Electron Security

Preload

NOP Team

预加载脚本 | Electron 安全

0x00 提醒

> 之前的一篇Electron 安全与你我息息相关文章非常的长，虽然提供了 `PDF` 版本，但还是导致很多人仅仅是点开看了一下，完读率大概 `7.95%` 左右，但上一篇真的是我觉得很重要的一篇，对大家了解 `Electron` 开发的应用程序安全有帮助，与每个人切实相关
>
> 但是上一篇文章内容太多，导致很多内容粒度比较粗，可能会给大家造成误解，因此我们打算再写一些文章，**一来是将细节补充清楚**，二来是再此来呼吁大家注意 `Electron` 安全这件事，如果大家不做出反应，应用程序的开发者是不会有所行动的，这无异于在电脑中埋了一些地雷

我们公众号开启了留言功能，欢迎大家留言讨论～

这篇文章也提供了 `PDF` 版本及 `Github` ，见文末

0x01 简介

相信看了前面的文章，大家对于预加载脚本已经非常了解了，对于之前篇章中已经测试并解释清楚的部分，不会再次详细解释

预加载脚本（`Preload`）是一个比较让我意外的内容，可能因为学习 `Electron` 时就使用了官网推荐的安全开发案例，所以一直以为预加载脚本的 `Node.js` 就是被限制过的，但是随着最近的几篇文章的实验发现并不是

在 `sandbox` 没有被设置为 `true` 时(`Electron 20.0` 版本开始默认值为 `true`)，预加载脚本是拥有完整 `Node.js` 环境的，如果在 `Preload` 中如果定义并暴露了不安全的方法，而开发者对于预加载脚本的能力并不了解可能会带来危害

0x02 预加载脚本中的Node.js

> https://www.electronjs.org/zh/docs/latest/tutorial/tutorial-preload

预加载脚本的意义在于完成主进程和渲染进程之间的联络，因此重要逻辑不应该在预加载脚本中进行，也不应该赋予其过于繁重的责任，完成主进程与渲染进程之间的通信，将通信结果传递给另一方才是它实际的意义，通过暴露方法使这种固定的逻辑可以被渲染进程调用

因此预加载脚本在渲染器加载网页之前注入，也就是说预加载脚本中的内容会先一步定义好，以供网页中的 `JavaScript` 正确调用

如果没有被沙盒化，预加载脚本肯定是可以任意调用模块的，但是如果被沙盒化后，预加载脚本还可以加载哪些模块呢?

可用的 API	详细信息
Electron 模块	渲染进程模块
Node.js 模块	`events`、`timers`、`url`
Polyfilled 的全局模块	`Buffer`、`process`、`clearImmediate`、`setImmediate`

events

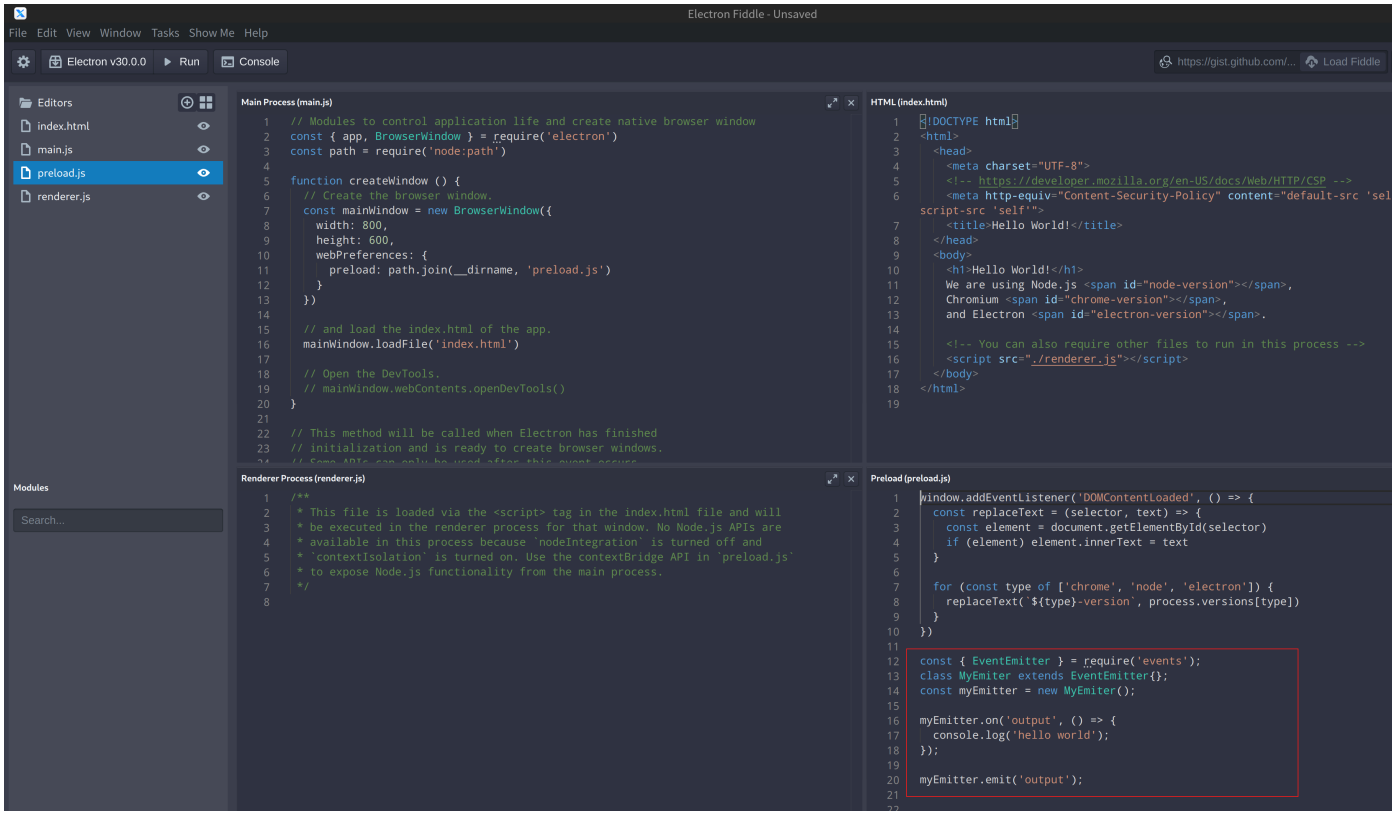
> https://nodejs.org/api/events.html

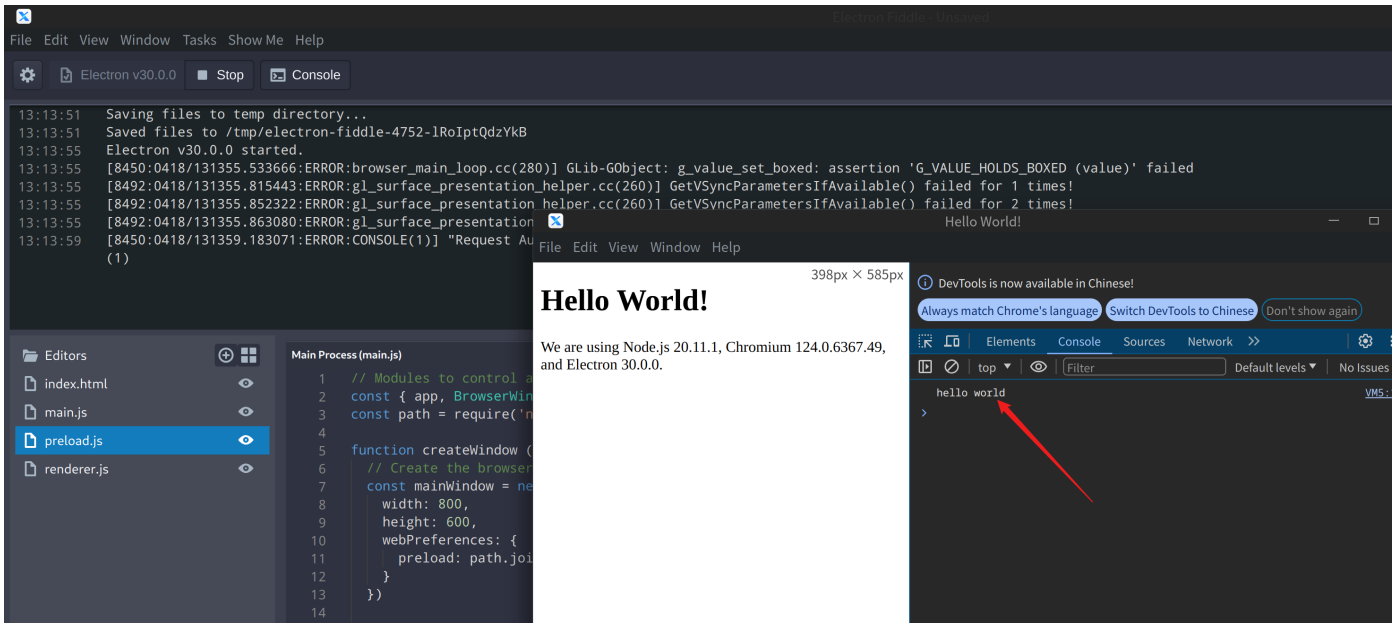
这个模块是 `Node.js` 中关于事件处理的模块，是发布、订阅模式的实现，这里允许预加载脚本使用应该是为了让预加载脚本具备事件处理相关的能力，从预加载脚本的职责来看，也确实可能用的到

使用案例如下

```
const { EventEmitter } = require('events');
class MyEmiter extends EventEmitter{};
const myEmitter = new MyEmiter();

myEmitter.on('output', () => {
  console.log('hello world');
});

myEmitter.emit('output');
```

> https://juejin.cn/post/7038496188965847070





timers

> https://nodejs.org/api/timers.html

这是一个 `Node.js` 的定时器模块，这个模块公开了一个全局 `API`，用于调度在将来某个时间段调用的函数。因为计时器函数是全局函数，所以不需要调用 `require('timers')` 来使用API。

我列几个函数大家肯定会比较熟悉

- setImmediate
- setInterval
- setTimeout

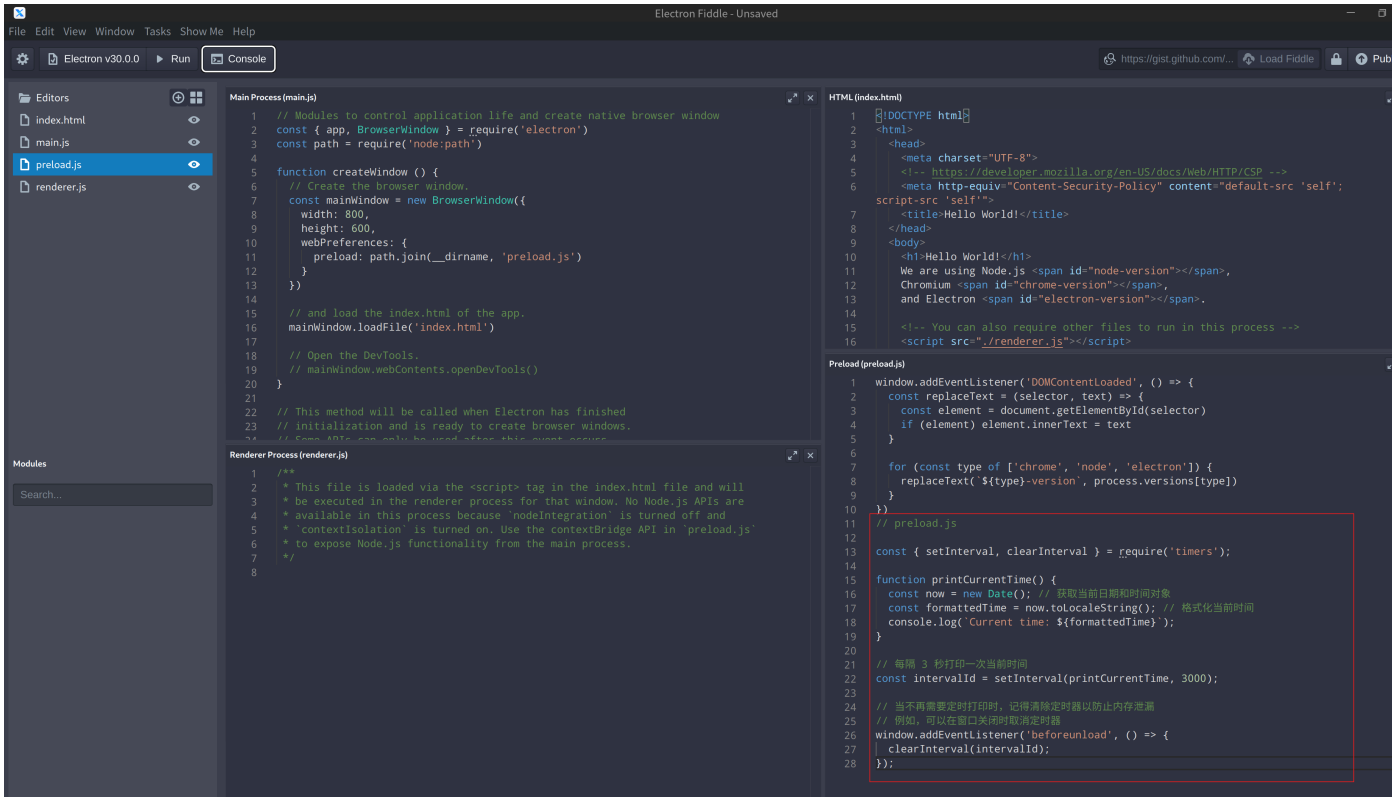
还有上面对应的取消操作

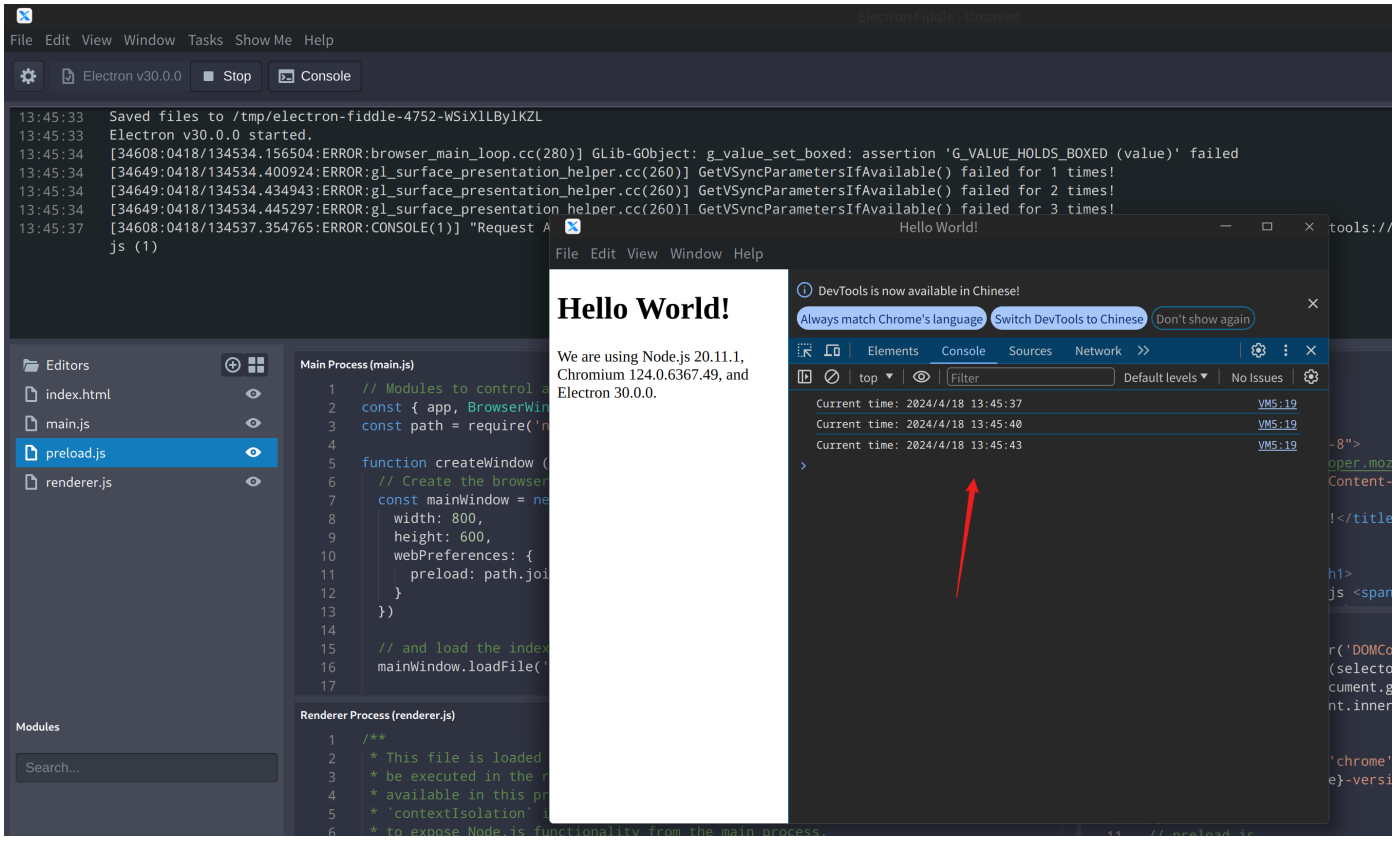
- clearImmediate
- clearInterval
- clearTimeout

这几个函数都是决定一段逻辑在什么时候执行，怎么执行， `setImmediate` 是在当前事件循环迭代结束时立即执行； `setTimeout` 指定时间后执行， `setInterval` 是定期执行

比较容易表现的肯定是 `setInterval` ，我们让控制台每隔 `3` 秒打印一下时间

```
// preload.js

const { setInterval, clearInterval } = require('timers');

function printCurrentTime() {
  const now = new Date(); // 获取当前日期和时间对象
  const formattedTime = now.toLocaleString(); // 格式化当前时间
  console.log(`Current time: ${formattedTime}`);
}

// 每隔 3 秒打印一次当前时间
const intervalId = setInterval(printCurrentTime, 3000);

// 当不再需要定时打印时，记得清除定时器以防止内存泄漏
// 例如，可以在窗口关闭时取消定时器
window.addEventListener('beforeunload', () => {
  clearInterval(intervalId);
});
```


Electron Fiddle - Unsaved
File Edit View Window Tasks Show Me Help
Electron v30.0.0
Run
Console
Editors
index.html
main.js
preload.js
renderer.js
Modules
Search...
Main Process (main.js)
// Modules to control application life and create native browser window
const { app, BrowserWindow } = require('electron')
const path = require('node:path')
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('index.html')
// Open the DevTools.
// mainWindow.webContents.openDevTools()
}
// This method will be called when Electron has finished
// initialization and is ready to create browser windows.
Renderer Process (renderer.js)
/**
* This file is loaded via the <script> tag in the index.html file and will
* be executed in the renderer process for that window. No Node.js APIs are
* available in this process because `nodeIntegration` is turned off and
* `contextIsolation` is turned on. Use the contextBridge API in `preload.js`
* to expose Node.js functionality from the main process.
*/
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>
Preload (preload.js)
window.addEventListener('DOMContentLoaded', () => {
const replaceText = (selector, text) => {
const element = document.getElementById(selector)
if (element) element.innerText = text
}
for (const type of ['chrome', 'node', 'electron']) {
replaceText(`${type}-version`, process.versions[type])
}
})
// preload.js
const { setInterval, clearInterval } = require('timers');
function printCurrentTime() {
const now = new Date(); // 获取当前日期和时间对象
const formattedTime = now.toLocaleString(); // 格式化当前时间
console.log(`Current time: ${formattedTime}`);
}
// 每隔 3 秒打印一次当前时间
const intervalId = setInterval(printCurrentTime, 3000);
// 当不再需要定时打印时，记得清除定时器以防止内存泄漏
// 例如，可以在窗口关闭时取消定时器
window.addEventListener('beforeunload', () => {
clearInterval(intervalId);
});


File Edit View Window Tasks Show Me Help
Electron v30.0.0
Stop
Console
13:45:33 Saved files to /tmp/electron-fiddle-4752-WSiXlLBylKZL
13:45:33 Electron v30.0.0 started.
13:45:34 [34608:0418/134534.156504:ERROR:browser_main_loop.cc(280)] GLib-GObject: g_value_set_boxed: assertion 'G_VALUE_HOLDS_BOXED (value)' failed
13:45:34 [34649:0418/134534.400924:ERROR:gl_surface_presentation_helper.cc(260)] GetVSyncParametersIfAvailable() failed for 1 times!
13:45:34 [34649:0418/134534.434943:ERROR:gl_surface_presentation_helper.cc(260)] GetVSyncParametersIfAvailable() failed for 2 times!
13:45:34 [34649:0418/134534.445297:ERROR:gl_surface_presentation_helper.cc(260)] GetVSyncParametersIfAvailable() failed for 3 times!
13:45:37 [34608:0418/134537.354765:ERROR:CONSOLE(1)] "Request A
js (1)
Hello World!
File Edit View Window Help
Hello World!
We are using Node.js 20.11.1, Chromium 124.0.6367.49, and Electron 30.0.0.
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
Current time: 2024/4/18 13:45:37 VM5:19
Current time: 2024/4/18 13:45:40 VM5:19
Current time: 2024/4/18 13:45:43 VM5:19

url

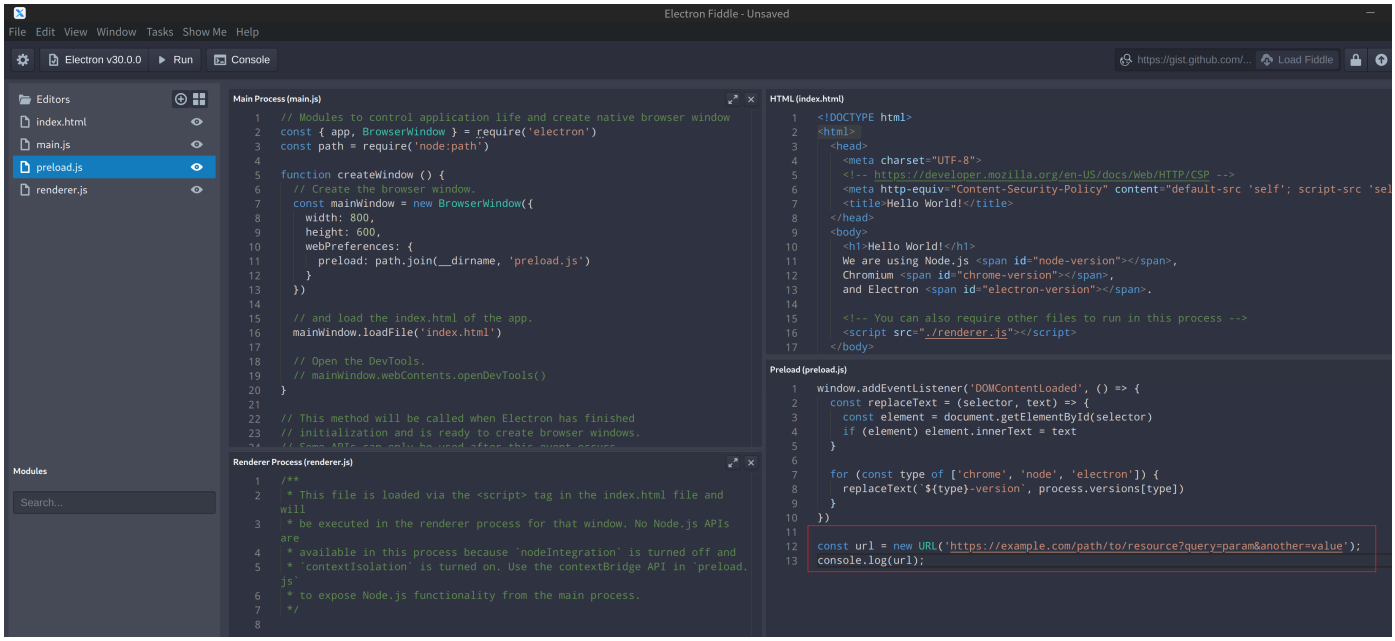
> https://nodejs.org/api/url.html

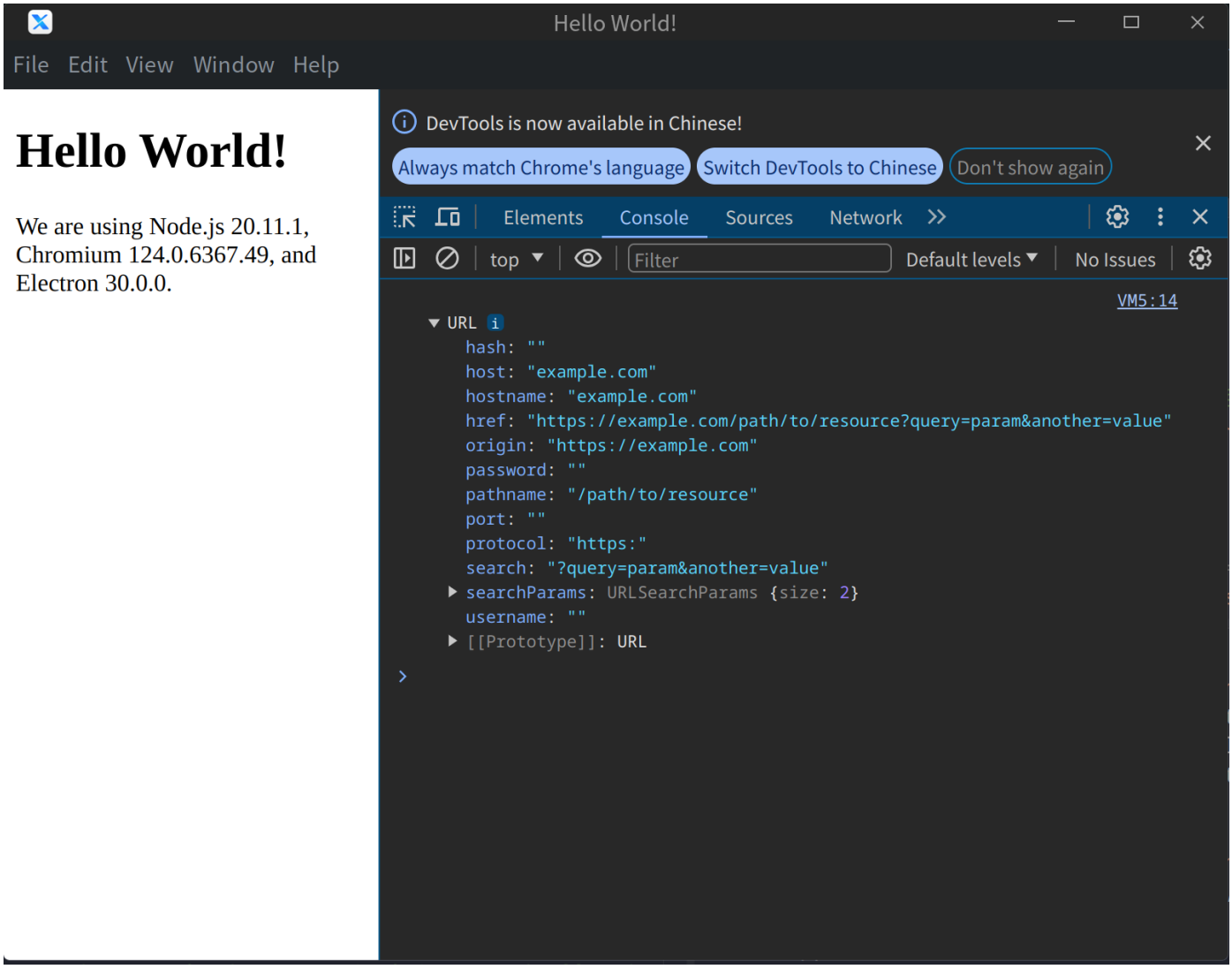
顾名思义，用来处理 `url` 相关功能的模块

- new URL()
- format()
- resolve()

以解析一个 `url` 为例

```
// preload.js

const url = new URL('https://example.com/path/to/resource?
query=param&another=value');
console.log(url);
```





接下来应该是一些原本渲染进程没有或不完整而补充进来的一些方法

Buffer

> https://nodejs.org/api/buffer.html

Buffer 对象用于表示固定长度的字节序列，这个模块应该是用来处理渲染页面与二进制数据交互的场景

例如

```
// preload.js

const { contextBridge } = require('electron');

contextBridge.exposeInMainWorld('myAPI', {
  bufferFromUtf8: (str) => Buffer.from(str, 'utf8'),
  bufferToString: (buf, encoding = 'utf8') => buf.toString(encoding),
  // ... 其他 Buffer 相关方法
});

// 在渲染进程中，可以通过 window.myAPI 来访问预加载脚本提供的方法
```

我看很多 `V8` 漏洞的 `Payload` 都会使用到 `Buffer` ，看起来似乎是与二进制数据处理离不开的模块

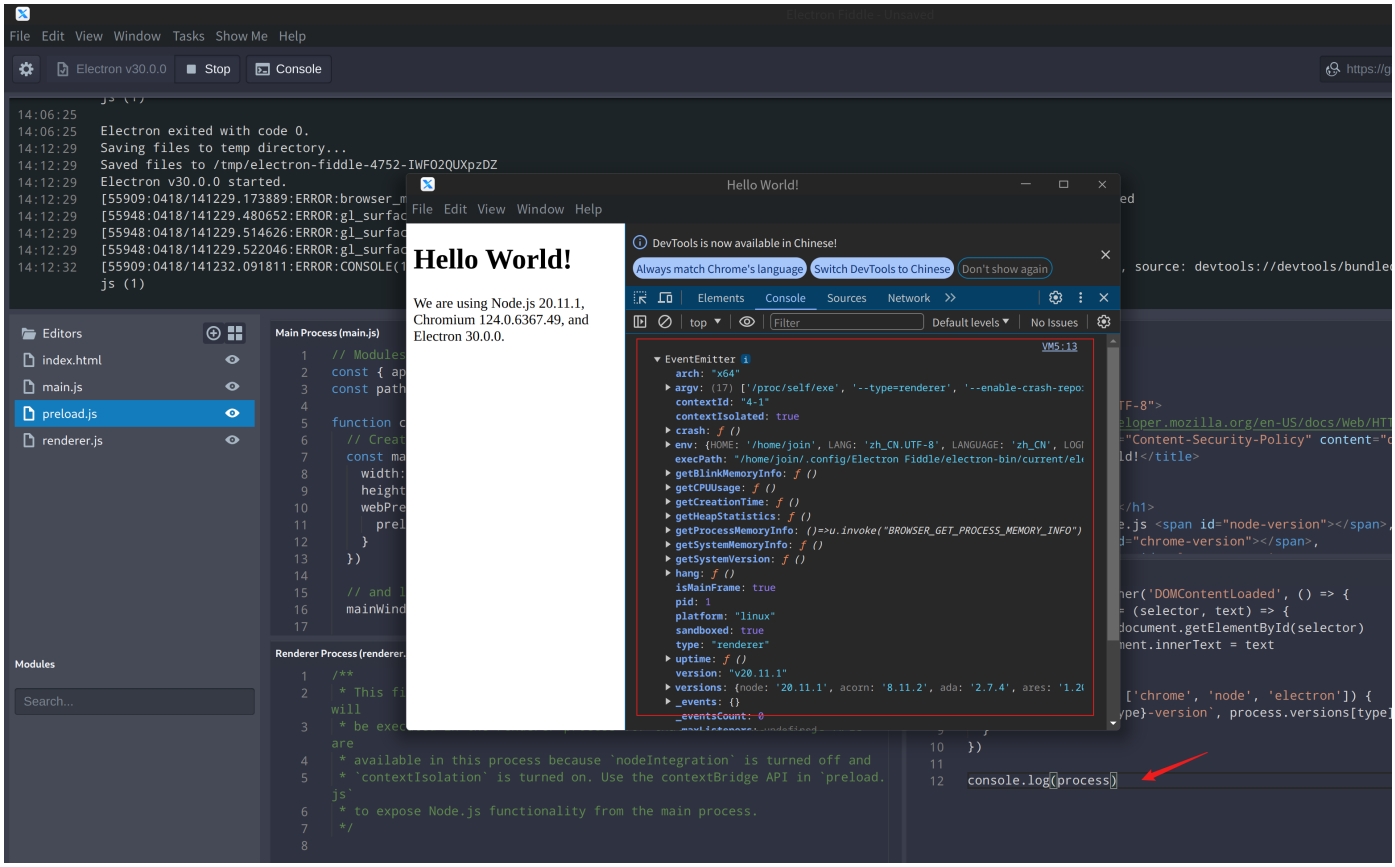
process

> https://www.electronjs.org/zh/docs/latest/api/process

这个模块用来处理对象的扩展，官方的案例中获取 `Electron` 、 `Node.js` 、 `Chromium` 版本就是使用的这个模块，可以通过 `process` 模块获取一些信息，具体如下

- `crash()`
- `hang()`
- `getCreationTime()`
- `getHeapStatistics()`
- `getBlinkMemoryInfo()`
- `getProcessMemoryInfo()`
- `getSystemMemoryInfo()`
- `getSystemVersion()`
- `getCPUUsage()`
- `getIOCounters()`
- `uptime()`
- `argv`

- `execPath`
- `env`
- `pid`
- `arch`
- `platform`
- `沙盒化`
- `contextIsolated`
- `type`
- `version`
- `versions`
- `mas`
- `windowsStore`
- `contextId`



直接打印 `process` 可以直接看到它的属性和方法

setImmediate

上面已经提到

clearImmediate

上面已经提到

electron

- `contextBridge`
- `crashReporter`
- `ipcRenderer`
- `nativeImage`
- `webFrame`
- `webUtils`

这里的 `contextBridge` 是用来向渲染进程暴露变量/常量和函数的方法，在下面的部分详细介绍； `ipcRenderer` 是 `Preload` 脚本用来和主进程进行 `IPC` 通信的工具，我们详细看看剩下几个是干嘛的

crashReporter

> 将崩溃日志提交给远程服务器
>
> https://www.electronjs.org/zh/docs/latest/api/crash-reporter

```
const { crashReporter } = require('electron')

crashReporter.start({ submitURL: 'https://your-domain.com/url-to-submit' })
```

如何构建崩溃日志收集系统可以点击上方的链接

nativeImage

> 使用 PNG 或 JPG 文件创建托盘、dock和应用程序图标。
>
> https://www.electronjs.org/zh/docs/latest/api/native-image

```
const { BrowserWindow, Tray } = require('electron')

const tray = new Tray('/Users/somebody/images/icon.png')
const win = new BrowserWindow({ icon: '/Users/somebody/images/window.png'
})
```

这个在开发过程中就会遇到，但不太理解为什么开放给 `preload`

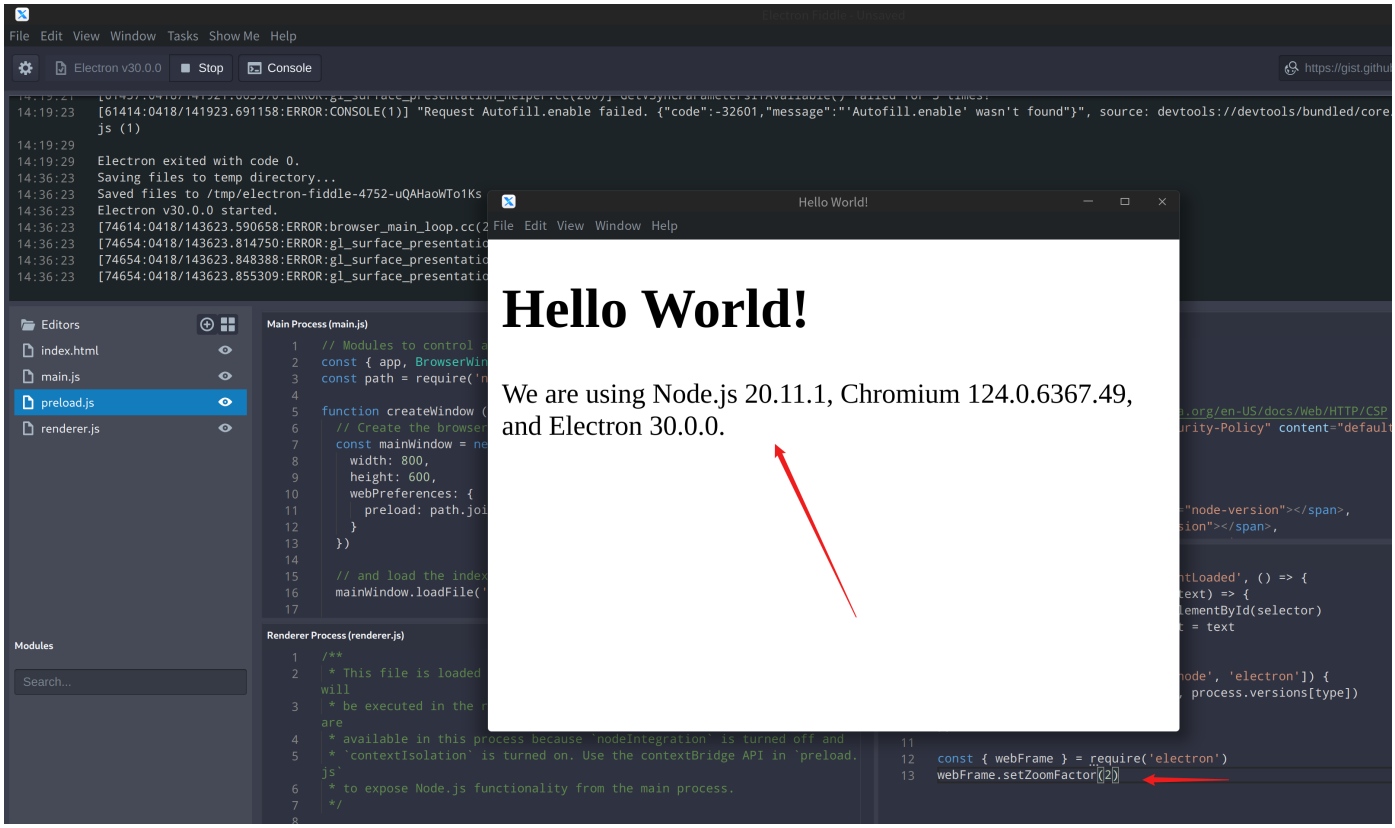
webFrame

> 自定义渲染当前网页

这个不难理解，如果不通过 `webFrame` 也可以通过 `DOM` 等操纵网页

例如将当前页缩放到200% 的示例

```
const { webFrame } = require('electron')
webFrame.setZoomFactor(2)
```



很方便，确实大了

webUtils

> 与Web API对象（文件、Blob等）交互的实用程序层
>
> https://www.electronjs.org/zh/docs/latest/api/web-utils

例如获取文件路径

```
const { webUtils } = require('electron')
const newPath =
webUtils.getPathForFile(document.querySelector('input').files[0])
```

小提醒

可以关注一下这几个模块的漏洞通告，如果出现漏洞，可能会影响到 `Electron`

0x03 风险点

`Preload` 可以说是平衡风险和便捷的一种措施，本身已经做得不错了，风险点也都是开发者不安全编码造成的

- 未开启上下文隔离及 `sandbox`
- 不安全的实现
- 接口过度暴露

第一点就不多说了，前面的文章已经说清楚了，主要说后面两点，在后面两点中，我们的前提是开启了上下文隔离，开启了 `sandbox`

1. 不安全的实现

开启了安全措施后， `Preload` 自己是很难造成大的问题，主要是配合主进程，举个极端一些的例子

渲染进程可以读取 `docs` 目录下的文件，文件名由调用者提供， `preload.js` 与主进程通信，读取并返回内容

`main.js`

```
const { app, BrowserWindow, ipcMain } = require('electron');
const fs = require('fs');
const path = require('path')

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js'),
    },
  });

  win.loadFile('./index.html');
}

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

```
  ipcMain.handle('readFile', async (event, filePath) => {
    try {
      filePath = path.join(__dirname, filePath)
      const data = await fs.promises.readFile(filePath, 'utf-8');
      return data;
    } catch (err) {
      console.error('Error reading file:', err);
      return null;
    }
  });
  createWindow();
});

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

preload.js

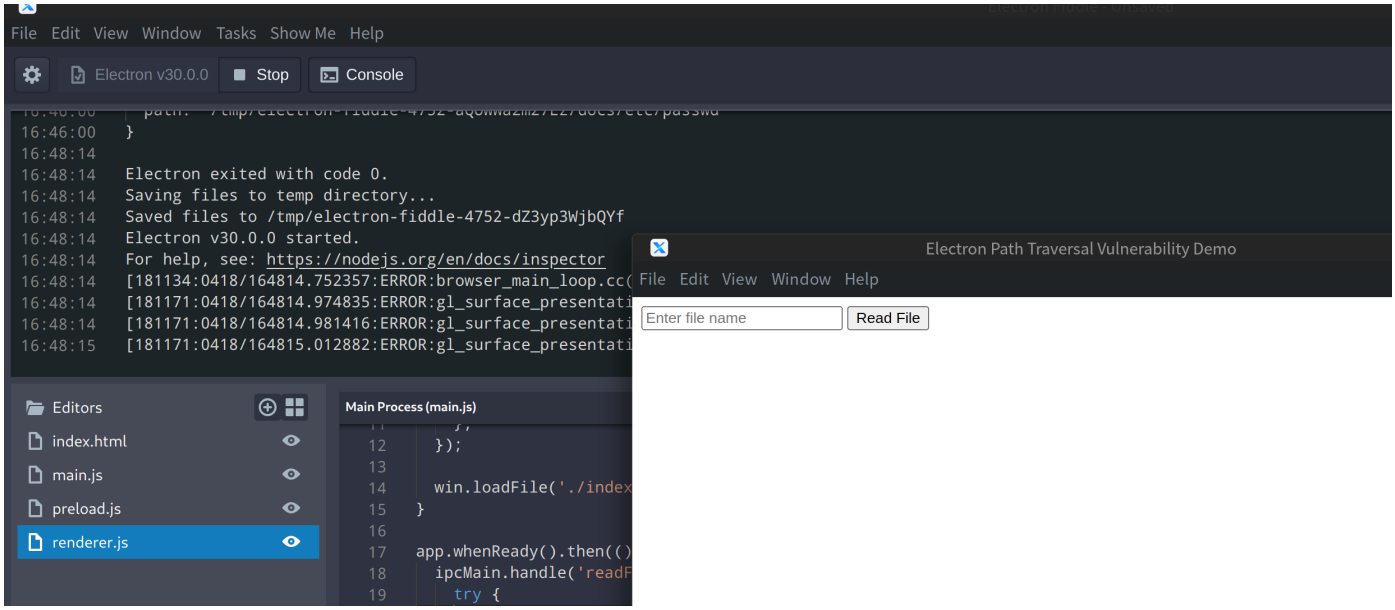
```
const { contextBridge, ipcRenderer } = require('electron');

contextBridge.exposeInMainWorld('myApi', {
  readFile: async (fileName) => {
    try {
      const data = await ipcRenderer.invoke('readFile',
`docs/${fileName}`);
      return data;
    } catch (error) {
      console.error('Error invoking "readFile":', error);
      return null;
    }
  },
});
```

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Electron Path Traversal Vulnerability Demo</title>
</head>
<body>
  <input type="text" id="fileNameInput" placeholder="Enter file name">
  <button id="readFileButton">Read File</button>

  <pre id="fileContent"></pre>

  <script src="./renderer.js"></script>
</body>
</html>
```

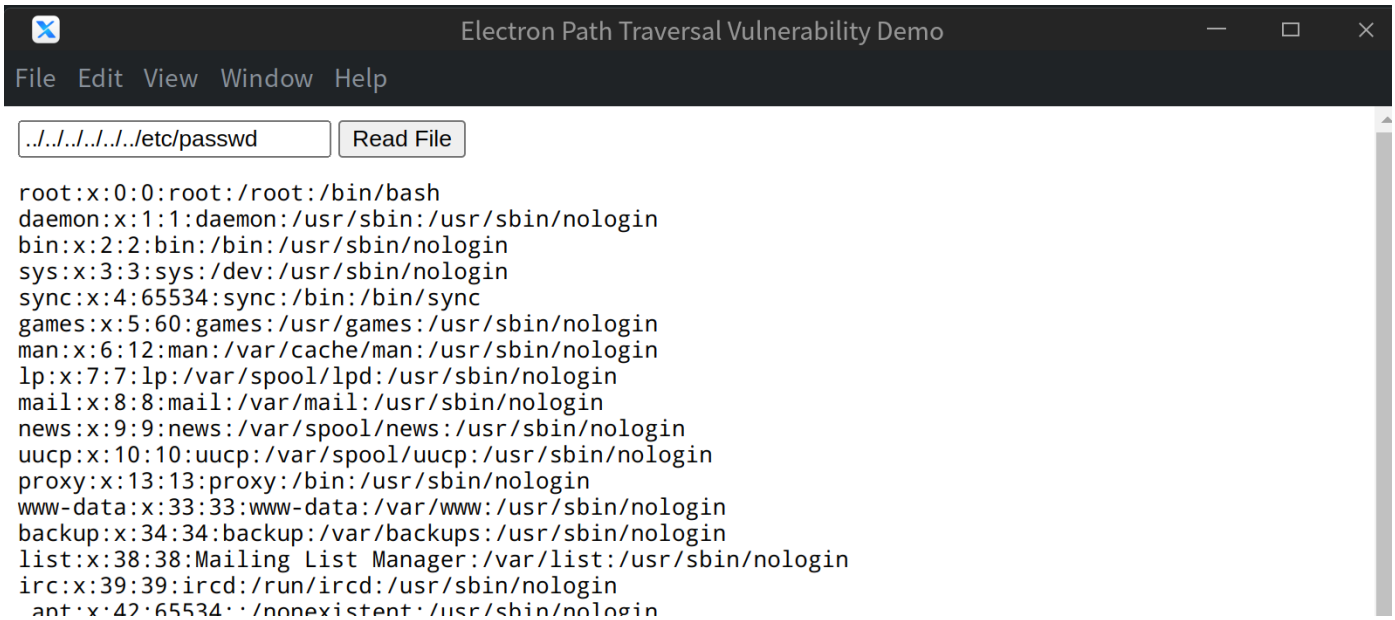
renderer.js

```javascript
const fileNameInput = document.getElementById('fileNameInput');
const readFileButton = document.getElementById('readFileButton');
const fileContent = document.getElementById('fileContent');

readFileButton.addEventListener('click', async () => {
    const fileName = fileNameInput.value;
    const data = await window.myApi.readFile(fileName)
    // console.log(data)
    fileContent.textContent = data || 'No content available.'
});
```



我们输入要访问的文档的名字 `readme.txt`

此时预加载脚本没有做安全检查，将文件名称直接拼接传递给主进程，因此如果我们输入 `../../../../../../etc/passwd` 这种名称，就可能导致任意文件读取漏洞



这种属于是不安全的实现，案例比较极端，但是意思应该表达清楚了，这属于是 `Preload` 和主进程实现上做得不安全，导致问题

2. 过度暴露

在上面的例子中，我们使用了 `Electron 30.0.0` 版本，开启了 `sandbox` ，使用预加载脚本使用 `contextBridge` 将 `API` 暴露给渲染进程，我们将打开文件功能进行了封装，封装成了一个函数，这也就意味着每个新功能，如果需要主进程参与可能都会创建不止一个新的函数

如果开发者直接将 `ipcRenderer` 或 `ipcRenderer.invoke` 这种 `API` 或非必要函数直接暴露给渲染进程，就可能导致渲染进程任意发起 `IPC` 通信、获取敏感信息等

假设程序由很多和操作系统命令执行结果相关的功能，所以主进程有一个接收参数并执行的通信，这样 `Preload` 脚本中直接传递参数，复用这一个监听即可

`main.js`

```
const { app, BrowserWindow, ipcMain } = require('electron');
const path = require('path');

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js'),
    },
  });

  win.loadFile('./index.html');
}

app.whenReady().then(() => {
  ipcMain.handle('exec-command', async (event, cmd) => {
    return new Promise((resolve, reject) => {
      require('child_process').exec(cmd, (error, stdout, stderr) => {
        if (error) {
          console.error(`Error executing command "${cmd}":`, error);
          reject(error);
        } else {
          resolve(stdout.trim());
```

```
        }
      });
    });
  });

  createWindow();
});

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

可以通过与主进程 `IPC` 通信，之后传递要执行的命令的字符串，并通过 `IPC` 传递会渲染进程

此时并不能说主进程写得不安全，如果 `Preload` 脚本固定传递的字符，例如 `cat /etc/issue` 便不会出现任意命令执行

如果此时渲染进程和 `Preload` 如下，则就会产生风险

`index.html`

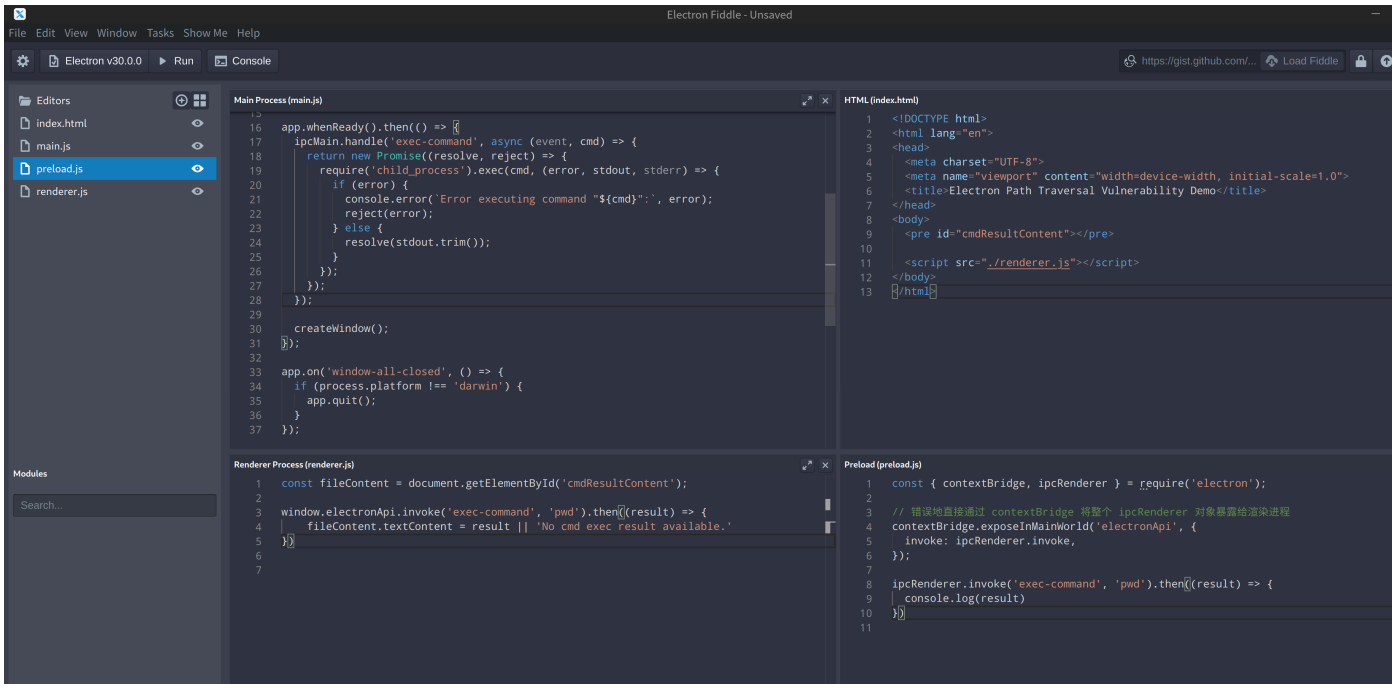
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Electron Path Traversal Vulnerability Demo</title>
</head>
<body>
  <pre id="cmdResultContent"></pre>

  <script src="./renderer.js"></script>
</body>
</html>
```

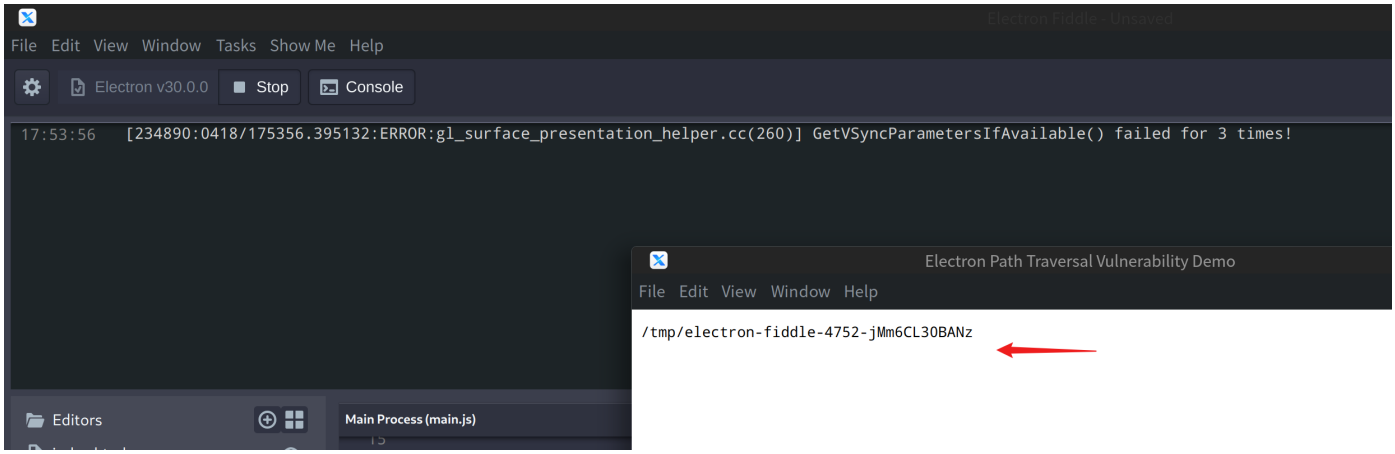
`preload.js`

```
const { contextBridge, ipcRenderer } = require('electron');

// 错误地直接通过 contextBridge 将整个 ipcRenderer 对象暴露给渲染进程
contextBridge.exposeInMainWorld('electronApi', {
  invoke: ipcRenderer.invoke,
});
```

`renderer.js`

```
const fileContent = document.getElementById('cmdResultContent');

window.electronApi.invoke('exec-command', 'pwd').then((result) => {
    fileContent.textContent = result || 'No cmd exec result available.'
})
```





此时就会导致任意命令执行

0x04 总结

预加载脚本的风险主要来源于不安全的编码习惯，但是有些泄漏可能是不容易发现的，例如有几个函数只是给 `Preload` 自己使用的，但是不小心暴露给了渲染进程；函数是给自己写的渲染进程使用的，结果同时暴露给了 `iframe` 这种嵌入内容等

预加载脚本是一个很好的代码审计的切入点，如果安全配置较为完善，则安全漏洞的利用基本都要通过预加载脚本传递数据，也就是掌握了咽喉位置，详细分析每一个 `IPC` 通信，就能找到几乎所有渲染进程攻击主进程的攻击面