Security

Security

:::info Reporting security issues For information on how to properly disclose an Electron vulnerability, see SECURITY.md.

For upstream Chromium vulnerabilities: Electron keeps up to date with alternating Chromium releases. For more information, see the Electron Release Timelines document. :::

Preface

As web developers, we usually enjoy the strong security net of the browser — the risks associated with the code we write are relatively small. Our websites are granted limited powers in a sandbox, and we trust that our users enjoy a browser built by a large team of engineers that is able to quickly respond to newly discovered security threats.

When working with Electron, it is important to understand that Electron is not a web browser. It allows you to build feature-rich desktop applications with familiar web technologies, but your code wields much greater power. JavaScript can access the filesystem, user shell, and more. This allows you to build high quality native applications, but the inherent security risks scale with the additional powers granted to your code.

With that in mind, be aware that displaying arbitrary content from untrusted sources poses a severe security risk that Electron is not intended to handle. In fact, the most popular Electron apps (Atom, Slack, Visual Studio Code, etc) display primarily local content (or trusted, secure remote content without Node integration) — if your application executes code from an online source, it is your responsibility to ensure that the code is not malicious.

General guidelines

Security is everyone's responsibility

It is important to remember that the security of your Electron application is the result of the overall security of the framework foundation (*Chromium*,

Node.js), Electron itself, all NPM dependencies and your code. As such, it is your responsibility to follow a few important best practices:

- **Keep your application up-to-date with the latest Electron framework release.** When releasing your product, you're also shipping a bundle composed of Electron, Chromium shared library and Node.js. Vulnerabilities affecting these components may impact the security of your application. By updating Electron to the latest version, you ensure that critical vulnerabilities (such as *nodeIntegration bypasses*) are already patched and cannot be exploited in your application. For more information, see "Use a current version of Electron".
- **Evaluate your dependencies.** While NPM provides half a million reusable packages, it is your responsibility to choose trusted 3rd-party libraries. If you use outdated libraries affected by known vulnerabilities or rely on poorly maintained code, your application security could be in jeopardy.
- **Adopt secure coding practices.** The first line of defense for your application is your own code. Common web vulnerabilities, such as Cross-Site Scripting (XSS), have a higher security impact on Electron applications hence it is highly recommended to adopt secure software development best practices and perform security testing.

Isolation for untrusted content

A security issue exists whenever you receive code from an untrusted source (e.g. a remote server) and execute it locally. As an example, consider a remote website being displayed inside a default `BrowserWindow`. If an attacker somehow manages to change said content (either by attacking the source directly, or by sitting between your app and the actual destination), they will be able to execute native code on the user's machine.

Under no circumstances should you load and execute remote code with Node.js integration enabled. Instead, use only local files (packaged together with your application) to execute Node.js code. To display remote content, use the `<webview>` tag or `BrowserView`, make sure to disable the `nodeIntegration` and enable `contextIsolation`.

:::info Electron security warnings Security warnings and recommendations are printed to the developer console. They only show up when the binary's name is Electron, indicating that a developer is currently looking at the console.

You can force-enable or force-disable these warnings by setting `ELECTRON_ENABLE_SECURITY_WARNINGS` or `ELECTRON_DISABLE_SECURITY_WARNINGS` on either `process.env` or the `window` object. :::

Checklist: Security recommendations

You should at least follow these steps to improve the security of your application:

1. Only load secure content
2. Disable the Node.js integration in all renderers that display remote content
3. Enable context isolation in all renderers that display remote content
4. Enable process sandboxing
5. Use `ses.setPermissionRequestHandler()` in all sessions that load remote content
6. Do not disable `webSecurity`
7. Define a `Content-Security-Policy` and use restrictive rules (i.e. `script-src 'self'`)
8. Do not enable `allowRunningInsecureContent`
9. Do not enable experimental features
10. Do not use `enableBlinkFeatures`
11. `<webview>`: Do not use `allowpopups`
12. `<webview>`: Verify options and params
13. Disable or limit navigation
14. Disable or limit creation of new windows
15. Do not use `shell.openExternal` with untrusted content
16. Use a current version of Electron

To automate the detection of misconfigurations and insecure patterns, it is possible to use Electronegativity. For additional details on potential weaknesses and implementation bugs when developing applications using Electron, please refer to this guide for developers and auditors.

1. Only load secure content

Any resources not included with your application should be loaded using a secure protocol like `HTTPS`. In other words, do not use insecure protocols like `HTTP`. Similarly, we recommend the use of `WSS` over `WS`, `FTPS` over `FTP`, and so on.

Why? `HTTPS` has three main benefits:

1. It authenticates the remote server, ensuring your app connects to the correct host instead of an impersonator.
2. It ensures data integrity, asserting that the data was not modified while in transit between your application and the host.
3. It encrypts the traffic between your user and the destination host, making it more difficult to eavesdrop on the information sent between your app and the host.

How? "'js title='main.js (Main Process)' // Bad browserWindow.loadURL('http://example.com')

// Good browserWindow.loadURL('https://example.com')

```html title='index.html (Renderer Process)'
<!-- Bad -->
<script crossorigin src="http://example.com/react.js"></script>
<link rel="stylesheet" href="http://example.com/style.css">

<!-- Good -->
<script crossorigin src="https://example.com/react.js"></script>
<link rel="stylesheet" href="https://example.com/style.css">
```

2. Do not enable Node.js integration for remote content

This recommendation is the default behavior in Electron since 5.0.0.

It is paramount that you do not enable Node.js integration in any renderer (`BrowserWindow`, `BrowserView`, or `<webview>`) that loads remote content. The goal is to limit the powers you grant to remote content, thus making it dramatically more difficult for an attacker to harm your users should they gain the ability to execute JavaScript on your website.

After this, you can grant additional permissions for specific hosts. For example, if you are opening a BrowserWindow pointed at `https://example.com/`, you can give that website exactly the abilities it needs, but no more.

Why? A cross-site-scripting (XSS) attack is more dangerous if an attacker can jump out of the renderer process and execute code on the user's computer. Cross-site-scripting attacks are fairly common - and while an issue, their power is usually limited to messing with the website that they are executed on. Disabling Node.js integration helps prevent an XSS from being escalated into a so-called "Remote Code Execution" (RCE) attack.

How? "'js title='main.js (Main Process)' // Bad const mainWindow = new BrowserWindow({ webPreferences: { contextIsolation: false, nodeIntegration: true, nodeIntegrationInWorker: true } })

mainWindow.loadURL('https://example.com')

```js title='main.js (Main Process)'
// Good
const mainWindow = new BrowserWindow({
  webPreferences: {
    preload: path.join(app.getAppPath(), 'preload.js')
  }
})

mainWindow.loadURL('https://example.com')
```

"'html title='index.html (Renderer Process)'

```
When disabling Node.js integration, you can still expose APIs to your website that
do consume Node.js modules or features. Preload scripts continue to have access
to `require` and other Node.js features, allowing developers to expose a custom
API to remotely loaded content via the [contextBridge API](../api/context-bridge.md).

### 3. Enable Context Isolation for remote content

:::info
This recommendation is the default behavior in Electron since 12.0.0.
:::

Context isolation is an Electron feature that allows developers to run code
in preload scripts and in Electron APIs in a dedicated JavaScript context. In
practice, that means that global objects like `Array.prototype.push` or
`JSON.parse` cannot be modified by scripts running in the renderer process.

Electron uses the same technology as Chromium's [Content Scripts](https://developer.chrome.c
to enable this behavior.

Even when `nodeIntegration: false` is used, to truly enforce strong isolation
and prevent the use of Node primitives `contextIsolation` **must** also be used.

:::info
For more information on what `contextIsolation` is and how to enable it please
see our dedicated [Context Isolation](context-isolation.md) document.
:::info

### 4. Enable process sandboxing

[Sandboxing](https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md)
is a Chromium feature that uses the operating system to
significantly limit what renderer processes have access to. You should enable
the sandbox in all renderers. Loading, reading or processing any untrusted
content in an unsandboxed process, including the main process, is not advised.

:::info
For more information on what `contextIsolation` is and how to enable it please
see our dedicated [Process Sandboxing](sandbox.md) document.
:::info

### 5. Handle session permission requests from remote content

You may have seen permission requests while using Chrome: they pop up whenever
the website attempts to use a feature that the user has to manually approve (
like notifications).
```

```
The API is based on the [Chromium permissions API](https://developer.chrome.com/extensions/p
and implements the same types of permissions.

#### Why?

By default, Electron will automatically approve all permission requests unless
the developer has manually configured a custom handler. While a solid default,
security-conscious developers might want to assume the very opposite.

#### How?

```js title='main.js (Main Process)'
const { session } = require('electron')
const URL = require('url').URL

session
  .fromPartition('some-partition')
  .setPermissionRequestHandler((webContents, permission, callback) => {
    const parsedUrl = new URL(webContents.getURL())

    if (permission === 'notifications') {
      // Approves the permissions request
      callback(true)
    }

    // Verify URL
    if (parsedUrl.protocol !== 'https:' || parsedUrl.host !== 'example.com') {
      // Denies the permissions request
      return callback(false)
    }
  })
```

6. Do not disable `webSecurity`

This recommendation is Electron's default.

You may have already guessed that disabling the `webSecurity` property on a renderer process (`BrowserWindow`, `BrowserView`, or `<webview>`) disables crucial security features.

Do not disable `webSecurity` in production applications.

Why? Disabling `webSecurity` will disable the same-origin policy and set `allowRunningInsecureContent` property to `true`. In other words, it allows the execution of insecure code from different domains.
```

How? `js title='main.js (Main Process)' // Bad const mainWindow = new BrowserWindow({ webPreferences: { webSecurity: false } })`

`js title='main.js (Main Process)' // Good const mainWindow = new BrowserWindow()`

“‘html title=‘index.html (Renderer Process)’

`### 7. Define a Content Security Policy`

`A Content Security Policy (CSP) is an additional layer of protection against cross-site-scripting attacks and data injection attacks. We recommend that they be enabled by any website you load inside Electron.`

`#### Why?`

`CSP allows the server serving content to restrict and control the resources Electron can load for that given web page. \`https://example.com\` should be allowed to load scripts from the origins you defined while scripts from \`https://evil.attacker.com\` should not be allowed to run. Defining a CSP is an easy way to improve your application's security.`

`#### How?`

`The following CSP will allow Electron to execute scripts from the current website and from \`apis.example.com\`.`

```plaintext
// Bad
Content-Security-Policy: '*'

// Good
Content-Security-Policy: script-src 'self' https://apis.example.com
```

CSP HTTP headers Electron respects the `Content-Security-Policy` HTTP header which can be set using Electron’s `webRequest.onHeadersReceived` handler:

“‘javascript title=‘main.js (Main Process)’ const { session } = require(‘electron’)

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

`#### CSP meta tag`

`CSP's preferred delivery mechanism is an HTTP header. However, it is not possible`

to use this method when loading a resource using the `file://` protocol. It can be useful in some cases to set a policy on a page directly in the markup using a `<meta>` tag:

```html title='index.html (Renderer Process)'
<meta http-equiv="Content-Security-Policy" content="default-src 'none'">
```

8. Do not enable `allowRunningInsecureContent`

This recommendation is Electron's default.

By default, Electron will not allow websites loaded over `HTTPS` to load and execute scripts, CSS, or plugins from insecure sources (`HTTP`). Setting the property `allowRunningInsecureContent` to `true` disables that protection.

Loading the initial HTML of a website over `HTTPS` and attempting to load subsequent resources via `HTTP` is also known as "mixed content".

Why? Loading content over `HTTPS` assures the authenticity and integrity of the loaded resources while encrypting the traffic itself. See the section on only displaying secure content for more details.

How? `js title='main.js (Main Process)' // Bad const mainWindow = new BrowserWindow({ webPreferences: { allowRunningInsecureContent: true } })`

`js title='main.js (Main Process)' // Good const mainWindow = new BrowserWindow({})`

9. Do not enable experimental features

This recommendation is Electron's default.

Advanced users of Electron can enable experimental Chromium features using the `experimentalFeatures` property.

Why? Experimental features are, as the name suggests, experimental and have not been enabled for all Chromium users. Furthermore, their impact on Electron as a whole has likely not been tested.

Legitimate use cases exist, but unless you know what you are doing, you should not enable this property.

How? `js title='main.js (Main Process)' // Bad const mainWindow = new BrowserWindow({ webPreferences: { experimentalFeatures: true } })`

`js title='main.js (Main Process)' // Good const mainWindow = new BrowserWindow({})`

10. Do not use `enableBlinkFeatures`

This recommendation is Electron's default.

Blink is the name of the rendering engine behind Chromium. As with `experimentalFeatures`, the `enableBlinkFeatures` property allows developers to enable features that have been disabled by default.

Why? Generally speaking, there are likely good reasons if a feature was not enabled by default. Legitimate use cases for enabling specific features exist. As a developer, you should know exactly why you need to enable a feature, what the ramifications are, and how it impacts the security of your application. Under no circumstances should you enable features speculatively.

How? `js title='main.js (Main Process)' // Bad const mainWindow = new BrowserWindow({ webPreferences: { enableBlinkFeatures: 'ExecCommandInJavaScript' } })`

`js title='main.js (Main Process)' // Good const mainWindow = new BrowserWindow()`

11. Do not use `allowpopups` for WebViews

This recommendation is Electron's default.

If you are using `<webview>`, you might need the pages and scripts loaded in your `<webview>` tag to open new windows. The `allowpopups` attribute enables them to create new `BrowserWindows` using the `window.open()` method. `<webview>` tags are otherwise not allowed to create new windows.

Why? If you do not need popups, you are better off not allowing the creation of new `BrowserWindows` by default. This follows the principle of minimally required access: Don't let a website create new popups unless you know it needs that feature.

How? "'html title='index.html (Renderer Process)'

`### 12. Verify WebView options before creation`

`A WebView created in a renderer process that does not have Node.js integration enabled will not be able to enable integration itself. However, a WebView will always create an independent renderer process with its own` `webPreferences`.

`It is a good idea to control the creation of new [`<webview>`][webview-tag] tags from the main process and to verify that their webPreferences do not disable security features.`

`#### Why?`

```
Since `<webview>` live in the DOM, they can be created by a script running on your
website even if Node.js integration is otherwise disabled.

Electron enables developers to disable various security features that control
a renderer process. In most cases, developers do not need to disable any of
those features - and you should therefore not allow different configurations
for newly created [`<webview>`][webview-tag] tags.

#### How?

Before a [`<webview>`][webview-tag] tag is attached, Electron will fire the
`will-attach-webview` event on the hosting `webContents`. Use the event to
prevent the creation of `webViews` with possibly insecure options.

```js title='main.js (Main Process)'
app.on('web-contents-created', (event, contents) => {
  contents.on('will-attach-webview', (event, webPreferences, params) => {
    // Strip away preload scripts if unused or verify their location is legitimate
    delete webPreferences.preload

    // Disable Node.js integration
    webPreferences.nodeIntegration = false

    // Verify URL being loaded
    if (!params.src.startsWith('https://example.com/')) {
      event.preventDefault()
    }
  })
})
```

Again, this list merely minimizes the risk, but does not remove it. If your goal is to display a website, a browser will be a more secure option.

13. Disable or limit navigation

If your app has no need to navigate or only needs to navigate to known pages, it is a good idea to limit navigation outright to that known scope, disallowing any other kinds of navigation.

Why? Navigation is a common attack vector. If an attacker can convince your app to navigate away from its current page, they can possibly force your app to open web sites on the Internet. Even if your `webContents` are configured to be more secure (like having `nodeIntegration` disabled or `contextIsolation` enabled), getting your app to open a random web site will make the work of exploiting your app a lot easier.
```

A common attack pattern is that the attacker convinces your app's users to interact with the app in such a way that it navigates to one of the attacker's pages. This is usually done via links, plugins, or other user-generated content.

How? If your app has no need for navigation, you can call `event.preventDefault()` in a `will-navigate` handler. If you know which pages your app might navigate to, check the URL in the event handler and only let navigation occur if it matches the URLs you're expecting.

We recommend that you use Node's parser for URLs. Simple string comparisons can sometimes be fooled - a `startsWith('https://example.com')` test would let `https://example.com.attacker.com` through.

"'js title='main.js (Main Process)' const URL = require('url').URL

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()
}
```

}) })

```
### 14. Disable or limit creation of new windows

If you have a known set of windows, it's a good idea to limit the creation of
additional windows in your app.

#### Why?

Much like navigation, the creation of new `webContents` is a common attack
vector. Attackers attempt to convince your app to create new windows, frames,
or other renderer processes with more privileges than they had before; or
with pages opened that they couldn't open before.

If you have no need to create windows in addition to the ones you know you'll
need to create, disabling the creation buys you a little bit of extra
security at no cost. This is commonly the case for apps that open one
`BrowserWindow` and do not need to open an arbitrary number of additional
windows at runtime.

#### How?

[`webContents`][web-contents] will delegate to its [window open
handler][window-open-handler] before creating new windows. The handler will
receive, amongst other parameters, the `url` the window was requested to open
and the options used to create it. We recommend that you register a handler to
```

monitor the creation of windows, and deny any unexpected window creation.

```js title='main.js (Main Process)'
const { shell } = require('electron')

app.on('web-contents-created', (event, contents) => {
  contents.setWindowOpenHandler(({ url }) => {
    // In this example, we'll ask the operating system
    // to open this event's url in the default browser.
    //
    // See the following item for considerations regarding what
    // URLs should be allowed through to shell.openExternal.
    if (isSafeForExternalOpen(url)) {
      setImmediate(() => {
        shell.openExternal(url)
      })
    }

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

15. Do not use `shell.openExternal` with untrusted content

The shell module's `openExternal` API allows opening a given protocol URI with the desktop's native utilities. On macOS, for instance, this function is similar to the `open` terminal command utility and will open the specific application based on the URI and filetype association.

Why? Improper use of `openExternal` can be leveraged to compromise the user's host. When openExternal is used with untrusted content, it can be leveraged to execute arbitrary commands.

How? `js title='main.js (Main Process)' // Bad const { shell } = require('electron') shell.openExternal(USER_CONTROLLED_DATA_HERE)`

`js title='main.js (Main Process)' // Good const { shell } = require('electron') shell.openExternal('https://example.com/index.html')`

16. Use a current version of Electron

You should strive for always using the latest available version of Electron. Whenever a new major version is released, you should attempt to update your app as quickly as possible.

Why? An application built with an older version of Electron, Chromium, and Node.js is an easier target than an application that is using more recent versions of those components. Generally speaking, security issues and exploits for older versions of Chromium and Node.js are more widely available.

Both Chromium and Node.js are impressive feats of engineering built by thousands of talented developers. Given their popularity, their security is carefully tested and analyzed by equally skilled security researchers. Many of those researchers disclose vulnerabilities responsibly, which generally means that researchers will give Chromium and Node.js some time to fix issues before publishing them. Your application will be more secure if it is running a recent version of Electron (and thus, Chromium and Node.js) for which potential security issues are not as widely known.

How? Migrate your app one major version at a time, while referring to Electron's Breaking Changes document to see if any code needs to be updated.

17. Validate the `sender` of all IPC messages

You should always validate incoming IPC messages `sender` property to ensure you aren't performing actions or sending information to untrusted renderers.

Why? All Web Frames can in theory send IPC messages to the main process, including iframes and child windows in some scenarios. If you have an IPC message that returns user data to the sender via `event.reply` or performs privileged actions that the renderer can't natively, you should ensure you aren't listening to third party web frames.

You should be validating the `sender` of **all** IPC messages by default.

How? "'js title='main.js (Main Process)' // Bad ipcMain.handle('get-secrets', () => { return getSecrets(); });

// Good ipcMain.handle('get-secrets', (e) => { if (!validateSender(e.senderFrame)) return null; return getSecrets(); });

function validateSender(frame) { // Value the host of the URL using an actual URL parser and an allowlist if ((new URL(frame.url)).host === 'electronjs.org') return true; return false; } "'