Electron Security Checklist

A guide for developers and auditors

Created by Luca Carettoni - @lucacarettoni

WWW.DOYENSEC.COM

@DOYENSEC

Table of Contents

Revision History	1
Contacts	1
Abstract	2
About Doyensec	2
Introduction	3
Electron Security Checklist	5
Disable nodeIntegration for untrusted origins	5
Use sandbox for untrusted origins	7
Review the use of command line arguments	8
Review the use of preload scripts	9
Do not use disablewebsecurity	11
Do not allow insecure HTTP connections	12
Do not use Chromium's experimental features	14
Limit navigation flows to untrusted origins	15
Use setPermissionRequestHandler for untrusted origins	16
Do not use insertCSS, executeJavaScript or eval with user-supplied content	17
Do not allow popups in webview	18
Review the use of custom protocol handlers	19
Review the use of openExternal	20
Bibliography	21

Revision History

Version	Date	Description	Author
0.1	June, 26 2017	First internal draft	Luca Carettoni
0.2	July, 14 2017	First release to peers	Luca Carettoni
0.3	July, 16 2017	BlackHat white paper release	Luca Carettoni
0.4	July, 31 2017	Minor changes. Typo fixes	Luca Carettoni

Contacts

Company	Name	Email
Doyensec, LLC.	Luca Carettoni	luca@doyensec.com

Abstract

Despite all predictions, native Desktop applications are back. After years porting stand-alone apps to the web, we are witnessing an inverse trend. Many companies have started providing native desktop software built using the same technologies as their web counterparts. In this trend, Github's Electron has become a popular framework to build cross-platform desktop apps with JavaScript, HTML, and CSS. While it seems to be easy, embedding a web application in a self-contained web environment (Chromium, Node.Js) leads to new security challenges.

This document introduces a checklist of security anti-patterns and must-have features to illustrate misconfigurations and vulnerabilities in Electron-based applications. Software developers and security auditors can benefit from this document as it provides a concise, yet comprehensive, summary of potential weaknesses and implementation bugs when developing applications using Electron.

As part of our study of Electron security, we have implemented a tool (**Electronegativity** - available on Doyensec's Github https://github.com/doyensec/electronegativity) that checks for the security anti-patterns discussed in this document.

About Doyensec

Doyensec is an independent security research and development company focused on vulnerability discovery and remediation. We work at the intersection of software development and offensive engineering to help companies craft secure code.

Research is one of our founding principles and we invest heavily in it. By discovering new vulnerabilities and attack techniques, we constantly improve our capabilities and contribute to secure the applications we all use.

“Several experts have told me in all seriousness that browser security models are now so complex that I should not even write a section about this”

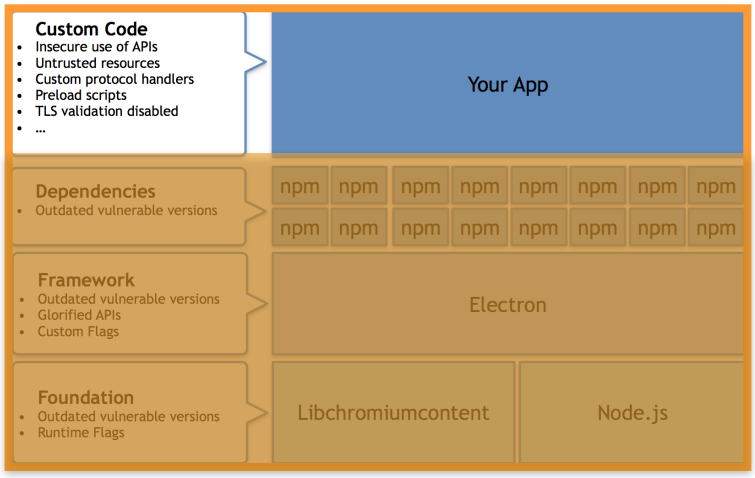
Threat Modeling - Adam Shostack

Introduction

Web security is complicated. Modern browsers are enforcing numerous security mechanisms to ensure isolation between sites, facilitate web security protections and preventing untrusted remote content to compromise the security of the host. When working with Electron, things get even more complicated. While Electron is based on Chromium’s Content module, it is not a browser. Since it facilitates the construction of complex desktop applications, Electron gives the developer a lot of power. In fact, thanks to the integration with Node.js, JavaScript can access operating system primitives to take full advantage of native desktop mechanisms. As we know, with great power comes great responsibility.

We assume that the reader is already familiar with Electron and its inner-working mechanisms, as we will be discussing security-relevant topics without providing any introduction to the framework. If you are not familiar with Electron’s core philosophy and APIs, please review the Electron online documentation. There is a growing community of Electron developers that have produced many excellent tutorials for beginners.

During our research, we have extensively studied the security of the Electron framework itself and reported vulnerabilities to the core team. However, in this document, we will be focusing on application-level design and implementation flaws only.



As a software developer, it is important to remember that the security of your application is the result of the overall security of the framework foundation (*Libchromiumcontent*, *Node.js*), Electron itself, all dependencies (NPM packages) and your code. As such, it is your responsibility to follow a few important best practices:

- **Keep your application in sync 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 to abuse your application

- **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. Remember, OpenSource does not necessarily mean security by default

- **Know your framework (and its limitations)**
    - Certain principles and security mechanisms implemented by modern browsers are not enforced in Electron. For instance, even the latest Electron release at the time of writing does not fully enforce the *Same Origin Policy (SOP)* and still does not restrict the file:// handler from web origins, thus a remote untrusted page can read the content of local resources without user interaction. Adopt defense in depth mechanisms to mitigate those deficiencies. For more details, please refer to our *Black Hat 2017* "Electronegativity, A study of Electron Security" presentation

- **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 recommend to adopt secure software development best practices and perform security testing.

Electron Security Checklist

Disable nodeIntegration for untrusted origins

By default, Electron renderers can use Node.js primitives. For instance, a remote untrusted domain rendered in a browser window could invoke Node.js APIs to execute native code on the user's machine. Similarly, a Cross-Site Scripting (XSS) vulnerability on a website can lead to remote code execution. To display remote content, nodeIntegration should be disabled in the webPreferences of BrowserWindow and webview tag.

Risk	If enabled, nodeIntegration allows JavaScript to leverage Node.js primitives and modules. This could lead to full remote system compromise if you are rendering untrusted content.

Auditing	nodeIntegration and nodeIntegrationInWorker are boolean options that can be used to determine whether node integration is enabled.
	
	For BrowserWindow, default is true. If the option is not present, or is set to *true/1*, nodeIntegration is enabled as in the following examples:
	
	`mainWindow = new BrowserWindow({`
	`  "webPreferences": {`
	`  "nodeIntegration": true,`
	`  "nodeIntegrationInWorker": 1`
	`  }`
	`});`
	
	Or simply:
	
	`mainWindow = new BrowserWindow()`
	
	For webview tag, default is false. When this attribute is present, the guest page in webview will have node integration:
	
	`<webview src="https://doyensec.com/" nodeintegration></webview>`
	
	When sanbox is enabled (see below), nodeintegration is disabled.

Auditing	Please note that it is also possible to use the will-attach-webview event to verify (and potentially change) any attribute of webPreferences. This event is emitted when a webview is being attached to the web content.
	
	Since this mechanism can be used to change the webPreferences configuration, please carefully review the implementation of the callback. At the same time, this is a powerful mechanism to validate all settings and ensure a secure instance of webview, as demonstrated in this implementation:
	
	app.on('web-contents-created', (event, contents) => {
	  contents.on('will-attach-webview', (event, webPreferences, params) => {
	  // Strip away preload scripts if unused
	  // Alternatively, verify their location if legitimate
	  delete webPreferences.preload
	  delete webPreferences.preloadURL
	
	  // Disable node integration
	  webPreferences.nodeIntegration = false
	
	  // Verify URL being loaded
	  if (!params.src.startsWith('https://doyensec.com/')) {
	  event.preventDefault()
	  }
	  })
	})

Use sandbox for untrusted origins

While nodeIntegration tackles the problem of limiting access to Node.js primitives from a remote untrusted origin, it does not mitigate security flaws introduced by Electron's "glorified" APIs. In fact, Electron extends the default JavaScript APIs (E.g. window.open returns an instance of BrowserWindowProxy) which leads to a larger attack surface (as demonstrated by our recent nodeIntegration bypass bug, fixed in v1.6.8).

Instead, sandboxed renderers are supposed to expose default JavaScript APIs. We use the "supposed to" form as the current implementation (at the time of writing) is experimental and does not revert the behavior of all "glorified" APIs. If set, this option will sandbox the renderer associated with the window, making it compatible with the Chromium OS-level sandbox.

Additionally, a sandboxed renderer does not have a Node.js environment running (with the exception of preload scripts) and the renderers can only make changes to the system by delegating tasks to the main process via IPC.

While still not perfect, this option should be enabled whenever there is a need of loading untrusted content in a browser window.

Risk	Even with nodeIntegration disabled, the current implementation of Electron does not completely mitigate all risks introduced by loading untrusted resources. As such, it is recommended to enable sandboxing.

Auditing	For BrowserWindow, sandboxing needs to be explicitly enabled:
	
	`mainWindow = new BrowserWindow({`
	`  "webPreferences": {`
	`  "sandbox": true`
	`  }`
	`});`
	
	To enable sandboxing for all BrowserWindow instances, a command line argument is necessary:
	
	`$ electron --enable-sandbox app.js`
	
	Please note that programmatically adding the command line switch "enable-sandbox" is not sufficient, as the code responsible for appending arguments runs after it is possible to make changes to Chromium's sandbox settings. Electron needs to be executed from the beginning with the enable-sandbox argument.
	
	At the time of writing, sandboxing for the webview tag is still not supported.

Review the use of command line arguments

With Electron, it is possible to programmatically insert command line arguments to modify the behavior of the framework foundation (*LibChromiumcontent* and *Node.js*) and Electron itself. For instance, setting the variable —proxy-server will force Chromium to use a specific proxy server, despite system settings. To debug JavaScript executed in the main process, Electron allows to attach an external debugger. This feature can be enabled using the --debug or --debug-brk command line switch. Additionally, the application can implement custom command line arguments.

Risk	The use of additional command line arguments can increase the application attack surface, disable security features or influence the overall security posture. For example, if Electron's debugging is enabled, Electron will listen for V8 debugger protocol messages on the specified port. An attacker could leverage the external debugger to subvert the application at runtime.

Auditing	Review all occurrences of appendArgument and appendSwitch:
	
	`const {app} = require('electron')`
	`app.commandLine.appendArgument('debug')`
	`app.commandLine.appendSwitch('proxy-server', '8080')`
	
	Search for custom arguments (e.g. —debug or —debug-brk) in package.json, and within the application codebase.

Review the use of preload scripts

Despite disabling nodeIntegration and enabling sandbox, preload scripts have access to Node.js APIs. When node integration is turned off, the preload script can reintroduce Node global symbols back to the global scope. Also, the current implementation of the Chromium sandbox still allows access to all underlying Electron/Node.js primitives using either the remote module or internal IPC:

#1 - Sandbox bypass in preload scripts using *remote*

```
app = require('electron').remote.app
```

#2 - Sandbox bypass in preload scripts using internal Electron IPC messages

```
{ipcRenderer} = require('electron')
app = ipcRenderer.sendSync('ELECTRON_BROWSER_GET_BUILTIN', 'app')
```

As demonstrated in the examples above, a malicious preload script can still obtain a reference to the application object by leveraging the remote module, which provides a simple way to do inter-process communication (IPC) between the renderer process and the main process. Alternatively, it is also possible to emulate the internal IPC mechanism sending a message to the main process synchronously via ELECTRON_ internal channels. Considering the privileged access available in preload, the code of preload scripts need to be carefully reviewed.

Additionally, it is highly recommend to leverage the experimental context isolation feature. contextIsolation introduces JavaScript context isolation for preload scripts, as implemented in Chrome Content Scripts. This option should be used when loading potentially untrusted resources to ensure that the loaded content cannot tamper with the preload script and any Electron APIs being used. The preload script will still have access to global variables, but it will use its own set of JavaScript builtins (Array, Object, JSON, etc.) and will be isolated from any changes made to the global environment by the loaded page.

Risk	Improper use of preload scripts can introduce nodeIntegration or sandbox bypasses, in addition to other vulnerabilities. If context isolation is not used, there is a risk that malicious code in preload scripts could tamper JavaScript native functions that the preload script and Electron APIs make use of.

Auditing	Search for preload within the webPreferences of BrowserWindow. Manually review the imported scripts. If preload is used, make sure that webPreferences also includes contextIsolation: true or contextIsolation: 1

Do not use disablewebsecurity

This flag gives access to the underline disablewebsecurity Chromium option. When this attribute is present, the guest page will have web security disabled. For instance, Same-Origin Policy will not be enforced.

Please note that the Same-Origin Policy is actually not strictly enforced by the current implementation of Electron, due to a design flaw. As a result, this option is practically irrelevant at the moment. Apart from this, disabling web security will impact the application on future Electron releases, in which SOP will be supposedly enforced.

At the time of writing this document, even with web security enabled, remote pages can still inject JavaScript in a different domain using one of the following tricks:

#1 - SOP bypass using window.location

```
<script>
win = window.open("https://doyensec.com");
win.location = "javascript:alert(document.domain)";
</script>
```

#2 - SOP bypass using BrowserWindowProxy eval

```
<script>
win = window.open("https://doyensec.com");
win.eval("alert(document.domain)");
</script>
```

Risk	When enabled, SOP is not enforced and mixed content is allowed (e.g. an https page using JavaScript, CSS from http origins).

Auditing	In the webPreferences of BrowserWindow, look for webSecurity:false or webSecurity:0
	
	`mainWindow = new BrowserWindow({`
	`  "webPreferences": {`
	`  "webSecurity": false`
	`  }`
	`});`
	
	In the webview tag, look for disablewebsecurity:
	
	`<webview src="https://doyensec.com/" disablewebsecurity></webview>`
	
	Additionally, search for the runtime flag —disable-web-security in package.json, and within the application codebase.

Do not allow insecure HTTP connections

When using HTTP as the transport, security is provided by Transport Layer Security (TLS). TLS, and its predecessor SSL, are widely used on the Internet to authenticate a service to a client, and then to provide confidentiality to the channel.

Transport security is a critical mechanism for every Electron application. Three problematics are particularly important:

- **HTTP**. Directly fetching content using plain-text HTTP opens your application to Man-in-the-Middle attacks
- **Mixed content**. Mixed content occurs when the initial HTML page is loaded over a secure HTTPS connection, but other resources (such as images, videos, stylesheets, scripts) are loaded over an insecure HTTP connection
- **Insecure TLS Validation.** Security issues and voluntary opt-out of TLS certificates validation may allow an attacker to bypass Transport Layer Security

Risk	HTTP, Mixed Content and TLS validation opt-out should not be used, as it makes possible to sniff and tamper the user's traffic. If nodeIntegration is also enabled, an attacker can inject malicious JavaScript and compromise the user's host.

Auditing	• **HTTP**
	
	Search for resources loaded using http like:
	
	`win = new BrowserWindow(...);`
	`win.loadURL('http://example.com/');`
	
	• **Mixed content**
	
	Search for allowRunningInsecureContent set to true/1 within the webPreferences of BrowserWindow or in the webview tag:
	
	`mainWindow = new BrowserWindow({`
	`  "webPreferences": {`
	`  "allowRunningInsecureContent": true`
	`  }`
	`});`
	
	`<webview src="https://doyensec.com" webpreferences ="allowRunningInsecureContent"></webview>`

Auditing	• **Insecure TLS Validation**
	
	Verify that the application does not explicitly opt-out from TLS validation. Look for occurrences of certificate-error and setCertificateVerifyProc:
	
	app.on('certificate-error', (event, webContents, url, error, certificate, callback) => {
	if (url === 'https://doyensec.com') {
	callback(true) //Go ahead anyway
	} else {
	callback(false)
	}})
	
	win.webContents.session.setCertificateVerifyProc((request, callback) => {
	const {hostname} = request
	if (hostname === 'doyensec.com') {
	callback(0) //success and disables certificate verification
	} else {
	callback(-3) //use the verification result from chromium
	}})
	
	Additionally, verify custom TLS certificates imported into the platform certificate store with app.importCertificate(options, callback).

Do not use Chromium's experimental features

The experimentalFeatures, experimentalCanvasFeatures, blinkFeatures flags can be used to enable Chromium's features, which increase the overall attack surface for production applications. blinkFeatures allows to selectively specify a feature of Blink (Chromium web browser engine) to be enabled during runtime; a complete list of flags is available in the Chromium source code repository.

Risk	Experimental features may introduce bugs and increase the application attack surface.

Auditing	Search for experimentalFeatures, experimentalCanvasFeatures flags set to true/1 within the webPreferences of BrowserWindow or in the webview tag:
	
	`mainWindow = new BrowserWindow({`
	`  "webPreferences": {`
	`  "experimentalCanvasFeatures": true`
	`  }`
	`});`
	
	Also, look for blinkFeatures selections:
	
	`<webview src="https://doyensec.com/" blinkfeatures="PreciseMemoryInfo, CSSVariables"></webview>`

Limit navigation flows to untrusted origins

The creation of a new browser window or the navigation to untrusted origins may lead to severe vulnerabilities. Additionally, middle-click causes Electron to open a link within a new window. Under certain circumstances, this can be leveraged to execute arbitrary JavaScript in the context of a new window.

Risk	Navigation to untrusted origins can facilitate attacks, thus it is recommend to limit the ability of a BrowserWindow and webview guest page to initiate new navigation flows. Middle-click events can be leverage to subvert the flow of the application.

Auditing	Creation of a new window or the navigation to a specific origin can be inspected and validated using callbacks for the new-window and will-navigate events. Your application can limit the navigation flows by implementing something like:
	
	`win.webContents.on('will-navigate', (event, newURL) => {`
	`  if (win.webContents.getURL() !== 'https://doyensec.com' ) {`
	`  event.preventDefault();`
	`  }`
	`})`
	
	In the default configuration of Electron (at the time of writing), middle-click support needs to be explicitly disabled by the application using:
	
	`mainWindow = new BrowserWindow({`
	`  "webPreferences": {`
	`  "disableBlinkFeatures": "Auxclick"`
	`  }`
	`});`

Use setPermissionRequestHandler for untrusted origins

When loading remote untrusted content, it is recommended to enable Session's permissions handler, which can be used to respond to permission requests.

It is possible to access the session of existing pages by using the session property of WebContents, or from the session module.

```
win = new BrowserWindow()
win.loadURL('https://doyensec.com')

ses = win.webContents.session
console.log(ses.getUserAgent())
```

Using setPermissionRequestHandler, it is possible to write custom code to limit specific permissions (e.g. *openExternal*) in response to events from particular origins.

```
ses.setPermissionRequestHandler((webContents, permission, callback) => {
  if (webContents.getURL() !== 'https://doyensec.com' && permission === 'openExternal') {
    return callback(false)
  } else {
    return callback(true)
 }})
```

Please note that Electron's Session object is a powerful mechanism with access to many properties of the browser sessions, cookies, cache, proxy settings, etc. Use with caution!

Risk	This setting can be used to limit the exploitability in applications that load remote content. Not enforcing custom checks for permission requests (*media*, *geolocation*, *notifications*, *midiSysex*, *pointerLock*, *fullscreen*, *openExternal*) leaves the browser session under full control of the remote origin.
Auditing	Look for occurrences of setPermissionRequestHandler. If used, manually evaluate the implementation and security of the custom callback. If not used, the application does not limit session permissions at all.

Do not use insertCSS, executeJavaScript or eval with user-supplied content

insertCSS, executeJavaScript functions allow to inject respectively CSS and JavaScript from the main process to the renderer process. Instead, eval allows JavaScript execution in the context of a BrowserWindowProxy. If the arguments are user-supplied, they can be leveraged to execute arbitrary content and modify the application behavior.

Risk	In a vulnerable application, a remote page could leverage these functions to subvert the flow of the application by injecting malicious CSS or JavaScript.
Auditing	Search for occurrences of insertCSS, executeJavaScript and eval in both BrowserWindow, webview tag and all other JavaScript resources.

Do not allow popups in webview

When the allowpopups attribute is present, the guest page will be allowed to open new windows. Popups are disabled by default.

Risk	Disabling popups reduces the risk of UI-redressing attacks and limits the exploitability of window abuses. Additionally, popups are often used for intrusive advertising and persistency in JavaScript-based attacks.
Auditing	Search for occurrences of allowpopups in webview tags: <webview src="https://doyensec.com/" allowpopups></webview>

Review the use of custom protocol handlers

Electron allows to define custom protocol handlers so that the application can use mobile-like deep linking to exercise specific features. An example is the fb://profile/33138223345 URI to open a specific Facebook profile. Since custom protocol handlers can be triggered by arbitrary origins, it is important to evaluate how they are implemented and whether user-supplied parameters can lead to security vulnerabilities (e.g. injection flaws).

Risk	The use of custom protocol handlers opens the application to vulnerabilities triggered by users clicking on custom links or arbitrary origins forcing the navigation to crafted links.
Auditing	To register a custom protocol handler, it is necessary to use one of the following functions: • setAsDefaultProtocolClient • registerStandardSchemes • registerServiceWorkerSchemes • registerFileProtocol • registerHttpProtocol • registerStringProtocol • registerBufferProtocol Search for those occurrences and manually review the implementation.

Review the use of openExternal

Shell's openExternal() allows opening a given external protocol URI with the desktop's native utilities. For instance, on macOS, this function is similar to the 'open' terminal command utility and will open the specific application based on the URI and filetype association. When openExternal is used with untrusted content, it can be leveraged to execute arbitrary commands, as demonstrated by the following example:

```
const {shell} = require('electron')
shell.openExternal('file:///Applications/Calculator.app')
```

Risk	Improper use of openExternal can be leveraged to compromise the user's host. Electron's Shell provides powerful primitives that must be used with caution.
Auditing	Manually review all occurrences of openExternal to ensure that no user-supplied content can be injected without validation.

Bibliography

- Electron Documentation - https://electron.atom.io/docs/
- Electron Source Code - https://github.com/electron/
- Electron Issues - https://github.com/electron/electron/issues
- As it stands, Electron security - http://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html
- An update on Electron Security - http://blog.scottlogic.com/2016/06/01/An-update-on-Electron-Security.html
- Hacking Mattermost #2: Year of Node.js on the Desktop - http://haxx.ml/post/145508617751/hacking-mattermost-2-year-of-nodejs-on-the?is_related_post=1
- Chrome Content Scripts - https://developer.chrome.com/extensions/content_scripts#execution-environment
- Chromium Sandbox - https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md
- Supported Chrome Command Line Switches - https://github.com/electron/electron/blob/master/docs/api/chrome-command-line-switches.md