Rise of Inspectron: Automated Black-box Auditing of Cross-platform Electron Apps

Mir Masood Ali, Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis
University of Illinois Chicago, {mali92, mghas2, ckanich, polakis}@uic.edu

Abstract

Browser-based cross-platform applications have become increasingly popular as they allow software vendors to sidestep two major issues in the app ecosystem. First, web apps can be impacted by the performance deterioration affecting browsers, as the continuous adoption of diverse and complex features has led to bloating. Second, re-developing or porting apps to different operating systems and execution environments is a costly, error-prone process. Instead, frameworks like Electron allow the creation of standalone apps for different platforms using JavaScript code (e.g., reused from an existing web app) and by incorporating a stripped down and configurable browser engine. Despite the aforementioned advantages, these apps face significant security and privacy threats that are either *non-applicable* to traditional web apps (due to the lack of access to certain system-facing APIs) or *ineffective* against them (due to countermeasures already baked into browsers). In this paper we present Inspectron, an automated dynamic analysis framework that audits packaged Electron apps for potential security vulnerabilities stemming from developers' deviation from recommended security practices. Our study reveals a multitude of insecure practices and problematic trends in the Electron app ecosystem, highlighting the gap filled by Inspectron as it provides extensive and comprehensive auditing capabilities for developers and researchers.

1 Introduction

The contemporary client-side web programming ecosystem has enabled effectively effortless cross-platform web app development: a full-featured web app can present a unified experience across Linux, Windows, MacOS, or any other platform that supports a fully functioning modern browser. This ease of portability, along with the standardization of access to lower-level OS functionality through the Node.js platform, gave rise to Electron, a system that allows combining the open-source Chrome and Node.js projects with a developer's code to create a freestanding desktop app, which does not require access to a system browser or the Internet to provide its functionality.

While there are clear advantages to relying on these two incredibly well-engineered components, doing so introduces unique challenges. First, there are inevitable issues when using these software artifacts outside of the context for which they were designed. Second, the web platform's ubiquity and importance has resulted in it attracting significant malicious attention and, thus, substantial effort is put into the rapid release and distribution of browser updates. Finally, these artifacts are themselves massively complex (necessarily so), and using them as an abstraction upon which to build yet more complexity is a fraught endeavor.

In spite of these drawbacks, the benefit derived from fully cross-platform desktop apps that can reuse large parts of existing web-based interface code is substantial: Slack, Discord, Twitch, WhatsApp, and many more segment-leading companies distribute Electron-based desktop apps. Thus, it is important to more closely investigate the risks inherent in the use of the Electron platform. Relying on a stripped-down version of Chrome's engine results in certain security mechanisms not becoming available in a timely manner. More crucially, existing security protections that have been baked into web browsers for years now become a configurable option for developers; prior research has shown how developers struggle with correctly configuring or deploying security mechanisms [1–4]. This can also lead to a fragmented ecosystem where different apps have different versions of the underlying Chrome engine or Electron framework, akin to the fragmentation problem affecting the Android ecosystem [5, 6]. As web and mobile apps are known to lag behind the latest version of third-party libraries [7, 8], such patterns within the Electron ecosystem could expose users to significant threats. Because Electron apps package static versions of their upstream dependencies, attackers can leverage known exploits during the window between patching in Chrome and the distribution of new versions of Electron apps that incorporate the updated Chrome engine. Finally, cross-platform apps have additional capabilities compared to their web counterparts that are closer to those of native applications. Electron's security model aims to isolate web-facing

functionality from system-facing functionality; however, insecure developer practices and misconfigurations can lead to web-facing code influencing system-facing functionality.

While many weaknesses of the Electron platform can be mitigated through proper use of tools like Elecronegativity or defenses against specific classes of attacks [9, 10], there is a clear necessity for a system that is both dynamic and automated, which can continually investigate a more comprehensive range of failure scenarios than existing tools. To this end, we develop Inspectron, a framework designed for uncovering the misconfiguration of security mechanisms, or the lack thereof, in Electron apps, through an automated black-box auditing process. Our automated black-box system uses instrumented versions of Electron to detect and report on information flow in various entities that could affect an app's security. These entities include Inter-process communication (IPC), page navigation, and cross-context JavaScript execution. Our system locates the binary executable and determines the version of Node.js in use. It then uses Puppeteer to run the application on the instrumented Electron, enabling us to dynamically perform automated client-side checks.

In summary, our work makes the following contributions:

- We build Inspectron, a dynamic, black-box framework that audits Electron apps for 16 classes of misconfigurations without source code access, by detecting the runtime behavior of apps and gathering the evaluated definitions of function calls, event handlers, and framework preferences.
- We evaluate 109 Electron apps and find an array of issues in the implementation of various framework components, while outperforming the state-of-the-art.
- We perform a more comprehensive examination of 10 popular Electron apps by including pre-recorded user interaction traces, and find vulnerabilities in four apps and two instances of incorrect implementations of web standards by the Electron framework.
- We have responsibly disclosed our findings to the affected vendors, which has already resulted in a series of patches.

2 Background

The process model adopted by the Electron framework largely splits the app into two differently privileged contexts [11] (see Figure 1). This design is based on the motivation that a single process that renders arbitrary, insecure content could make the app susceptible to malicious code. Instead, Electron renders each new frame in its own process, while a single privileged process controls them and the app as a whole.

System-side (Main) process. This is the privileged process that controls the app, and has access to system-level functionality, including the native operating system's UI and Node.js modules. It also creates and interacts with less-privileged renderer processes. To protect sensitive user resources, the framework restricts access from third-party resources loaded in renderer processes.

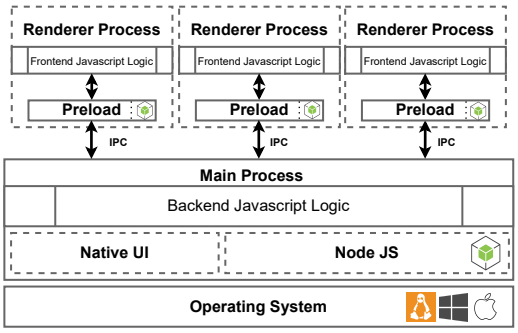

Figure 1: Architecture of Electron apps.

Renderer process. Each window (or embed) that is opened spawns a separate renderer process. Electron uses Chromium's Blink engine [12] to render web content – from HTML, CSS, and Javascript – within these windows. The execution logic within a renderer process can not directly access Node modules and, instead, interacts with the main process for requesting actions that need additional privileges.

Communication between processes. *Preload scripts* are used to expose functionality from the main process to a renderer process in two ways: (i) *Shared* `Window` *object.* Preload scripts in the renderer process have access to the global `Window` object and specific Node.js APIs. Scripts can wrap Node-based functionality and set global variables for the embedded web content to access. (ii) *Inter-process Communication (IPC).* Scripts can create new events on an IPC channel, and pass information between a renderer process and the main process. This allows the main process to securely handle privileged execution by verifying incoming messages. Next, we describe Electron's various security and privacy features, including those that the framework inherits from Chrome and instances where it deviates.

Web Preferences. Each new web embed (including a pop-up/webview/iframe) creates a corresponding `BrowserWindow` object that inherits the parent window's preferences by default. These include restrictions on the window's functionality. Appendix A.1 provides several detailed examples as well as their corresponding security implications.

Navigation Handling. While websites loaded within the browser regularly navigate away to different websites, this type of navigation is often considered undesirable within such an application. The Electron framework does not, by default, restrict any window from navigating to different domains or from opening new windows. Instead, developers may handle and verify navigation within their apps using several methods.

Content Security Policy (CSP). CSP is a security feature that allows developers to specify which resources a webpage can load and which sources of executable code are considered trusted. Developers can use CSP to block requests to external domains or to prevent the execution of untrusted code. By default, Electron does not implement a CSP on the web

content loaded within an app, and instead recommends that developers add their own implementations.

Sessions and Cookies. Web content is loaded within a default, persistent session that handles any information stored within cookies, storage, and other caches. As these sessions are managed by the app, information accessed from any third-party content can be directly managed by the app itself.

Cookies in the main process. While partitioning restrictions exist within the renderer process, the main process can access all cookies loaded within the app. The app can additionally alter session cookies to make them persistent, and also access HttpOnly cookies using Javascript code.

Origin partitioning. The renderer process inherits partitioning from Chrome. Any third-party content loaded in an iframe is limited in its interaction with the top-level browsing context, and can only access its own cookie jar.

Plaintext cookies. While the Chrome browser encrypts cookies when stored in the filesystem, Electron apps store them in plaintext by default. The framework offers an optional *fuse* in case an app intends to encrypt cookies.

Permissions. Electron inherits Chrome's permission API and handles the same types of permissions [13]. However, unlike Chrome, the framework automatically approves all permissions. Developers have the option to prompt users to request access, and are encouraged to explicitly handle permission requests to avoid providing default access to third-party content loaded within the application. Below, we describe how Electron's implementation diverges from Chrome's.

Media Device Access. Within both Chrome and Electron, access to media devices can be made by calling `navigator.mediaDevices.getUserMedia()`. While approving permission access, Electron lets developers either grant access to all media or deny any access, without individually allowing access to the webcam, microphone, or screen.

Screen Recording. Chrome makes access to the screen available through a different API call, i.e., `getDisplayMedia()`. However, Electron requires access to screen recording through a separate `desktopCapturer` object that developers need to expose from the main process, and the renderer process can make the same API call used to access other media, `getUserMedia()`, with different constraints passed in the arguments. Electron's requirement to explicitly expose screen capture makes it more secure from third-party access.

File System Access API. The WICG directive recommends restricting the files that can be picked to be loaded by the user to avoid picking sensitive files under the root or `/etc` folders [14]. Chrome implements this restriction to content loaded within the browser but Electron does not enforce such a restriction.

2.1 Threat Model

Due to the distinct nature of Electron apps, prior to conducting our research we first examined more than 50 vulnerability

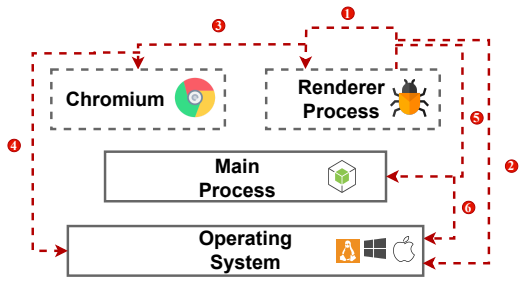

Figure 2: An overview of the threat model is shown here.

reports that have been submitted to various Electron apps over the past 7 years [15]. In this preliminary analysis we summarized and categorized potential avenues for exploiting vulnerabilities; these exploitability patterns helped define our threat model and also guided our design of Inspectron. In this work, we encapsulate knowledge gathered from a wide-range of attack vectors into a comprehensive testing framework. While the attack vectors have been demonstrated in real-world exploits, to the best of our knowledge, Inspectron is the first *dynamic analysis* tool that extensively evaluates Electron apps for these vectors. Additionally, Inspectron's analysis includes five attack vectors that are not considered by Elecronegativity, a state-of-the-art static analysis tool (discussed in §4.1 and Appendix A.3). The Electron framework encompasses multiple components and, as a result, the majority of reported exploits stem from the interconnectedness of bugs across these different components. We detail a motivating example in Appendix A.2. Below, we describe our threat model, which covers different paths for exploiting vulnerabilities in Electron apps. We consider an attacker that is either (1) a user of the application attacking another user, or (2) a malicious third-party component loaded within the app, e.g., scripts loaded from third-party libraries, content loaded within iframes and webviews, and third-party webpages to which the app permits in-app navigation. Figure 2 serves as a point of reference and offers a high-level overview.

Code Execution in the Renderer Process. In Electron apps, external users and externally-sourced third-party resources engage with the application's logic through the renderer process. By gaining control over code execution within the renderer process, attackers can begin compromising the security of other components of the application. When attempting to execute code within the app's existing user-facing window, various techniques can be employed. For instance, unsafe Content Security Policy (CSP) configurations can create an avenue for executing cross-site scripting (XSS) attacks. Another approach involves taking over third-party resources, and leveraging their vulnerabilities to execute malicious code. Additionally, bypassing input sanitization measures can enable the injection and execution of harmful code within user-facing components. Safeguarding against these vulnerabilities necessitates implementing secure

CSP configurations, robust input sanitization practices, and stringent controls on the interaction with third-party resources. To execute code on a different page or window, attackers need additional strategies. One method involves bypassing restrictions to navigation to malicious third-party sites, allowing them to load and execute code within a different page under their control. Similarly, opening these sites in a new window or frame can enable executing code outside the confines of the current window. Once an attacker finds a way to execute code within the renderer process, they can then chain their attack by taking one of the following routes.

❶ → ❷ **Privileged Renderer Process.** The renderer process can have direct access to Node.js modules. Additionally, when sharing a context with preloaded APIs, the renderer process can use prototype pollution attacks [16] to also gain direct access to Node.js modules. Direct access to Node.js modules within the renderer process can help malicious code compromise the underlying system.

❶ → ❸ → ❹ **Chromium-based Exploits.** Chrome regularly releases reports on vulnerabilities discovered within the Blink and V8 engines, which are the underlying components powering the browser and, consequently, Electron apps. Despite the fact that the Chrome team promptly releases patches and updates for their browser to address these vulnerabilities, app developers who rely on older versions of the Chromium framework may remain exposed to these exploits. These vulnerabilities capitalize on the inner workings of the Blink and V8 engines, thus enabling attackers to directly execute shell code on the underlying system.

❶ → ❺ → ❻ **Incomplete Checks in the Main Process.** In the absence of vulnerabilities enabling one of the previously described exploitation approaches that require a privileged renderer process, this route offers an alternative exploit that takes advantage of incomplete checks in the main process. Malicious code may take advantage of a lack of checks on the origin of inter-process communication (IPC) messages, including oversight in responses to messages from the Preload API. Additionally, they may exploit the use of incomplete checks on the use of custom protocols during navigation, and sanitization errors in cross-context JS execution.

3 Inspectron: Design and Implementation

Here we detail the design and implementation of Inspectron. Figure 3 provides an overview of our system and workflow.

❶ **Packaged App.** Electron apps are distributed with varying directory structures depending on the target OS. Depending on the distributable, Inspectron temporarily mounts the packaged app and extracts relevant files. Inspectron accesses app-specific logic from a `resources` directory, which is also the directory from where Electron accesses source code [17]. It then identifies the app's binary executable file, which is used for version checks. When the binary file is executed as a Node.js process using the `ELECTRON_RUN_AS_NODE` command line flag, Inspectron can access and use the `process.versions` object to determine the Node.js version that the app uses. This object contains key-value pairs that indicate the Node.js version, the V8 JavaScript engine, and other modules used to build the app.

❷ **Instrumented Electron.** Electron has a different app Binary Interface (ABI) from a Node.js binary. Therefore, while Electron supports developers using native Node.js modules, those modules must be recompiled [18]. As a result, the app-specific code extracted from the `resources` directory in the previous step can only be run against an Electron binary compiled with the same Node.js module version. Our instrumented version of Electron modifies relevant functions to output the status of specific variables and arguments when called, enabling Inspectron to identify and report on points of interest.

Web Preferences. Developers can customize the behavior of each page in a window or frame using the `webPreferences` property, enabling or disabling features such as `nodeIntegration`, `contextIsolation`, and `sandbox`. These features impact available privileges, and developers must evaluate them correctly throughout their app. Inspectron checks 12 security-related web preferences.

Command-line Switches. These can be used to configure an Electron app, enable or disable features, modify its behavior, or set debugging options. Inspectron provides runtime reports on the setting of 33 command-line switches.

Navigation Handling. Navigation can be constrained by adding event listeners to each opened window, so as to ensure users stay within the app's domain. The built-in `will-navigate` event allows intercepting and verifying navigation requests before being sent, enabling URL modification or cancellation. Additionally, developers must handle the `new-window` event by either preventing its opening or creating a new window with secure preferences. Even though the `new-window` event is deprecated in Electron v22, it remains relevant for apps that have yet to update their frameworks.

Inter-process Communication (IPC). In Electron, IPC is commonly used to communicate between the main process and the renderer processes. The main process controls the app's lifecycle and manages system resources, while the renderer processes handle rendering the user interface. Developers can share data, trigger events, and invoke methods using IPC messages. To ensure security, Electron recommends verifying the sender of IPC messages to prevent potential threats. If the sender is not trusted, the message should be rejected, preventing potential security threats [19] and ensuring the integrity of IPC messages. Inspectron reports and highlights custom IPC calls that require further evaluation to ensure that the sender of a message is always verified.

Cross-context JS Execution. Using `executeJavaScript()` developers can explicitly enable the injection of JavaScript from the main process to a renderer process. However, when user-supplied arguments are used with these functions, they can potentially execute harmful content and modify the app's

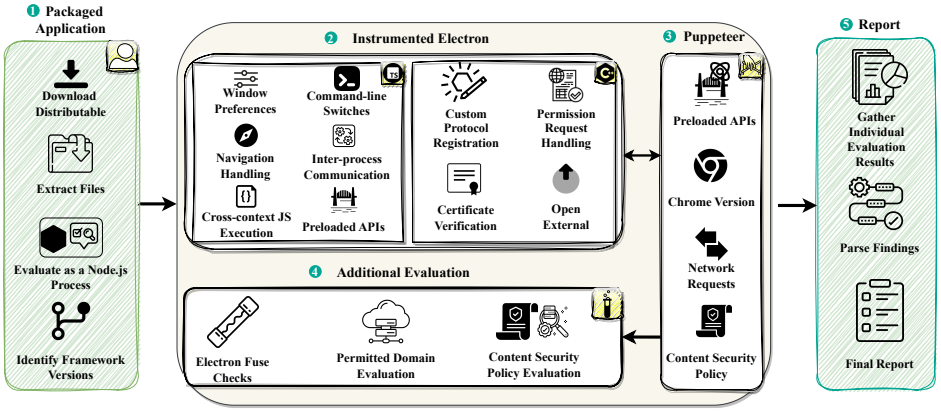

Figure 3: Inspectron's components and auditing workflow for evaluating Electron apps.

behavior. To mitigate this risk, it is important to verify the use of such functions with dynamic arguments. When the calls to `executeJavaScript()` are triggered, Inspectron reports on the use of such functions for further evaluation.

Preloaded APIs. Preload scripts and Electron's `contextBridge` can be used to expose functionality across contexts, set up custom event listeners, inject CSS styles and JavaScript code, and modify built-in browser APIs. Since APIs may be exposed as arbitrary objects and functions on the client-side `window` object, our framework collects all calls to the `contextBridge` available at a singular endpoint. Unlike the aforementioned checks, calls made to the `contextBridge` involve the renderer process and therefore cannot be directly reported to a file. Inspectron instead collects them in a global variable, `window.reportExposedAPIs`, which can be accessed by a Puppeteer script.

Custom Protocol Registration. New URI schemes can be registered for handling app requests, especially for custom network protocols or unavailable resources via HTTP/HTTPS. Custom protocols should be carefully considered when handling navigation, verifying IPC messages, and overriding certificate verification. Inspectron reports on custom protocol registrations to consider during the evaluation of other checks.

Permission Request Handling. Electron grants complete access to devices such as the camera, microphone, Bluetooth, and screen by default. It is recommended that developers explicitly handle permission requests. Inspectron reports on whether an app properly handles incoming permissions.

Certificate Verification. When loading resources over HTTPS, it is important to verify X.509 certificates - a functionality that is baked into Electron by default. However, developers may opt out of these checks and handle specific domains differently, especially during development. Inspectron reports on overrides of certificate verification.

Open External. Electron apps can open external apps or files using the system's default apps. This is useful for displaying content outside the app, like opening external links or viewing files. The `openExternal` function can also launch email or calendar apps for user interaction. However, it is important to verify and sanitize values passed to this function to prevent misuse and potential security risks. Inspectron reports on `openExternal` invocations for further analysis. Since calls to this functionality are made within the main process, we cannot automatically trigger them from a Puppeteer script. However, the function is often called when limiting in-app navigation, and we manually evaluate navigation handlers to identify any additional sanitization that they may perform before passing links to the system.

3 Puppeteer Script. Remote debugging can be enabled in Electron by specifying a port number during app launch. This allows using debugging tools like Chrome DevTools to inspect and debug the app from a remote device. We run each app with an instrumented version of Electron and a debugging port, and attach a Puppeteer script for client-side checks.

Preloaded APIs. The instrumented version of Electron collects functionality exposed via the `contextBridge` and makes it available at `window.reportExposedAPIs`. The Puppeteer script accesses this global variable on the client-side and adds it to our framework's report.

Chrome Version. In addition to collecting the underlying libraries in Step 1, the Puppeteer script reports on the Chrome version used by the app by parsing the `navigator.appVersion` object on the client-side.

Network Requests. The Puppeteer script intercepts all network requests performed within the app, and gathers a list of all accessed domains for further analysis.

CSP. Electron recommends that developers set a CSP on each window within their app, as it can greatly reduce the risk of XSS attacks. Developers can set a CSP policy using either a meta tag or HTTP headers. The meta tag approach is the same as with web pages loaded in a browser, i.e., it involves adding a `<meta>` tag to the app's HTML file with the `http-equiv` attribute set to `Content-Security-Policy`. The content attribute of this tag then contains the CSP rules in the form of

a string. Alternatively, developers can use HTTP headers to set the CSP for the app. Electron recommends [19] adding an event listener within the main process, `onHeadersReceived`, to intercept network requests made from the app. Developers may then add or modify response headers to ensure that the `Content-Security-Policy` header includes rules specific to the app. Since Electron provides methods for app developers to modify CSP within HTTP response headers *after* they have been received by the main process, network logging approaches such as proxies and Chromium's `netLog` command-line flag will fail to capture such modifications. However, our Puppeteer script observes network responses after developer modifications, and accurately captures the responses received by the renderer process.

❹ **Additional Checks.** Once the app has been analyzed using the instrumented Electron and Puppeteer script, we perform more checks that do not require running the app.

CSP Evaluation. Inspectron evaluates the CSP captured from the Puppeteer script using Google's CSP Evaluator [20]. The library parses policy rules and recommends ways to harden them, and it includes support for backward compatibility with older versions of CSP.

Permitted Domain Evaluation. While Inspectron gathers a list of domains from network requests and CSP rules, we evaluate these domains in two ways. First, it identifies apps that load solely from packaged, local files instead of gathering remotely loaded content. Next, considering remotely loaded resources, the tool visits these domains using XSStrike [21], and reports on the use of Web App Firewalls and evaluates reachable domains for potential DOM XSS.

Electron Fuse Checks. This is a feature subset that enables developers to dynamically disable default functionality in production apps. Fuses are security flags that determine enabled and disabled features at runtime. Inspectron reports on the use of fuses to determine if apps explicitly enable the encryption of cookies stored on the disk with OS level cryptographic keys. If an app stores cookies in plaintext, malicious access to files from within the app, and other software on the user's system, can read or modify the sensitive information stored in the app's cookies.

❺ **Report.** The results of all evaluations performed in Steps 2, 3, and 4 are stored across multiple files and in varying formats. Inspectron reads and evaluates the results of individual tests and combines them in a single, parsable report. The final report of the analysis highlights scenarios and configurations of relevant checks that require further evaluation of misconfigurations that can result in potential vulnerabilities.

The report generated by Inspectron highlights insecure practices within packaged apps. However, it is important to note that these findings do not verify that the app is entirely exploitable. Rather, they indicate the presence of problematic practices that could potentially be exploited; in certain cases that could involve chaining together multiple insecurities in the context of the app's specific architecture. In the next section we report on how such practices are not isolated incidents, but rather indicative of a broader ecosystem-wide problem.

4 Evaluation

In this section, we present our extensive experimental evaluation of Inspectron, as well as the results from our black-box auditing study of the Electron app ecosystem.

Terminology. We first define the terminology we use.

Insecure practices. Individual checks included in the reports generated by Inspectron highlight known, insecure practices. The report highlights app configurations that Electron warns against and scenarios that have been used in prior exploits [19, 22]. However, these reports represent flaws and do not prove that apps can be exploited in practice.

Exploits. Apps that adopt insecure practices can potentially be exploited when these practices are considered in the context of individual configurations and use cases. Creating proof-of-concept exploits requires manual effort, which we demonstrate for a subset of our findings. Unless stated otherwise, the findings presented in this section highlight insecure practices. Within descriptions of checks for individual insecure practices, we present examples of potential exploits.

False Positives. We consider the incorrect inclusion of insecure practices in reports to be false positives (i.e., reporting an insecure practice when the app actually does something securely). In our test set of 109 apps (see **App Dataset** below), we manually verified every detection and confirmed that none were false positives.

Electron Dataset. A crucial element of Inspectron relies on utilizing an instrumented version of Electron. As mentioned in §3, Electron apps can only be evaluated against an Electron version with a matching app Binary Interface (ABI). Consequently, we developed multiple instrumented versions, each corresponding to a major release version of Electron and the underlying platform, such as Linux or MacOS. To facilitate this process, we employed an existing wrapper [23], which streamlined the synchronization of relevant dependencies, while also enabling the retrieval of specific Electron and Chromium versions for each build of the Electron source code. It is worth noting that even though older versions are accessible, they are no longer actively maintained. As a result, when we encountered difficulties in retrieving all the required files to construct a specific version of Electron, we discontinued the build process. We encountered this challenge when attempting to build Linux-specific Electrons $<$v13, as certain underlying Debian build files were no longer available or accessible. For each version, we obtained the Electron source code, instrumented specific TypeScript and C++ code, and rebuilt the framework. Subsequently, we extracted the resulting distributable for the respective version and platform. Through this process, we successfully created a dataset comprising 24 instrumented versions of Electron

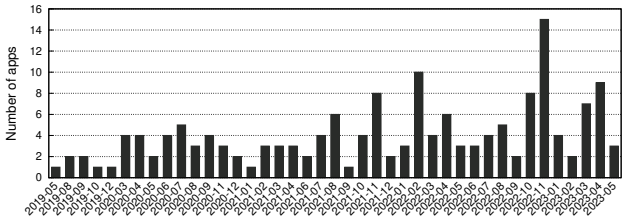

Figure 4: Release date of the Electron binary used by the latest version (as of May 2023) of the apps audited by Inspectron.

(v14-v24 for Linux, v12-v24 for MacOS), which we then used for the analysis described in this section.

App Dataset. The Electron website provides a showcase of apps that have been developed using the framework. This showcase lists a diverse array of apps, including both new and old releases, commercial and free downloads, as well as packaged and open-source projects. Through a manual investigation of 282 apps' pages, and after filtering out apps that were unavailable, discontinued, incompatible with Linux and MacOS, or required some form of payment, we successfully downloaded 167 apps, each of which we further examined to determine their underlying Electron version. The findings of this evaluation are presented below. Subsequently, from this collection, we identified 109 apps for which we had successfully built an equivalent instrumented Electron version to test against, which we use for the remainder of our analysis.

Electron Versions. We conducted a comprehensive evaluation of the underlying Electron framework versions utilized by a total of 167 downloaded apps. Electron regularly releases new stable versions every eight weeks [24]. The latest stable version available at the time of this writing (May 2023) was v24, which had been accessible for a minimum of four weeks before our testing. Furthermore, Electron offers support for up to four stable major versions, implying that apps relying on versions as low as v21 could potentially receive updates if required [25]. However, our findings, shown in Figure 4, reveal that apps depend on Electron versions that are up to four years old, with the majority of them relying on releases between six months and two years prior to our analysis. It is worth noting that Electron strongly recommends that developers keep their apps up to date with the latest release [19], as that ensures the incorporation of numerous security fixes for Chrome, Node.js, and the framework itself. Unfortunately, as we discuss below, our analysis reveals that a significant number of apps use older versions of Electron which remain vulnerable to well-known exploits, even if developers adhere to the best security practices available for the older Electron version they rely upon.

Web Preferences. As aforementioned, each new frame or window within an Electron app possesses a set of preferences that determine the privileges and functionality available to the web content. While Electron advises limiting these preferences [19], our observations unveiled a significant number of apps that explicitly enabled functionalities that could exacerbate compromises on the renderer process (see Table 1). We discovered that the majority of apps ($n = 54$) failed to isolate the context between their preloaded scripts and the renderer process, thereby leaving them vulnerable to exploits like prototype pollution attacks that have been previously documented [26]. Surprisingly, we observed that 8 applications expicitly disabled `web security`, thus disabling the enforcement of the Same Origin Policy (SOP), arguably the most fundamental web security measure. Developer-focused apps like Postman [27] (an API development app) and Altair GraphQL [28] (a GraphQL server debugging app) disable SOP to allow interaction with different endpoints, and include additional measures like CSPs and restricted windows to limit its impact. This also helps apps (e.g., IPTVnator [29], a TV streaming app) easily host content from multiple external services, but still reduces their overall security. More concerning was the discovery that out of the 109 apps analyzed, 49 of them granted the renderer process complete access to Node.js functionality. This configuration allows malicious code within the renderer process to import any Node.js module and directly execute shell commands on the user's system. It is essential to emphasize that these options are disabled by default, indicating that developers deliberately chose to override the app's inherent security measures in favor of enhanced functionality.

Table 1: Insecure Web Preferences detected by Inspectron.

Web Preference	Insecure Value	# Apps
Node Integration	True	49
Context Isolation	False	54
Sandbox	False	64
Web Security	False	8
Allow Running Insecure Content	True	6
Disable Popups	False	64
Enable WebSQL	True	81
Javascript	True	83
WebView Tag	True	15

Reliance on Insecure Defaults. Over time, the Electron framework has made significant strides in enhancing its security measures. Notably, certain preferences such as `nodeIntegration` and `contextIsolation` have been transitioned to secure defaults since v5 and v12 of Electron, respectively. However, it was not until v20 (Aug. 2022) that Electron introduced sandboxing of processes as the default behavior; prior to this, it strongly recommended that developers implement sandboxing. As the majority of apps are built on older versions of Electron, we found that a significant number of developers ($n = 64$) have left their processes unsandboxed, potentially exposing their apps to exploits. We made similar observations regarding developers enabling WebSQL, despite it being a largely-deprecated storage mechanism that is infrequently used in modern browsers [30].

Despite its diminishing relevance, many developers ($n=81$) still enabled WebSQL in their Electron apps. We further found popups being commonly allowed in the renderer processes ($n=64$), effectively permitting the creation of new windows. These observations highlight the prevalence of certain insecure practices that undermine the overall security posture of Electron apps, warranting a closer examination.

Limiting Preferences on WebViews. Each new window or frame possesses its own set of associated preferences; consequently, when a window in an app loads external content in a WebView, the WebView inherits the preferences of its parent by default [31]. However, a malicious WebView has the capability to create new renderer processes with elevated privileges, regardless of its parent, enabling the execution of code on the underlying system. To mitigate potential security risks, Electron recommends that apps actively listen for the new creation of each WebView as it is attached, and explicitly impose limitations on the available preferences. By doing so, developers can exert greater control over the behavior and permissions of WebViews within their apps. However, out of the 15 apps that utilized WebViews, only 4 implemented the recommended practice of listening to the relevant event and enforcing preference limitations.

Command-line Switches. Developers have the option of overriding app-wide defaults and controlling runtime flags that can be passed to Node.js-based and Chromium-based processes. Most apps we evaluated resort to defaults and do not enable experimental, command-line switches. However, we found 3 apps that increased V8's garbage-collected heap size available at runtime [32]. Overriding the heap space taken up by the application and improper garbage collection can affect the system's memory use. We observed 10 apps that disabled default features offered by Chromium.

Cross-Origin-Opener-Policy (COOP). The COOP HTTP response header aims to improve isolation between documents and origins by requesting a new browsing context and process, which can help mitigate exploits like cross-window and process-wide attacks [33]. These types of attacks can occur when a loaded document shares a browsing context and process with cross-origin documents, potentially allowing malicious code to leak data. The COOP header aims to mitigate these issues by allowing loaded resources to sever all references to other browsing contexts, making it easier for browsers to load documents in a new process, preventing attacks like Spectre. Three of the apps that we evaluated, Colibri (a browser) [34], Ferdi (an app-in-app ecosystem) [35], and Biscuit (a browser) [36], explicitly disabled support for this feature, thereby rendering their apps vulnerable.

Out-of-Blink CORS. The Cross-Origin Resource Sharing (CORS) protocol is an established web standard used to safeguard servers against unexpected cross-origin network accesses [37]. Previously, Chrome implemented this protocol within the rendering engine, Blink, which ran in a renderer process. However, the Out-of-Blink CORS feature, enabled by default since Chromium v83, moves the inspection of network accesses out of the renderer, to be handled by a separate process, the network service. This change was motivated by several historical design, reliability, and security issues [37, 38]. However, Advanced Rest Client (a developer tool) [39] explicitly disables this feature.

Navigation Handling. When a user interacts with an Electron app by clicking on third-party links or triggering code execution that modifies the `window.location` or opens new windows, Electron generates events (`will-navigate` and `new-window`) that, if not handled, cause these links to open within the app similarly to browsers. This behavior can be problematic for both functionality and security. To address this, developers need to actively listen to these events and ensure that users remain within the designated app pages. However, our evaluation of 109 apps revealed that only 24 of them implemented navigation limitations. Furthermore, only 32 prevented the loading of arbitrary content in new windows. This situation is concerning, particularly because when pages fail to restrict navigation, the third-party domain loaded within the same window gains access to additional preloaded APIs. This access enables interactions with the main process, which would typically be unavailable within a web browser. Additionally, when new windows are opened, these windows have the ability to create further windows with extended privileges and relaxed security boundaries. Therefore, the lack of proper navigation restrictions and content loading prevention poses a significant risk.

Use of deprecated event handlers. In Electron v22 (Nov. 2022), the `new-window` event was deprecated [40]. Prior versions included warnings about this deprecation, and in newer versions Electron requires developers to handle the creation of new windows using `setWindowOpenHandler()`. However, this has not yet been widely implemented in existing apps. Inspectron found that only 23 apps incorporated handlers using the new approach, while 11 apps continued to rely on the deprecated event, which is compatible with their older versions of Electron. It is crucial that these 11 apps adopt the new approach when they eventually update their version of Electron, to ensure that their checks remain effective.

Inter-process Communication. Electron's renderer process is inherently limited in its privileges. However, apps can utilize IPC calls to delegate privileged execution tasks to the main process. Nonetheless, a compromised renderer process can potentially exploit these IPC channels to trigger malicious functionality. Therefore, it is crucial for developers to implement sender verification mechanisms before executing relevant code based on IPC messages. We discovered that 43 apps established custom IPC channels to their renderer process. We manually verified the handlers used by these applications and determined that only 13 of these apps implemented sender verification. In the remaining 30 apps, a compromised renderer process would have unrestricted access to trigger IPC channels without any checks in place.

Preloaded APIs. We found 19 apps that exposed select additional functionality from the main process to the renderer process, using a context bridge. Of these, 7 apps did not isolate contexts between the preload script and the renderer process. Note that in the absence of context isolation, the renderer process can gain access to Electron internals and Node.js APIs by compromising the preload script. This can be achieved through prototype pollution attacks [16] that override definitions of built-ins like `Array` or `Object` to take control over the execution of the preload script [26].

Custom Protocols. When utilizing custom non-standard protocol handlers for requests that target internal functionality (which may even be registered by third-party libraries like Sentry [41]), developers must consider the associated values when implementing navigation restrictions. We found 36 apps that register custom protocols; upon manual inspection, we found that only 4 of them also take into consideration requests involving custom protocols when determining whether to allow or prevent navigation attempts.

Permission Request Handling. In contrast to Chrome, Electron approves any request made to hardware devices, such as the camera, microphone, and screen. However, developers can add handlers that prompt users for permission and verify the integrity of incoming requests. Inspectron found only 11 apps that handled permission requests, while the rest granted access by default, further highlighting the prevalence of insecure defaults in Electron apps. For example, Wordpress [42], which allows users to manage their websites, should not need the screen-recording and microphone permissions. However, it permits in-app navigation to external domains, which can access the user's device, including camera, microphone, and screen, without prompting the user.

Certificate Verification. While Electron handles the verification of X.509 certificates by default, apps have the option to proceed with network requests despite errors in certificate verification. We found 8 apps that overrode and logged such errors instead of resorting to Electron's default behavior. Upon manual verification we observed that the apps overrode certificate errors only for specific domains.

Open External. The `openExternal` functionality enables developers to open links or files using the operating system's defaults, rather than within the Electron app itself. This feature is particularly useful for handling links or files that should be opened outside the app, such as URLs or local files that require specific apps. However, it is crucial to ensure proper verification and sanitization when utilizing this functionality, as Electron passes the link to a shell command in the underlying operating system (example, `xdg-open` in Linux). During our evaluation, we identified 56 apps that made use of this functionality. We additionally examined the navigation handlers that were previously highlighted. Surprisingly, we discovered that *none* of these handlers perform any additional URL sanitization when passing it to the shell command. Consequently, while this practice prevents external links from opening within the app, it can result in passing malformed or even malicious links directly to the underlying system. To mitigate these risks, it is imperative that developers implement thorough verification and sanitization measures even when these links do not directly concern the app's functionality.

Content Security Policy (CSP). CSP is an important safeguard against cross-site scripting and data injection attacks, as it grants developers control over which resources are allowed to be loaded, thereby reducing the risk of unauthorized or malicious content being executed. In our evaluation of 109 apps, we discovered that only 18 apps had implemented a CSP. However, upon further analysis using Google's CSP Evaluator [20], we found that the CSPs of 16 of these apps returned warnings. Of these, 15 apps included a directive with an attributed severity value of 50, associated with a possible medium severity finding.

Cookie Encryption. Starting from Electron v15 (Sept. 2021), developers can encrypt cookies stored on the user's file system [43]. However, Inspectron found only two apps (Front [44] and Slack [45]) doing this, while all other apps stored cookie values in plaintext. This poses a significant security risk since, in contrast to mobile platforms, these files will be readable to essentially any other process being executed. Upon manual evaluation, we found 66 apps that store sensitive information, including information necessary for authentication/sign-in (e.g., ChatWork [46], an enterprise team chat application, and Wordpress [42]).

Popular apps. We also perform a more in-depth examination of 10 popular Electron apps. First, we download multiple historical versions of each app and report on the frequency of their updates. Next, for each app we consider the latest available version as of May 2023, and employ proof-of-concept exploits affecting V8 and Blink to verify if these bugs have a trickle-down effect on Electron apps. Finally, we augment Inspectron with pre-recorded user interaction traces to increase coverage (see Appendix **??**).

User Interactions. These apps have unique features and capabilities, so we developed a series of custom-tailored user interaction patterns. These include actions such as (a) signing in, (b) opening and closing tabs and windows, (c) engaging with and providing text input (e.g., within messaging interfaces), (d) interacting with and uploading files (e.g., media attachments), and (e) clicking on links within the app. These additional interactions enabled Inspectron to provide more extensive reports. To facilitate further research and analysis, we will make these interactions available upon publication.

Historical Versions. Beginning in September 2021, Electron moved to a new release cadence, with a new stable version released every 8 weeks, following Chrome's Extended Stable release cycles. As a result, Electron keeps up-to-date with alternating Chrome releases [24]. These regular updates are intended to help Electron-based apps stay updated with upstream fixes, including from Chrome, in terms of performance and security. While our larger analysis showed

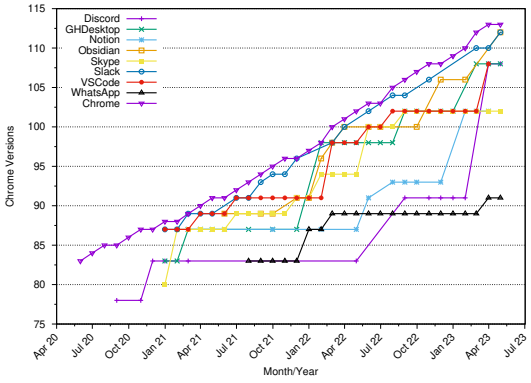

Figure 5: Distribution of Chrome versions of popular apps.

Table 2: Comparative impact analysis of various CVEs.

Apps	Chrome \ CVE	2021-30632 (High)	2022-1364 (High)	2022-3656 (Medium)	DiffCSP [47] –
WordPress.com	89.0.4389.128	✓	✓	✓	✓
Postman	100.0.4896.160	✗	✗	✓	✓
WhatsApp	91.0.4472.164	✗	✓	✓*	✓⁂**
Chrome	113.0.5672.127	✗	✗	✗	✗

✓denotes that the app is vulnerable.
* Implements custom handlers for file drag & drop. We expand upon this in the text.
** Vulnerable to `'javascript:alert()'`. We expand upon this in the text.

that apps rarely use the latest Electron version, this is also the case with widely popular apps. We gathered the release versions of 8 popular apps between August 2020 and May 2023, and matched their underlying Chromium versions.[1] An overview of our findings is presented in Figure 5. Despite Electron's regular releases, these apps are consistently behind the latest version of Chrome. Additionally, each app follows its own release and update cycle, independent of Electron and Chrome. As a result, security fixes and updates remain unfixed and known vulnerabilities remain exploitable for months before apps update to newer versions.

Chrome Version and V8/Blink-based Exploits. Electron apps also depend on Chrome's implementation of V8 and Blink. Chrome regularly receives high-severity exploits of these components, with some attacks even granting remote code access to the user's system [48]. As a result, when bugs are reported in these components, they also affect Electron. While Chrome quickly ships patches, Electron apps can only take advantage of these patches if and when they update to the latest version of Chrome available in Electron. Next, we chose three CVEs with publicly available proof-of-concept exploits. These CVEs make use of vulnerabilities in V8 and Blink and have known usage in Remote Code Execution (RCE) attacks. In Table 2, we show that the latest available versions of Wordpress, Postman, and WhatsApp desktop apps are vulnerable to exploits that are up to 2 years old as of May 2023. First, we identified the underlying version of Chrome that each of these apps relies on. Next, we identified three CVEs that affected the V8 or Blink engines and had proof-of-concept exploits that were publicly available [49–51]. To evaluate each CVE on the app, we opened it and navigated to the DevTools console. We executed the CVE's proof-of-concept code and verified successful execution, i.e., it reported an expected type confusion [50], heap corruption [49], or provided access to sensitive files [51]. Therefore, we confirmed that these bugs also affect the latest version of Electron apps despite being patched in Chrome.

[1] We do not include Wordpress and Postman as we could not find prior versions of the former, and the latter has non-dated release information.

Table 3: Overview of vulnerable components of popular apps.

App	Renderer Process	Chromium	Main Process
Wordpress.com	●	●	◐
Postman	●	●	◐
WhatsApp	◐	●	◐
Notion	◐	●	◐
Obsidian	◐	●	◐
Discord	○	●	◐
Skype	-	●	○
VS Code	◐	●	○
Slack	◐	●	○
GitHub Desktop	◐	●	○

Vulnerabilities across components. Our threat model (§2.1) highlights the risk of chaining multiple vulnerabilities across components for exploiting existing app vulnerabilities. In Table 3, we detail vulnerable components in popular apps. Note that a successfully chained exploit requires compromise and code execution within the renderer process that can then be chained with compromises in Chromium components (V8/Blink) or with vulnerabilities in the Main Process. Below, we discuss examples of insecure practices and present scenarios for exploits which we responsibly disclosed.

Wordpress. The Wordpress desktop app utilizes outdated versions of Chromium, V8, and Blink, which contain bugs that have been targeted in RCE attacks. Furthermore, the app lacks proper restrictions on external navigation; when users click on links they are navigated to these links within the app, allowing the sites accessed through these links to maintain access to JavaScript execution on the renderer process. This flaw becomes particularly critical due to the app's use of older versions of Electron, which do not implement default process sandboxing. Consequently, an attacker can leverage this vulnerability by posting a comment containing a link on a Wordpress blog or sending a message to a Wordpress account. If the victim, who manages their blog using the Wordpress app, clicks on the provided link, they will be unwittingly redirected to a malicious site that can execute arbitrary code.

Postman. This app also similarly relies on outdated versions of Chromium, V8, and Blink. Since this app hosts documentation for public APIs, which often contain external links, it has implemented a protective measure by opening clicked links within a new window that operates in a restricted and sandboxed environment. This setup aims to limit the

reach and impact of third-party content. However, the app's use of older Chromium versions introduces a significant weakness, as known bugs in Blink and V8 can bypass the restrictions imposed by the sandbox. Consequently, despite the attempt to confine the impact of external links, the outdated dependencies increase the risk of successful RCE attacks.

WhatsApp. The latest version (May 2023) incorporates Electron v13 (Chromium v91), which is currently 15 months old. Over this period, several critical security vulnerabilities have been identified and addressed. However, the app still operates within an insecure web environment, lacking context isolation and the utilization of Chromium's sandbox features. Furthermore, it permits the use of a deprecated feature that allows the remote loading of node modules, further compromising its security posture. While WhatsApp restricts users from navigating to third-party websites within the app, it does grant access to https://www.facebook.com, thus relying on the security measures of that particular domain. This exposes the app to any vulnerabilities that may appear on Facebook's website. Moreover, the app employs an insecure CSP that permits the execution of scripts from multiple origins, including potentially vulnerable paths. For example, XSStrike [21] reported at least one vulnerable path under https://maps.googleapis.com. WhatsApp implements custom handlers when a user drags-and-drops a file, which interferes with the proof-of-concept exploit available for CVE 2022-3656 [51]; however, the Chromium version it depends on remains vulnerable. This version is also vulnerable to CSP enforcement bugs found by Wi et al. [47], which erroneously allow the execution of arbitrary javascript code despite limiting such execution using the `script-src-elem` directive.

Notion. Notion is a popular productivity app that is widely utilized by organizations for content management and creation, and collaboration and task coordination among teams. By default, the Notion app follows a security-oriented approach where external links are passed to the host operating system, ensuring that third-party sites cannot be loaded within the app. However, there is a special provision in place that allows the app to allowlist Single Sign-On (SSO) domains associated with user logins, including organizational SSO redirects. This means that if a team configures its employees to access the app using email addresses like *employee@company.com*, which redirects to a designated SSO domain such as *sso.company.com*, Notion permits navigation to that specific SSO domain within the app.

Furthermore, when the app navigates to these allowlisted third-party links, the Notion app retains access to preloaded APIs that trigger unverified IPC calls to the main process. This design decision enables the renderer process to maintain connectivity and functionality with essential features handled by the main process, which does not verify the sender. As a result, these third-party links now possess the capability to pass messages to the main process, allowing for actions such as (1) accessing, modifying, and deleting cookies, and (2) accessing auth-tokens utilized for the app's websocket-based communication with Notion's servers. In the previously stated example, if an organization's SSO redirect were to be compromised, its members would face privacy risks as sensitive information from the team's Notion workspace could be extracted.

Additional evaluation. Here we discuss additional findings; first, we present two new attack vectors that we reported to the developers of Electron.js. Next, we evaluate how Electron apps inherit CSP bugs from Chromium.

Permissions-Policy. This directive offers a way for developers to control access to specific features, including permission to access hardware devices, like the camera and microphone. This can be configured by either setting the `allow` attribute on iframes or by including the directive in the HTTP response header. Electron relies on the underlying Chromium source to enforce the `Permissions-Policy`. Consequently, when the corresponding directives are detected, Electron restricts access to the camera or microphone by restricting calls to `navigator.getUserMedia()` which correctly blocks access. Similarly, when the `Permissions-Policy` directive aims to limit access to the screen by including the `display-capture` directive, Electron imposes restrictions on the use of the `navigator.getDisplayMedia()` function. However, Electron instead exposes access to the display through calls made to `navigator.getUserMedia()` [52] (see §2), which remain unaffected by the `Permissions-Policy` directive. This results in an erroneous implementation that fails to limit access to the screen even when explicitly directed to do so.

X-Frame-Options. Electron implements the `<webview>` tag as an out-of-process iframe (OOPIF). Consequently, it is important to respect the `X-Frame-Options: DENY` header when loading content within the `<webview>` tag. When loading content within a regular `<iframe>` tag, we found that the framework relies on Chrome's implementation of restrictions and prohibits the loading of content that includes an `X-Frame-Options: DENY` header in its response; however, it does not do the same with content loaded within the `<webview>` tag. Allowing cross-site content to load within another frame can potentially result in manipulation of sensitive content within those frames. This problem may be further exacerbated depending on the Electron app's specific implementation of privileges exposed to the webview, including IPC communication and preloaded APIs. We reported this finding to the Electron team and were informed that this is "expected and desired behavior" of the `<webview>` tag: "It bypasses certain traditional restrictions of iframes, includes *[sic]* X-Frame-Options, but also allows more capabilities that would also violate the traditional web security model."

Content Security Policy (CSP) Enforcement. Wi et al. [47] conducted an extensive analysis of how various CSP directives were enforced across different web browsers, and reported six critical bugs to the Chrome browser. We reached out to the authors and accessed the proof-of-concept snippets that they had included in their disclosures. After replicating

Table 4: Number of apps that did not pass each type of check.

Checks	Electronegativity	Inspectron	Intersection
Web Preferences*	29 (6)	**66**	17
Navigation Handling	91 (19)	**75**	59
Command-line Switches	5	**10**	2
Cross-context JS Execution	24	**27**	9
Preloaded APIs	11	**19**	7
Permission Request Handling	97 (1)	**98**	96
Custom Protocols	29	**36**	22
Certificate Verification	**13**	8	5
Open External	**75**	56	52
Content Security Policy	**101** (9)	87	78
Total True Positives	440	**482**	347

* We report on `nodeIntegration`, `contextIsolation`, and `sandbox`.

the issues in Chrome v99 (the version they used), we then evaluated the CSP implementations of the corresponding Electron framework v17.4.11. We discovered that the incorrect enforcements observed in Chrome had trickled down to Electron as well. Consequently, these security flaws also impact any app developed using Electron, thus amplifying the potential risks posed by Electron's reliance on Chrome.

4.1 Comparison to State-of-the-Art

Electronegativity is a state-of-the-art static analysis tool for app developers to assess their Electron apps for potential security concerns [53]. Given a directory that contains an app's code, Electronegativity thoroughly examines HTML, JavaScript, and JSON files, and utilizes an Abstract Syntax Tree (AST) to conduct checks at two distinct levels. First, it performs "atomic" checks that evaluate branches within the codebase to identify potential vulnerabilities. Then, it applies "global" checks that combine atomic checks and discard false positives, before reporting points of concern.

Checks and Capabilities. Despite adopting fundamentally different approaches, both Electronegativity and Inspectron report on certain overlapping attributes of Electron apps. We developed Inspectron with a larger purview of checks and capabilities, in order to provide a more comprehensive assessment of app behavior. Table 5 (Appendix) presents an overview of the differences in the tools' capabilities. Briefly, Inspectron exclusively handles 5 checks that are not considered by Electronegativity. Of the 10 overlapping checks, Inspectron employs additional, in-depth evaluation for 7 factors, which include important aspects missed by Electronegativity. We provide a more detailed comparison in §A.3 (Appendix).

App Evaluation. We compared both tools by generating reports on the same app dataset. We evaluated each app following our *one-touch* approach, i.e., opening the app but not interacting with it. Our setup limited the coverage gained by Inspectron but allowed a comparison on a wider array of apps for comparison. For Electronegativity we explicitly provided the Electron version of the framework instead of relying on Electronegativity's incomplete detection. This way, we ensured that the reports provided by both tools address the same underlying framework version. We find that even without app-exercising user interactions, Inspectron outperforms Electronegativity in identifying and reporting potential vulnerabilities for the majority of common checks, as it conducts a more comprehensive analysis. A comparison of the potential vulnerabilities reported by Inspectron and Electronegativity is presented in Table 4, where (#) indicates false postives (e.g., 29 (6) indicates 23 true positive findings).

Static and Computed Configurations. Electronegativity relies on analyzing multiple files spread across the application's directory and looks for specific nodes and relationships within the constructed AST. Even so, the tool experiences difficulty in correctly gathering configuration values, including *command-line switches* that enable/disable experimental features and *web preferences* on individual windows. Apps declare both checks as JSON objects but include these objects in different locations, e.g., within the app's metadata declared within a `package.json` file, within an environment (`.env`) file, or within code but in an obfuscated manner that is computed at runtime (e.g., setting `nodeIntegration: !0`). Eletronegativity managed to correctly identify insecure preferences in only 23 apps (vs. 66 apps reported by Inspectron) and identified the use of experimental features in 5 apps (vs. 10 apps reported by Inspectron). Our findings indicate that Electronegativity is limited in its ability to parse inter-file relationships and to compute actual configuration values.

Coverage. We used Inspectron to perform a *one-touch* comparison and, as a result, did not trigger functionality specific to each application. As a result, the numbers reported for event-based triggers (e.g., *cross-context JS execution*, *open external*) and window-specific handlers (e.g., *navigation handling*, *preloaded APIs*, *certificate verification*) present a lower bound. Inspectron can report on these checks only after observing their use, which is triggered by interaction. On the other hand, Electronegativity can scan the source code of the entire application and therefore does not face that limitation. As a result, it reports the use of *open external* in 75 apps (vs. 56 apps reported by Inspectron), *certificate verification* bypasses within 13 apps (vs. 8 apps reported by Inspectron), and also the use of *preloaded APIs* on 4 apps for which Inspectron did not open the corresponding window. However, even with its advantage in coverage, Electronegativity missed 18 apps that attempted JS execution in a cross-context manner, and 16 apps that included handlers for in-app navigation, which were detected by Inspectron. Similarly, while Inspectron outperforms Electronegativity in reporting the remaining 7 checks, in the *one-touch* comparison Inspectron misses findings for these checks that are observed by Electronegativity. However, researchers can overcome this limitation by creating scripts to simulate user interaction specific to each application, as we demonstrated with the 10 popular apps (§A.3 in the Appendix).

False Positives. The static analysis approach adopted by Electronegativity is limited in its ability to correctly

determine apps' runtime configurations. We manually evaluated the reports gathered for each application and observed that the practices highlighted by Electronegativity include numerous false positives. While reporting on in-app navigation it does not consider the use of Electron.js's `setWindowOpenHandler`, and includes incorrect reports for 19 apps. Additionally, Electronegativity cannot evaluate CSP values that are set at runtime, (e.g., with network response headers, and within remote content), and therefore incorrectly reports the absence of a CSP in 9 apps. Finally, as highlighted earlier, the tool is limited in its ability to determine computed values, and reports 6 apps as using insecure preferences when the eventual preference set at runtime are actually secure. Inspectron does not suffer from these limitations since it highlights insecure practices only upon observing them at runtime.

5 Discussion and Limitations

Limitations. While Inspectron offers a comprehensive evaluation of packaged apps at runtime, it is important to acknowledge its limitations. First, Inspectron necessitates the use of an equivalent instrumented version of Electron. While the initial engineering investment was nontrivial, the necessary modifications have remained roughly consistent across versions, and we expect that maintaining this patchset should not be burdensome for the Electron project, security researchers, or interested downstream applications. Additionally, if Electron (or security researchers) released an instrumented version for each major version, this would eliminate the burden on individual app developers and streamline the use of Inspectron.

Second, our system encounters challenges when dealing with the unique directory structures and integrity checks implemented by app developers. This includes (1) additional dependent resources being placed outside of the designated *resources* directory, (2) non-standard helper libraries and modified Electron versions being used to build within packaged apps, (3) additional integrity checks hindering the execution of files copied from the *resources* directory in a different environment, and (4) restricting the use of command-line switches at runtime, limiting our ability to connect to the app via the DevTools protocol and test it using a Puppeteer script.

Finally, Inspectron is a dynamic analysis tool, and fully analyzing an app requires UI-based interaction (as we did for 10 popular apps). This additional workload can be offset by developers recording UI traces once for their app, and reusing these traces by integrating them into their automated Inspectron testing. Despite these limitations, our tool surpasses the state-of-the-art in identifying security violations in apps.

Countermeasures and guidelines. Our study has illuminated multiple problematic aspects of the Electron app ecosystem. While Electron has evolved toward more secure default configurations over time, older versions have significant omissions. Moreover, as we found many cases of developers removing protections offered by the default configurations, Electron maintainers should explore strategies for constraining the level of customization possible in security-critical functionality and implementing stricter default policies. This approach can be particularly beneficial for less "security-aware" developers who may not have in-depth knowledge of secure coding practices. Next, even though Electron provides regular updates, our findings indicate that most apps do not keep up with them. As such, it is crucial that app developers ensure that they always rely on the latest version of the Electron framework. However, while enforcing regular updates can guarantee that Chromium and V8 receive the latest security patches, it is important to note that frequent updates can present maintenance challenges (e.g., handling newly added or deprecated features). Nonetheless, until such solutions are explored, developers can integrate Inspectron into their testing pipeline and regularly test if their apps violate secure practices.

Ethics and disclosure. All of our experiments were carried out locally without any interaction or impact on real users; for apps that required authentication we used test accounts. When using XSStrike [21] we only evaluated domains for DOM XSS to report potentially vulnerable objects, and did not adversely affect any domain. Prior to our initial paper submission, we submitted reports to 4 popular apps in June 2023. Between June and November 2023, we performed an additional round of manual verification of our findings across all evaluated apps, and prepared individual reports. For each app, we parsed their website or repository (if available), and identified their stated disclosure procedure (i.e., email, custom portal, GitHub/GitLab issue, or a specific disclosure process within repositories). We submitted reports to an additional 100 apps; we did not submit reports to 4 popular apps (VSCode, Slack, Obsidian, and Discord), since we found that they include additional measures as mitigations against our attack vectors (e.g., prompting users). Additionally, Discord RPC Maker [54] was archived before we could submit a report. We received responses from 43 apps, and 11 apps have deployed corresponding fixes. We received rewards from three apps (Postman, Wordpress, and Cacher), and our disclosure to Altair GraphSQL was evaluated as a "High Severity" CVE by NIST, while GitHub released an advisory based on our report. We also submitted two reports to the Electron framework regarding their implementations of web standards, i.e., *Permissions-Policy* and *X-Frame-Options*.

Availability. We are making our tool's source code and UI traces available, along with an extended version of this paper that further details the implications of our checks [55].

6 Related Work

Inspectron is a novel, automated, dynamic analysis system that evaluates Electron apps. In this section, we discuss prior work that analyzed the web and app ecosystems.

Browser testing. Web browsers have been extensively studied in the past with various frameworks evaluating imple-

mentations of a range of security-relevant features. Singh et al. [56] built a framework for analyzing the usage of browser features in the wild and detecting access-control flaws. De Groef et al. [57] developed a browser that implements precise and flexible information flow controls for web scripts. Schwenk et al. [58] showed that a lack of specification resulted in browsers including varying implementations of the Same-Origin policy. Similarly, Wi et al. [47] found variation in the enforcement of CSP directives across modern browsers. Luo et al. [59] developed a browser-agnostic framework and studied UI vulnerabilities in mobile browsers. Jueckstock and Kapravelos [60] developed VisibleV8, an dynamic analysis framework hosted in V8, that reported property accesses at runtime. Similarly, Sarker et al. [61] developed an instrumented Chromium and used dynamic analysis to identify JS obfuscation through API calls in the wild. Numerous other works have evaluated the implementation of cookies and caching mechanisms [62–68], authentication flows [69–71], and access control and authorization pitfalls [72–75].

Automated app testing. Kals et al. [76] developed a vulnerability scanner that evaluated web apps for various vulnerabilities including SQL injections and Cross-Site Scripting (XSS). Doupé et al. [77] adopted a way to infer the web app's internal state, which was incorporated in their vulnerability evaluation. Duchéne et al. [78, 79] implemented a fuzzing and reverse engineering approach to infer control and data flows for XSS detection. More recently, Eriksson et al. [80] used navigation modeling, traversing, and the tracking of inter-state data dependencies for developing a web app scanner. Drakonakis et al. [81] presented a scanner-agnostic middleware framework that performs black-box evaluation that mediates the scanner's interactions with the web app with the help of an instrumented web browser.

Evaluating Electron. Carettoni [82] presented the static analysis tool, Electronegativity, and covered the state of Electron security, addressing its implications and adoption back in 2017. Krishna et al. [83] presented examples of exploits in popular Electron apps due to insecure web preferences. More recently, Xiao et al. [10] studied Remote Code Execution (RCE) attacks within cross-platform desktop apps. They instrumented the V8 source code on a single version of the Electron framework to identify and defend against cross-context control flow between the renderer and main processes. Their approach covers Electron's IPC communication, which is one of the checks covered by Inspectron. They state that their instrumentation also covers 36 Node.js APIs and 2 native Javascript APIs, i.e., calls to libraries other than those used by the Electron framework – unfortunately, we have not been able to obtain their code to conduct a more comprehensive comparison. While their approach attempts to limit IPC, our work highlights that vulnerabilities within Electron apps can result from numerous components beyond communication channels alone. In addition to checking IPC channels, we additionally report on vulnerabilities resulting from insecure configurations within the main process, fuses and command-line switches that affect the app as a whole, and resources and various web-based practices adopted within the renderer process.

Jin et al. [9] evaluated Electron apps for vulnerabilities resulting from unintended modifications to the DOM-tree. They instrumented Blink to enforce a parallel type-based DOM, analogous to the implementation of the Trusted Types specification [84]. Their approach requires developers to comprehensively evaluate all features of their app against the instrumented Electron so that it learns and builds a type-based DOM tree. While their approach requires significant overhead in participation and effort from developers, a comprehensively-evaluated app could successfully protect it against sanitization-based vulnerabilities. Their defense addresses some potential concerns that we report on, i.e., incorrect handling of new windows and webviews. Nonetheless, Inspectron covers a vast range of additional vulnerabilities, including those resulting from insecure CSPs, misuse of preloaded APIs, and all of the checks covered within the main process. In summary, we perform a more extensive evaluation of numerous security violations and cover a larger threat model beyond the scope of prior work.

Despite the popularity and wide adoption of Electron apps, they have received limited scrutiny from researchers. The general disregard for web standards and good security practices that we have found within this ecosystem is particularly concerning. We hope that our work will incentivize additional research and investigations from the security community.

7 Conclusion

The heterogeneity of execution environments poses a major challenge for software companies that aim to have a presence on different application platforms. As a result, cross-platform apps have become an attractive solution, due to the ability to reuse large parts of their existing web-based application code when creating standalone apps for various platforms. However, as our study reveals, this comes at a significant cost. Using Inspectron, we conducted a black-box auditing of a wide range of Electron apps that differ in terms of functionality, capabilities, and popularity. Our findings reveal a fragmented ecosystem fraught with insecure practices, misconfigurations, and outdated components. Crucially, we find that the entire ecosystem exhibits a significant *regression* in terms of the protections offered to users, as the configurability of the Electron framework has resulted in apps that are vulnerable to attacks that have become obsolete in the web ecosystem due to the security mechanisms already baked into modern browsers. Overall, our research sheds light on the problematic practices of Electron app developers, highlighting the need for more constraints in the configuration of security-relevant functionality and more stringent policies about keeping the core components of Electron apps up-to-date.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This project was supported by the National Science Foundation (CNS-2211574, CNS-2143363). The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF.

References

[1] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex security policy? a longitudinal analysis of deployed content security policies," in *NDSS*, 2020.

[2] J. Chen, J. Jiang, H.-X. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang, "We still don't have secure cross-domain requests: an empirical study of cors." in *USENIX Security*, 2018.

[3] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content security problems? evaluating the effectiveness of content security policy in the wild," in *ACM CCS*, 2016.

[4] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig, "Investigating system operators' perspective on security misconfigurations," in *ACM CCS*, 2018.

[5] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *IEEE/ACM ASE*, 2016.

[6] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *ACM CCS*, 2016.

[7] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *NDSS*, 2017.

[8] P. Salza, F. Palomba, D. Di Nucci, A. De Lucia, and F. Ferrucci, "Third-party libraries in mobile apps: When, how, and why developers update them," *Empirical Software Engineering*, vol. 25, pp. 2341–2377, 2020.

[9] Z. Jin, S. Chen, Y. Chen, H. Duan, J. Chen, and J. Wu, "A security study about electron applications and a programming methodology to tame dom functionalities," in *NDSS*, 2023.

[10] F. Xiao, Z. Yang, J. Allen, G. Yang, G. Williams, and W. Lee, "Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem," in *ACM CCS*, 2022.

[11] "Process Model | Electron," 2023. [Online]. Available: https://electronjs.org/docs/latest/tutorial/process-model

[12] "Blink (Rendering Engine)," 2023. [Online]. Available: https://www.chromium.org/blink/

[13] "Diving Into Electron Web API Permissions · Doyensec's Blog," 2022. [Online]. Available: https://blog.doyensec.com/2022/09/27/electron-api-default-permissions.html

[14] "File System Access," 2023. [Online]. Available: https://wicg.github.io/file-system-access/#privacy-considerations

[15] "Copies of Existing Electron Vulnerability Reports," 2023. [Online]. Available: https://anonymous.4open.science/r/electron-past-bug-reports-33C6

[16] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *ACM ESEC/FSE*, 2021.

[17] "Application Packaging | Electron," 2023. [Online]. Available: https://electronjs.org/docs/latest/tutorial/application-distribution

[18] "Native Node Modules | Electron," 2023. [Online]. Available: https://www.electronjs.org/docs/latest/tutorial/using-native-node-modules

[19] "Security | Electron," 2023. [Online]. Available: https://electronjs.org/docs/latest/tutorial/security

[20] "CSP Evaluator | Google," 2023. [Online]. Available: https://csp-evaluator.withgoogle.com/

[21] "XSStrike | GitHub," 2023. [Online]. Available: https://github.com/s0md3v/XSStrike

[22] "Doyensec | Awesome Electronjs Hacking | Vulnerabilities Write-Ups and Exploits," 2022. [Online]. Available: https://github.com/doyensec/awesome-electronjs-hacking#vulnerabilities-write-ups-and-exploits

[23] "Electron build tools," 2023. [Online]. Available: https://github.com/electron/build-tools

[24] "New Electron Release Cadence," 2021. [Online]. Available: https://www.electronjs.org/blog/8-week-cadence

[25] "Electron Releases," 2023. [Online]. Available: https://www.electronjs.org/docs/latest/tutorial/electron-timelines

[26] L. Carretoni, "Preloading Insecurity In Your Electron," https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Carettoni-Preloading-Insecurity-In-Your-Electron.pdf, 2019.

[27] Postman, Inc., "Postman api platform," 2023. [Online]. Available: https://www.postman.com/

[28] Altair, "Altair graphql client," 2023. [Online]. Available: https://altairgraphql.dev/

[29] IPTVnator, "Cross-platform IPTV player application with multiple features, such as support of m3u and m3u8 playlists, favorites, tv guide, tv archive/catchup and more." 2023. [Online]. Available: https://github.com/4gray/iptvnator

[30] T. Steiner, "Deprecating and removing WebSQL," aug 2022. [Online]. Available: https://developer.chrome.com/blog/deprecating-web-sql/

[31] "desktopCapturer | Electron," 2023. [Online]. Available: https://www.electronjs.org/docs/latest/api/desktop-capturer

[32] T. Junghans, "Node V8 Option max-old-space-size," may 2023. [Online]. Available: https://gist.github.com/tjunghans/90ff3bbf575b8b1da41f3fb56e374931

[33] "Trusted Types," sep 2022. [Online]. Available: https://www.w3.org/TR/trusted-types/

[34] "Colibri: Browse without tabs," may 2023. [Online]. Available: https://colibri.opqr.co/

[35] "Ferdi: All your apps in one place," may 2023. [Online]. Available: https://getferdi.com/

[36] "Biscuit: A browser so your apps donŕ get buried in tabs," may 2023. [Online]. Available: https://chromestatus.com/feature/5768642492891136

[37] "Feature: Out-Of-Renderer Cross-Origin Resource Sharing (aka OOR-CORS or OutOfBlinkCors)," oct 2019. [Online]. Available: https://www.chromium.org/Home/loading/oor-cors/

[38] "OOR-CORS: Out of Renderer CORS," oct 2018. [Online]. Available: https://eatbiscuit.com/

[39] "Advanced REST Client," may 2023. [Online]. Available: https://install.advancedrestclient.com/

[40] "Breaking Changes | Electron," 2023. [Online]. Available: https://www.electronjs.org/docs/latest/breaking-changes#removed-webcontents-new-window-event

[41] "Sentry | Electron," 2023. [Online]. Available: https://docs.sentry.io/platforms/javascript/guides/electron/

[42] Wordpress.com, "Give wordpress a permanent home in your dock," 2023. [Online]. Available: https://apps.wordpress.com/desktop/

[43] P. Krill, "Electron framework adds encryption API," sep 2021. [Online]. Available: https://www.infoworld.com/article/3634383/electron-framework-adds-encryption-api.html

[44] "Front: Stay connected from any device," may 2023. [Online]. Available: https://front.com/download

[45] "Slack: Where work happens," may 2023. [Online]. Available: https://slack.com/

[46] Chatwork, "Group chat for global teams," 2023. [Online]. Available: https://go.chatwork.com/en/

[47] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son, "Diffcsp: Finding browser bugs in content security policy enforcement through differential testing," in *NDSS*, 2023.

[48] "CVE: Seach Results," may 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=v8

[49] "CVE-2021-30632 Detail," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-30632

[50] "CVE-2022-1364 Detail," 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-1364

[51] "2022-3656 Detail," 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-3656

[52] "Web Embeds," 2023. [Online]. Available: https://www.electronjs.org/docs/latest/tutorial/web-embeds

[53] "Doyensec | Electronegativity Official Documentation," 2022. [Online]. Available: https://github.com/doyensec/electronegativity/wiki/Home

[54] ThatOneCalculator, "DiscordRPCMaker: The best way to make and manage custom discord rich presences with buttons," 2023. [Online]. Available: https://github.com/thatonecalculator/discordrpcmaker

[55] "Inspectron Repository," 2024. [Online]. Available: https://github.com/masood/inspectron

[56] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *IEEE Symposium on Security and Privacy*, 2010.

[57] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: A web browser with flexible and precise information flow control," in *ACM CCS*, 2012.

[58] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin policy: Evaluation in modern browsers," in *USENIX Security*, 2017.

[59] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis, "Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers," in *ACM CCS*, 2017.

[60] J. Jueckstock and A. Kapravelos, "Visiblev8: In-browser monitoring of javascript in the wild," in *ACM IMC*, 2019.

[61] S. Sarker, J. Jueckstock, and A. Kapravelos, "Hiding in plain site: Detecting javascript obfuscation through concealed browser api usage," in *ACM IMC*, 2020.

[62] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *ACM CCS*, 2016.

[63] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *ACM CCS*, 2020.

[64] M. M. Ali, B. Chitale, M. Ghasemisharif, C. Kanich, N. Nikiforakis, and J. Polakis, "Navigating Murky Waters: Automated Browser Feature Testing for Uncovering Tracking Vectors," in *NDSS*, 2023.

[65] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk, "Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers," in *ACM CCS*, 2021.

[66] J. Rautenstrauch, G. Pellegrino, and B. Stock, "The leaky web: Automated discovery of cross-site information leaks in browsers and the web," in *IEEE Symposium on Security and Privacy*, 2023.

[67] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: Http cookie hijacking and the exposure of private information," in *IEEE Symposium on Security and Privacy*, 2016.

[68] K. Solomos, J. Kristoff, C. Kanich, and J. Polakis, "Tales of favicons and caches: Persistent tracking in modern browsers," in *NDSS*, 2021.

[69] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, and J. Polakis, "O single Sign-Off, where art thou? an empirical analysis of single Sign-On account hijacking and session management on the web," in *USENIX Security*, 2018.

[70] M. Ghasemisharif, C. Kanich, and J. Polakis, "Towards automated auditing for account and session management flaws in single sign-on deployments," in *IEEE Symposium on Security and Privacy*, 2022.

[71] A. Sudhodanan and A. Paverd, "Pre-hijacked accounts: An empirical study of security failures in user account creation on the web," in *USENIX Security*, 2022.

[72] S. Roth, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, "The security lottery: Measuring Client-Side web security inconsistencies," in *USENIX Security*, 2022.

[73] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks," in *IEEE Symposium on Security and Privacy*, 2011.

[74] S. Calzavara, T. Urban, D. Tatang, M. Steffens, and B. Stock, "Reining in the Web's Inconsistencies with Site Policy," 2021.

[75] S. Karami, P. Ilia, and J. Polakis, "Awakening the web's sleeper agents: Misusing service workers for privacy leakage," in *Network and Distributed System Security Symposium*, 2021.

[76] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: A web vulnerability scanner," in *WWW*, 2006.

[77] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A State-Aware Black-Box web vulnerability scanner," in *USENIX Security*, 2012.

[78] F. Duché, S. Rawat, J.-L. Richier, and R. Groz, "Ligre: Reverse-engineering of control and data flow models for black-box xss detection," in *Working Conference on Reverse Engineering (WCRE)*, 2013.

[79] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *ACM CODASPY*, 2014.

[80] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *IEEE Symposium on Security and Privacy*, 2021.

[81] K. Drakonakis, S. Ioannidis, and J. Polakis, "Rescan: A middleware framework for realistic and robust black-box web application scanning," in *NDSS*, 2023.

[82] L. Carettoni, "Electronegativity - A Study of Electron Security," Las Vegas, NV, USA, Jul. 2017. [Online]. Available: https://infocondb.org/con/black-hat/black-hat-usa-2017/electronegativity-a-study-of-electron-security

[83] M. S. R. Krishna, M. Garrett, A. Purani, and W. Bowling, "ElectroVolt: Pwning Popular Desktop Apps While Uncovering New Attack Surface on Electron," Aug. 2022. [Online]. Available: https://www.youtube.com/watch?v=Tzo8ucHA5xw

[84] "Cross-Origin-Opener-Policy," may 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy

[85] M. Kinugawa, "Discord Desktop app RCE," 2020. [Online]. Available: https://mksben.l0.cm/2020/10/discord-desktop-rce.html

A Appendix

A.1 Web Preferences Examples

This section provides detailed examples showcasing the functionality and restrictions of inheritable preferences and their security implications in Electron apps.

Node Integration. [default: `False`] If set to `True`, any content rendered within the window, including third-party scripts, has direct access to Node modules, and can execute code on the system. Similarly, the *nodeIntergrationInSubframes* and *NodeIntegrationInWorkers* preferences determine if access to exposed Node APIs can be made available to embedded iframes and workers respectively. Electron (>=v5; 2019) sets a secure default for this preference.

Preload scripts. [default: None] This preference lets developers provide a path to *preload scripts* that expose additional functionality to the rendered web content. Improper configuration of these scripts can make the app vulnerable. Electron does not provide a preload script by default.

Context Isolation. [default: `True`] This preference determines whether global variables are shared between *preload scripts* and renderer processes. If they share the same state, a malicious script in the renderer process can perform *prototype pollution* attacks, i.e., it may override API calls on the `window` object, or change the definition of the `Array` data type to bypass checks used within the *preload script*, and gain access to Node.js APIs. Electron (>=v12; 2021) sets a secure default for this preference.

Sandboxing. [default: `True`] Borrowing from Chromium's sandboxing implementation, this option uses the underlying operating system to limit accesses available to the renderer process. This option further reduces the Node modules that can be exposed even to *preload scripts*. The renderer process would instead need to open new *Inter-process Communication* (IPC) channels and send requests to the main process, which can instead interact with the system on behalf of the renderer process. Electron (>=v20; 2022) sets a secure default for this preference.

Other. Electron provides other options that may be set to unsafe defaults. These include *allowpopups*, *disablewebsecurity*, *enableBlinkFeatures*. Each of these options can enable a different type of insecure access, not necessarily enabling remote access to execution on the system. Electron sets secure defaults for all of the above preferences by default.

A.2 Motivating Example

To better understand the types of attacks that can affect cross-platform frameworks like Electron, we discuss a previously-reported vulnerability against Discord. A Remote Code Execution (RCE) attack that was reported in 2020 [85].

Insecure web preferences. In Electron, each new window (or web embed, e.g., iframe) is associated with its renderer process, which is associated with a list of web preferences that determine the level of privilege it can access. Two important options are `nodeIntegration`, which determines whether the renderer process has access to all Node modules, and `contextIsolation`, which determines whether the *preload script* and the web content loaded in the renderer process share the same context. The Discord app disabled context isolation, exposing its renderer process to potential misuse.

Table 5: Capability comparison to Electronegativity.

Capabilities	**Electronegativity**	**Inspectron**
Handle Packaged & Obfuscated Code	○	●
Bypass Integrity-based Restrictions	●	◐
Window-level Reporting Granularity	○	●
Capture Network Requests	○	●
Report Function & Handler Definitions	○	●
Detect Electron Version	◐	●
Checks	**Electronegativity**	**Inspectron**
Web Preferences	●	●
Navigation Handling	◐	●
Inter-process Communication	○	●
Command-line Switches	◐	●
Cross-context JS Execution	◐	●
Preloaded APIs	◐	●
Custom Protocols	◐	●
Permission Request Handling	◐	●
Certificate Verification	●	●
Open External	●	●
Content Security Policy	◐	●
Cookie Encryption	○	●
Chrome/V8 Versions	○	●
Permitted Domain Evaluation	○	●
Fuse Checks	○	●

XSS in loaded contents. The app's CSP, through the `frame-src` directive, allowed third-party content from a list of domains to be loaded within iframes. One of the allowed domains, sketchfab.com, was vulnerable to XSS. If a hosted HTML file included a particular script, it would execute within the iframe in the Discord app.

Navigation handling. A bug in the Electron framework ensured that a *will-navigate* event was not triggered if the top-browsing context was navigated away from a call by an iframe if the top-level frame and the iframe were from different origins. The embedded frame could, therefore, navigate the top window to an attacker-hosted site, leading to RCE.

The RCE attack was the result of combining three separate bugs. While the Discord app had set an insecure web preference (`contextIsolation`), the attack was made possible by external vulnerabilities, i.e., an XSS on a third-party domain (sketchfab.com) and a bug within the Electron framework. The app's CSP did not suffice in preventing the attack, and the navigation restriction bypass bug in the Electron framework itself enabled the app to be successfully exploited. This example illustrates how the unique capabilities and characteristics of a cross-platform framework like Electron can expose users to severe security threats. The Discord app's vulnerabilities showed how even recommended security checks could not prevent an attack when there are external vulnerabilities, highlighting the importance of constantly monitoring and patching an application's security and the need for a comprehensive auditing framework that can guide app developers towards better securing their applications.

A.3 Comparison to Electronegativity

We provide additional details on the checks and capabilities of both tools in Table 5. We elaborate on the differences between the two tools in an extended version of this paper [55].

- **Runtime Behavior.** Electronegativity does not provide insights into the runtime behavior of an app. Instead, it analyzes the code starting from a potential entry point and reports vulnerabilities based on the Abstract Syntax Tree (AST) it manages to create at that point.
- **Packaged and Obfuscated Code.** Navigating packaged apps and obfuscated code poses a challenge for Electronegativity. Automatic detection of entry points and dependencies becomes difficult when they are spread across multiple files or when the code is intentionally obfuscated. Analyzing specific code snippets can be challenging when dealing with minified code. Electronegativity attempts to point to the location of reported vulnerabilities, but the analysis becomes more difficult when code is heavily minified. To improve manageability, it is important to report specific event listeners, handlers, and procedures when they are registered with the Electron framework.
- **Reporting Granularity.** Electronegativity reports potential vulnerabilities at the overall app-level, but it may not specify which specific window or frame of an Electron app is responsible for a particular vulnerability. This information is crucial for effectively identifying and addressing the reported issues.
- **Network Requests.** Electronegativity focuses on analyzing JS and HTML files and does not capture network requests or analyze loaded resources beyond these file types.
- **Electron Version Detection.** Extracting information about the underlying version of Electron used by the app can be limited. While Electronegativity tries to determine the Electron version from the package.json file, this information may not be available in packaged apps and can only be retrieved at runtime.
- **Web Preferences and Command-line Switches.** Electronegativity attempts to capture web preferences and command-line switches from multiple locations, i.e., the `package.json` file in the app's root folder, the app's JavaScript code, and from attributes included in HTML tags. However, code is distributed across multiple files, and these preferences are also computed at runtime, making it difficult to accurately capture the eventual value used by the application.
- **Nagiation Handling.** The tool does not check for the use of Electron's updated `setWindowOpenHandler` and, as a result, incorrectly reports the absence of limitations on navigations within applications.
- **Preloaded APIs.** Electronegativity detects the use of preloaded APIs from HTML tags and from declared web preferences, in a similar manner to its detection of web preferences, and only points to the file location where the configuration was added. However, its analysis does not provide any insight into the functionality that is exposed.
- **Custom Protocols.** Electronegativity additionally reports on whether an application sets a custom protocol. However, the associated underlying protocol, i.e., the use of alternative values for `file:` and `http:` links, is necessary for understanding the implications of the protocol itself, but this is neither detected nor reported by the static analysis tool.
- **Certificate Verification and Open External.** In these overlapping checks, both, Electronegativity and Inspectron attempt to cover similar checks and usage. Given Electronegativity's advantage in terms of code coverage, it detects and reports the use of these provisions in a larger number of instances. Inspectron's reporting of these checks is dependent on the functionality being triggered
- **Additional Checks and Capabilities.** Inspectron performs multiple additional evaluations that are otherwise not considered by Electronegativity. These additional checks are important given their inclusion in recommendations from Electron and use in prior app exploits [15, 19].

Table 6: Number of apps that did not pass each type of check from an evaluation of 10 popular apps. (#) indicates false positives and {#} indicates the intersection with Electronegativity.

Checks	Electronegativity	Inspectron (One-touch)	Inspectron (w/ Interaction)
Web Preferences*	1	6 {1}	6 {1}
Navigation Handling	6 (4)	3 {2}	4 {2}
Command-line Switches	0	2	3
Cross-context JS Execution	1	1	2 {1}
Preloaded APIs	2	4 {2}	6 {2}
Permission Request Handling	6 (1)	5 {5}	5 {5}
Custom Protocols	5	6 {4}	8 {5}
Certificate Verification	2	2 {1}	3 {2}
Open External	6	2 {2}	9 {6}
Content Security Policy	10 (2)	9 {7}	10 {8}
Total True Positives	32	40 {24}	56 {32}

* We report on `nodeIntegration`, `contextIsolation`, and `sandbox`.

Next, compare the reports generated by Inspectron for 10 popular apps for which we developed user interaction (UI) traces, against Inspectron's baseline one-touch approach as well as the reports generated by Electronegativity (see Table 6). While Inspectron with a one-touch approach reports more problematic practices than Electronegativity, it also misses checks captured by the static analysis tool. This is especially highlighted in its detection of `shell.openExternal()`. We made similar observations with reports of insecure CSP, certificate verification, and the use of custom protocols. However, with the UI traces, Inspectron captures checks missed by the one-touch approach and further reports on findings missed by Electronegativity as well. These findings further highlight Inspectron's effectiveness when compared to the state-of-the-art, as well as the performance boost offered by UI traces which increase the coverage obtained by our dynamic analysis tool.