕过，无非就是配置得不是很合理或被允许的对象不安全，大家钻漏洞而已，没什么意思，但还是贴一些链接进来

当然，这里我还是要再强调一些 `JSONP` 的安全问题，网络上大部分 `JSONP` "服务器"(被访问方)的配置是不安全的，很容易成为绕过 `CSP` 的利器

> https://book.hacktricks.xyz/v/cn/pentesting-web/content-security-policy-csp-bypass
>
> https://xz.aliyun.com/t/7372?time__1311=n4%2BxnD0GDtYmqAK0QtDsA3xCqh4jruxiKoq4ex&alichlgref=https%3A%2F%2Fwww.google.com%2F
>
> https://geeeez.github.io/posts/xss%E4%B9%8BCSP%E7%9A%84%E5%8E%9F%E7%90%86%E5%92%8C%E7%BB%95%E8%BF%87/

给大家推荐一个自动分析 `CSP` 策略的网站，它们也有对应的浏览器插件

> https://csp-evaluator.withgoogle.com/

输入 `CSP` 策略字符串或目标 `URL` 即可

Content Security Policy

Sample unsafe policy Sample safe policy

```
object-src 'none';
base-uri 'self';
script-src 'nonce-xV2P02Bm1-TxZA4PWs89rw' 'strict-dynamic' 'report-sample' 'unsafe-eval' 'unsafe-inline' https:
    http:;
report-uri https://csp.withgoogle.com/csp/gws/other-hp
```

CSP Version 3 (nonce based + backward compatibility checks)

CHECK CSP

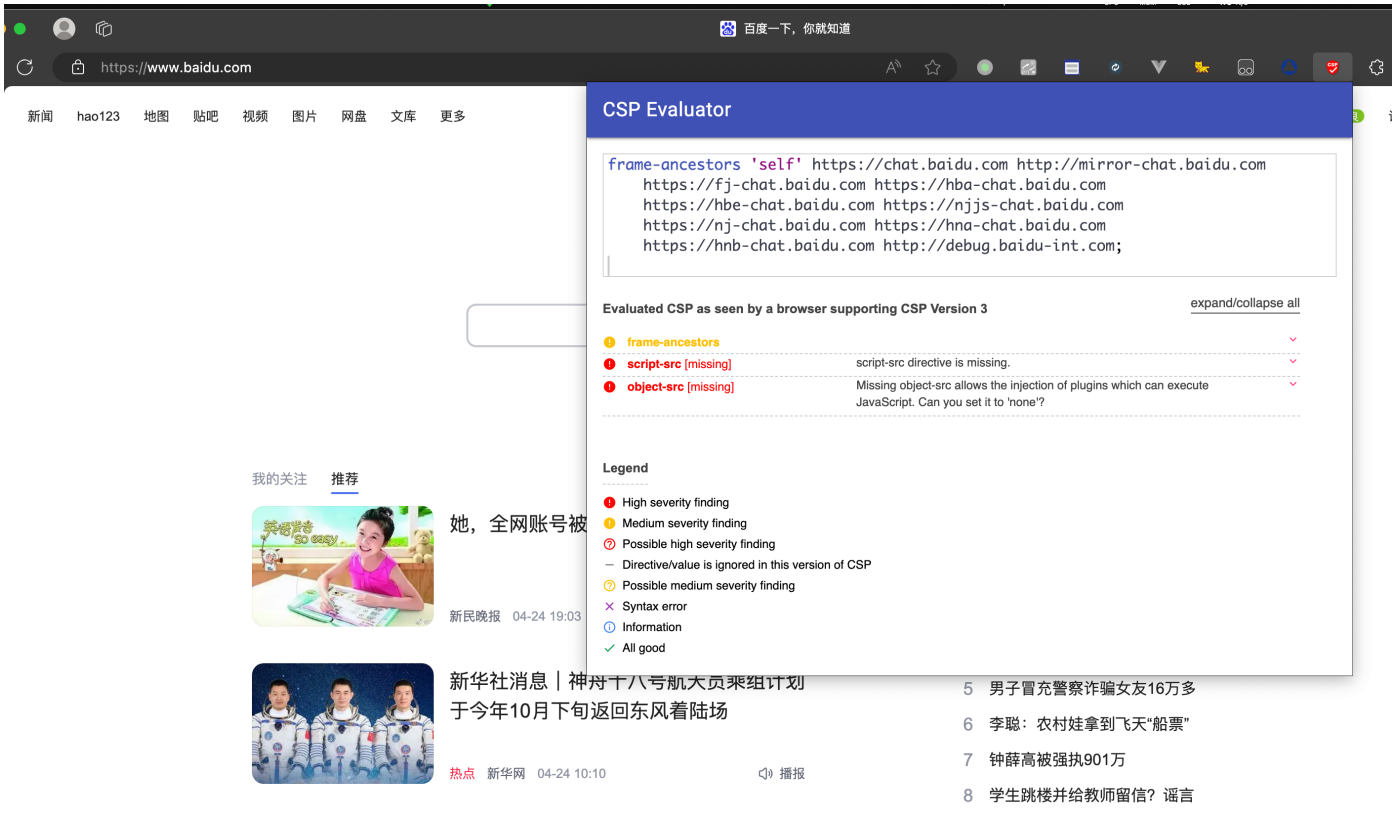
Evaluated CSP as seen by a browser supporting CSP Version 3

expand/collapse all

- ✓ object-src
- ✓ base-uri
- ? script-src
- ✓ report-uri
- ⓘ require-trusted-types-for [missing] — Consider requiring Trusted Types for scripts to lock down DOM XSS injection sinks. You can do this by adding "require-trusted-types-for 'script'" to your policy.

Legend

- High severity finding
- Medium severity finding
- Possible high severity finding
- Directive/value is ignored in this version of CSP
- Possible medium severity finding
- Syntax error
- Information
- All good



浏览器插件更好用一些

0x06 总结

`Electron` 中的 `CSP` 的意义比网站中更加重要，在防 `XSS` 方面真的是一把好手，但也会因为一些配置影响正常功能，或者给正常功能开发带来困难，这就又到方便与安全之间的权衡了