

Electron Security

新窗口创建



NOP Team



新窗口创建问题 | Electron 安全

0x00 简介

大家好，今天和大家讨论的是新窗口创建问题，通常来说，我们打开一个 `Electron` 程序，映入我们眼帘的就是主窗口，基本上是通过 `BrowserWindow` 创建的

如果我们点击某个功能，突然在当前窗口之外跳出来一个窗口，那就是一个新窗口创建了

在 `Electron` 中，一个新窗口创建背后都意味着存在对应的管理操作，这种管理可能可以让窗口赋予非凡的权限，例如执行 `Node.js`

创建新窗口分为两种，一种是主进程创建的，一种是渲染进程创建的，我们今天会针对两种情况进行讨论

参考文章

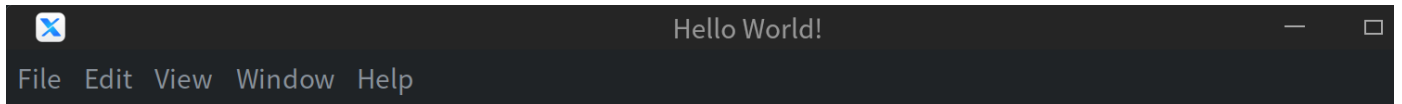
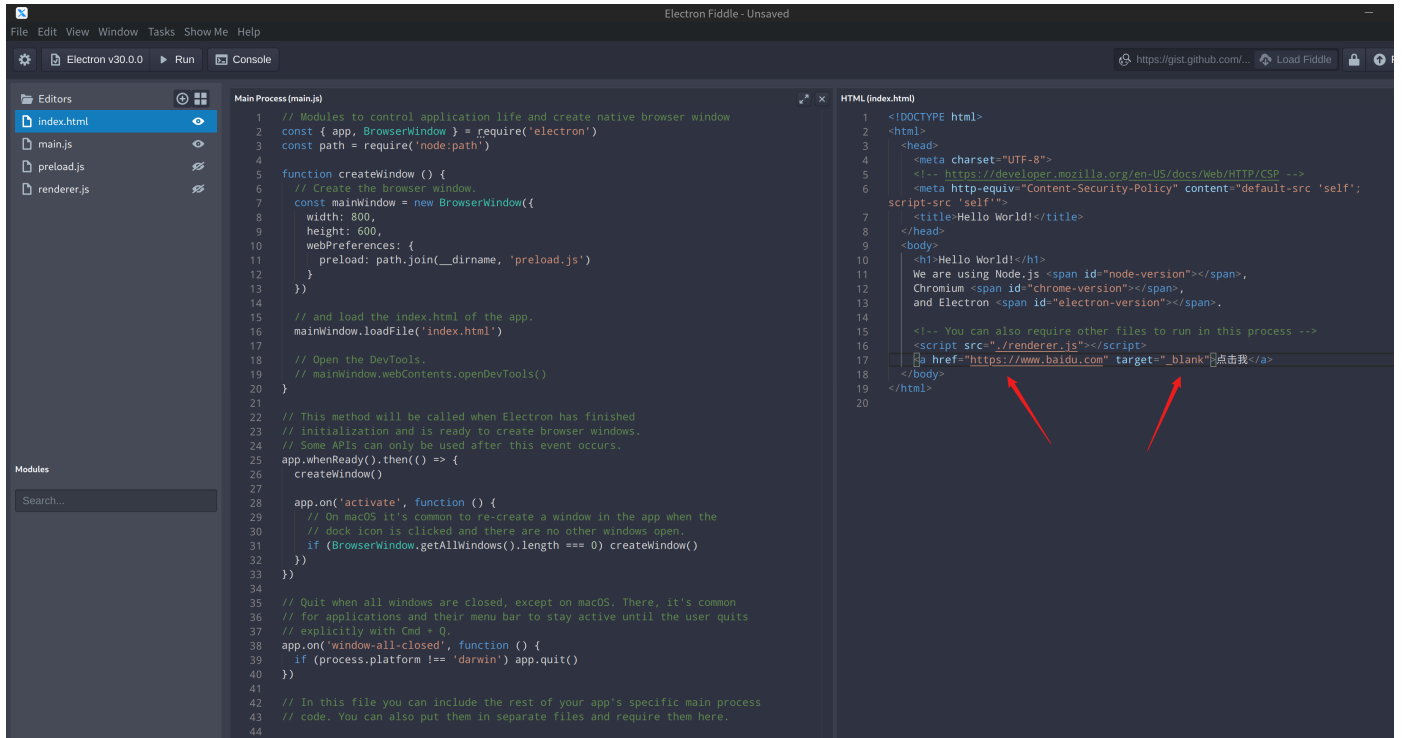
<https://www.electronjs.org/docs/latest/tutorial/security#14-disable-or-limit-creation-of-new-windows>

0x01 哪些情况下会创建新窗口

在之前的章节中，我们尝试过使用 `BrowserWindow`、`BaseWindow` 在主进程中创建窗口，同时我们尝试过在渲染进程中通过 `window.open` 创建新的窗口

除此之外还有两个特例，就是 `a` 标签和 `form` 标签，当 `a` 标签的 `target` 属性被设置为 `_blank` 时，点击标签会创建新窗口

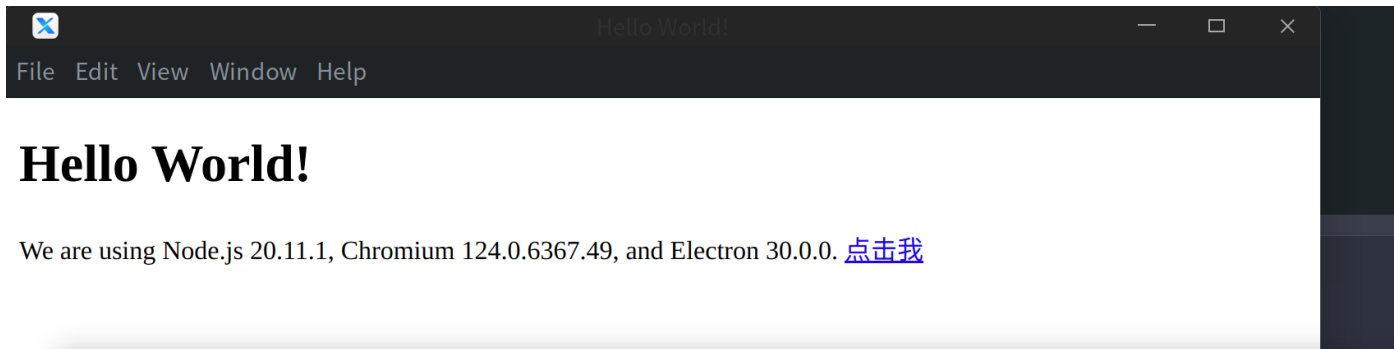
当 `form` 标签渲染的表达被提交时，也会打开新窗口



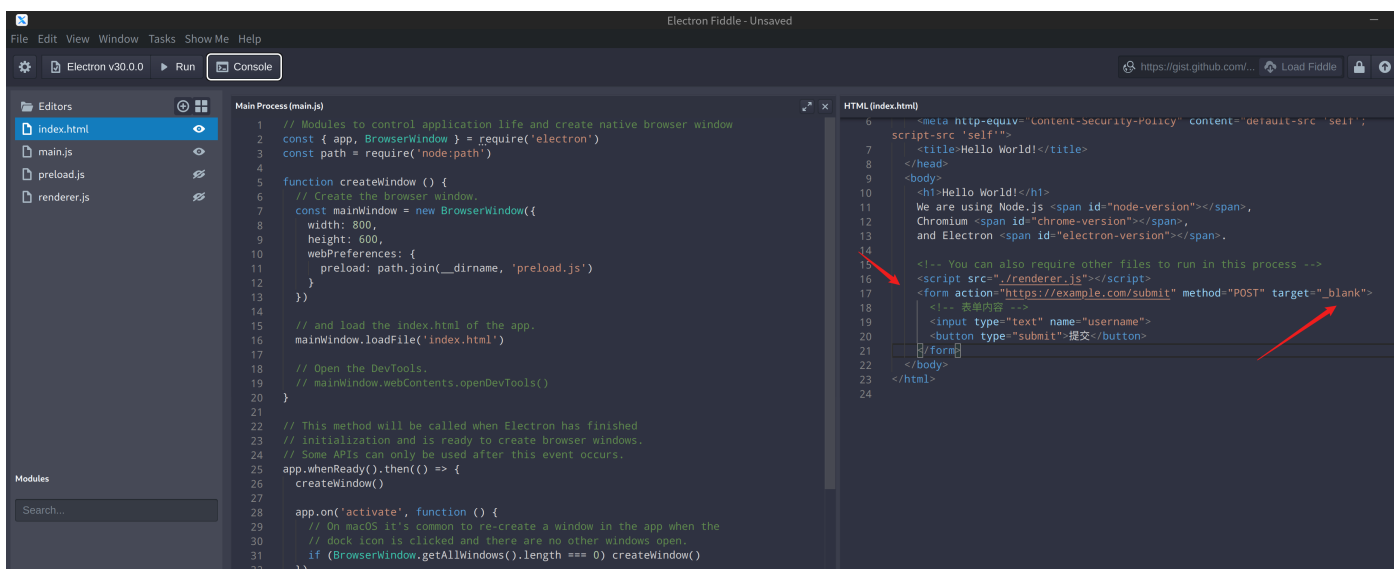
Hello World!

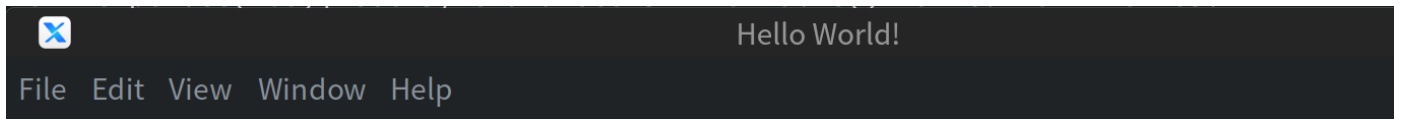
We are using Node.js 20.11.1, Chromium 124.0.6367.49, and Electron 30.0.0. [点击我](#)





We are using Node.js 20.11.1, Chromium 124.0.6367.49, and Electron 30.0.0. [点击我](#)





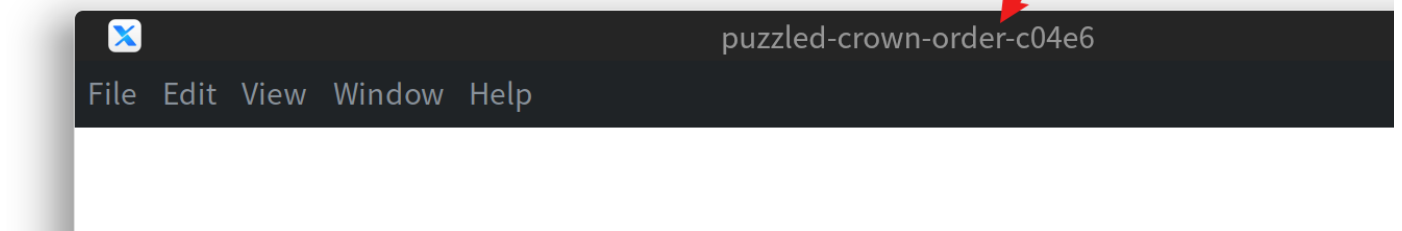
Hello World!

We are using Node.js 20.11.1, Chromium 124.0.6367.49, and Electron 30.0.0.

Hello World!

We are using Node.js 20.11.1, Chromium 124.0.6367.49, and Electron 30.0.0.

除此之外的 `alert` 等创建的弹窗就不在讨论的范畴了

<https://www.electronjs.org/zh/docs/latest/api/window-open>

0x02 创建新窗口带来的危害

我们还是按照两类来说，主进程创建新窗口和渲染进程创建新窗口

主进程创建新窗口基本上都是固定的窗口，所以如果说危害，除了窗口安全配置不合理，权限分配不合理之外，如果窗口创建的配置参数中存在用户可控制的情况(这里主要是窗口加载的内容以及安全配置)，可能带来一些危害

渲染进程创建新窗口在之前的文章中出现过绕过安全限制的情况(`iframe + window.open`)，但 `window.open` 不仅仅是绕过安全限制那么简单，其实在 `Electron` 中 `window.open` 是可以配置安全策略的，也就是说有可能执行 `Node.js` 的

`window.open` 打开的窗口配置的优先级为（向下递减）

- 在 `webContents.setWindowOpenHandler` 中指定的选项。
- 从父窗口继承安全相关的 `webPreferences`
- 从 `window.open()` 的 `features` 字段传入的选项

注意，`webContents.setWindowOpenHandler` 有最终解释权和完全权限，因为它是在主进程中调用的。

而且 `window.open` 也是本地文件读取漏洞的范畴内的工具之一，这个会在这篇文章中简单提到一嘴，后期出单独文章

所以今天的主角其实是 `window.open`

0x03 window.open 介绍

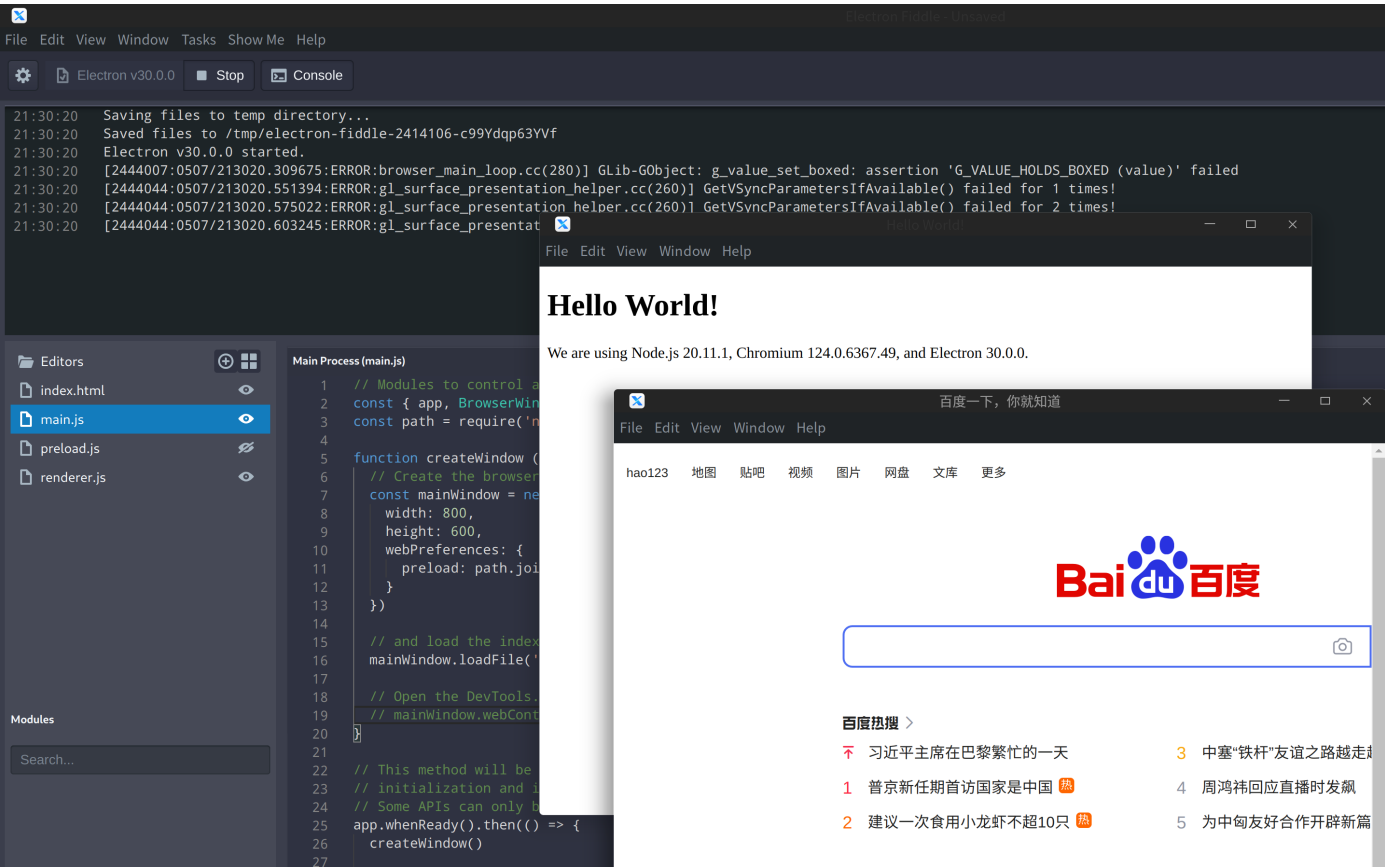
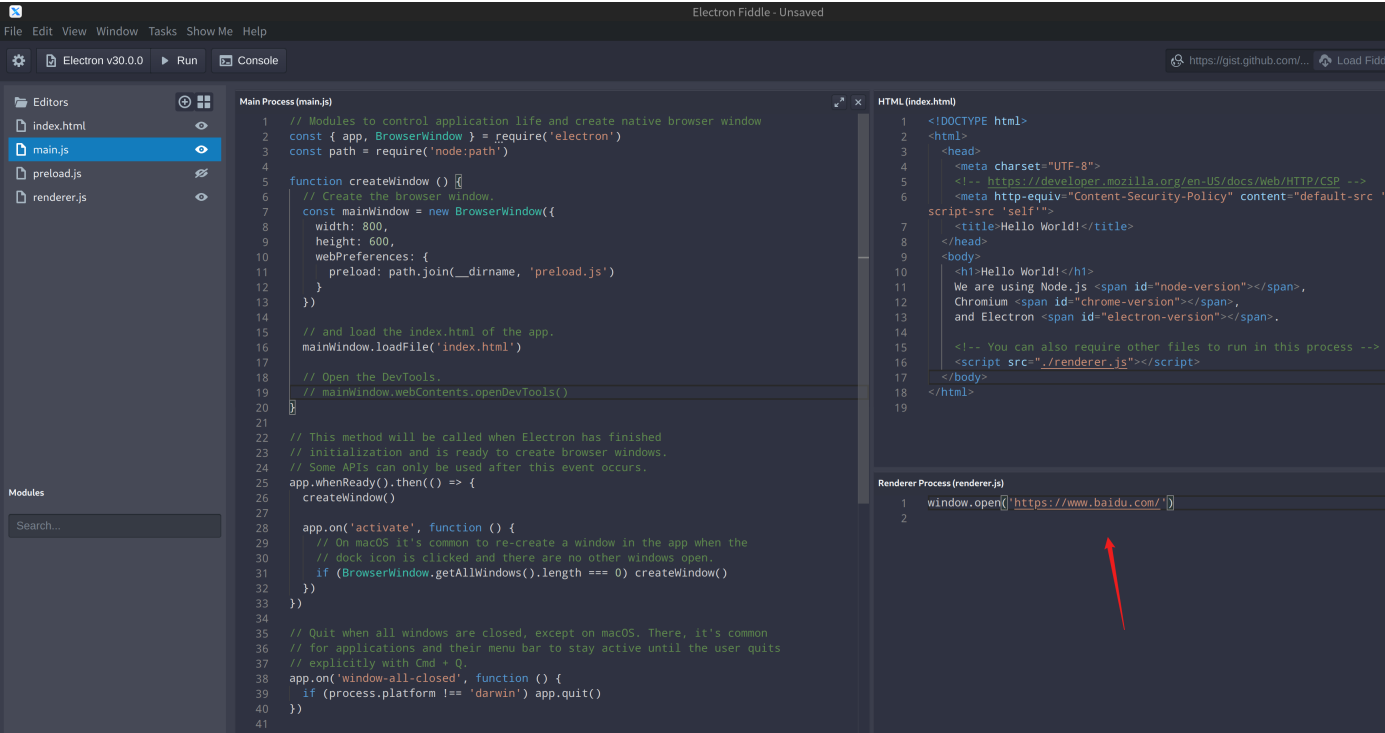
```
window.open(url[, frameName][, features])
```

其中各个参数解释如下

- url
- frameName 名称
- features 特性

渲染进程中的 `window.open` 其实相对 web 原本的 `window.open` 是做了一些改动的，下面我们一点一点解析

1. 效果测试

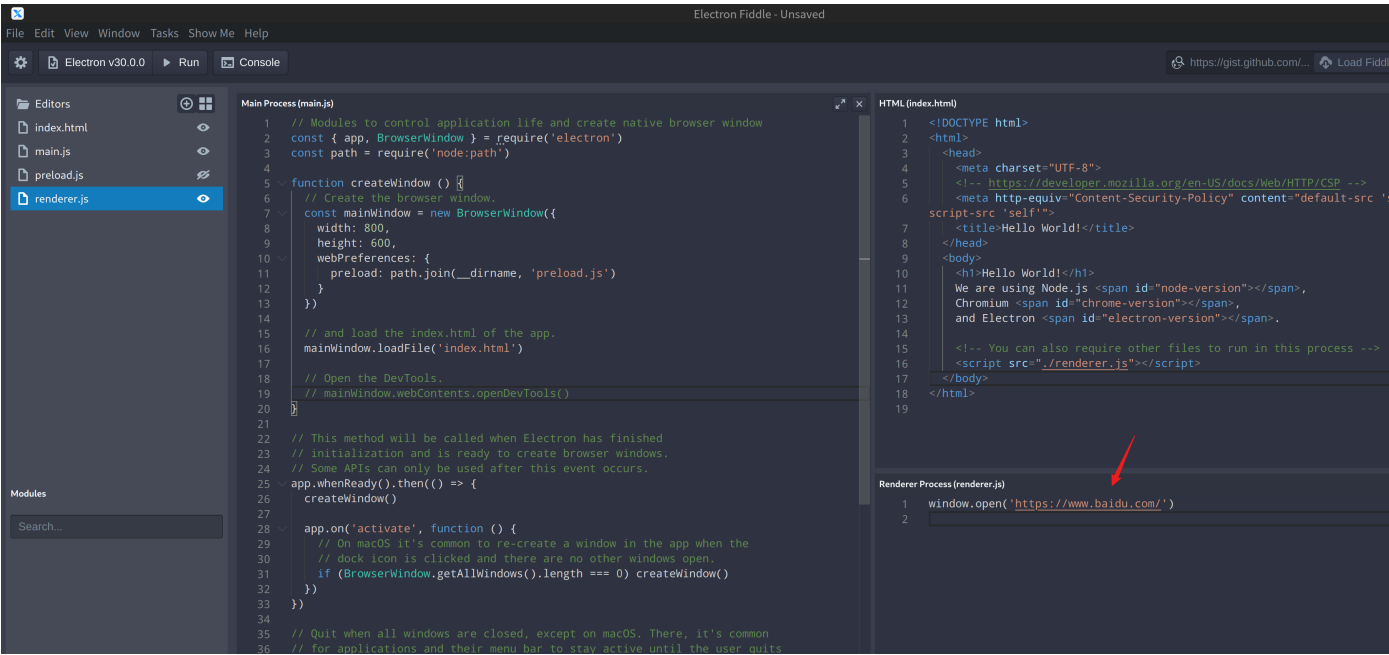


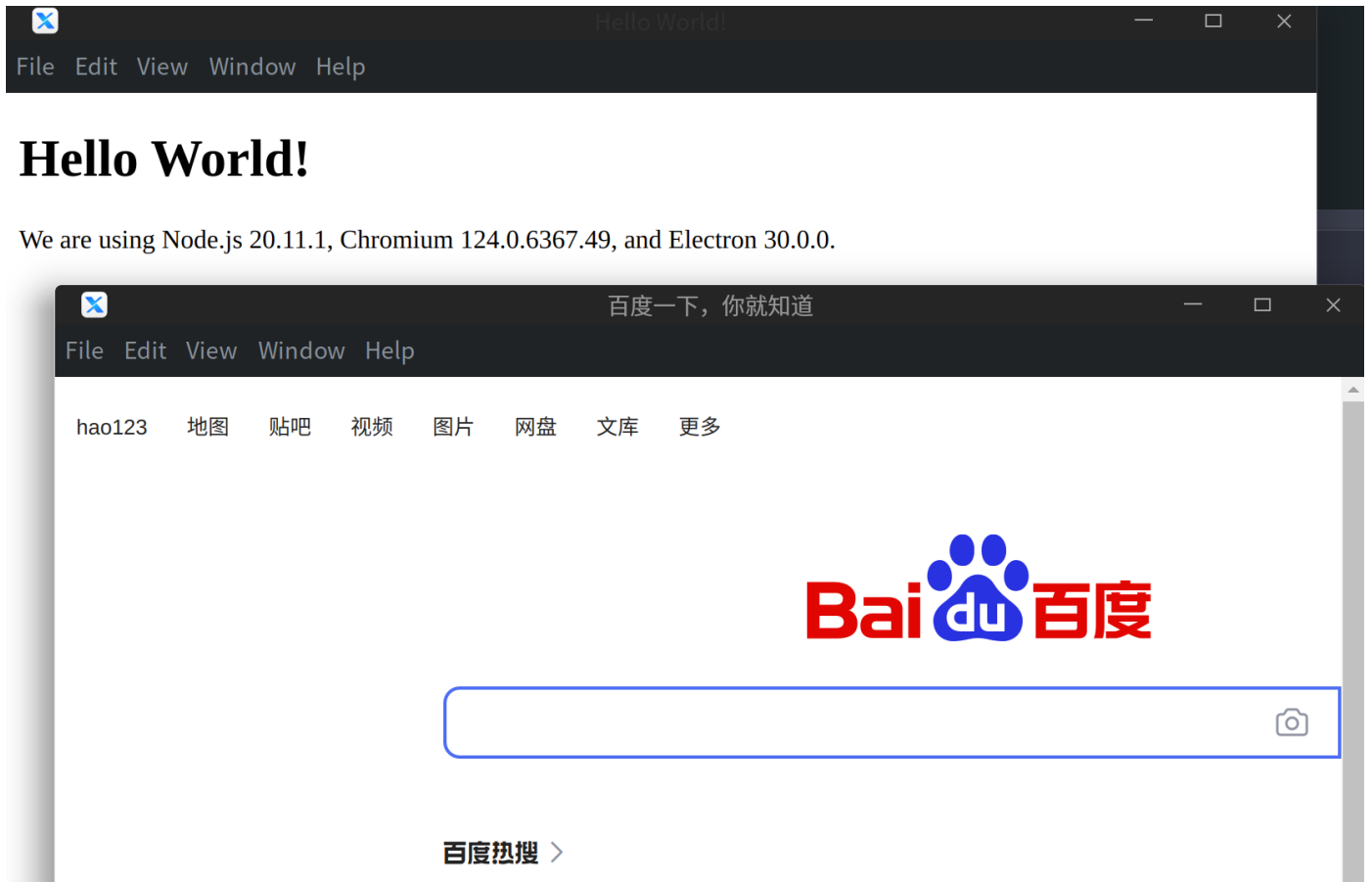
2. url

一个字符串，表示要加载的资源的 URL 或路径。如果指定空字符串（""）或省略此参数，则会在目标浏览上下文中打开一个空白页

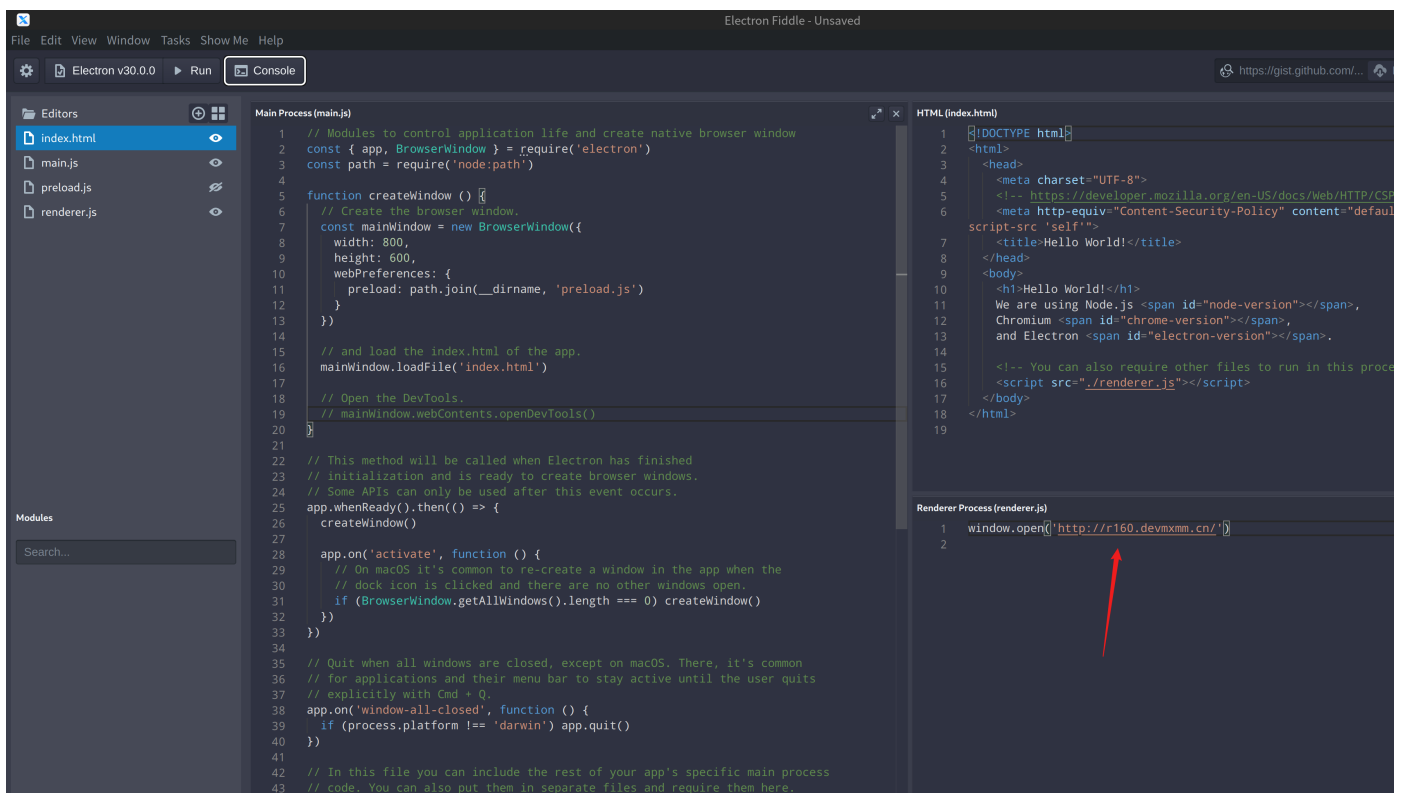
在 Electron 官网中对 url 参数并没有特别多的描述，但是我们搞安全的肯定得测试一下，了解其风险

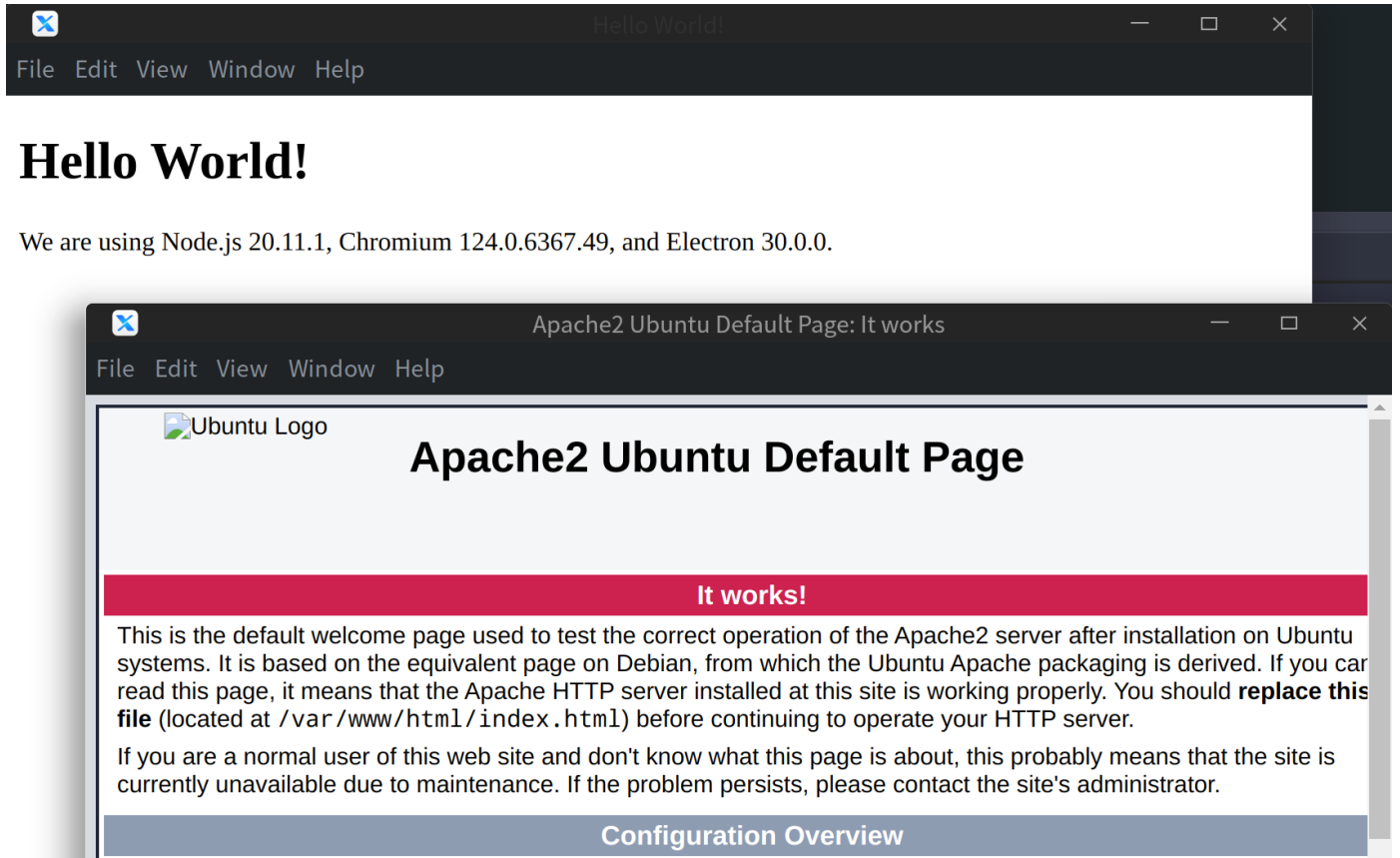
1) http(s) 网址



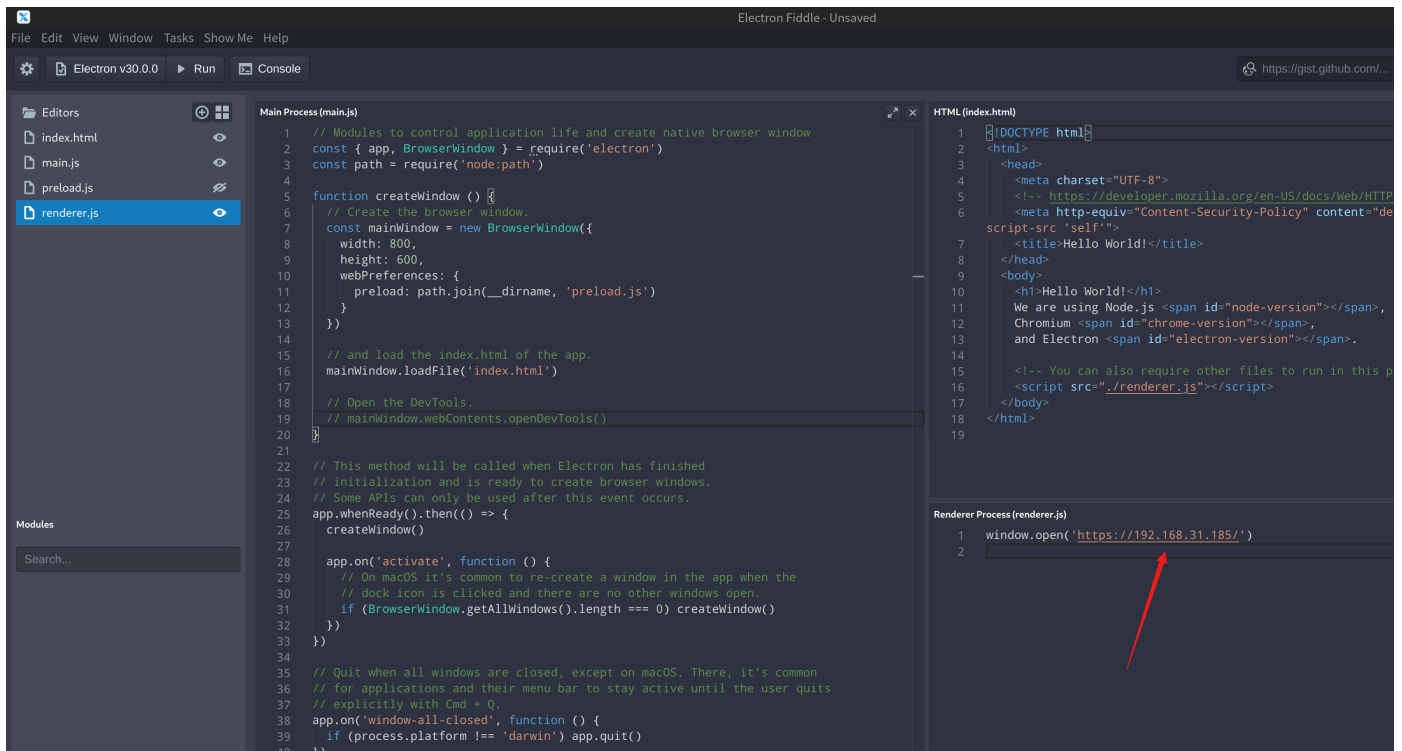


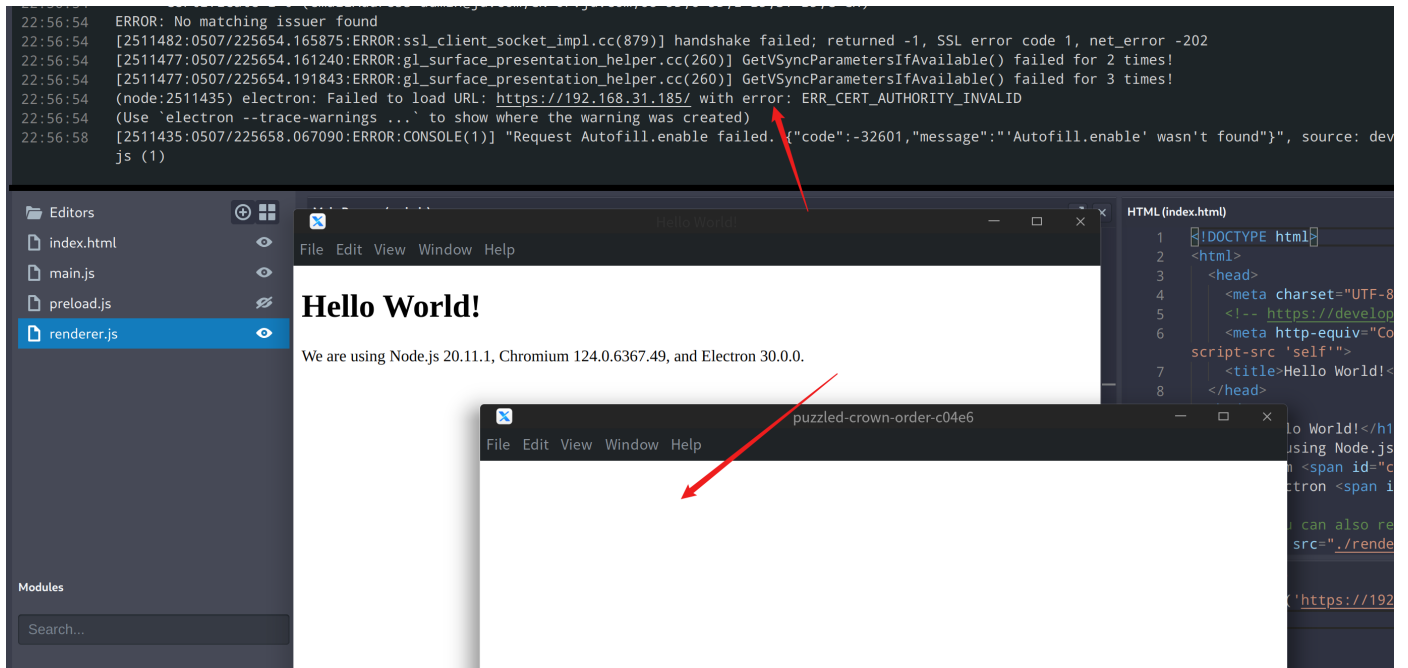
打开 https 的网址没问题





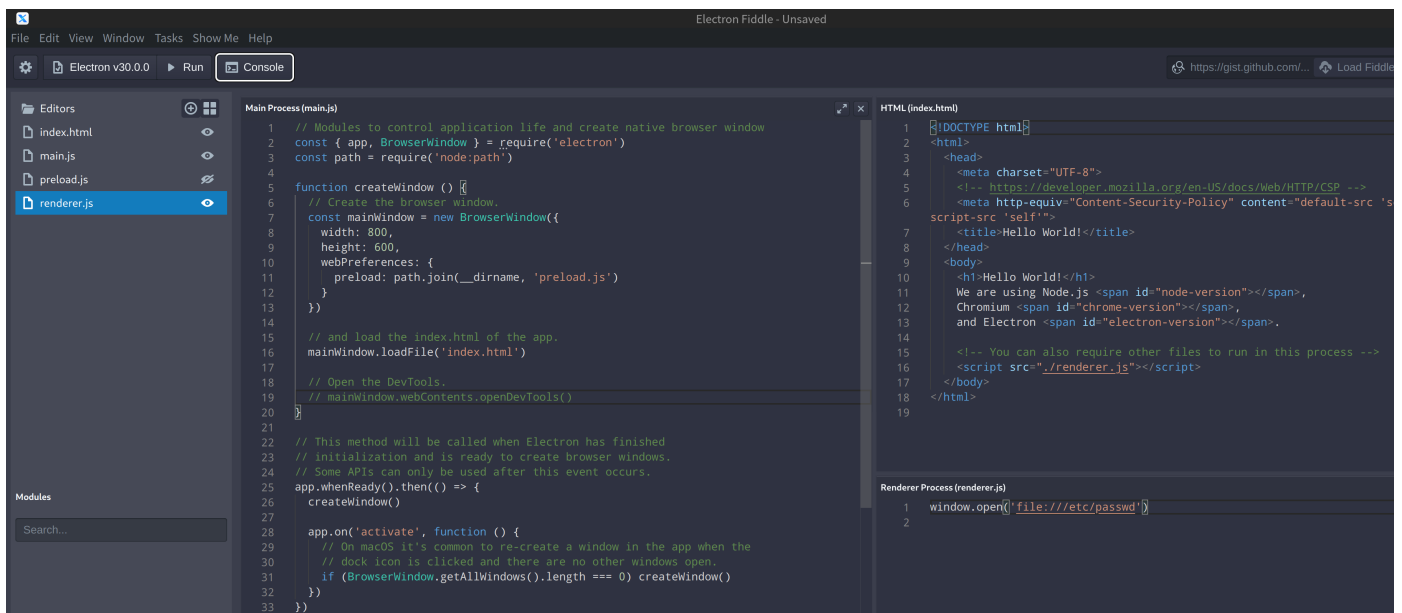
打开 http 网站没有问题

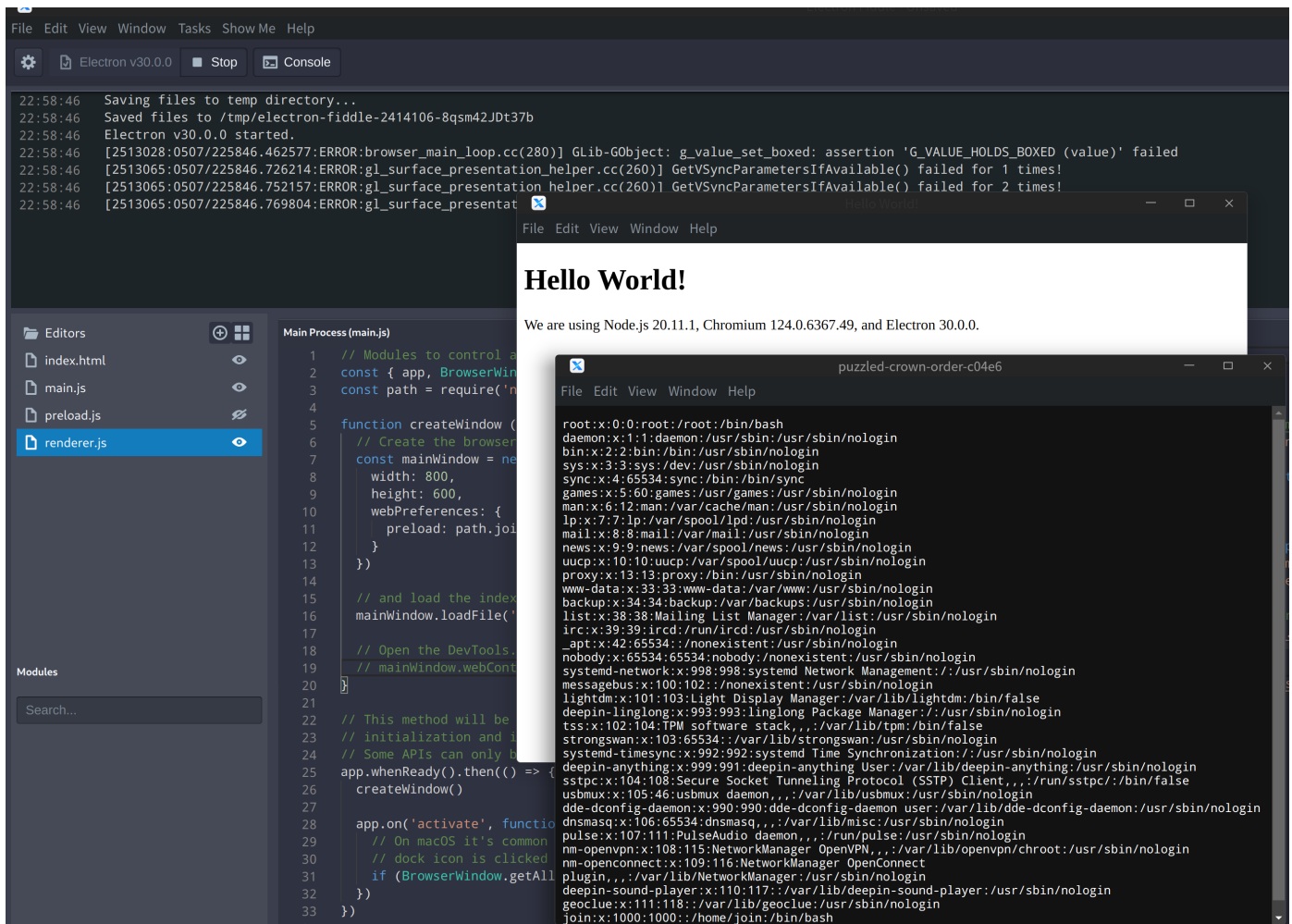




自签名证书不行

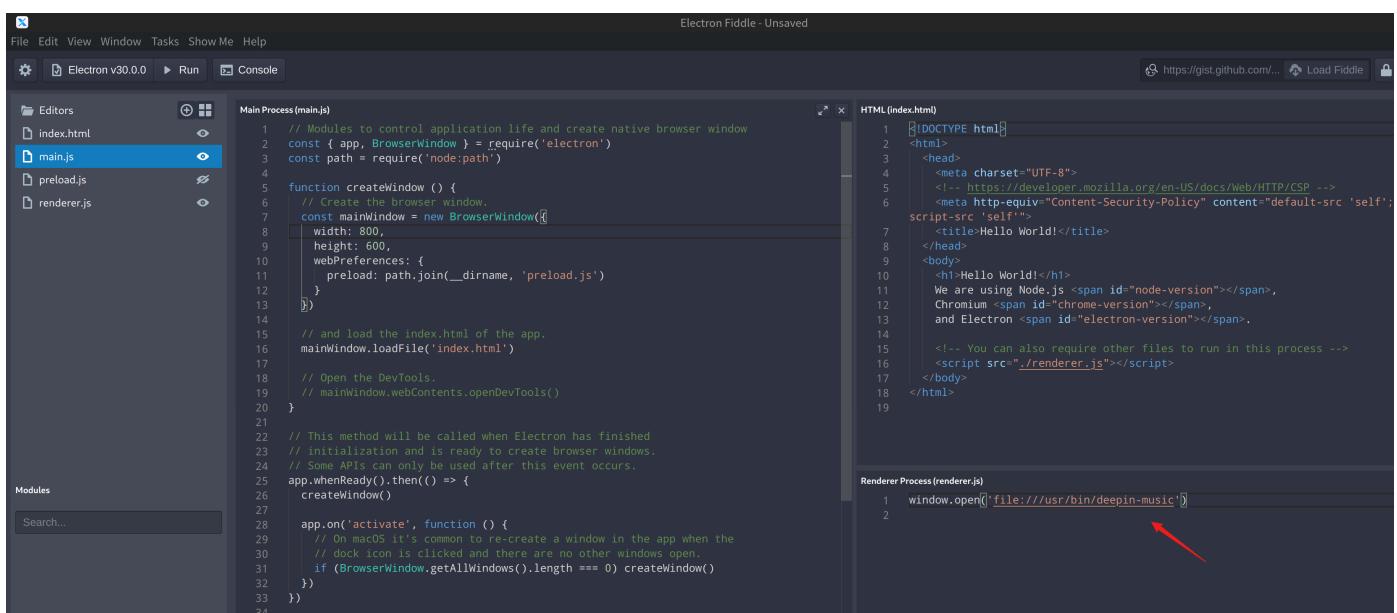
2) file 协议加载本地文件

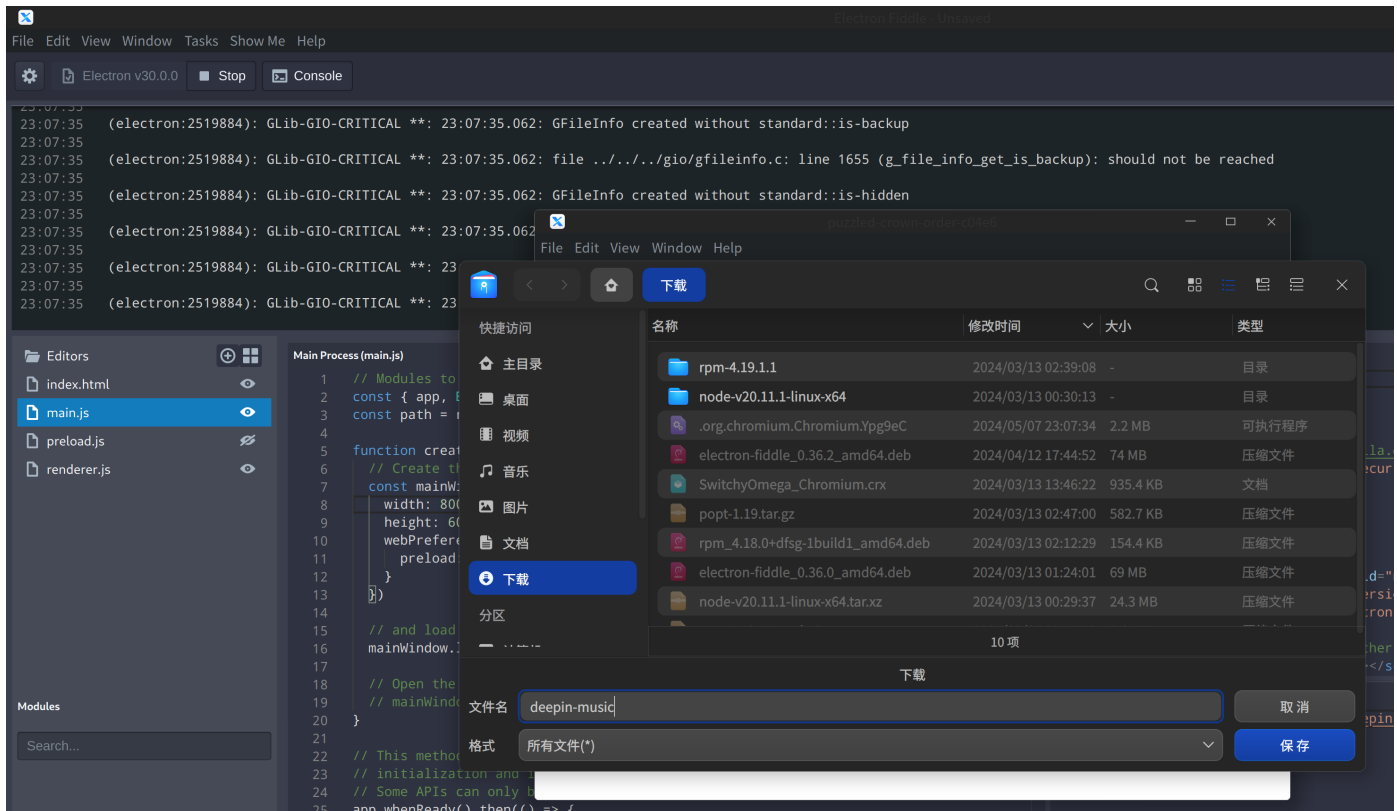




如果直接加载可执行二进制文件是什么效果呢？

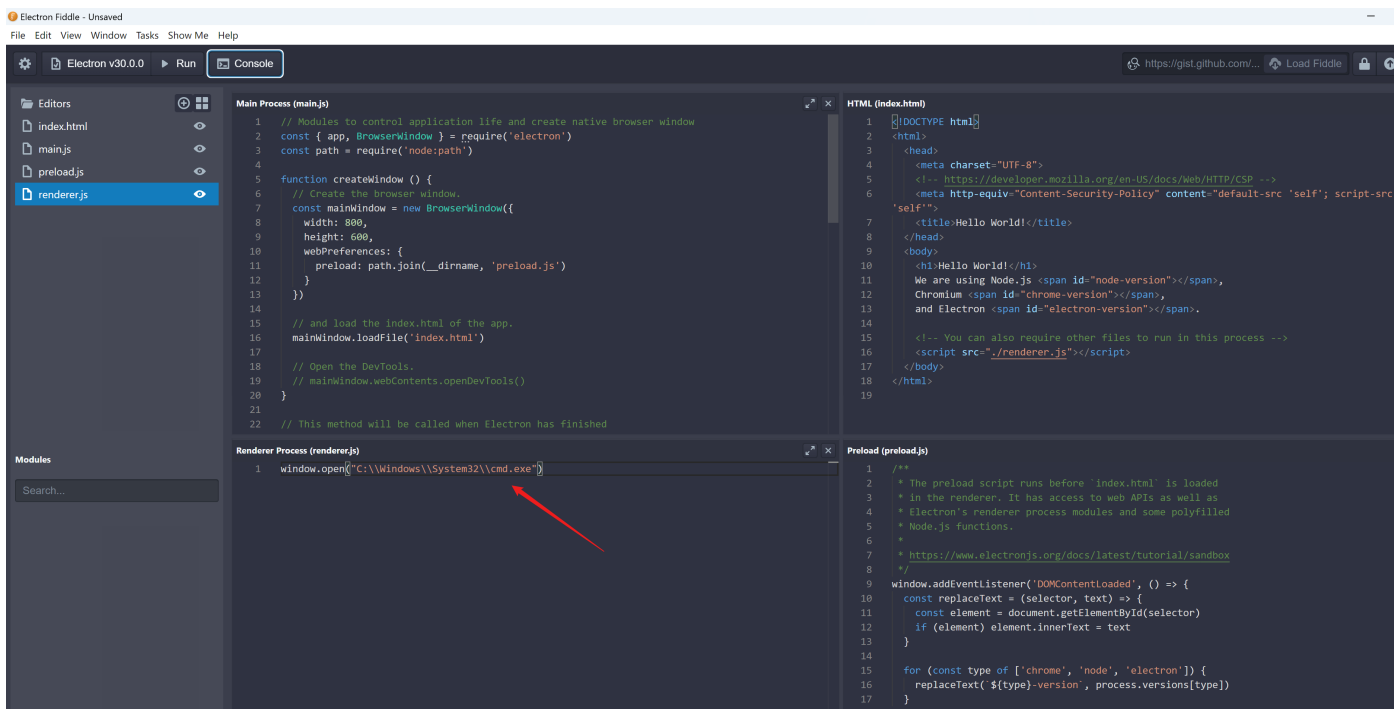
Deepin Linux

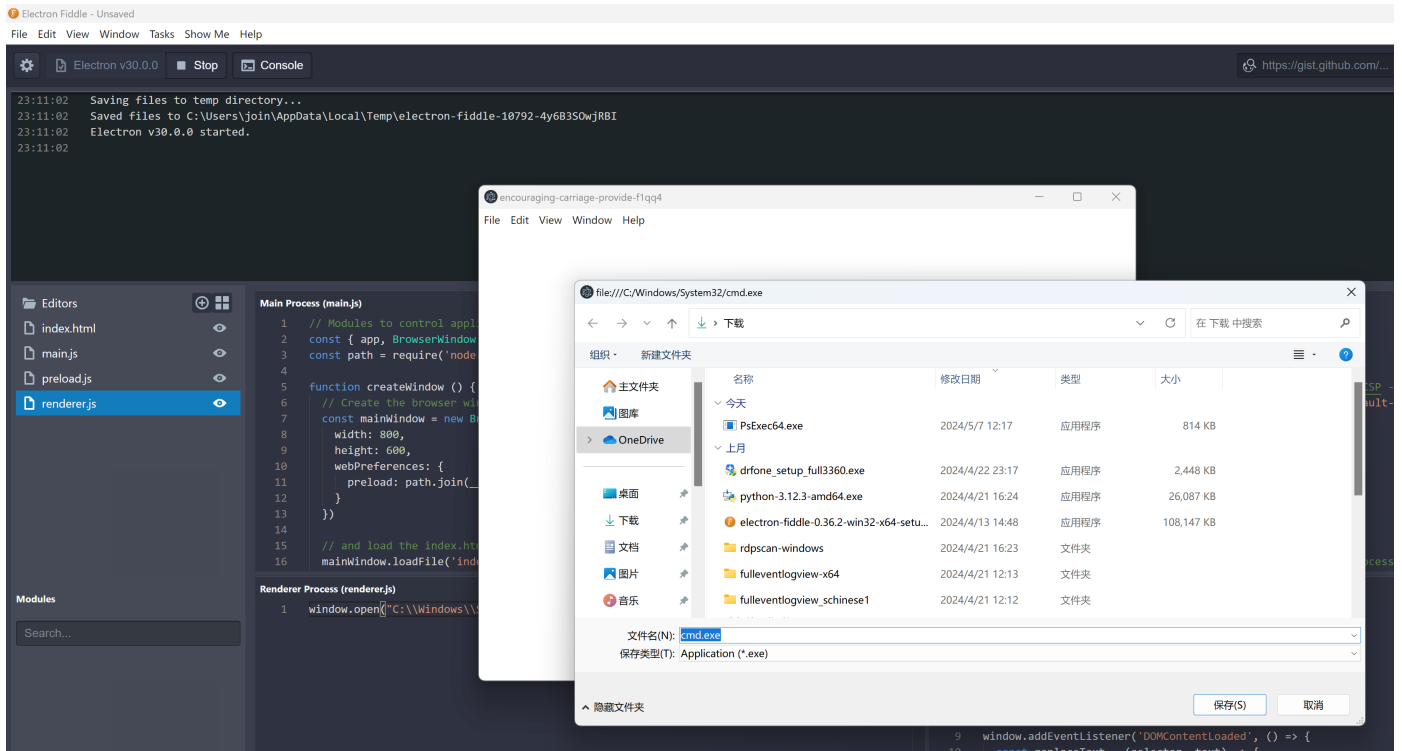




会直接变成下载文件

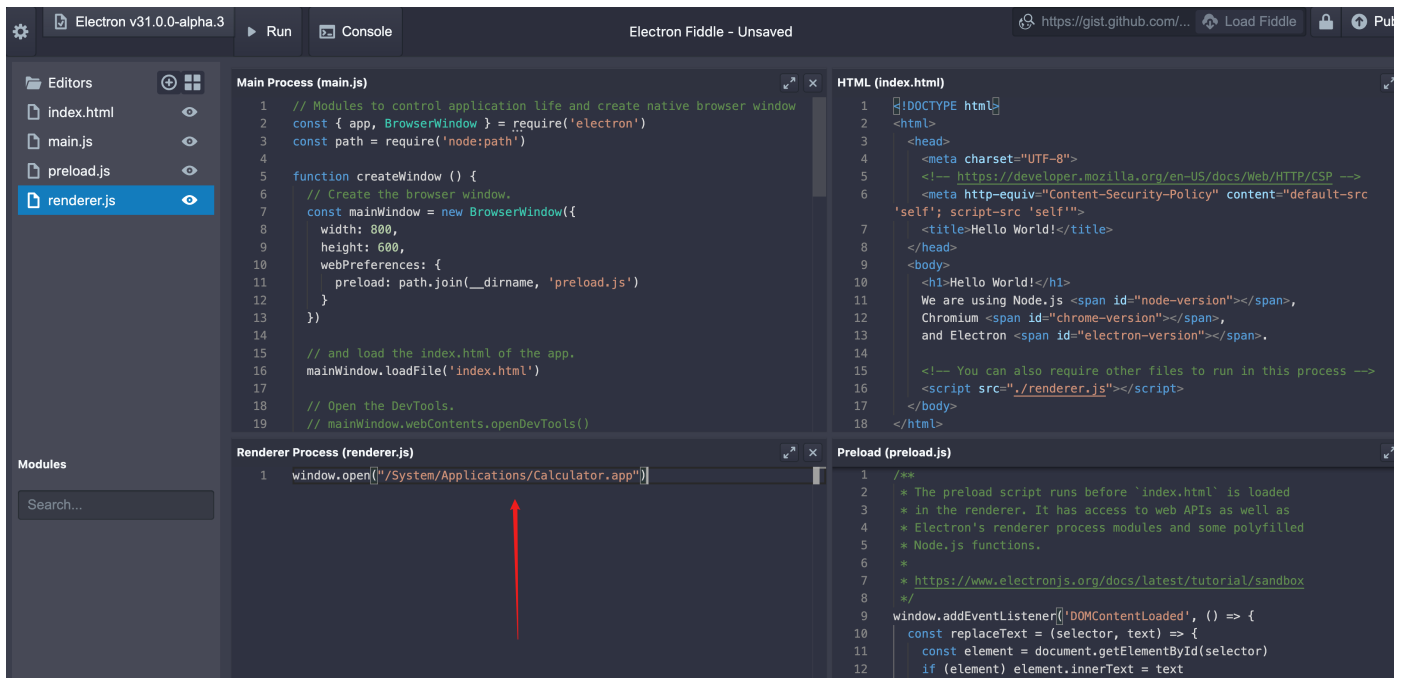
Windows 11

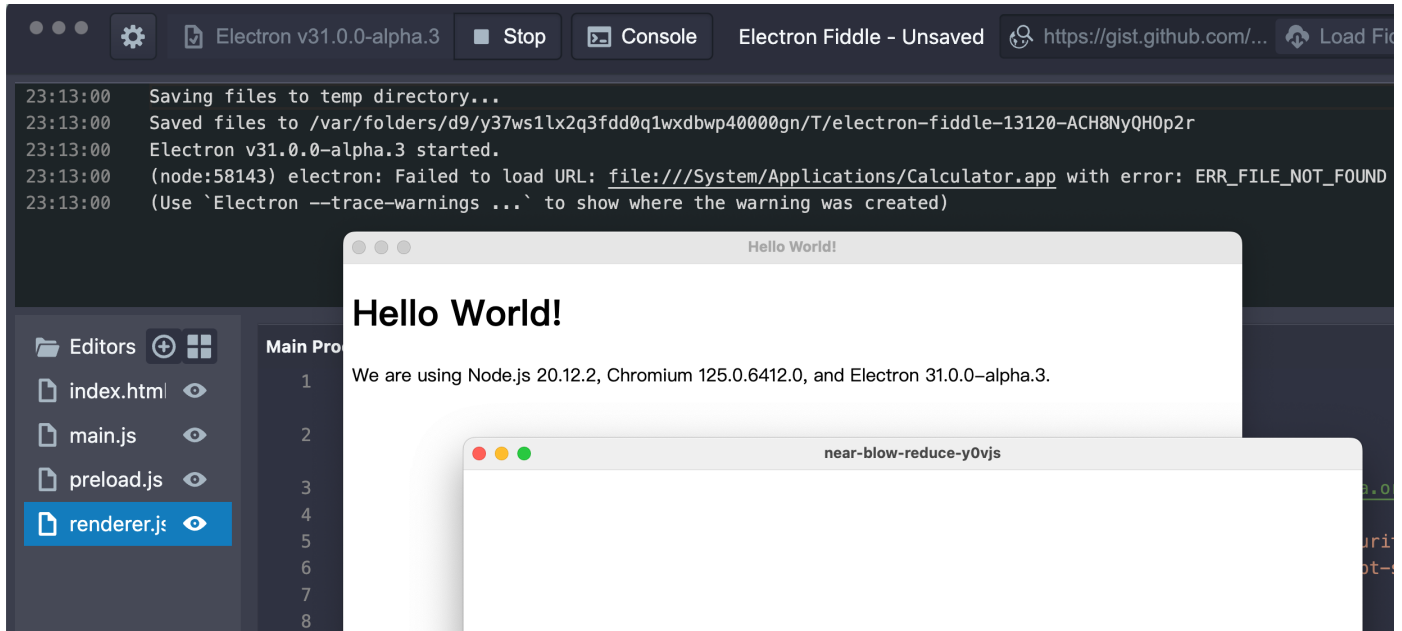




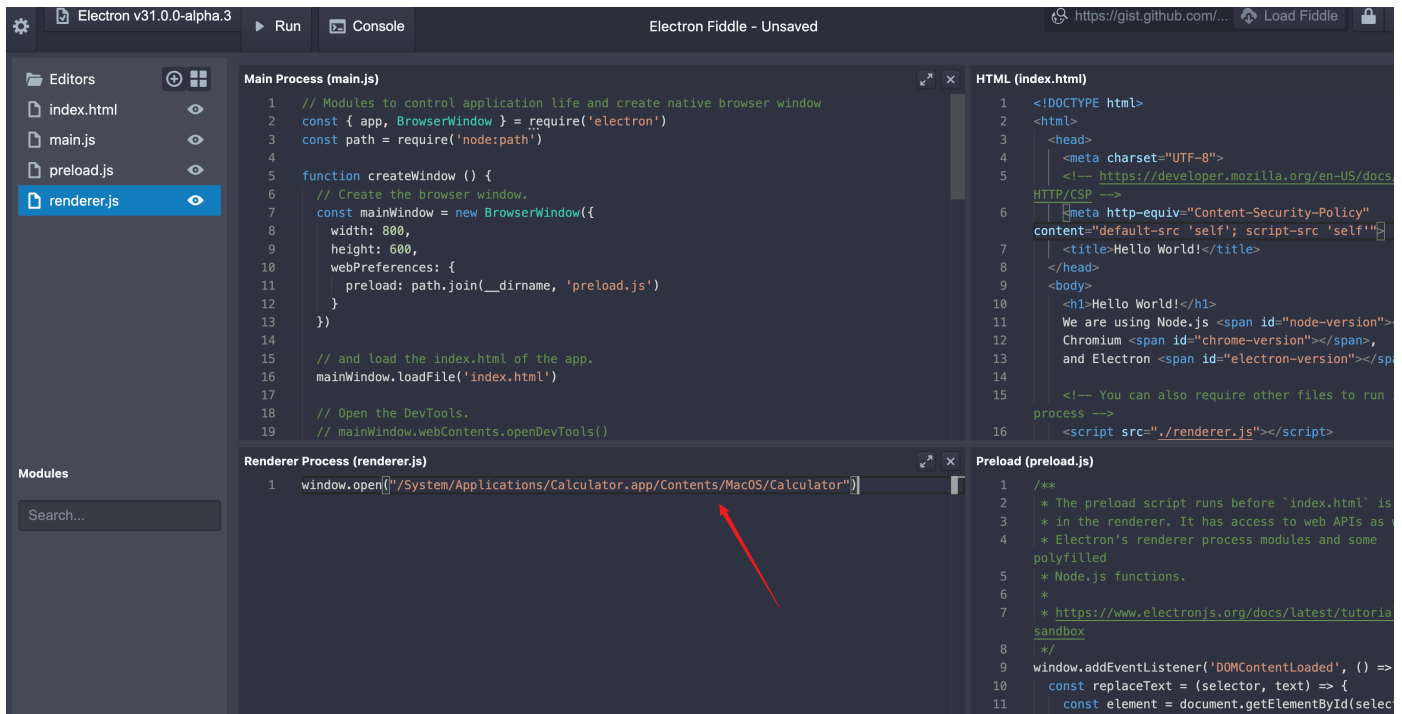
与 Deepin Linux 表现一致

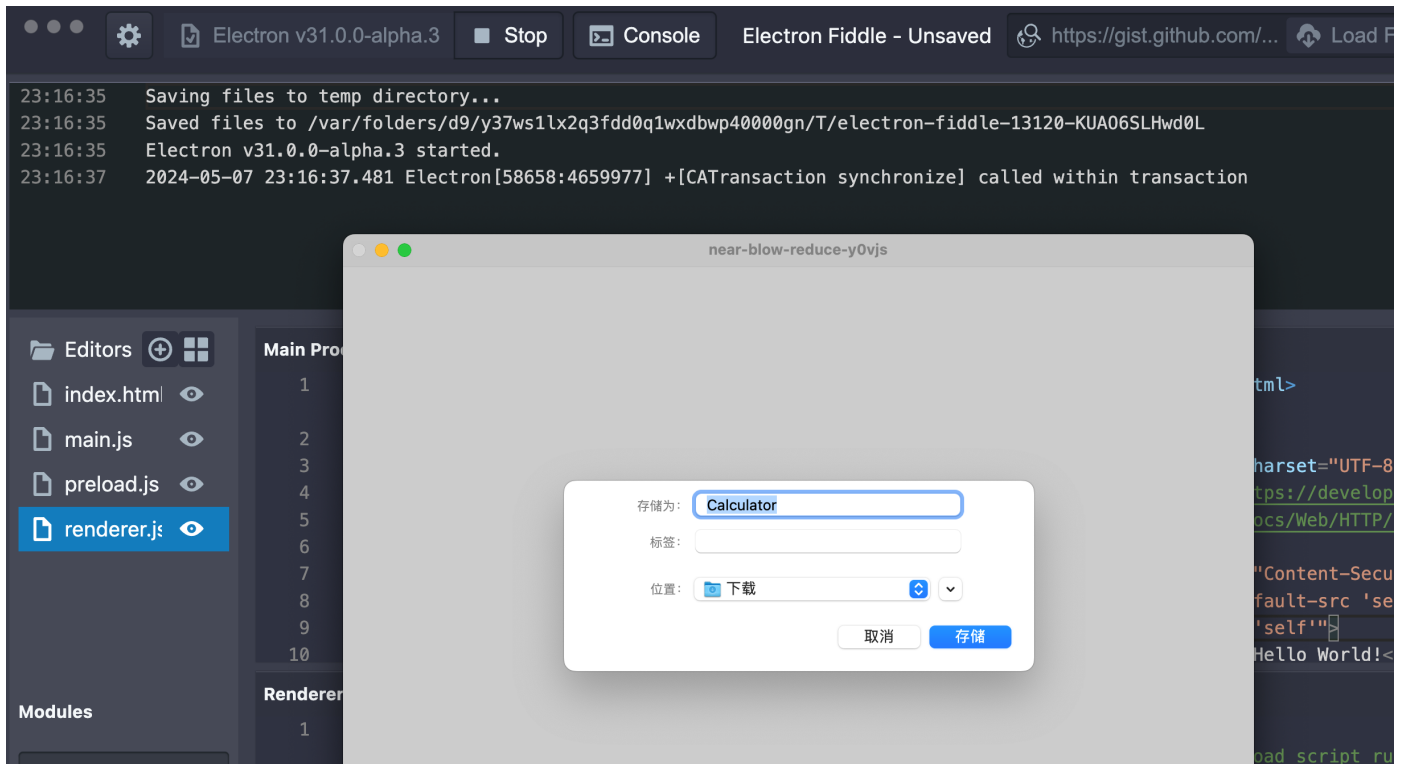
MacOS





报错是找不到文件，可能是将 `.app` 视为目录看待的

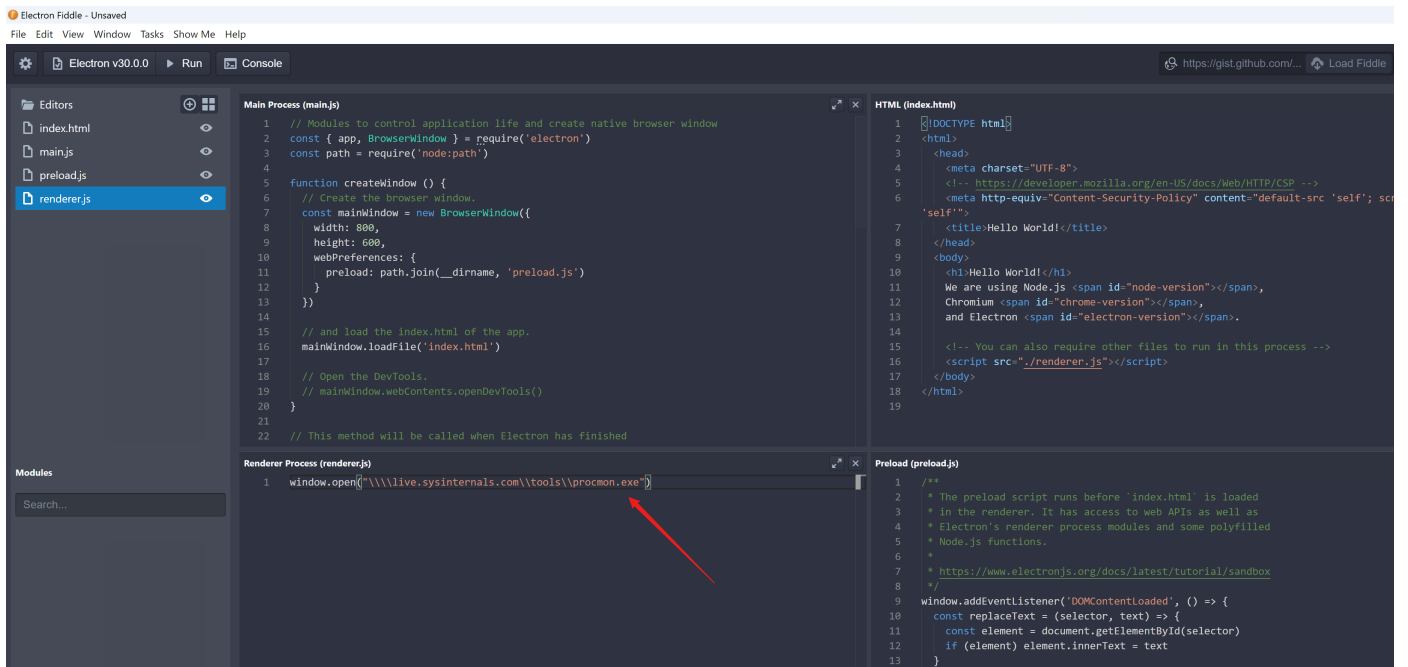


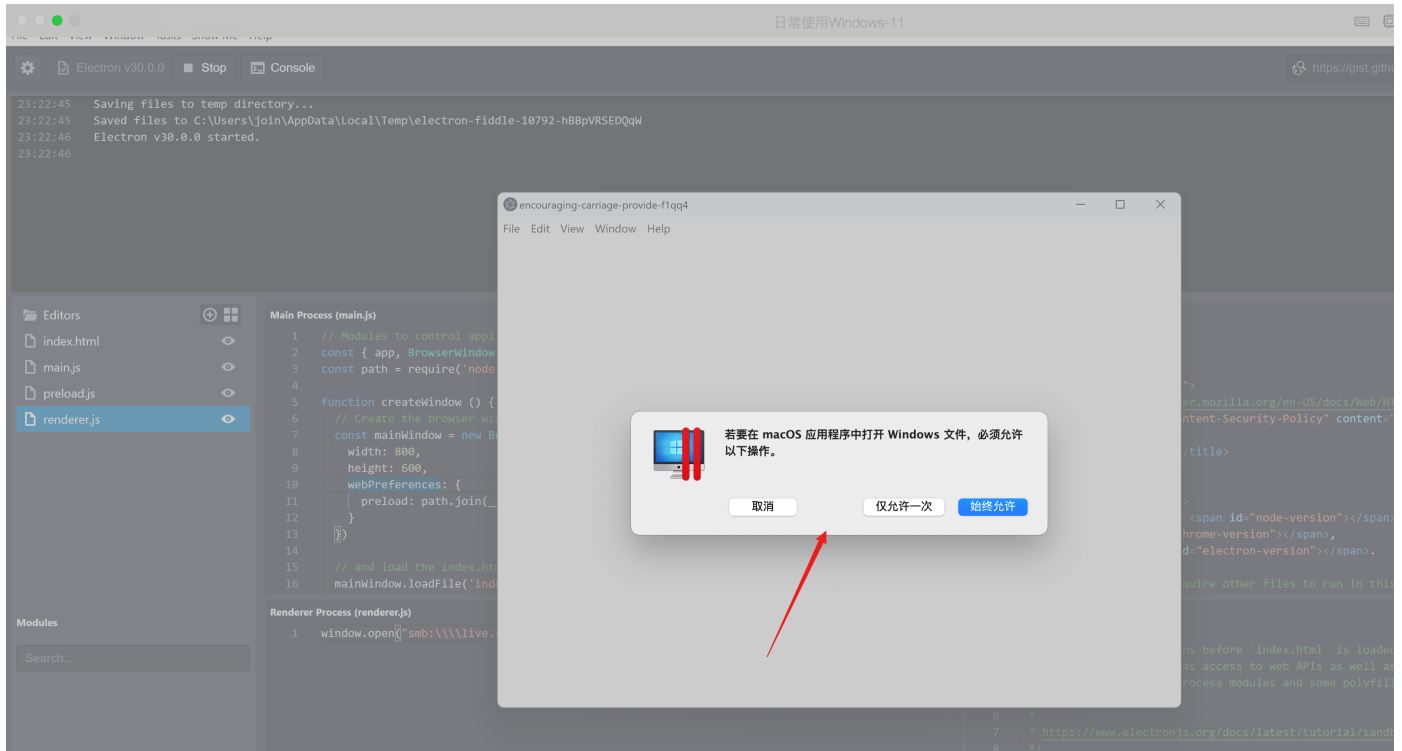


与 Deepin Linux 一致

3) SMB协议

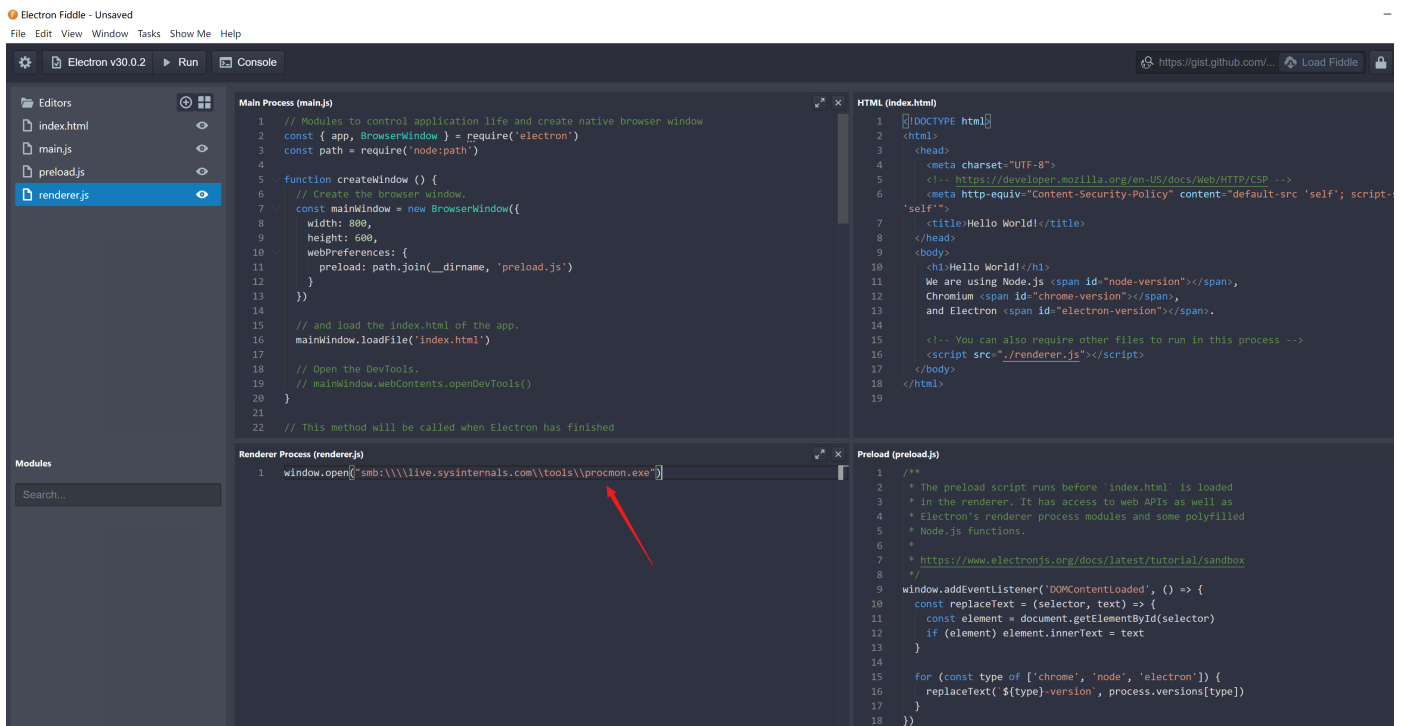
刚好之前测试了 `shell.openExternal` ，我们顺手测试一下 smb 协议

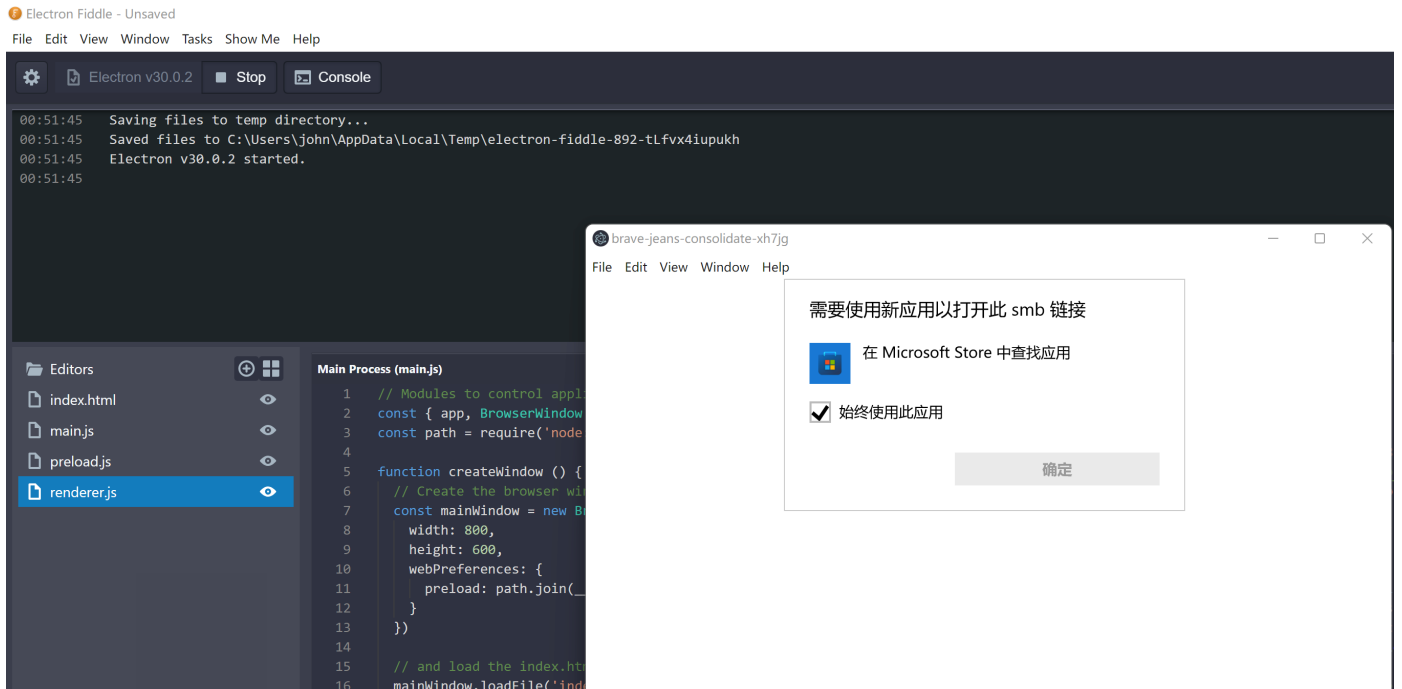




结果比较奇怪，因为是在虚拟机中测试的 Windows，它的行为是请求我的 MacOS 物理机打开 exe 程序，如果不在虚拟机里，会是什么样呢？我们换一个虚拟机试一下

使用 `vmware` 装一个 windows 11，再次测试

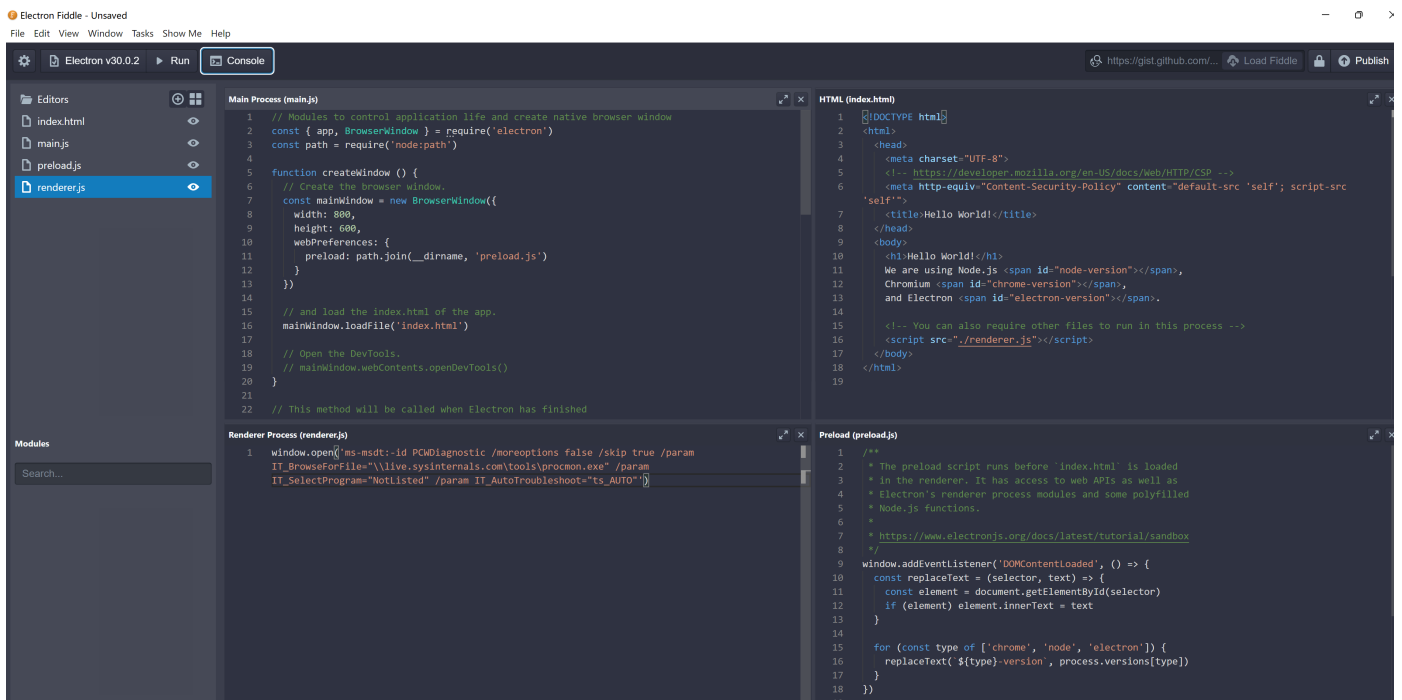


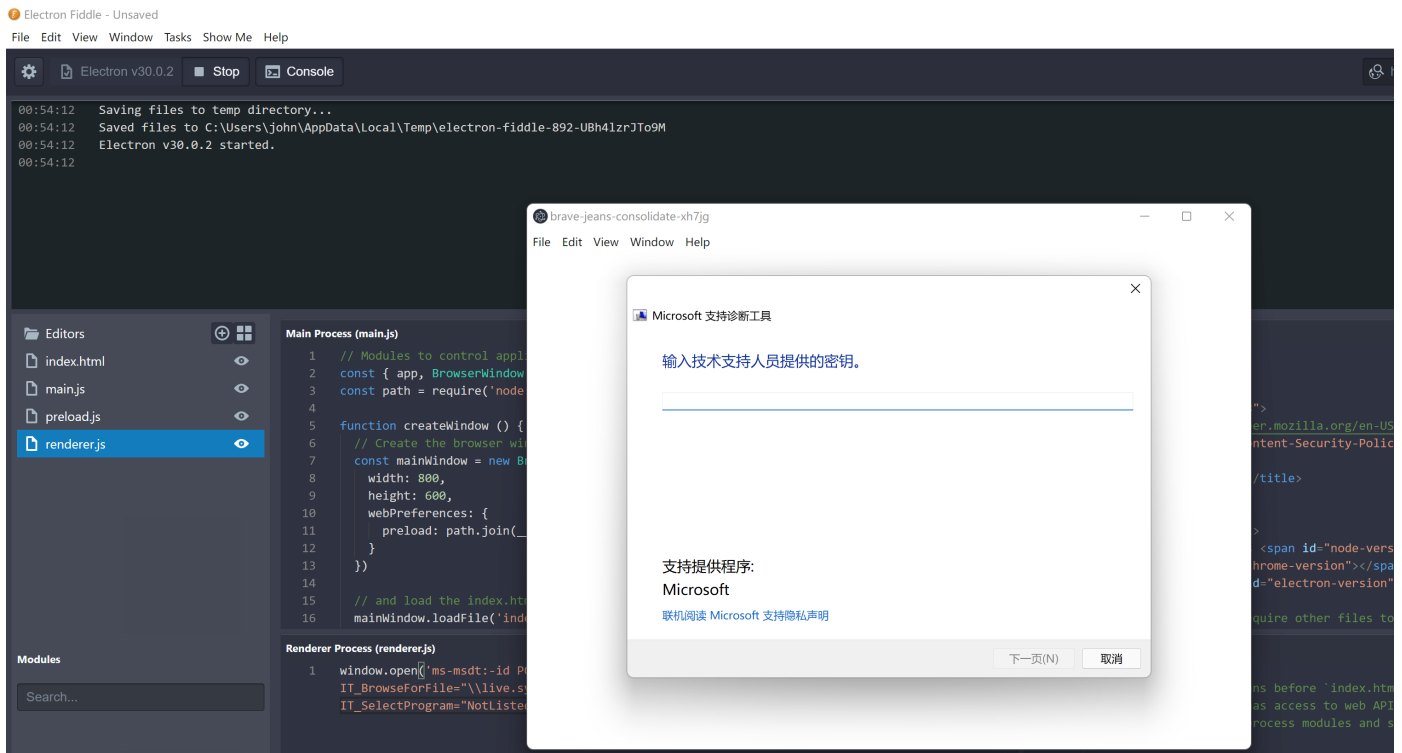


原来是这么一个结果

4) msdt

```
ms-msdt:-id PCWDiagnostic /moreoptions false /skip true /param
IT_BrowseForFile="//live.sysinternals.com\tools\procmon.exe" /param
IT_SelectProgram="NotListed" /param IT_AutoTroubleshoot="ts_AUTO"
```





竟然执行成功了，虽然因为 `Payload` 以及系统变更的原因，导致最终执行了这么个东西，但是大家需要注意，此时是可以正常解析的，和我们之前讨论的 `shell.openExternal` 是一致的

所以大家要关注这类系统注册协议的安全性(URI scheme)，之前就出现过 `ms-officecmd` 协议的注入类漏洞，这可是安全策略全开的情况下，直接从渲染进程发起的攻击

参考文章

<https://blog.xlab.app/p/8fbec25/#%E6%BC%8F%E6%B4%9E%E6%8C%96%E6%8E%98>

<https://positive.security/blog/ms-officecmd-rce>

3. `frameName`

`frameName` 其实就是原本 web 技术中 `window.open` 的 `target` 属性，所以 `frameName` 遵循 `target` 的规定

一个不含空格的字符串，用于指定加载资源的浏览上下文的名称。如果该名称无法识别现有的上下文，则会创建一个新的上下文，并赋予指定的名称。

窗口的名字主要用于为超链接和表单设置目标（targets）。窗口不需要有名称。

该名称可用作 `a` 或 `form` 元素的 `target` 属性

除了普通名称以外，`frameName`（`target`）还有几个特殊的关键字：

- `_self`
- `_blank`
- `_parent`
- `_top`

这几个关键字直接理解不太好理解，我们借 `a` 标签来理解，这几个特殊的关键字在 `a` 标签中完全支持

那 `a` 标签中 `target` 的意义是什么呢？

该属性指定在何处显示链接的 URL，作为浏览上下文的名称（标签、窗口或 `iframe`）

其实就是，我在当前页面点击了一个 `a` 标签，标签 `href` 指向的是百度的地址，你想在哪里看到点击后的结果，是当前页面呢？还是当前页面的父页面？还是顶级导航的页面，还是干脆新打开一个标签/窗口来展示

- `_self`：当前页面加载。（`a` 标签默认）
- `_blank`：通常在新标签页打开，但用户可以通过配置选择在新窗口打开。
- `_parent`：当前浏览环境的父级浏览上下文。如果没有父级框架，行为与 `_self` 相同。
- `_top`：最顶级的浏览上下文（当前浏览上下文中最“高”的祖先）。如果没有祖先，行为与 `_self` 相同。

4. features

`features` 一个字符串，包含以逗号分隔的窗口特性列表，形式为 `name=value`，布尔特性则仅为 `name`

官方给了一个案例

```
window.open('https://github.com', '_blank',  
'top=500,left=200,frame=false,nodeIntegration=no')
```


在 web 技术中，这个参数叫做 `windowFeatures`，但 `Electron` 将 `windowFeatures` 扩充了，支持 `BrowserWindowConstructorOptions` 的配置，也就是构建 `BrowserWindow` 可以使用的配置，同时将 `WebPreferences` 中的一部分拿出来，也作为快捷的配置，例如

- `zoomFactor`
- `nodeIntegration`
- `preload`
- `javascript`
- `contextIsolation`
- `webviewTag`

具体参考

<https://www.electronjs.org/zh/docs/latest/api/structures/browser-window-options>

<https://www.electronjs.org/zh/docs/latest/api/window-open>

除了 `Electron` 添加的这些以外，其他配置如下

1) popup

如果启用此特性，则要求使用最小弹出窗口。弹出窗口中包含的用户界面功能将由浏览器自动决定，一般只包括地址栏。

如果未启用 `popup`，也没有声明窗口特性，则新的浏览上下文将是一个标签页。

备注：在 `windowFeatures` 参数中指定除 `noopener` 或 `noreferrer` 以外的任何特性，也会产生请求弹出窗口的效果。

要启用该特性，可以不指定 `popup` 值，或将其设置为 `yes`，`1` 或 `true`。

例如：`popup=yes`、`popup=1`、`popup=true` 和 `popup` 的结果完全相同。

2) width 或 innerWidth

指定内容区域（包括滚动条）的宽度。最小要求值为 100

3) height 或 innerHeight

指定内容区域（包括滚动条）的高度。最小要求值为 100

4) left 或 screenX

指定从用户操作系统定义的工作区左侧到新窗口生成位置的距离（以像素为单位）

5) top 或 screenY

指定从用户操作系统定义的工作区顶部到新窗口生成位置的距离（以像素为单位）

6) noopener

如果设置了此特性，新窗口将无法通过 `Window.opener` 访问原窗口，并返回 `null`。

使用 `noopener` 时，在决定是否打开新的浏览上下文时，除 `_top`、`_self` 和 `_parent` 以外的非空目标名称会像 `_blank` 一样处理

7) noreferrer

如果设置了此特性，浏览器将省略 `Referer` 标头，并将 `noopener` 设为 `true`。更多信息请参阅 `rel="noreferrer"`

5. 权限继承关系

- 如果在父窗口中禁用了 Node integration, 则在打开的 `window` 中将始终被禁用。
- 如果在父窗口中启用了上下文隔离, 则在打开的 `window` 中将始终被启用。
- 父窗口禁用 Javascript, 打开的 `window` 中将被始终禁用
- 非标准功能 (不由 Chromium 或 Electron 提供) 给定 `features` 将传递给注册 `webContents` 的 `did-create-window` 事件处理函数的 `options` 参数。
- 当打开 `about:blank` 时, 子窗口的 `WebPreferences` 将从父窗口复制, 并且没有办法覆盖它, 因为Chromium在这种情况下跳过浏览器侧导航。

6. 小结

从 web 技术对于 `window.open` 的描述以及它的相关属性来看其实 `window.open` 并不等同于打开新窗口, 更加准确的描述应该是 **用指定的名称将指定的资源加载到新的或已存在的浏览上下文 (标签、窗口或 `iframe`) 中**

打开的地址可以是 `http(s)` 这种web地址, 也可以是本地路径和其他协议的地址, 如果攻击者能够控制 `url`, 是可能结合 `URI scheme` 方面的漏洞实现全安全策略下渲染进程发起的 `RCE` 的

所以 `target` 属性就是指定你加载的资源要在哪个窗口(标签或 `iframe`) 中加载并显示, 如果设置 `_blank` 就会打开新窗口, 如果 `target` 的值指向一存在的窗口名字就会复用窗口

根据 web 技术中对 `window.open` 的描述, 也和之前 web 嵌入章节一样, 如果父窗口和子窗口同源, 则可以通过对象关系进行访问, 不同源则不行

当然, 在 `features` 中也有 `noopener` 这种特性会破坏这种引用关系

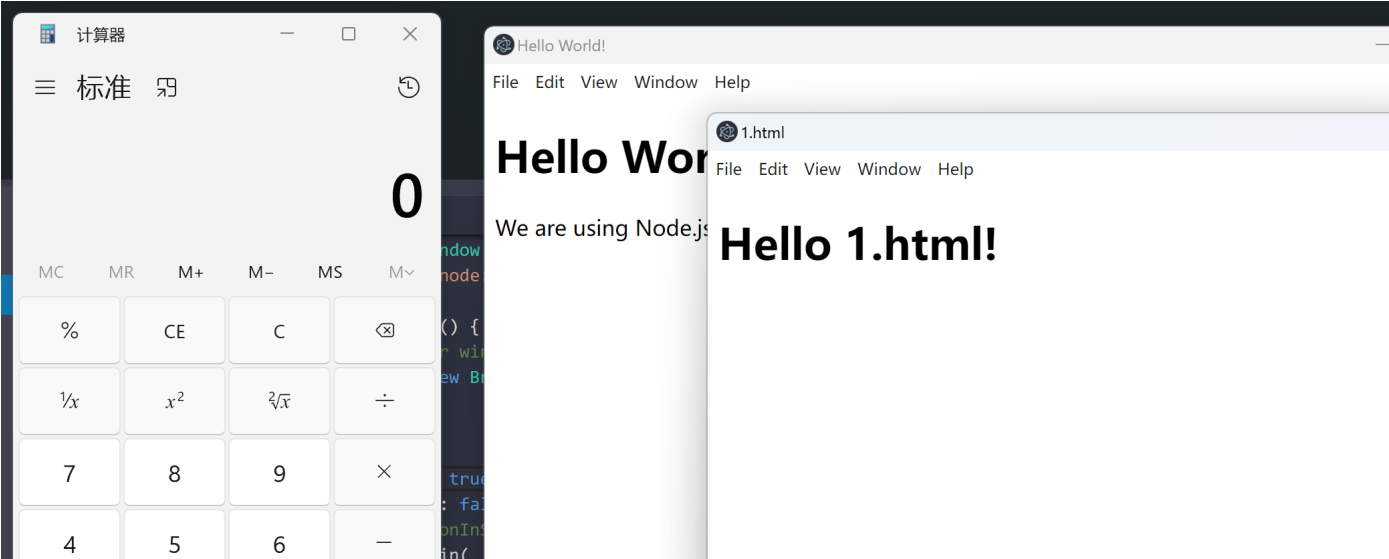
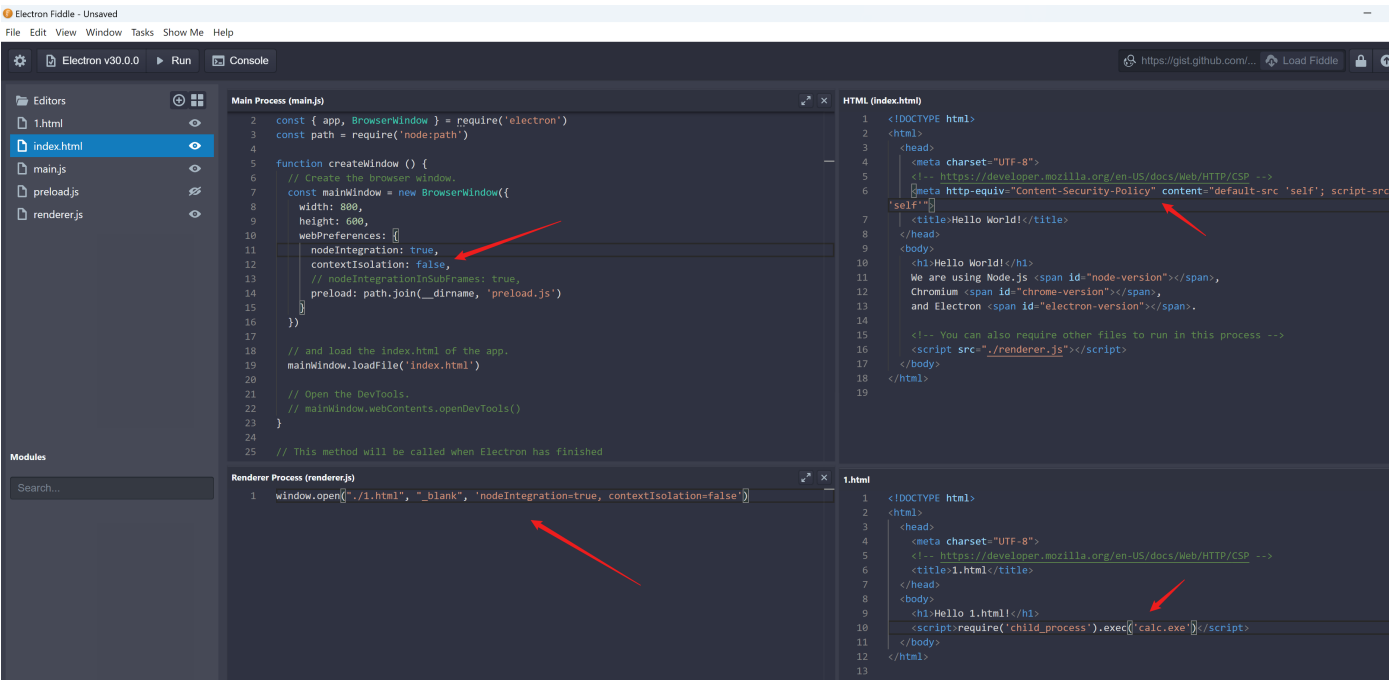
参考文章

<https://developer.mozilla.org/en-US/docs/Web/API/Window/open#parameters>

<https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/a#target>

0x04 window.open Node.js 测试

按照官方文档，只有当父窗口具备 Node.js 能力时，window.open 设置了相关安全策略才可能获取到 Node.js 的能力



确实可以执行 Node.js

经过测试，window.open 打开的窗口想要具备 Node.js 能力，需要父窗口开启 nodeIntegration 关闭上下文隔离，同时 window.open 的 feature 中配置 nodeIntegration 和上下文隔离

如果父窗口不具备 Node.js 执行能力，但是 window.open 配置了 Node.js 支持，并且 frameName 设置为一个已经存在并且具备 Node.js 能力的窗口，此时 window.open 加载的内容是否具备 Node.js 能力呢？

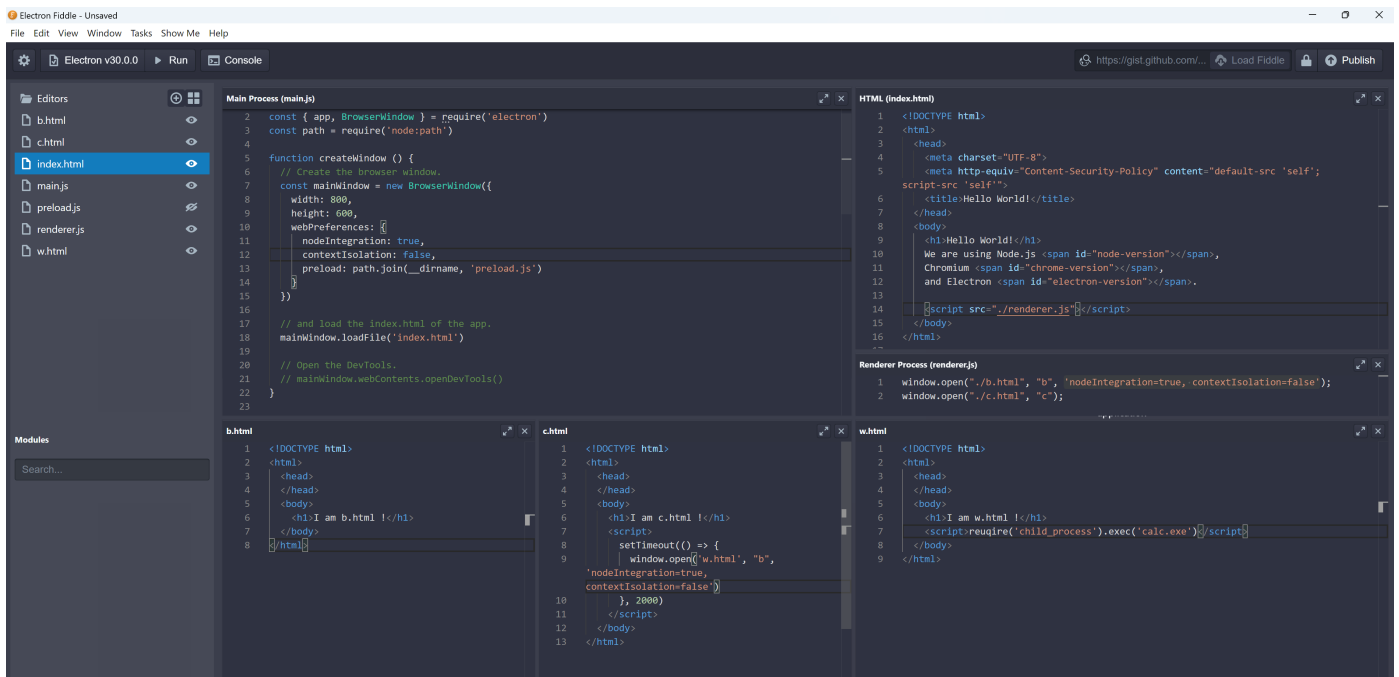
这个实验还挺复杂的，因为我们需要模拟一个具备 Node.js 的窗口，一个不具备 Node.js 的窗口，之后还要在不具备 Node.js 的窗口里 window.open ，还有最基础的主窗口

主窗口代号为 a ，加载 index.html ，需要具备 Node.js 能力

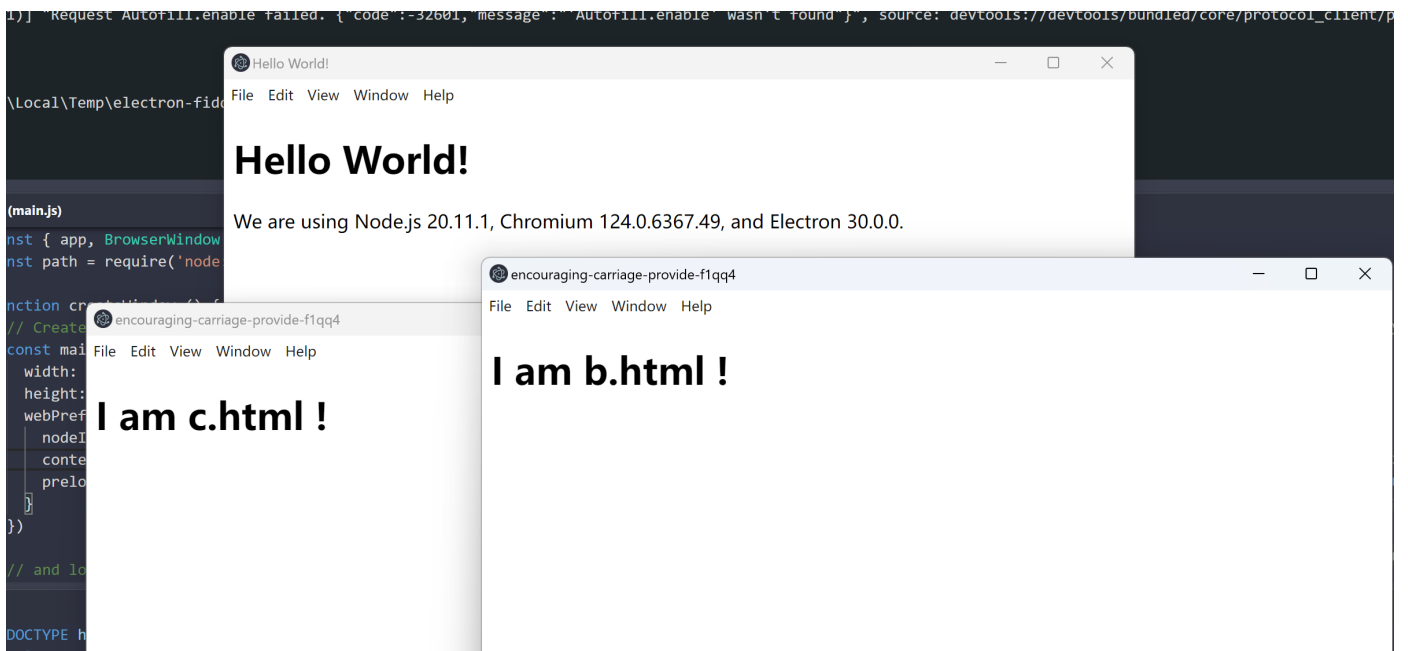
主窗口创建的具备 Node.js 能力的窗口 代号为 b ，加载 b.html

主窗口创建的不具备 Node.js 能力的窗口代号为 c ，加载 c.html

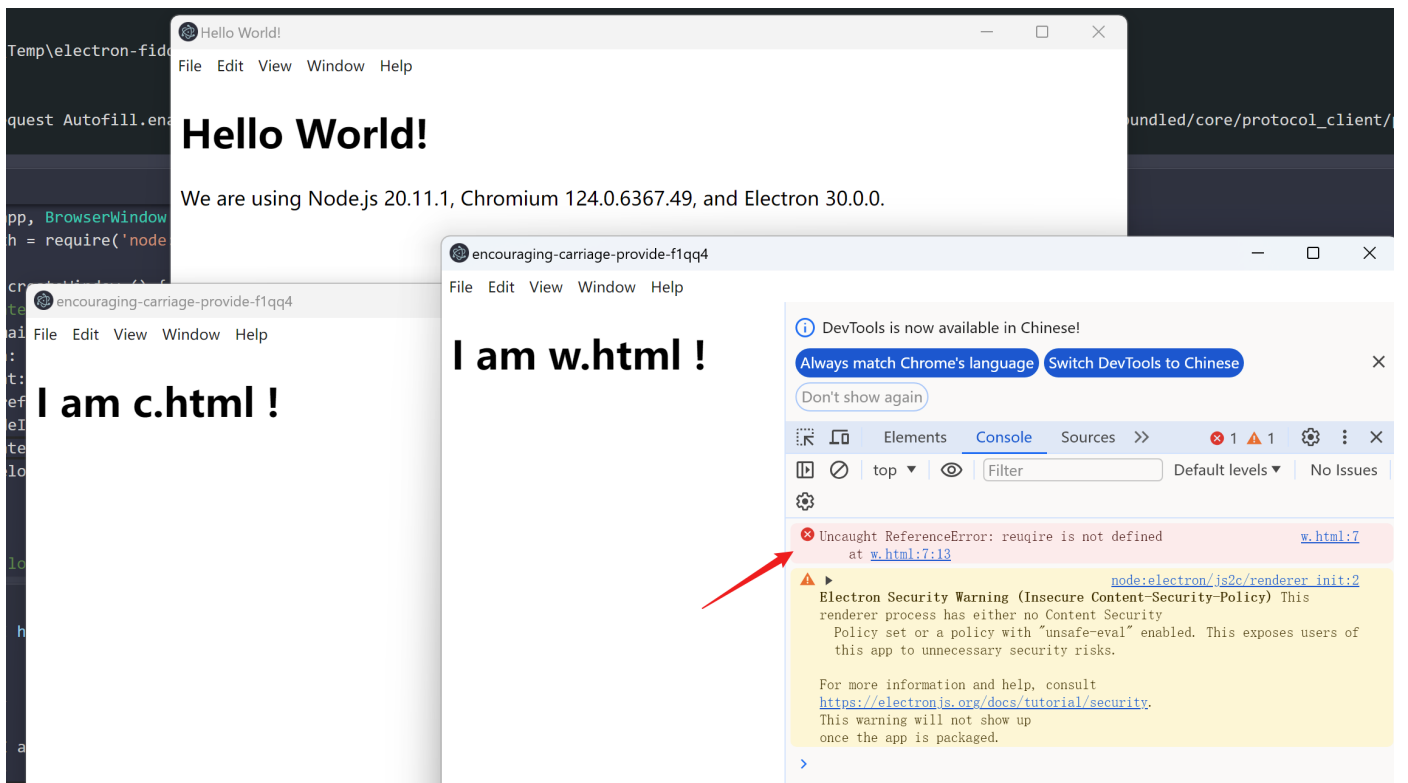
c 窗口使用 window.open 抢占 b 窗口，加载 w.html ，测试是否存在 Node.js 能力



执行测试



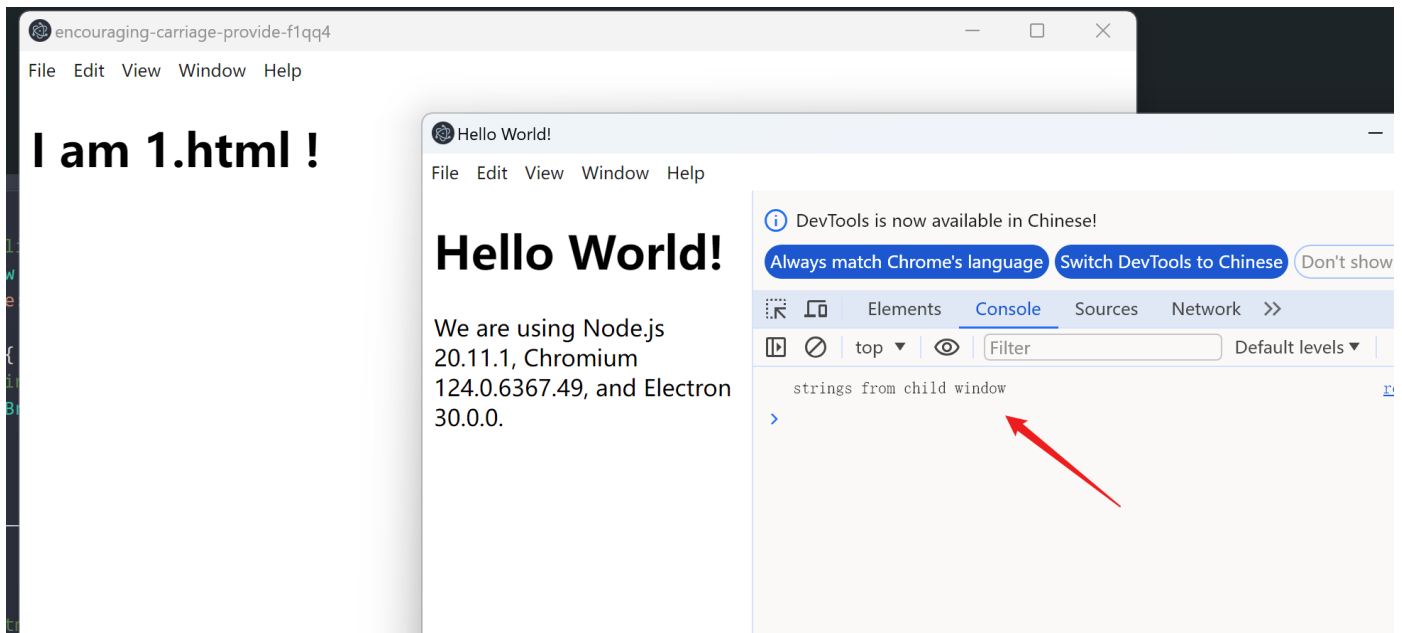
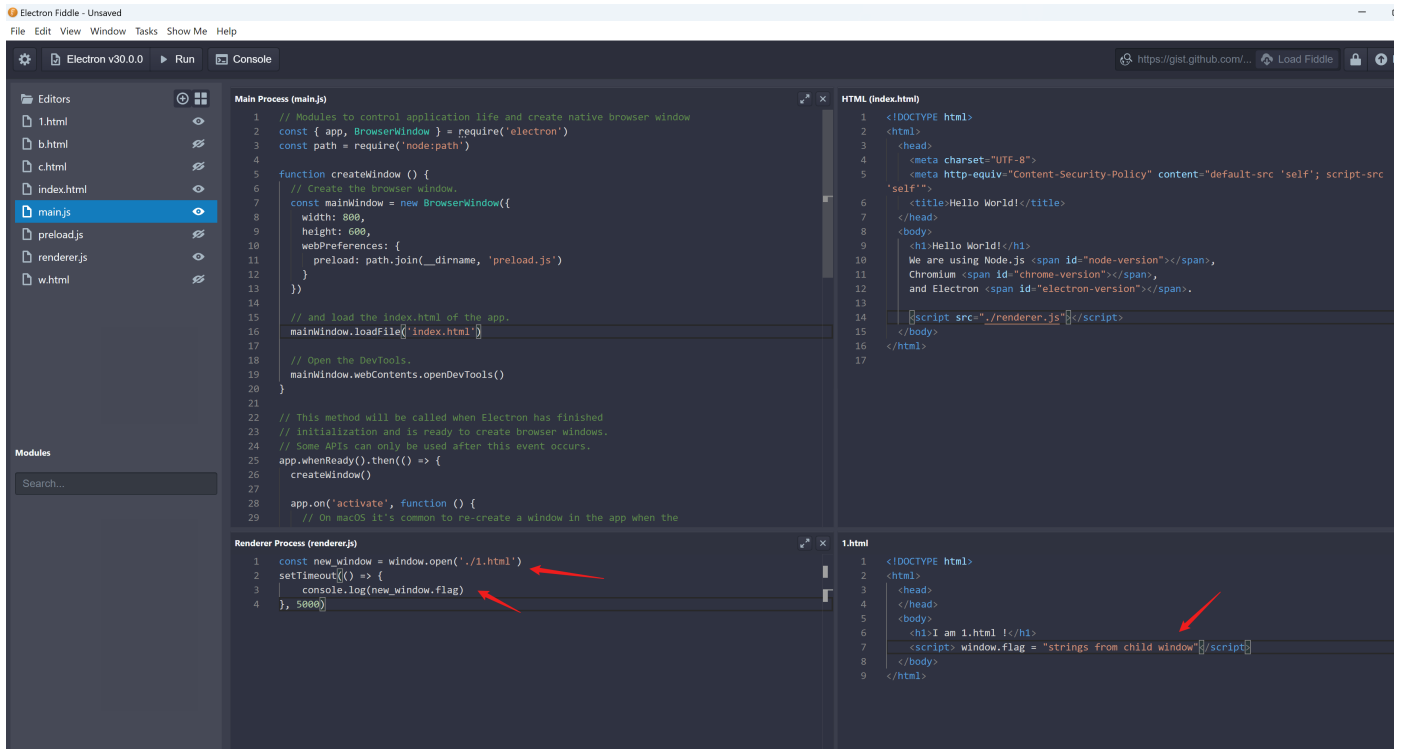
过了 2 秒后



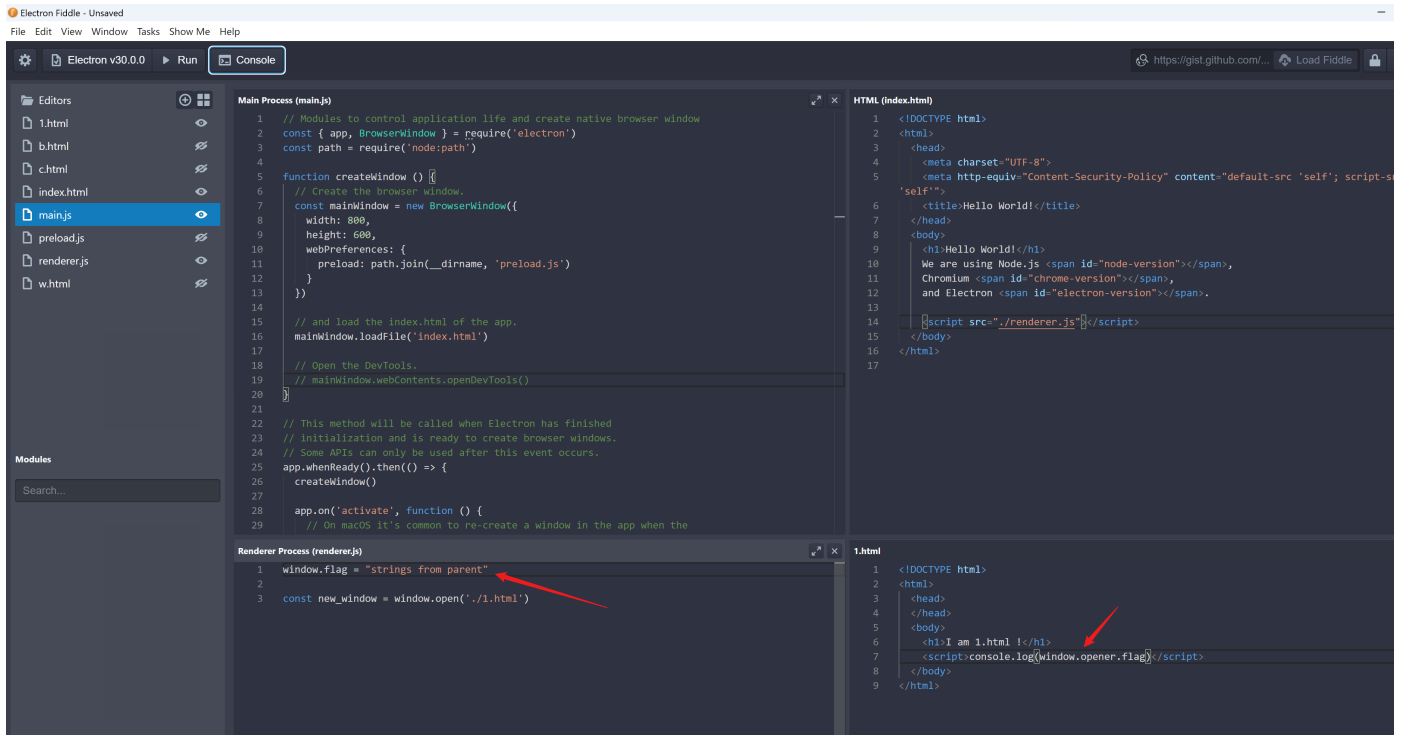
w.html 成功抢占 b 窗口，但其权限还是继承的 c 窗口，即其父窗口，无法执行 Node.js

0x05 window.open 上下文情况

父窗口调用 `window.open` 创建子窗口时会返回一个指向新窗口对象的引用，父窗口可以通过这个引用直接访问子窗口的上下文

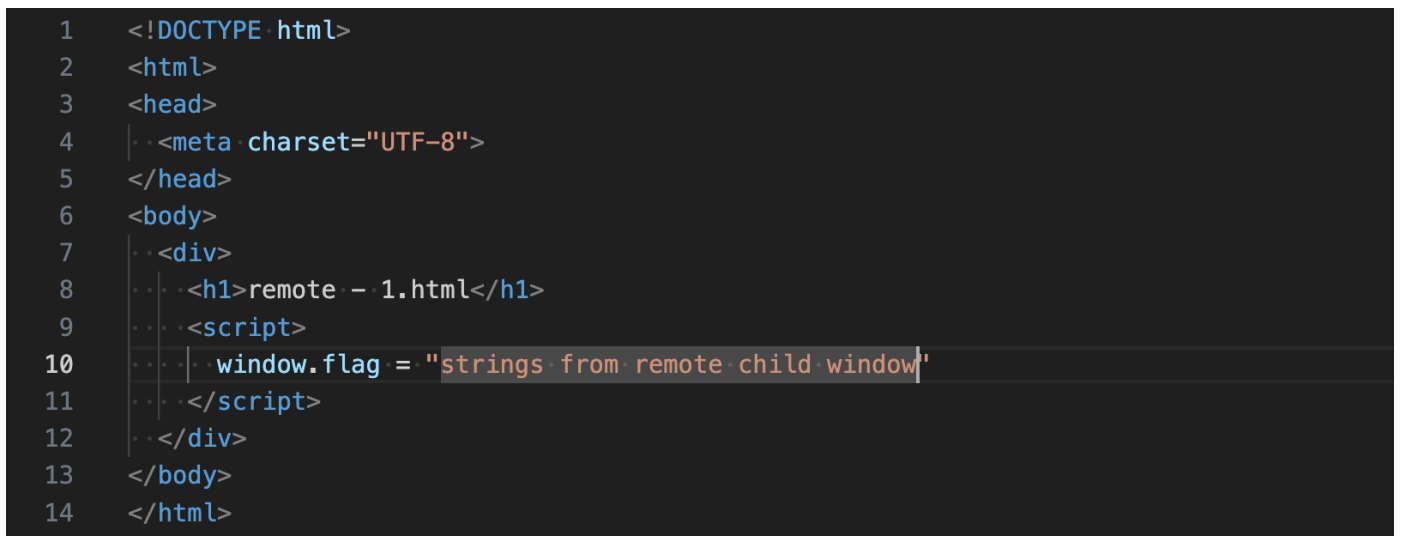


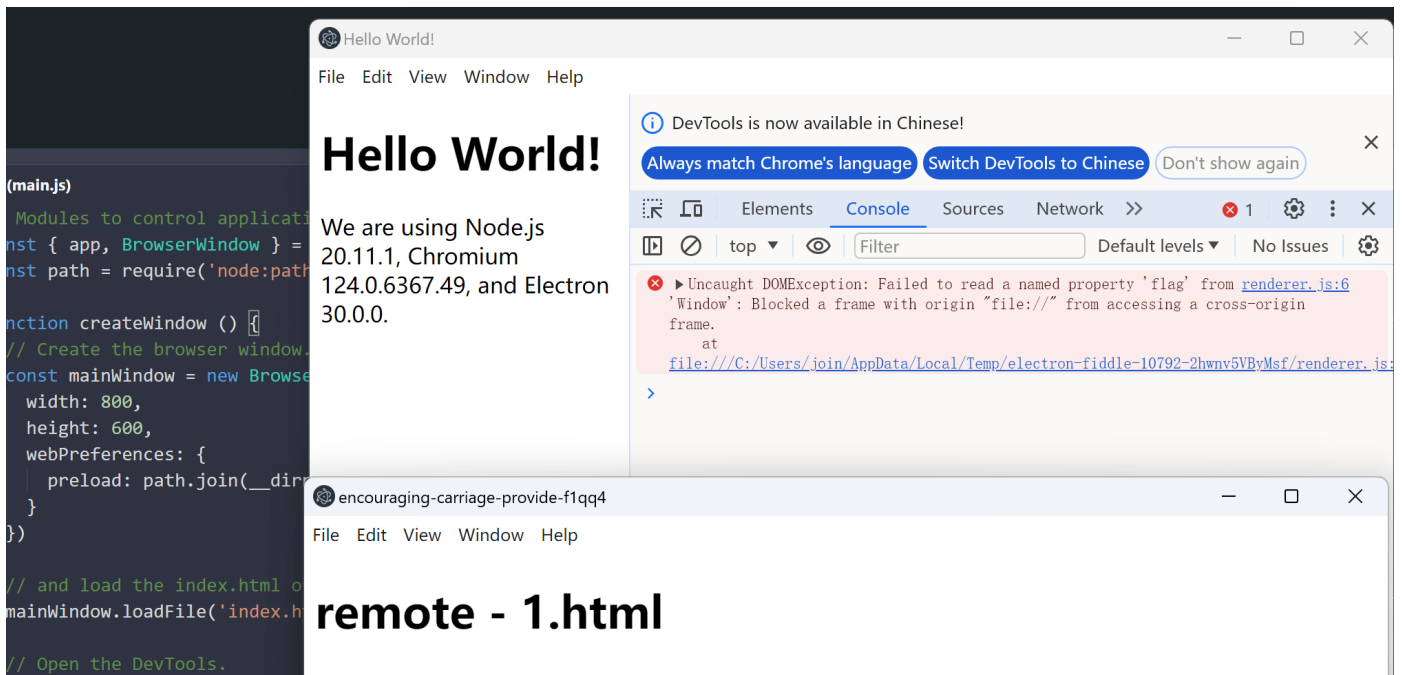
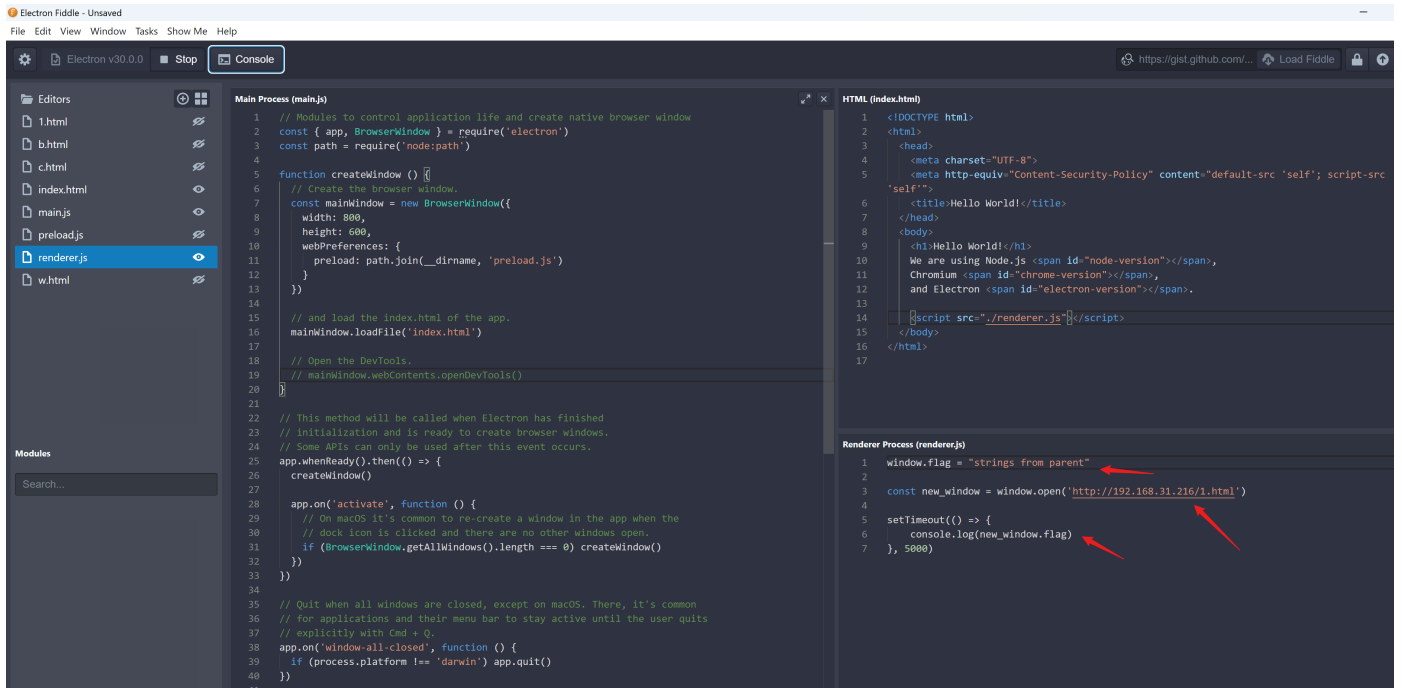
同源情况下，子窗口获取父窗口上下文测试



同源情况下的访问是双向的，与之前 `iframe` 、 `object` 之类的没有区别

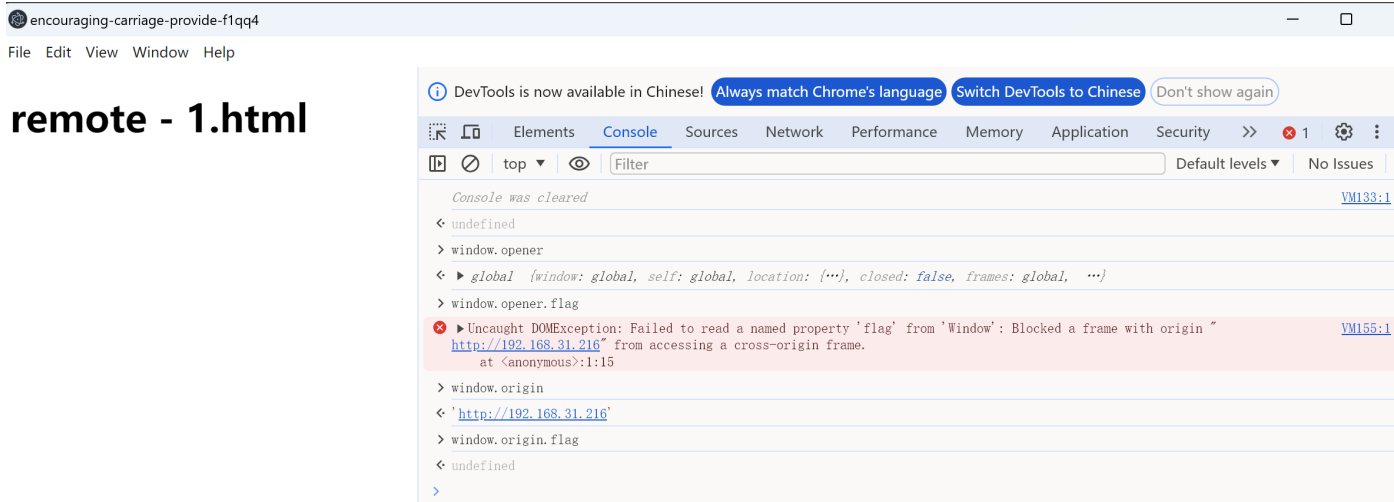
非同源情况下，按照正常来说，父窗口访问子窗口应该还是一样的





结果并不是我们想的那样，虽然有返回对象，但是获取不到子窗口的上下文

我们可以直接在子窗口上打开开发者工具，进入控制台，输出 `window.opener` 看看是否存在内容



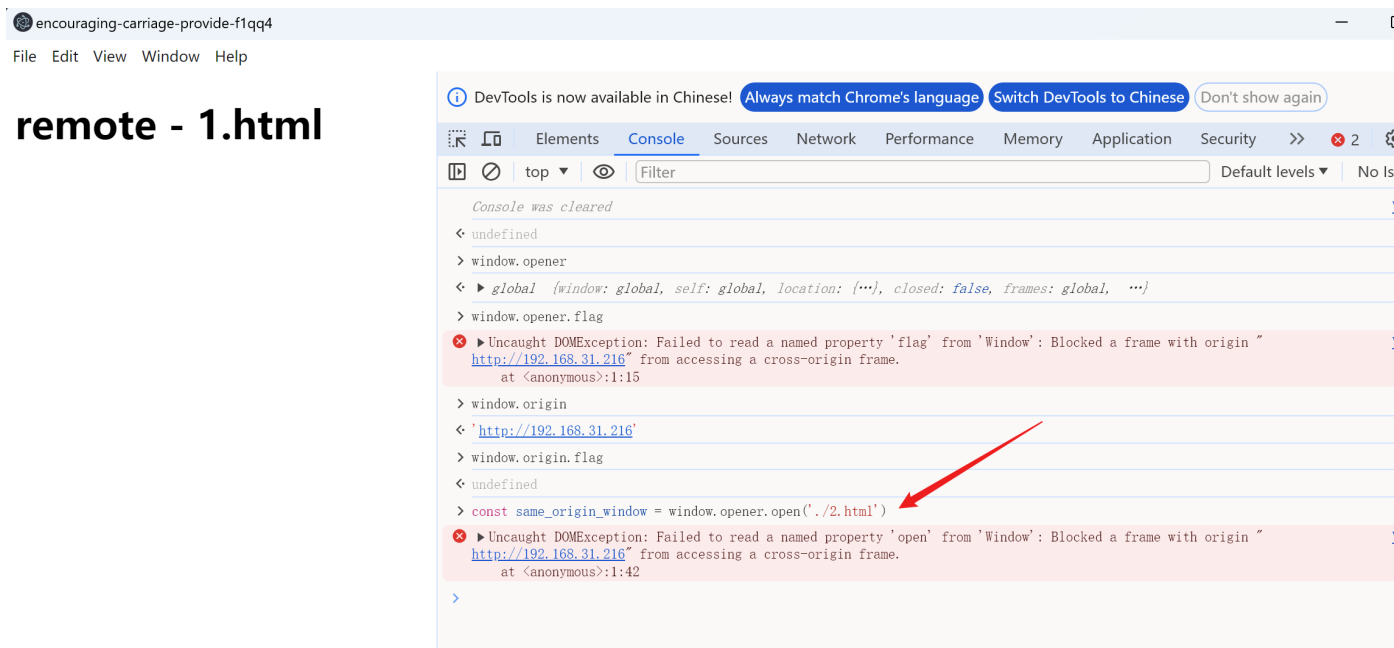
存在 `window.opener` 但是获取不到父窗口的上下文，如果此时，在子窗口使用 `window.opener` 对象的 `open` 方法再打开一个与父窗口同源的新窗口，并且获取新窗口对象，用这个对象与父窗口进行通信，会不会就可以获取到父窗口的上下文了呢？

与父窗口同源的 `2.html` 内容如下

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>I am 2.html !</h1>
  </body>
</html>
```

此时我在非同源的这个子窗口的控制台执行

```
const same_origin_window = window.opener.open('./2.html')
```



失败了，但即使成功的话，这次新建的窗口与非同源的窗口之间的关系也是非同源的，其实没啥用的，这个思路就不行，有点骑驴找驴的意思

0x06 漏洞案例

远古时期，`window.open` 可以通过 `file://` 远程加载 `html`

<https://github.com/electron/electron/issues/5151>

比较早的版本中 `window.open` 出现过权限绕过的漏洞，详情参考

<https://www.electronjs.org/blog/window-open-fix>

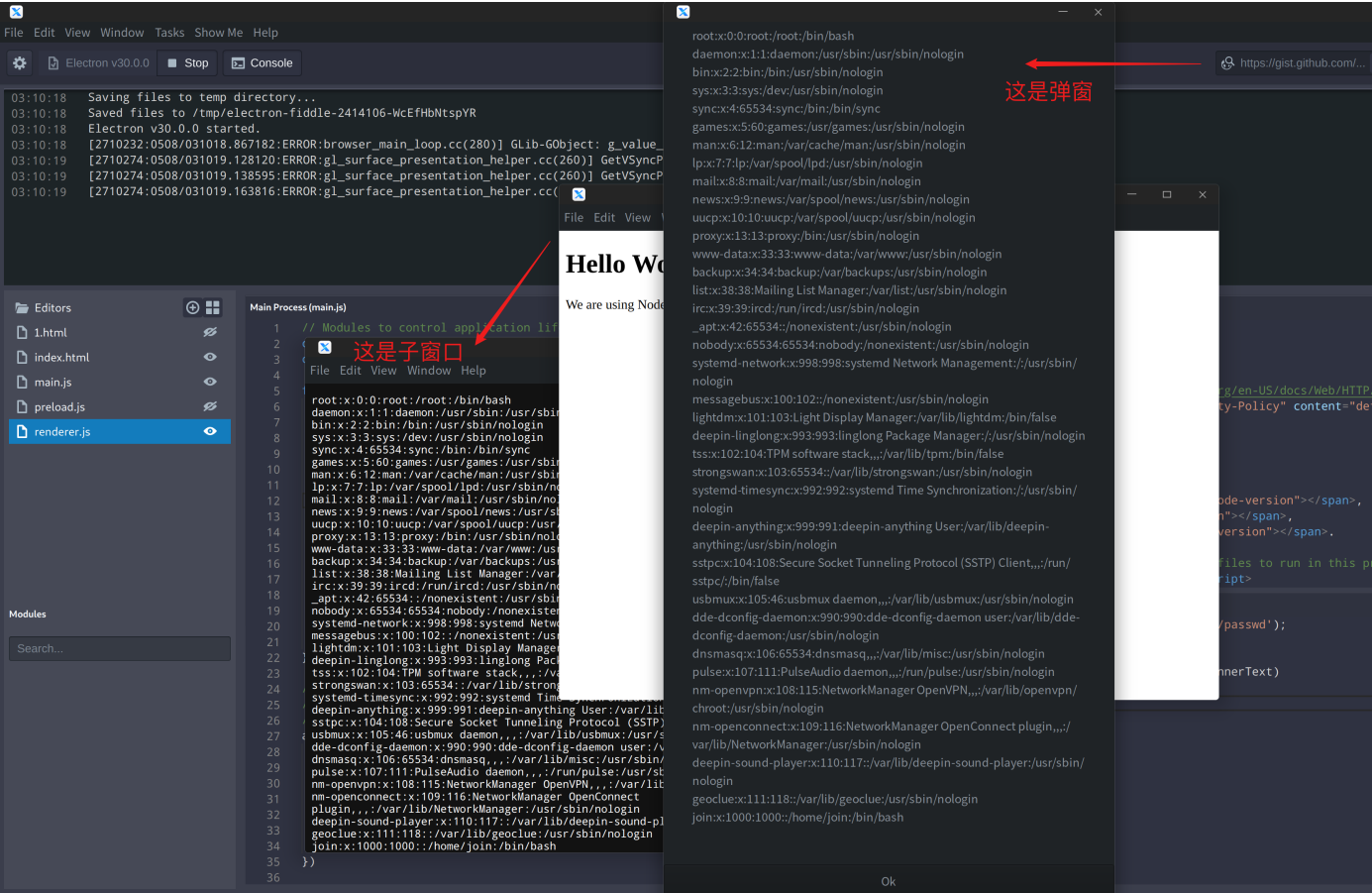
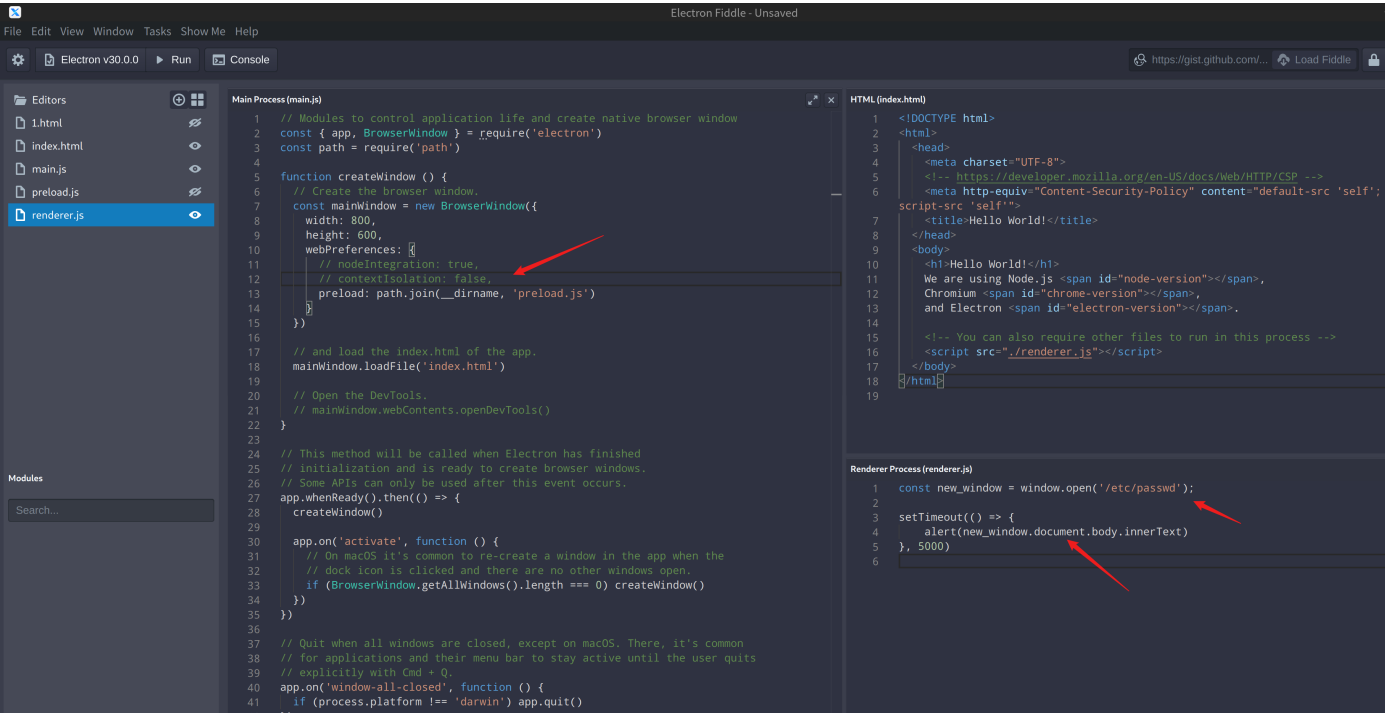
14.0 版本中修复 `iframe + window.open` 创建新窗口绕过安全策略漏洞

`electrovolt` 的文章中，在进行 `Discord RCE` 时，使用 `window.open` 绕过了沙箱，具体操作是 `window.open` 加载和 `Discord` 同源或者允许的网页地址，之后立即通过 `.location` 属性修改当前页面的 `url` 为恶意地址，实现绕过沙箱加载恶意页面

<https://blog.electrovolt.io/posts/discord-rce/>

任意文件读取

在这个案例中，`window.open` 只是一个小工具，用 `iframe` 等标签也可以做到，简单来说就是 `window.open` 支持打开本地文件，大部分程序是通过本地文件创建主窗口的，那刚好同源，就可以通过 `window.open` 的返回对象，获取到读取的内容，之后通过 `javascript` 传递给攻击者，我们通过 `alert` 来证明我们可以获取到值



0x07 window.open 防御手段

`window.open` 执行时会触发 `web-contents-created` 事件的，所以可以在主进程对该事件进行监听，之后进行有效处理

官方给出了一个案例

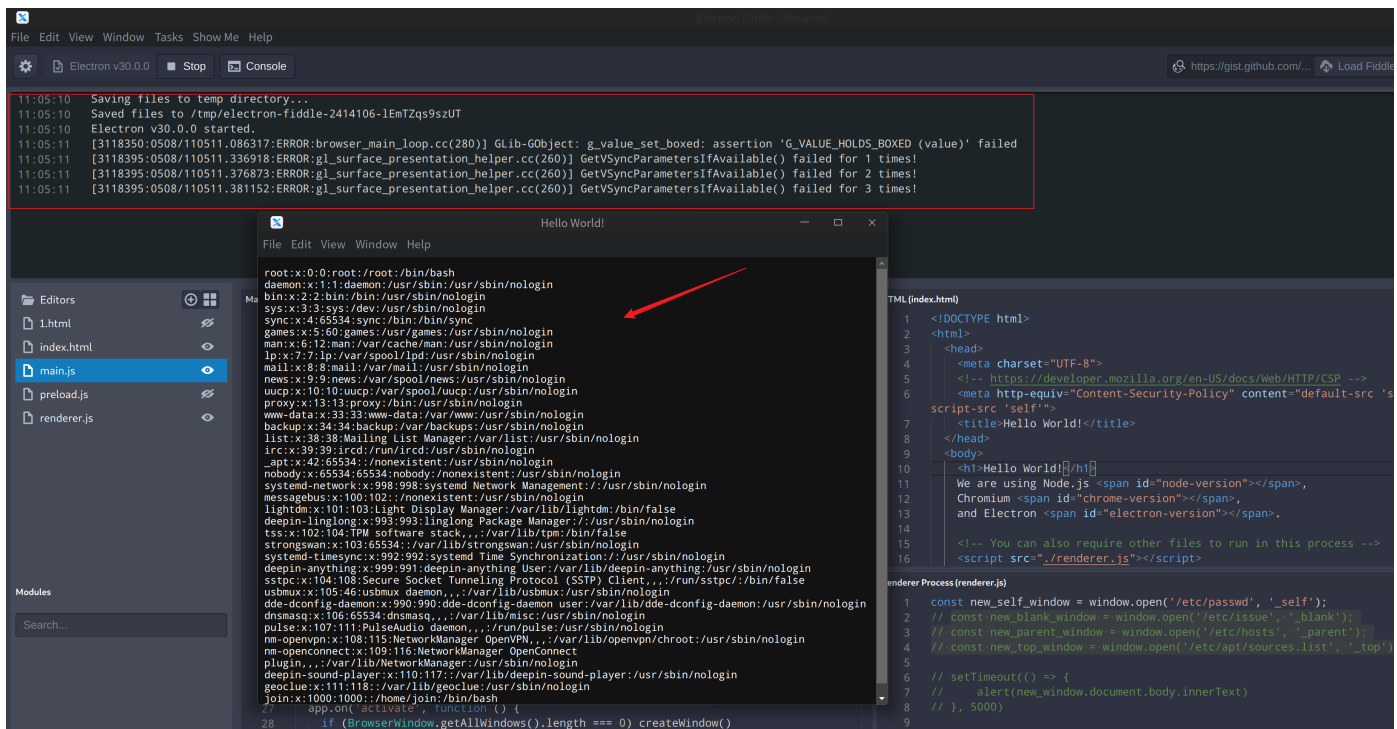
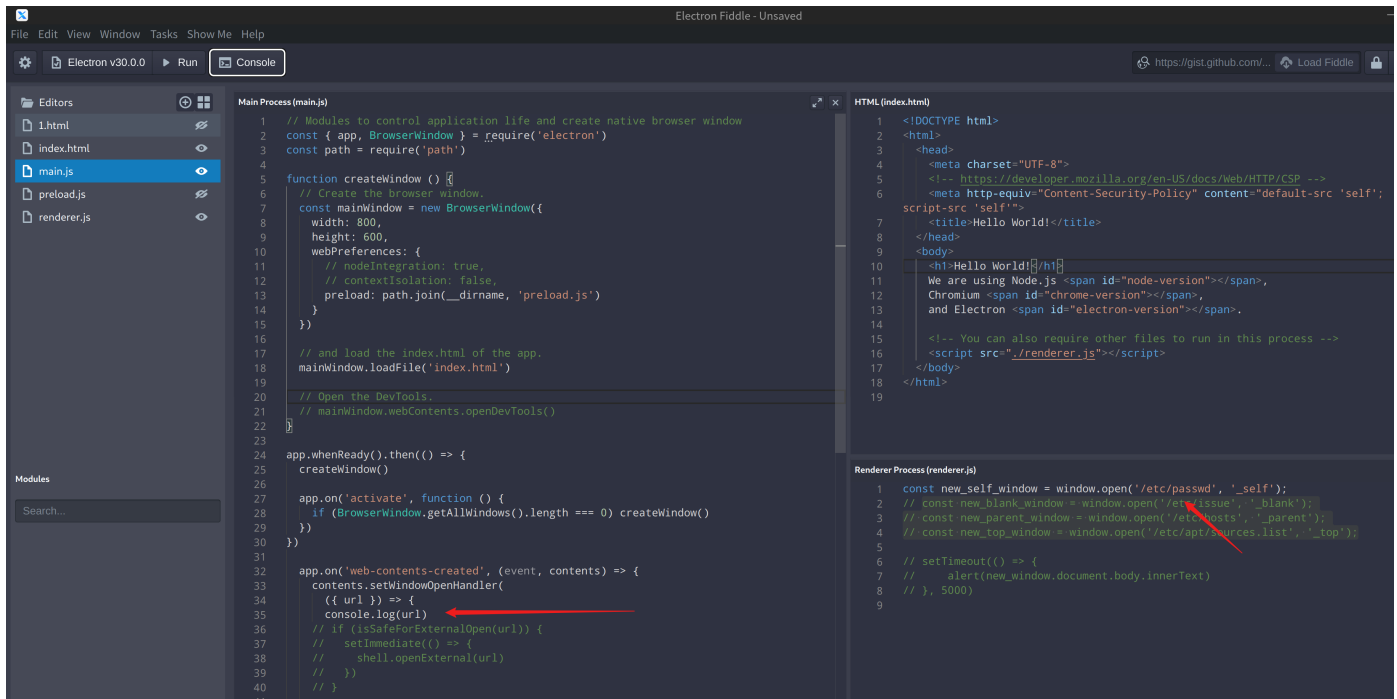
```
const { app, shell } = require('electron')

app.on('web-contents-created', (event, contents) => {
  contents.setWindowOpenHandler(({ url }) => {
    // 在这个例子中，我们要求操作系统
    // 在默认浏览器中打开此事件的URL
    //
    // 关于哪些URL应该被允许通过shell.openExternal打开，
    // 请参照以下项目。
    if (isSafeForExternalOpen(url)) {
      setImmediate(() => {
        shell.openExternal(url)
      })
    }

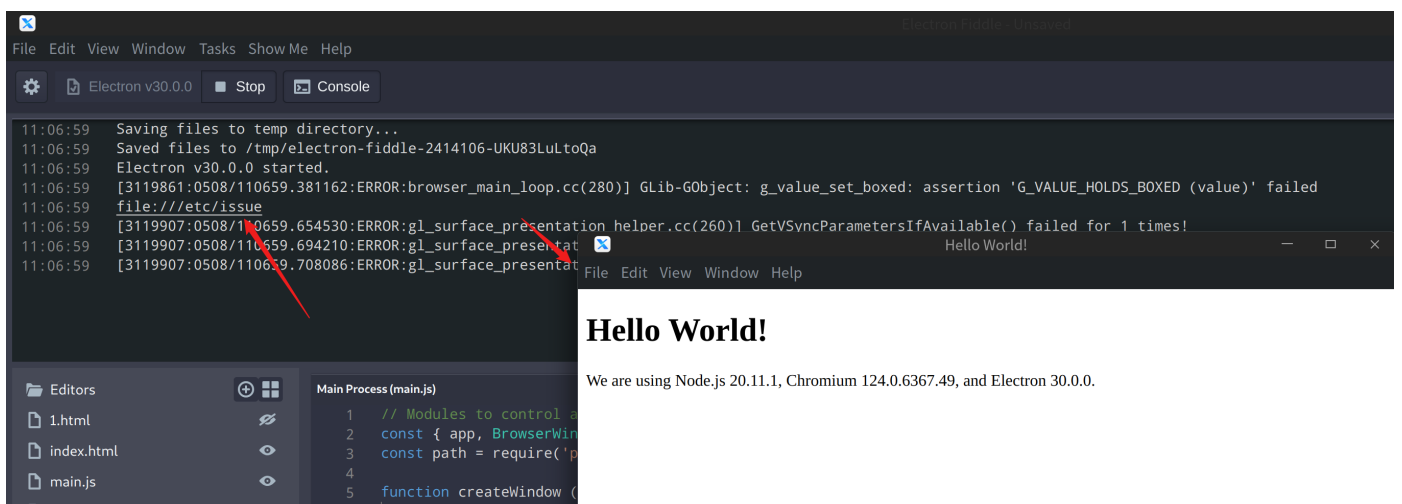
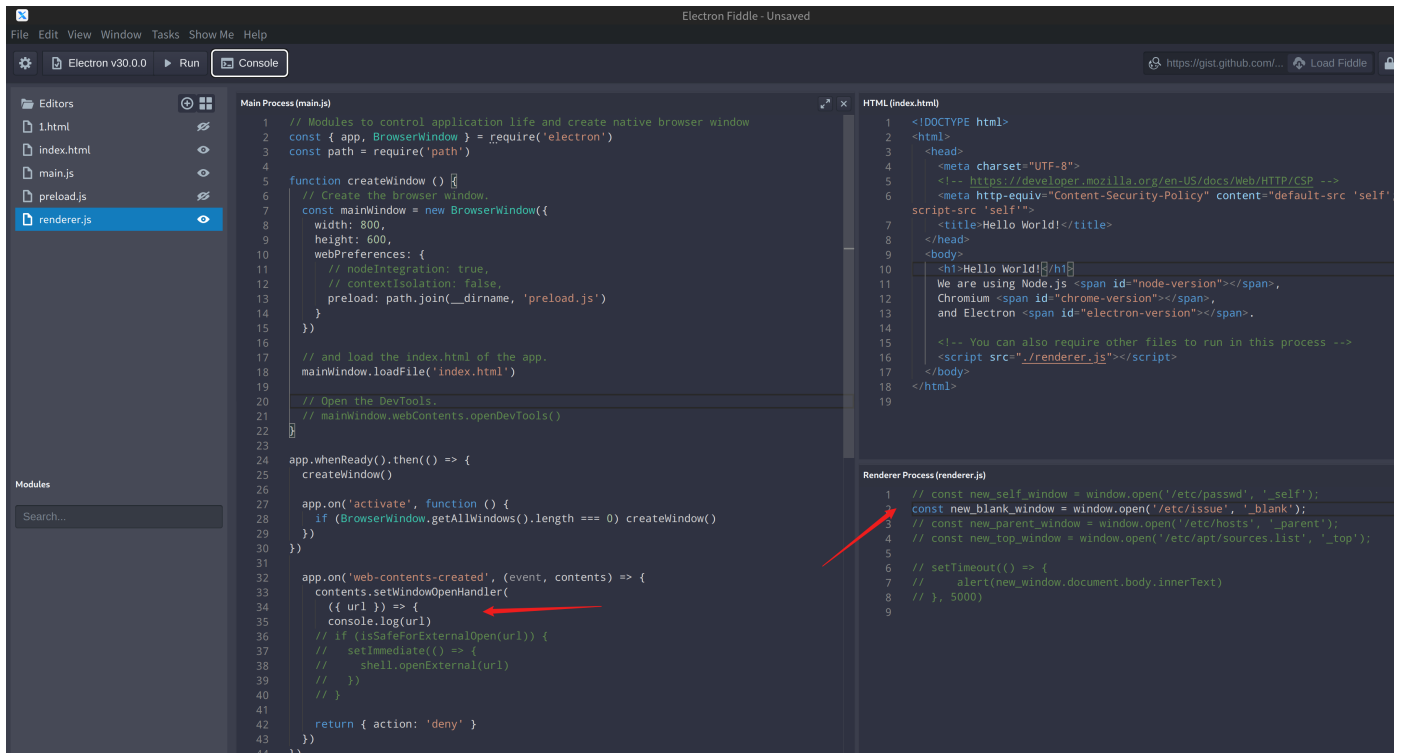
    return { action: 'deny' }
  })
})
```

这个案例检查的是 `url` 是否符合规定，如果符合就使用 `shell.openExternal` 进行打开，不符合就阻止，阻止 `window.open` 的方法是返回 `{ action: 'deny' }`

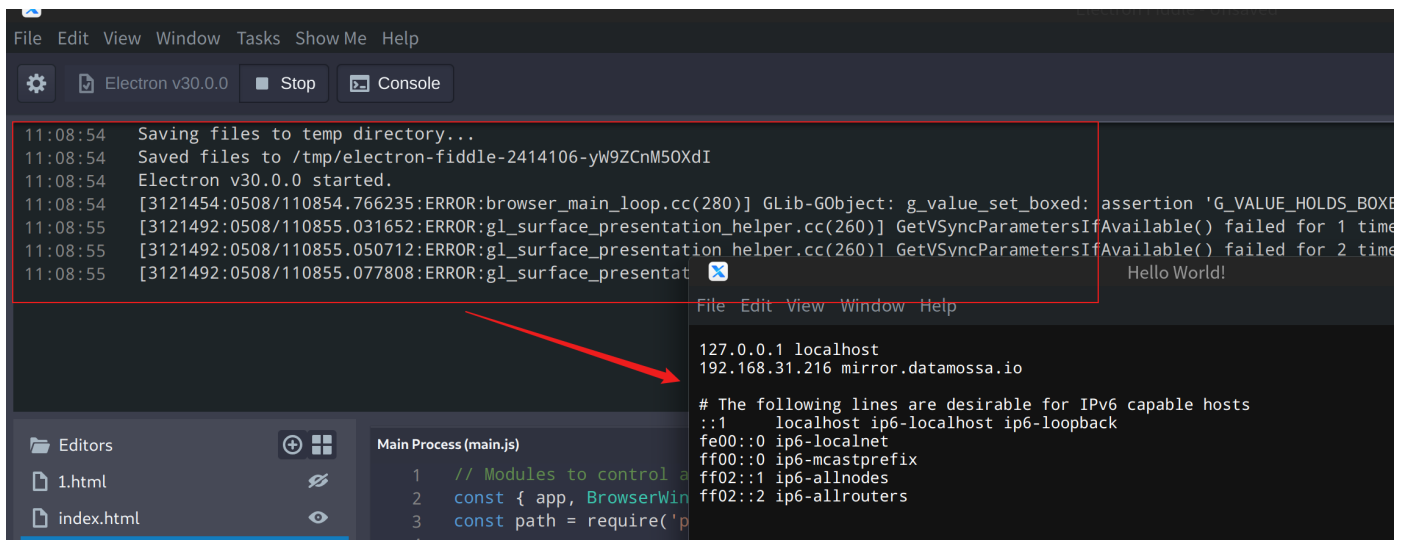
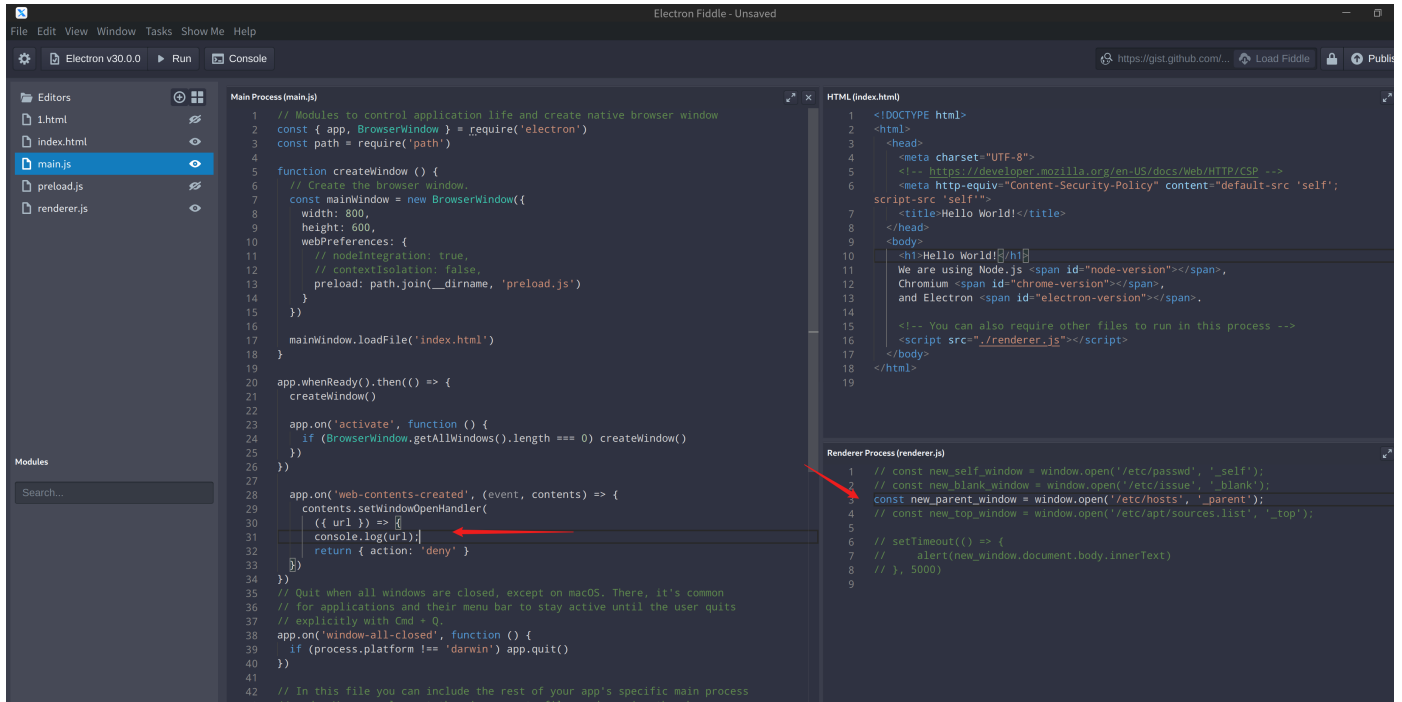
我们测试一下，是否能够监听到 `window.open`，我们就用一个最简单的，主进程控制台打印 `url`，之后拒绝创建新窗口



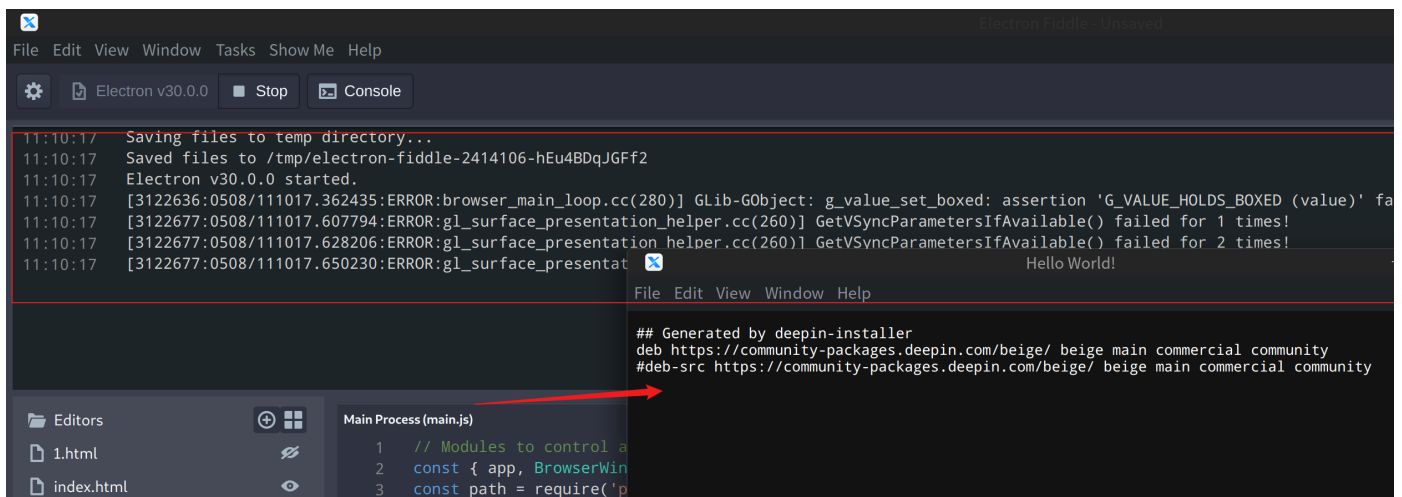
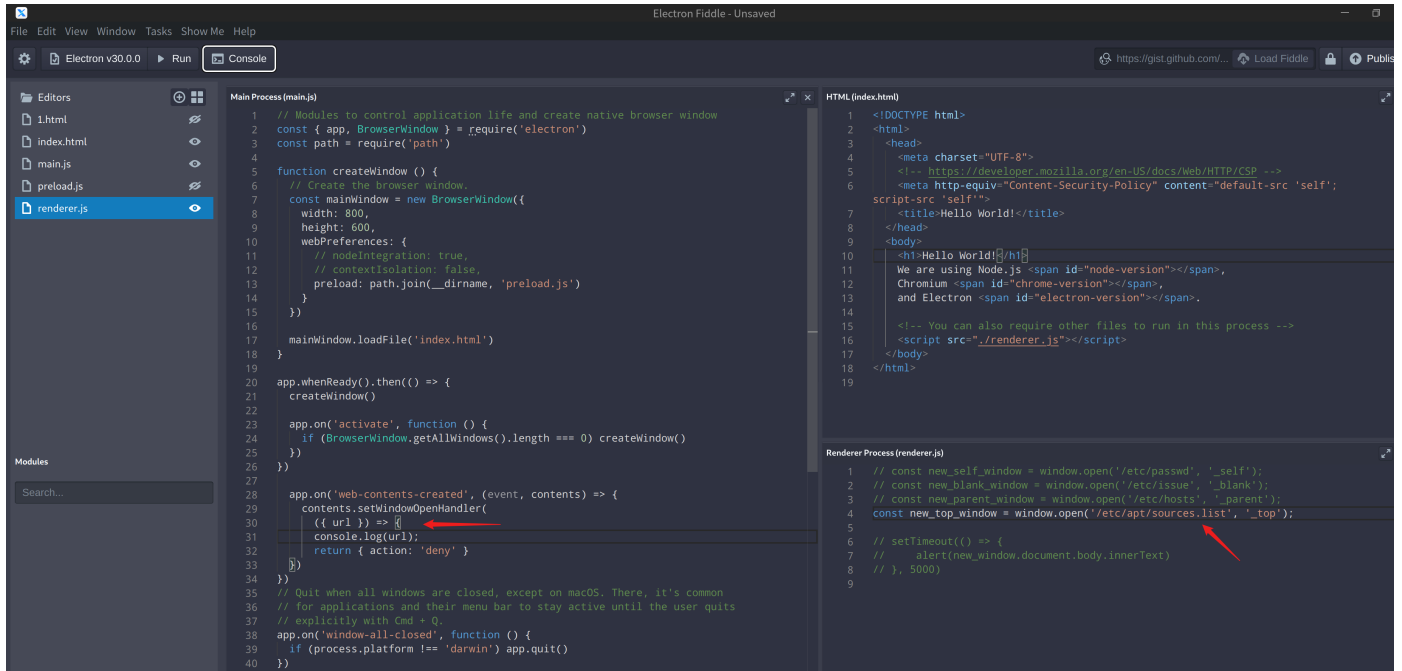
对于 `_self` 没有监听和拦截效果



对于 `_blank` 具备监听和拦截效果



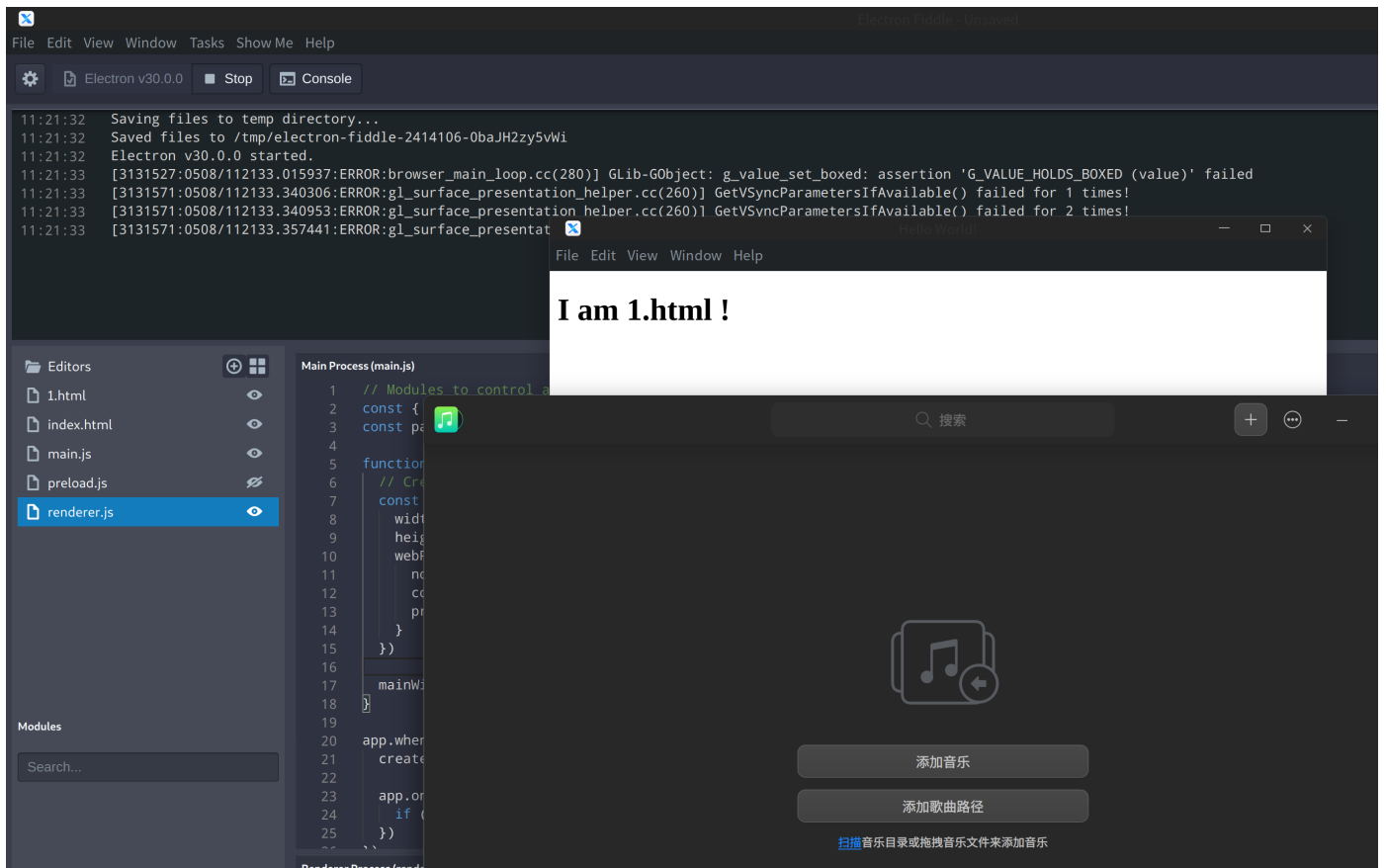
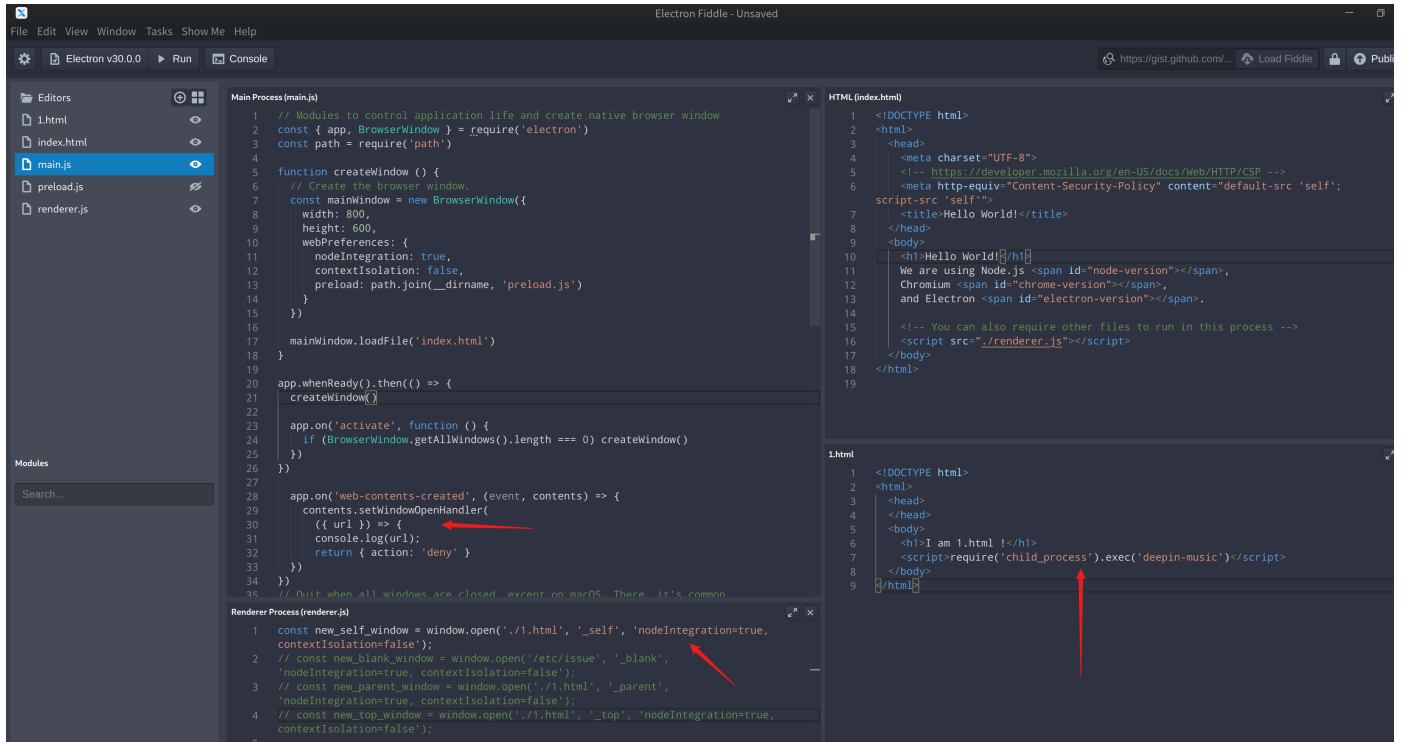
对于 `_parent` 没有监听和拦截



对 `_top` 没有拦截

如果开发者只关注新创建窗口(`_blank`)了, 没有关注其他 `frameName` 的 `window.open` 可能会有一些遗漏, 但这些遗漏会造成危害吗?

我们测试一下遗漏的几种 `frameName(target)` 是否可以配置执行 `Node.js`



`_self` 可以执行 `Node.js`，经过测试，`_parent` 和 `_top` 也是可以的

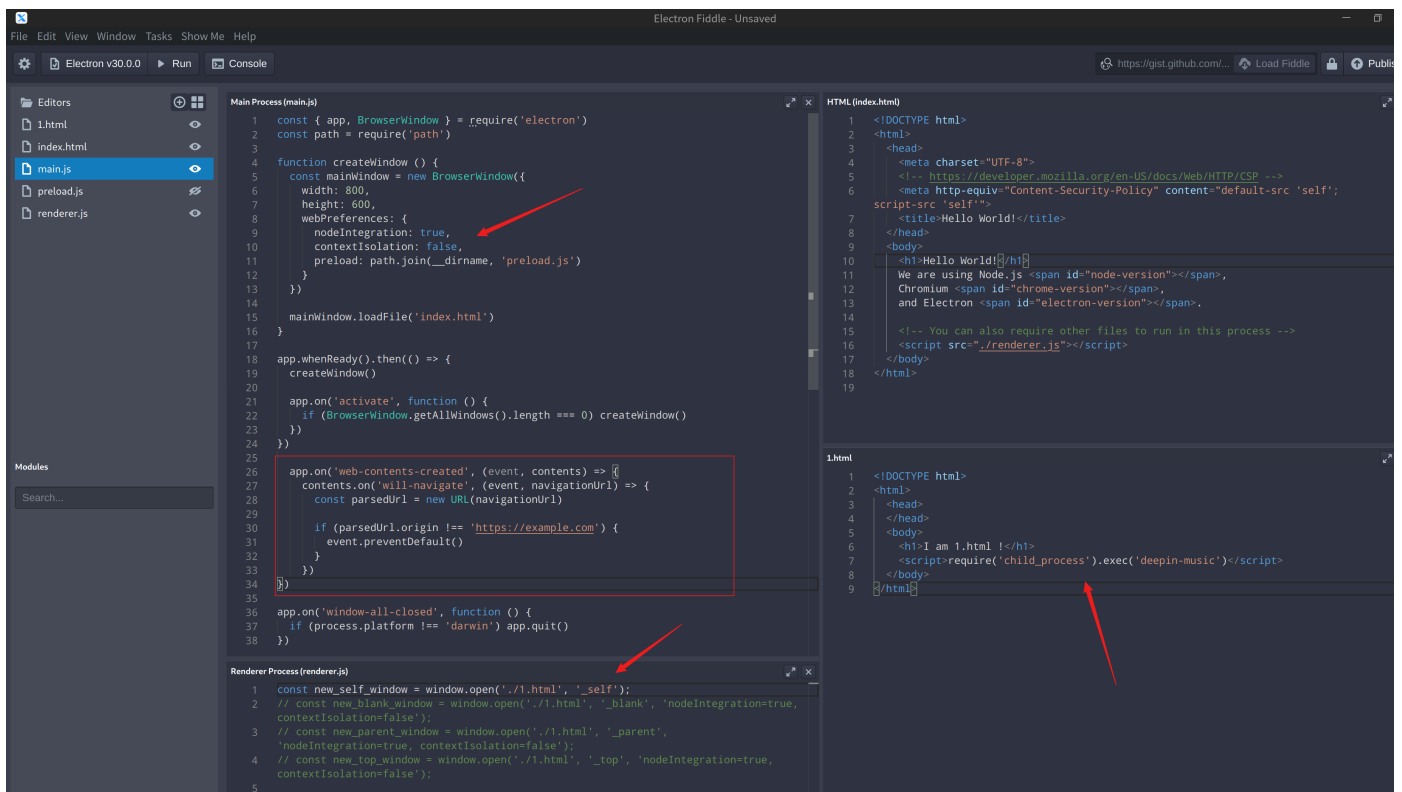
其实这里 `window.open` 不设置 `'nodeIntegration=true, contextIsolation=false'` 也是可以执行的，毕竟是继承父窗口的权限嘛

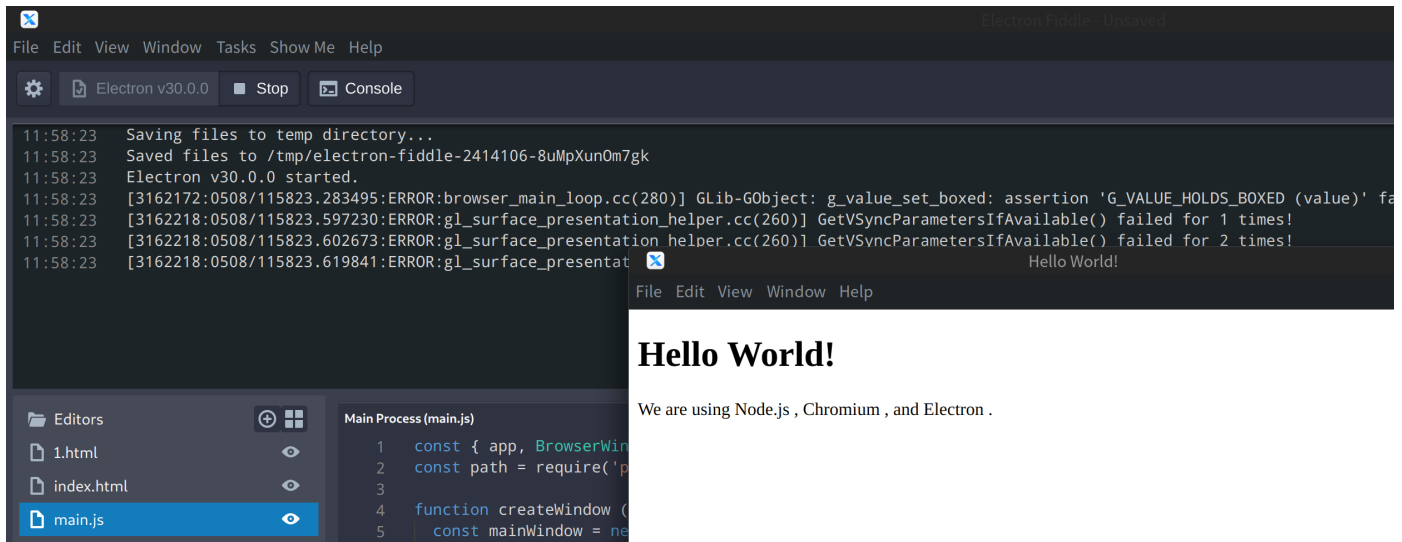
由于这部分是新窗口创建，而当 `frameName(target)` 设置为 `_self`、`_parent` 和 `_top` 都属于是导航范畴，所以 `Electron` 官网给出上面的关于新窗口监听和拦截案例对其是无效的，可以需要参照 `Electron` 中关于导航相关的代码

```
const { URL } = require('url')
const { app } = require('electron')

app.on('web-contents-created', (event, contents) => {
  contents.on('will-navigate', (event, navigationUrl) => {
    const parsedUrl = new URL(navigationUrl)

    if (parsedUrl.origin !== 'https://example.com') {
      event.preventDefault()
    }
  })
})
```

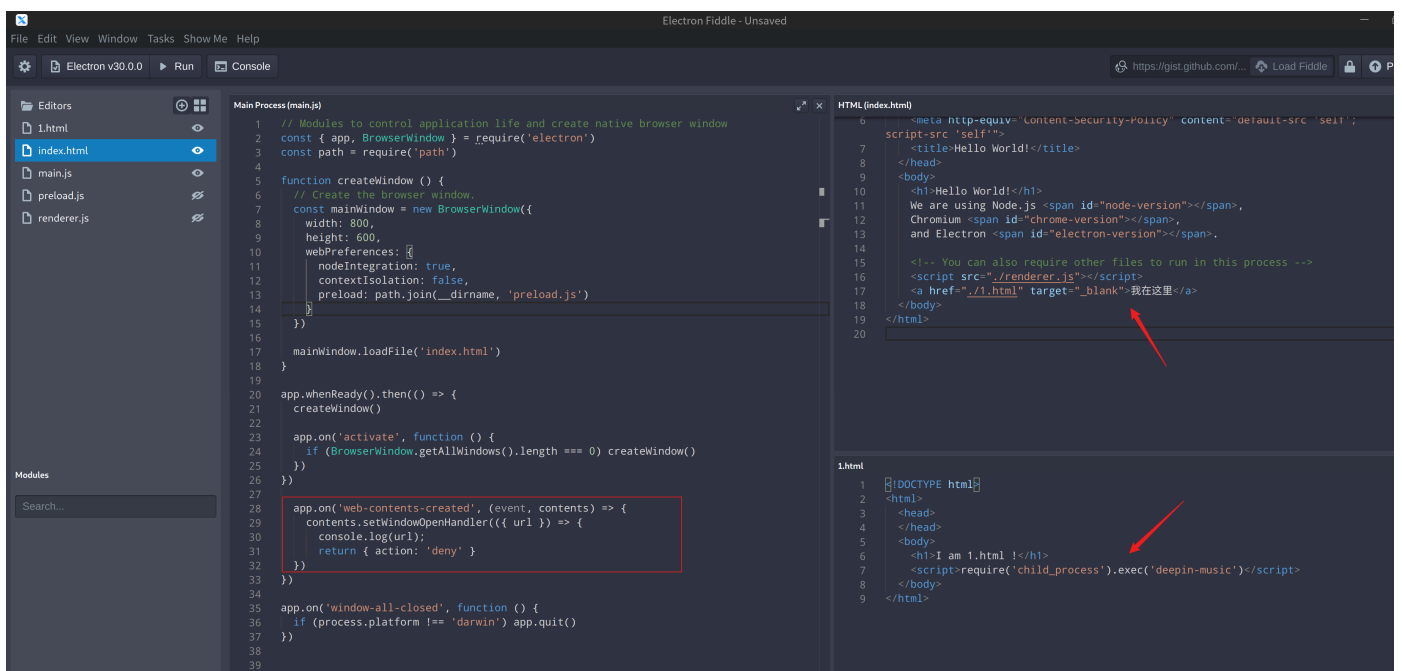


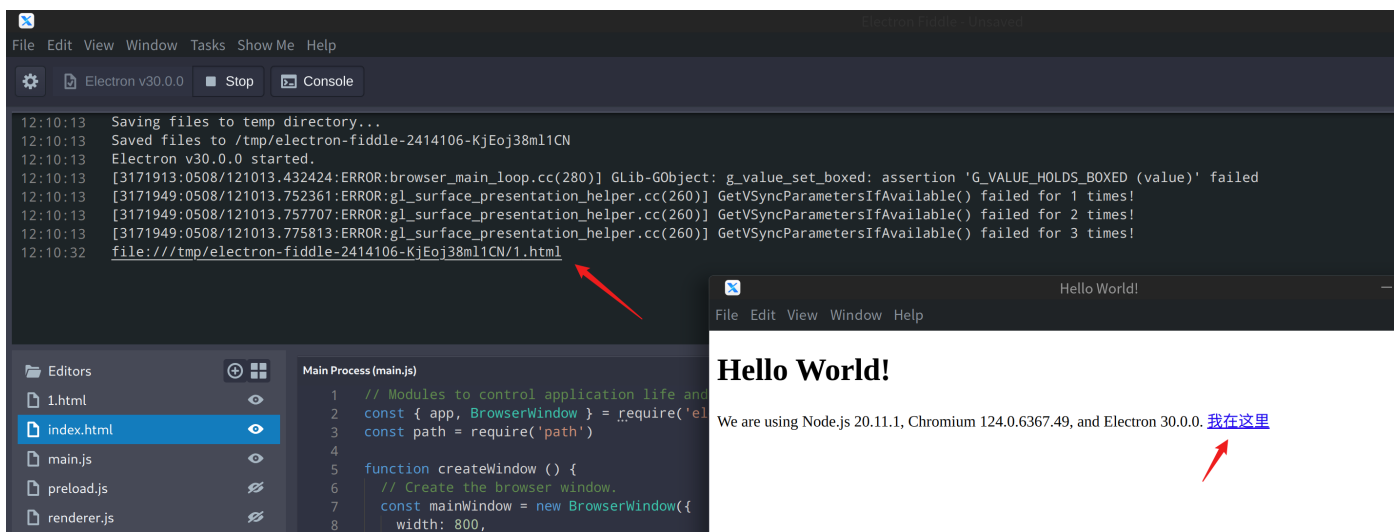
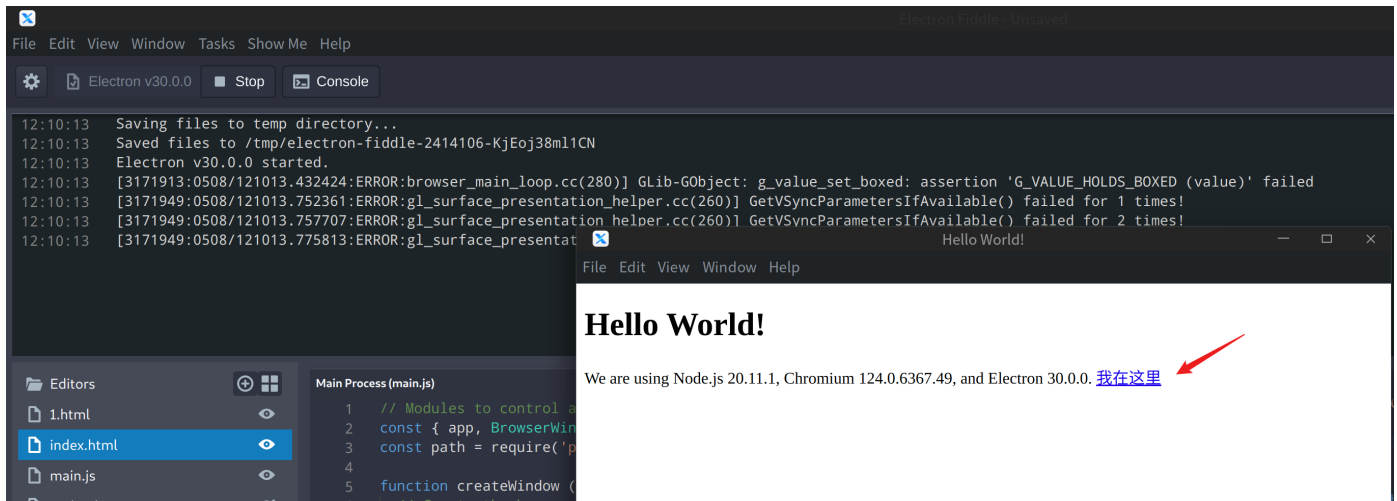


这样在 `frameName(target)` 设置为 `_self`、`_parent` 和 `_top` 时就会被监听和拦截了

经过测试发现，`frameName(target)` 设置为 `_blank` 时也会触发 `'will-navigate'` 事件，但导航事件可能在其他功能中使用到，所以开发者应该同时监听新窗口创建和导航，做更精细化地管理

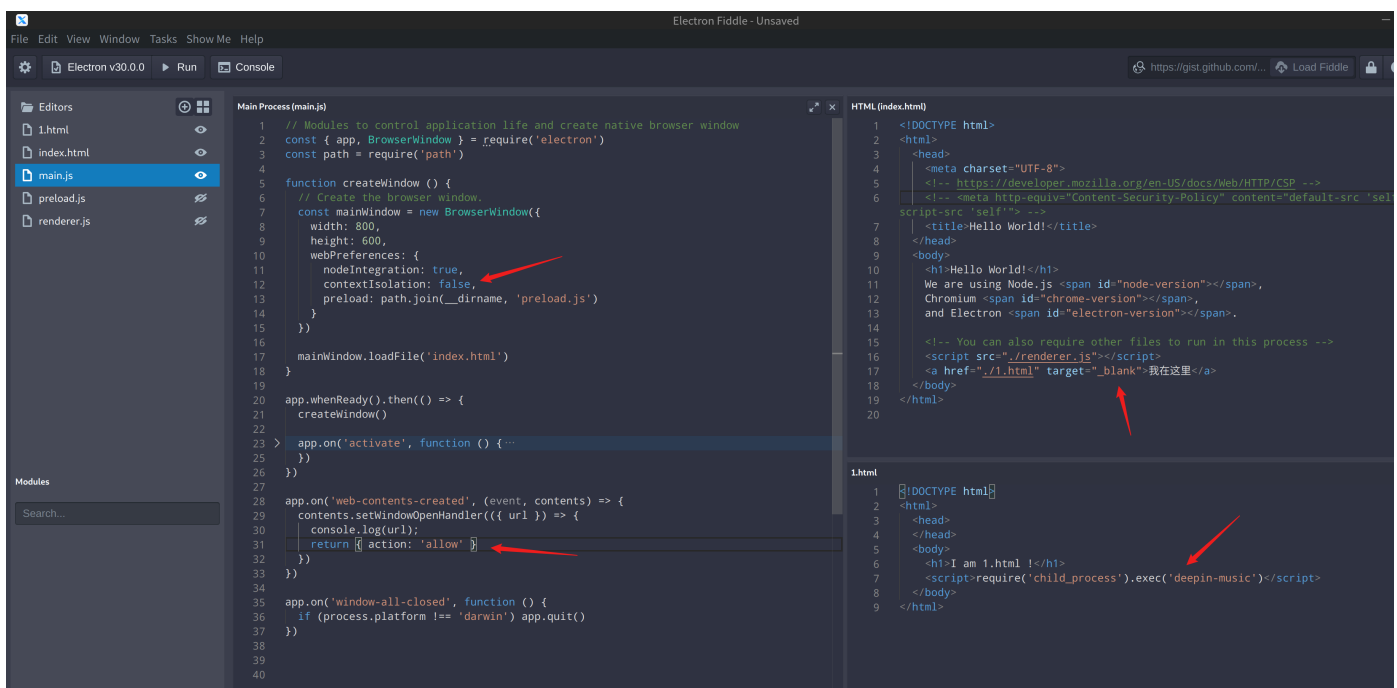
a 标签和 form 标签设置 `target="_blank"` 时会被监听和拦截吗？

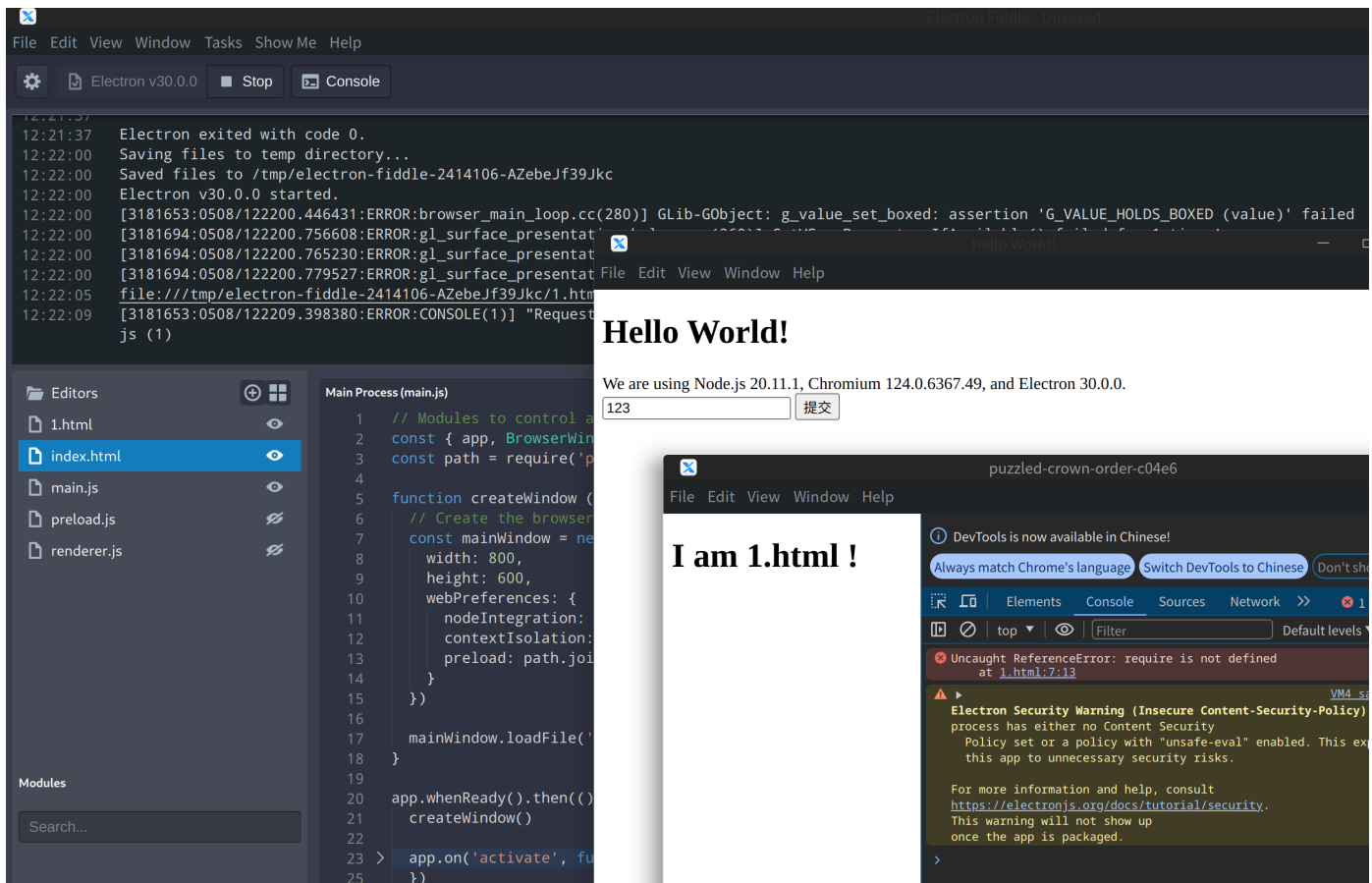




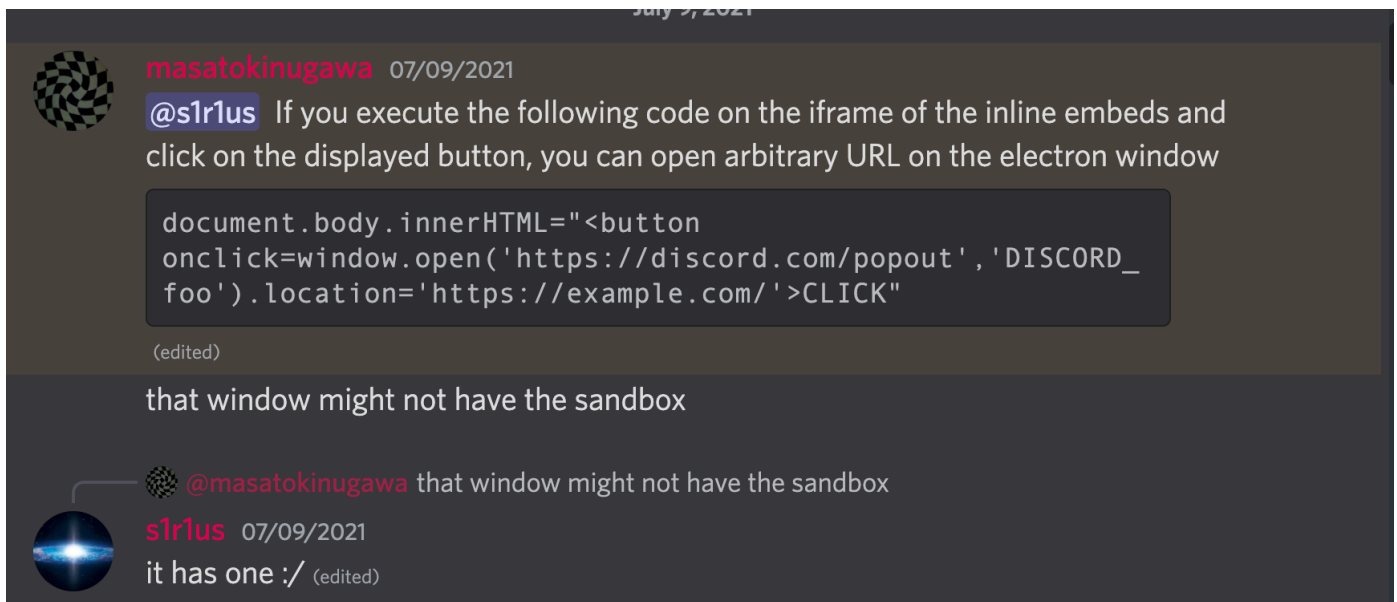
点击链接后，控制台打印要加载的地址，没有新窗口创建，也没有执行 `Node.js`，`'web-contents-created'` 事件成功监听并拦截 `a` 标签创建新窗口的行为

将 `action` 的值设置为 `allow`，即允许创建窗口



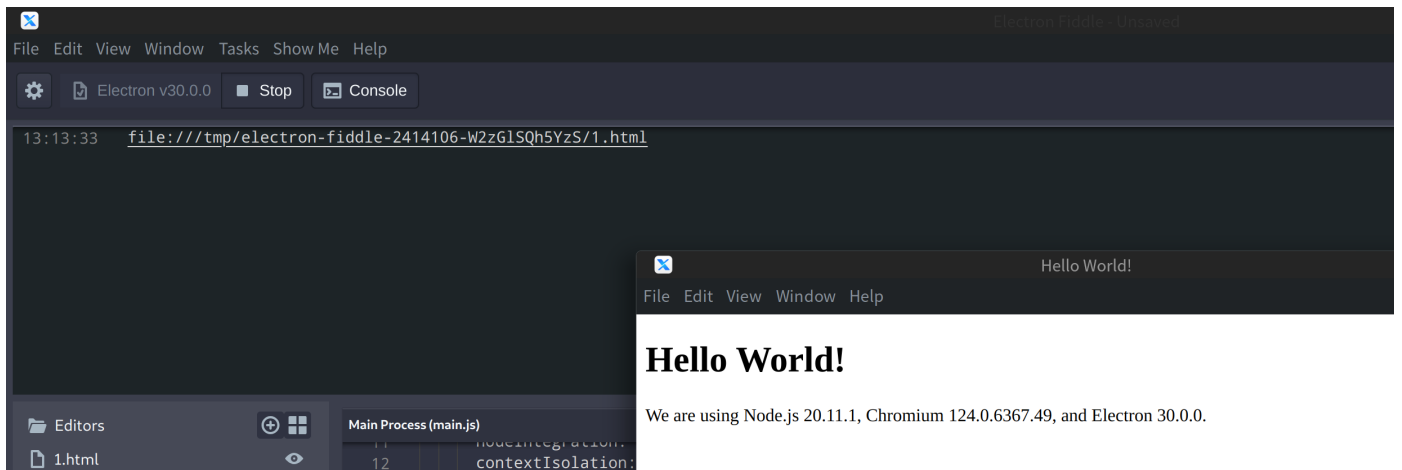
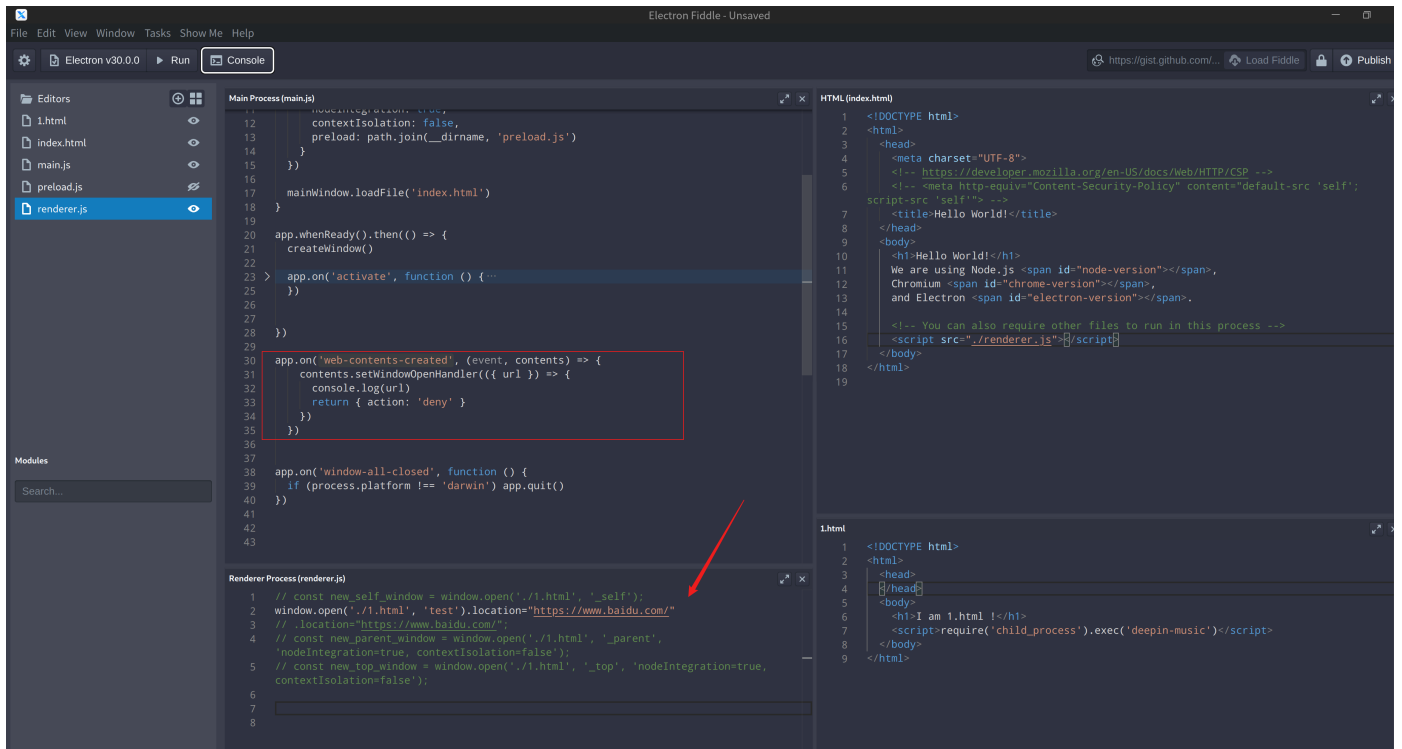


现在我们再来看之前 `electrovolt` 这种 `window.open().location` `payload`



通过 `window.open` 打开一个官方地址，`frameName` 名称不是特殊的名称，会创建新窗口或者利用旧窗口，之后立即跳转到恶意地址

如果使用的是 `'web-contents-created'` 事件监听，应该是可以拦截的



当然，这是 Electron 30.0 版本了，在 10.0.0 版本，代码都会报错，而且据文章描述，Discord 用的是 new-window 事件进行监听的，具体如何做的校验文章也没有描述

具体可以参考以下链接

<https://www.electronjs.org/zh/docs/latest/tutorial/security#14-%E7%A6%81%E7%94%A8%E6%88%96%E9%99%90%E5%88%B6%E6%96%B0%E7%AA%97%E5%8F%A3%E5%88%9B%E5%B%BA>

<https://www.electronjs.org/zh/docs/latest/api/window-open>

Ox08 总结

本篇文章主要是讨论创建新窗口带来的一些危害，测试主要是用的最新版本 `Electron`，我们将创建新窗口分为两类

- 主进程创建新窗口
- 渲染进程创建新窗口

其中主进程创建新窗口可讨论的内容较少，除非攻击者可以控制构造过程中的参数，不然很难发起攻击，大部分都是写死的

渲染进程创建新窗口又可以分为两类

- `window.open` 打开窗口
- `a` 标签和 `form` 标签设置 `target="_blank"` 打开新窗口

其中 `a` 标签和 `form` 标签打开新窗口并不能执行 `Node.js`，危害不是很大

`window.open` 则不同，它打开或重用的窗口默认会继承父窗口的权限，也就是说如果从渲染进程调用 `window.open`，恰巧渲染进程具备执行 `Node.js` 的能力，那么新打开或重用的窗口也会具备 `Node.js` 的能力，除非显式地设置 `features`，限制其能力

在上下文方面，`window.open` 表现与之前的 `iframe` 等基本一致，父子窗口同源情况下可以通过引用获取上下文，非同源就需要 IPC 通信了

`window.open` 不支持打开自签名证书的 https 网站

官方建议不用 `window.open`，同时也给出了一些事件来监听新窗口的创建，`app` 对象监听 `web-contents-created` 事件可以监听到 `window.open` 的行为

当创建新窗口时，并可以自定义验证过程，通过设置 `contents.setWindowOpenHandler` 决定是否创建，

但是如果 `frameName(target)` 设置为 `_self`、`_parent`、`_top`，则 `window.open` 的行为会变成导航行为，此时设置 `contents.setWindowOpenHandler` 就不管用了，导航后的窗口也是继承父窗口权限，会在 `web-contents-created` 事件内部的 `will-navigate` 监听到，并在其中进行处理

`web-contents-created` 对 `a` 标签和 `form` 标签同样有监听和拦截作用，可以使用 `contents.setWindowOpenHandler` 进行处理

开发者在做校验时，需要考虑到 `window.open(xxx).location` 这种情况，做有效验证

