informatics

MDPI

Article

Mayall: A Framework for Desktop JavaScript Auditing and Post-Exploitation Analysis

Adam Rapley, Xavier Bellekens *, Lynsay A. Shepherd and Colin McLean

School of Design and Informatics, Abertay University, Dundee DD1 1HG, UK; me@adamrapley.com (A.R.); lynsay.shepherd@abertay.ac.uk (L.A.S.); c.mclean@abertay.ac.uk (C.M.)
* Correspondence: x.bellekens@abertay.ac.uk; Tel.: +44-13-8280-8482

Received: 17 September 2018; Accepted: 11 December 2018; Published: 17 December 2018

Abstract: Writing desktop applications in JavaScript offers developers the opportunity to create cross-platform applications with cutting-edge capabilities. However, in doing so, they are potentially submitting their code to a number of unsanctioned modifications from malicious actors. Electron is one such JavaScript application framework which facilitates this multi-platform out-the-box paradigm and is based upon the Node.js JavaScript runtime—an increasingly popular server-side technology. By bringing this technology to the client-side environment, previously unrealized risks are exposed to users due to the powerful system programming interface that Node.js exposes. In a concerted effort to highlight previously unexposed risks in these rapidly expanding frameworks, this paper presents the Mayall Framework, an extensible toolkit aimed at JavaScript security auditing and post-exploitation analysis. This paper also exposes fifteen highly popular Electron applications and demonstrates that two-thirds of applications were found to be using known vulnerable elements with high CVSS (Common Vulnerability Scoring System) scores. Moreover, this paper discloses a wide-reaching and overlooked vulnerability within the Electron Framework which is a direct byproduct of shipping the runtime unaltered with each application, allowing malicious actors to modify source code and inject covert malware inside verified and signed applications without restriction. Finally, a number of injection vectors are explored and appropriate remediations are proposed.

Keywords: JavaScript; Node.js; security vulnerabilities; arbitrary code execution; post-exploitation

1. Introduction

Application designers often seek to produce high-quality, cross platform applications in order to maximize their customer base. This introduces a number of difficulties regarding code reuse, with applications having to be modified, re-written or recompiled for each individual operating system.

Newer '*write once, run everywhere*' languages, with their multi-platform out-the-box paradigm, attempt to revolutionize the way in which native applications are developed. These languages, such as Go and Node.js, have garnered rapid popularity through their inherent power and ease of development [1]. In the case of Node.js, this rapid rate of expansion brought with it a staggering number of modules and frameworks, along with an ever growing list of security issues.

Electron is one such framework aiming to bring this JavaScript runtime to the native operating system, unifying web technologies and distilling them into what is effectively a native executable. In doing so, developers are not only potentially opening themselves up to traditional web exploits such as Single-Origin Policy misconfiguration [2]—albeit this time in the much more dangerous context of the operating system—but also exploits that are unique to the framework in question.

Frameworks such as Electron, NW.js and to some extent, the Chrome Embedded Framework, are rapidly expanding and as such, it is important to address the security surrounding these products. There has been an increasing number of recent discoveries and events which bring into question the level of focus on the security surrounding the modules available on *npm*, the de facto standard for Node.js package management. As Electron is beginning to emerge as the clear winner of the two Node.js-based frameworks, this paper will focus predominantly on the security surrounding Electron and the increasingly popular applications which are built upon it.

When contemporary frameworks grow at such exponential rates, including adoption from major players such as Microsoft [3], NVIDIA [4] and Slack [5], it is crucial that the security awareness surrounding the product does not go amiss. This reliance on the Node.js JavaScript runtime and the package manager that is used so prolifically with it compounds the security issues, as it adds additional levels of complexity which each require individual consideration with regards to security. There is comparatively little in the way of security research for Node.js and its accompanying frameworks when set side by side with other native programming languages, and what has been implemented in the way of security fixes is often reactive, rather than proactive. This is especially true with regards to the debacle over the *kik* module in March 2016, where an author unpublished a package on which many other packages depended [6].

This crucial oversight highlights a clear need to analyze the platform for other potential issues, and to consider remediations for them before they occur. By investigating the method by which dependencies are handled and the way in which code is executed, the Mayall framework (named after the astronomical body, Mayall's Object, which consists of two colliding galaxies; the framework's primary aim is to inject a module into the original application and modify its execution) presented in this paper allows developers to do just that. This paper explores the current state of security surrounding the Electron framework and the underlying Node.js constructs and demonstrates real world risks associated with this through the introduction of a post-exploitation framework built in Node.js, leveraging the modular structure of the language. The contributions from this paper are three-fold:

1. An Exploit able to take full advantage of popular Electron-based applications and providing command and control to a malicious user. Applications include Slack, Discord, etc.
2. Inherent issues to Electron are highlighted and described.
3. The Mayall framework, an open source and extensible security framework able to analyse and identify vulnerabilities in Node.js applications.

The remainder of this paper is organized as follows: In Section 2, an assessment of the current state of JavaScript security is performed and categories of vulnerabilities, language constructs and application management are discussed. This current review is taken further in Section 3 where we expose the vulnerabilities of fifteen popular applications built with Electron. Moreover, we present a proof of concept exploit which allows us to take advantage of Electron-based applications. The results from the audit are presented in Section 4 and a discussion on how to remediate the risks from exploitation is the focus of Section 5.

2. Background and Related Work

JavaScript security is a long discussed and multi-faceted affair; this is no different within the context of the Electron and Node.js runtimes. Now that the JavaScript language can be run natively, outwith the context of the browser, additional security concerns are brought to the fore. This is compounded by the rapid growth experienced by both JavaScript and Node.js over the past few years; JavaScript has consistently been top of the most popular technologies in the Developer Survey Results produced by [7].

This section explores the current security themes that are directly affecting the Node.js landscape. Specific examples are drawn upon where applicable to highlight the relevance of these themes when applied to Node.js. Whilst on the surface it may appear that some of these examples are self contained and acutely specific, the general risks discussed can be applied to Node.js and its desktop application counterpart, Electron. It is therefore important to discuss these precursor attacks that have direct relevance to these new languages and frameworks going forward.

2.1. Web-Based JavaScript

When looking at traditional web-only-based applications written in JavaScript, there are a series of recurring patterns that introduce a possibility for security vulnerabilities. For example, contained within JavaScript are a set of functions which allow for dynamic parsing and production of JavaScript code, the primary function being `eval()`. One particular study by Richards et al. [8] took a large sample of the top 10,000 websites and discovered that 89% of them used JavaScript. Furthermore, they discovered that "*over 82% of the top 100 pages use* `eval`*, and 50% of the remaining 10,000 pages do as well*" [8]. The primary issue that occurs with the use of `eval` is when arbitrary and unsanitized user data is passed to the function. When this occurs, it is possible for an attacker to execute the string they provide to the web application as JavaScript code. Whilst static code analysis often highlights security issues within an application, this becomes more challenging when dynamic elements such as `eval` are introduced. One method of analyzing such code is to perform an automatic transformation of "*many common uses of* `eval` *into other language constructs with the use of a dataflow analyzer*" [9]. These transformations are able to make some headway towards restoring the effectiveness of static code analysis on JavaScript applications. This is as a direct result of reducing the number of malleable and untrusted variables within the code, which subsequently reduces the number of mechanisms available to exploit such code.

While `eval()` is arguably the most relevant security risk when applied to modern cross-platform JavaScript applications due to its alarming prevalence in Node.js, there still remain a large number of additional JavaScript vulnerabilities that apply to both web and desktop JavaScript, the most common of which is cross-site scripting (XSS) [10]. Marked as third in the OWASP Top 10 vulnerabilities list, XSS works similarly to injection vulnerabilities (such as those described above) in that it arises from a mishandling of user input and can result in arbitrary JavaScript execution within the web application [11]. Furthermore, JavaScript is used in the development of browser add-ons, i.e., Web Extensions in Mozilla Firefox. Such vulnerabilities have the potential to allow for the development of malicious extensions, posing a security risk [12].

2.2. Node.js and Electron

2.2.1. Node.js

Node.js is a JavaScript runtime that is built upon the V8 JavaScript engine developed by Google. Its primary aim is to provide a highly scalable environment on which JavaScript code can be run, allowing it to be deployed on desktop and server-side environments. Its focus is driven towards asynchronicity and concurrency in handling a large volume of connections whilst preventing thread-locks within the application [13]. It can be reasoned that Node.js is more akin to a standard library than a web framework and is comparable to PHP, Ruby or Python rather than the frameworks that run atop these platforms such as Laravel, Ruby on Rails and Flask. This is an idea bolstered by Eloff et al. [14] and their analysis of Node.js as a baseline platform for web services.

Its adoption by commercial players such as Uber [15] and Netflix, [16] amongst others has helped to springboard the language towards success.

As it aims to be a powerful programming language in its own right, this requires it to contain a number of libraries and Application Programming Interfaces (APIs) that allow it to perform "*low level networking, basic HTTP server functionality, file system operations, compression and many other common tasks*" [17]. As a result of this, the `exec()` command was introduced to the Node.js specification in order to expose shell interaction functionality on the operating system level. Similar to the way `eval()` works, data that is provided to `exec()` is interpreted as a command, however it is passed to the operating system shell rather than the JavaScript engine [18].

2.2.2. Electron

The Electron framework is an open source library designed for cross-platform application development in web-based languages HTML, CSS and JavaScript [19]. It combines Chromium and Node.js into a single package so that traditional back-end and front-end code can be run within a single runtime environment.

A basic Electron application consists of three components: the `package.json` file, the Electron executable binary and the JavaScript source code.

2.3. Node Security

The tendency for writing insecure *eval* statements is not a scenario that is just limited to the web—desktop JavaScript is also affected by this [18]. There are two categories of dependencies used in Node.js. The first is a "*direct dependency*", a dependency which the developer explicitly includes in their project [20]. The other is a "*transitive dependency*" which the developer does not explicitly list in their project—"*a dependency that comes from anywhere in the tree of your direct dependencies' dependencies*" [20].

As we see an increase in desktop JavaScript applications being shipped (examples include the increasingly popular Slack and Discord [5]), the programming paradigms that were popular on the web, are also unsurprisingly popular on desktop frameworks. The transference of such insecure coding practices is dangerously widespread as highlighted by Staicu et al. [18]. The study was carried out across a sample set of 235,850 Node.js modules. Their results demonstrate that approximately 20% of these modules made use of the potentially unsafe `eval()` and `exec()` API calls directly or indirectly after all the first- and second-level dependencies were accounted for. A reasonable majority of these modules did not directly call these APIs however, with only 3% and 4% of the sample calling the `exec()` and `eval()` APIs directly—the remainder were run within the module dependent code [18].

Whilst popular belief may lead to the conclusion that these insecure modules are unpopular, this hypothesis is challenged by the discovery that the contrary is true and that in fact "*various vulnerable modules and injection modules are highly popular, exposing millions of users to the risk of injections*" [18].

The immediacy of these findings does not present itself until it is noted that the sandboxes present in the web browsing environments (such as Chrome and Firefox) are not present within the Electron framework [21] and that any application running on Node.js has full system access equivalent to the account that is running the process.

2.4. The npm Registry and Module Security

When the number of external *includes* present in an application increases, so does the potential for unchecked vulnerabilities and the level of trust placed in unknown module authors. This idea is backed up by research into remote inclusions which put forward the important notion that "*whenever developers of a web application decide to include a library from a third-party provider, they allow the latter to execute code with the same level of privilege as their own code*" [22]. The package management system that is used for Node.js is known as `npm` and is where the vast majority of Node modules reside. Additionally, as part of the module install process, npm exposes the functionality to run additional scripts on the target machine [23]. These so-called `preinstall` and `postinstall` scripts have historically been a cause for concern within the npm registry and Node.js landscape. These script hooks introduce a new danger to the npm and Node.js environment as the scripts will run on the host with the same privileges as the user installing the module.

2.4.1. Typosquatting

Tschacher et al. [24] bring to light the dangers of typosquatting within package manager. In their dissertation, they explore the malicious module that was released into the npm registry by a malware author on 26 January, 2015 [24]. The module in question was known as `rimrafall` and contained "*a preinstall hook that executed the command* `rm -rf /*`" [17,25]. As a result of this pre-install hook, any unsuspecting user that entered `npm install rimrafall` in a terminal prompt would end up with all their files being deleted from the machine. Whilst this module was removed from the registry less than two hours after being published, it does highlight the risks present in blindly *requiring* (similar to a Python `import` statement, a `require` statement is the equivalent method by which external JavaScript modules are imported to a JavaScript application) numerous modules into a project. The primary goal of this module appeared to be to highlight that "`npm install`" can be as dangerous as "`curl dangerous.com | sh`" [26].

Whilst Lift Security provide tools to actively monitor modules that are *required* as well as systematically audit the most popular modules present on npm [25], the fact remains that malicious modules can be introduced to the npm registry. Although methods exist to prevent scripts from running on install, the majority of users simply do not apply these checks when *requiring* modules into their codebase.

This lack of concern and consideration employed by some developers is evidenced in research performed on the effectiveness of typosquatting in a wide range of package managers for a multitude of different languages. In total, 214 different packages were uploaded to the respective registries with typos in their names, each of which acquired 92 installations on average per package [24]. Whilst this may not sound like much, "*the most installed package (*`urllib2`*) received 3929 unique installations in almost 2 weeks*" and "*the most installed package per day was* `bs4` *with 366 unique daily installations on average*" [24].

What is more alarming is the origin where these downloads occurred, as well as the prevalence of installs that were run with administrative privileges. It was found that 43.6% of module downloads and inclusions were run with administrative rights [24], thereby giving the accompanying install hooks administrative access on the machines where they were installed. In addition to the checking of administrative rights, [24] performed reverse DNS lookups with the installed module and discovered that his module had been installed on a surprising number of government and educational domains.

2.4.2. Trojan Modules

In addition to the direct installs of first-level dependencies (direct dependency) and the evident risks associated with directly installing unchecked code, there exists a risk of hidden malware further down the dependency tree (transitive dependency) [27]. If a malicious author can influence code that is included in numerous other modules, the infection rate will be vastly increased. This exact scenario has occurred in the past on a variety of modules, however [22] points to a specific instance with popular jQuery plugin, `qTip2`. Nikiforakis et al. state that "*The* `qTip2` *library was modified, and the malicious version was distributed for 33 days*" [22]. The compromised code made frequent callbacks to a specific IP address located in Russia and transmitted data including "*[the] site's hostname, [the browser's] user agent, and the referer*" [28].

These types of risk are not exclusive to WordPress and the web environment; this threat has presented itself in the npm registry. An npm developer encountered legal issues with Kik Interactive Inc.—developers of Kik Messenger—when he published a module under the name "kik". The npm developer was unable to come to an agreement with the company and after a fall out with the npm registry over the use of "kik" as his module name, the developer took the decision to remove all 273 packages that he authored from the registry. As a result of this, thousands of Node.js applications—including the widely used `babel` library (https://babeljs.io/ a JavaScript compiler used by Facebook, CloudFlare and Netflix, amongst others)—started issuing dependency errors and failed to execute [29].

In addition to the breaking changes that were made due to the unpublishing from the npm registry, the removal of these modules presented an even greater threat. Every single module name-space that had been removed had now become available for any developer to claim control over. This allowed any actor, either benevolent or malicious, to potentially inject code into thousands of Node projects *requiring* any of the widely used modules that he had authored [6]. npm responded reflexively by blocking the registration of any of the removed modules as well as applying the same tactic to any future removed application, but not before a vast number of modules dropped from the registry had already been claimed by other developers.

2.5. Developer Attitudes

As with a lot of systems, updating is often an easy way to ensure that applications are free from vulnerabilities. However, this is a major issue with the npm repository. As a result of the codebase being highly distributed over an exceedingly large number of developers, high code quality and a guarantee of constant security updates is not always available. Whilst auditing the top npm packages, Staicu et al. [18] discovered a number of vulnerabilities within modules which they responsibly disclosed to the developers. They stated that "*most of the developers acknowledge the problem, and they want to fix it. However, over the course of several months, only three of the 20 vulnerabilities have been completely fixed. The majority of issues are still pending, showing the lack of maintenance for many of the modules involved.*". This lack of maintained code is not the only issue present in the Electron and Node.js landscape; the prevalence of version pinning also prevents a large majority of applications from being updated even after vulnerabilities are patched. This has become the mainstream method of dependency management after GitHub actively recommended "*[setting] a fixed version number (`1.1.0` instead of `^1.1.0`) to ensure that all upgrades of Electron are a manual operation made by the developer*" [30]. This prevents applications suffering when breaking changes are introduced into the Electron framework, however it does place the onus on the developers to pay attention to updates and manually push these security fixes to the end users.

The following section expands on the work done by previous researchers, with a sharper focus on desktop applications written in JavaScript and JavaScript-based frameworks, by presenting a framework centered around the security of Node.js and its counterpart front-end, Electron. The framework is highly extensible and allows for the use of cross-platform payloads for remote execution of malicious code, taking advantage of the lack of code signing enforcement either remotely or locally.

3. Design and Implementation

This section aims to highlight the stages of development of the Mayall framework, an extensible toolkit aimed at JavaScript security auditing and post-exploitation. The framework provides two algorithms presented in Sections 3.2 and 3.3 respectively. Both aim at assisting a security analyst during an audit with automated JavaScript vulnerability scanning. Additional security vulnerabilities are brought to the fore in Section 3.4 with regards to how Electron handles updating application bundles and an exploit is demonstrated in Section 3.4.3. This notion of exploit development is expanded further with the production of a fully functional 'backdoor module' in Section 3.5 which discusses the methods by which Node.js malware can be introduced into a system and is rounded out in Section 3.6 with the amalgamation of the produced tools into one cohesive framework that can be used for JavaScript security analysis.

3.1. Application Auditing

Due to the high prevalence of insecure modules present within the npm repository, it is deemed appropriate to perform an analysis of modules in use by some of the most popular Electron-based applications. Electron applications are bundled into what is known as an *asar archive*, which is a concatenation of all files in the source code folder, similar to a tar archive. Electron can then access files from this archive during execution of the application. The `asar` package can be installed globally onto a system by using `npm` and the command `npm install -g asar`, after which point the `asar` command can be called directly from a terminal prompt. The asar archiving process does not contain any encryption or obfuscation and the tool is freely available and open source. As a result of this, it is possible to issue the command `asar e app.asar app` to retrieve the entire source code folder for an application, complete with formatting, comments and module dependencies as written by the developer. It is the latter which is of interest during this phase of the investigation as attempts will be made to discover if any of the modules contain vulnerabilities. Modules that have been included by either the application author directly (henceforth referred to as *primary modules*) or any of its dependencies (henceforth referred to as *tertiary modules*) are located in the `node_modules` folder within the decompressed asar archive. Each of these modules contains an individual `package.json` file which is specific to that module. This file contains details such as the name, version and author of the module, but also additional information such as the repository url for that module. Of the modules evaluated, only a negligible amount of modules did not provide a GitHub repository location and it was therefore very easy to further analyze these modules through commits and issues present on the respective repositories.

3.2. appScanner.js Algorithm

An algorithm (`appScanner.js`) was developed to assist in the version analysis of all imported modules, as the number of tertiary modules can be vastly larger than the number of primary modules, thereby creating excessive amounts of workload for a manual audit, as shown in Section 4. By interfacing with the GitHub API, it was possible to pinpoint the individual git commit which corresponded to the release number inside a module's `package.json` file. This was done by querying the repository listed within the `package.json` file for tagged commits with the version number of the module. Following this, a comparison of the returned commit to the latest commit on the master branch was made and a count was returned of the number of commits the master branch was ahead of the imported module.

Modules that were a substantial number of commits behind the master branch—above or equal to 150—were considered to be suspect packages and could help an analyst look in the right places for existing bugs and vulnerabilities which would affect an application.

The value of 150 was chosen as an arbitrary cut-off point for the purposes of this study, however, a lack of pulled commits may themselves not necessarily indicate a directly actionable exploit or vulnerability—this idea is discussed in more detail later in Section 5.1.2. If an application has not been updated to the latest build, it may be to prevent breaking changes rather than a lack of concern for updating.

3.3. nspCheck.js Algorithm

Whilst checking how far a module is behind the master branch helps to give indicators to the security status of an application, a git commit record does not immediately indicate security flaws within the respective modules without additional analysis. To this end, another algorithm was developed (`nspCheck.js`) allowing immediate highlighting of known and reported security vulnerabilities within a module.

The Node Security Platform (NSP) maintains a list of security advisories along with a corresponding API. By comparing the installed module versions against the NSP database, immediate security vulnerabilities can be flagged to developers. By querying this API and parsing the response data into an easily human readable format, the algorithm makes strides towards building a toolkit for security researchers and analysts assessing overall Node.js security within a network.

When auditing for vulnerabilities, it does not suffice to simply check the `package.json` for the main application. Although a developer may *require* only modules that are up-to-date, they are not directly responsible for the management of (transitive) dependencies, with this responsibility falling to the module author. As a result of this, vulnerabilities can be introduced despite a developer including only up-to-date version numbers. This is demonstrated in Theorem 1 and Figure 1, where it can be seen that whilst the developer has required up-to-date module versions, those modules in turn could depend on vulnerable code which is subsequently imported into the application.

Theorem 1. *Let A be the main source code, where $B, C = 1$st level (direct) dependencies and $D, E, F = 2$nd level (transitive) dependencies. Let X_n be the module version, where $n = 1$ indicates the most recent version and $n = 0$ represent outdated versions. Let X_r be the set of requirements for a module.*

$$A \supset \{B, C\} \supset \{D, E, F\}$$
$$A_r \mapsto B_1 \cup C_1$$
$$B_r \mapsto D_1$$
$$C_r \mapsto E_1 \cup F_0$$
$$\therefore A \mapsto F_0$$

This demonstrates that a single pinned version requirement further down a transitive dependency tree can result in the inclusion of outdated modules, despite the developer updating all of their primary (direct) dependencies.

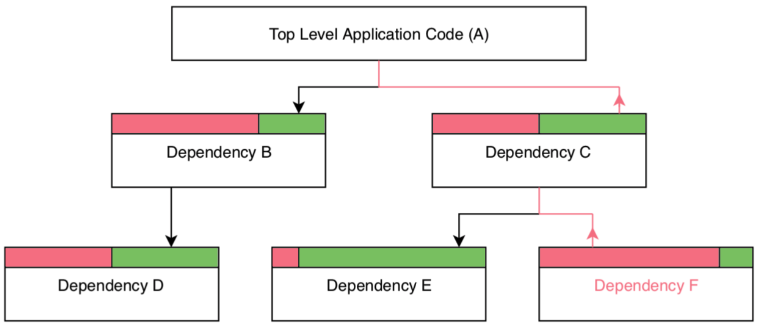


Figure 1. Nested dependencies.

Figure 1 illustrates the transitive dependency tree of an application, and demonstrates how outdated modules can be imported into top-level applications. Transitive dependencies can introduce vulnerabilities through version pinning. In the diagram, arrows represent transitive dependency calls, while the red and green sections represent vulnerable and patched module versions respectively. Many of the applications tested contained known vulnerable elements (Table 1) however, a number of well known applications passed this scan with no vulnerabilities detected. With this in mind, it was necessary to widen the scope to audit other aspects of the application processes, one of which was the update process.

Table 1. An automated scan of the applications against the Node Security Platform advisories list. The number of advisories may contain duplicate entries if the module has been included more than once.

Application	Total No. Advisories	Highest CVSS Score
1Clipboard	25	7.5
Atom	25	7.5
Caret	2	7.5
Discord	3	7.5
Ghost	0	—

Table 1. *Cont.*

Application	Total No. Advisories	Highest CVSS Score
Hyper	0	—
Mattermost	0	—
MongoDBCompass	17	7.5
NVIDIA	11	7.5
Popcorn Time	39	7.5
Popkey	7	7.5
Slack	0	—
Tofino Browser	0	—
SteelSeries	1	7.5
Wire Messenger	5	7.5

Popcorn Time is not an Electron app and is instead based on NW.js.

3.4. Malicious Update Process

Updating Electron applications is typically managed via the Squirrel Framework which is provided to Electron developers as an interface through the `autoUpdater` module shipped with Electron. The update process is not uniform across platforms with the Windows process requiring additional steps in order to produce the update packages.

3.4.1. Update Electron Packages

When Electron applications are updated on macOS, the only step that is taken is to recompile the entire application and push it to an update server. This update server is then polled at application startup by Electron applications running the `autoUpdater` code. For Windows, it is possible to create *delta packages* which contain the changes made since the last version.

The update server follows a specific directory layout where the platform specific updates are stored in sub-folders for that platform. The Windows subdirectory contains a `RELEASES` file which contains a list of packages available on the server in the format '`SHA1_checksum filename size_in_bytes`'. This `RELEASES` file is queried by the remote applications at startup to determine if an update is available. If one of the entries in the `RELEASES` file contains an application with a greater version number than the one provided by the application, the update process will automatically download and update to the latest version without any user input required.

Automatic updates on Windows will occur so long as code signing has not previously been put in place and the version number is higher than the current version performing the update check. On the other hand, macOS applications will not automatically update unless the update package is signed with a valid code signature. For this reason, a proof of concept was built on the Windows platform to eliminate additional overheads which are outwith the scope of this study, namely code signature bypasses.

3.4.2. Enumerating Update Endpoints

As a result of requiring no user input, it may be possible to covertly install malicious updates without user interaction. In order to demonstrate the vulnerability, a freely available application built on the Electron Framework was downloaded and its contents extracted with `asar e app.asar app`. The code in the resultant `app` folder was then examined and additional code was inserted. The added code resulted in the writing of a new file to the user's home folder on launch.

Due to the lack of code signing in effect on the application, and the lack of a secure HTTPS connection to the update server, it was possible to man-in-the-middle (MITM) the entire update process. When the application was launched, the connection request to the remote update server was intercepted and handled by a rogue update server that mimicked the responses that the application was expecting.

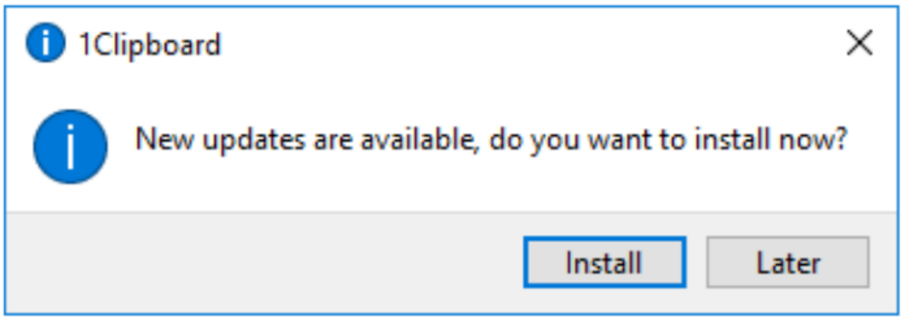
In order to build this rogue server, the update endpoints needed to be determined. This was achieved by setting an HTTP server handling all requests from the application. By modifying the hosts file to redirect the target application to localhost, all URL endpoints could be enumerated by the rogue server. It was subsequently possible to recreate the folder structure required for a Windows update and have the rogue server resolve the requests made by the insecure application.

3.4.3. Building Malicious Updates

The Windows update process involves downloading a delta package (a package containing the changes made since the last version) or, if one is unavailable, a complete package from the server. The package system in use is based on the Windows NuGet Package Manager and application updates are provided to the Squirrel Framework as `.nupkg` files.

A `.nupkg` file is compressed in the same way as a `.zip` folder and contains the source code for the application along with associated metadata such as version numbers. By unzipping the full nupkg file for the most recent version of the application and modifying the metadata and source code, a fully functional update could be created and served by the rogue server.

As a result of these steps, the application automatically updated on launch to the 'most recent version' as supplied to it by the rogue server, thereby writing a file to the user's home folder. As a presumed formality, the application presented the user with a dialog box informing that an update was available and asked the user if they would like to update now or later. The dialog box presented in Figure 2 only appeared after the automatic update had already been installed and it was irrelevant whether the user clicked 'Install' or 'Later'. If the user clicked 'Install', the application would open a web page in the user's default browser showing the patch notes for the version just installed. To alleviate suspicion from a user, this webpage (http://1clipboard.io/update) was cloned and served to the user through the malicious server.


1Clipboard

New updates are available, do you want to install now?

Figure 2. Man-In-The-Middle forcing an update.

3.5. Malicious Exploit Design

While injecting via an update is a viable infection method if the application does not correctly implement code signing, there remain a number of applications which do code sign and thus this method may sometimes be impractical. Despite the fact that code signing is in place during the update process, this security check is not enforced at application runtime and code signing is disregarded. To this end, if it is possible to gain access to a machine—through a social engineering or phishing attack for example—it would be possible to modify the application source code to inject malicious content. This is possible as the Electron executable does not check the code integrity of the application on runtime.

3.5.1. Malicious Exploit Design—A Dropper Module

Before the injection of an application is demonstrated, it is necessary to generate a payload that is able to communicate with and receive commands from a remote Command and Control (C2) server as demonstrated in Listing 1 representing the target code, and Listing 2 representing the pseudo code for the C2 Server. As Node.js is the underlying language that Electron is based on, the basic module was built in Node.js without Electron in mind—Electron is primarily used as the visual bolt-on to a Node.js application, exposing the Graphical User Interface (GUI) and other interaction methods to an end-user and Node modules can still be run underneath Electron.

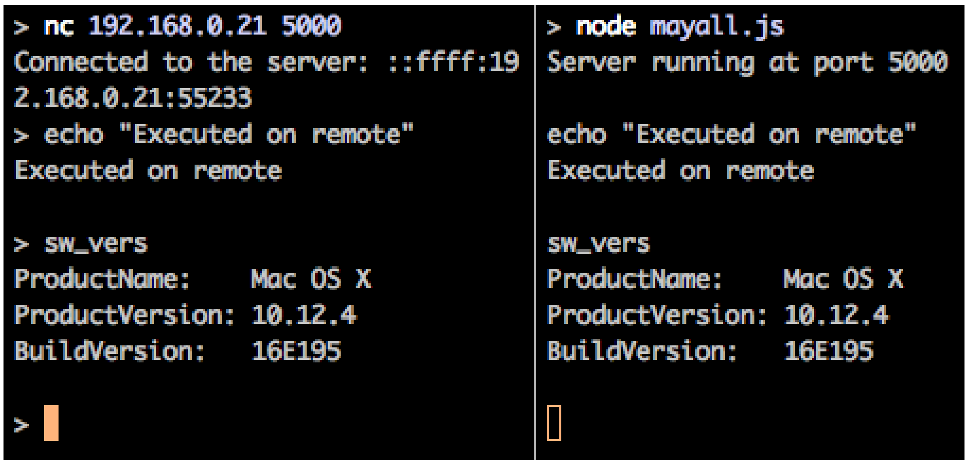
Listing 1: Mayall Interaction Pseudocode—Target Code.

```
const inj = new Function(conn_port) {
  net.createServer(function (socket) {
    // On the receipt of 'exec' data, execute as a commend.
    socket.on('exec', function (data) {
      exec(command, (err, stdout, stderr) => {
        socket.write(stdout);
      });
    });
  }).listen(conn_port);
}
```

Listing 2: Mayall Interaction Pseudocode—C2 code.

```
const c2 = new Function(local_port) {
  io.on('connection', function(socket) {
    // Initial agent activation
    dbInterface.addToAgents(socket.id);
    socket.on('hello', function(msg) {
      socket.emit('c2','world');
    });
    // Returning a response from a command exection
    socket.on('agentChatter', function(msg) {
      process.stdout.write(msg)
    });
  });
  // Commands can be sent to specific agents with specific types
  // Type classification can be seen in line 5 of listing 1.2
  exports.pushCmd = function(agent, type, command) {
    io.to(agent).emit(type, command);
  }
}
```

In order to receive commands, a websocket functionality was introduced, opening a port on the target machine on which Mayall listens for commands. This can be seen where the payload waits for data tagged with the string 'exec', the contents of which it executes in an `exec` command, writing the results back out to the socket. Once the module is running on a target machine, it is then possible to connect with a networking tool such as netcat, as shown in Figure 3.



```
> nc 192.168.0.21 5000
Connected to the server: ::ffff:19
2.168.0.21:55233
> echo "Executed on remote"
Executed on remote

> sw_vers
ProductName:    Mac OS X
ProductVersion: 10.12.4
BuildVersion:   16E195

>
```

```
> node mayall.js
Server running at port 5000

echo "Executed on remote"
Executed on remote

sw_vers
ProductName:    Mac OS X
ProductVersion: 10.12.4
BuildVersion:   16E195
```

Figure 3. Remote Command and Control example through Netcat.

3.5.2. Covertly Embedding Modules

Running a node module outright from the command line, although effective in demonstrating Node.js capabilities, is not going to have a high conversion rate from payloads delivered to remote shells returned. In order to have a greater activation rate from the Mayall payload, it was deemed necessary to embed it within applications that a user trusts and installed themselves. An additional benefit of embedding an application with the Mayall malware (rather than running it directly) is that any firewall notifications will be requested on behalf of that injected application. For instance, if Mayall is injected into Slack—a popular team communication product—any initial firewall request by Mayall to open ports will appear to come from Slack and not the malicious module. Furthermore, once this request has been accepted once, it will continue to be allowed to run on future execution of Slack.

The Mayall payload is developed further to allow agnostic infection across multiple operating systems and applications. A list of popular Electron applications is compiled and shipped alongside the module, complete with an automatic injector. Figure 4 demonstrates the method by which the payload embeds itself within an application, starting with an initial operating system detection. After determining the platform inside which it is ran (exposed through Node.js' `process.platform` variable), the injector is subsequently handed off to the appropriate dropper module. This module will search the typical installation locations for Electron applications for the system type and inject Mayall malware into any applications that it discovered.

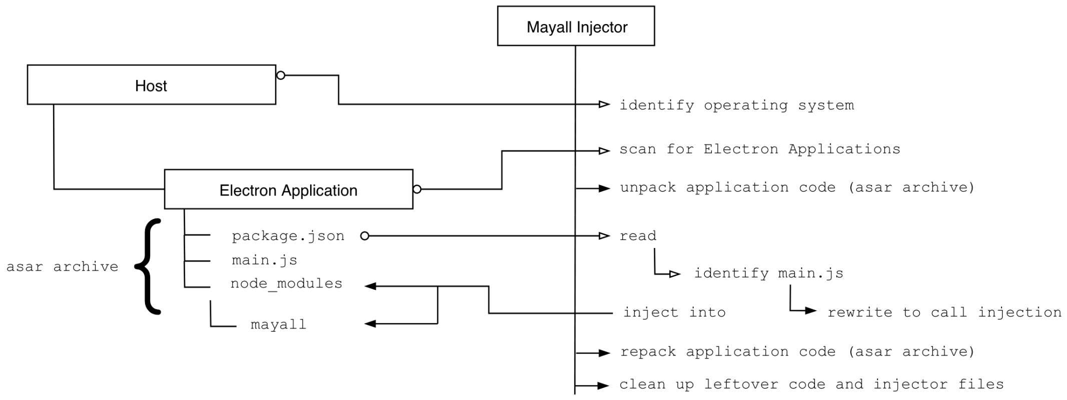

Mayall Injector
Host
Electron Application
asar archive
package.json
main.js
node_modules
mayall
identify operating system
scan for Electron Applications
unpack application code (asar archive)
read
identify main.js
inject into
rewrite to call injection
repack application code (asar archive)
clean up leftover code and injector files

Figure 4. Flow of execution for injecting malicious modules into trusted Electron Applications.

This is achieved by first extracting the asar archive associated with the application and executing a read of the `.main` field from the `package.json` file located within the archive. This field details the entry point of the application. If a valid file is discovered, the dropper downloaded the mayall module into the `node_modules` folder in the extracted archive. This is followed by a pre-pending of the entry file with a require statement for the newly downloaded module, causing it to be loaded into memory when the application next executes. Once in place, the injector then repackages the asar archive and cleans up any files that were left behind as a result of the execution.

As with many modern day applications, developers of Electron applications exhibit a tendency to have their applications launch on boot. As a result of this software design trait, persistence is gained when Mayall is injected into an application which exhibits this behavior—examples include the highly popular Slack, Discord and Tidal Music applications (Figure 5).

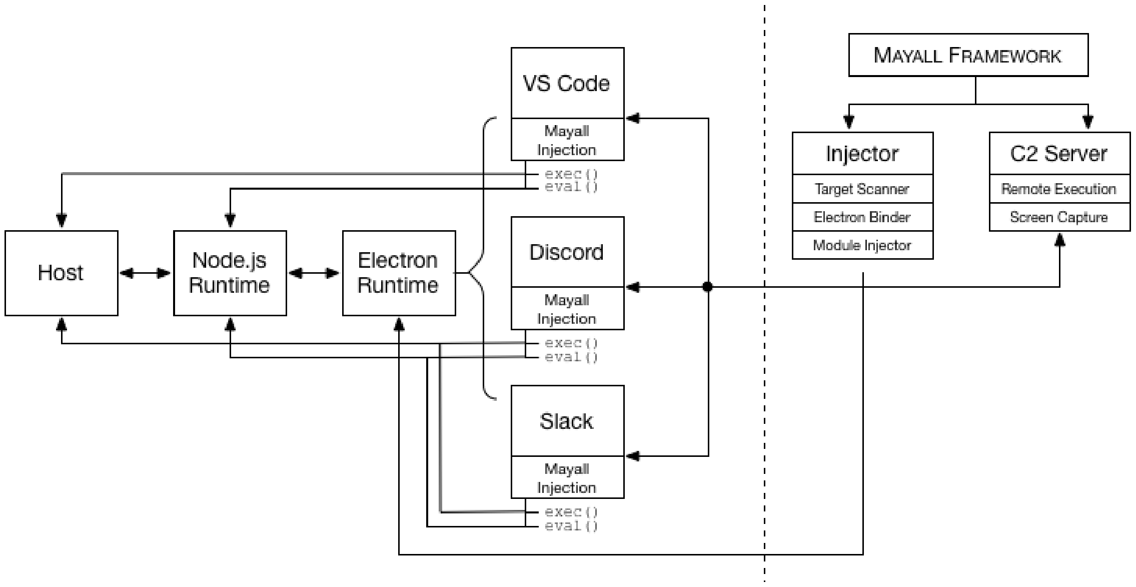


Figure 5. Framework flow between the infected system and command and control server running under Mayall.

3.5.3. Executing JavaScript Natively on a Target Machine

As this injection module is written in Node.js, it may be assumed that the Node.js is a necessary installation prerequisite for executing a successful attack. However this is not the case. If Electron applications are the target then it is possible to use operating specific scripts (for example, batch files on Windows or bash scripts on *nix systems) to identify these applications. Once identified, the injector module can be executed with the Node.js binaries that are provided alongside the Electron executables. As aforementioned, the Electron application does not concern itself with the integrity of the code that is being run and as such it would be possible to decouple the main application code from the Electron application and instead run the malicious code against this decoupled binary including a cleanup and replacement of the old code base. The Node.js binary can often be found in unexpected locations. NVIDIA is one such example. When installing the appropriate drivers for an NVIDIA graphics card, a web helper is also installed for the NVIDIA GeForce Experience companion application. This application handles driver updates, game optimisation and the ability to record and stream gameplay through ShadowPlay. Additionally, this application is installed by default when drivers are installed for the graphics card.

Contained within this package is a binary titled `NVIDIA Web Helper.exe` which upon further inspection is a Node.js executable which has been resigned by NVIDIA. This provides a binary upon which malicious code can be run with the added bonus of being signed by NVIDIA, therein bypassing certain security features provided by Windows—namely application white-listing [4].

3.6. A Framework for Post-Exploitation

In a concerted effort to raise awareness and kickstart the introduction of actionable change in the JavaScript security landscape, the Mayall Framework will be released as an open source tool for the open source community, allowing security researchers to efficiently and effectively drill down on the security issues plaguing modern JavaScript applications, in a similar vein to other frameworks such as MetaSploit and PowerShell Empire.

4. Results

This section presents the findings of the investigation carried out on the different applications.

4.1. Scanning Module

The scanning module produced in Section 3.2 was run against a number of the most popular applications written in the Electron framework, the results of which can be seen in Table 2.

Table 2. An automated scan of fifteen popular Electron applications.

Application	No. Inc.	No. Deps.	Commits	Avg. Comm.
1Clipboard	24	374	48,724	157.23
Atom	56	578	34,181	82.36
Caret	29	89	2615	46.70
Discord	10	130	9805	89.14
Ghost	10	45	1014	25.35
Hyper	17	69	2039	29.99
Mattermost	11	85	7652	99.38
MongoDBCompass	86	841	132,281	269.96
NVIDIA	N/A	116	6425	69.84
Popkey	12	84	11,523	177.28
Tidal	13	115	5339	58.03
Slack	97	257	38,411	185.56
Tofino Browser	4	277	7365	31.21
SteelSeries	1	4	364	182
Wire Messenger	11	336	19,306	70.20
Average	27	227	21,803	104.95

This shows how many modules are imported directly by the developer, followed by the subsequent total number of modules that are imported to the application through the full transitive dependency tree. The commits column shows the number of commits behind the master branches in the associated GitHub repositories, as well as the number of commits each module is behind on average.

In addition, the number of application dependencies were compared to the number of modules explicitly imported by the developers (direct dependencies). This demonstrates that a relatively low number of direct imports can result in mass inclusions of additional dependent modules, vastly increasing the code base with an approximate five-fold increase in total modules against primary imports. In addition to the GitHub upstream checker, the same applications were audited against the Node Security Platform and the list of advisories that NSP maintain on npm modules. Table 1 (previously shown in Section 3.3) details the number of unique advisories found along with the highest CVSS score found.

4.2. Update Injection and Malicious Modules

By producing a malicious update package and placing it on a server which responded to URL requests from Electron applications, it was possible to inject an application which had not taken basic security measures to ensure server authenticity. If the malicious update claimed a high version number (such as `v.50` when the current version is `v.0.8.1`), future updates would also be blocked as the application would opt out of downgrading its version number. Once injected (either through update hijacking or script running), it is possible to exfiltrate data through the execution of commands on the remote system.

5. Discussion

This section discusses the implementation of the Mayall framework, focussing on its strengths and limitations. It then goes on to discuss current remediations for security issues in Node.js and Electron.

5.1. Strengths and Limitations of the Mayall Framework

5.1.1. The Implementation of the Mayall framework

The Mayall framework seeks to be an extensible toolkit, which aims to tackle the issues of security auditing, and post-exploitation. To this end, two algorithms were developed: `appScanner.js` and `nspCheck.js`.

The `appScanner.js` portion of the Mayall framework was successful in returning the number of commits the master branch was ahead of the imported module, indicating the possibility of a security vulnerability residing within the imported modules. However, it should be noted that being a large number of commits behind the master branch does not necessarily indicate a problem, and this issue is discussed in Section 5.1.2. The `nspCheck.js` algorithm of the Mayall framework worked in conjunction with `appScanner.js`, and was able to successfully highlight known security vulnerabilities within a module. Although research within this paper was focussed on the top 15 Electron applications, similar results would be expected if the scope of the study was widened.

The Mayall framework is divided into four primary sections: Listeners, Agents, Droppers and Modules. In the proof of concept exhibited within this paper, only the first two, listeners and agents, have been fully implemented with droppers and modules coming in a later release.

5.1.2. Git Commit Divergence

As briefly mentioned in Section 3.2, whilst a notable divergence in commits from the master branch may not necessarily indicate the certainty of a security risk, it is still enough to raise concern. This disparity in module update mentality when compared to the urgency and immediacy of updates seen elsewhere (operating systems, etc.) may have been caused in part by this notion of version pinning, combined with the prolific lack of code maintenance—an issue that was also noted by [18] in Section 2.5. Whilst version pinning will prevent unwanted change (be it breaking API changes or malicious trojan inserts), it may also simultaneously prevent security updates from being patched into a system. This was the case when modules in the test sample were found to contain known vulnerabilities when cross-referenced against the Node Security Platform Advisories List [31]. By modifying the `appScanner.js` tool to additionally search for key phrases such as 'security', 'urgent' or 'vulnerability' within the commit messages or issues that are present between the current version and the version in use, it should be possible to create more meaningful data that developers will be able to respond to quickly. The primary risk with this method is centered around the list of search terms against which commits and issues are compared. Each developer and repository has their own type and there is a risk that tag names will be missed if they are stylized differently (for example, a tag with the name 'Type: Bug' instead of just 'Bug'). If crucial tags are missed, then vulnerabilities could slip through the net, giving developers a false sense of security that their applications are secure. The Node Security Platform is making strong steps towards providing a searchable database of known security vulnerabilities, however more work is required to ensure that module maintainers are submitting these vulnerabilities to the database as they patch, in order to provide a central queryable database for all developers.

5.1.3. Update Process

Evergreen applications are gaining greater traction as part of the software development life cycle. Evergreen here refers to the process of automatic self-updating and is adopted by widely used applications such as Chrome and Spotify as well as the majority of applications built on Electron. As a result of this well documented methodology, it is easy to configure an update server which applications can poll at regular intervals as part of this self-updating process. Consequently, it has also proved trivial for an attacker to configure a rogue server that mimics expected application endpoints if developers have not taken basic measures to protect the security of this process.

Whilst TLS-encrypted communication and application code signing is not an issue caused directly by Electron, steps could be taken to enforce these more secure practices. For example, macOS applications cannot be automatically updated unless a valid code signature is provided. Whilst this is a security feature present in macOS, this ideology could be transferred from the operating system paradigm into the cross-platform Electron Framework. By rejecting all application updates that do not have valid code signatures, developers will be forced to move towards code signing in order to release a popular and relevant application. Although this may seem like a drastic measure, Google are performing similar measures with regards to their "Not Secure" stance on web pages that do not employ HTTPS [32]. In addition to this, the introduction of code signing across the board may help to unify the fragmented update process currently in place. By calling for a redesign of the process, it may also be beneficial to reassess the packaging process so that the entire procedure can be made simpler for developers, potentially with the introduction of certificate acquisition scripts which are a practice currently recommended by [33] with regards to TLS. This can also bring Linux `autoUpdating` in line with macOS and Windows, a platform which is currently unsupported by the Squirrel updater.

5.1.4. Application Signing Against Runtime

While the `Program Files` directory in Windows requires administrative privileges to write to, the majority of Electron applications are installed in `AppData/Local` on Windows which does not require any special permissions. This places application source code in an unprotected environment and is free to be modified by any process. In order to protect the system from any malicious activity caused by a change in source code, code integrity checking mechanisms must be implemented at the execution stage of an application. This is a process that is not only confined to Windows, as macOS also suffers the same issue. By running simple scripts with the same privilege of the current user, all Electron applications can be injected with malware, potentially without any knowledge of this from the user. If Electron were to implement rigid code signing policies, any malicious changes made to the source code of such applications would result in a non-execution of the application at launch. Unfortunately, this change to the framework would create massive breaking changes. If the next version of Electron were to implement this type of code signing, there may not be anything stopping an attacker from replacing the Electron binary with a downgraded version which does not check for code signatures. As a result, application breaking code changes may be required in order to make downgrading a time inefficient attack for an adversary.

5.1.5. Secure TLS Connections

One major oversight was the lack of transport layer security implementations that allowed for malicious update injections due to the absence of cryptography on all connections to the update server. As Electron allowed updates to occur over HTTP, it was trivial to execute a man-in-the-middle attack and perform arbitrary remote code execution on the target machine. If Electron were to prevent such connections from occurring, it would prove much more difficult to inject applications in this way. In addition to this, Electron should issue warnings to developers concerning the downloading of resources over HTTP and begin deprecation of this plaintext transport mechanism. At the time of writing, there are no less than 142 individual security advisories on the Node Security Platform for applications which download resources over HTTP, leaving themselves vulnerable to MITM attacks, potential code injection and arbitrary execution. This downloading of binary resources through insecure means opens applications up to the types of attack that have been demonstrated throughout this paper as an attacker will be able to replace the resource being downloaded with their own malicious versions.

5.2. Repackaging Binaries

The practice of shipping JavaScript applications is becoming widespread, however the act of resigning a Node.js binary and shipping it as part of a wider product has not been properly observed or documented in the wild until it was discovered in an NVIDIA package by [4]. By providing the Node.js binary to vast swathes of users, one of the main barriers to entry for JavaScript malware is removed. One potential explanation as to why JavaScript malware has only seen a slow uptick in recent years could be due to the lack of an available runtime against which to execute it. As more applications are beginning to ship with Node.js included either through Electron or as a directly repackaged binary, it may be the case that a notable increase in JavaScript malware is observed. The ability to be able to produce truly cross-platform malware that can run on both servers and clients across all major operating systems could have potentially catastrophic outcomes.

5.3. Developer Involvement with the Node Security Platform

Although it may be easy for security researchers to pin the blame on the insecure practices of developers, this is neither helpful nor constructive. In order to remediate the risks associated with the use of known vulnerable modules, more awareness of the dangers and available scanning tools must be made available to developers. The npm package manager has the ability to issue warnings at the terminal prompt whenever vulnerable or deprecated modules are `npm install`ed, however these warnings do not flag up for every vulnerability listed in the NSP Advisories List.

It is proposed that npm issue a warning detailing the risks of installing these modules and have developers explicitly accept these risks before the installation continues. In addition to this, it may be beneficial to warn developers of out of date, dangerous modules at runtime, each time the application is tested or executed. This upfront alerting mechanism may influence developers to start taking measures to secure their applications before deployment.

5.4. Current Remediations

A number of suggestions have been made to combat the risk of uncontrolled JavaScript in web and client-side applications, however a lot of these struggle to find their place in the full-stack JavaScript development environment [34,35]. For example, [22] recommends sandboxing remote code as well as manually importing remote includes by writing the code directly into the main source. While sandboxing has been a long discussed issue on the Electron GitHub repository and work was done by developers of the Brave Browser to reintroduce the Chrome Sandbox in their forked version of Electron [36], there is still a lot of discussion and work to be done in order to merge these changes back into the upstream Electron repository [37]. With regards to minimizing external includes, the issue lies with the Node.js model of programming where small modular includes are a major part of the language style. Whilst it would be possible to rewrite code from external modules directly into the source code, it is not the widely accepted or adopted method of Node.js application development. In addition to this, there are occasions where this is simply infeasible and could end up causing more harm than good—for instance, with regards to cryptography and authentication modules. By analyzing how other languages and frameworks handle code integrity, the immaturity of some elements of

Node.js and Electron become clear. Ref. [38] poses multiple solutions for ensuring software integrity including the introduction of 'file system integrity checkers', 'code signing' and 'visualization' in the event of attempted malicious code execution. Whilst some of these elements may be in the process of implementation, these measures simply have not been effectively implemented in Electron. There have been a number of methods proposed by which arbitrary code execution could be mitigated in both the context of secure application development and in application runtime monitoring and threat modelling. One such method is presented by [18] who proposed a tool that reduced the impact and effectiveness that vulnerable call sites (such as `eval` and `exec`) have on a system. Whilst this research study discusses "*[their] runtime mechanism effectively prevents 100% of the attacks*", they also highlight that "*developers who both use and maintain JavaScript libraries are reluctant to use analysis tools and are not always willing to fix their code*". This reluctance to employ third-party security tools suggests the need to push greater security measures into the frameworks and runtimes themselves, rather than placing the onus on either the developer or the user to enforce good security practices at runtime. Another solution to protect against malicious modules is based around runtime modification and surrounds itself with the idea of threat modeling. Ref. [39] investigates the process of creating "*access control policies on interactions between libraries and their environment*", giving modules only the access that they require based on a model of 'least-privilege'. Whilst this is a potentially feasible implementation when dealing with server-side JavaScript, this policy enforcement cannot be guaranteed when applied to Electron, as an attacker could have full control over the source code and execution of the application.

6. Conclusions

In this manuscript, a fully-functional framework is presented, demonstrating a technique to successfully inject user-installed applications with malicious Mayall modules. The framework presented was made of two primary modules: the injector and the command and control server, as well as the vulnerability scanning algorithms. By using the Electron runtime as a standalone executable, the injector was able to execute on the remote host using the Electron and Node APIs bundled within the Electron binary. Furthermore, by providing the ability to pass `eval` and `exec` parameters to the payload from a remote server, execution is possible not only within the Electron and Node.js runtimes, but also from the native operating system shell.

The findings drawn from this manuscript highlight the need for increased awareness and action surrounding JavaScript security. As Node.js inevitably continues to grow, the awareness of risks associated with running untrusted code needs to grow alongside it. Unlike other popular programming languages (such as Python), Electron applications are not compiled directly into native executables for that operating system (in the same way py2exe would for Python) and are rather executed via the generic runtime environment that is distributed alongside the application. This introduces potential and previously unseen risks—as demonstrated by NVIDIA's packaging and release of their rebranded and signed Node.js binary—as it provides attackers with a new increasingly available toolset with inbuilt system interfacing capabilities. Moreover, this study demonstrated the lack of accountability held by an Electron binary for its associated code base. The ability to entirely rewrite sections of an application without throwing alerts or warnings is a major risk that could be easily leveraged by an attacker under the appropriate circumstances.

Future Work

In order to be an extensible framework (similar to Powershell Empire and MetaSploit), the source code will be made available online for other researchers to contribute. By leveraging the module dependency nature of Node.js, we plan on building extensions as standalone modules which are needed by the core framework. Moreover, we plan to port popular penetration testing tools such as Mimikatz onto the Mayall framework.

Author Contributions: A.R. carried out the investigation, and wrote the first draft of the paper. C.M., L.A.S., and X.B. supervised the project, and wrote the paper.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Tilkov, S.; Vinoski, S. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Comput.* **2010**, *14*, 80–83. [CrossRef]
2. Meyerovich, L.A.; Zhu, D.; Livshits, B. Secure cooperative sharing of JavaScript, browser, and physical resources. In Proceedings of the Workshop on Web 2.0 Security and Privacy, Oakland, CA, USA, 20 May 2010.
3. Carter, B. HTML Educational Node. js System (HENS): An Applied System for Web Development. In Proceedings of the 2014 Annual Global Online Conference on Information and Computer Technology (GOCICT), Louisville, KY, USA, 3–5 December 2014, pp. 27–31.
4. Freingruber, R. Abusing NVIDIA's node.js to Bypass Application Whitelisting, 2017. Available online: http://blog.sec-consult.com/2017/04/application-whitelisting-application.html (accessed on 23 April 2017).
5. GitHub. Apps Built on Electron, 2017. Available online: https://electron.atom.io/apps/ (accessed on 6 April 2017).
6. Schlueter, I.Z. kik, left-pad, and npm, 2016. Available online: http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm (accessed on 15 February 2017).
7. Stack Overflow. Developer Survey Results, 2016. Available online: http://stackoverflow.com/insights/survey/2016 (accessed on 9 April 2017).
8. Richards, G.; Hammer, C.; Burg, B.; Vitek, J. *The Eval That Men Do*; Springer: Berlin/Heidelberg, Germany, 2011.
9. Jensen, S.H.; Jonsson, P.A.; Møller, A. Remedying the eval that men do. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, Minneapolis, MN, USA, 15–20 July 2012; pp. 34–44.
10. The OWASP Foundation. OWASP Top 10—2013, 2013. Available online: https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf (accessed on 7 April 2017).
11. Binnie, R.; McLean, C.; Seeam, A.; Bellekens, X. X-Secure: Protecting users from big bad wolves. In Proceedings of the 2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech), Balaclava, Mauritius, 3–6 August 2016; pp. 158–161.
12. Shepherd, L.A.; Archibald, J.; Ferguson, R.I. Reducing Risky Security Behaviours: Utilising Affective Feedback to Educate Users. *Future Internet* **2014**, *6*, 760–772. [CrossRef]
13. Joyent. About Node.js, 2017. Available online: https://nodejs.org/en/about/ (accessed on 6 March 2017).
14. Eloff, E.; Torstensson, D. An Investigation into the Applicability of Node.js as a Platform for Web Services. Ph.D. Thesis, Department of Computer and Information Science, Linköpings Universitet, Linköping, Sweden, 2012.
15. Node.js. How Uber Uses Node.js to Scale Their Business, No Date. Available online: https://nodejs.org/static/documents/casestudies/Nodejs-at-Uber.pdf (accessed on 10 April 2017).
16. Trott, K. Node.js Interactive Conference—Node.js at Netflix, 2015. Available online: https://www.youtube.com/watch?v=p74282nDMX8&feature=youtu.be&t=12m11s (accessed on 10 April 2017).
17. Ojamaa, A.; Düüna, K. Security assessment of Node. js platform. In Proceedings of the 2012 International Conference for Internet Technology and Secured Transactions, London, UK, 10–12 December 2012; pp. 35–43.
18. Staicu, C.A.; Pradel, M.; Livshits, B. Understanding and Automatically Preventing Injection Attacks on NODE. JS. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018, San Diego, CA, USA, 18–21 February 2018.
19. GitHub. Electron Documentation—About Electron, 2017. Available online: https://electron.atom.io/docs/tutorial/about/ (accessed on 9 April 2017).
20. Andrew Goode. Dealing With Problematic Dependencies in a Restricted Network Environment, 2016. Available online: https://blog.npmjs.org/post/145724408060/dealing-with-problematic-dependencies-in-a (accessed on 17 November 2018).

21. Kerr, D. As It Stands—Electron Security, 2016. Available online: http://blog.scottlogic.com/2016/03/09/As-It-Stands-Electron-Security.html (accessed on 10 March 2017).
22. Nikiforakis, N.; Invernizzi, L.; Kapravelos, A.; Van Acker, S.; Joosen, W.; Kruegel, C.; Piessens, F.; Vigna, G. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, Raleigh, NC, USA, 16–18 October 2012; pp. 736–747.
23. npm. npm-scripts—How npm Handles the "Scripts" Field, 2017. Available online: https://docs.npmjs.com/misc/scripts (accessed on 10 April 2017).
24. Tschacher, N.P. Typosquatting in Programming Language Package Managers. Bachelor Thesis, Department of Informatics, University of Hamburg, Hamburg, Germany, 2016. Available online: http://incolumitas.com/data/thesis.pdf (accessed on 9 April 2017).
25. Baldwin, A. A Malicious Module on npm, 2015. Available online: https://blog.liftsecurity.io/2015/01/27/a-malicious-module-on-npm/ (accessed on 10 April 2017).
26. Jeronimo, J. rimrafall—npm install could be dangerous, 2015. Available online: https://github.com/joaojeronimo/rimrafall (accessed on 10 April 2017).
27. Pfretzschner, B.; ben Othmane, L. Identification of Dependency-based Attacks on Node. js. In Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, 29 August–2 September 2017; p. 68.
28. Nigro, D.L. Pulling Apart a WordPress Hack, Unobfuscating Its Code, 2011. Available online: https://dan.cx/2011/11/pulling-apart-wordpress-hack (accessed on 10 April 2017).
29. Williams, C. How One Developer Just Broke Node, Babel and Thousands of Projects in 11 Lines of JavaScript, 2015. Available online: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/ (accessed on 12 April 2017).
30. GitHub. Electron Documentation—Electron Versioning, 2017. Available online: https://electron.atom.io/docs/tutorial/electron-versioning/ (accessed on 12 April 2017).
31. Node Security. Advisories, 2017. Available online: https://nodesecurity.io/advisories (accessed on 24 April 2017).
32. Schechter, E. Moving Towards a More Secure Web, 2016. Available online: https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html (accessed on 26 April 2017).
33. Electronic Frontier Foundation. Certbot, 2017. Available online: https://certbot.eff.org/ (accessed on 26 April 2017).
34. Kirda, E.; Kruegel, C.; Vigna, G.; Jovanovic, N. Noxes: A client-side solution for mitigating cross-site scripting attacks. In Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 23–27 April 2006; pp. 330–337.
35. Prokhorenko, V.; Choo, K.K.R.; Ashman, H. Context-oriented web application protection model. *Appl. Math. Comput.* **2016**, *285*, 59–78. [CrossRef]
36. lostfictions. [Security] Investigate Switching to Brave's Fork of Electron-Prebuilt? 2016. Available online: https://github.com/jiahaog/nativefier/issues/288 (accessed on 12 April 2017).
37. de Arruda, T. Allow Electron Renderers to Be Run Inside Chromium Sandbox, 2016. Available online: https://github.com/electron/electron/issues/6712 (accessed on 6 April 2017).
38. Catuogno, L.; Galdi, C. Ensuring Application Integrity: A Survey on Techniques and Tools. In Proceedings of the 2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, Santa Catarina, Brazil, 8–10 July 2015; pp. 192–199.
39. De Groef, W.; Massacci, F.; Piessens, F. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14, New Orleans, LA, USA, 8–12 December 2014; pp. 446–455.