DOYENSEC | Application Security
Offensive Engineering

Electronegativity

User Manual - June 2019

Welcome to Electronegativity

Welcome to Electronegativity Official Documentation

Electronegativity is a tool to identify misconfigurations and security anti-patterns in Electron-based applications. It leverages AST and DOM parsing to look for security-relevant configurations, as described in ourElectron Security Checklist - A Guide for Developers and Auditors white-paper.

The Electron security white-paper 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. Based on this study, we developed Electronegativity to automate the discovery of those security anti-patterns.

Besides Application Vulnerabilities...

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. In addition to all application code vulnerabilities *likely* discovered by this tool, 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

> 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 If you use outdated libraries affected by known vulnerabilities or rely on poorly maintained code, your application security could be in jeopardy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

Getting Started

While installing and using Electronegativity is fairly easy, the following section covers some of the details around the tool command line arguments and internal heuristics.

To get Electronegativity up and running, it's possible to use NPM since major releases are pushed to the repository:

```
$ npm install @doyensec/electronegativity -g
```

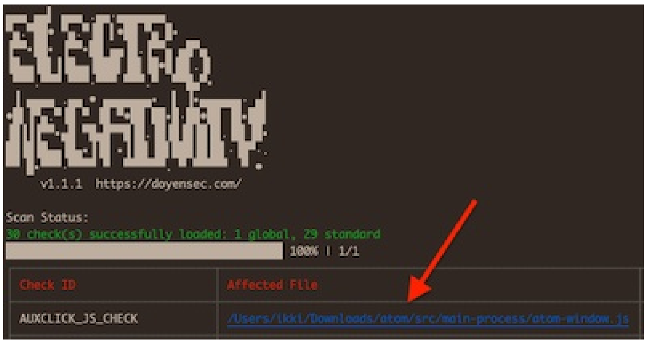
At this stage, the tool can be invoked by using the following command line:

```
$ electronegativity -i [TARGET]
```

Where [TARGET] is either the application source code directory, a single JavaScript file (*js*), a single HTML file (*.html*), or an application ASAR package (*.asar*). In case of single files, the tool will only invoke checks that are relevant for the target. Please note that it is also possible to use wildcard * characters.

Since the tool can be used on both source code and application packages, it is easy to perform testing on both open-source and commercial applications.

On macOS (and potentially other platforms), it is possible to leverage the Command (or Cmd) ⌘ key to highlight file paths within terminal output and files in order to quickly open resources that require manual validation.



```
    v1.1.1  https://doyensec.com/

Scan Status:
30 check(s) successfully loaded: 1 global, 29 standard
                                      100% | 1/1
 Check ID            Affected File
 AUXCLICK_JS_CHECK   /Users/ikki/Downloads/atom/src/main-process/atom-window.js
```

In addition to the mandatory `-i [TARGET]` command line argument, the user can also specify the following options:

Output selection

> -o, --output

This is used for selecting a different output type. By default, the application uses standard terminal output to display results in a compact table. The user can select Comma-Separate file (CSV) or Sarif (SARIF) to obtain a more detailed description of the findings. SARIF stands for Static Analysis Results Interchange Format and it defines a standard format for the output of static analysis tools.

```
$ electronegativity -i [TARGET] -o output.csv
```

will return the output in standard output as well as generating *"output.csv"*. The following fields will be included in the comma-separated file: *issue*, *filename*, *location*, *sample*, *description*, *url*

```
$ electronegativity -i [TARGET] -o output.sarif
```

will return the output in standard output as well as generating *"output.sarif"*. Sarif is a json-based format following the http://json.schemastore.org/sarif-2.0.0 schema.

By default the file locations are provided in their absolute path notation. If you wish to display only their project-relative path, you can do so by providing the `-r` flag (`--relative`).

Checks selection

> -l, --checks

By default, the tool will run all checks on the target file(s). However, the user may be interested in specific classes of vulnerabilities only, thus it is possible to select a list of checks (comma-separated). Checks are defined by their names (e.g. CSPGlobalCheck)

```
$ electronegativity -i [TARGET] -l CSPGlobalCheck
```

will return the output for the *CSPGlobalCheck* only.

Check Typologies

Electronegativity is build in such a way to easily allow the development of new security checks.

There are three different check types that work respectively on three types of application resources:

JS
HTML
JSON

Checks can be either "atomic" or "global":

Atomic: Defines a simple check type, where the check is execute on each branch of the AST/DOM and a single pass of the execution is sufficient to identify vulnerabilities. For example, checks in this category can verify whether a specific flag has value 'true' or 'false'
Global: This class of checks is executed after the first round of atomic checks and they work on the array of issues generated by the atomic checks to determine further vulnerabilities or discard false positives. Global checks provide an improved

decisional process, which comes in handy for checks that need to scan every JS/HTML of the target application before determining the presence or absence of a vulnerability. Global checks can specify their dependence on other atomic checks (e.g. "CSPJSCheck" and "CSPHTMLCheck" need to be executed before "CSPGlobalCheck").

Depending on the specific heuristics used by the check and the type of vulnerability that we're trying to detect, the tool may mark findings as `*Review Required*` meaning that the user is required to manually review the affected resources. Going forward, we plan to introduce the concept of *Severity* and *Confidence* to help screening between false positives and real issues.

Electronegativity Checks

Electronegativity currently implements the following checks:

AffinityGlobalCheck.js -
AFFINITY_GLOBAL_CHECK
AllowPopupCheck.js -
ALLOWPOPUPS_HTML_CHECK
AuxclickJSCheck.js -
AUXCLICK_JS_CHECK
AuxclickHTMLCheck.js -
AUXCLICK_HTML_CHECK
AvailableSecurityFixesGlobalCheck.js -
AVAILABLE_SECURITY_FIXES_GLOBAL_CHECK
BlinkFeaturesJSCheck.js -
BLINK_FEATURES_JS_CHECK
BlinkFeaturesHTMLCheck.js -
BLINK_FEATURES_HTML_CHECK
CertificateErrorEventJSCheck.js -
CERTIFICATE_ERROR_EVENT_JS_CHECK
CertificateVerifyProcJSCheck.js -
CERTIFICATE_VERIFY_PROC_JS_CHECK
ContextIsolationJSCheck.js -
CONTEXT_ISOLATION_JS_CHECK
CustomArgumentsJSCheck.js -
CUSTOM_ARGUMENTS_JS_CHECK
CustomArgumentsJSONCheck.js -
CUSTOM_ARGUMENTS_JSON_CHECK
CSPGlobalCheck.js -
CSP_GLOBAL_CHECK
DangerousFunctionsJSCheck.js -
DANGEROUS_FUNCTIONS_JS_CHECK
ElectronVersionJSONCheck.js -
ELECTRON_VERSION_JSON_CHECK
ExperimentalFeaturesHTMLCheck.js -
EXPERIMENTAL_FEATURES_HTML_CHECK
ExperimentalFeaturesJSCheck.js -
EXPERIMENTAL_FEATURES_JS_CHECK
HTTPResourcesJSCheck.js -
HTTP_RESOURCES_JS_CHECK
HTTPResourcesHTMLCheck.js -
HTTP_RESOURCES_HTML_CHECK
InsecureContentHTMLCheck.js -
INSECURE_CONTENT_HTML_CHECK
InsecureContentJSCheck.js -

INSECURE_CONTENT_JS_CHECK

LimitNavigationJSCheck.js -

LIMIT_NAVIGATION_JS_CHECK

LimitNavigationGlobalCheck.js -

LIMIT_NAVIGATION_GLOBAL_CHECK

NodeIntegrationHTMLCheck.js -

NODE_INTEGRATION_HTML_CHECK

NodeIntegrationAttachEventJSCheck.js -

NODE_INTEGRATION_ATTACH_EVENT_JS_CHECK

NodeIntegrationJSCheck.js -

NODE_INTEGRATION_JS_CHECK

OpenExternalJSCheck.js -

OPEN_EXTERNAL_JS_CHECK

PermissionRequestHandlerJSCheck.js -

PERMISSION_REQUEST_HANDLER_JS_CHECK

PermissionRequestHandlerGlobalCheck.js -

PERMISSION_REQUEST_HANDLER_GLOBAL_CHECK

PreloadJSCheck.js -

PRELOAD_JS_CHECK

ProtocolHandlersJSCheck.js -

PROTOCOL_HANDLER_JS_CHECK

SandboxJSCheck.js -

SANDBOX_JS_CHECK

SecurityWarningsDisabledJSCheck.js -

SECURITY_WARNINGS_DISABLED_JS_CHECK

SecurityWarningsDisabledJSONCheck.js -

SECURITY_WARNINGS_DISABLED_JSON_CHECK

WebSecurityHTMLCheck.js -

WEB_SECURITY_HTML_CHECK

WebSecurityJSCheck.js -

WEB_SECURITY_JS_CHECK

AFFINITY_GLOBAL_CHECK

AffinityGlobalCheck - Review the use of the `affinity` property

When specified, renderers with the same affinity will run in the same renderer process. Note that due to reusing the renderer process, certain `webPreferences` options will also be shared between the web pages even when you specified different values for them. This can lead to unexpected security configuration overrides.

Risk

Improper use of `affinity` property can cause the unwanted share of `webPreferences` options

Auditing

It is suggested to use exact same `webPreferences` for web pages with the same affinity. Look for all occurrences of the `affinity` attribute and compare their values:

HTML

```html
<webview src="https://doyensec.com" webpreferences="affinity=secPrefs"></webview>
```

JS

```js
firstWin = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true,
      affinity: "secPrefs"
    }
  })

  secondWin = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: false,
      affinity: "secPrefs"
    }
  })
```

References

- https://electronjs.org/docs/all#new-browserwindowoptions

ALLOWPOPUPS_HTML_CHECK

ALLOWPOPUPS_HTML_CHECK - 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 the specific `allowpopups` blinkfeatures attribute in `webview` tags:

```
<webview src="https://doyensec.com/" allowpopups></webview>
```

References

- https://electronjs.org/docs/all#10-do-not-use-allowpopups

AUXCLICK_JS_CHECK

AUXCLICK_JS_CHECK - 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 *willnavigate* 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();
    }
})
```

However, libchromiumcontent will trigger middle-click events as `auxclick` instead of `click`. Your application has to explicitly disable this insecure behaviour using something like:

```
mainWindow = new BrowserWindow({
    "webPreferences": {
        "disableBlinkFeatures ": "Auxclick"
    }
});
```

References

- https://github.com/electron/electron/issues/10315

AUXCLICK_HTML_CHECK

AUXCLICK_HTML_CHECK - 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 *willnavigate* 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();
    }
})
```

However, libchromiumcontent will trigger middle-click events as `auxclick` instead of `click`.

If you use `webview`, your application has to explicitly disable this insecure behaviour using something like:

```
<webview src="https://www.github.com/" disableblinkfeatures="Auxclick"></webview>
```

References

- https://github.com/electron/electron/issues/10315

AVAILABLE_SECURITY_FIXES_GLOBAL_CHECK

AVAILABLE_SECURITY_FIXES_GLOBAL_CHECK - Detect if there are security patches available for your current Electron version

Electronegativity has a mechanism to automatically fetch the latest Electron releases along with their change logs from electron/releases. The repository is managed by the original @electron-bot and is updated every ten minutes. A new commit is published if any of the following change:

- number of Electron releases on GitHub
- number of Electron releases on npm
- npm `electron@beta` version
- npm `electron@latest` version

The resulting `releases.<etag>.json` file is parsed by the `AVAILABLE_SECURITY_FIXES_GLOBAL_CHECK`, which determines whether the Electron version of the targeted application is missing the latest available security patches or if it can be regarded as safe.

In order to make this decision Electronegativity parses the changelog body of the interested versions, looking for the presence of `[security]` tags or **Security** sections.

Right now Electronegativity is using `ELECTRON_VERSION_JSON_CHECK` to infer the target's Electron version by parsing its `package.json` file. As suggested in #30, there may be other ways to detect it for already packed applications.

Risk

Not using the latest safest version of Electron may expose the application to security risks.

Auditing

Compare your Electron version with the latest Electron releases, looking for security patches.

References

- https://github.com/electron/releases

BLINK_FEATURES_JS_CHECK

BLINK_FEATURES_JS_CHECK - Do not use Chromium’s experimental features

The `blinkFeatures` / `enableBlinkFeatures` flag can be used to selectively enable Blink (Chromium web browser engine) features, which increases the overall attack surface for production applications.

Risk

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

Auditing

Search for `blinkFeatures` / `enableBlinkFeatures` flags set to true within the webPreferences of BrowserWindow:

```
mainWindow = new BrowserWindow({
  "webPreferences": {
    "enableBlinkFeatures": "CSSVariables"
  }
});
```

References

- https://electronjs.org/docs/all#9-do-not-use-enableblinkfeatures

BLINK_FEATURES_HTML_CHECK

BLINK_FEATURES_CHECK - Do not use Chromium’s experimental features

The `blinkFeatures` / `enableBlinkFeatures` flag can be used to selectively enable Blink (Chromium web browser engine) features, which increases the overall attack surface for production applications.

Risk

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

Auditing

Search for `blinkFeatures` / `enableBlinkFeatures` flags set to true within webview tags:

```
<webview src="https://doyensec.com/" blinkfeatures="PreciseMemoryInfo,CSSVariables"></webview>
```

References

- https://electronjs.org/docs/all#9-do-not-use-enableblinkfeatures

CERTIFICATE_ERROR_EVENT_JS_CHECK

CERTIFICATE_ERROR_EVENT_JS_CHECK - Insecure TLS Validation

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. This check looks for a common mistake that lead to insecure TLS validation which happens when the app voluntary opts-out of TLS certificates validation.

Risk

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

Auditing

Verify that the application does not explicitly opt-out from TLS validation.

Look for occurrences of `certificate-error`:

```
app.on('certificate-error', (event, webContents, url, error, certificate, callback) => { //error in cert
    if (url === 'https://doyensec.com') {
        callback(true) //its okay, go ahead anyway
    } else {
        callback(false)
    }
})
```

References

- https://en.wikipedia.org/wiki/HTTPS
- https://letsencrypt.org/how-it-works/

CERTIFICATE_VERIFY_PROC_JS_CHECK

CERTIFICATE_VERIFY_PROC_JS_CHECK - Insecure TLS Validation

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. This check looks for a common mistake that lead to insecure TLS validation which happens when the app voluntary opts-out of TLS certificates validation, or import untrusted certificates.

Risk

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

Auditing

Verify that the application does not explicitly opt-out from TLS validation.

Look for occurrences of `setCertificateVerifyProc`:

```
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
    }
})
```

Or `importCertificate`:

```
import { app } from "electron";

let options, callback;
app.importCertificate(options, callback);
```

References

- https://en.wikipedia.org/wiki/HTTPS
- https://letsencrypt.org/how-it-works/

CONTEXT_ISOLATION_JS_CHECK

CONTEXT_ISOLATION_JS_CHECK - Review the use of the `contextIsolation` option

`contextIsolation` introduces JavaScript context isolation for preload scripts, as implemented in Chrome Content Scripts. Using this **important** option, it is possible to obtain:

- Different JS contexts between renderers and preload scripts
- Different JS contexts between renderers and Electron’s framework code

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.

Even if you disabled `nodeIntegration`, `contextIsolation` is required for isolation. As of today, not enabling ContextIsolation allows malicious JavaScript code to execute Node APIs.

Risk

If `contextIsolation` is not used, malicious JS code can tamper JavaScript native functions as well as preload script code via *prototype pollution*.

Auditing

Ensure that `contextIsolation` is always set: `contextIsolation: true`

Starting from Electron v5, it is expected to be enabled by default.

References

- https://electronjs.org/docs/all#3-enable-context-isolation-for-remote-content
- https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en
- https://electronjs.org/docs/all#planned-breaking-api-changes-50

CUSTOM_ARGUMENTS_JS_CHECK

CUSTOM_ARGUMENTS_JS_CHECK - 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. This check will compare the custom command line arguments set using `appendArgument` or `appendSwitch` against a blacklist of dangerous arguments. The full blacklist is reported below:

Argument	Description
--ignore-certificate-errors	Ignores certificate-related errors.
--ignore-certificate-errors-spki-list	A set of public key hashes for which to ignore certificate-related errors. If the certificate chain presented by the server does not validate, and one or more certificates have public key hashes that match a key from this list, the error is ignored. The switch value must a be a comma-separated list of Base64-encoded SHA-256 SPKI Fingerprints (RFC 7469, Section 2.4).
--ignore-urlfetcher-cert-requests	Causes net::URLFetchers to ignore requests for SSL client certificates, causing them to attempt an unauthenticated SSL/TLS session. This is intended for use when testing various service URLs (eg: kPromoServerURL, kSbURLPrefix, kSyncServiceURL, etc)
--disable-web-security	Don't enforce the same-origin policy. (Used by people testing their sites.)
--host-rules	A comma-separated list of rules that control how hostnames are mapped.
--host-resolver-rules	Like `--host-rules` but these rules only apply to the host resolver.
--auth-server-whitelist	A comma-separated list of servers for which integrated authentication is enabled.
--auth-negotiate-delegate-whitelist	A comma-separated list of servers for which delegation of user credentials is required. Without `*` prefix the url has to match exactly.
--js-flags	Specifies the flags passed to the Node JS engine. It has to be passed when starting Electron if you want to enable the flags in the main process.
--allow-file-access-from-files	By default, file:// URIs cannot read other file:// URIs. This is an override for developers who need the old behavior for testing.
--allow-no-sandbox-job	Enables the sandboxed processes to run without a job object assigned to them. This flag is required to allow Chrome to run in RemoteApps or Citrix. This flag can reduce the security of the sandboxed processes and allow them to do certain API calls like shut down Windows or access the clipboard.

Argument	Description
--allow-running-insecure-content	By default, an https page cannot run JavaScript, CSS or plugins from http URLs. This provides an override to get the old insecure behavior
--cipher-suite-blacklist	Comma-separated list of SSL cipher suites to disable.
--debug-packed-apps	Adds debugging entries such as Inspect Element to context menus of packed apps.
--disable-features	Lists separated by commas the name of features to disable. See Chromium's base::FeatureList::InitializeFromCommandLine for details
--disable-kill-after-bad-ipc	Don't kill a child process when it sends a bad IPC message. Apart from testing, it is a bad idea from a security perspective to enable this switch.
--disable-webrtc-encryption	Disables encryption of RTP Media for WebRTC. When Chrome embeds Content, it ignores this switch on its stable and beta channels
--disable-xss-auditor	Disables Blink's XSSAuditor. The XSSAuditor mitigates reflective XSS.
--enable-local-file-accesses	Enable file accesses.
--enable-nacl-debug	Enables debugging via RSP over a socket.
--remote-debugging-address	Use the given address instead of the default loopback for accepting remote debugging connections. Should be used together with --remote-debugging-port. Note that the remote debugging protocol does not perform any authentication, so exposing it too widely can be a security risk.
--remote-debugging-port	Enables remote debug over HTTP on the specified port
--inspect	Node's inspect flag
--inspect-brk	Node's inspect and break flag
--explicitly-allowed-ports	Explicitly allows additional ports using a comma-separated list of port numbers.
--expose-internals-for-testing	Exposes the window.internals object to JavaScript for interactive development and debugging of layout tests that rely on it.
--gpu-launcher	Extra command line options for launching the GPU process (normally used for debugging). Use like renderer-cmd-prefix.
--nacl-dangerous-no-sandbox-nonsfi	Disable sandbox even for non SFI mode. This is particularly unsafe as non SFI NaCl heavily relies on the seccomp sandbox.

Argument	Description
--nacl-gdb-script	GDB script to pass to the nacl-gdb debugger at startup.
--net-log-capture-mode	Sets the granularity of events to capture in the network log. The mode can be set to one of the following values: "Default" "IncludeCookiesAndCredentials" "IncludeSocketBytes" See the functions of the corresponding name in Chromium's net_log_capture_mode.h for a description of their meaning.
--no-sandbox	Disables the sandbox for all process types that are normally sandboxed.
--reduce-security-for-testing	Enables more web features over insecure connections. Designed to be used for testing purposes only.
--unsafely-treat-insecure-origin-as-secure	Treat given (insecure) origins as secure origins. Multiple origins can be supplied.

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')
```

References

- https://electronjs.org/docs/all#appcommandlineappendargumentvalue
- https://electronjs.org/docs/all#appcommandlineappendswitchswitch-value
- https://electronjs.org/docs/all#supported-chrome-command-line-switches

CUSTOM_ARGUMENTS_JSON_CHECK

CUSTOM_ARGUMENTS_JSON_CHECK - Review the use of command line arguments in package.json

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. This check will compare the custom command line arguments set in the `package.json` scripts and configuration objects against a blacklist of dangerous arguments. The full blacklist is reported below:

Argument	Description
--ignore-certificate-errors	Ignores certificate-related errors.
--ignore-certificate-errors-spki-list	A set of public key hashes for which to ignore certificate-related errors. If the certificate chain presented by the server does not validate, and one or more certificates have public key hashes that match a key from this list, the error is ignored. The switch value must a be a comma-separated list of Base64-encoded SHA-256 SPKI Fingerprints (RFC 7469, Section 2.4).
--ignore-urlfetcher-cert-requests	Causes net::URLFetchers to ignore requests for SSL client certificates, causing them to attempt an unauthenticated SSL/TLS session. This is intended for use when testing various service URLs (eg: kPromoServerURL, kSbURLPrefix, kSyncServiceURL, etc)
--disable-web-security	Don't enforce the same-origin policy. (Used by people testing their sites.)
--host-rules	A comma-separated list of rules that control how hostnames are mapped.
--host-resolver-rules	Like `--host-rules` but these rules only apply to the host resolver.
--auth-server-whitelist	A comma-separated list of servers for which integrated authentication is enabled.
--auth-negotiate-delegate-whitelist	A comma-separated list of servers for which delegation of user credentials is required. Without `*` prefix the url has to match exactly.
--js-flags	Specifies the flags passed to the Node JS engine. It has to be passed when starting Electron if you want to enable the flags in the main process.
--allow-file-access-from-files	By default, file:// URIs cannot read other file:// URIs. This is an override for developers who need the old behavior for testing.
--allow-no-sandbox-job	Enables the sandboxed processes to run without a job object assigned to them. This flag is required to allow Chrome to run in RemoteApps or Citrix. This flag can reduce the security of the sandboxed processes and allow them to do certain API calls like shut down Windows or access the clipboard.

Argument	Description
--allow-running-insecure-content	By default, an https page cannot run JavaScript, CSS or plugins from http URLs. This provides an override to get the old insecure behavior
--cipher-suite-blacklist	Comma-separated list of SSL cipher suites to disable.
--debug-packed-apps	Adds debugging entries such as Inspect Element to context menus of packed apps.
--disable-features	Lists separated by commas the name of features to disable. See Chromium's base::FeatureList::InitializeFromCommandLine for details
--disable-kill-after-bad-ipc	Don't kill a child process when it sends a bad IPC message. Apart from testing, it is a bad idea from a security perspective to enable this switch.
--disable-webrtc-encryption	Disables encryption of RTP Media for WebRTC. When Chrome embeds Content, it ignores this switch on its stable and beta channels
--disable-xss-auditor	Disables Blink's XSSAuditor. The XSSAuditor mitigates reflective XSS.
--enable-local-file-accesses	Enable file accesses.
--enable-nacl-debug	Enables debugging via RSP over a socket.
--remote-debugging-address	Use the given address instead of the default loopback for accepting remote debugging connections. Should be used together with --remote-debugging-port. Note that the remote debugging protocol does not perform any authentication, so exposing it too widely can be a security risk.
--remote-debugging-port	Enables remote debug over HTTP on the specified port
--inspect	Node's inspect flag
--inspect-brk	Node's inspect and break flag
--explicitly-allowed-ports	Explicitly allows additional ports using a comma-separated list of port numbers.
--expose-internals-for-testing	Exposes the window.internals object to JavaScript for interactive development and debugging of layout tests that rely on it.
--gpu-launcher	Extra command line options for launching the GPU process (normally used for debugging). Use like renderer-cmd-prefix.
--nacl-dangerous-no-sandbox-nonsfi	Disable sandbox even for non SFI mode. This is particularly unsafe as non SFI NaCl heavily relies on the seccomp sandbox.

Argument	Description
--nacl-gdb-script	GDB script to pass to the nacl-gdb debugger at startup.
--net-log-capture-mode	Sets the granularity of events to capture in the network log. The mode can be set to one of the following values: "Default" "IncludeCookiesAndCredentials" "IncludeSocketBytes" See the functions of the corresponding name in Chromium's net_log_capture_mode.h for a description of their meaning.
--no-sandbox	Disables the sandbox for all process types that are normally sandboxed.
--reduce-security-for-testing	Enables more web features over insecure connections. Designed to be used for testing purposes only.
--unsafely-treat-insecure-origin-as-secure	Treat given (insecure) origins as secure origins. Multiple origins can be supplied.

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 every `script` or `config` entry in the `package.json` of the target application:

```
"scripts": {
  "run": "node dist/index.js --disable-web-security"
}
```

References

- https://docs.npmjs.com/misc/scripts
- https://docs.npmjs.com/misc/scripts#special-packagejson-config-object
- https://electronjs.org/docs/all#supported-chrome-command-line-switches

CSP_GLOBAL_CHECK

CSP_GLOBAL_CHECK - CSP presence check and review

Electron apps when possible should implement a Content Security Policy (CSP) as an additional layer of protection against cross-site-scripting attacks and data injection attacks. There are two ways to set a CSP in Electron: via the `webRequest.onHeadersReceived` handler or directly in the markup using a `<meta>` tag.

This check determines whether a CSP policy is set or or is missing, both via JS or HTML:

- If a CSP is detected, Electronegativity looks for weak directives using a library based on the csp-evaluator.withgoogle.com online tool.
- If no CSP is found, Electronegativity issues a warning.

Risk

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.

Auditing

Check whether a CSP is defined and use the csp-evaluator.withgoogle.com tool to review its directives.

References

- https://github.com/electron/electron/blob/master/docs/tutorial/security.md#6-define-a-content-security-policy

DANGEROUS_FUNCTIONS_JS_CHECK

DANGEROUS_FUNCTIONS_JS_CHECK - Do not use dangerous functions with user-supplied data

`insertCSS`, `executeJavaScript` functions allow to inject respectively CSS and JavaScript from the main process to the renderer process. Also `eval`, `Function`, `setTimeout`, `setInterval` and `setImmediate` 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. This check detects the use of dangerous functions with dynamic arguments, and delegates the review to the user.

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`, `eval`, `Function`, `setTimeout`, `setInterval` and `setImmediate` with user-supplied input in both `BrowserWindow`, webview tag and all other JavaScript resources.

References

- https://electronjs.org/docs/all#winevalcode
- https://electronjs.org/docs/all#contentsinsertcsscss
- https://electronjs.org/docs/all#contentsexecutejavascriptcode-usergesture-callback

ELECTRON_VERSION_JSON_CHECK

ELECTRON_VERSION_JSON_CHECK - Keep dependencies up-to-date

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□

Risk

Older versions of the Electron framework may contain vulnerabilities, including nodeIntegration bypasses.

Auditing

Ensure that the Electron version bundled with your software is the latest stable release.

References

- https://electronjs.org/releases

EXPERIMENTAL_FEATURES_HTML_CHECK

EXPERIMENTAL_FEATURES_HTML_CHECK - Do not use Chromium’s experimental features

The `experimentalFeatures` and `experimentalCanvasFeatures` flags can be used to enable Chromium’s experimental features, which increase the overall attack surface for production applications.

Risk

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

Auditing

Search for `experimentalFeatures`, `experimentalCanvasFeatures` flags set to true within the webPreferences of webview tags:

```
<webview src="https://doyensec.com" webpreferences="experimentalCanvasFeatures=true"></webview>
```

References

- https://electronjs.org/docs/all#8-do-not-enable-experimental-features

EXPERIMENTAL_FEATURES_JS_CHECK

EXPERIMENTAL_FEATURES_JS_CHECK - Do not use Chromium’s experimental features

The `experimentalFeatures` and `experimentalCanvasFeatures` flags can be used to enable Chromium’s experimental features, which increase the overall attack surface for production applications.

Risk

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

Auditing

Search for `experimentalFeatures`, `experimentalCanvasFeatures` flags set to true within the webPreferences of BrowserWindow:

```
mainWindow = new BrowserWindow({
    "webPreferences": {
        "experimentalCanvasFeatures": true
    }
});
```

References

- https://electronjs.org/docs/all#8-do-not-enable-experimental-features

HTTP_RESOURCES_JS_CHECK

HTTP_RESOURCES_JS_CHECK - 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.

Directly fetching content using plain-text HTTP opens your application to Man-in-the-Middle attacks.

Risk

Man-in-the-Middle attacks. If `nodeIntegration` is also enabled, an attacker can inject malicious JavaScript and compromise the user’s host.

Auditing

Look for resources loaded using *http*, for example:

```
const win = new BrowserWindow({...});
win.loadURL('http://example.com/');
```

References

- https://electronjs.org/docs/all#1-only-load-secure-content

HTTP_RESOURCES_HTML_CHECK

HTTP_RESOURCES_JS_CHECK - 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.

Directly fetching content using plain-text HTTP opens your application to Man-in-the-Middle attacks.

Risk

Man-in-the-Middle attacks. If `nodeIntegration` is also enabled, an attacker can inject malicious JavaScript and compromise the user's host.

Auditing

Look for resources loaded using *http*, for example:

```
<webview src="http://doyensec.com"></webview>
```

References

- https://electronjs.org/docs/all#1-only-load-secure-content

INSECURE_CONTENT_HTML_CHECK

INSECURE_CONTENT_HTML_CHECK - 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.

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.

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

Search for allowRunningInsecureContent set to true/1 within the`webPreferences` attribute in the `webview` tag:

```
<webview src= "https://doyensec.com" webPreferences="allowRunningInsecureContent=true"></webview>
```

References

- https://electronjs.org/docs/all#7-do-not-set-allowrunninginsecurecontent-to-true

INSECURE_CONTENT_JS_CHECK

INSECURE_CONTENT_JS_CHECK - 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.

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.

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

Search for `allowRunningInsecureContent` set to true/1 within the `webPreferences` of BrowserWindow:

```
mainWindow = new BrowserWindow({
   "webPreferences": {
      "allowRunningInsecureContent": true
   }
});
```

References

- https://electronjs.org/docs/all#7-do-not-set-allowrunninginsecurecontent-to-true

LIMIT_NAVIGATION_JS_CHECK

LIMIT_NAVIGATION_JS_CHECK - Detects if `on()` for 'will-navigate' and 'new-window' events is used

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.

Electron applications can limit the navigation flows by implementing a similar code snippet:

```
win.webContents.on('will-navigate', (event, newURL) => {
  if (win.webContents.getURL() !== 'https://doyensec.com' ) {
    event.preventDefault();
  }
})
```

Risk

This setting can be used to limit the exploitability of certain issues. Not enforcing navigation limits leaves the Electron application under full control to remote origins in case of accidental navigation.

Auditing

Check every callback of the `will-navigate` and the `new-windows` events. These callbacks should be reviewed thoroughly to exclude potential flaws in the origin's validation mechanism.

References

- https://electronjs.org/docs/api/web-contents#event-will-navigate

LIMIT_NAVIGATION_GLOBAL_CHECK

LIMIT_NAVIGATION_GLOBAL_CHECK - Missing navigation limits using `.on` 'new-window' and 'will-navigate' events

This Global Check detects if the atomic LIMIT_NAVIGATION_JS_CHECK is missing, meaning that the `.on` call for 'new-window' and 'will-navigate' is never made in the target application to limit the application's navigation only to trusted origins.

Risk

This setting can be used to limit the exploitability of certain issues. Not enforcing navigation limits leaves the Electron application under full control to remote origins in case of accidental navigation.

Auditing

Check for callbacks on the `will-navigate` and the `new-windows` events. These callbacks should be implemented to exclude potential flaws in the origin's validation mechanism.

References

- https://electronjs.org/docs/api/web-contents#event-will-navigate

NODE_INTEGRATION_HTML_CHECK

NODE_INTEGRATION_HTML_CHECK - 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 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>
```

References

- https://electronjs.org/docs/all#2-disable-nodejs-integration-for-remote-content
- https://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html

NODE_INTEGRATION_ATTACH_EVENT_JS_CHECK

NODE_INTEGRATION_ATTACH_EVENT_JS_CHECK - 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

It is 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, 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 by 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()
        }
    })
})
```

References

- https://electronjs.org/docs/all#2-disable-nodejs-integration-for-remote-content
- https://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html

NODE_INTEGRATION_JS_CHECK

NODE_INTEGRATION_JS_CHECK - 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:

```
const mainWindow = new BrowserWindow();
```

References

- https://electronjs.org/docs/all#2-disable-nodejs-integration-for-remote-content
- https://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html

OPEN_EXTERNAL_JS_CHECK

OPEN_EXTERNAL_JS_CHECK - 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.

References

- https://electronjs.org/docs/all#shellopenexternalurl-options-callback
- https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en?slide=39

PERMISSION_REQUEST_HANDLER_JS_CHECK

PERMISSION_REQUEST_HANDLER_JS_CHECK - 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)
    }
})
```

The current version of Electron allows control of the following permissions:

- media
- geolocation
- notifications
- midiSysex
- pointerLock
- fullscreen
- openExternal

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 of certain issues. Not enforcing custom checks for permission requests (e.g. media) leaves the Electron application under full control of the remote origin. For instance, a Cross-Site Scripting vulnerability can be used to access the browser media system and silently record audio/video. While browsers have implemented notification to inform the user that a remote site is capturing the webcam stream, Electron does not display any notification.

Auditing

Review all occurrences of `setPermissionRequestHandler`. If used, manually evaluate the implementation and security of the custom callbacks. If not used, the application does not limit session permissions at all thus the configuration is open to abuses.

References

- https://electronjs.org/docs/all#sessetpermissionrequesthandlerhandler
- https://electronjs.org/docs/all#4-handle-session-permission-requests-from-remote-content

PERMISSION_REQUEST_HANDLER_GLOBAL_CHECK

PERMISSION_REQUEST_HANDLER_GLOBAL_CHECK - Detect missing `setPermissionRequestHandler` for untrusted origins

This GlobalCheck detects if the atomic PERMISSION_REQUEST_HANDLER_JS_CHECK is missing, meaning that the `setPermissionRequestHandler` is not set in the target application.

Risk

This setting can be used to limit the exploitability of certain issues. Not enforcing custom checks for permission requests (e.g. media) leaves the Electron application under full control of the remote origin. For instance, a Cross-Site Scripting vulnerability can be used to access the browser media system and silently record audio/video. While browsers have implemented notification to inform the user that a remote site is capturing the webcam stream, Electron does not display any notification.

Auditing

Review all occurrences of `setPermissionRequestHandler`. If used, manually evaluate the implementation and security of the custom callbacks. If not used, the application does not limit session permissions at all thus the configuration is open to abuses.

References

- https://electronjs.org/docs/all#sessetpermissionrequesthandlerhandler
- https://electronjs.org/docs/all#4-handle-session-permission-requests-from-remote-content

PRELOAD_JS_CHECK

PRELOAD_JS_CHECK - Review the use of preload scripts

Despite disabling `nodeIntegration` or 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

```
const { 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. Finally, 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 must be carefully reviewed.

Risk

Improper use of preload scripts can introduce nodeIntegration or sandbox bypasses, in addition to other vulnerabilities. If `contextIsolation` is not used, there is also the risk that malicious code may be able to tamper sensitive operations with prototype pollution attacks.

Auditing

Search for the `preload` directive within the `webPreferences` of `BrowserWindow`. Manually review all imported scripts.

References

- https://electronjs.org/docs/all#preload

PROTOCOL_HANDLER_JS_CHECK

PROTOCOL_HANDLER_JS_CHECK - Review the use of custom protocol handlers

Electron allows to define custom protocol handlers so that the application can use deep linking to exercise specific features. Since external 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
- registerStreamProtocol

Our check searches for those occurrences. Users are required to manually review the implementation of each of them.

References

- https://electronjs.org/docs/all#appsetasdefaultprotocolclientprotocol-path-args
- https://electronjs.org/docs/all#protocolregisterstandardschemesschemes-options
- https://electronjs.org/docs/all#protocolregisterserviceworkerschemesschemes
- https://electronjs.org/docs/all#protocolregisterfileprotocolscheme-handler-completion
- https://electronjs.org/docs/all#protocolregisterstringprotocolscheme-handler-completion
- https://electronjs.org/docs/all#protocolregisterhttpprotocolscheme-handler-completion
- https://electronjs.org/docs/all#protocolregisterstreamprotocolscheme-handler-completion
- https://electronjs.org/docs/all#protocolregisterbufferprotocolscheme-handler-completion

SANDBOX_JS_CHECK

SANDBOX_JS_CHECK - Use `sandbox` for untrusted origins

While nodeIntegration tackles the problem of limiting access to Node.js primitives from a remote untrusted origin, it does neither mitigate security flaws introduced by Electron’s “glorified” APIs, nor it solves prototype pollution. Electron extends the default JavaScript APIs (e.g. `window.open` returns an instance of `BrowserWindowProxy`) which leads to a larger attack surface.

Instead, sandboxed renderers expose default JavaScript APIs. 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.

This option should be enabled whenever there is a need of loading untrusted content in a browser window**Please note that at the time of writing, `sandbox` is still experimental and may introduce functional side-effects.**

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 `sandbox`.

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.

References

- https://electronjs.org/docs/all#sandbox-option

SECURITY_WARNINGS_DISABLED_JS_CHECK

SECURITY_WARNINGS_DISABLED_JS_CHECK - Detects if Electron's security warnings have been disabled by the developers via JS

> From Electron 2.0 on, developers will see warnings and recommendations 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.

It is possible to force-enable or force-disable these warnings by setting `ELECTRON_DISABLE_SECURITY_WARNINGS` on either process.env or the window object:

- `process.env['ELECTRON_DISABLE_SECURITY_WARNINGS'] = true;`
- `win.webContents["ELECTRON_DISABLE_SECURITY_WARNINGS"] = true;`
- `window.ELECTRON_DISABLE_SECURITY_WARNINGS = true;` as detailed in electron/electron#11970 too.

Note that if you decide to avoid these warnings because - for example - you run Electronegativity for each build, you may filter this tool's finding by setting the desired output severity (e.g. `--severity=Low`).

Risk

Disabling this warning may hide the presence of misconfigurations or insecure patterns to the developers.

Auditing

Check if the flag is disabled in one of the aforementioned ways.

References

- https://electronjs.org/docs/tutorial/security#electron-security-warnings

SECURITY_WARNINGS_DISABLED_JSON_CHECK

SECURITY_WARNINGS_DISABLED_JS_CHECK - Detects if Electron's security warnings have been disabled by the developers via `package.json`

> From Electron 2.0 on, developers will see warnings and recommendations 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.

It is possible to force-enable or force-disable these warnings by setting `ELECTRON_DISABLE_SECURITY_WARNINGS` as an env flag via the `scripts` or `config` objects:

```
{
  "name": "test-app",
  "version": "0.0.1",
  "description": "Here is a test program description",
  "main": "desktop/src/index.js",
  "license": "MIT",
  "scripts": {
    "build": "../node_modules/.bin/tsc -p .",
    "start": "yarn run build && cd ../dist && ELECTRON_DISABLE_SECURITY_WARNINGS=true ../desktop/node_modules/.bin/electron .",
    "postinstall": "install-app-deps"
  }
}
```

Note that if you decide to avoid these warnings because - for example - you run Electronegativity for each build, you may filter this tool's finding by setting the desired output severity (e.g. `--severity=Low`).

Risk

Disabling this warning may hide the presence of misconfigurations or insecure patterns to the developers.

Auditing

Check if the flag is disabled in one of the aforementioned ways.

References

- https://electronjs.org/docs/tutorial/security#electron-security-warnings

WEB_SECURITY_HTML_CHECK

WEB_SECURITY_HTML_CHECK - 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 (SOP) will not be enforced.

> Please note that the Same-Origin Policy is 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.

Risk

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

Auditing

In the `webview` tag, look for `disablewebsecurity` attribute:

```
<webview src="https://doyensec.com/" disablewebsecurity></webview>
```

Additionally, search for the runtime flag `—disable-web-security` in the `package.json`, and within the application codebase.

References

- https://electronjs.org/docs/all#5-do-not-disable-websecurity
- https://electronjs.org/docs/all#disablewebsecurity

WEB_SECURITY_JS_CHECK

WEB_SECURITY_JS_CHECK - 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 (SOP) will not be enforced.

> Please note that the Same-Origin Policy is 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.

Risk

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

Auditing

Check the `webPreferences` object passed to `BrowserWindow`, and look for `webSecurity` false:

```
mainWindow = new BrowserWindow({
   "webPreferences": {
      "webSecurity": false
   }
});
```

Additionally, search for the runtime flag `—disable-web-security` in the `package.json`, and within the application codebase.

References

- https://electronjs.org/docs/all#5-do-not-disable-websecurity