# Lost in Translation: Exploring the Risks of Web-to-Cross-platform Application Migration

Claudio Paloscia
University of Illinois Chicago
cpalos2@uic.edu

Kostas Solomos
University of Illinois Chicago
ksolom6@uic.edu

Mir Masood Ali
University of Illinois Chicago
mali92@uic.edu

Jason Polakis
University of Illinois Chicago
polakis@uic.edu

## Abstract

The cross-platform application-development paradigm alleviates a major challenge of native application development, namely the need to re-implement the codebase for each target platform, and streamlines the deployment of applications to different platforms. Essentially, cross-platform application development relies on migrating web application code and repackaging it as a native application. In other words, code that was designed and developed to execute within the confines of a browser, with all the security checks and safeguards that that entails, is now deployed within a completely different execution environment. In this paper, we explore the inherent security and privacy risks that arise from this migration, due to the fundamental differences between these two execution environments, which we refer to as security *lacunae*.[1] To that end, we establish a differential analysis workflow and develop a set of customized tests designed to uncover divergent behaviors of web code executed within a browser and as an Electron cross-platform application. Guided by the findings from our empirical exploration, we retrofit part of the Web Platform Tests (WPTs) testing suite so as to apply to the Electron framework, and systematically assess mechanisms that relate to *isolation* and *access control*, and critical *security policies* and *headers*. Our research uncovers semantic gaps that exist between the two execution environments, which affect the enforcement of critical security mechanisms, thus exposing users to severe risks. This can lead to privacy issues such as the exposure of sensitive data over unencrypted connections or unregulated third-party access to the local filesystem, and security issues such as the incorrect enforcement of CSP script execution directives. We demonstrate that directly migrating web application code to a cross-platform application, without refactoring the code and implementing additional safeguards to address the conceptual and behavioral mismatches between the two execution environments, can significantly affect the application's security and privacy posture.

[1]We borrow the term *lacunae* from linguistics, where it is used to describe the "semantic gaps that can lead to misunderstanding, misinterpretation of certain concepts in an intercultural context" [35]. While we explore both the privacy and security implications of the lacunae that affect the web-to-cross-platform code migration process, we refer to them as security lacunae for brevity.

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
*Proceedings on Privacy Enhancing Technologies 2025(4), 24–39*
© 2025 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2025-0117

## 1 Introduction

Web services have become an inextricable part of our everyday lives, affecting multiple facets of our personal, professional, and social activities. In desktop environments, browsers have traditionally provided access to the Internet by mediating the vast majority of users' online activities, leading to web applications dominating the software ecosystem, with a wide range of diverse applications being available. This includes browser-based alternatives to applications that had been widely established as native applications, such as messaging and productivity applications. However, supporting such a diverse set of applications necessitates an extensive collection of underlying features, which has resulted in web browsers being in a constant state of evolution, with new APIs being continuously adopted [9, 32, 41]. This has led to our current state of affairs, where modern web browsers have massive and complex codebases (Chromium is comprised of over 33 million lines of code [46]). Naturally, as browsers become more bloated [47], their performance can be negatively affected [44].

Consequentially, in recent years, major web services have been motivated to deploy standalone applications for desktop environments. Previously, creating standalone native applications was a cumbersome and expensive process as vendors would have to create custom-tailored native apps using different programming languages for each target platform (e.g., Swift for macOS, C# for Windows). The cross-platform application-development paradigm aims to alleviate these challenges by allowing vendors to reuse significant chunks of their codebase across different platforms, with popular cross-platform frameworks like Electron leveraging the JavaScript code that also powers those services' web applications. As such, high-profile services like Slack, Signal, and WhatsApp are now offered as cross-platform Electron apps [19].

In more detail, Electron is a framework that enables the development of cross-platform applications using HTML, CSS, and JavaScript, similar to the process of constructing a regular web application. Electron achieves this by integrating Node.js for the backend and Chromium for the frontend. However, due to the inherent differences between Electron and standard browsers, the Electron framework often requires modifications to its dependencies, particularly Chromium, to align with specific requirements [18]. This necessitates a certain level of adaptability, which can pose challenges in keeping pace with the 4-week release cycle of Chromium. As such, Electron has adopted an even-number release strategy, releasing every 8 weeks [23].

In essence, cross-platform apps are powered by JavaScript code that was developed for a specific execution environment (i.e., *sandboxed inside a web browser*) but is then migrated to an entirely different environment (i.e., as a *native app*). Prior work on Electron apps has explored their susceptibility to remote code execution attacks [65], and have also found extensive evidence of misconfigurations and outdated browser engines that introduce significant security risks [5]. In this paper, we explore the inherent *security lacunae* that affect this code migration, due to the fundamental differences in the capabilities and constraints present in these two different execution environments. In other words, we aim to better understand which Electron app and framework capabilities necessitate an intrinsically different design approach due to the mismatch between execution environments. Specifically, we aim to explore the ramifications of web code being executed outside of the confines of a web browser, as native execution on top of the operating system provides direct access to powerful Native APIs. Moreover, given the different components that comprise cross-platform apps, code isolation and sandboxing are particularly important.

To that end, we develop a comprehensive analysis workflow that employs differential testing techniques for uncovering security lacunae that exist when web application code is migrated to a cross-platform application setting. First, we develop a series of customized tests for uncovering semantic gaps and nuanced differences that exist between web and cross-platform apps. Next, we incorporate Web Platform Tests (WPTs) [7], a cross-browser testing suite that provides an extensive collection of code snippets that evaluate specific browser mechanism implementations for assessing whether they adhere to web specifications and are compatible with other implementations. Guided by our empirical findings, we focus on WPTs that assess mechanisms that relate to *isolation* and *access control*, and critical *security policies* and *headers*. Since WPTs are not designed for deployment within cross-platform apps, we design an automated deployment pipeline that retrofits WPTs to the Electron framework. Our system uses a carefully crafted Electron application and a Chrome instance fitted with a custom extension, to conduct a differential testing comparison of sensitive browser mechanism implementations between Chrome and the Electron framework, thereby revealing divergent behaviors.

Our experiments reveal shortcomings in existing web isolation and access control mechanisms when deployed within a cross-platform application, where direct access to the local filesystem and APIs is permitted, and loading local files is commonplace. We find that the use of local files blurs the origins of remotely fetched code, thus undermining the *same origin policy*, arguably the most fundamental web security mechanism. We also find that Electron by default delegates access control for powerful Native APIs to the underlying operating system, which can lead to untrusted remote third-party code having unrestricted access to sensitive data due to the user trusting the app itself, thus introducing major privacy risks. In other words, while browsers require explicit permission through user approval for first-party origins (i.e., opt-in) and prevent third-party access, in cross-platform apps, developers need to restrict certain types of embeddings (e.g., WebViews) and explicitly enforce regulations between local and remote resources. Our assessment of 30 real-world Electron apps reveals how security lacunae manifest in the wild, thereby exposing users to significant risks.

Overall, our study highlights the significant security and privacy risks of migrating web application code to cross-platform applications, and the need for a refactoring process that incorporates additional safeguards and checks to account for the inherent mismatches between these two execution environments. We hope that our research will motivate additional research on the risks posed by the migration of web code to different types of cross-platform applications, and the development of automated tools that can streamline the migration process by incorporating the necessary safeguards to mitigate the threats that we have uncovered.

In summary, our research has the following contributions:

- We provide an empirical analysis of the inherent security and privacy risks of migrating web application code to cross-platform applications, which we term *security lacunae*.
- We introduce an automated testing framework that leverages differential testing for uncovering security lacunae in cross-platform applications.
- We conduct an extensive experimental evaluation of our framework, uncovering significant differences in the enforcement of fundamental security and privacy mechanisms by cross-application frameworks. Our artifacts, including the source code and the testing templates, are available in [1].
- We conduct an empirical analysis of real-world Electron apps that demonstrates the manifestation of security lacunae in practice.

## 2 Background

In this section, we provide the necessary background information on concepts pertinent to our study, including web security standards, the Web Platform Tests testing suite, and the Electron cross-platform application development framework.

### 2.1 Web Security Standards

The web's security model is built upon fundamental principles that govern interactions between websites, users, and the network. A traditional threat model for web applications considers two primary adversaries: *web attackers* and *network attackers* [10]. A web attacker controls a malicious website and exploits vulnerabilities to attack other Web applications. Classic web attack techniques include Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF), which manipulate the browser's execution environment to compromise security policies. A network attacker, on the other hand, also has control over network traffic, allowing them to intercept and modify unencrypted data. The primary countermeasure against network attackers is the use of HTTPS for encrypting communication between the browser and the server to ensure confidentiality and integrity, and complementary mechanisms like HSTS [29]. Browsers implement secure contexts, as defined by the W3C specification [61], to identify web pages that meet minimal security requirements and can safely transmit sensitive information.

The `Same-Origin Policy` (SOP) restricts how different Web pages interact with one another. Under SOP, an origin is defined as a tuple consisting of a scheme (e.g., HTTP or HTTPS), a host (e.g., `www.example.com`), and a port (e.g., 443 for HTTPS). This mechanism ensures that a Web page at `https://malicious.com` cannot access data served by `https://secure.com`. While this fine-grained isolation prevents unauthorized data access, it can be overly

restrictive in certain scenarios. As a result, browsers implement the notion of site-based security, which extends SOP by grouping subdomains under a common effective top-level domain (eTLD+1), based on the Public Suffix List (PSL). For example, `example.com` and `example.net` represent distinct sites, whereas `a.example.com` and `b.example.com` are considered part of the same site. Relaxed security policies based on the same-site model improve usability while isolating distinct entities.

**Cookies.** As a client-side storage mechanism, cookies are structured as name-value pairs and can be set via JavaScript (document.cookie) or by the server using the `Set-Cookie` header in an HTTP response. By default, cookies are accessible to JavaScript and are automatically attached to all subsequent requests sent to the host that issued them. The scope of a cookie can be extended to subdomains using the `Domain` attribute, which allows for cookie sharing across sibling subdomains (e.g., a cookie set by a.example.com with `Domain=example.com` can be accessed by b.example.com). Cookies often store sensitive data, such as session identifiers, and browsers therefore implement various security attributes to mitigate potential attacks.

The `Secure` attribute ensures that a cookie is only transmitted over HTTPS, preventing exposure to network attackers [26, 53]. The `HttpOnly` attribute restricts JavaScript access to the cookie, protecting it from XSS-based exfiltration [11, 16]. Additionally, the `SameSite` attribute helps prevent CSRF by restricting the automatic inclusion of cookies in cross-site requests. Setting the same site attribute as `Strict` prevents all cross-site requests from attaching the cookie, whereas `SameSite=Lax` allows cookies to be attached to top-level navigations initiated via safe HTTP methods (e.g., `GET`). Web developers can also use special prefixes such as `__Secure-` which restricts cookies to secure channels, and `__Host-` which scopes cookies to a specific host.

**Resources.** All resources in a webpage, such as scripts, stylesheets, and iframes, should also be securely loaded. Fetching any resource over an unencrypted channel (HTTP) introduces a mixed content vulnerability, allowing a `network attacker` to modify the response and inject malicious code. The W3C specification [55] classifies mixed content into two categories based on its security implications. `Blockable` mixed content refers to active content, such as scripts and iframes, which pose a high security risk if loaded insecurely. Modern browsers block these resources by default. `Upgradeable` mixed content, on the other hand, refers to passive elements such as images, audio, and video, which are considered lower risk. For these resources, browsers attempt an automatic protocol upgrade, rewriting HTTP URLs to HTTPS and loading them securely. If the upgraded resource is unavailable, browsers do not load them, thereby maintaining the integrity of the secure context.

## 2.2 Web Platform Tests

Web Platform Tests (WPT) are an essential suite of automated, cross-platform tests designed to evaluate conformance to web standards [63]. These tests are collaboratively developed and maintained by browser vendors, standards organizations, and web developers to ensure interoperability and compliance with specifications. WPT covers a broad range of web technologies, including HTML, CSS, JavaScript, network security features, and various APIs.

The test framework operates by running predefined test cases, checking whether implementations adhere to the expected behaviors outlined in web standards. By executing WPT, browser vendors can detect deviations, uncover security vulnerabilities, and verify compliance with evolving security requirements. The WPT suite is used by browser developers, web standards bodies (e.g., W3C, WHATWG), and researchers who analyze security properties of web technologies.

Using WPT for web security validation provides several advantages. It enables automated and reproducible testing of security mechanisms like the Same-Origin Policy, Content Security Policy (CSP), cookie attributes, and mixed content blocking. Furthermore, WPT helps ensure that security policies are consistently enforced across platforms, reducing the risk of fragmentation and unintended security gaps. WPT is a crucial tool for verifying that security features are correctly implemented on new platforms and that regressions do not introduce vulnerabilities.

## 2.3 Electron Framework

Electron is a popular framework that allows developers to create cross-platform desktop applications using web technologies, including HTML, CSS, and JavaScript. Electron applications use a modified version of Chromium's Blink rendering engine and V8 JavaScript engine, and interact with similar web standards as browsers.

**Process Model.** The Electron process model divides the application into two separate contexts. The *Main Process* is the privileged system-side process that manages the application's lifecycle, interacts with the operating system, and controls renderer processes. It has access to Node.js modules and can execute system-level operations. The *Renderer Process* uses Chromium's Blink engine to display web content. Each Electron window or embedded web view runs in its own renderer process. Unlike browsers, renderer processes in Electron lack direct access to Node.js and must communicate with the main process to execute privileged operations.

**Communication.** Electron applications facilitate communication between processes through two primary mechanisms. *Preload Scripts* run in the renderer process and expose customized functions to loaded web contents. *IPC Channels* enable secure message passing between the renderer and main processes, allowing the main process to verify and handle requests that require additional privileges.

**Cookies and Sessions.** Electron handles sessions and cookies similarly to browsers but with key deviations. Unlike browsers, Electron's main process can directly access all cookies stored within the application, modify session cookies, and override expiration policies. Additionally, Electron stores cookies in plaintext on the filesystem by default, unlike Chromium, which encrypts them. However, Electron inherits Chromium's origin partitioning mechanism. As a result, third-party iframes loaded within an Electron application have their own cookie jars, which are isolated from the top-level frame and other third-party iframes.

**Permissions.** Electron uses Chromium's permission mechanisms by retaining web API calls under the `navigator` object. However, the framework also allows developers to handle permissions within the main process. Importantly, it does not inherit Chromium's user-facing permission notification interface. Considering its deviations in implementation, it is important to understand its compliance

with expected behaviors. Unlike browsers that prompt users for permission grants, Electron applications approve all permission requests by default unless explicitly managed by developers. Electron provides developers with control over media device permissions but does not allow fine-grained approvals available within browsers (e.g., specific microphone or camera access must be granted in full).

**Electron Security Testing.** Electron enables desktop applications to leverage web technologies but introduces security complexities due to its integration of system-level access with web-based rendering. While it inherits browser security models such as the Same-Origin Policy and cookie attributes, its architecture diverges from standard browser implementations, creating potential security and privacy inconsistencies. However, unlike traditional browsers, Electron lacks a dedicated testing framework, leaving critical security mechanisms unexplored at scale. To address this gap, we adapt WPT (§3.2) to evaluate Electron's security policies, extending its coverage to Electron-specific embedding contexts, network configurations, and security headers. This enables a systematic comparison between Chrome and Electron, detecting deviations that introduce security risks in Electron-based applications (§4).

## 3 System Overview

This section details our methodology for uncovering fundamental differences in how Electron-based applications conform to specific security and privacy standards compared to web applications executed within browsers, while focusing on key aspects of our testing framework's design and deployment. As illustrated in Figure 1, our approach is comprised of two phases: (i) a manually curated empirical testing phase that identifies critical header inconsistencies, and (ii) a large-scale automated testing phase that retrofits Web Platform Tests (WPT) into our framework for systematically evaluating security policies under both server-hosted and local execution scenarios.

### 3.1 Empirical Testing

To detect improperly enforced security headers, we design targeted tests that assess Electron's internal security mechanisms and commonly used application headers. Specifically, we analyze CORS-related headers, HSTS, X-Content-Type Options, and X-Frame Options, which constitute some of the major components of web application security and for ensuring data privacy. We further extend this scope to include additional headers, ensuring comprehensive coverage of Electron's security model. Unlike browsers, which restrict embedding to iframes, Electron enables embedding through WebViews, WebContents, and nested configurations (e.g., iframe-in-WebView). Our tests assess these mechanisms across multiple configurations to identify security inconsistencies and potential semantic gaps across the two execution environments.

**Test templates.** Each test file encapsulates the core logic for evaluating security headers and is structured as a self-contained HTML page with embedded scripts that trigger expected security behaviors. These tests range from basic API calls verifying header enforcement to complex scenarios involving multi-domain redirections and cross-origin interactions. Additionally, we deploy an automated logging mechanism that records real-time results and determines success or failure. To minimize redundancy, we employ a template-based approach, dynamically injecting scenario-specific parameters instead of generating static test cases for every header variation. This method efficiently evaluates multiple execution contexts without duplicating test logic.

Listing 1 illustrates this approach with a Set-Cookie header test, where the template dynamically assigns a domain, embed type (top-level or iframe), and expected compliance. The resulting script verifies whether the cookie is accessible based on Electron's handling of the `Domain` attribute and logs the result. However, while templating simplifies test generation, it relies on server-side processing to render customized test instances. Since Electron supports local file execution, we introduce additional mechanisms to ensure consistency across execution environments.

*Remote tests.* For tests requiring resource-fetching over the network, we deploy a controlled server to host test files and configure the necessary security headers. In addition to dynamically generating test cases, the server handles header-specific requirements, ensuring accurate policy enforcement. Our setup spans multiple domains and subdomains, allowing a thorough evaluation of cross-site and cross-origin behaviors. To simplify endpoint management, we leverage URL query strings to specify header attributes dynamically, and each test independently requests the relevant endpoint and retrieves the necessary header variations. For example, a request is issued to an endpoint such as `/setcookie?samesite=strict` to evaluate the Set-Cookie header with specific attributes. The server processes the query string, generates a Set-Cookie header with the specified SameSite attribute, and dynamically serves the corresponding test file. This approach reduces redundancy while maintaining flexibility to support a wide range of test scenarios.

*Local tests.* To evaluate Electron's handling of local file execution via the `loadFile` method, we developed a standalone version of each test that operates without a server. Since template-based generation is not applicable in this setup, we rely on pre-generated HTML files that comprehensively cover all test scenarios. However, only the `top-level` frame is loaded locally, containing the test logic, while all additional resources — including libraries, redirected pages, and embedded content — are fetched from our server. This hybrid approach ensures consistency when evaluating external dependencies while preserving a controlled local execution environment.

*Testing process.* Our automated testing approach systematically evaluates security mechanisms in both Electron and Chrome, enabling direct behavioral comparison across execution environments. Each test is executed by loading an HTML test file from a remote server or the local file system. The test interacts with the browser environment by issuing network requests, navigating between pages, and invoking browser APIs to trigger security-relevant behaviors. Upon execution, results are logged and transmitted to a remote server for structured analysis. Each test runs independently, accessing predefined server endpoints or local test files to maintain reproducibility. Expected behaviors are inferred from official specifications and documentation [42, 48]; however, differential testing is necessary for local execution. Differential testing involves executing the same set of security tests in both Chrome and Electron under identical conditions to systematically identify discrepancies in their enforcement of security and privacy policies. Since Electron operates in a hybrid environment that integrates browser functionality with system-level access, it may enforce security headers differently from traditional web browsers. Specifically, locally loaded

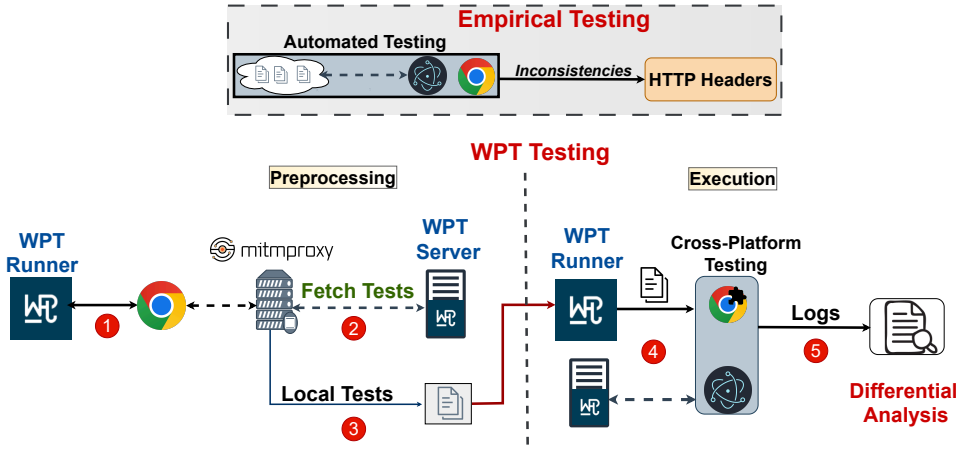


**Figure 1: Overview of our methodology for Differential Header Testing in Chrome and Electron. Our methodology consists of two phases: (i) an `Empirical Testing Phase`, where we manually identify critical headers, and (ii) an `Automated Testing Phase` (`WPT Testing`), where WPT tests are adapted for Electron to enable large-scale automated evaluation.**

**Listing 1: Domain attribute test for the Set-Cookie header.**

```
<script>
  const name = '#{name}',
      domain = '#{domain_value}', visited = '#{visited
    }',embed = '#{embed}', firstParty = '#{firstparty
    }', expected = '#{expected_result}' === 'true';
  const cookieExists = document.cookie.split(
    ';').some(c => c.trim().startsWith(name + '='));
  const resultCode
      = expected === cookieExists ? "1" : "0";
  const reportBody = `Set-
    Cookie, ${embed}, Check Domain: Domain=${domain},
     ` +`Cookie ${cookieExists ? "available" : "not
    available"} `+`for ${visited} -> ${resultCode}`;
  fetch(`https://${firstParty}/report`, {
    method: "POST", headers: {
      "Content-Type": "text/plain" }, body: reportBody
  });
</script>
```

resources are not part of web applications' execution, which creates additional pitfalls in the enforcement of security mechanisms. By comparing results across both environments, we detect deviations in Electron's security model that could introduce vulnerabilities not present in standard web browsers.

**Header inconsistencies.** Our empirical analysis of Electron's security headers highlights inconsistencies in enforcement, particularly in mechanisms governing cross-origin interactions and content isolation. While some security policies align with expected behavior, our preliminary findings indicate inconsistencies in CORS enforcement and embedding restrictions within local execution contexts. These findings corroborate that Electron's security model differs from standard browser enforcement and requires a structured, in-depth, and scalable evaluation to identify semantic gaps and mismatched behaviors. To that end, we extend our methodology to systematically assess these discrepancies by integrating Web Platform Tests (WPT), enabling large-scale analysis and deeper inspection of security policies. Table 1 presents the headers tested in our empirical phase and the subset selected for WPT-based evaluation. For this phase, we prioritized headers based on (i) their role in enforcing web security policies, and (ii) the inconsistencies identified in local execution contexts. By incorporating WPT, we establish a comprehensive framework for evaluating Electron's security policies relative to Chrome's enforcement standards.

## 3.2 WPT Testing

Utilizing WPT for local execution presents architectural challenges due to its reliance on a dedicated server environment. Specifically, it introduces the following constraints, which have to be addressed for identifying the security lacunae that exist between web and cross-platform execution environments.

- `Server Dependency`: WPT requires an active server to resolve dependencies, making direct execution from the local file system infeasible.
- `Unresolved Placeholders`: WPT tests rely on placeholders (e.g., `domain[www]`) which are dynamically replaced by the server with assigned domains. In a local setup, these remain unresolved, leading to test failures.
- `Resource Handling`: WPT tests often compute resources' URLs at runtime. In a local setup, the browser APIs can behave differently and return unexpected values, causing network-fetching APIs to behave inconsistently or fail.
- `Policy Misalignment`: Certain tests rely on network policies such as Same-Origin and Cross-Origin restrictions, which behave differently in a local execution context, affecting test validity.

To address these limitations, we implement a multi-stage approach that reconstructs WPT's execution model while preserving test integrity.

**WPT testing workflow.** We execute WPT tests in Chrome and Electron, running them under default browser conditions, while capturing test requests via a proxy server. We then replay these tests in a modified WPT platform that enables structured local execution.

**Table 1: Headers analyzed in the empirical phase, and the subset selected for more comprehensive WPT-based evaluation.**

| Header | Preliminary Tests | | | | | | WPT |
|---|---|---|---|---|---|---|---|
| | Remote Fetch | | | Local Fetch | | | |
| | top-level | iframe | webview | top-level | iframe | webview | |
| **X-Frame-options** | N/A | ✓ | ✓ | N/A | ✓ | ✓ | ✓ |
| **HTTPS Upgrades** | ✓ | N/A | N/A | N/A | N/A | ✓ | ✓ |
| Access Control Allow Origin | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **Permissions** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Permissions-policy** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| X-Content-Type-Options | ✓ | ✓ | ✓ | N/A | N/A | N/A | |
| Cross-Origin-Embedder-Policy | ✓ | N/A | N/A | N/A | N/A | N/A | |
| Cross-Origin-Opener-Policy | ✓ | ✓ | ✓ | N/A | N/A | N/A | |
| Cross-Origin-Resource-Policy | ✓ | N/A | N/A | N/A | N/A | N/A | |
| Cache-Control | ✓ | ✓ | ✓ | N/A | N/A | N/A | |
| **clear-site-data** | ✓ | ✓ | ✓ | N/A | N/A | N/A | ✓ |
| Expires | ✓ | ✓ | ✓ | N/A | N/A | N/A | |
| Timing-Allow-Origin | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Accept-Charset | ✓ | ✓ | ✓ | N/A | ✓ | ✓ | |
| Access-Control-Allow-Credentials | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Access-Control-Expose-Headers | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Access-Control-Request-Headers | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Access-Control-Request-Method | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Content-Disposition | ✓ | ✓ | ✓ | N/A | ✓ | ✓ | |
| **encoding** | ✓ | ✓ | ✓ | N/A | ✓ | ✓ | ✓ |
| Content-Language | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Content-Type | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **Cookies** | ✓ | ✓ | ✓ | N/A | ✓ | ✓ | ✓ |

**Table 2: Taxonomy of WPT tests incorporated into our analysis pipeline.**

| Category | Headers |
|---|---|
| **Security** | Secure-contexts, Mimesniff, Referrer-Policy, Mixed-content, Permissions-policy, Document-policy, Cookie-deprecation-label, Browsing-topics |
| **APIs & Permissions** | Fullscreen, Geolocation-sensor, Navigation-API, Fullscreen, Geolocation-sensor, Navigation-API, Credential-management, Clipboard-APIs, Permissions-request, Permissions-revoke, Notifications, Idle-detection, Server-timing |
| **Web Components** | Shadow-DOM, Indexed-DB, Service-workers,Beacon, Captured-mouse-events, Payment-method-id, Payment-handler, Encoding, JS, Webauthn, URL, XHR |
| **User Interaction & Media** | Screen-capture, Picture-in-picture, Screen-details, Bluetooth, Remote-playback, Audio-output, PNG, Device-memory, Contacts, Payment-request, Background-sync |

1 **Initial test execution.** A preliminary execution is necessary to store the required test files and modify them for local execution. During this phase, we run WPT under default configurations, triggering multiple tests for each of the intended security headers. Table 2 presents the complete set of WPT-tested headers. While our primary objective is to identify security and privacy risks, we also evaluate a broader set of functional and application-related headers. To ensure consistency, all security headers analyzed in our empirical phase are also assessed using WPT.

2→3 **Extracting test files.** To enable local execution, we introduce a proxy server between the WPT server and the browser. The proxy intercepts responses from the WPT server, which by default processes and replaces all placeholders—such as dynamically assigned domains and ports—with fully-resolved values before serving test files. This behavior ensures that URLs, security policies, and execution contexts are properly structured without requiring additional modifications. The proxy then stores only the top-level frames on the local file system, while all additional resources remain server-hosted to preserve their intended behavior and prevent inconsistencies introduced by local execution.

Storing all resources locally may disrupt execution, as network-dependent tests rely on server-provided headers. For example, in `X-Frame-Options` tests, a top-level frame embeds an iframe that expects to load a resource with the required header. If stored locally, the browser does not enforce the expected security policy for the embedding content, leading to inaccurate results. By maintaining only top-level frames locally and fetching dependent resources from the server, we preserve test integrity.

4 **WPT runner.** WPT assumes predefined domain, subdomain, and port assignments dynamically configured by the WPT server. To support local execution, we introduce a new argument, *--local-files-path*, allowing the WPT runner to load test files from a specified

**Listing 2: URL rewriting example.**

```
Original URL:
file://www1./electron_app_path/cors/304.py?id=1

Transformed URL:
http://www1.web-platform.test:8000/cors/304.py?id=1
```

**Algorithm 1:** URL Rewriting for Local Execution

**Input:** Original test URL
**Output:** Rewritten URL for local execution
$scheme \leftarrow getScheme(URL)$
**if** $schemeIsNotFile(URL)$ **then**
  $scheme \leftarrow getScheme(URL)$
**else**
  $scheme \leftarrow getDefaultScheme()$
**end if**
$port \leftarrow getPort(URL)$
**if** $\neg containsPort(URL)$ **then**
  $port \leftarrow getDefaultPort()$
**end if**
$subdomain \leftarrow getSubdomain(URL)$
$domain \leftarrow getDomain(URL)$
**if** $domainIsLocalPath(URL)$ **then**
  $domain \leftarrow getDefaultDomain()$
**end if**
$path \leftarrow getPath(URL)$
$final\_url \leftarrow scheme + subdomain + domain + port + path$
**return** $final\_url$

directory instead of the server. However, tests may contain hard-coded relative URLs or placeholders that WPT resolves dynamically. When executed locally, these URLs would be relative to the local file system, or contain errors (e.g., subdomain followed by a local path). To address this, we implement request interception mechanisms in both Electron and Chrome, ensuring a reconstructed test environment that aligns with WPT's execution model.

*Electron request handling.* Executing tests in Electron requires intercepting outgoing requests to ensure URLs are structured appropriately and directed to the dedicated server. We develop a lightweight Electron application that processes and rewrites request URLs utilizing the `onBeforeRequest` event of the webRequest module [17], which adjusts the scheme, subdomain, and port according to the test specifications. By default, test files may generate URLs using the `file://` scheme or omit necessary subdomain and port values. To handle this inconsistency, our application automatically rewrites tests defining the `file://` scheme, replacing it with `http://`, and assigning default subdomain and port values when required. Listing 2 illustrates a URL modification example where the appropriate protocol is specified. Our approach ensures compatibility with WPT's execution model while maintaining test integrity by applying these transformations before requests are issued.

*Chrome request modification.* Unlike Electron, which supports network request interception internally, Chrome lacks a native mechanism for modifying outgoing requests. To support this functionality, we develop a Chrome extension that acts as a proxy server capable of intercepting requests. We leverage the `declarativeNetRequest` API [27], enabling structured URL rewriting through predefined rules. Since WPT relies on dynamic placeholders and predefined test domains (e.g., `web-platform.test`) to standardize execution, our extension reconstructs URLs to match WPT's expected structure.

Algorithm 1 outlines our approach for reconstructing URLs by ensuring compliance with WPT's expected domain, subdomain, and port structure. Our Chrome extension processes multiple URL patterns, including:

- URLs that already contain `web-platform.test`, which remain unchanged.
- URLs using the `file://` scheme, either with a specified port or without one.
- URLs using a different scheme, either with a specified port or without one.

Since regular expressions are evaluated sequentially, we prioritize more specific rules before broader ones to ensure correct transformations while minimizing redundant checks.

**5 Differential analysis.** Efficient log analysis is essential for identifying discrepancies in test execution outcomes. To automate this process, we develop a structured compliance analysis tool that processes log files, extracts execution results, and compares behaviors across Chrome and Electron. Our tool employs regular expression-based parsing to extract key details, including test names, subtest descriptions, and failure conditions. To ensure consistency, we uniformly classify failures (e.g., error, fail, timeout), enabling systematic comparisons across multiple test runs. While the analysis is automated, we also conduct manual validation in edge cases that require additional exploration (e.g., multiple failures of the same test). The final compliance report details execution results, highlighting security inconsistencies and deviations from browser enforcement standards.

**Table 3: Evaluated Electron and Chromium versions [22].**

| Electron Version | Chromium Version | Release Date |
|---|---|---|
| 12.0.2 | 89.0.4389.90 | March 2021 |
| 25.2.0 | 114.0.5735.134 | June 2023 |
| 26.2.1 | 116.0.5845.140 | September 2023 |
| 27.0.2 | 118.0.5993.89 | October 2023 |
| 28.1.4 | 120.0.6099.216 | January 2024 |
| 29.1.4 | 122.0.6261.129 | March 2024 |
| 30.0.6 | 124.0.6367.207 | May 2024 |
| 31.3.0 | 126.0.6478.183 | July 2024 |

## 4 Experimental Evaluation

In this section, we leverage our automated testing framework to uncover security lacunae and behavioral divergences that exist when comparing features executed within browser and cross-platform environments. We focus our analysis on security headers, network policies, and origin enforcement to highlight the pitfalls that developers face when migrating their web application code to Electron due to fundamental semantic mismatches.

**Table 4: Security mechanism inconsistencies and corresponding attacks across tested Electron and Chrome versions. ✓ indicates a previously reported vulnerability that has been patched, ✗ denotes a known vulnerability that remains unaddressed, and ★ marks vulnerabilities newly identified by our framework.**

| Security Mechanism Vulnerability | Attack Impact | Affected Versions | State |
|---|---|---|---|
| SOP: Local File Privileges | Access to sensitive user data and credentials. | All | ★ |
| CORS: Local File Bypass | Data exfiltration via arbitrary cross-origin requests. | All | ★ |
| CSP: Local File Enforcement | No restrictions on local file execution or verification. | All | ★ |
| HTTPS: WebView Enforcement | MitM via embedded insecure content. | All | ★ |
| X-Frame-Options: WebView Enforcement | Clickjacking enabling unauthorized interactions and data exfiltration. | All | ✗ |
| Permissions-Policy: WebView Inheritance | Embeddings inheriting sensitive API permissions. | All | ✗ |
| Cookies: Injection of Invalid Characters | Session fixation and cookie jar desynchronization. | All | ✗ |
| Cookies: SameSite Attribute | CSRF and session manipulation. | 12.0.2 | ✓ |

*Experimental setup.* Preliminary tests were performed on Electron 30.0.6, while Web Platform Tests were executed across eight Electron versions and their corresponding Chromium builds, covering a three-year period (2021–2024). The selected Electron versions cover major version updates in the platform, allowing a detailed analysis of both persistent and newly introduced vulnerabilities. A detailed breakdown of the tested Electron and Chromium versions, along with their release dates, is given in Table 3. Regarding our automated testing, we leverage Selenium [6] for automation during the empirical phase, and utilize mitmproxy [14] to proxy requests during WPT testing. The Electron-based WPT deployment was built using commit `d92dadb2` of the WPT repository (pushed on August 16, 2024). Our experiments were conducted on a MacBook Air M1 running macOS Sonoma 14.6.

**Threat Model.** For our analysis, we consider an attacker *embedded* within the renderer process of an Electron application. The attack vector is a third-party script that can be introduced through various common scenarios, such as: (i) a compromised NPM package included at build time [67], (ii) a dynamically loaded remote script, or (iii) a user-driven import of a malicious file through social engineering. This model captures realistic deployment practices in Electron, where external dependencies and remote content are integrated into the app. The attacker can then exploit the security lacunae that we have identified in Electron's design to access local resources, bypass origin-based restrictions, or escalate privileges within the application.

**Results.** Table 4 details the specific versions tested across both platforms, the vulnerabilities detected by our framework, and their current mitigation status. Our analysis reveals that Electron diverges from standard browser security models, particularly in the enforcement of security mechanisms within Electron-specific components (e.g., local files and WebViews) that do not directly map to features available during browser-based execution of web code. These deviations introduce security and privacy vulnerabilities, exposing sensitive user data to unauthorized access. Notably, even when vulnerabilities are addressed in newer versions, their impact may often persist due to the incomplete or inconsistent adoption of updated Electron dependencies across applications [5]. Below, we detail the nature of these divergences, how attackers can exploit these flaws, and their security implications for affected applications.

**Same Origin Policy.** To prevent unauthorized cross-origin resource access, Chrome enforces the Same-Origin Policy (SOP) by assigning a `null` origin to local files. As a result, web pages cannot programmatically load or execute local files. Any access requires explicit user permission, such as selecting a file through the browser's interface. This restriction aligns with the web security model, which prevents web pages from arbitrarily interacting with local files. However, Electron deviates from this model by treating *all* local files as originating from the *same origin*. As a result, any script executed from a local file—including an attacker-controlled HTML file—can access other local files and system resources without restriction.

We demonstrate the security and privacy implications of this flaw through a practical attack scenario. Specifically, an Electron application that loads a malicious local HTML file (e.g., through a third-party script) will treat the malicious file as part of the application's origin. This attack requires no user interaction once the Electron application is launched. If an attacker—whether an invasive or compromised third party running within the application— can store a malicious file in the local file system through automated download, the application will execute it as a trusted resource. Consequently, the injected file inherits the application's privileges, allowing it to access and manipulate local resources, sensitive user files, and stored credentials. Electron also permits automatic FTP connections [8], a feature that attackers can exploit to retrieve and execute additional malicious files — an attack vector explicitly blocked in modern browsers.

Listing 3 presents an iframe-based injection attack that initiates an FTP connection, retrieves a malicious HTML file, and embeds it within an iframe inside the Electron application. Once loaded, the iframe executes arbitrary JavaScript, gaining full access to local resources within the Electron context. While our attack leverages iframes, prior work [5] has shown that Electron applications commonly permit WebViews to enhance functionality. If an application allows WebViews, an attacker could achieve the same attack by injecting a WebView instead of an iframe, thereby extending their ability to execute arbitrary local files and escalate privileges.

*Attack demonstration.* To evaluate the feasibility of this attack across operating systems, we conducted experiments on both macOS and Windows. On Windows, the attack is stealthy as Electron permits FTP access without triggering system-level notifications, thus allowing an attacker to establish a connection and retrieve additional payloads without user awareness. On macOS, the effectiveness of the attack is constrained by its reduced stealthiness,

**Listing 3: Fetching and loading a malicious HTML file**

```
<script>
   function fetchMaliciousFile(){
      window
     .location.href = "ftp://user:psw@ftp.server";
      setTimeout(() => {

     const iframe = document.createElement("iframe");
         iframe.src
      = "file:///Volumes/ftp.server/malicious.html";
         document.body.appendChild(iframe);
      },
      5000);
   }
   fetchMaliciousFile();
</script>
```

as initiating an FTP connection through Electron prompts a system notification requiring explicit user approval. This mechanism limits the attacker's capabilities, as users must manually authorize the FTP request before the connection is established. While this restriction introduces a limitation on macOS, the attack remains highly effective due to Windows' broad adoption and the absence of similar security prompts, ensuring a large attack surface across Electron applications deployed on Windows systems.

This security flaw has severe implications for applications that embed user-generated content or rely on third-party dependencies (which is typical of any modern application). If a local file is loaded within the application, it can escalate privileges by reading other local files, and more critically, it can use JavaScript (e.g., `fetch()`) to access sensitive system files. Specifically, an attacker can determine the logged-in username by leveraging system log files and application data that store absolute file paths and access additional users-specific credentials and resources (e.g., browser cookies, SSH keys). In essence, there is a fundamental difference between how local files are expected to be used by a native application and a web application code running on the user's device. Accordingly, Electron's approach to treating local files as the same origin bypasses critical security policies enforced in modern browsers, exposing applications to unauthorized file access.

**Cross-Origin Resource Sharing.** CORS prevents unauthorized cross-origin requests in web browsers by restricting access to specific HTTP methods, headers, and credential transmissions unless the server explicitly permits them. While Electron effectively enforces CORS for network-fetched resources, it deviates from standard browser security policies when handling local HTML files. Our framework found that cross-origin requests from local files failed in Chrome, which is expected, due to the absence of an `Access-Control-Allow-Origin` header. In contrast, Electron permits these requests, demonstrating that local execution contexts are exempt from CORS restrictions. Specifically, Electron allows local files to issue unrestricted cross-origin requests, further highlighting inherent mismatches that exist between web and native execution environments, and the risks of direct code migration.

This deviation significantly expands the attack surface of Electron applications, as attackers may load a local HTML file and issue unauthorized requests to any origin. Applications utilizing WebViews under insecure configurations (e.g., `nodeIntegration: true`, which is common [5]) are susceptible to such exploitation. Under these conditions, attackers bypass security restrictions without relying on additional application-specific vulnerabilities, enabling unauthorized data access or cross-origin interactions that would otherwise be restricted in traditional browser environments.

**Content Security Policy.** Our empirical evaluation of Content Security Policy (CSP) enforcement in Electron reveals that CSP does not regulate local files and does not apply restrictions to them. While CSP is a critical web security mechanism designed to mitigate script injection attacks and restrict unauthorized resource loading in web contexts (`https://`), it lacks directives for controlling local file execution. Browsers do not provide a CSP directive for `file://` URLs, and Electron's CSP enforcement does not extend to local files. Even if developers define a strict CSP policy for an Electron application, it applies only to content served from remote origins, leaving local files unrestricted.

Additionally, Electron does not inherit browsers' security restrictions on file URLs, which typically block such behavior. As a result, no policy-based mechanism prevents local files from executing scripts or loading resources. This design flaw allows an attacker-controlled local file to execute arbitrary JavaScript and load untrusted local resources without CSP-based restrictions. This design limitation weakens Electron's security posture, exposing applications to script execution risks and resource injection when handling untrusted local content.

**HTTPS enforcement.** Modern browsers enforce secure-by-default policies to prevent insecure connections by automatically upgrading HTTP requests to HTTPS [33]. In contrast, Electron does not implement automatic HTTPS upgrades and strictly follows the protocol specified in the developer's `loadURL()` call. As a result, an application that loads `http://example.com` is vulnerable to man-in-the-middle (MitM) attacks and session hijacking [54, 56]. While Electron supports `preload` scripts for exposing additional functions in the renderer process, it does not natively enforce HSTS, requiring developers to implement HTTPS enforcement explicitly through a preload script [21].

Furthermore, our analysis revealed that Electron WebViews do not enforce Mixed Content Blocking and allow unprotected HTTP content within secure environments. This behavior introduces a practical attack vector in Electron applications, particularly for Electron applications that embed user-generated content (e.g., content management systems such as WordPress). For instance, an attacker may exploit an Electron application with relaxed WebView configurations by dynamically injecting an `<iframe>` targeting an HTTP page. Since Electron lacks both Mixed Content Blocking and automatic HTTPS upgrades, the WebView will successfully render the insecure resource without triggering security warnings or user notifications. Applications that rely on Chromium's secure defaults may unintentionally introduce persistent downgrade vulnerabilities, exposing users to attacks that are mitigated in traditional browser-based environments.

**Permissions Policy**. Electron lacks native permission prompts, necessitating that developers explicitly enforce permissions through the dedicated APIs (i.e., `setPermissionRequestHandler`). In the absence of these handlers, Electron defaults to indiscriminately granting all permission requests, thereby rendering applications

susceptible to security risks by permitting unrestricted access to sensitive APIs. While the Permissions-Policy header is enforced, its applicability is restricted to cases where the top-level document is served only from a remote origin, leaving local files unregulated. This poses a major pitfall for migrated web application code that does not explicitly account for the enforcement deviation.

Additionally, Electron applications rely on operating system-level permissions, which apply globally to the entire application rather than being scoped to individual web origins. Consequently, once a user grants access to resources such as the camera or filesystem, all embedded content and WebViews within the application automatically inherit these privileges. This absence of origin-based isolation introduces a significant attack surface, allowing a malicious embedded page to leverage the granted permissions and gain unauthorized access to user data and system resources. Filesystem access can have severe implications (e.g., the ability to access encryption keys stored in plaintext, as was the case with Signal [2]).

**X-Frame-Options**. Chrome enforces the `X-Frame-Options` header to restrict remote content-embedding in iframes, effectively mitigating clickjacking attacks. In contrast, Electron does not apply the header to `WebViews` and `WebContentViews`, which embed external web content within applications. Unlike iframes, WebViews operate in separate processes, inherit Chromium's rendering capabilities, and bypass security policies designed for embedded content. If a webserver specifies `X-Frame-Options: DENY`, standard iframes automatically block external content loading. However, the same restriction is not enforced within WebViews, allowing attackers to inject a WebView into an Electron application to execute clickjacking attacks. This exploitation leverages techniques similar to traditional web-based attacks but omits existing browser-imposed protections. Moreover, Electron applications typically embed third-party content from remote resources, increasing the likelihood of untrusted sources being loaded within WebViews. This divergence in webview protection is not an unintended security flaw, since prior work and disclosures [5] confirm that Electron explicitly bypasses `X-Frame-Options` enforcement to provide developers with greater flexibility when embedding content. However, this trade-off weakens security guarantees, as developers migrating their web application code to Electron may assume browser-like protections, thus exposing their users to additional risks.

**Cookies**. Both Chrome and Electron permit the setting of cookies with non-printable characters exceeding ASCII code 127, despite official specifications restricting cookie names to US-ASCII characters and excluding control and separator characters. While these malformed cookies are successfully stored, `document.cookie` returns an empty string. This discrepancy results in cookie jar desynchronization, a flaw previously demonstrated against Firefox [54] and also reported in Chrome [57]. An attacker controlling a subdomain (e.g., `attacker.example.com`) can exploit this behavior by injecting a malformed cookie for `example.com`, resulting in Electron applications rejecting legitimate session cookies while accepting the attacker's crafted value. Consequently, the attacker may perform a session fixation attack by pre-setting a known session identifier, forcing the victim into authenticating within the attacker's controlled session.

Additionally, our analysis detected that Electron version 12 fails to enforce the `SameSite` attribute, thereby allowing cross-site cookie leakage. This deviation from expected behavior exacerbates the risk of cross-site request forgery (CSRF) attacks, as cookies intended to be restricted to same-site requests may instead be included in cross-origin requests, leading to unintended state modifications within an Electron application (e.g., session termination) [34].

**Table 5: Deployment characteristics of 30 Electron applications, including origin-to-inclusion relationships, WebView usage, and Content Security Policy configuration. *Local* denotes `file://` origins while *Remote* denotes `https://` origins.**

| Deployment Property | Applications |
|---|---|
| **Application Origin → Inclusions** | |
| – Local → Local | 17 |
| – Local → Remote | 8 |
| – Remote → Remote | 5 |
| **WebView Usage** | 5 |
| – WebView embedding local file | 3 |
| **Content Security Policy** | 12 |

**Evaluating Real-World Applications**. We conduct an empirical analysis of 30 Electron applications to evaluate the real-world manifestation of the security lacunae presented in our work. We randomly select a diverse set of applications from the official Electron showcase [19], varying in popularity and functionality. Following the methodology proposed by Ali et al. [5], we launch each application in debugging mode and use Puppeteer [13] to extract their runtime resources. Our analysis focuses on: (i) local file inclusions (e.g., iframes and scripts), (ii) local file deployment within WebViews, and (iii) the deployment of Content Security Policy (CSP). This approach focuses on extracting resources and configuration patterns to identify architectural and deployment *risks*. However, the security assessment of dynamic behaviors—such as runtime permission requests, API-level access, and WebView execution flows—requires additional techniques based on dynamic instrumentation and execution monitoring. Prior systems (e.g., Inspectron [5]) demonstrate the challenges of conducting dynamic analysis within Electron's hybrid architecture. While this dimension is outside the scope of our work, our findings provide a foundation for a more granular security assessment of Electron applications. A detailed breakdown of applications' configurations is provided in Appendix A.

Table 5 presents aggregated deployment characteristics across the analyzed Electron applications. In total, 25 applications assign a local origin to their main renderer by loading it from a `file://` URL, a typical deployment pattern in which core logic is served through a bundled HTML file. Notably, 8 of these applications fetch remote resources or inclusions, thereby effectively *blending* native and web application behaviors. Among these, 3 applications incorporate third-party resources that provide analytics, consent management, advertising, and federated authentication (i.e., Google SSO). Furthermore, 2 applications embed remote content via iframes; one is sourced from a third-party consent management provider (i.e., Cookiebot [15]), while the other originates from a first-party service. These inclusions introduce external resources into the local execution context, increasing the application's exposure to external injection origins. Even though this hybrid deployment is widespread,

it is inherently insecure as it violates origin isolation. For instance, `Figma`, a popular collaborative design platform, loads its main interface from a local file while retrieving updates and resources from first- and third-party remote servers. In the absence of strict enforcement mechanisms, this configuration bypasses browser security controls such as the Same-Origin Policy, allowing unauthorized access to local resources.

Furthermore, 5 applications deployed at least one WebView to support additional functionality or navigation capabilities. In 3 of these cases, our analysis detected WebViews that load content directly from the file system, a configuration that bypasses multiple browser-enforced security mechanisms, as we have shown. For instance, the browser application `Biscuit` permits loading local files from its main interface. An attacker may exploit this behavior by delivering a crafted `file://` link to the user via an in-app message, email, or external messaging platform (e.g., Slack). Once the file is loaded within the application, it is rendered in a WebView, where it is treated as trusted and inherits the application's privileges. This case is a textbook example of the attack vectors identified through our evaluation and highlights the real-world risks of insecure WebViews.

Finally, 12 applications deployed a CSP, offering a baseline level of protection. Despite including the `'self'` value in their directives, or defining separate CSPs for local and remote components, these deployments remain insufficient to control *all* origins within the application. This limitation reflects the fundamental inability of CSP to regulate local content when files are assigned default origins. One application (`Cacher`) defined all directives using the unsupported `file:` scheme, an invalid expression which is ignored during CSP enforcement. This misconfiguration illustrates the challenges developers face when applying web-based control in native contexts. In contrast, the `VS Code` application, a widely popular developer platform, mitigates local file vulnerabilities by assigning a custom scheme (`vscode://`) to its internal resources and deploying its CSP accordingly. While practical and effective, this approach further highlights the additional complexity and effort required to enforce security constraints in hybrid environments.

Overall, our findings provide empirical evidence that Electron's architecture introduces persistent security risks, thereby corroborating our motivation to evaluate the framework design. Our analysis demonstrates that the platform's default behavior is insecure by design, and that mitigating these risks requires architectural changes and introducing additional application-specific safeguards and secure practices.

## 5 Discussion

**Disclosures.** We submitted a report to the developers of the `Cacher` application, detailing their misconfigured Content Security Policy and the ineffectiveness of the `file:` scheme. We also highlighted the limitations of Electron's security model and included deployment recommendations based on our analysis to support secure CSP configuration. Furthermore, to enhance Electron's security, we reported our findings to the Electron deployment team, providing a detailed technical report and proof-of-concept demonstrations. The team acknowledged our submission and clarified that the identified behaviors align with Electron's intended security model. Similar to the issues reported by prior studies [5], our findings were not treated as vulnerabilities but as behavior consistent with design decisions. Specifically, the team noted that the handling of the `file://` protocol is intentional and documented, and that developers are expected to avoid such deployments or explicitly define their own security models. Nonetheless, we want to emphasize that we believe that Electron's decision to deviate from the norms and behaviors established by browsers creates significant pitfalls for developers. We hope our disclosure will incentivize internal discussions to align Electron's architecture with web security standards and adopt stricter default protections.

**Pitfalls of migration.** Our study sheds light on the inherent pitfalls that afflict web application code migrated to a cross-application Electron environment. Apart from divergent behaviors for specific security mechanisms (e.g., not upgrading to encrypted connections, not blocking mixed content, etc.), more importantly, we have uncovered how cross-platform applications present a *fundamentally different* execution environment. Capabilities that are available to the code powering native applications (e.g., loading local scripts, having direct access to the filesystem or native APIs), are either prohibited, heavily constrained, or occur under specific user actions (e.g., a user *explicitly* opening a local HTML file in a browser). Consequently, those capabilities can either result in corner cases not handled by existing security mechanisms or require specific additional safeguards for protecting migrated web application code (e.g., additional code for preventing third-party code from accessing the camera).

**Countermeasures and guidelines.** Our findings reveal that Electron's security model diverges significantly from modern browser security standards, thus rendering migrated web application code inherently vulnerable by default.

*Enforcing Secure Permission Handling.* Electron's permissive handling of local file execution supports offline functionality but also introduces additional risks, allowing attackers to bypass standard web security restrictions. Given Electron's incomplete security model, developers should proactively enforce security controls to mitigate these classes of vulnerabilities. For instance, permission handlers should be defined to restrict access to sensitive APIs, ensuring that features like the camera and microphone are inaccessible without explicit user consent.

*Restricting Local File Execution.* WebViews should not execute local files unless explicitly required for their functionality. If local file execution is necessary, additional security layers should be enforced beyond process isolation. By default, local files should be treated as untrusted, and their execution should be eliminated. Applications may use specific APIs (e.g.,`protocol.registerFileProtocol`) ensuring that only authorized files are accessible while preventing unrestricted filesystem access [20]. Additionally, local file execution should be constrained within a controlled environment where a preload script enforces strict content validation and origin restrictions. As we detail in §4, `VS Code` application adopts an effective mitigation strategy by disabling `file://` usage and registering a custom scheme for local files. This approach offers a practical solution that Electron applications should adopt to enhance their deployment security.

*Adopting Electron-dedicated CSP.* Deploying CSP in Electron applications enhances security by restricting resource access and mitigating common attack vectors. However, as CSP is adopted from the

**Listing 4: Electron-based CSP example deployment.**

```
script-src  google.com;
frame-src analytics.com;
file-src 'self' path-/application/updates/resources/;
connect-src server.com
```

browser's security model without modifications, it does not support `file://` URLs. This design prevents developers from utilizing CSP to regulate local file execution. To address this limitation, we propose the `file-src` directive, a CSP directive specifically designed for hybrid applications. Since CSP controls resource execution policies, extending it to regulate local files ensures a uniform enforcement model rather than relying on separate security mechanisms.

Similar to the `script-src` directive, `file-src` provides fine-grained restrictions for local resource execution while adapting to the unique characteristics of native execution. The directive defines a set of values that determine how local files may be loaded and executed within an Electron application. Specifically, it supports the following keywords:

- `none`: Blocks execution of all local files.
- `self`: Restricts execution to files located within the application's system-protected installation directory (e.g., on Windows the `C:\Program Files\directory`).
- `sha256-`: Restricts execution to files whose SHA-256 hash matches a predefined encoded value.
- `nonce-`: Allows execution of dynamically loaded files only when they include a valid base64-encoded nonce.
- `path-`: Limits execution to specific absolute paths designated by the developer.

Listing 4 illustrates the application of `file-src` within a restrictive CSP policy. In this example, `file-src` permits execution of files from the application's directory (`self`) and a specified developer-defined external path. Notably, the directive enforces restrictions on local file execution without modifying or conflicting with existing CSP directives, ensuring that policies for web-based and local execution remain independent.

The `file-src` directive provides granular control over local file execution, mitigating the arbitrary execution risks and SOP bypasses introduced in §4. Specifically, the `self` value limits execution to application-controlled directories, ensuring that only system-protected installation paths are permitted. Cryptographic enforcement mechanisms, such as hash- and nonce-based restrictions, preserve file integrity by preventing unauthorized modifications. Additionally, the `path-` restriction further mitigates risks by blocking execution from user-writable locations (e.g., the Downloads folder).

As with web-based CSP directives, combining multiple `file-src` keywords enhances protection by enforcing strict execution control and integrity verification. This explicit access control mechanism reduces the attack surface for origin violations and filesystem-based exploits in Electron applications. Building on our design, we plan to propose the adoption of the `file-src` directive as an Electron-specific security mechanism. By evaluating the new directive in practice, we aim to assess its effectiveness in real-world applications and gather insights for its potential standardization within cross-platform CSP.

*Security Enforcement Challenges.* Given Electron's design and architecture, developers may assume that the platform enforces protections consistent with browser environments. However, this assumption does not hold in practice due to fundamental architectural divergences and the absence of internal mechanisms for detecting security inconsistencies or alerting developers to misconfigurations. For example, protections inherently enforced by browsers, such as Same-Origin Policy, CORS, and permission prompts, must be manually configured in Electron. Moreover, the hybrid environment redefines the concept of an origin, allowing local and remote resources to coexist within the same application context, complicating security enforcement. Prior studies have also shown that developers widely adopt default configurations without assessing their security implications [3, 28, 39, 45, 50, 58]. Ultimately, shifting the responsibility of managing conceptual mismatches, enforcement gaps, and deployment safeguards onto developers exposes users to significant risks. To address these limitations, our work can assist developers in aligning their deployments with modern web security models. Until stronger defaults are adopted, we view the development of automated code retrofitting frameworks that streamline the migration of web application code to Electron and address the inherent security lacunae we uncovered, as a promising future direction.

## 6 Related Work

**Web Standards Adoption.** The security of users on the web is highly dependent on the adoption and correct implementation of constantly evolving standards. In 2010, Zhou and Evans [66] found that even though HTTP-only cookies (introduced in 2002) could be easily deployed to protect sites against cookie stealing attacks, most major websites did not use them. Weichselbaum et al. [59] conducted a large-scale study of CSP adoption and identified significant flaws that can lead to bypasses in 94.72% of all adopted policies. In 2023, Weissbacher et al. [60] found limited CSP adoption, with only 1% of the Alexa Top 100 websites enforcing their CSPs. Kranch et al. [37] evaluated the adoption of HSTS and reported that Chrome's preload list in 2015 contained only 19 of the Top 1K non-Google domains, and numerous websites in the list returned 404 errors or redirected to a plain-HTTP URL. Luo et al. [40] performed a longitudinal analysis of mobile browsers and reported multi-year windows between popular websites requesting a security standard and mobile browsers supporting the mechanism. In 2016, Felt et al. [24] showed positive trends in HTTPS adoption, highlighting that HTTPS traffic doubled as a percentage of all web traffic from 2014 to 2017. They showed that the increased adoption is a direct consequence of support from client platforms (i.e., browsers) and server frameworks.
**Cross-platform Security Mechanism Implementations.** While adopting new security mechanisms is a crucial step, it is equally important to ensure their consistent implementation across different platforms (e.g., browsers). Schwenk et al. [51] analyzed 544 test cases of Same-Origin Policy (SOP) implementations across 10 major browsers and found varying behaviors in 23% of these cases, calling for clarity via formal specification. Singh et al. [52] performed a principle-driven analysis of incoherencies across browsers and reported numerous inconsistencies in the implementation of access control policies around shared resources (e.g., cookies, local storage) and non-shared resources (e.g., postMessage, clipboard).

Thomas et al. [30] developed an automated testing framework, *BrowserAudit*, which assessed the implementations of CSP, CORS, and HSTS, and reported bugs in Firefox's CSP implementation. Wi et al. [62] performed a differential analysis of 4 CSP directives across Chrome, Firefox, and Safari and uncovered 37 browser bugs resulting from varying interpretations of the CSP specification. Franken et al. [25] evaluated the third-party cookie policies of 7 major browsers and 46 browser extensions and reported that virtually every browser-enforced and extension-enforced policy could be bypassed. Squarcina et al. [54] performed a cross-browser evaluation of cookie integrity attacks and reported vulnerabilities resulting not only from inconsistent implementations but also from oversight in the formal specification. Kondracki et al. [36] assessed the risks posed by the implementation of the data-saving feature of different mobile browsers. Rautenstrauch et al. [49] analyzed side-channel-based state inference attacks (XS-Leaks), and reported multiple leaks including the potential for an attacker to determine a user's login state on 77/100 top-ranked Tranco websites. Ali et al. [4] evaluated the implementation of storage, cache, access control, and policy mechanisms on 7 browsers and reported 20 vulnerabilities resulting from inconsistencies that could be used as privacy-invasive tracking vectors. Mendoza et al. [43] demonstrated inconsistencies in security headers included in HTTP responses for requests sent from desktop and mobile browsers, and demonstrated that developers often reduce or omit security practices when providing cross-platform support. Bernardo et al. [7] leveraged the WPT suite to evaluate browser implementations of security mechanisms based on their execution traces, which highlighted variable interpretations of the formal specification of the same security mechanisms across browser vendors. Finally, Xiao et al. [64] deployed a hybrid analysis tool that detects `Hidden Property Abusing` attacks, where attackers exploit non-enumerable and symbol-based properties in JavaScript objects to compromise server-side applications built in Node.js.

**Electron Security.** While prior work has focused on the implementation of standard security mechanisms on browsers, there has been limited analysis of the Electron framework. However, researchers have analyzed vulnerabilities in apps built using the Electron framework. Carettoni [12] built a static analysis tool, Electronegativity, that gathers the HTML and JS files of an Electron app and checks for the presence of insecure coding practices based on known implementation flaws. Krishna et al. [38] reported practical exploits of popular apps that use insecure web preferences. Xiao et al. [65] instrumented V8 to perform a taint-analysis of potential cross-context control flow exploits based on communication between the renderer and main processes. Jin et al. [31] proposed a framework to protect Electron apps from unintended modifications to the DOM tree. Ali et al. [5] instrumented the Electron framework to perform dynamic analysis of 109 apps for 16 classes of security feature misconfigurations and reported that almost all apps had at least one misconfiguration.

Despite the popularity of Electron apps, the framework itself has received limited attention for its implementation of standard security mechanisms. The framework uses security mechanisms in its interaction with various web services but adheres to different threat models than traditional web browsers. Vulnerabilities resulting from inconsistent implementations of security mechanisms in Electron apps can have significant consequences, since these apps often have access to sensitive user data and system resources. Our work highlights the need to consider non-browser web platforms (e.g., Electron) in evaluating the implementation of web standards.

## 7 Conclusions

Recently, there has been an increase in popular web services offering a standalone application as an alternative to their browser-based web application. One contributing factor is the cross-platform application-development paradigm, which allows web services to reuse significant portions of their web application codebase, thereby avoiding the expensive and error-prone development process traditionally required for developing native applications for different operating systems. While a few prior studies have found various security flaws in cross-platform applications, in this paper we aimed to explore whether inherent semantic gaps (i.e., *lacunae*) exist between web-based and native code execution, thereby demonstrating the inherent security risks of directly migrating web application code to a cross-platform Electron application. For instance, our findings highlight how even a concept as essential as that of an *origin*, can fundamentally differ between these two contexts. While loading and executing code from the local filesystem is typical for a cross-platform application's execution, such a notion is inconceivable for a web application. Therefore, directly migrated web application code will lack the necessary safeguards to properly enforce access control and context isolation when executed as part of a cross-platform application. The implications of this semantic mismatch become more pronounced when considering the use of bundled third-party libraries and SDKs which will be loaded from the local filesystem, and can then fetch remote third-party code. Crucially, the responses to our disclosure argued that these are intentional design choices by the Electron framework, thereby shifting the onus of incorporating appropriate safeguards to the developers. Given the well-documented struggles of developers with the deployment and configuration of existing web security mechanisms, relying on them to proactively introduce additional safeguards during the migration process so as to prevent threats that are not applicable in the browser environment, or anticipate problematic behaviors that deviate in nuanced ways, will ultimately result in users being exposed to significant threats.

## Acknowledgments

We thank the anonymous reviewers and the revision editor 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] 2025. Paper artifacts. https://github.com/masood/electron-wpt.
[2] Lawrence Abrams. 2024. Signal downplays encryption key flaw, fixes it after X drama. https://www.bleepingcomputer.com/news/security/signal-downplays-encryption-key-flaw-fixes-it-after-x-drama/.
[3] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 289–305. https://doi.org/10.1109/SP.2016.25
[4] Mir Masood Ali, Binoy Chitale, Mohammad Ghasemisharif, Chris Kanich, Nick Nikiforakis, and Jason Polakis. 2023. Navigating Murky Waters: Automated Browser Feature Testing for Uncovering Tracking Vectors. In *Network and*

*Distributed System Security (NDSS) Symposium*. Internet Society, San Diego, CA, USA, 1–18. https://doi.org/10.14722/ndss.2023.24072

[5] Mir Masood Ali, Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis. 2024. Rise of Inspectron: Automated Black-box Auditing of Cross-platform Electron Apps. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 775–792. https://www.usenix.org/conference/usenixsecurity24/presentation/ali

[6] SeleniumHQ and. 2004. Selenium: Browser Automation Framework. https://www.selenium.dev/.

[7] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. 2024. Web Platform Threats: Automated Detection of Web Security Issues With WPT. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 757–774. https://www.usenix.org/conference/usenixsecurity24/presentation/bernardo

[8] Bugcrowd Disclosure. 2025. Critical Local File Read in Electron Desktop App. https://bugcrowd.com/disclosures/f7ce8504-0152-483b-bbf3-fb9b759f9f89/critical-local-file-read-in-electron-desktop-app.

[9] Giuseppe Calderonio, Mir Masood Ali, and Jason Polakis. 2024. Fledging Will Continue Until Privacy Improves: Empirical Analysis of Google's Privacy-Preserving Targeted Advertising. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4121–4138. https://www.usenix.org/conference/usenixsecurity24/presentation/calderonio

[10] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. 2017. Surviving the Web: A Journey into Web Session Security. *ACM Comput. Surv.* 50, 1, Article 13 (March 2017), 34 pages. https://doi.org/10.1145/3038923

[11] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, Ben Stock, et al. 2021. Reining in the web's inconsistencies with site policy. In *Proceedings of the Network and Distributed System Security Symposium 2021*. INTERNET SOC, Internet Society, Virtual, 1–16. https://doi.org/10.14722/ndss.2021.23091

[12] Luca Carettoni. 2017. Electronegativity - A Study of Electron Security. , 23 pages. https://infocondb.org/con/black-hat/black-hat-usa-2017/electronegativity-a-study-of-electron-security

[13] Chrome DevTools Team. 2025. Puppeteer: Headless Chrome Node.js API. https://pptr.dev.

[14] Contributors. 2010. mitmproxy: A Free and Open Source Interactive HTTPS Proxy. https://mitmproxy.org/.

[15] Cookiebot. 2025. Cookiebot CMP - GDPR and ePrivacy Compliant Cookie Consent. https://www.cookiebot.com/. Accessed: 2025-05-21.

[16] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1953–1970. https://doi.org/10.1145/3372297.3417869

[17] Electron. 2025. WebRequest API - Electron Documentation. https://www.electronjs.org/docs/latest/api/web-request.

[18] Electron Contributors. 2024. Patches in Electron. https://www.electronjs.org/docs/latest/development/patches.

[19] Electron Contributors. 2025. Electron - Showcase. https://www.electronjs.org/apps.

[20] Electron Contributors. 2025. Electron Protocol API. https://www.electronjs.org/docs/latest/api/protocol.

[21] Electron Contributors. 2025. Security Tutorial. https://www.electronjs.org/docs/latest/tutorial/security.

[22] Electron Development Team. 2025. Electron Releases. https://releases.electronjs.org/releases/stable.

[23] Electron Development Team. 2025. Electron Releases: Chromium Version Support. https://www.electronjs.org/docs/latest/tutorial/electron-timelines.

[24] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. 2017. Measuring HTTPS Adoption on the Web. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1323–1338. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/felt

[25] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. 2018. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 151–168. https://www.usenix.org/conference/usenixsecurity18/presentation/franken

[26] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. 2018. O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1475–1492. https://www.usenix.org/conference/usenixsecurity18/presentation/ghasemisharif

[27] Google Chrome Developers. 2025. Declarative Net Request API. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/.

[28] Peter Leo Gorski, Luigi Lo Iacono, Stephan Wiefling, and Sebastian Möller. 2018. Warn if Secure or How to Deal with Security by Default in Software Development?.. In *HAISA*. Springer, Mytilene, Greece, 170–190.

[29] Jeff Hodges, Collin Jackson, and Adam Barth. 2012. *Http strict transport security (hsts)*. Technical Report. Internet Engineering Task Force.

[30] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. 2015. BrowserAudit: automated testing of browser security features. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 37–47. https://doi.org/10.1145/2771783.2771789

[31] Zihao Jin, Shuo Chen, Yang Chen, Haixin Duan, Jianjun Chen, and Jianping Wu. 2023. A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities. In *Network and Distributed System Security (NDSS) Symposium*. Internet Society, San Diego, CA, USA, 1–16. https://doi.org/10.14722/ndss.2023.24305

[32] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *Network and Distributed System Security Symposium*. Internet Society, Virtual, 1–16. https://doi.org/10.14722/ndss.2021.23104

[33] Christoph Kerschbaumer, Julian Gaibler, Arthur Edelstein, and Thyla van der Merwe. 2021. HTTPS-Only: Upgrading all connections to https in Web Browsers. In *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. Internet Society, Virtual, 1–11. https://doi.org/10.14722/madweb.2021.23010

[34] Soheil Khodayari and Giancarlo Pellegrino. 2022. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1590–1607. https://doi.org/10.1109/SP46214.2022.9833637

[35] Natalia N Kislitsyna, Natalia N Kislitsyna, et al. 2021. Lacunae in language and speech: intercultural and textual aspects. *European Proceedings of Social and Behavioural Sciences* 102 (2021), 497–505. https://doi.org/10.15405/epsbs.2021.02.02.62

[36] Brian Kondracki, Assel Aliyeva, Manuel Egele, Jason Polakis, and Nick Nikiforakis. 2020. Meddling Middlemen: Empirical Analysis of the Risks of Data-Saving Mobile Browsers. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, Virtual, 810–824. https://doi.org/10.1109/SP40000.2020.00077

[37] Michael Kranch and Joseph Bonneau. 2015. Upgrading HTTPS in Mid-Air: An Empirical Study of HTTP Strict Transport Security and Key Pinning. In *Network and Distributed System Security (NDSS) Symposium*. Internet Society, San Diego, CA, USA, 1–15. https://doi.org/10.14722/ndss.2015.23162

[38] Mohan Sri Rama Krishna, Max Garrett, Aaditya Purani, and William Bowling. 2022. ElectroVolt: Pwning Popular Desktop Apps While Uncovering New Attack Surface on Electron. https://www.youtube.com/watch?v=Tzo8ucHA5xw

[39] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl. 2017. "I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1339–1356. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/krombholz

[40] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. 2017. Hindsight: Understanding the Evolution of UI Vulnerabilities in Mobile Browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 149–162. https://doi.org/10.1145/3133956.3133987

[41] Francesco Marcantoni, Michalis Diamantaris, Sotiris Ioannidis, and Jason Polakis. 2019. A Large-scale Study on the Risks of the HTML5 WebAPI for Mobile Sensor-based Attacks. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 3063–3071. https://doi.org/10.1145/3308558.3313539

[42] MDN Web Docs. 2025. MDN Web Docs: Resources for Developers. https://developer.mozilla.org/en-US/.

[43] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. 2018. Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 247–256. https://doi.org/10.1145/3178876.3186091

[44] Javad Nejati, Meng Luo, Nick Nikiforakis, and Aruna Balasubramanian. 2020. Need for mobile speed: A historical analysis of mobile web performance. In *Network Traffic Measurement and Analysis Conference (TMA)*. IFIP, Berlin, Germany, 1–9.

[45] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. Association for Computing Machinery, New York, NY, USA, 296–305. https://doi.org/10.1145/2664243.2664254

[46] OpenHub Contributors. 2025. Black Duck - Chromium. https://openhub.net/p/chrome/analyses/latest/languages_summary.

[47] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

[48] R. Fielding and J. Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://www.rfc-editor.org/rfc/rfc7231
[49] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. 2023. The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Fransisco, CA, USA, 2744–2760. https://doi.org/10.1109/SP46215.2023.10179311
[50] Jukka Ruohonen. 2025. SoK: The design paradigm of safe and secure defaults. *Journal of Information Security and Applications* 90 (2025), 103989.
[51] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. 2017. Same-Origin Policy: Evaluation in Modern Browsers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 713–727. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk
[52] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. 2010. On the Incoherencies in Web Browser Access Control Policies. In *2010 IEEE Symposium on Security and Privacy*. IEEE, Berkeley, CA, USA, 463–478. https://doi.org/10.1109/SP.2010.35
[53] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. 2016. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 724–742. https://doi.org/10.1109/SP.2016.49
[54] Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. 2023. Cookie Crumbles: Breaking and Fixing Web Session Integrity. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5539–5556. https://www.usenix.org/conference/usenixsecurity23/presentation/squarcina
[55] Emily Stark, Mike West, and Carlos Ibarra Lopez. 2023. Mixed Content. https://www.w3.org/TR/mixed-content/
[56] Avinash Sudhodanan and Andrew Paverd. 2022. Pre-hijacked accounts: An Empirical Study of Security Failures in User Account Creation on the Web. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1795–1812. https://www.usenix.org/conference/usenixsecurity22/presentation/sudhodanan
[57] Ankur Sundara. 2024. Cookie Bugs: The Hidden Threats in Web Applications. https://blog.ankursundara.com/cookie-bugs/.
[58] Dirk Van Der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein Than Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. 2020. Schrödinger's security: opening the box on app developers' security rationale. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery,, Seoul, South Korea, 149–160.
[59] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1376–1387. https://doi.org/10.1145/2976749.2978363
[60] Michael Weissbacher, Tobias Lauinger, and William Robertson. 2014. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses*, Angelos Stavrou, Herbert Bos, and Georgios Portokalidis (Eds.). Springer International Publishing, Cham, 212–233.
[61] Mike West and Yan Zhu. 2023. Secure Contexts. https://w3c.github.io/webappsec-secure-contexts/.
[62] Seongil Wi, Trung Tin Nguyen, Jiwhan Kim, Ben Stock, and Sooel Son. 2023. DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing. In *Network and Distributed System Security (NDSS) Symposium*. Internet Society, San Diego, CA, USA, 1–16. https://doi.org/10.14722/ndss.2023.24200
[63] wpt contributors. 2019. web-platform-tests documentation. https://web-platform-tests.org/.
[64] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing Hidden Properties to Attack the Node.js Ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Online, 2951–2968. https://www.usenix.org/conference/usenixsecurity21/presentation/xiao
[65] Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. 2022. Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2975–2988. https://doi.org/10.1145/3548606.3559340
[66] Zhou Yuchen and Evans David. 2010. Why Aren't HTTP-only Cookies More Widely Deployed?. In *Proceedings of 4th Web 2.0 Security and Privacy Workshop (W2SP)*. IEEE, Oakland, CA, USA, 1–5. https://www.ieee-security.org/TC/W2SP/2010/papers/p25.pdf
[67] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

# A Appendix: Real-World Electron App Dataset

Here we provide additional details about our real-world Electron application analysis.

Table 6 presents the deployment characteristics of the 30 Electron applications analyzed in our study. The selection includes a diverse range of productivity, developer, and utility tools from the official Electron showcase [19]. For each application, we extract multiple configuration and deployment characteristics:

(1) We detect whether the main renderer is loaded from a local origin (`file://`) and assess the source of embedded resources, specifically whether scripts and iframes are loaded from local and remote origins.
(2) We analyze WebView usage, distinguishing between instances that load content from local files and those that fetch remote resources.
(3) We evaluate Content Security Policy (CSP) deployment by inspecting both `<meta>` tags and HTTP response headers.

This analysis provides a detailed view of real-world deployment practices and highlights configurations that impact origin isolation or introduce elevated privilege risks, as we detail in §3.

**Table 6: Deployment characteristics of the 30 Electron applications we evaluated.**

| Application | Origin→Inclusions | | | WebView Embedding | | CSP |
|---|---|---|---|---|---|---|
| | File→File | File→Remote | Remote→Remote | Remote Source | Local Source | |
| **Advanced REST Client** | ✓ | – | – | – | – | – |
| **Airtame** | ✓ | – | – | – | – | – |
| **Altair GraphQL Client** | ✓ | – | – | – | – | – |
| **Asana** | – | ✓ | – | – | – | ✓ |
| **Bibico** | ✓ | – | – | – | – | – |
| **Biscuit** | – | ✓ | – | – | ✓ | – |
| **Boxhero** | ✓ | – | – | – | – | ✓ |
| **Buckets** | ✓ | – | – | – | – | – |
| **Cacher** | – | ✓ | – | ✓ | – | ✓ |
| **Discord** | ✓ | – | – | – | – | – |
| **Dynobase** | – | ✓ | – | – | – | – |
| **Etcher** | ✓ | – | – | – | – | – |
| **Figma** | – | ✓ | – | ✓ | – | ✓ |
| **Flat** | – | – | ✓ | – | – | – |
| **Fontbase** | – | ✓ | – | – | – | – |
| **GitHub** | – | – | ✓ | – | – | ✓ |
| **GitKraken** | ✓ | – | – | – | – | ✓ |
| **Heroic** | ✓ | – | – | – | – | ✓ |
| **Hive** | ✓ | – | – | – | – | – |
| **Loom** | ✓ | – | – | – | – | – |
| **Obsidian** | ✓ | – | – | – | – | – |
| **Postman** | – | ✓ | – | ✓ | ✓ | – |
| **Signal** | ✓ | – | – | – | – | ✓ |
| **Slack** | – | – | ✓ | – | – | ✓ |
| **Splice** | – | – | ✓ | – | ✓ | – |
| **Tidal** | – | ✓ | – | ✓ | – | ✓ |
| **Tropy** | ✓ | – | – | – | – | – |
| **VSCode** | ✓ | – | – | – | – | ✓ |
| **WordPress** | – | – | ✓ | – | – | – |
| **Zettlr** | ✓ | – | – | – | – | ✓ |