TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Silvia Väli IVCM153489

ANALYSIS OF ELECTRON-BASED APPLICATIONS TO IDENTIFY XSS FLAWS ESCALATING TO CODE EXECUTION IN OPEN-SOURCE APPLICATIONS

Master's thesis

Supervisor: Olaf Maennel
Ph.D. (Dr.rer.nat.)
Professor

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Silvia Väli

30th of December 2017

Abstract

JavaScript which has conquered the worlds of client-side and server-side programming has encouraged the creation and popularity of cross-platform frameworks like Electron. This has introduced a way to create native applications suitable for multiple platforms by only using web technologies which web developers are familiar with. Unfortunately, the transmission of web technologies, like JavaScript from sandboxed browsers to the desktop environment has introduced vulnerabilities common in web applications to a whole new environment.

Web vulnerabilities have been extensively studied long over a decade, so it could be assumed that developers are aware of the consequences of security issues and the respective preventative measures. However, cross-site scripting has nevertheless remained to be a common phenomenon in web applications.

By default, Electron-based applications are executed in an unsandboxed environment, where cross-site scripting on a single page application can pose risks more severe than they ever would in a web application. This has encouraged research on framework specific security issues in order to bring awareness on the matter with appropriate prevention methods.

The contribution made in this thesis aims to validate the hypothesis that many of the open-source Electron-based applications, currently available in Github, are vulnerable to cross-site scripting that has the capability to evolve into code execution. In achieving that, an analysis method in two stages was followed to firstly gather statistical information and secondly to conduct manual analysis on observed applications. From manual analysis, ~37% of the applications were identified as vulnerable by following a three step process.

This thesis is written in English and is 77 pages long, including 8 chapters, 54 figures and 9 tables.

Abstrakt

Analüüs Electron'i põhistest rakendustest tuvastamaks XSS turvavea eskaleerumist koodikäivituseks

JavaScript, algselt loodud kui kliendipoolne programmeerimiskeel, on tänaseks sujuvalt kasutusel ka serveri poolel. Selline areng on toetanud mitmeplatvormiliste rakenduse raamistike nagu Electron loomist ning populariseerimist. Tänu sellistele raamistikele on võimalik luua desktop rakendusi põhinedes vaid veebitehnoloogiatel, millega veebirakenduste arendajat juba tuttavad on. Veebitehnoloogiate, nagu näiteks JavaScript'i üleminek turvakaalutlustel isoleeritud (*sandboxed*) brauseri keskkonnast desktopi rakenduste keskkonda on toonud kaasa olukorra kus algselt vaid brauseri keskkonnas esinevad turvahaavatavused esinevad ka mujal.

Veebihaavatavusi on uuritud põhjalikult üle kümnendi, seega võiks oletada, et arendajad on teadlikud haavatavustega seotud tagajärgedest ning õigetest kaitsemeetoditest. Sellest hoolimata on jäänud veebis esinevad turvahaavatavused nagu Cross-Site Scripting (XSS), tihti esinevaks nähtuseks.

Electron'i rakendused käivitatakse originaalis vaikeväärtusega, millest tulenevalt brauserile tavapärast keskkonna isoleeritust (*sandbox*) ei rakendata. See on toonud kaasa olukorra, kus haavatavuse, nagu XSS, esinemisel on riski võimalik tagajärg palju tõsisem, kui selle esinemisel brauseris. Sellest tulenevalt on hoogustunud ka Electron'i raamistikupõhiste turvahaavatavuste ning võimalike kaitsemeetodite uurimine.

Antud lõputöös panustatakse avatud lähtekoodiga Electroni-põhiste rakenduste uurimisega valideerimaks järgnevat hüpoteesi: paljud Github'is olevad Electroni-põhised rakenduse on haavatavad XSS-tüüpi rünnetele, millest tulenevalt on võimalik ohvri masinas koodi käivitada. Selle saavutamiseks, kasutati kahest etapist koosnevat meetodit, kogumaks statistilisi andmeid analüüsitavate rakenduste kohta ning teostades rakendustel manuaalse testimise valideerimaks hüpoteesi. Manuaalse testimise tulemusena tuvastati haavatavus 37% analüüsitud rakendustest.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 77 leheküljel, 8 peatükki, 54 joonist, 9 tabelit.

List of abbreviations and terms

OWASP	The Open Web Application Security Project
CVE	Common Vulnerabilities and Exposures
HTTP	Hypertext Transfer Protocol
HTTPS	HyperText transfer Protocol Secure
AST	Abstract Syntax Tree
US	United States
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
CPU	Central Processing Unit
API	Application Programming Interface
NW	Node-Webkit
JIT	Just in Time
IPC	Interprocess communication
GUI	Graphical user Interface
XML	Extensible Markup Language
DOM	Document Object Model
XSS	Cross-site scripting
URL	Uniform resource locator
NOSQL	Non-relational Structured Query Language
SSJI	Server-side JavaScript Injection
JSON	JavaScript Object Notation
SQL	Structured Query Language
GPU	Graphics Processing Unit
npm	Node package manager

Table of Contents

List of Figures

List of Tables

1 Introduction

As data theft, fraud and breaches are an everyday phenomenon in the news, application security has gotten more attention with every coming year. Why those types of news make the headlines is all dependent on the impact of the vulnerability, as the same type of vulnerabilities can present themselves differently in every application. Any existing vulnerabilities in the software we use, can expose personal information to third parties with criminal intentions.

“Common Vulnerabilities and Exposures” - CVE details, which keeps track of known vulnerabilities, also provides some interesting yearly statistics on “Top 50 Products By Total Number of ‘Distinct’ vulnerabilities” [1] . Examining the first top 20 products from that list, one may see that it contains all four widely used browsers Chrome, Firefox, Safari and Internet Explorer. By the W3Counter statistics, in 2017 September, these four browsers make up of 90% of the browser market share [2] . This makes it highly likely that a targeted user is using at least one or more of those four browsers, therefore might be using a vulnerable application.

To access many of the other resources served over the World Wide Web, confidential data such as banking information, private documents and more is trusted to be hosted by a third party. According to the report, published in 2017 by the Verizon, 571 databreaches out of 1935 that they investigated were web application attacks [3] . Another yearly report published by the Trustwave, claims to have observed 13% of the web attacks to be related to cross-site scripting vulnerabilities and with 99.7% of the applications in present of at least one vulnerability [4] . Cross-Site scripting is not a new phenomenon in the scene of web application vulnerabilities and continues to occur despite being thoroughly researched more over than a decade.

By 2017 JavaScript, originally built to be the client side programming language, has conquered both worlds and being actively used also for the server-side development. This has encouraged the creation of frameworks like Electron and Nw.js, which enable web developers to develop multiplatform desktop applications by only using web technologies. This unfortunately has introduced client-side vulnerabilities also to the server-side code which due to their potentially elevated impact has created emerging interest for understanding these types of vulnerabilities.

1.1 Motivation

The Open Web Application Security Project (OWASP), has made it a priority to advocate the overall web application security scene, by producing community dedicated material including Web Application Security Top 10.

OWASP Top 10 brings an overview of the main web application security issues based on the collected data, surveys and personal experience of the OWASP team. They have emphasized the changes within the past four years (the previous OWASP Top 10 was published in 2013) in application's architecture which accelerated the release of the top 10 published in 2017. As the result of their research they have concluded that [5] :

- JavaScript has become one of the most popular languages in web development;
- older enterprise applications are replaced by microservices written in Node.js and Spring boot;
- there has been a rise of the single page applications written in JavaScript frameworks such as Angular or React;
- wider usage of modern web frameworks like Bootstrap, Angular, React and Electron has resulted in formerly backend source code to be running in the client side, in the untrusted browsers.

These notable changes have presented a quick and comfortable way to create multiplatform desktop applications, such as Electron. Electron is an open source project, previously known by the name of Atom shell [6] , developed and maintained by Github. It has been adopted and actively used in the development of desktop versions for applications such as Wire, Skype, Wordpress, Shopify, Github, GitKraken, Tidal, Trello and many others [7] .

Presenting web applications in a desktop like environment using the components of Electron - Chromium, Node.js and V8 - has presented web application vulnerabilities to a new environment. Exploitation of those vulnerabilities can have unforeseen impact in the desktop environment that web developers might not be aware of as presented by the application security company Doyensec [8] . Therefore, it is important to have a good understanding of the characteristics of those vulnerabilities and their impact in desktop environment in order to improve the security of Electron based applications.

1.2 Problem Statement and the contribution

In 2017 Black Hat US, security checklist for Electron-based applications [9] was presented by Luca Carettoni from Doyensec. Quoting from the report:

“Many companies have started providing native desktop software built using the same technologies as their web counterparts. In this trend, Github's Electron has become a popular framework to build cross-platform desktop apps with JavaScript, HTML, and CSS. While it seems to be easy, embedding a web application in a self-contained web environment (Chromium, Node.Js) leads to new security challenges. ”

As part of their research, they introduced a security checklist which involved the overview of misconfigurations and vulnerabilities that can occur in Electron-based applications. They had also implemented a tool to check for the presented security issues called Electronegativity, which on this day is not publicly available anymore for unknown reasons. The cohesive set of vulnerabilities they presented were in majority connected with the settings of web page’s features called as WebPreferences object. A noteworthy section of the report was discussing the combination of rendering untrusted content on a page where JavaScript is allowed to access operating system primitives. This means untrusted content has access to taking advantage of native desktop mechanisms like reading and writing files which refers to wider attack surface.

As Electron is a relatively new framework, used by the web developers who are familiar with web technologies, the following hypothesis is posed:

Many of the open source Electron-based applications are vulnerable to cross-site scripting, which due to misconfigurations or missing configurations in WebPreference options escalates to code execution.

In order to validate or disproof the hypothesis, following novel contributions are presented within two stages of analysis conduted in this study.

First stage focuses on analying open-source Electron-based applications in order to collect statistical information on three subjects of interest for finding common features. The subjects analysed are:

- the most common Electron specific modules required through out the applications;
- webPreference options which are the options that control specific features of the application page displayed to the user;
- remote content included to the application via loadURL in BrowserWindow and BrowserView, which are the classes in Electron framework to create and manage browser windows; remote content included via window.open() method or a webview tag used to display external web content.

As first stage focuses on gathering statistical data, the second stage approaches directly the validation or disproof of the hypothesis. This incorporates a method with three subsequent steps to be followed in the analysis of each application from the dataset. For the observed application to be identified as vulnerable, following attempts must result in success:

  - first step attempts to identify the presence of a cross-site scripting vulnerability from the observed application;

    - second step attempts to determine whether the browser window with identified cross-site scripting vulnerability has granted access to using node modules which represent a set of functions that can be required in the application;
    - third, and the final step of the second stage involves producing a payload to showcase the seriousness of the issue with a realistic attack vector for the vulnerable application.

Based on the results gathered from second stage:

- as a conclusive outcome, findings on vulnerable applications and the statistical data gathered from analysed Electron-based applications is presented in chapter 6.

1.3 The scope

The scope of this thesis is focused on Electron-based applications in Github that are available for code review and are successfully executable in the desktop environment. The statistical data is based purely on the source code examination. The detection of cross-site scripting vulnerabilities follows the similar actions taken to detect cross-site scripting vulnerabilities in web applications, which we assume the reader to be familiar with.

1.4 Related research

This subsection of “Related research” will not present any substantial amount of related research papers as no academic papers were found to directly approach or tackle Electron related research questions relevant to the hypothesis to be proven in this thesis.

As Electron framework uses web technologies, hence the exposure to cross-site scripting vulnerability, any research on XSS could be counted as related work, however indirectly. Therefore XSS was introduced in this thesis in section 3 via examples of client- and server-side JavaScript.

Performing searches in popular search engines with relevant search terms to Electron did not present any academically acceptable research papers, while this thesis contributes to achieving exactly that.

2 Electron framework

Electron is an open source framework for developing cross-platform desktop applications by using JavaScript, HTML and CSS as defined by the official Electron documentation [10] . Developed and maintained by Github, it has come a long way from 2015 when what was originally developed and called as Atom Shell, was then named Electron. At the present day, the popularity of Electron is thriving and the number of downloads has reached up to 1.2 million [11] .

When talking about the Electron framework, it really means talking about the three core components on which it is based on: Node.js, Chromium and V8 - the JavaScript engine. Following sections will discuss the Electron framework, its core components and structure, in order to give more complete understanding of the studied framework and what it provides to the applications.

2.1 Electron components

Electron framework has been implemented based on the three core components: Node.js, Chromium and V8 JavaScript engine. A single instance of the V8 engine is used both by Node.js and Chromium.

2.1.1 Chromium

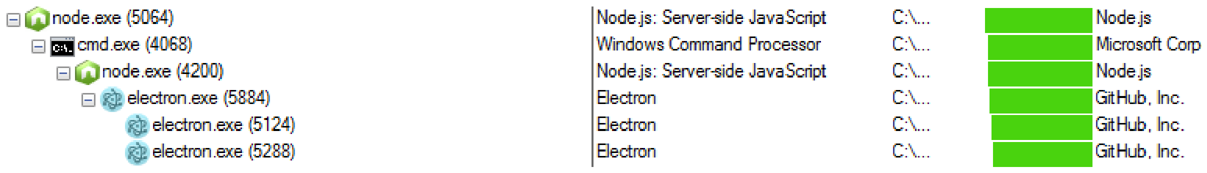
Chromium is an open source browser project, developed and maintained by the Chromium Project and is the basis to the Google Chrome browser. Electron only uses the rendering library from Chromium in order to maintain and limit the scope of the framework.

The content module used by the Electron is consumed from the Chromium's repository [12] , where it is packaged and in Electron repository known as the libchromiumcontent

[13] . This module holds every piece of code needed to render a page in a multiprocess sandboxed browser [14] . Content module, however, does not include the Chrome features such as autofills, spelling, extensions and others, but implements the APIs on which these features can be built upon.

Electron follows the concept of multi-process architecture [15] implemented in Chromium, where one or multiple renderer processes are launched from the main process. Each of the renderer processes can hold one or many renderview objects. This is analogous to the Chrome browser and how different tabs within the browser window are managed. Each renderer process has access to communication with its parent process, while being in isolation with the other renderer processes. An example of a process creation can be shown by launching the Electron "Quick Start" [16] application and observing the created processes via Process Monitor as shown in Figure 1.

A simple application such as "Quick Start", displaying the text and the versions of Node.js, Chromium and Electron, will create the following processes:



```
node.exe (5064)                 Node.js: Server-side JavaScript   C:\...   Node.js
  cmd.exe (4068)                Windows Command Processor         C:\...   Microsoft Corp
    node.exe (4200)             Node.js: Server-side JavaScript   C:\...   Node.js
      electron.exe (5884)       Electron                          C:\...   GitHub, Inc.
        electron.exe (5124)     Electron                          C:\...   GitHub, Inc.
        electron.exe (5288)     Electron                          C:\...   GitHub, Inc.
```

Figure 1. Processes created when Electron app is launched

Electron.exe (5884) is where the application is eventually launched. From there on electron.exe (5124) on figure 2 and electron.exe (5288) on figure 3 show the commands for launching the GPU and the renderer processes.

electron.exe (5124) – GPU process

```
"C:\Users\user\electron-quick
start\node_modules\electron\dist\electron.exe" --type=gpu-process
--no-sandbox "
```

Figure 2. Command used for the GPU process

electron.exe (5288) – renderer process

```
"C:\Users\user\electron-quick-
start\node_modules\electron\dist\electron.exe" --type=renderer –no-
sandbox  ... --lang=en-US --app-path="C:\Users\user\electron-quick-
start" --node-integration=true --webview-tag=true --no-sandbox
--enable-pinch --device-scale-factor=1“
```

Figure 3. Command used for the renderer process

Security is one of the most important goals for Chromium [17] , but by following the executed commands it is evident that by default Electron is launched in the Chromium wrapper without the sandbox feature turned on. It also has enabled node integration and webview tag usage, which respectively enable access to the Node APIs and to embedding third party content. Therefore, the content displayed in the applicvationhas access to the operating system and its native APIs to perform operations such as reading and writing files.

Electron, using web technologies, is challenged by the same security issues as any web application running inside the Chrome browser [18] , but the impact of an attack can reach much further. In a sandboxed environment having a vulnerability such as cross-site scripting is somewhat contained, but yet has extensive impact. It can enable the attacker to perform session hijacking by stealing user’s session cookie, performing malicious redirections, access confidential data and even perform actions on victim’s behalf. Without having the sandboxed environment the consequences can reach much further. This will be discussed within the upcoming sections about Electron security.

2.1.2 Node.js

With the popularity of JavaScript, its stable establishment to the software world has been inevitable. Breaking the boundaries of only running JavaScript on the client side has paved the way for server-side applications, as well as mobile and desktop application frameworks.

Node.js is an open source JavaScript runtime which uses the V8 engine to parse, compile and run JavaScript. As Node.js is asynchronous, it allows to handle many operations concurrently. It comes with its own package manager npm, which holds the largest number of open source libraries in the world. In December of 2015, the number of modules was slightly passing over 200 000 [19] . Currently, approximately 540 478 modules, with average of 569 new modules added per day [20] has proven its growing popularity and adoption by the community.

The launch of Node.js in 2009 enabled JavaScript to go beyond the barrier of web applications and contribute to the development of server-side software. Traditionally, server-side software has always been built with platform specific methods. However, JavaScript-based platforms, NW.js and Electron skyrocketed the development of cross-platform desktop applications. NW.js having over 206,439 downloads [21] in 2017 and Electron over 4,368,327 [22] downloads based on the statistics provided by npm-stat module. JavaScript has made it possible for the web developers to be building desktop applications by using web technologies.

Electron is highly dependent on Node and the possibilities it provides for the development. It provides the application with integrated node access to require node modules which then can be used within the application. This includes requiring modules such as file system - to read/write files, os - to use operating system-related methods, for example to ask information about the CPU, home directory location, user’s machine’s hostname, amount of the free memory on the system, to acquire IP-s and network interfaces etc., path - to work with file and directory paths, and many other modules.

While the number of available packages provided by Node is humongous, it is advisable for security reasons to choose the third party packages responsibly. In the Node Interactive North America conference, presentation on Node security was made by a company called Snyk. Snyk is a company which builds tools to secure the dependencies used in open source projects, to monitor Node.js npm and Ruby packages. They brought a good perspective on a typical application using Node packages, where the actual code contribution written by the developer is rather small comparing to the actual size of the application what comes from the code in npm. This in turn means that most of the vulnerabilities come from the npm and as they announced in the December of 2016, 14% of those packages were known to be vulnerable at that time [23] .

2.1.3 V8

V8 is the open source JavaScript runtime engine used in Chrome browser and in Node.js. What it does is that it takes JavaScript code, parses it, creates an abstract syntax tree (AST) and eventually ends up generating the bytecode that will be processed. V8 is embedded inside the Node.js, which similarly is written in C++. As Node.js hooks to the V8 engine, it extends its capabilities by adding additional features such as reading and writing files. In Electron framework, Chromium and Node.js share a single V8 instance.

The main advantage of using V8 is the JIT (Just in Time) compiler, which enables fast JavaScript execution and optimized code, based on observations made at runtime. As mentioned by the Electron team, the V8 version updates rely on the V8 version used for Chromium, which is then patched for it to work in Node, but this might presumably change according to the news published by the Node.js foundation [24] . Quoting from that release:

“The V8 team, a group in charge of Google’s open source high-performance JavaScript engine, now prioritizes Node.js alongside Chromium ensuring V8 cannot be upgraded if it crashes Node.js, which means less strain on the Node.js maintainer community, added stability and earlier adoption of ESNext features.”.

2.2 Processes

Similarly to Chromium, Electron uses the multi-process architecture, in which each of the processes runs concurrently and in isolation with eachother. It also implements the hierarchy where the main process can launch one or multiple renderer processes in which the web content is displayed. Within the next two subsections “Main process” and “Renderer process”, a simple overview will be given of each.

2.2.1 Main process

In Electron, the main process is the process that creates and manages browser windows. The main process is launched when the file, marked as the entry point in the dependency file is executed. Having control over creating and managing browser windows, the main process is entitled to create multiple renderer processes as shown in figure 4.

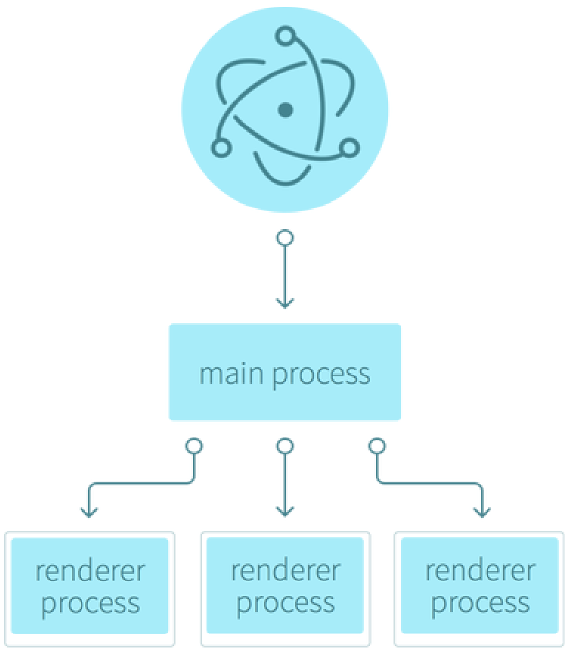


Figure 4. Main and Renderer processes [25]

In addition to managing browser windows, the main process also has access to all the main process Electron APIs [26] that enable to create menus, define keyboard shortcuts, launch frameless windows, control file downloads and access Node.js modules.

For communication with the renderer process, the IPCmain [27] module is used. This module is able to handle asynchronous and synchronous messages sent and received from the renderer.

2.2.2 Renderer process

Each page in Electron runs in its own separate process, which is called the renderer process. Multiple renderer processes can be associated with the main process. The main process is what starts and stops the application and the dependency file is what holds the line “main”: “main.js” marking the entry point to the application. Figure 5 illustrates the creation of a browser window in main.js where index.html is loaded via a specified path once the application is ready. When the instance of webContents is loaded into the browser window, a new renderer process is created.

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

let win

function createWindow() {
      win = new BrowserWindow({width: 800, height: 600})
      win.loadURL(url.format ({
            pathname: path.join(__dirname, 'index.html'),
            protocol: 'file:',
            slashes: true
      }))
}

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

Figure 5. Creating a new window instance with contants displayed from index.html

Another interesting notation is that one browser window can host one or multiple webContent instances, therefore multiple renderer processes can be created from the same window. That situation can occur for example when webviews are used to display the remote content.

The same way the main process has a module to send messages to the renderer process, there is a module called ipcRenderer. This can be used by the renderer process to send and receive messages from the main process [28] . As an example, this allows the renderer process to write into the console, which would not be possible otherwise.

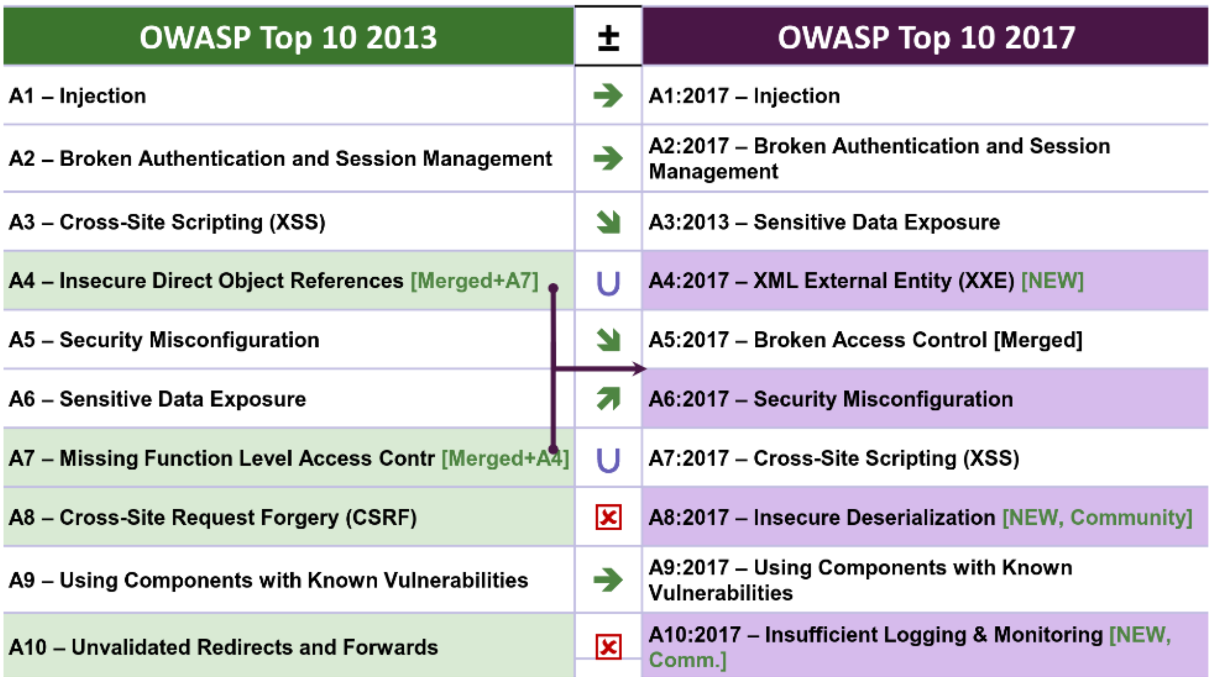
While ipcMain and ipcRenderer are used to send messages between the processes, there is a module for the renderer process which allows to evoke methods only available in the main process. By using the remote module [29] , renderer process is able to access GUI-related operations. This includes access to browser window creation. In order to do so, the renderer process creates an instance of a browser window by using the browserWindow module available in the main process. It is important to note that while the renderer process is the one to create the instance, the instance itself belongs to the main process.

3 Application security

Github managed to build a framework for creating desktop applications by only using web technologies. To make it feel and look more like the classic desktop experience, access to the operating system's native primitives was enabled by default. This way, merging the two worlds Electron-based applications became challenged by the client-side and server-side vulnerabilities. Within the following section an overview of the top 10 web application vulnerabilities will be given with further insight to cross-site scripting on the client and the server side.

3.1 OWASP Top 10

OWASP Top 10 is a document that presents the ten most critical web application security flaws based on the data gathered from over 114,000 applications and 550 community members [30] . OWASP Top 10, release of 2017, is the updated version of the document published in 2013. Modified methodology and working with the community has brought some significant changes, which will be discussed based on the OWASP published document. The conclusive results of the OWASP top 10 survey were discussed as the motivation behind the topic of choice for this thesis, and a short insight to top 10 will be given subsequently.



OWASP Top 10 2013	±	OWASP Top 10 2017
A1 – Injection	→	A1:2017 – Injection
A2 – Broken Authentication and Session Management	→	A2:2017 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	↘	A3:2013 – Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017 – XML External Entity (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017 – Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017 – Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017 – Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017 – Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017 – Insufficient Logging & Monitoring [NEW, Comm.]

Figure 6. OWASP TOP 10 published in 2017 [30]

All 10 categories on figure 6 present a potential path for the attacker to cause harm. Within four years, injection flaws with the combination of severity, impact and likelihood pose the biggest risk. Having an injection flaw can result in loss of confidentiality, integrity and availability.

The prevalence of authentication issues is considered widespread and as lists of usernames and passwords have leaked over the years, attackers have gained information assisting them to conduct brute force attacks more efficiently in order to gain access to user's accounts. Authentication weaknesses or mismanaged sessions can only facilitate that.

European Union General Data Protection Regulation, among other regulations dealing with data protection, have got significantly more attention due to the occurred data breaches broadcasted in the news about the healthcare system [31] . The most common problem is considered to be storing sensitive data unencrypted and to reflect the importance of data confidentiality acquired for the customer's data, sensitive data exposure has been ranked as third from the top.

XML External Entity is included in the top 10 for the first time, which itself indicates a shift of priorities in terms of vulnerabilities discovered from applications. Taking into account XML-based web services which parse XML received from remote resources or from user input, this flaw is considered to be with average exploitability but can have a severe impact.

Broken access control flaws are common as there is lack of functional testing by the developers or testers within the development project. The impact of the flaw can change from low to severe quickly, depending on the data confidentiality, as attackers are able to access the data or perform actions on behalf of the users.

With easy exploitability and detectability, security misconfigurations are considered very widespread in all levels of the application stack and can elevate any of the previously introduced security flaws.

In previous editions of the OWASP Top 10, cross-site scripting has always been ranked highly, but as for the version of 2017, it has been been ranked as seventh from the top. Cross-site scripting flaws come from the problem of displaying untrusted input without proper encoding for the output document. It is still considered to be easily exploitable, detectable and very widespread but the impact evaluated as moderate for reflected and DOM-based XSS and severe for stored XSS with remote code execution in the victim's browser.

Using components with known vulnerabilities is as widespread as before, since there is often no overview of the component's versions used in the application and whether they contain any known vulnerabilities. This can be challenging for applications built on modern platforms like Node.js which require their packages through npm.

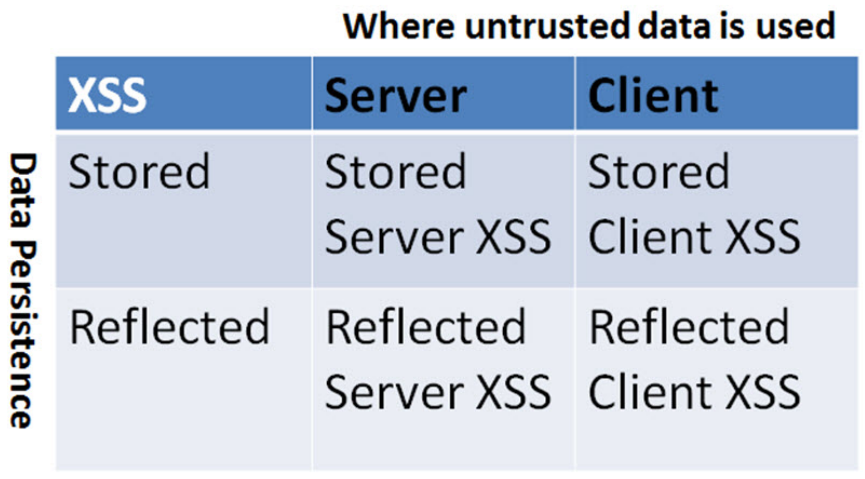
Building applications in Electron framework means using web technologies such as HTML, CSS and JavaScript. Therefore, these applications have the potential to be vulnerable to what is known as cross-site scripting attacks which occur due to improper encoding of untrusted input for the output document. The definition to cross-site scripting and its significant types will be briefly discussed in the coming subsections to clarify the context.

3.2 Cross-site scripting

OWASP defines cross-site scripting (XSS) as “type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.” [32] .

From the definition brought by the the OWASP, it is visible that the definition is aimed at web applications as XSS is considered to be the web browser based attack. However, applications built with Electron framework are considered as desktop applications, which use the web technologies as do the web applications. Similarly, they can potentially suffer from cross-site scripting vulnerabilities. Possibly more proper definition to XSS in Electron-based applications can be referenced from o’reilly archive as: ”Cross-site scripting is a security hazard that allows crackers to interfere with your program’s logic by inserting their own logic into your HTML.” [33] .

OWASP also introduces a document about types of cross-site scripting, which present three types – reflected, stored and DOM based XSS, which in turn overlap and can be introduced as the client-side and the server-side XSS [34] as shown on the figure 7.



		Where untrusted data is used	
	XSS	Server	Client
Data Persistence	Stored	Stored Server XSS	Stored Client XSS
	Reflected	Reflected Server XSS	Reflected Client XSS

Figure 7. Client-side and server-side XSS [34]

The definition of XSS that does not include the relation to the context of HTML, JavaScript or CSS where XSS can occur and the relation to XSS types – reflected, stored, DOM-based XSS, but uses more broad classification can be derived from [35] with a small modification for this thesis.

Def. *XSS is a class of vulnerabilities which allow injection of code into client- and server-side of the application.*

Within the following sections simple overview of client-side and server-side code injections along with the three types - reflected, stored and DOM-based XSS – is given. As this thesis focuses on identifying the presence of cross-site scripting later in the analysis and not its specific classification, the overview given in 3.2.1 – 3.2.4 will not cover all the contexts in which XSS can occur but introduce the essential.

3.2.1 Client-side and server-side XSS

Examination of the literature on XSS refers to client-side XSS mostly as DOM-based cross-site scripting vulnerability [34] [36] [37] . "Web applications often make use of JavaScript code that is embedded into web pages to support dynamic client-side behavior." [36]. Also cited from the OWASP document, "client side XSS occurs when untrusted user supplied data is used to update the DOM..." [34] .

Server-side XSS differs from the client-side code injection as the untrusted data is included to the response that is generated by the server.

In both of the cases with client-side or server-side code injection, the untrusted input could be also sourced from the request made to the server or from a stored location which will be discussed in the subsections 3.2.2-3.2.4.

3.2.2 Reflected cross-site scripting

Reflected XSS is a non-persistent code injection attack where the injected script is reflected back to the user within the scope of the web application. Therefore, the attacker does not need the malicious script to be stored on the target server.

A typical example of reflected XSS shown on figure 8 and 9 can be described with a simple scenario. Attacker has discovered an XSS vulnerability on a website where the value of the parameter ‘name’ in the URL is displayed back to the user without encoding. So the attacker then crafts a following URL:

```
http://example.com/sample?name=<script>alert(‘XSS’)</script>
```

Figure 8. Example – Code snippet - Reflected cross-site scripting

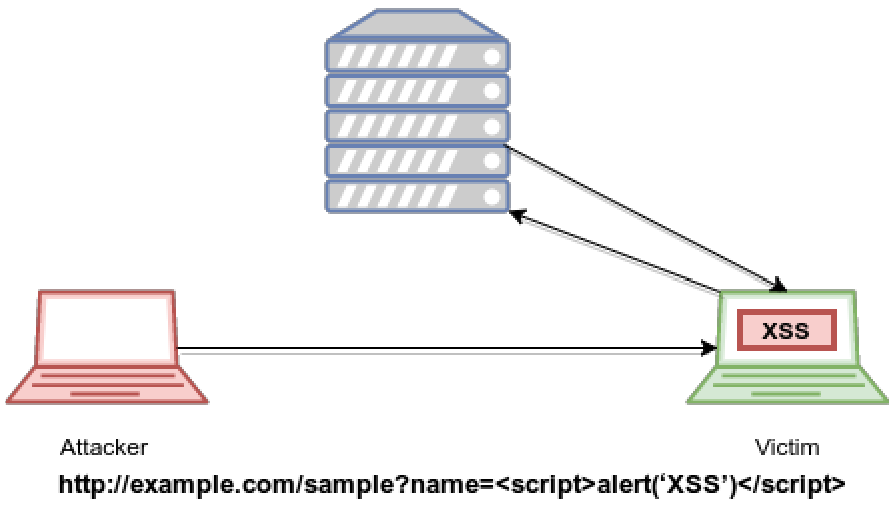

XSS
Attacker
Victim
http://example.com/sample?name=<script>alert(‘XSS’)</script>

Figure 9. Reflected cross-site scripting

There are many ways, how an attacker could entice the victim to click on that link. The link could be delivered to the user for example via an e-mail that looks legitimate but raises the victim’s interest enough to click on it out of curiosity. When the victim has clicked on the link, an alert box will be displayed with a message ‘XSS’ written on it. This script was not part of the original source code of this application, which means the attacker managed to successfully inject malicious code to the application which then executed within the victim’s browser.

Injecting a script that displays a message proves that the page is vulnerable, but does not serve any malicious purpose from the attacker’s perspective. Taking it one step further, the consequences of an XSS depend on the application and the possible attack vectors. It could result in session hijacking, stealing user credentials, malicious redirects, forcing user’s browser to download malware and so much more. The criticality of an XSS will depend case by case. From a user’s perspective, an XSS on a banking application that results in session hijacking can be much more critical than session hijacking on a web application where no personal data is kept.

3.2.3 Stored cross-site scripting

Stored XSS is a persistent code injection attack where the injected script is stored on the server of the application. The malicious script is served to the user whenever stored information is requested. To successfully execute the payload of a stored XSS, the first step is to locate where user input is stored and displayed back to the user. A typical scenario would be of an attacker discovering the possibility to inject scripts to the comment section of the web application by submitting the title and the message as on figure 10.

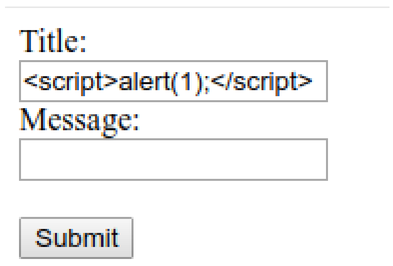

Title:
<script>alert(1);</script>
Message:

Figure 10. Stored cross-site scripting

After submitting the comment, the title is displayed back to the user without proper encoding for the output document:

```
<p>Title: <script>alert(1);</script></p>
```
Figure 11. Example: Code snippet - Stored cross-site scripting

This would result in code execution whenever a user is loading the page which contains the attacker injected malicious code.

3.2.4 DOM based cross-site scripting

Document Object Model (DOM) based XSS is a client side injection attack in which the malicious script never reaches the server as opposed to the previously described reflected and stored XSS vulnerabilities. Therefore, it is never included in the HTTP response sent from the server and it does not become part of the source code of the served page. The injected code is processed by a vulnerable client-side script which modifies the DOM in an unexpected way.

With DOM based injections, the payload could originate both from the parameter of a URL as shown in the example of reflected XSS or from an element in the output document as shown in the example of a stored XSS.

A typical example of a DOM based XSS would be of an application where the user can choose between the multiple tabs. After the page has already finished loading all of its contents, user can select between the tabs without any request made to the server. The content will be displayed to the user, depending on which tab the user has selected [38] . So when a user visits a page https://xss-game.appspot.com/level3/frame# and selects a tab, JavaScript function chooseTab() will be executed, which will display an image back to the user based on user's choice.

ChooseTab function:

```
<script>
function chooseTab(num) {
var html = "Image " + parseInt(name) + "<br>";
html += "<img src='/static/level3/cloud" + num + ".jpg' />";
$(‘#tabContent’).html(html);

window.location.hash = num;
. . .
</script>
```

Figure 12. chooseTab function to select between various tabs in the application

So if a user selects the tab ‘Image1’ the number of the selected tab will be displayed in the URL after the #-sign https://xss-game.appspot.com/level3/frame#1 and the image /static/demos/cloud1.jpg will be served back to the user. So to take advantage of that, the attacker could insert a payload after the #-sign, asking for an image1.jpg, and trigger alert(1) every time user is moving the mouse over the displayed image.

The payload would be executed when the attacker has managed to get the victim to visit the following link and the attack can be visually presented as shown in figure 14:

```
https://xss-game.appspot.com/level3/frame#1.jpg'
onmouseover="alert(1);">
```

Figure 13. Payload to trigger an alert

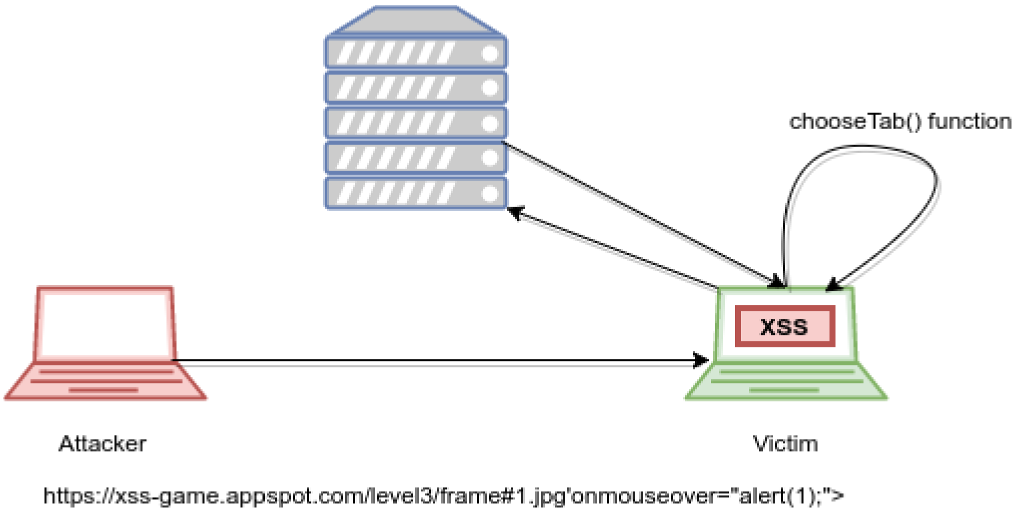

chooseTab() function
XSS
Attacker
Victim
https://xss-game.appspot.com/level3/frame#1.jpg'onmouseover="alert(1);">

Figure 14. DOM based cross-site scripting

The output with the attacker's payload on figure 15 is only displayed in the DOM and not in the source code of the application:

```
<div id=”#tabContent”>Image1</div>
“Image1”
<br>
<img src=/static/level/cloudmy1.jpg' onmouseover="alert(1);">
```

Figure 15. Payload displayed in the DOM

3.3 Server-side JavaScript

Based on the survey conducted by Stack Overflow, JavaScript was and still remains to be the most popular programming language in the current time [39] . Node.js and AngularJS are the most commonly used technologies [40] and Node.js the most used server-side JavaScript framework [41] . Nowadays JavaScript has the capability to produce full stack applications. The vulnerabilities that were introduced in applications due to client-side JavaScript have been extensively studied, but yet are commonly met vulnerabilities in web applications. While client-side JavaScript is bounded by the browser, server-side JavaScript can get a lot more dangerous than that if the same mistakes are made.

Based on the example shown at Black Hat “Server-Side JavaScript Injection, Attacking and Defending NOSQL and NODE.JS” [42] presentation, eval() function used to process user input can introduce a server-side JavaScript injection exactly like it would introduce a JavaScript injection in client-side JavaScript.

Web applications use the eval() function to evaluate expressions or execute statements. If the data parsed with eval() is without any validation, the application is vulnerable to a JavaScript injection. An example of server-side javascript injection (SSJI) vulnerability can be introduced with a snippet of code [43] shown in figure 16, in which user’s data is handled as a parameter for eval() when a POST request is made. In the following example, a block of JavaScript code executing on the server side to implement Node.js webserver parses JSON requests.

```
var http = require('http');
http.createServer(function (request, response) {
if (request.method === 'POST') {
var data = ''; request.addListener('data', function(chunk) {
data += chunk; });
request.addListener('end', function() {
var stockQuery = eval("(" + data + ")");
getStockPrice(stockQuery.symbol); …
});
```

Figure 16. SSJI vulnerability from handling user input as a parameter for eval

When the incoming request contains JSON data, it is evaluated in the eval() function. As the code runs on the server side the effects of a present vulnerability can be much more severe. If a legitimate JSON message gets sent:

```
{“symbol” : “AAAA”}
```

Figure 17. Example – code snippet of a JSON message

Then the same string is evaluated as:

```
eval({“symbol” : “AAAA”})
```

Figure 18. Example – code snippet of a JSON message passed into eval()

However, if the attacker has a way to send an arbitrary piece of JavaScript, such as shown below, the server code would execute it and return only the word success in the response body. This is the first indication that the server executed the arbitrary JavaScript, and more damage could be done.

```
response.end(‘success’)
```

Figure 19. Example – code snippet of a JSON message

Few examples of the possible damage could be a denial-of-service attack which forces the server to use 100% of the processor time in a loop and the server must be manually restarted before processing any of the other incoming requests. Another thing the attacker could do is to require fs module in order to read from/write files to the local system. The permission to write files on the system allows the attacker to write binary files which then can be executed, thus serving malicious exploit payloads.

Denial-of-service attack:

```
while(1);
```

Figure 20. Example – code snippet which can cause a DOS attack

Requiring file module to read the contents of a file:

```
response.end(require(‘fs’).readFileSync(filename))
```

Figure 21. Example – code snippet to read the contents of a file

Execution of binary files:

```
require('child_process').spawn(filename);
```

Figure 22. Example – code snippet to execute binary files

The execution of arbitrary code due to improper encoding of untrusted input resembles more to the SQL injection than cross-site scripting attacks. It allows access to the local machine where the server-side code is running without any social engineering trying to trick the user into clicking a link, or visiting a page with stored payload.

4 Electron Security

Web security issues can indeed occur in desktop applications, as on the 21th of Novermber in 2017 remote code execution vulnerability [44] was reported to be found in a text editor developed by Github – Atom. Atom is written using Electron framework and the way Github decided to mitigate XSS issues was by using Content-Security-Policy [45] to forbid all inline JavaScript. The mitigation was bypassed and malicious payload delivered and executed via embedding local file that contained JavaScript payload. The actual attack vector in this case came from the community-supplied packages which can be included to the application with a simple click on the package name and which in this case triggered the payload to execute.

As there are still a lot of the "unknown waters" or the less-explored areas in terms of vulnerabilities in desktop applications which use web technologies, the initiative was taken to produce a security checklist for Electron-based applications. Some of the contents of that list are presented and discussed in the following subsection 4.1.

4.1 Security checklist

Doyensec, an independent security research company has presented a security checklist [46] for Electron-based applications at Black Hat 2017 in the US. Security research company presenting the security issues relevant to Electron-based applications in such an influential event to the IT security community as is Black Hat, allows to presume the relevance and the timeliness of the topic. They explain how modern browsers have tried to enforce various security mechanisms from site isolation to other web security protections to prevent untrusted remote content from compromising the hosts. Electron, on the other hand, using Chromium's content module, which is only a part of the Chromium browser, is re-introducing some of the vulnerabilities that modern browsers would normally help to prevent.

Within the Electron Security checklist they present 13 bulletpoints to follow in development of an application. The majority of those are tied to the options offered by the WebPreferences object as the report focuses on the application-level design and the implementation flaws.

4.1.1 Node integration

Node integration is an essential feature for giving the native desktop like feeling in Electron-based applications. By default, Electron renderer processes can use Node.js by invoking APIs to execute code on the user's machine as shown in chapter 2.2.1. The risk in that can occur in the situation when untrusted content is rendered with node integration enabled. This can possibly lead to full-host compromise.

In Electron framework, one way to control access to the node modules from the renderer process can be by setting the nodeIntegration option shown in figure 23. NodeIntegration is a boolean type option available in the WebPreferences of a specific browserWindow instance.

```
let win = new BrowserWindow({
      "webPreferences": {
      "nodeIntegration": true
      }
});
```

Figure 23. Creating a new browser window instance with node integration enabled

Another location in Electron framework to have control over node integration shown in figure 24 is when BrowserView, currently experimental feature, is used to embed web content inside the BrowserView.

```
let view = new BrowserView({
      webPreferences: {
            nodeIntegration: false
      }
})
win.setBrowserView(view)
```

Figure 24. Creating a new browserView instance with node integration disabled

To embed guest content, such as a web page into the application, Electron supports the use of webview tags, where the content to be displayed can be specified in the source attribute. Webview creates a separate renderer process and has a nodeIntegration attribute in order to control access to requiring node modules. While previous two examples of node integration control flags were set on the main process, then webview on figure 25 allows to control node integration from the renderer process.

```
<webview id="foo" src="https://www.github.com/" style="display:inline-
flex; width:640px; height:480px" nodeIntegration></webview>
```

Figure 25. Remote content displayed via webview with node integration enabled

Launching a new browser window from the renderer process can be accomplished via window.open(). On figure 26, new instance of BrowserWindow is created, which will inherit the parent window's Webpreferences' options by default. Similarly to webview, window.open() is used from the renderer process.

```
window.open('https://example.com');
```

Figure 26. Opening a new window displaying remote content

While window.open() does not have an option to enable node integration as in the previous examples, it does have an option to disable it (figure 27).

```
window.open('https://example.com', '', 'nodeIntegration=0');
```

Figure 27. Opening a new window displaying remote content while node integration is disabled

Depending on the implementation of the application, enabling node integration should be done with caution, especially if content which is not entirely trusted is included to the application.

4.1.2 Sandbox

Sandbox is a feature that Chromium uses by default for every renderer process. It is a key security feature to contain the reach of exploits' from extending to the user's machine. For obvious reasons this defaults to False in Electron-based applications, as otherwise node integration would turn insignificant and there would be no access to the

Node.js JavaScript APIs. When sandbox is enabled, renderer processes can only make system changing operations by delegating those tasks to the main process via the IPC.

Sandboxing feature can be controlled from webPreferences options of a browserWindow instance. Depending on the value of sandbox option, node integration originally defaults to true, but if sandbox is enabled (figure 28), to false.

```
let view = new BrowserView({
      webPreferences: {
            sandbox: true
      }
})
win.setBrowserView(view)
```

Figure 28. Creating a new BrowserView instance with sandbox feature enabled

The sandbox option is recommended to be enabled whenever untrusted content is loaded in the browser window.

4.1.3 Preload scripts

Preload scripts, which can be specified for example within the webview tag, are the scripts that are instructed to load prior to any other scripts on the guest page. Despite disabling node integration for the renderer process or enabling the sandbox feature, preloaded scripts will have access to Node.js modules.

Improper use of preload scripts can result in the remote content bypassing the disabled node integration and sandbox features. Preload script has somewhat privileged position due to access to node modules, so it can re-introduce the application object via remote module and carry on communication via inter-process communication between the renderer and the main process as shown in the report on page 9.

4.1.4 Websecurity

WebSecurity is a flag used to control among other things, whether the same-origin policy [47] is enforced or not. In the Doyensec document, they have presented two possibilities to bypass the enabled same-origin policy restriction, by using window.location and eval() function.

Websecurity can be controlled from various locations, either applied in WebPreferences for the created BrowserWindow instance (figure 29):

```
Let win = new BrowserWindow({
      "webPreferences": {
            "websecurity": true
      }
});
```

Figure 29. Creating a new BrowserWindow instance with websecurity feature enabled

or in a webview tag with disablewebsecurity flag (figure 30).

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

Figure 30. Disablewebsecurity flag in webview element

4.1.5 Insecure HTTP connections

Serving application over HTTP instead of HTTPS is known to open application to Man-in-the Middle attacks where the attacker is able to observe and tamper with the user's unencrypted traffic.

Including remote content that is served entirely or partially over HTTP could have potentially elevated outcome when node integration happens to be enabled. Giving the chance to tamper with unencrypted traffic can result in remote code execution.

To audit applications for this problem, observing the protocols of the included resources is important. It is also possible to deny serving any included contents over HTTP by setting the allowRunningInsecureContent for the created browserWindow instance or for webview content.

4.1.6 Navigation to untrusted origins

Navigation to untrusted origins can occur, when content is added to the browser window or any other element used to display remote content. Allowing the use to navigate to other location than the one specifically displayed can lead to severe vulnerabilities. These vulnerabilities can escalate even more when not displayed in a sandboxed environment and/or allowed access to node modules.

To limit the unforeseen navigation flows, ‘will-navigate’ event in figure 31 can be used to detect that navigation is about to occur and restrict it if location is not allowed:

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

Figure 31. ‘Will-navigate’ event to detect if navigation occurs to unspecified location

The ‘will-navigate’ event will get emitted only when window.location is changing. If the change occurs due to change in window.location.hash, or it is an in-page navigation (user is not navigating to another website), the event is not emitted.

4.1.7 Popups in webview

Webview [48] contains numerous useful attributes, among which ‘allowpopups’ is used to enable guest page to open new windows. It must be set specifically, as by default webview does not allow this.

In web applications, popup windows are often used for advertising, or to deliver and execute JavaScript-based attacks. To allow popups from a webview containing untrusted content, user could be tricked into performing unwanted actions/unwanted clicks part of a ClickJacking/UI-redressing attack.

4.1.8 Shell.OpenExternal

Electron framework provides a shell module which contains functions related to the desktop integration. OpenExternal function among those, can be used to open an external protocol URL in the desktop's default manner [49] .

However, openExternal can pose a security risk which leverages to system compromise if user-supplied content can be injected without any proper validation in the application. In presence of an injection vulnerability like XSS, shell.openExternal() can be taken advantage of to launch local files or applications.

5 Methodology in collecting statistical data from Electron-based applications

Chapter 5 discusses the novel contributions made in this thesis by introducing the analysis method in subsection 5.1 used for gathering the statistical data and conducting the manual analysis.

Manual analysis of the applications aims to validate or disproof the hypothesis set in this thesis about whether there are many **o**pen-source Electron-based applications found to be vulnerable to the combination of XSS with enabled node integration as Electron being relatively new framework.

The explained method will be carried out on a set of Electron-based applications, to which the results will be presented in chapter 6 along with the conclusions.

5.1 Analysis method

The analysis of Electron-based applications conducted in this thesis presents itself in two separate stages shown in figure 32.

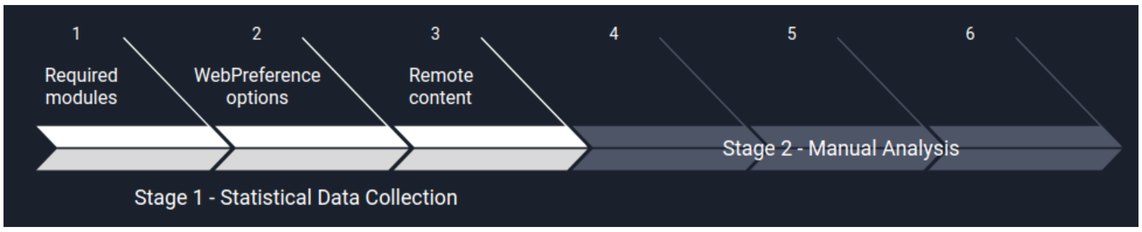

1
Required modules
2
WebPreference options
3
Remote content
4
5
6
Stage 2 - Manual Analysis
Stage 1 - Statistical Data Collection

Figure 32. Stage 1 – collection of statistical data

The first stage involves examining the source code of the applications in order to distinguish data related to three subjects of interest:

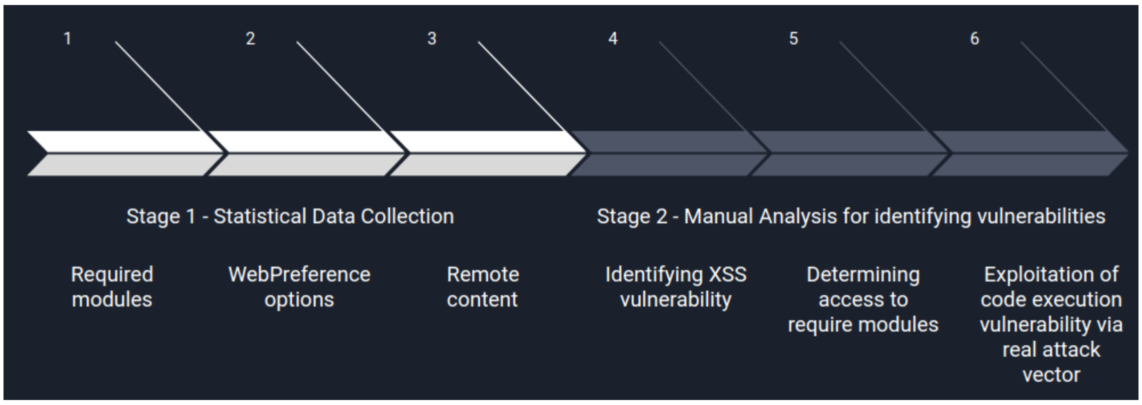
- Modules – we aim to gather statistical information on the required modules, which will be presented in three separate categories: main process modules, renderer process modules, and the modules available to both processes. The occurence of those modules will help to determine their significance for Electron-based application development and will help to describe the needs of an application functionality wise.

- WebPreference options – webPreferences available for the browser window instance help to determine the features set for the content displayed to the user. Depending on those options, content is allowed or denied certain behaviour or functionality. For the interest of the second step taken in the analysis to determimne vulnerabilities, particular interest in this section is node integration.
- Remote content – analysis determines to gather information on the remote content included to the application via loadURL method in browserWindow and browserView instance, via window.open() method and via webview tag. The particular interest in this section is the permissions given, by either enabling or disabling node integration, to remote content included to the application. The data gathered from this section aims to determine how commonly is remote content included via four described paths and whether or not developers have decided to enable node integration. This could bring insight to possible attack vectors through remote content.

The second step of the analysis is conducted on the same applications as examined during the first stage. As the first stage aims to collect statistical data on the applications, the second stage focuses on manual analysis (figure 33) of the applications with two pre-requisites that inevitably might reduce the number of subjects in the final dataset of applications. The first pre-requisite for the application to remain within the dataset for stage two is the successful execution on its respective platform. The second pre-requisite is that the execution of the application has to be successful without any modifications to the source code in order to modify it for its intended execution platform.

The aim of stage two is to manually examine successfully executed applications in desktop environment to validate or disproof in subsequent steps applications vulnerability to cross-site scripting attacks; to validate or disproof access to requiring

Node or Electron specific modules; to validate or disproof whether cross-site scripting vulnerability is allowed to evolve into code execution.


1
2
3
4
5
6
Stage 1 - Statistical Data Collection
Stage 2 - Manual Analysis for identifying vulnerabilities
Required modules
WebPreference options
Remote content
Identifying XSS vulnerability
Determining access to require modules
Exploitation of code execution vulnerability via real attack vector
Figure 33. Stage 1 and 2 – collection of statistical data and manual analysis

The expected result by the end of phase one is a collection of statistical data helping to describe Electron-based applications in the bounds of three above mentioned subjects. The expected result by the end of phase two, is to validate or disproof the hypothesis set in this thesis:

"*Many of the open source Electron-based applications are vulnerable to cross-site scripting, which due to misconfigurations or missing configurations in WebPreference options escalates to code execution.".*

Within the following three sub-sections we aim to further discuss the subjects part of this analysis and by which methods the data is gathered.

5.1.1 Subject 1 – Modules

Node uses 'require' module, available in the global scope of the application, to manage module dependencies. The modules required through out the application can mirror the purpose of the application and its possible functionality. To conduct the analysis in the scope of Electron framework, only modules listed as main and renderer process modules in the Electron API documentation were examined.

The approach to collect particular information from the source code was decided upon prior examination of the applications. It was determined that in order to collect required modules, the following three search strings would be used:

```
require(‘electron’)
electron.
from ‘electron’
```
Figure 34. Search strings to collect required modules

Using these three search strings the results would present modules part of the Electron API documentation. Examples of search results would produce following output:

```
const electron = require('electron')
import { app, BrowserWindow, Menu, dialog, shell } from 'electron';

const electron = require('electron');
const app = electron.app;
```
Figure 35. Possible results gathered from search strings

5.1.2 Subject 2 - WebPreference options

WebPreference options give the mechanism to control a variety of features for the created windows from the main process. For the interest of this thesis, which is directed towards application security, the webPreferences that deal with applications appearance, such as CSSvariables, scrollBounce, webaudio and others are not among those being observed. The preferences that are of the interest to this thesis are nodeIntegration, JavaScript, webview, sandbox, webSecurity, allowRunningInsecureContent, experimentalFeatures, preload, contextIsolation, nativeWindowOpen.

The collection of Webpreference options from the source code is done by identifying the created window instances, to which webPreferences are added directly, or included from a separate object.

As an example, a collection of webPreferences could be identified from the following construct on figure 36:

```
const win = new BrowserWindow({
    icon: path.join(__dirname, 'build', 'icon.ico'),
    titleBarStyle: 'hidden-inset',
    ...
    autoHideMenuBar: true,
    webPreferences: {
      preload: jsPath,
      nodeIntegration: false,
      plugins: true
    }
  })
```

Figure 36. WebPreferences set for a BrowserWindow instance

5.1.3 Subject 3 - Remote content

There are four common ways to include remote content to the Electron-based application which, if set with specific flags or options can give the remote content further access to the user's machine. This could turn out to be a security risk, if the content included is from an untrusted source or content that might fall under the control of an untrustworthy third party. Subject three involves examining four specific methods by which remote content could be included to the application. If remote content is included, it will be examined whether the content is first of all served over https or http, and if the content has granted further access via enabled node integration.

The following four sections will discuss including remote content via BrowserWindow and BrowserView loadURL method, via webview tag and via window.open() method. Each of the four will include a demonstrative example on content's permissions with enabled node integration to distinguish the difference on content's permissions.

BrowserWindow

BrowserWindow is the main process API used to create and control browser windows. It carries a variety of options to specify window size, appearance, as well as webPreferences options, which include nodeIntegration, sandboxing, session handling, JavaScript support and more [50] . Each of those options can be set for a specific window instance and loadURL instance method is used to specify the remote URL or local HTML file to be loaded.

As taken from the security checklist, having node integration enabled for a renderer process, where unrusted remote content is included, can be a harmful combination.

Following versions of npm, node and electron were used in the example for BrowserWindow as well as BrowserView, webview and window.open():

```
node --version
v8.6.0

npm --version
5.3.0

electron --version
v1.7.8
```

Figure 37. Versioning information for Node, npm and Electron

and the application structure for the application project:

```
/electron-application
--package.json
--index.js
--index.html //index.html file for webview and window.open() examples
```

Figure 38. Application structure for the application project

Within the index.js file (figure 38), a BrowserWindow instance is created with nodeIntegration set to True (figure 39) in which remote content is displayed via loadURL instance method. For the purpose of this and the following examples, remote content was set up and served from a virtual machine.

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

let win

function createWindow() {
    win = new BrowserWindow({width: 800, height: 600,
'webPreferences': {nodeIntegration:true} })
    win.loadURL('http://192.168.56.1/index.html');
}

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

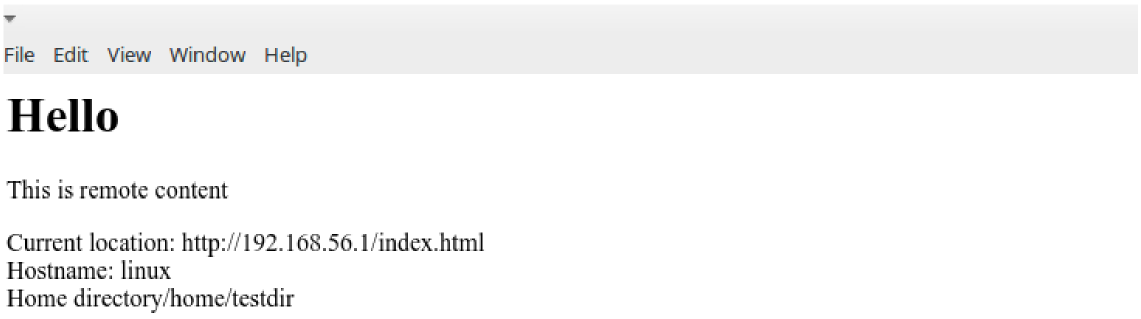
Figure 39. Remote content displayed in the browser window with enabled node integration

The file served from remote location contains the following JavaScript code (figure 40), which needs access to node APIs in order to require the 'os' module. The home directory and the operating system platform used on the user's machine will be displayed back to the used as a result.

```
...
<script>
    var currentLocation = window.location.href;
    document.getElementById('status').innerHTML = 'Current location: '
+ currentLocation;
    var os = require("os");
    var hostname = os.platform();
    var homedir = os.homedir();
    document.getElementById('host').innerHTML = 'Hostname: ' +
hostname + '</br>' + 'Home directory' + homedir + '</br>';
</script>
```

Figure 40. Contents of the remote content served from /index.html

When the Electron-based application is executed, the following outputis displayed:



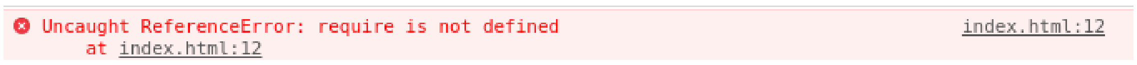
Hello

This is remote content

Current location: http://192.168.56.1/index.html
Hostname: linux
Home directory/home/testdir

Figure 41. Remote content displaying hostname and home directory

If node integration would have been disabled, only the current location would be displayed since requiring modules would not be available for the remote content.



```
Uncaught ReferenceError: require is not defined                    index.html:12
      at index.html:12
```

Figure 42. Node integration disabled – Uncaught ReferenceError

The following scenario, where node integration is enabled, could pose a risk to the Electron-based application, if content included is either untrusted, or if there is a exploitable content injection vulnerability which then automatically escalates to remote code execution on the user’s machine.

BrowserView

BrowserView is an Electron specific main process API to embed additional web content inside the browser window. It was implemented to be the alternative to the webview tag and at the moment is still considered experimental.

The same combination observed in the previous example can also pose a risk when the BrowserView element is used to include remote content to the renderer process with enabled node integration.

For this example, same environment was used as in the previous example, with the only modifications made to the index.js file on figure 43:

```
const {app, BrowserView, BrowserWindow} = require('electron')
const url = require('url')
const path = require('path')

function createWindow() {
    let win = new BrowserWindow({width: 800, height: 600,
'webPreferences': {nodeIntegration:true} })

    let view = new BrowserView({
          webPreferences: {
            nodeIntegration: true
          }
    })
    win.setBrowserView(view)
    view.setBounds({ x: 0, y: 0, width: 300, height: 300 })
    view.webContents.loadURL('http://192.168.56.1/index.html')
}

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

Figure 43. Index.js file – displaying remote content from BrowserView instance with enabled node integration

Instead of serving remote content from within the BrowserWindow, it is now served from the BrowserView loadURL instance method where node integration has been enabled.

When the Electron-based application is executed, the same type of content will be displayed to the user as before. When including content via BrowserView loadURL, multiple combinations to enable/disable node integration from BrowserView and BrowserWindow are available. As visible from the table no. 1 below, the only combinations that enable node integration for included resource are explicitly those where it has been set to true in the BrowserView webPreferences.

Table 1. Node integration - BrowserWindow and BrowserView WebPreferences.

BrowserWindow	Default = True	False	False	True	True
BrowserView	Default = True	False	True	False	True
Access	Access	No access	Access	No access	Access

The content included via BrowserView should be trusted when node integration is enabled, as otherwise untrusted third party has access to user's machine and a chance to take advatnage of it. Otherwise, webview is a good way to enable trusted content to access node APIs while the rest of the content should not have that privilege.

Webview tag

Webview is used to create hybrid mobile and desktop applications. In Electron it is used to embed guest content to a page in the Electron-based application. It creates a new separate process, and the permissions can be managed separately from the permissions of its parent process.

WebView possesses nodeIntegration flag which allows to specifically allow or deny it. One browser window can contain multiple webviews, therefore multiple new processes with separate permissions can be created.

In order to enable node integration, webview can be specified on the page in the following way:

```
<webview src="http://192.168.56.1/index.html"
nodeintegration></webview>
```

Figure 44. Displaying remote content via webview element

As visible from the table no. 2 below, remote content will have node integration enabled only if it is enabled for the renderer process in which webview is used and explicitly allowed for the content served from webview. When browser window is not granted with node integration, using webview tag is restricted [51] .

Table 2. Node integration - BrowserWindow and WebView

BrowserWindow	**Default = True**	**False**	**False**	**True**	**True**
WebView	**Default = False**	**False**	**True**	**False**	**True**
Access	No access	No access	No access	No access	Access

If webview is used from within a BrowserView, it can also be verified that content within webview can use node APIs only if node integration is enabled for the BrowserView instance and nodeintegration flag set for the webview. That is despite the disabled node integration for the BrowserWindow.

Table 3. Node integration - BrowserWindow, BrowserView and Webview

Browser Window	Default = True	F	F	F	F	T	T	T	T
BrowserView	Default= True	F	F	T	T	F	F	T	T
WebView	Default = False	F	T	F	T	F	T	F	T
Access	No	No	No	No	Yes	No	No	No	No

When using the webview tag, it must be verified whether the content included is trusted or untrusted and assured into which context remote content is included.

Window.open()

Window object provides a method open() to load a resource specified by a URL or local file path [52] . With every call to window.open(), a new instance of BrowserWindow is created.

By Electron documentation, content included via window.open() will inherit the parent window's webPreferences option values by default. For example, if node integration has been enabled for the parent window, where content is included via window.open, then node integration for that resource is enabled.

Similarly to webview tag, window.open() method had access to setting nodeIntegration flag, but due to reported issue #4026 [53] in github, which allowed to override the disabled node integration for the browser window, it was removed. This could have permitted the attackers to re-enable node integration to deliver their payload and access user's machine, if content injection vulnerability was found from the application.

Nevertheless, window.open implements the node integration flag shown in figure 45 in order to deny node integration for the remote content.

```
<script>
     var newW = window.open('http://192.168.56.1/index.html', '',
                         'nodeIntegration=0');
</script>
```

Figure 45. Displaying remote content in a new window with node integration disabled

That can prove to be a useful feature to explicitly deny node integration when content is included.

5.1.4 Subject 4 - Vulnerability to cross-site scripting attacks to evolve into code execution

The second stage of this analysis is conducted by manual examination of applications. Manual examination is presented in three steps that will be followed along with every examined application.

First step is to determine whether the application is vulnerable to cross-site scripting attacks by including user input to the output document without proper encoding. The test strings used in order to determine that are identical to the test strings used for identifying XSS in web applications. If application is identified to be vulnerable to XSS, the manual analysis is continued in step two.

Step two is to determine the state of node integration option for the browser window where XSS flaw was determined to be present. This will help to understand whether the

XSS flaw has the capability to evolve into code execution by being able to require node modules. The step two can be completed in two ways. The first includes the review of collected statistical data on BrowserWindow instance from phase one connected to the subject analysed. The second includes using payloads that attempt to require node modules.

If node integration is determined as enabled for the browser window instance, step three is attemped by exploiting the vulnerability of XSS evolving into code execution. As the end result, all payloads used on the vulnerable applications will be presented, along with their respective attack vectors and impact.

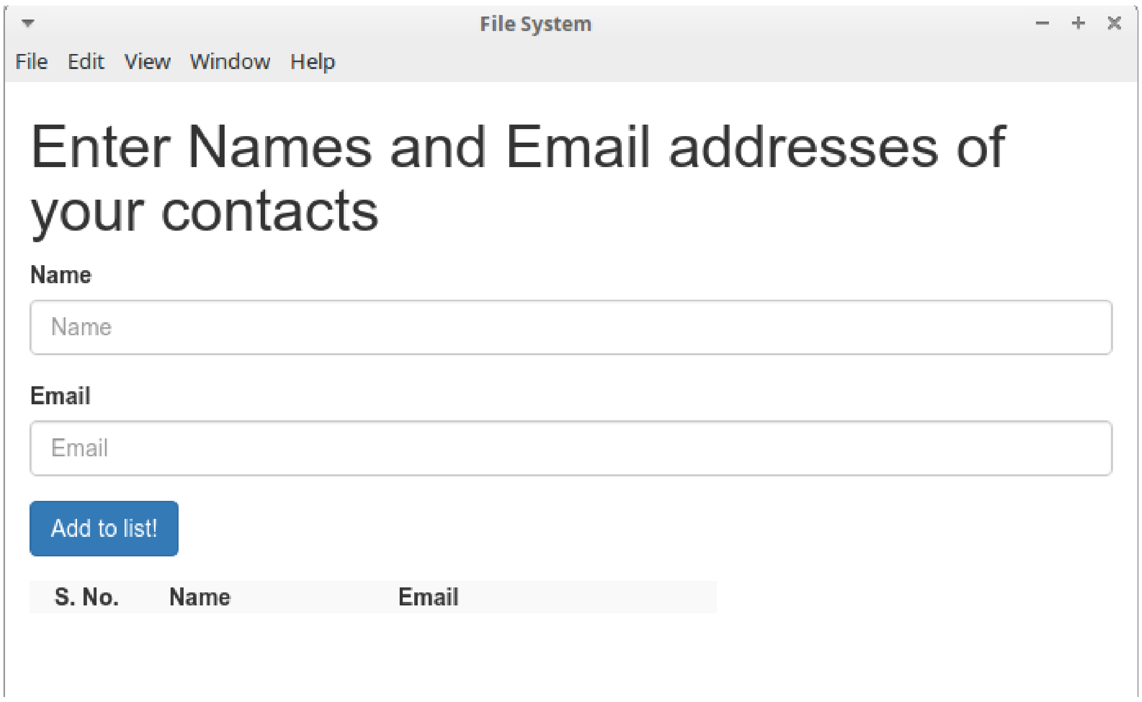
To showcase the "perfect scenario" in following through steps one to three in phase two, an example vulnerable application where XSS escalates to code execution is presented. The source for this application is taken from an example presented in Tutorialspoint to learn file handling in Electron-based applications [54] . The example was determined as vulnerable.

Following the same application structure as in the tutorial, application contains three files:

```
--main.js
--index-html
--view.js
```

Figure 46. Application structure

The main.js file will be creating a new browser window where the contents of index.html will be displayed (figure 47). View.js manages the file created at the start of the application and serves user input back to the user interface.



Enter Names and Email addresses of your contacts

Name

Name

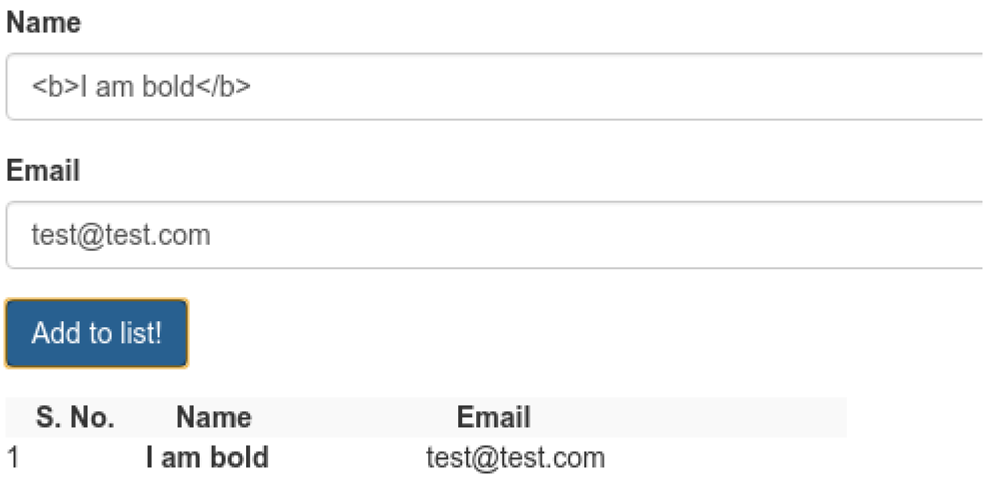
Email

Email

S. No.	Name	Email

Figure 47. For to submit name and e-mail

When the button “Add to list” is clicked, user input is submitted and stored in the local file, then loaded and displayed back to the user (figure 48). Already with a simple test string it is possible to detect that user input is not encoded for the output document and interpreted as HTML code.



Name

<b>I am bold</b>

Email

test@test.com

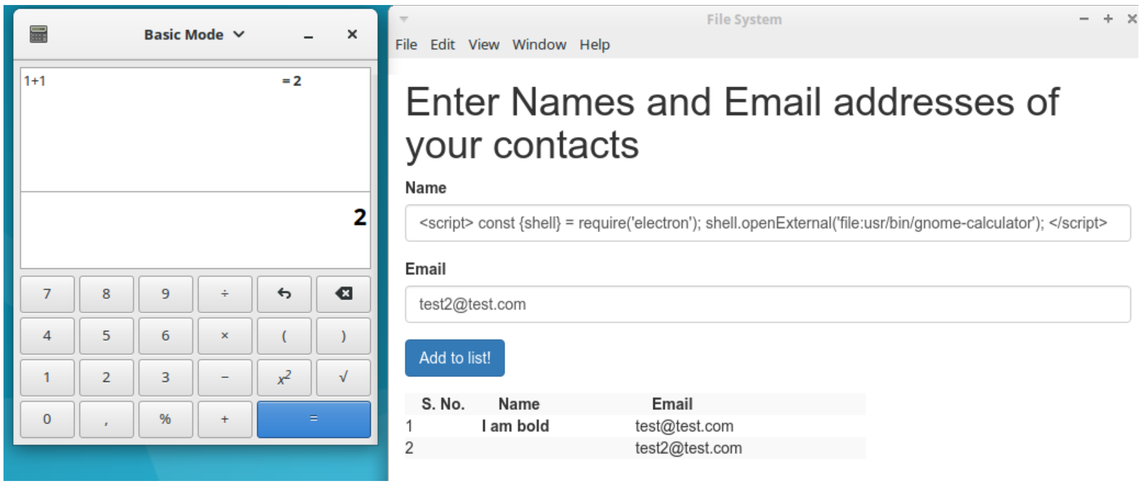
S. No.	Name	Email
1	**I am bold**	test@test.com

Figure 48. Dialog with user input displayed in the output document as bolded

What makes cross-site scripting vulnerability evolve into code execution in this case is that node integration has not been disabled. To take advantage of that, the following payload on figure 49 is used to require shell module and open the calculator application (figure 50) from the user's machine:

```
<script>
const {shell} = require('electron');
shell.openExternal('file:usr/bin/gnome-calculator');
</script>
```

Figure 49. Payload used to open a calculator via shell.openExternal method



Enter Names and Email addresses of your contacts

Name

<script> const {shell} = require('electron'); shell.openExternal('file:usr/bin/gnome-calculator'); </script>

Email

test2@test.com

Add to list!

S. No.	Name	Email
1	I am bold	test@test.com
2		test2@test.com

Figure 50. Shell.OpenExternal method used for opening calculator via payload.

This example represents the perfect scenario following the three steps to determine the presence of XSS, to determine the enabled state of node integration option and the successful attempt to exploit this combination.

6 Analysis and results

This chapter presents the analysis results gathered from examining Electron-based applications. The analysis conducted on each application consisted on following through the steps presented for stage one and stage two of the method discussed in chapter 5.

The first stage included gathering the information on the occurence of main and renderer process modules that were required in the applications. It also included observing a set of WebPreference options for every new browser window instance created and a set of four distinct methods for including remote content in the application. The second stage required manual analysis of the successfully executed applications in three subsequent steps. Firstly, applications were examined for the presence of XSS vulnerability. Secondly, if the presence was determined, steps were taken to identify whether node integration was enabled for the browser window with the XSS vulnerability. If steps one and two were determined as present, third step was taken to exploit the XSS vulnerability to demonstrate the escalation to code execution.

Based on the outcome of stage two results, ~37% of the applications were determined to be vulnerable. The hypothesis raised in this thesis about many of the open source Electron-based applications being vulnerable to XSS, which evolves into code execution, was therefore validated to be true. Further results on that will be discussed in the section of this chapter presenting the results of stage two.

Following subsection will present the selected dataset on which the analysis was carried out along with statistical data and the conclusions.

6.1 Dataset

The dataset of applications selected for this study consists of publicly available open-source Electron-based application projects. As a source from where to derive the projects, Github - a web-based platform for hosting open-source software project repositories was chosen. To narrow down on the Github repositories to serve the interest along with the aim of this thesis, a search phrase “electron” was provided: https://github.com/search?utf8=%E2%9C%93&q=electron&type [Search phrase on github: electron] [55] The search phrase resulted in 24,678 repositories in response. However, it must be noted that not all of the results were the projects that could be counted to belong to the final dataset. Projects related to chemistry and physics, which due to the relatable phrase in these fields, “electron”, was included into the resulting repositories, do not belong to the scope. Before including the application to the final dataset, it was assured that the subject is an actual Electron-based application and did not belong to study of physics nor chemistry. If the application was not identified as such, it was discarded.

Another criteria contributing to the selection of the final dataset was developed based on the observation of the projects’ Wiki page or other relevant and descriptive information on the projects’ repository that would hint the use of the application and its features. A portion of the Electron-based application projects were clearly developed to teach the framework itself or to bring samples of a specific usecase. Based on the assumption of code samples teaching the basic knowledge and therefore commonly discarding the security aspect, those types of projects were identified to be of no interest to this thesis. Applications selected to the final dataset, based on the observation and the judgement of the author of this thesis, were applications with clear functionality developed for a legitimate userbase.

Overall 30 applications were selected and examined as part of this analysis. Applications observed were built for Windows, Linux, and Mac OS platforms. Majority of the applications however were executed on Linux platform. The manual analysis for the detection of XSS that could evolve into code execution was conducted only on the applications that were executed successfully on their respective platforms, without any modifications to their source code. Based on that requirement, 19 applications out of the 30 were analyzed in phase two.

The applications observed could be described by their functionality to belong to the following categories:

- chatting/communication applications
- note taking/writing applications
- GUI for an existing server data
- host management applications
- desktop music player applications
- text/markdown editors

6.2 Modules

This section presents the results gathered from stage one of collecting information on the main and the renderer process modules required in each of the examined applications. Tabel no 4. presents each of the modules identified from 30 applications based on the number of occurences and how many applications required every particular module.

Table 4. Main and renderer processes modules required in applications

Main process modules			**Renderer process modules**		
Module	No. of occurences	No. of applications	Module	No. of occurences	No. of applications
app	102	30	remote	94	21
BrowserWindow	73	30	ipcRenderer	65	24
dialog	41	16	webFrame	2	2
Menu	40	23			
ipcMain	21	17			
tray	12	10			
MenuItem	6	5	**Both**		
globalShortcut	8	6	shell	58	21
autoUpdater	6	3	clipboard	11	6
powerSaveBlocker	2	2	crashReporter	7	6

session	2	2	screen	7	4
systemPreferences	1	1	nativeImage	2	2

The main process modules with the most occurences present the modules that control the life time of the application - app, and the creation and management of browser window instances - BrowserWindow. Being essential for every Electron-based application, this was the expected outcome. From the rest of the main process modules, it can however be determined that dialog module, to open and save files, as well as display message windows is a highly used functionality which assumably provides the desktop like experience that Electron aimed for. More than half of the applications also used the Menu module to create custom menus reflecting the needs of the application for saving/opening files, copy/paste functionality, links to remote resources and more. The last module used by more than half of the examined applications was ipcMain, representing the importance of carrying out communication from the main process to the renderer process.

Electron API documentation presents seven available modules for the renderer process, which are not available for the main process. From the seven, three occurred in the examined 30 applications: ipcRenderer, remote and webFrames. Remote module presented with 94 occurencies in 21 applications shows predominantly the importance of renderer process accessing modules available for the main process. This can allow to come to assumption that most of the Electron-based applications' functionality is implemented in its main process modules.

As for modules available for both, the main and the renderer process, file and URL management via shell module, for example to open URL via openExternal method, is used quite often as being present in 21 applications and required 58 times. To keep track of the occurred crash reports, the reports in most of the applications observed were saved locally or directed to a remote server by using crashReport module. Crash report module was counted to have 7 occurencies within 6 applications. One of the particular locations occuring more than once for crash reporting was connected with a github project, also Electron-based project, https://electron-crash-reporter.appspot.com/ created by developer from Google.

6.3 WebPreference options

The second subject of the first stage analysis was about gathering information on the Webpreferences used in creating new BrowserWindow instances. Table no 5 presents the WebPreferences set to TRUE, FALSE and to its default value, by not including the options to the created window instance.

Table 5. WebPreferences set for browser window instances

	TRUE	**FALSE**	**NOT SET (default)**
Preload	used 6 times by 4 applications out of 30 in total		
Node integration	5	6	41 (true)
JavaScript	-	-	52 (true)
Plugins	3	-	49 (false)
WebSecurity	-	1	51 (true)
allowRunningInsecure Content	1	-	51 (false)
experimentalFeatures	1	-	51 (false)
Sandbox	1	-	51 (false)
AllowDisplayingInsec ureContent	1	-	51 (false)

It was observed that majority of the applications tends to leave WebPreferences to their default values. Whether it is done by not being aware of the WebPreference options Electron framework provides or willingly, was not studied within this research.

However, if to focus on the WebPreference options that were specifically set to either TRUE or FALSE, it can be observed that node integration which already defaults to TRUE was set to TRUE manually in five cases. This can hint that developers are not being aware of the the default values assigned to WebPreferences related to node integration, as this did not occur with any other WebPreferences.

As other WebPreference options were not used more than once or twice, the conclusions on them can be left for the future work where dataset holds larger number of subjects.

At the moment, the only conclusion to come to can be that these WebPreferences were not needed for the observed applications in the dataset, which might limit the outcome and precision of the data gathered in this section.

6.4 Remote content

Third subject of the stage one analysis gathered information on the remote content included to the application via browserWindow, WebView, window.open() and BrowserWindow. As it was observed, majority of the applications also used shell.openExternal method (total of 73 times by 20 applications out of 30) to open external links with user's default browser. For the interest of this thesis in remote content, occurencies of shell.openExternal are also included to this section.

It was observed that remote content was included via loadURL for BrowserWindow total of 5 times, where 3 out of 5 times the content was served over http (table no. 6). Remote content served via WebView occurred only once and the content was not granted the access to node APIs. All the links provided to include remote content were static, and did not include parameters taking user input.

Table 6. Remote content included via http/https

	BrowserWindow	WebView	Window.open	BrowserView
http:// + nodeIntegration: False	2	-	-	-
https:// + nodeIntegration: False	2	-	-	-
http:// + nodeIntegration: True	-	-	-	-
https:// + nodeIntegration: True	-	-	-	-
http:// + nodeIntegration: default	1	-	-	-
https:// + nodeIntegration: default	-	1	-	-

Shell module was required total of 58 times in 21 applications, and in majority of the times to open external links served over https, by using openExternal method (table no. 7).

Table 7. External links, served over http/https by using shell.openExternal

	Number of occurences	**Number of applications**
shell.openExternal('https://*')	57	17

shell.openExternal('http://*')	15	10
shell.openExternal('http://*/https://*')	1	1

The conclusion based on the data of the last subject analysed in stage one is that remote content is not ofted included via loadURL from BrowserWindow and BrowserView, neither via window.open method and webview tag. Most of the content displayed to the user is included from local files. Instead of opening remote content via window.open as a new browser window, shell.openExternal is preferred as it opens the content within user's default browser. As the URLs opened via shell.openExternal are often hidden behind descriptive titles in the menu bar, this can be an effective way to trick user visiting a malicious site.

6.5 Applications vulnerable to cross-site scripting evolving into code execution

This chapter presents the results from the stage two of the analysis in the essence that was discussed in chapter 5. By the pre-requisites introduced in subsection 5.1 discussing the dataset for stage two, a subset of the original dataset went through the manual examination. By the pre-requisites, only the applications that were successfully executed on their respective platforms without any modifications to their source code to make them compatible, were selected for manual examination. The original dataset consisting of 30 applications, due to the pre-requisites mentioned above, was reduced to 19 applications.

Stage two was conducted by following three subsequent steps, from which second and third step presumed the expected result from the previous step. First step in manual examination was to determine applications vulnerability to XSS flaws. If application was identified as vulnerable, step two was proceeded with, else application was identified as not vulnerable and further step were not taken. Second step attempted to identify whether node modules were available from the browser window where the XSS flaw was identified. This was either determined by reflecting back to the statistical data gathered prior to stage two, or by using a modified payload which attempted to require the node modules. Based on the results, access to node modules was either determined or proceedings to step three were not subsequently followed any longer. Step three attempted, based on the identified XSS flaw along with access to node modules, to take advantage of this vulnerable combination in order to craft an exploit payload and produce the possible attack vectors and evaluate the impact of the flaw on the application.

As the result of the three step process, ~37% of the applications were determined to be vulnerable to the combination of XSS flaw being present in the browser window with access to node modules, due to which XSS was allowed to evolve into code execution on the user's machine. All the owners of vulnerable applications have received reports on the identified issues in order to produce fixes as quickly as possible.

Further insight of the vulnerable applications identified is given to the extent that allows to keep the anonymity in order to give the developers the time to fix the issues before disclosure.

6.5.1 Overview of the vulnerable applications

All vulnerable applications are open-source applications selected from Github. In Github, popularity of a project is represented with number of stars it has received from the user's community. Vulnerable applications were determined to have received stars from 40 up to 5000 community members, where the higher the count, the higher the popularity. Applications were observed to have 1-20 contributors and an average of 52.2 open issues reported per application during the time of analysis.

The 7 vulnerable applications created total of 11 BrowserWindow instances. All 11 had node integration enabled, not due to setting it to a specific value, but leaving it to its default value (default value: True) by not specifiying webPreference options. Also any other WebPreferences were not explicitly set to a value either.

Via shell.openExternal() method, mixed content served over http:// and https:// was opened in user's default browser, and no remote content got included via loadURL from browserWindow and browserView. Neither of the 7 applications made use of the webview and the window.open() method.

6.5.2 Attack vectors

The attack vector, path which the attacker can use to gain access to the victim's computer, could be categoried into three for the seven vulnerable applications by their intended functionality.

Attack vector no. 1 – Attacker crafts a file with a specific payload to trigger when user opens the file in the application. As a result, user's file contents can be sent to the remote server hosted by the attacker.

Attack vector no. 2 – Attacker has gained access to the web application which has implemented an Electron-based desktop application that synchronizes with any edits made in the web version. Gaining access to the web version of the application could occur through a vulnerability or if the attacker belongs to the same group of users with the victim, sharing the data in the application. Attacker can deliver the payload to the victim through a shared data field in the web application.

Attack vector no. 3 – Two or multiple users are sharing the same source for the data displayed on the vulnerable GUI built on Electron. Attacker is either one of those user's with the access or has managed to gain access to edit the data source through a vulnerability. Payload is delivered and executed on the victim's machine due to data field value being displayed to the user in the output document without proper encoding.

5 applications out of the 7 that were found vulnerable, were found to be corrresponding to vector no. 1 as shown in table no. 8.

Table 8. Number of applications corresponding to attack vectors no. 1-3

Attack vector no. 1	**Attack vector no. 2**	**Attack vector no. 3**
5	1	1

6.5.3 Step one to step three validation and results

Applications were identified as vulnerable via three subsequent steps, where each step required a different payload to prove that certain situation was present in the application. Payloads needed to prove vulnerability to XSS; they needed to validate the state of node integration in the browser window with the identified security flaw; and finally payload needed to be produced that could take advantage of the vulnerable combination of XSS with enabled node integration by following a realistic attack vector.

The payload that proved to be successful to determine the presence of XSS in all seven applications was a test string to present user input with bolded formatting in the document. This allowed fast affirmation based on the applications observed behaviour.

```
<b>My words in bold</b>
```

Figure 51. Test string to determine the presence of XSS

If the payload was treated in the application as program code, it was displayed back without the <b>-tags, in bolded formatting as "**My words in bold**". If the payload was not treated as part of the program code, it would have been partially or completely displayed as it was submitted.

By the end of first step in phase two, 8 applications were determined interpreting user input for the output document as program code, therefore identified as vulnerable to XSS. Second step was proceeded with 8 applications from total of 19.

Second step aimed to identify the state of node integration for the browser window instance. This required a test string that would attempt to require node modules and allow to affirm the success of the step two from applications behaviour. Following the test string from figure 52, requiring the 'os' module to display the hostname and the homedirectory of the user's machine in the alert box proved to be successful.

```
<onmouseover="alert(1)"> <s onmouseover="var os = require('os'); var
hostname = os.platform(); var homedir = os.homedir(); alert('Host:' +
hostname + 'directory: ' + homedir);">Hallo</s>
```

Figure 52. Test string to determine the state of node integration

If node modules were not available to be required, an error would be produced in the console and/or alert box would not display the expected values. 7 out of the 8 applications were identified to have node integration enabled for the browser window instance with XSS flaw as shown in table no. 9. Based on the results, step three was pursued with 7 vulnerable applications out of the original 19. All seven were identified to have a real attack vector from which one is presented in the subsection 6.5.4

Table 9. Results of the second phase for steps one to three for applications no. 1-19

	Category	**Step 1**	**Step 2**	**Step 3**
Application no. 1	Note keeping application	x	x	x
Application no. 2	Markdown editor	x		
Application no. 3	Text editor	x	x	x
Application no. 4	GUI for the server	x	x	x
Application no. 5	Markdown editor	x	x	x
Application no. 6	Markdown editor	x	x	x
Application no. 7	Markdown editor	x	x	x
Application no. 8	Note keeping application	x	x	x
Application 9-19	Various	-	-	-

All findings were reported to the lead developers of each project and two out of the 7 reports got an immediate response within hours, where the vulnerability was patched within the next two days. Only one report has not gotten any response after the developers were contacted via e-mail and issue created in Github. It can only be assumed that the developers do not acknowledge the issue in their application as they took steps to cover it up by closing the issue in Github without a fix. Eventually full disclosure on the issue will follow, to retain users from using the vulnerable version of the application.

All reports presented a first-time identification of code execution in all vulnerable applications and the responses to the reports gathered appreciation from most of the developers. As code execution had not been identified in those applications before, it can be asserted that it is an issue which developers are not aware of.

6.5.4 XSS to code execution – showcase on a vulnerable markdown editor

This subsection will present a successful exploitation of XSS to code execution on a markdown editor identified to be vulnerable during the time of the analysis. The exploitation of the vulnerability followed the attack vector number one, in which the attacker tricks the victim to open a malicious file in the vulnerable application. The pre-requisites of conducting the attack involved attacker creating a file that contains the attack payload and delivering that file to the victim.

The attack payload that took advantage of the attack vector was presented to the victim as a legitimate README.md file. Payload in that file, shown in figure 53, spawns a shell on the background and executes the command to make a connection to 192.168.8.100:1337. After the connection is established, the content of '/etc/passwd' is sent to the specified location.

```
<s <onmouseover="alert(1)"> <s onmouseover="const exec =
require('child_process').exec;
exec('nc -w 3 192.168.8.100 1337 < /etc/passwd', (e, stdout, stderr)=>
{
if (e instanceof Error) {
console.error(e);
throw e;
}
console.log('stdout ', stdout);
console.log('stderr ', stderr);
});alert('1')">Hallo</s>
```

Figure 53. Attack payload for sending the contents of '/etc/passwd' file to the attacker

It is evident that from the perspective of traceability, showcased payload reveals the remote location to which the file is sent to, but to showcase the seriousness of the vulnerability it is considered to be the right fit.

For the attacker to receive the contents of the file, it is necessary for the attacker to be listening on the port the malicious payload will try to connect to.

```
nc -l -p 1337 > passwd.txt
```

Figure 54. Command to listen on the port and receive the contents into passwd.txt file

After these conditions have been met (1. attacker has crafted the payload and delivered it to the user; 2. attacker is listening on the port the triggered payload will try to connect to;) comes the period of waiting until the victim opens the malicious file. Once the victim has opened the file, malicious payload triggers and the content of '/etc/passwd' is sent to the attacker.

The success of this particular attack, code execution on the victim's machine, reveals confidential information. With a modified payload however, the effect on the user's machine could be anything from the imagination of the attacker. The risk rating of that vulnerability is evaluated as high.

6.5.5 Risk

By the OWASP risk rating methodology, in order to estimate the risk, likelihood and impact should be evaluated.

Estimating likelihood involves evaluation of the skill level needed for the attackers as well as evaluating the ease of discovery and exploitability. To identify the presence of the vulnerable combination, only simple steps were followed which allows to state that the vulnerability is easily discovered and easily exploitable by the actors with little to some programming skills. To have more harmful effect more advanced programming skills are required in understanding Node and Electron specific features. One of the possible effects of that was shown in chapter 6.5.4.

Estimating the impact includes evaluation of data confidentiality and possible losses to the business and its users. As each vulnerable application identified resulted in system compromise and code execution, all factors – confidentiality, integrity and availability were affected.

According to these terms, it can be evaluated to be highly likely that the vulnerability is discovered by the attackers, as identification is considered very trivial proven by the three steps taken in phase two of the analysis, which worked in all seven cases. The business impact can vary depending on the application, but each of them would possibly suffer from reputation damages, and as the result of an existing attack vector the very least disclose private information.

Therefore, XSS in combination with enabled node integration is estimated to have risk level of high.

6.6 Suggestions

The identified 37% of vulnerable applications is the evidence of first of all Electron being relatively new framework, where exploring security issues is still in its baby shoes. Electron is built as a framework where web developers can apply their knowledge of web application development on building a desktop application. From security perspective, this makes an assumption that web developers are aware of the possibile vulnerabilities in web applications, therefore know how to prevent them in Electron-based applications. As it appeared from the analysis results, developers continue to do the same mistakes as those that result in a vulnerability in web applications. In suggestion of improving the security of Electron applications, the following could be done:

- to raise overall awareness on web security issues like XSS and the preventative measures;
- to raise awareness that Electron, much like web applications, is open to web vulnerabilities, therefore all caution should be taken when handling any untrusted input;
- derived from the analysis results, where neither of the vulnerable applications set any webPreferences, the assumption can be made that developers might not be aware of those options. Therefore following is suggested: with every created browseWindow and browserView instance, a mandatory value assignment to each of the boolean webPreference options or a subset of them could be implemented. This might bring more awareness on the options and their function in the application;
- as Electron-based applications rely on using node modules for their functionality, a suggestion is to default the node integration to false. Having node integration disabled by default is certainly more inconvenient for the

developers as they would then need to open the documentation and identify how to enabled it. But on the other hand, it will encourage the developers to study the framework and understand the capabilities of that feature;

- and as a final suggestion, Electron applications could make use of being able to control the required modules via configuration file. In case of an XSS vulnerability in the application, this would help to prevent the attacker being able to require new modules which are not defined in the configuration file. Therefore, suppressing the number of harmful actions due to unavailable functionality to the attacker.

7 Conclusion and future work

In this thesis, the background information related to Electron framework along with the relevant security issues were discussed in chapters two to four. The research conducted for the 2017 release of the OWASP top 10 concluded with an acknowledgement towards a steady shift in wider usage of frameworks based on web technologies like Electron.

Web vulnerabilities in desktop environment present wider attack surface as presented in the security checklist by Doyensec [9] , therefore even more caution should be applied to possible security issues as the consequences can be more severe.

Within this thesis, the hypothesis aimed to validate that most open-source Electron-based applications are vulnerable to a web security issue - XSS, which evolves to code execution. The hypothesis proved to be true as 37% of the examined applications were identified to be vulnerable to code execution resulting in full system compromise. All findings were reported to the respective application owners and total of 7 CVEs were and will be requested. By the defense of this thesis, 2 of the CVE-s have already been assigned: CVE-2017-1000491 and CVE-2017-1000492.

The result was achieved by following a method of analysis conducted in two stages. Within the first stage, statistical data was collected about the required modules, included remote content and webPreference options. This allowed to gather insight on the applications being examined. Second stage was a three step process focusing on the validation or disproof of the hypothesis which produced the number of vulnerable applications. As a final step of this thesis, each of the vulnerable applications was exploited via a real attack vector and findings reported to the developers of each application to be fixed.

The high number of vulnerable applications reflects the need for awareness on Electron-based application security issues as future work. Evidently, the root cause of XSS to code execution issue lays in poor knowledge on web security issues, which are then brought to desktop environment where they elevate.

From the perspective of a security researcher, part of the future work should hold studying Electron framework and its specific vulnerabilities which would help to provide higher resistance to web vulnerabilities.

8 References

[1] "Top 50 products having highest number of cve security vulnerabilities", Cvedetails.com, 2017. [Online]. Available: https://www.cvedetails.com/top-50-products.php. [Accessed: 30- Nov- 2017]

[2] "W3Counter: Global Web Stats", W3counter.com, 2017. [Online]. Available: https://www.w3counter.com/globalstats.php. [Accessed: 30- Nov- 2017]

[3] Verizon. "2017 Data Breach Investigations Report", 10th ed. 2017, p. 38

[4] Trustwave Global Security Report. 2017, pp. 33, 79.

[5] OWASP Top 10 2017- The Ten Most Critical Web Application Security Risks. 2017, pp. 5-16.

[6] Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS.", Electron.atom.io, 2017. [Online]. Available: https://electron.atom.io/. [Accessed: 22- Oct- 2017]

[7] "Electron Apps | Electron", Electron.atom.io, 2017. [Online]. Available: https://electron.atom.io/apps/. [Accessed: 22- Oct- 2017]

[8] "Modern Alchemy: Turning XSS into RCE · Doyensec's Blog",Blog.doyensec.com, 2017. [Online]. Available: https://blog.doyensec.com/2017/08/03/electron-framework-security.html. [Accessed: 22- Oct- 2017]

[9] C.Luca. "Electron Security Checklist: A guide for developers and auditors. 2017, pp. 5-21

[10] "About Electron | Electron", Electron.atom.io, 2017. [Online]. Available: https://electron.atom.io/docs/tutorial/about/. [Accessed: 30- Oct- 2017]

[11] "Electron 1.0 | Electron Blog", Electron.atom.io, 2016. [Online]. Available: https://electron.atom.io/blog/2016/05/11/electron-1-0. [Accessed: 30- Oct- 2017]

[12] "Content - chromium/src.git - Git at Google", Chromium.googlesource.com. [Online]. Available: https://chromium.googlesource.com/chromium/src.git/+/master/content/. [Accessed: 30- Oct- 2017]

[13] "Electron/libchromiumcontent", GitHub, 2017. [Online]. Available: https://github.com/electron/libchromiumcontent. [Accessed: 03- Nov- 2017]

[14] "Content module - The Chromium Projects", Chromium.org, 2017. [Online]. Available: https://www.chromium.org/developers/content-module. [Accessed: 12- Nov- 2017]

[15] "Multi-process Architecture - The Chromium Projects", Chromium.org, 2017. [Online]. Available: https://www.chromium.org/developers/design-documents/multi-process-architecture. [Accessed: 12- Nov- 2017]

[16] "electron/electron-quick-start", GitHub, 2017. [Online]. Available: https://github.com/electron/electron-quick-start. [Accessed: 15- Nov- 2017]

[17] "Sandbox", Chromium.googlesource.com, 2017. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md. [Accessed: 30- Nov- 2017]

[18] D. Kerr, "As It Stands - Electron Security", Scott Logic, 2016. [Online]. Available: http://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html. [Accessed: 17- Nov- 2017].

[19] Progress "FUTURE OF JAVASCRIPT IN 2017 AND BEYOND. 2017, p. 28

[20] "Modulecounts", Modulecounts.com, 2017. [Online]. Available: http://www.modulecounts.com/. [Accessed: 20- Nov- 2017]

[21] P. Vorbach, "npm-stat: download statistics for NPM packages", Npm-stat.com, 2017. [Online]. Available: https://npm-stat.com/charts.html?package=nw&from=2017-01-01&to=2017-12-31. [Accessed: 31- Dec- 2017].

[22] P. Vorbach, "npm-stat: download statistics for NPM packages", Npm-stat.com, 2017. [Online]. Available: https://npm-stat.com/charts.html?package=electron&from=2017-01-01&to=2017-12-31. [Accessed: 31- Dec- 2017]

[23] "Writing Secure Node Code: Understanding and Avoiding the Most Common Node.js Security Mistakes", YouTube, 2016. [Online]. Available: https://www.youtube.com/watch?v=QSMbk2nLTBk&feature=youtu.be. [Accessed: 30- Nov- 2017]

[24] "NEWS: Node.js 8 Moves into Long-Term Support and Node.js 9 Becomes the New Current Release Line", Medium, 2017. [Online]. Available: https://medium.com/the-node-js-collection/news-node-js-8-moves-into-long-term-support-and-node-js-9-becomes-the-new-current-release-line-74cf754a10a0. [Accessed: 30- Nov- 2017].

[25] "comSysto Blog: Building a desktop application with Electron", Comsysto.com, 2015. [Online]. Available: https://comsysto.com/blog-post/building-a-desktop-application-with-electron. [Accessed: 24- Nov- 2017]

[26] "API | Electron", Electronjs.org, 2017. [Online]. Available: https://electronjs.org/docs/api. [Accessed: 11- Nov- 2017].

[27] "electron/electron", GitHub, 2017. [Online]. Available: https://github.com/electron/electron/blob/master/docs/api/ipc-main.md. [Accessed: 23- Nov- 2017].

[28] "electron/electron", GitHub, 2017. [Online]. Available: https://github.com/electron/electron/blob/master/docs/api/ipc-renderer.md. [Accessed: 20- Nov- 2017]

[29] "electron/electron", GitHub, 2017. [Online]. Available: https://github.com/electron/electron/blob/master/docs/api/remote.md. [Accessed: 30- Nov- 2017]

[30] OWASP Top 10 - The Ten Most Critical Web Application Security Risks. 2017

[31] Laura Donnelly, "Security breach fears over 26 million NHS patients", The Telegraph, 2017. [Online]. Available: http://www.telegraph.co.uk/news/2017/03/17/security-breach-fears-26-million-nhs-patients/. [Accessed: 17- Nov- 2017]

[32] "Cross-site Scripting (XSS) - OWASP",Owasp.org, 2016. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). [Accessed: 04- Nov- 2017]

[33] S. Edd Dumbill, "Glossary: Appendix E - Learning Rails - O'Reilly Media", Oreilly.com, 2008. [Online]. Available: http://oreilly.com/ruby/excerpts/ruby-learning-rails/ruby-glossary.html. [Accessed: 30- Nov- 2017]

[34] "Types of Cross-Site Scripting – OWASP", Owasp.org, 2017. [Online]. Available: https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting. [Accessed: 06- Nov- 2017]

[35] B. Fabrice "Cross-Site Scripting (XSS)", http://diuf.unifr.ch/drupal/tns/sites/diuf.unifr.ch.drupal.tns/files/Teaching/2006_2007/Computer_Security_Threats_and_Counter_Measures/Bodmer_CrossSiteScripting.pdf, 2006.

[36] "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks", Technical University of Vienna, University of California, Santa Barbara, 2006

[37] Steven Cook "A Web Developer's Guide to Cross-Site Scripting"", Sans.org, 2003. [Online]. Available: https://www.sans.org/reading-room/whitepapers/securecode/web-developers-guide-cross-site-scripting-988. [Accessed: 13- Nov- 2017]

[38] "XSS-game-appspot", Xss-game.appspot.com, 2017. [Online]. Available: https://xss-game.appspot.com/level3/frame#. [Accessed: 30- Nov- 2017]

[39] "Stack Overflow Developer Survey 2017", Stack Overflow, 2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017#technology-most-popular-languages-by-occupation. [Accessed: 05- Nov- 2017]

[40] "Stack Overflow Developer Survey 2017", Stack Overflow, 2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017#technology-frameworks-libraries-and-other-technologies. [Accessed: 06- Nov- 2017]

[41] "Stack Overflow Developer Survey 2017", Stack Overflow, 2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted-frameworks-libraries-and-other-technologies. [Accessed: 07- Nov- 2017]

[42] "Writing Secure Node.js Code - Danny Grander | @RisingStack", RisingStack Community, 2017. [Online]. Available: https://community.risingstack.com/writing-secure-node-js-code-danny-grander/. [Accessed: 07- Nov- 2017].

[43] S. Bryan. "Server-Side JavaScript Injection". 2011, pp. 2-5. [Online] Available: https://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf. [Accessed: 14-Nov-2017]

[44] R. Lukas, "From Markdown to RCE in Atom",Statuscode.ch, 2017. [Online]. Available: https://statuscode.ch/2017/11/from-markdown-to-rce-in-atom/. [Accessed: 30- Nov- 2017]

[45] "Content Security Policy (CSP)", Mozilla Developer Network, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. [Accessed: 01- Nov- 2017]

[46] C. Luca "Modern Alchemy: Turning XSS into RCE · Doyensec's Blog", Blog.doyensec.com, 2017. [Online]. Available:

https://blog.doyensec.com/2017/08/03/electron-framework-security.html. [Accessed: 30- Nov- 2017]

[47] "Same-origin policy", Mozilla Developer Network, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. [Accessed: 28- Oct- 2017]

[48] "<webview> Tag | Electron", Electronjs.org, 2017. [Online]. Available: https://electronjs.org/docs/api/webview-tag. [Accessed: 21- Nov- 2017]

[49] "shell | Electron", Electronjs.org, 2017. [Online]. Available: https://electronjs.org/docs/api/shell. [Accessed: 13- Nov- 2017]

[50] "BrowserWindow | Electron", Electronjs.org, 2017. [Online]. Available: https://electronjs.org/docs/api/browser-window. [Accessed: 15- Nov- 2017]

[51] "<webview> Tag | Electron", Electronjs.org, 2017. [Online]. Available: https://electronjs.org/docs/api/webview-tag. [Accessed: 13- Nov- 2017]

[52] "window.open Function | Electron", Electronjs.org, 2017. [Online]. Available: https://electronjs.org/docs/api/window-open. [Accessed: 14- Nov- 2017]

[53] "Prohibit `nodeIntegration` from being re-enabled with `window.open` · Issue #4026 · electron/electron", GitHub, 2016. [Online]. Available: https://github.com/electron/electron/issues/4026. [Accessed: 11- Nov- 2017]

[54] "Electron File Handling", www.tutorialspoint.com, 2017. [Online]. Available: https://www.tutorialspoint.com/electron/electron_file_handling.htm. [Accessed: 14- Oct- 2017]

[55] "Build software better, together", GitHub, 2017. [Online]. Available: https://github.com/search?utf8=%E2%9C%93&q=electron&type=. [Accessed: 09- Nov- 2017]