Introduction: Thick Client

Thick client applications, called desktop applications, are full-featured computers that are connected to a network. Unlike thin clients, which lack hard drives and other features, thick clients are functional whether they are connected to a network or not.

While a thick client is fully functional without a network connection, it is only a "client" when it is connected to a server. The server may provide the thick client with programs and files that are not stored on the local machine's hard drive.

Example Application:

- Firefox
- Chrome
- Burp Suite
- OWASP ZAP
- Zoom
- Desktop games
- Music Player
- Text editor

Common Architectures of Thick Clients

- Two Tier
- Three Tier

Two Tier: The two-tier is based on Client-Server architecture. The two-tier architecture is like a client-server application. Direct communication takes place between client and server. There is no intermediate between client and server.

Example Applications:

- Desktop Games
- Music Player
- Text Editor

Three Tier: The three-tier is based on Client – Application Server – Database Server architecture. The Application server is the mediator between client and server, it transfers data from client to server and vice versa.

Example Applications:

- Firefox
- Chrome
- Burp Suite
- Zap Proxy

As we have understood what thick client is and its architecture, now we'll move on to penetration testing approach.

Starting checks (Enumeration)

- Application Architecture
- Business Logic

- Platform Mapping
- Understanding Application & Infrastructure
- Languages and Frameworks
- Network Connection with WireShark/TCPview

Common Tools and Vulnerabilities

Tools

- CFF Explorer
- Wireshark/TCPview
- Procmon
- Detect It Easy
- Echo Mirage

Vulnerabilities

- Hardcoded Sensitive Information in Code & Config/Log files
- Unquoted Service Paths
- DLL Hijacking
- SQL Injection
- Lack of code obfuscation
- Buffer Overflow

An Intro to Electron Application Penetration Testing

what is ElectronJs.

The Electron is an open-source desktop application framework. You might be wondering what's special with Electron JS. The key takeaway is that you won't need to spend time learning new programming languages to develop desktop apps if you're already familiar with JS, HTML, and CSS. You can get straightaway into building the desktop app if you have adequate web development skills. On another note, Electron JS enables you to create cross-platform programs that run on Windows, Mac OS X, and Linux

Electron combines Chromium and Node.js, a JavaScript runtime based on Chrome V8 JavaScript engine.

For more: https://www.electronjs.org/

Reversing .Exe Files

In the Windows platform, reversing the electron application is quite simple. Please follow the steps outlined below.

- You can download the Notable app from the following link.
- Install the downloaded application.
- Right-click on the application > Open file location.
- You may locate the application installed directory using the steps indicated above. There might be a chance to find something interesting and sensitive in this directory.
- Navigate the "resources" directory.
- You can now see the app.asar file, which will be utilized for further examination and analysis.

Reversing .asar

For those who are not so familiar with asar files, an asar file is an archive that contains the source code for electron applications. Let's try to reverse the .asar file that we obtained in the previous step.

Note: To continue, you'll need to install node js and npm. The installation instructions can be found at the following URL: https://nodejs.org/en/download/.

- Install asar by npm install -g asar
- Extract app.asar file by asar extract app.asar dest-folder
- .asar package has been extracted, now we can examine the source code.

Identifying dependencies' vulnerabilities

Third-party or open-source code is turning out to be inevitable for most of the applications that we deal with on a daily basis. If any dependencies utilized in applications are insecure, the programme is likewise vulnerable. Which would definitely lead to undesired results.

The npm audit command comes handy here. It will assist you in identifying known vulnerabilities in your project's dependencies. Let's now take a look at how to execute the npm audit command below.

To scan the Notable app, use the command following inside the extracted directory.

```
npm audit
```

You will most likely see an error like this. To resolve the problem, run the command.

```
npm i --package-lock-only
```

Run the npm audit command again.

```
npm audit
```

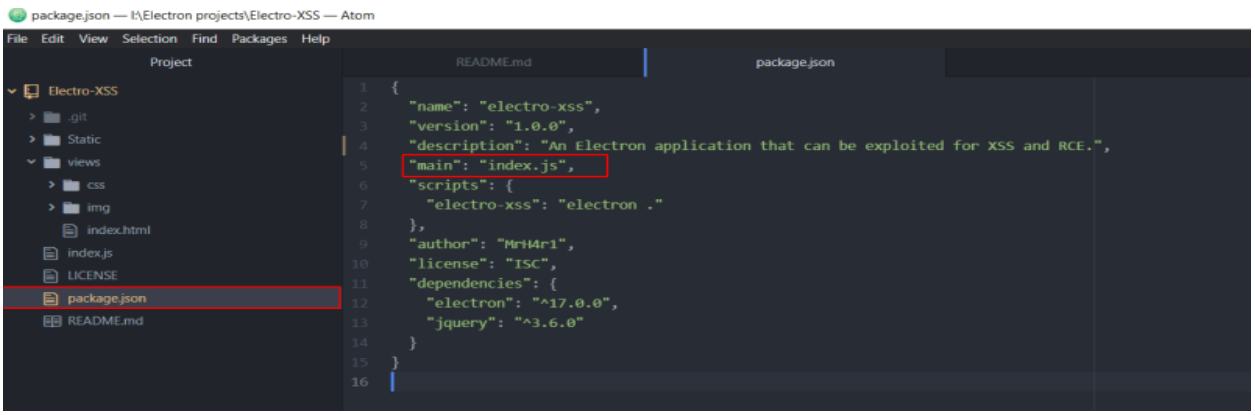
Electron XSS exploitation

Install the Electro-XSS vulnerable app by downloading it from here. The installation instructions can be found in the git repository.

Before we begin exploiting the application, let's have a look at the source code and identify the security flow. Open the Electro-XSS project in any text editor to begin exploring the source code.

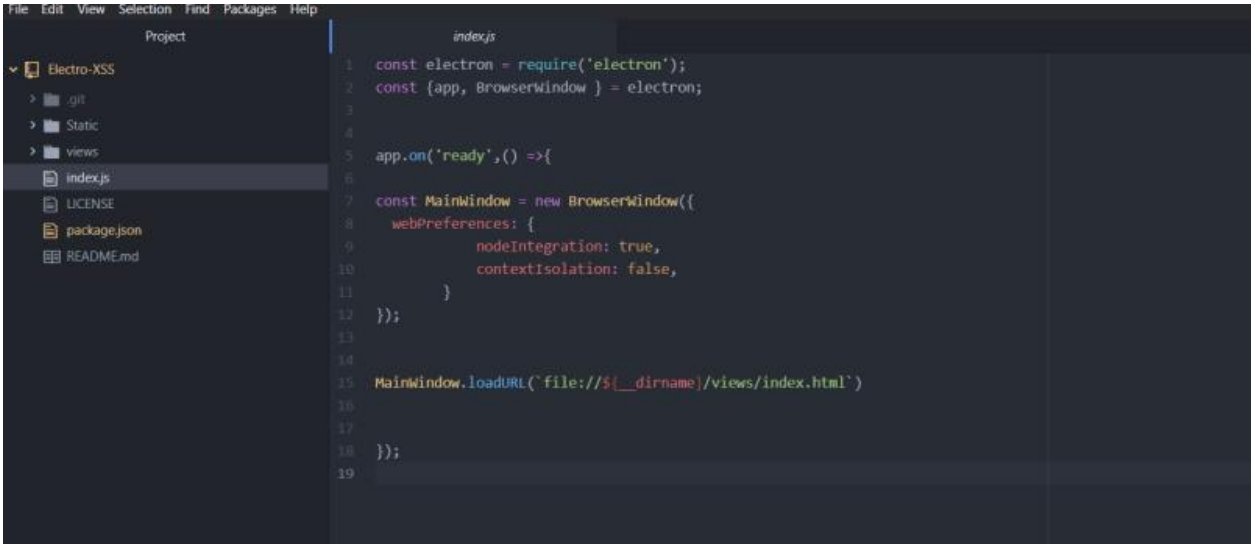
File: Package.json

The package.json file contains various metadata, as well as dependency and version details.

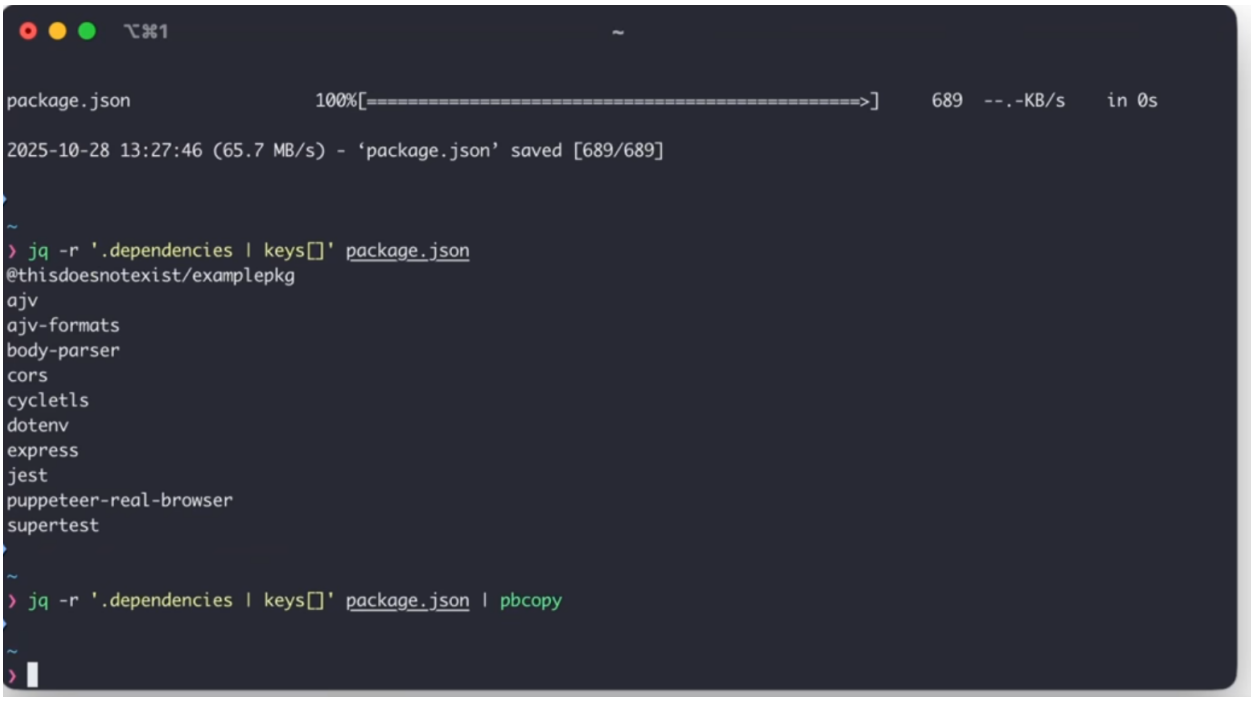


You can see the "main" field in the package.json file, which points to the project primary entry point. Let's take a look at index.js, which is our primary entry point in this case.

File: Index.Js



https://app.jsmon.sh/tools/npm-validator/



Get the package.json and check on the below link: https://app.jsmon.sh/tools/npm-validator/

This application uses the app and BrowserWindow electron modules.

app: This module can be used to manage your application's events.

BrowserWindow: The application windows can be created and maintained using this module.

If the application's basic startup is completed, the ready event function callback is invoked, and the BrowserWindow builds the main window, and then loads the index.html file using MainWindow.loadURL.

File: Index.Html

The application's frontend is the index.html page. The frontend is what you see and interact with on your browser.

- Go to the views folder and locate the index.html file.

- At the bottom of the index.html file, look for the jQuery script.

Payload:

Tools:

1. Echo mirage
2. Procmon
3. Strings.exe
4. Sysinternals Suite
5. Nmap
6. Testssl
7. Process Hacker
8. Dnspy/ Dot Peek/ VB decompiler
9. Metasploit (To create Mal. DLL file)
10. Fiddler/Burpsuite
11. Wireshark
12. Ollydbg
13. .Net Reflector
14. Winhex

Electron Security Best Practices

Electron's official website itself provides a great set of security best practices and, if followed properly, help mitigate common security issues that one may encounter during the testing. Some of the security best practices to follow while developing Electron applications are:

1. **Ensure use of Secure Protocols:** The application should use secure communication, data transfer and other activities. It is recommended to use HTTPS, FTPS, WSS, etc., over HTTP, FTP or WS.

Risk: An attacker might be able to eavesdrop on the communication over an insecure channel and perform a Man-in-the-Middle attack to steal sensitive information.

2. **Disable nodeIntegration for Untrusted Sources/Remote Content:** The application should not have nodeIntegration enabled in any renderer function such as BrowserWindows, BrowserView, etc that may load remote content. Ensure that the nodeIntegration is disabled by setting its value to "false".

Risk: An attacker might be able to utilize the nodeIntegration to perform attacks such as arbitrary JavaScript execution leading to cross-site scripting, local file read or even code execution in certain scenarios.

3. **Context Isolation:** Electron provides a feature to run code in preload scripts and in Electron APIs in a dedicated JavaScript context. Along with disabled nodeIntegration, it is also recommended to use Context Isolation.

Risk: An attacker might be able to utilize the insecure preload scripts in absence of Context Isolation to bypass the nodeIntegration or Sandbox to perform further impactful attacks.

4. **Enable WebSecurity:** Disabling webSecurity will disable the same-origin policy and set allowRunningInsecureContent property to true. In other words, it allows the execution of insecure code from different domains. Ensure that the webSecurity is enabled.

Risk: An attacker might be able to perform multiple attacks as the protection layer, i.e. WebSecurity is disabled.

5. **Implement a CSP:** Content-Security-Policy (CSP) acts as a defence-in-depth mechanism against some attacks such as Cross-Site Scripting. Implementing a CSP can help to prevent cross-site scripting attacks. Although it is not a permanent solution, a strong CSP can prevent this type of attack from happening unless bypassed.

Risk: An attacker might be able to perform attacks such as Cross-Site-Scripting and impact the application & its users in the absence of CSP.

6. **Disable Running Insecure Content:** The electron doesn't allow websites over HTTPS to load & execute content from insecure sources. However, if the property allowRunningInsecureContent is set to true, the protection is disabled.

Risk: An attacker might be able to load and execute content from insecure sources if the allowRunningInsecureContent is set to true.

7. **Disable Popups:** The applications using <webview> might require the pages and scripts to be loaded in <webview> tag to open new windows. The allowpopups attribute enables them to create new BrowserWindows using the window.open() method. <webview> tags are otherwise not allowed to create new windows.

Risk: An attacker might be able to abuse this behavior for performing attacks such as UI Redressing.

8. **Verify Webview Options:** A Webview created in a renderer process that does not have Node.js integration enabled will not be able to enable integration itself. However, a WebView will always create an independent renderer process with its own webPreferences. It is a good idea to control the creation of new <webview> tags from the main process and to verify that their webPreferences do not disable security features.

Risk: Since <webview> live in the DOM, an attacker might be able to create them by a script running on the website even if Node.js integration is otherwise disabled.

9. **Disable openExternal with Untrusted Content:** Shell's openExternal() allows opening a given external protocol URI with the desktop's native utilities. For instance, on macOS, this function is similar to the 'open' terminal command utility and will open the specific application based on the URI and filetype association.

Risk: An attacker might be able to utilize the misconfigured openExternal to compromise the user's host.

10. **Use Latest Version of Electron:** It is recommended to use the latest version of the Electron in order to ensure that any previous/known vulnerabilities are not impacting the application.

Risk: An attacker would be able to enumerate known vulnerabilities and use the public exploits/write exploits in order to impact the application.

Tool Repository: https://github.com/doyensec/electronegativity

Installation

npm install @doyensec/electronegativity -g

Using Electronegativity

a. Navigate to the Electron Application's Installation Directory which contains the asar file.

b. Open the terminal and run the following command:

CMD: electronegativity.cmd -i /file_path

Ex: electronegativity.cmd -i .

c. Observe that the tool returned with the misconfiguration and vulnerabilities.

Refer:

- https://payatu.com/blog/an-intro-to-electron/
- https://redfoxsec.com/blog/hacking-electron-apps/