Process Sandboxing

One key security feature in Chromium is that processes can be executed within a sandbox. The sandbox limits the harm that malicious code can cause by limiting access to most system resources — sandboxed processes can only freely use CPU cycles and memory. In order to perform operations requiring additional privilege, sandboxed processes use dedicated communication channels to delegate tasks to more privileged processes.

In Chromium, sandboxing is applied to most processes other than the main process. This includes renderer processes, as well as utility processes such as the audio service, the GPU service and the network service.

See Chromium's Sandbox design document for more information.

Electron's sandboxing policies

Electron comes with a mixed sandbox environment, meaning sandboxed processes can run alongside privileged ones. By default, renderer processes are not sandboxed, but utility processes are. Note that as in Chromium, the main (browser) process is privileged and cannot be sandboxed.

Historically, this mixed sandbox approach was established because having Node.js available in the renderer is an extremely powerful tool for app developers. Unfortunately, this feature is also an equally massive security vulnerability.

Theoretically, unsandboxed renderers are not a problem for desktop applications that only display trusted code, but they make Electron less secure than Chromium for displaying untrusted web content. However, even purportedly trusted code may be dangerous — there are countless attack vectors that malicious actors can use, from cross-site scripting to content injection to man-in-the-middle attacks on remotely loaded websites, just to name a few. For this reason, we recommend enabling renderer sandboxing for the vast majority of cases under an abundance of caution.

Note that there is an active discussion in the issue tracker to enable renderer sandboxing by default. See #28466) for details.

Sandbox behaviour in Electron

Sandboxed processes in Electron behave *mostly* in the same way as Chromium's do, but Electron has a few additional concepts to consider because it interfaces with Node.js.

Renderer processes

When renderer processes in Electron are sandboxed, they behave in the same way as a regular Chrome renderer would. A sandboxed renderer won't have a Node.js environment initialized.

Therefore, when the sandbox is enabled, renderer processes can only perform privileged tasks (such as interacting with the filesystem, making changes to the system, or spawning subprocesses) by delegating these tasks to the main process via inter-process communication (IPC).

Preload scripts

In order to allow renderer processes to communicate with the main process, preload scripts attached to sandboxed renderers will still have a polyfilled subset of Node.js APIs available. A `require` function similar to Node's `require` module is exposed, but can only import a subset of Electron and Node's built-in modules:

- `electron` (only renderer process modules)
- `events`

- `timers`
- `url`

In addition, the preload script also polyfills certain Node.js primitives as globals:

- `Buffer`
- `process`
- `clearImmediate`
- `setImmediate`

Because the `require` function is a polyfill with limited functionality, you will not be able to use CommonJS modules to separate your preload script into multiple files. If you need to split your preload code, use a bundler such as webpack or Parcel.

Note that because the environment presented to the `preload` script is substantially more privileged than that of a sandboxed renderer, it is still possible to leak privileged APIs to untrusted code running in the renderer process unless `contextIsolation` is enabled.

Configuring the sandbox

Enabling the sandbox for a single process

In Electron, renderer sandboxing can be enabled on a per-process basis with the `sandbox: true` preference in the `BrowserWindow` constructor.

```
// main.js
app.whenReady().then(() => {
  const win = new BrowserWindow({
    webPreferences: {
      sandbox: true
    }
  })
  win.loadURL('https://google.com')
})
```

Enabling the sandbox globally

If you want to force sandboxing for all renderers, you can also use the `app.enableSandbox` API. Note that this API has to be called before the app's `ready` event.

```
// main.js
app.enableSandbox()
app.whenReady().then(() => {
  // no need to pass `sandbox: true` since `app.enableSandbox()` was called.
  const win = new BrowserWindow()
  win.loadURL('https://google.com')
})
```

Disabling Chromium's sandbox (testing only)

You can also disable Chromium's sandbox entirely with the `--no-sandbox` CLI flag, which will disable the sandbox for all processes (including utility processes). We highly recommend that you only use this flag for testing purposes, and **never** in production.

Note that the `sandbox: true` option will still disable the renderer's Node.js environment.

A note on rendering untrusted content

Rendering untrusted content in Electron is still somewhat uncharted territory, though some apps are finding success (e.g. Beaker Browser). Our goal is to get as close to Chrome as we can in terms of the security of sandboxed content, but ultimately we will always be behind due to a few fundamental issues:

1. We do not have the dedicated resources or expertise that Chromium has to apply to the security of its product. We do our best to make use of what we have, to inherit everything we can from Chromium, and to respond quickly to security issues, but Electron cannot be as secure as Chromium without the resources that Chromium is able to dedicate.
2. Some security features in Chrome (such as Safe Browsing and Certificate Transparency) require a centralized authority and dedicated servers, both of which run counter to the goals of the Electron project. As such, we disable those features in Electron, at the cost of the associated security they would otherwise bring.
3. There is only one Chromium, whereas there are many thousands of apps built on Electron, all of which behave slightly differently. Accounting for those differences can yield a huge possibility space, and make it challenging to ensure the security of the platform in unusual use cases.
4. We can't push security updates to users directly, so we rely on app vendors to upgrade the version of Electron underlying their app in order for security updates to reach users.

While we make our best effort to backport Chromium security fixes to older versions of Electron, we do not make a guarantee that every fix will be backported. Your best chance at staying secure is to be on the latest stable version of Electron.