Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.4, Aug. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.4.573

Electron 기반 협업 프로그램 취약점 분석*

이 효 민,[1†] 장 연 석,[1] 권 용 희,[1‡] 임 은 지,[2] 김 종 민,[1] 박 진 우[3]
[1]고려대학교 (학생), [2]성신여자대학교 (학생), [3]가톨릭대학교 (학생)

Analysis of Vulnerability in Electron Based Collaboration Tools*

Hyomin Lee,[1†] Yeonseok Jang,[1] Yonghee Kwon,[1‡]
Eunji Lim,[2] Jongmin Kim,[1] Jinwoo Park[3]
[1]Korea University (Student), [2]Sungshin Women's University (Student),
[3]the Catholic University of Korea (Student)

요 약

COVID-19 팬데믹 상황 속 비대면 업무의 비중이 늘어나면서 협업 프로그램 시장이 빠르게 성장하고 있다. 시장 규모가 커짐에 따라 협업 프로그램에 대한 취약점이 꾸준히 공개되고 있으며, 이로 인해 협업 프로그램의 보안성에 관한 관심이 증가하고 있다. 본 논문에서는 다수의 협업 프로그램이 Electron 프레임워크를 기반으로 하고 있다는 점에 주목하여 국내 Electron 기반의 협업 프로그램에 대한 취약점 분석을 수행한 결과를 소개하고, 분석 결과를 바탕으로 Electron 기반 애플리케이션의 보안성을 높이기 위한 대응 방안을 제안한다.

ABSTRACT

As the proportion of non-contact work is increasing in the situation of COVID-19 pandemic, the collaboration program market is growing rapidly. As the size of the market grows, vulnerabilities in collaborative programs are constantly being disclosed which increases interest in the security of collaborative tools. In this paper, we introduce the results of vulnerability analysis on Electron-based collaboration programs, noting that a number of collaboration programs are based on the Electron framework, and propose countermeasures to enhance the security of Electron-based applications.

Keywords: Electron, Collaboration Tool, RCE

I. 서 론

COVID-19의 확산으로 다수의 기업이 비대면 업무 방식을 채택함에 따라, 기업을 주 사용층으로 하는 협업 프로그램의 사용량이 급격히 증가하고 있다. 메시지 전송을 포함한 일정 관리, 안내사항 전달, 파일 공유 등 협업에 필요한 다양한 기능을 제공하는 협업 프로그램이 비대면 업무의 효율성을 높이기 위한 도구로서 주목 받고 있는 것이다. 실제로 COVID-19 이후 북미의 협업 소프트웨어 시장 규모는 13억 달러에서 357억 달러로 빠르게 성장했으며[1], 시장 조사기관 International Data Corporation에 따르면, 2021년 협업 소프트웨어 시장의 규모는 430억 달러까지 성장할 것으로 전망된 바 있다.

이처럼 협업 프로그램의 사용량이 증가하자 협업 프로그램이 해커들의 주요한 표적이 되고 있으며, 이에 따라 협업 프로그램에 대한 보안성 확보의 필요성이 강조되고 있다. 따라서 본 연구에서는 협업 프로그램을 대상으로 한 취약점 분석을 통해 보안성을 검증하고자 한다.

Received(04. 01. 2021), Modified(06. 21. 2021), Accepted(06. 23. 2021)
* 본 연구는 KITRI 차세대 보안 리더 양성 프로그램(Best of the Best)의 연구 지원으로 수행되었습니다.
† 주저자, lhm1024@korea.ac.kr
‡ 교신저자, hee980410@korea.ac.kr(Corresponding author)

취약점 점검에 앞서 점검 대상을 선정하는 과정에서 다수의 협업 프로그램이 Electron 기반이라는 사실에 주목하였으며 (Electron 공식 사이트에 따르면, Electron 기반의 생산성, 소셜 네트워킹, 비즈니스 관련 애플리케이션은 363개에 달하고[2], Flow 협업 프로그램을 비롯한 일부 국내 협업 프로그램의 경우 공식 사이트에 등록되어 있지 않았기 때문에 실제 Electron 기반 협업 프로그램은 이보다 더 많을 것으로 보인다), Electron 기반의 대표적인 협업 프로그램에는 슬랙, 디스코드, 트렐로, 잔디 등이 있다.

최근 이러한 주요 협업 프로그램을 대상으로 한 취약점 보고가 계속해서 이어지고 있다. 본 연구에서는 공개된 여러 취약점 보고 중 nodeIntegration과 contextIsolation 설정과 관련한 취약점 사례에 주목해 해당 설정과 보안성의 관계에 대해 살펴보고, 이를 바탕으로 국내 협업 프로그램에 대한 점검을 진행하였다.

본 논문의 기여는 다음과 같다. 우선, 국내의 협업 프로그램을 대상으로 한 Electron 관련 취약점 보고가 없었다는 점에서, 공개된 취약점에 대한 점검을 수행했다는 점에서 의미가 있다. 또한, 협업 프로그램을 대상으로 한 취약점 점검 결과를 바탕으로 Electron 기반 데스크톱 애플리케이션에서 발생할 수 있는 취약점에 대한 대응 방안을 제시함으로써, Electron 기반 데스크톱 애플리케이션의 보안성 검증 및 보안성 향상에 기여할 수 있다.

본 논문의 2장에서는 Electron 기반 애플리케이션의 구조와 Electron 기반 애플리케이션에서 발생할 수 있는 취약점과 관련한 배경 지식을 소개하고, 3장에서 이와 관련해 실제 발생한 취약점 사례를 설명한다. 4장과 5장에서는 실제 국내 협업 프로그램에 대해 취약점 분석을 수행한 과정과 결과를 서술하며, 6장에서 대응 방안을 제시한 뒤 7장에서 결론을 내린다.

II. Electron 보안

Electron은 오픈 소스 프레임워크 중 하나로, Electron 프레임워크를 이용하면 적은 비용으로 기존에 개발된 웹 애플리케이션을 여러 플랫폼(Windows, MacOS, Linux)에서 동작 가능한 데스크톱 애플리케이션으로 만들 수 있다.

이번 장에서는 Electron 애플리케이션의 동작 구조를 설명하고, Electron 기반 애플리케이션에서 발생할 수 있는 취약점과 관련한 배경 지식을 소개한다.

2.1 Electron 구조

Electron 애플리케이션은 Main process와 Renderer process라는 2개의 process로 구성되며, 두 process가 서로 통신하며 동작한다(Fig 1).

Main process에서는 Node.js가 실행되며, 프로그램의 로직을 실행한 다음 Renderer process를 생성한다. Renderer process는 화면을 렌더링하며, 렌더링한 결과의 출력에는 Chromium이 사용된다.

일반적으로 웹 애플리케이션은 PC의 네이티브 리소스에 접근할 수 없다. 하지만 Electron 애플리케이션은 그 구조상, 취약한 설정값을 가지는 경우 Renderer process에서 Main process의 Node.js API를 통해 네이티브 리소스에 접근할 수 있게 되어 해커에 의해 악의적인 행위가 수행될 수 있다. nodeIntegration과 contextIsolation이라는 특정 옵션의 설정값에 따라 취약성의 유무가 결정되는데, 이에 대해서 이어지는 절에서 서술한다.

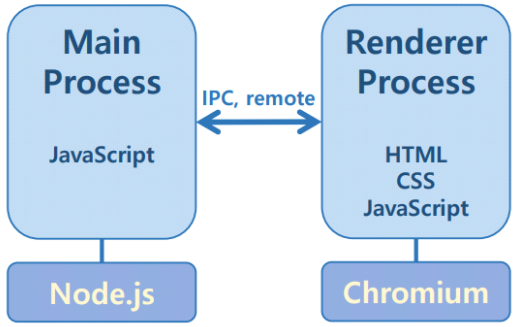

Fig. 1. Structure of Electron Framework

2.2 nodeIntegration

Main process와 Renderer process가 서로 통신하며 동작하는 애플리케이션의 경우, nodeIntegration 옵션을 통해 Renderer process에서의 Node.js API에 대한 접근을 관리할 수 있다.

nodeIntegration의 값이 true일 경우, Renderer process에서 Node.js API를 호출할 수 있고 해당 값이 false일 경우, Node.js의 기능 중 global 객체, process 객체, require 함수, module 객체 등을 사용할 수 없게 된다 [6]. 즉,

nodeIntegration의 값이 true인 경우 Renderer process에서 Node.js API를 통해 네이티브 리소스에 접근할 수 있게 되므로 취약성이 발견될 가능성이 있다.

2.3 contextIsolation

contextIsolation 설정은 Renderer process와 Electron 프레임워크의 javascript 컨텍스트 분리 여부를 결정한다. 해당 값은 기본적으로 false로 설정되어 있다. 이 값이 false로 설정되어 있을 경우 javascript 컨텍스트가 분리되어 있지 않기 때문에, 공격자가 XSS 등의 방식을 통해 Electron 내부 프레임워크의 로직에 대한 prototype pollution을 수행함으로써 악성 행위 수행이 가능하다. 따라서, contextIsolation의 값이 false인 경우, prototype pollution을 이용한 취약성이 발견될 가능성이 있다.

2.4 preload 스크립트

BrowserWindow에서 nodeIntegration이 비활성화되면 Renderer process에서 IPC 모듈을 호출할 수 없기 때문에 Main process와 통신이 불가능하다. 이러한 경우, preload 스크립트를 통해 process 간 통신이 가능하도록 할 수 있다. preload 스크립트는 BrowserWindow가 생성되는 시점에 동작하는 스크립트로, Renderer 프로세스가 생성되기 전에 실행된다. Renderer 프로세스에서 접근이 필요한 기능을 preload 스크립트에 정의해둠으로써, 이를 Renderer process에서 사용할 수 있다. Main process와 통신하기 위한 기능을 정의해둔다는 점에서, 공격자는 preload 스크립트에 정의된 기능을 통해 Main process에 영향을 줄 수 있다.

III. 기존 취약점 사례 분석

이번 장에서는 Electron 프레임워크를 이용해 개발된 디스코드 데스크톱 App에서 발생한 Remote Code Execution (RCE) 취약점에 대해 살펴본다.

디스코드는 인스턴트 메신저 소프트웨어로, 미국 기술 미디어 사이트인 CNET에 따르면 2019년 2억5천만 명의 사용자가 사용한 대중적인 소프트웨어이다. 사용자 수가 많은 만큼, 디스코드는 버그 바운티 운영을 통해 보안성 향상을 염두에 두고 있다[7].

디스코드의 RCE 취약점은 2020년 10월 Masato Kinugawa에 의해 공개되었다[8]. 해당 취약점은 3개의 취약점을 이용해 full exploit에 성공했으며, 이용한 세 가지 취약점은 다음과 같다.

1. contextIsolation 옵션의 부재로 인한 prototype pollution
2. iframe에서 발생한 XSS 취약점
3. Electron의 Navigation 제한 우회

현재 세 취약점은 모두 패치가 된 상태이며, 본 장에서는 취약점 패치 전을 기준으로 각 취약점의 발생 원인에 대해 설명한다.

3.1 contextIsolation 옵션의 부재

패치 이전의 디스코드는 nodeIntegration과 contextIsolation 옵션이 모두 false로 설정되어 있었다. nodeIntegration이 false이기 때문에 Renderer process에서 require를 비롯한 Node.js 기능을 사용할 수 없지만, contextIsolation이 false이므로 prototype pollution이 가능해 취약했다.

앞서 2장에서 언급했듯 contextIsolation이 비활성화된 경우 Renderer process와 preload 스크립트, Electron 내부 코드의 javascript 컨텍스트 분리가 되지 않기 때문에, Renderer 프로세스에서의 javascript 실행을 통해 preload 스크립트와 Electron 내부 코드에 영향을 줄 수 있다.

Masato는 DiscordNative.nativeModules.requireModule이라는 함수를 prototype pollution을 수행할 대상으로 선정했다. 해당 함수는 특정 모듈을 호출할 수 있는 함수로, preload 스크립트에서 이 함수를 export하고 있어 Renderer process에

```
1  RegExp.prototype.test = function () {
2    return false;
3  }
4  Array.prototype.join = function () {
5    return “calc”;
6  }
7  DiscordNative.nativeModules.requireModule (
8  ‘discord_utils’).getGPUDriverVersions ();
```

Fig. 2. Remote Code Execution PoC

서 호출할 수 있었다.

requireModule을 통해 호출 가능한 모듈들의 함수 중, discord_utils 모듈의 getGPUDriverVersions 함수에서 prototype pollution을 통한 RCE 취약점이 발견되었다. Fig 2.의 javascript는 RCE 취약점이 발생할 수 있다는 개념 증명에 해당하는 코드로, Renderer process에서 실행 시 계산기가 띄워진다.

getGPUDriverVersions 함수는 nvidiaSmiPath 의 nvidia-smi.exe 파일을 실행시키기 위해 execa 함수를 사용하고 있었다(Fig 3). 위 개념 증명 코드에서 prototype pollution의 대상은 RegExp.prototype.test 함수와 Array.prototype.join 함수로, 두 함수 모두 execa 함수에 의해 호출되는 javascript 내장 함수이다(Fig 4). RegExp.prototype.test 함수는 execa 함수의 인자인 commandFile이 실행 가능한 파일에 대한 경로인지 확인하기 위해 호출되며, Array.prototype.join 함수는 execa 함수의 인자를 이어붙여 완전한 명령어를 만들기 위해 호출된다. 개념 증명 코드에서는 이 두 함수를 각각 false, "calc"를 반환하는 함수로 재정의한다. 이를 통해 execa 함수는 항상 '실행 가능한 파일'인 'calc'를 실행하게 되며, execa 함수를 호출하는 getGPUDriverVersions 함수를 통해 계산기를 실행할 수 있게 된다.

```
module.exports.getGPUDriverVersions = async () => {
    if (process.platform !== 'win32') {
      return {};
    }

    const result = {};
    const nvidiaSmiPath = `${process.env['ProgramW6432']}/NVIDIA
    Corporation/NVSMI/nvidia-smi.exe`;

    try {
      result.nvidia = parseNvidiaSmiOutput(await execa
      (nvidiaSmiPath, []));
    } catch (e) {
      result.nvidia = {error: e.toString()};
    }

    return result;
  };
```

Fig. 3. *getGPUDriverVersions* Fuction Code

```
// We don't need a shell if the command filename is an executable
const needsShell = !isExecutableRegExp.test(commandFile);


if (parsed.options.forceShell || needsShell) {
    const needsDoubleEscapeMetaChars = isCmdShimRegExp.test(commandFile);

    parsed.command = path.normalize(parsed.command);

    // Escape command & arguments
    parsed.command = escape.command(parsed.command);
    parsed.args = parsed.args.map((arg) => escape.argument(arg,
    needsDoubleEscapeMetaChars));

    const shellCommand = [parsed.command].concat(parsed.args).join(' ');
```

Fig. 4. Part of *execa* Fuction Code

3.2 iframe embeds에서의 XSS 취약점

앞서 언급한 prototype pollution을 위해서는 임의의 javascript 실행이 선행되어야 한다. 본 취약점 사례의 경우, iframe embeds에서 XSS 취약점이 발생해 임의의 javascript 실행이 가능했다.

iframe embeds는 입력된 URL Link로부터 Open Graph Protocol(OGP) 정보(제목, 설명, 대표 이미지)를 가져와 나타내주는 역할을 한다.

디스코드에서는 Content Security Policy를 통해 iframe embeds 삽입이 허용되는 도메인을 제한해두었다(Fig 5). Masato는 해당 도메인 중 sketchfab.com에서 XSS 취약점을 발견했다. 해당 도메인에서 XSS 취약점이 발생한 링크를 https://l0.cm/discord_rce_og.html의 OGP 정보에 포함시킨 후, 해당 주소를 디스코드에 전송함으로써 iframe embeds에서 임의의 javascript 실행이 가능하도록 했다.

```
Content-Security-Policy: [...] ; frame-src https://*.youtube.com
https://*.twitch.tv https://open.spotify.com
https://w.soundcloud.com https://sketchfab.com
https://player.vimeo.com https://www.funimation.com
https://twitter.com https://www.google.com/recaptcha/
https://recaptcha.net/recaptcha/ https://js.stripe.com
https://assets.braintreegateway.com https://checkout.paypal.com
https://*.watchanimeattheoffice.com
```

Fig. 5. Discord CSP List

3.3 Electron의 Navigation 제한 우회

해당 취약점은 상단 창 탐색 제한 우회 취약점으로, CVE-2020-15174를 부여받았다.

Electron 기반 애플리케이션의 iframe에서는 Electron 내장 코드를 로드하지 않기 때문에, iframe에서 javascript 내장 함수를 재정의하더라도 Electron 내장 코드에는 반영되지 않는다. 따라서, prototype pollution을 통한 RCE를 수행하기 위해서는 iframe을 벗어나 최상단 BrowserWindow에서 javascript를 실행해야 한다. 이를 위해서는 iframe에서 새 창을 열거나 최상단 BrowserWindow를 다른 URL로 이동시켜야 한다. 두 경우는 각각 new-window 이벤트와 will-navigte 이벤트에 해당하며, 해당 이벤트 핸들러의 정의에서 이동 가능한 URL을 제한하는 코드가 존재했다(Fig 6).

이러한 제한이 있음에도, iframe에서 최상단 BrowserWindow를 다른 URL로 이동하는 행위가 will-navigate 이벤트를 발생시키지 않아 취약점이 발생했다. 실제로, iframe의 origin과 최상단 BrowserWindow의 origin이 같은 경우에는 will-navigate 이벤트가 발생하였지만, 같지 않은 경우 이벤트가 발생하지 않아 제한을 우회할 수 있었다고 한다.

위 세 가지 취약점을 이용하여 RCE가 가능했으며, 그 과정을 요약하자면 다음과 같다:

iframe의 XSS 취약점을 이용해, 최상단 BrowserWindow를 prototype pollution을 수행하는 코드가 포함된 사이트의 URL로 이동시킴으로써 RCE를 수행한다.

이 취약점 사례를 통해 Electron 기반 애플리케이션에서 nodeIntegration, contextIsolation 옵션의 중요성을 확인할 수 있다. 자체적으로 버그 바운티

```
mainWindow.webContents.on ( 'new-window', (e, windowURL, frameName,
disposition, options) => {
    e.preventDefault ();
    if (frameName.startsWith (DISCORD_NAMESPACE) && windowURL.
    startsWith (WEBAPP_ENDPOINT)) {
      popoutWindows .openOrFocusWindow (e, windowURL, frameName,
      options);
    } else {
      _electron.shell.openExternal (windowURL);
    }
  });
  [...]
  mainWindow.webContents.on ( 'will-navigate', (evt, url) => {
    if (! insideAuthFlow &&! url.startsWith (WEBAPP_ENDPOINT)) {
      evt.preventDefault ();
    }
  });
```

Fig. 6. *new-window* Event Handling Code

프로그램을 진행해온 디스코드에서 설정값 미흡으로 인해 파급력이 큰 취약점이 나왔다는 점에서 Electron 기반 애플리케이션에서 발생할 수 있는 보안 취약점에 대한 인식이 부족한 것으로 보인다. 이어지는 장에서는 이 가정을 검증하기 위해 국내 협업 프로그램을 대상으로 취약점을 점검한 결과에 관해 서술한다.

IV. 보안성 검증 방법

이번 장에서는 보안성 검증 방법에 관해 서술한다. 4.1에서 Electron 기반 애플리케이션의 소스코드를 확인하는 법을 소개하고, 4.2에서는 취약성을 판단할 수 있는 방법에 설명한다. 4.3에서는 electronegativity라는 분석 도구를 통해 Electron 기반 애플리케이션을 사전 분석하는 방법을 소개한다.

4.1 코드 확인

Electron 기반 애플리케이션은 애플리케이션이 설치된 파일 경로에서 ASAR 확장자의 파일을 찾을 수 있다(Fig 7, 주로 app.asar 라는 이름의 파일이 해당 경로에 존재한다). 이 파일은 Electron 기반 애플리케이션의 구동에 필요한 리소스 파일과 Node.js 소스코드 파일을 포함하고 있는 아카이브 포맷의 파일이다. Node.js 패키지인 asar 패키지의 extract 명령어를 이용하여 ASAR 파일에 포함된 파일을 추출할 수 있으며 (asar extract), 그 결과의 예시는 Fig 8.과 같다.

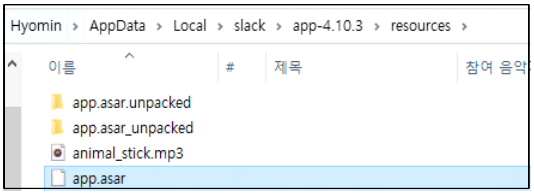

Fig. 7. Slack app.asar

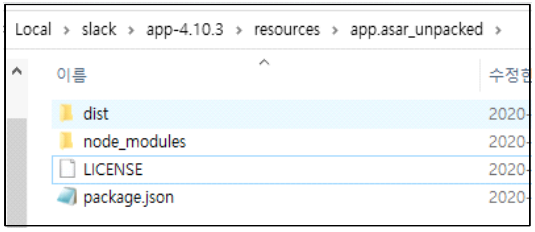

Fig. 8. After ASAR File Unzipped

4.2 취약성 판단

취약성을 판단하기 위해 가장 먼저 확인해야 할 것은 BrowserWindow 객체를 생성하는 부분이다. 해당 코드는 새 창을 생성할 때마다 존재하기 때문에 소스코드의 여러 부분에 존재한다(Fig 9).

BrowserWindow 객체 생성 시 전달되는 인자값 중 nodeIntegration, contextIsolation의 값을 확인해보아야 한다(Fig 10).

nodeIntegration이 true라면, 생성된 창에서 Node.js API를 호출할 수 있어 잠재적으로 RCE 취약점이 발생할 수 있으므로 취약한 설정이다.

nodeIntegration이 false라면, preload 스크립

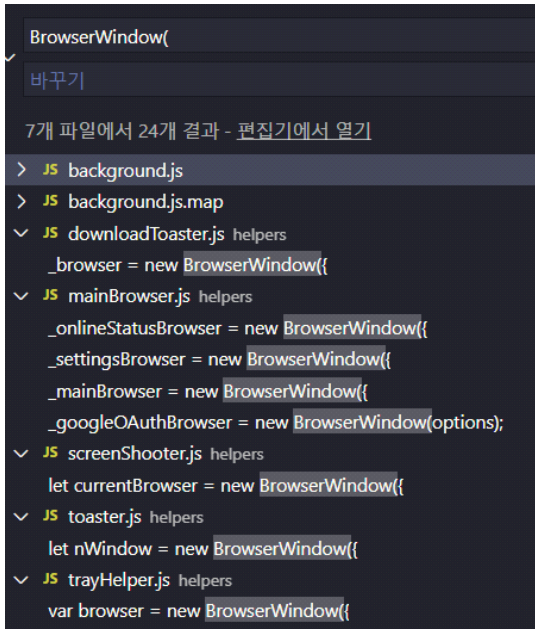

Fig. 9. Codes Using *BrowserWindow*

```
_settingsBrowser = new BrowserWindow({
  width: 540,
  height: settingBrowserHeight,
  show: false,
  fullscreenable: false,
  autoHideMenuBar: true,
  webPreferences: {
    nodeIntegration: true,
    contextIsolation: false
  }
});
```

Fig. 10. *BrowserWindow* Settings

```
webPreferences: {
  nodeIntegration: false,
  preload: path.join(__dirname,
  '/view/js/preload.js'),
  contextIsolation: false
},
```

Fig. 11. *preload* Settings

트에 정의된 코드에 취약한 부분이 있는지 확인해보아야 한다(Fig 11.의 경우 /view/js/preload.js 코드를 확인해 보아야 함).

contextIsolation이 false라면, Renderer process와 Electron 프레임워크의 javascript 컨텍스트 가 분리되어 있지 않다. 따라서 XSS 취약점이 존재하는 경우, prototype pollution을 통해 코드 실행 흐름이 조작될 수 있다. 이 경우 preload 스크립트와 Electron 내부 코드에서 prototype pollution을 통해 코드 실행 흐름을 임의로 바꿀 수 있는지 확인해보아야 한다.

두 설정값 확인 결과 취약한 설정이 존재하는 경우, 해당 BrowserWindow와 관련한 소스코드를 대상으로 분석을 진행한다. 구체적인 분석을 하기에 앞서, 해당 BrowserWindow를 생성하는 코드에 공격자가 도달할 수 있는지, 해당 BrowserWindow가 제공하는 기능이 다양하여 점검할 수 있는 요소가 충분히 있는지 확인하여야 한다. 이를 위해, 공격자가 임의로 도달할 수 있는 이벤트 핸들러 함수, 혹은 preload 스크립트에서 Main process와 통신하는 코드 등에 대해 코드 실행 흐름을 파악하고 취약성 여부를 확인해야 한다(Fig 12). BrowserWindow를 생성하는 코

```
function _attachMainBrowserEvents() {
  _mainBrowser.on('page-title-updated',
  _onTitleUpdatedMain);
  _mainBrowser.on('closed', _onClosedMain);
  _mainBrowser.on('close', _onCloseMain);
  _mainBrowser.on('show', _onShowMain);
  _mainBrowser.on('focus', _trayHelper.
  setDefaultIcon);
  _mainBrowser.on('app-command',
  _onAppCommandMain);
  _mainBrowser.webContents.on('will-navigate',
  _onWillNavigateMain);
  _mainBrowser.webContents.on('did-navigate',
  _onDidNavigateMain);
  _mainBrowser.webContents.on('new-window',
  _onNewWindowMain);
```

Fig. 12. In-Use Event Handler

드가 실행되기 전 검증이 수행되는 경우와 환경설정과 같은 제한적인 기능만 제공하는 경우를 구체적인 분석을 하기에 부적절한 예시로 들 수 있다.

이벤트 핸들러 함수의 경우, new-window와 will-navigate 이벤트에 대한 핸들러를 살펴보아야 한다. 두 이벤트 모두 애플리케이션에서 URL Link를 열 때 발생한다. 애플리케이션에서 새 창으로 신뢰할 수 없는 사이트를 열게 된다면 임의의 javascript가 실행될 수 있으므로, 해당 이벤트 핸들러 함수에는 기본 브라우저로 Link를 열 것인지, 애플리케이션에서 새 창으로 열 것인지 결정하는 검증 루틴이 존재한다. 임의의 javascript 가 실행된다면 다른 취약점으로 이어질 수 있으므로, 해당 검증 루틴이 취약한지를 검증해 보아야 한다.

preload 스크립트에는 Main process와 통신하기 위한 함수들이 정의되어 있다. 스크립트에 정의된 함수들을 대상으로 인자값을 조절해보면서 취약점 여부를 확인해야 한다. 예를 들어, 특정 URL을 애플리케이션의 창에서 열어주는 함수가 preload 스크립트에 정의된 경우, 악의적인 사이트를 로드시켜 임의의 javascript 실행이 가능한지 점검해볼 수 있다.

4.3 electronegativity

electronegativity는 Electron 기반 애플리케이션에서 잘못 구성된 설정이나 보안 안티패턴을 식별해주는 도구이다. 2017년 Blackhat USA 컨퍼런스에서 공개되었다.

도구에서 Electron 기반 애플리케이션을 점검하는데 사용하는 체크리스트 항목은 38가지이며, nodeIntegration나 contextIsolation, preload 스크립트에 대한 항목을 포함하고 있다(Fig 13).

아래는 실제 애플리케이션을 대상으로 도구를 실행한 결과이다(Fig 14). 각 항목의 점검 결과와 해당하는 소스코드의 위치, 부가설명이 나타나 있다. 점검 결과 위험도가 HIGH로 분류된 CONTEXT_ISOLATION_JS_CHECK, NODE_INTEGRATION_JS_CHECK 항목에 해당하는 소스코드를 살펴보면, 실제로 취약한 설정값임을 확인할 수 있다(Fig 15).

V. 국내 협업 프로그램 검증 결과

이번 장에서는 4장의 보안성 검증 방법을 바탕으로 국내 협업 프로그램을 검증한 결과에 관해 설명한다.

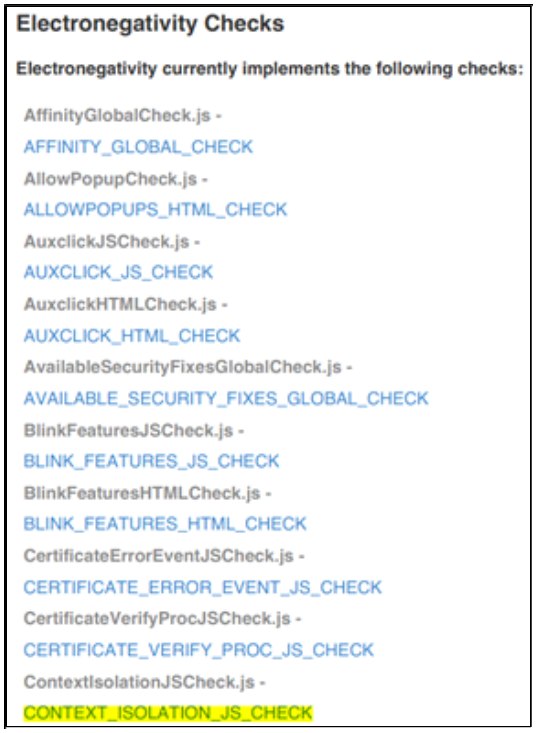

Fig. 13. Part of Electronegativity Checklists

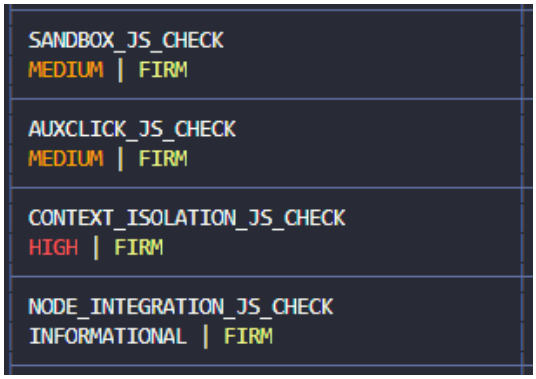

Fig. 14. Results of Running Electronegativity

5.1에서 설정값에 대한 검증 결과를 소개하고, 5.2, 5.3에서는 취약성이 발견된 A 협업 프로그램과 B 협업 프로그램 사례를 설명한다.

5.1 설정값 검증

국내 협업 프로그램의 점유율에 대한 공개된 통계 자료가 없으므로, 본 연구에서는 구글 플레이 스토어에서 "협업" 키워드로 검색한 결과 상위에 노출된 목록

```
Object.assign(options, {
  modal: true,
  parent: _mainBrowser,
  resizable: true,
  width: 620,
  height: 800,
  show: true,
  fullscreenable: false,
  autoHideMenuBar: true,
  webPreferences: {
    nodeIntegration: true,
    webviewTag: true,
    contextIsolation: false
  },
});
_googleOAuthBrowser = new BrowserWindow(options);
```

Fig. 15. Vulnerable Settings Discovered Using Electronegativity

중 누적 다운로드 수가 1만 이상인 국내 협업 프로그램을 우선 선정하였다. 그 중 Electron 기반 데스크톱 애플리케이션이 존재하는 경우에 한해 nodeIntegration과 contextIsolation의 설정값을 확인하였다. doyensec이 공개한 electronegativity 도구를 이용하였으며, 그 결과는 Table 1.과 같다.

contextIsolation부터 살펴보자면, 5개의 협업 프로그램 중에서 안전한 값인 true로 설정된 경우는 없었다 (contextIsolation 의 기본값은 false이다).

반면, nodeIntegration은 각 협업 프로그램마다 설정값에 차이가 있었다. E 프로그램의 경우 모든 window가 안전한 값인 false로 설정되어 있었으며, A, C 협업 프로그램의 경우 안전한 값인 false로 설정된 window와 안전하지 않은 값인 true로 설정된 window가 모두 존재했다. D 협업 프로그램은 모든 window가 true로 설정되어 있어 취약했다. B 협업 프로그램의 경우 nodeIntegration 값이 안전한 값인 fal

Table 1. Settings Check on Collaboration Programs

Collaboration Program	nodeIntegration	contextIsolation
Program A	false + true	false
Program B	false*	(default value)
Program C	false + true	(default value)
Program D	true	(default value)
Program E	false	(default value)

se임에도 취약성이 발견되었는데, 이에 대해서는 5.3.1에서 자세히 다루도록 한다.

5.2 A 협업 프로그램 검증 결과

5.2.1 nodeIntegration이 true인 BrowserWindow

new-event 이벤트는 window.open 함수의 호출 혹은 anchor 태그 클릭을 통해 새 창이 열릴 때 발생한다. A 협업 프로그램에서는 해당 이벤트에 대한 핸들러 함수에서 RCE로 이어질 수 있는 취약성이 발견되었다. 해당 핸들러 함수에서 nodeIntegration 값이 true인 BrowserWindow 객체를 생성하기 때문에, 생성된 창에서는 require와 같은 Node.js API를 호출할 수 있게 된다(Fig 16).

Fig 19.에 나타난 코드가 실행되기 전 두 차례의 검증이 수행되지만, 부적절한 함수의 사용으로 우회가 가능했다(Fig 17).

첫 번째 조건문에서는 새로 생성된 창에서 접속할

```
}else{
  Object.assign(options, {
    modal: true,
    parent: _mainBrowser,
    resizable: true,
    width: 620,
    height: 800,
    show: true,
    fullscreenable: false,
    autoHideMenuBar: true,
    webPreferences: {
      nodeIntegration: true,
      webviewTag: true,
      contextIsolation: false
    },
  });
  _googleOAuthBrowser = new BrowserWindow(options);
```

Fig. 16. Program A, Vulnerable Code 1

URL이 특정 domain 인지 여부를 확인한다. 일반적으로, URL로부터 domain 혹은 hostname을 추출한 뒤, 상수 문자열과 비교하는 것이 바람직하다. 하지만 Fig 17.의 코드에서는 indexOf 함수를 이용해 URL 이 특정한 상수 문자열(DROPBOX_API_URL,

```
if ( !util.isInternalUrl(url) &&
    url.indexOf(constant.URL.DROPBOX_API_URL) === -1 &&
    url.indexOf(constant.URL.GOOGLE_DRIVE_API_URL) === -1
){
  event.preventDefault();
  require('electron').shell.openExternal(url);
}else{
  if (frameName) {
    if (frameName === 'googleAuth') {
      event.preventDefault();
      var urlInfo = {
        openUrl: url,
        userAgent: 'Chrome',
      };

      if( _googleOAuthBrowser ){
        event.newGuest = _googleOAuthBrowser;
        _googleOAuthBrowser.webContents.send('load-auth-url', urlInfo);
        _googleOAuthBrowser.show();
      }else{
        Object.assign(options, {
          modal: true,
          parent: _mainBrowser,
          resizable: true,
          width: 620,
          height: 800,
```

Fig. 17. Program A, Vulnerable Code 2

GOOGLE_DRIVE_API_URL)을 포함 여부만을 확인한다. 따라서, 공격자는 URL에 해당 문자열을 단순히 포함하는 것만으로 검증을 우회할 수 있게 된다. GET 파라미터를 이용해 상수 문자열을 URL에 포함시킨 예시가 Fig 18.에 나타나 있다.

뒤이은 조건문에서 "googleAuth" 문자열과 비교되는 frameName 변수는 window.open 함수에 전달되는 두 번째 인자로, Fig 18.에서와 같이 함수 호출 시 인자로 전달해 값을 임의로 설정할 수 있다.

Fig 18.을 통해 생성된 창은 nodeIntegration 값이 true이므로 Node.js API를 실행함으로써 네이티브 리소스에 접근할 수 있다.

5.2.2 shell.openExternal의 인자에 대한 검증 부재

Electron의 shell.openExternal 함수는 인자로 전달된 외부 프로토콜 URL을 데스크톱의 기본 프로그램으로 연다. 해당 함수를 이용해 사용자의 컴퓨터에서 프로그램 실행이 가능하기에 전달되는 URL의 프로토콜에 대한 검증 로직이 있어야 하지만, A 협업 프로그램의 경우 인자에 대한 검증 로직이 없다. 따라서, 잠재적으로 RCE 취약점이 발생할 수 있다(Fig 19).

5.3 B 협업 프로그램 검증 결과

5.3.1 preload 스크립트에서 과도한 접근 범위 설정

preload 스크립트는 nodeIntegration이 false인 경우 (Render process에서 Node.js API의 실행을 제한하는 경우), 해당 스크립트를 통해 Renderer process에서 실행을 허용할 Node.js API를 전역 변수에 미리 정의해둠으로써 Renderer process에서 Node.js API의 실행을 부분적으로 허용하기 위해 사용된다.

B 협업 프로그램의 경우, nodeIntegration이 false이며 preload 스크립트를 위와 같은 목적으로 사용하고 있다. 하지만, 실행을 허용하는 범위가 너무 넓어서 문제가 발생한다.

Fig 20.의 코드에서 Renderer process에서 접근이 가능한 window.nodeRequire 전역변수에 require를 대입한다. require는 Node.js에서 모듈을 불러올 때 사용하는 함수로, 해당 함수를 이용해 다른 모듈을 불러와 여러 가지 행위를 수행할 수 있다. 대표적인 예로, 자식 process를 생성 및 호출할 수 있게 해주는 child_process 모듈을 불러오면, 임의의 파일 혹은 명령을 실행할 수 있게 된다. 해당 모듈을 이용해 사용자의 컴퓨터에서 계산기를 실행시키는 명령어는 Fig 21.과 같다.

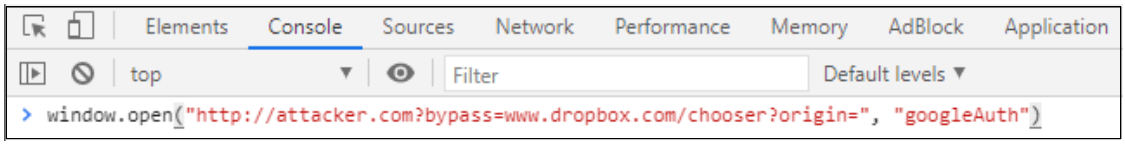

Fig. 18. RCE PoC (Program A, Vulnerability Related to *BrowserWindow*)

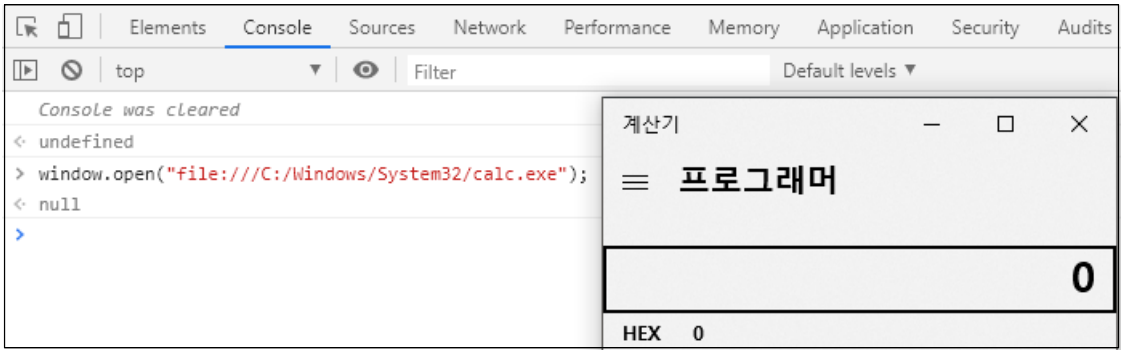

Fig. 19. RCE PoC (Program A, Vulnerability Related to *shell.openExternal*)

```
window.nodeRequire = require;
delete window.require;
delete window.exports;
delete window.module;
```

Fig. 20. Program B, Vulnerable Code 1

```
window.nodeRequire("child_process").exec("calc");
```

Fig. 21. RCE PoC (Program B, Vulnerability Related to preload script)

5.3.2 shell.openExternal의 인자에 대한 검증 부재

5.2.2와 동일하게, shell.openExternal 함수에 인자로 전달되는 URL의 프로토콜에 대한 검증 로직이 있어야 하지만, B 협업 프로그램 역시 인자에 대한 검증 로직이 없으므로 잠재적으로 RCE 취약점이 발생할 수 있다(Fig 22).

VI. 대응 방안

이번 장에서는 5장의 검증 결과를 바탕으로, 협업 도구를 포함한 Electron 기반 데스크톱 애플리케이션의 보안성을 높이기 위한 대응 방안을 서술한다. 전체 대응 방안을 요약한 내용은 Table 2.와 같다.

6.1 nodeIntegration 비활성화

nodeIntegration이 활성화된 경우, Renderer process에서 Node.js API 호출이 가능하게 된다. 이는 특정 조건이 만족 되었을 때, 잠재적으로 RCE 취약점의 발생으로 이어질 수 있는데 그 조건은 다음과 같다.

1. Renderer process에서 XSS 취약점이 발생한

Table 2. Vulnerable and Safe Code Example

Method	Vulnerable	Safe

6.1	const mainWindow = new BrowserWindow({
	webPreferences: {
	nodeIntegration: true,
	
	nodeIntegrationInWorker: true
	}
	})
	
	mainWindow.loadURL('https://example.com')
		const mainWindow = new BrowserWindow({
		webPreferences: {
		preload: path.join(app.getAppPath(), 'preload.js')
		}
		})
		
		mainWindow.loadURL('https://example.com')

6.2	// preload.js
	window.nodeRequire = require;
	
	const { readFileSync } = require('fs')
	
	window.readConfig = function () {
	const data = readFileSync('./config.json')
	return data
	}
		// preload.js
		const { readFileSync } = require('fs')
		
		window.readConfig = function () {
		const data = readFileSync('./config.json')
		return data
		}

6.3	const { shell } = require('electron') shell.openExternal(**URL_WITH_ARBITRARY_PROTOCOL**)	const { shell } = require('electron') shell.openExternal('**https://example.com/index.html**')

6.4	const mainWindow = new BrowserWindow({
	webPreferences:{
	contextIsolation: false
	}
	})
		const mainWindow = new BrowserWindow({
		webPreferences: {
		contextIsolation: true
		}
		})

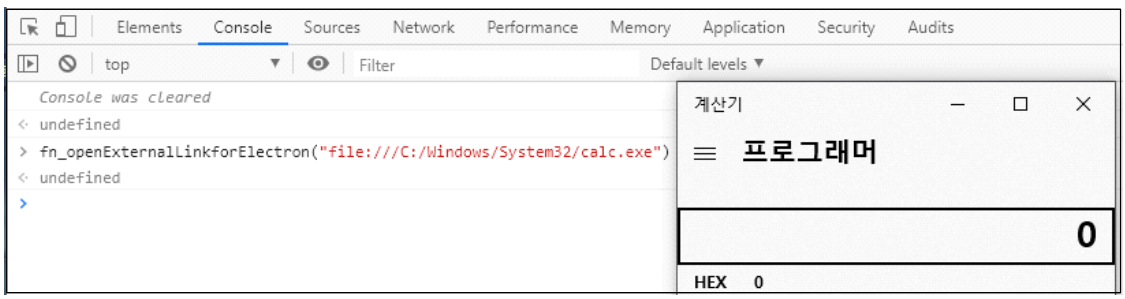

Fig. 22. RCE PoC (Program B, Vulnerability Related to *shell.openExternal*)

경우

2. Renderer process에서 신뢰할 수 없는 사이트의 javascript가 실행 가능한 경우

Renderer process에서 XSS 취약점이 발생한 경우에 nodeIntegration이 활성화되어 있다면, Node.js API를 통해 네이티브 리소스에 접근이 가능하므로, RCE 취약점으로 이어질 수 있다.

또는, Renderer process에서 신뢰할 수 없는 사이트에 접속이 가능하다면, 렌더링 과정에서 해당 사이트의 javascript가 실행됨으로써 Node.js API 호출을 통해 XSS 취약점이 발생한 것과 동일하게 RCE 취약점으로 이어질 수 있다.

따라서, RCE 취약점으로 이어질 가능성을 사전에 차단하기 위해, nodeIntegration을 비활성화하고 preload 스크립트를 이용해 Renderer process에서 사용할 Node.js API에 대해 접근을 허용해 RCE 취약점으로 이어질 가능성을 사전에 차단하는 것이 바람직하다.

6.2 preload 스크립트 점검

5.3.1에서 소개한 B 협업 프로그램 사례와 같이, preload 스크립트를 이용해 Renderer process에서 접근을 허용하는 Node.js API의 범위가 너무 넓으면 잠재적으로 RCE 취약점으로 이어질 수 있다. 즉, nodeIntegration 이 비활성화되어 있음에도 preload 스크립트에서 접근을 허용하는 범위가 너무 넓으면 nodeIntegration 이 활성화된 것과 동등하게 된다.

따라서, preload 스크립트에서는 Renderer process에서 사용할 최소한의 Node.js API들에 한해 접근을 허용함으로써 취약성을 줄일 수 있다.

6.3 openExternal 점검

5.2.2과 5.3.2에서 소개한 바와 같이 shell.openExternal 함수는 인자로 전달된 외부 프로토콜 URL을 데스크톱의 기본 프로그램으로 연다. 해당 함수에서 취약성이 발견되면, 임의의 프로그램 실행이 가능하기 때문에 RCE 취약점과 동등한 영향력을 지닐 수 있다.

따라서, 해당 함수의 인자로 전달되는 URL의 프로토콜에 대한 검증이 필수적이다. 실제 해외 협업 프로그램인 T 협업 프로그램의 경우 Fig 23, 24와 같은

```
const externalPolicyForURL = function(urlStr) {
  const { protocol } = getDetailsForURL(urlStr);
  const isSafe = TRUSTED_SCHEME.includes(protocol);
  const isUnsafe = LOCAL_SCHEMES.includes(protocol) ||
  NO_ACCESS_SCHEMES.includes(protocol);

  if (isSafe) {
    return 'allow';
  }
  if (isUnsafe) {
    return 'deny';
  }
  return 'prompt';
};
```

Fig. 23. Verification Code on *shell.openExternal* Arguments

```
const LOCAL_SCHEMES = ['file'];
const NO_ACCESS_SCHEMES =
['about', 'javascript', 'data'];
const TRUSTED_SCHEME =
['mailto', 'cal', 'webcal', 'http', 'https'];
```

Fig. 24. Lists of Allowed / Disallowed URL Schemes

검증 로직을 통해 특정 URL 프로토콜만을 허용하고 있다.

6.4 contextIsolation 활성화

contextIsolation이 비활성화되어 있으면, preload 스크립트 혹은 Electron framework의 코드에 대한 prototype pollution을 통해 RCE 취약점으로 이어질 수 있다.

따라서, contextIsolation을 활성화함으로써 preload 스크립트와 Electron framework의 javascript context와 Renderer process의 javascript context를 분리함으로써 prototype pollution을 통한 RCE 취약점의 발생 가능성을 사전에 차단해야 한다. 해당 보안 옵션은 Electron 12 버전부터 기본적으로 활성화되고 있다.

VII. 결 론

본 연구는 COVID-19로 인해 협업 프로그램 사용량이 급증함에 따라, 협업 프로그램에 대한 취약점 점검을 통해 보안성을 검증하고자 했다.

다수의 협업 프로그램이 Electron 프레임워크를 기반으로 하고 있다는 점에 주목해 Electron 기반 협

업 프로그램에서 보고된 취약점 사례를 선정, 분석하였다. Electron 기반 애플리케이션에서 특정 설정값(nodeIntegration, contextIsolation)과 관련해 최근 공개된 취약점 사례를 분석하였으며, 이를 바탕으로 Electron 기반 국내 협업 프로그램을 대상으로 취약점 점검을 수행하였다.

그 결과, 점검한 5개 협업 프로그램에서 모두 취약한 설정이 있음을 발견했고, 이 중 두 개 협업 프로그램에 대해서는 RCE로 이어질 수 있는 공격 벡터를 찾아내었다. 또한, 점검 결과를 바탕으로 Electron 기반 애플리케이션에서 발생할 수 있는 취약점을 방지하기 위한 대응 방안을 제시하였다.

본 연구에서는 Electron 보안 점검을 할 수 있는 도구인 electronegativity를 사용하여 취약점 점검을 수행하였으며, 향후 이 도구를 개선하는 연구를 진행하고자 한다.

References

〔1〕 Fortune Business Insights(2019), Team Collaboration Software Market Size, Share & COVID-19 Impact Analysis

〔2〕 Electron Homepage (https://www.electronjs.org)

〔3〕 "Golden Age of Business Collaboration Tools", ChosunBiz, https://biz.chosun.com/site/data/html_dir/2020/04/23/2020042304167.html, Apr, 2020

〔4〕 "Rapid Growth of Collaboration Tools with the Trend of Working-At-Home", Sedaily, https://www.sedaily.com/NewsVIew/1Z065EKQ6P, Mar, 2020

〔5〕 Fortune Business Insights, "Market Research Report", 2020

〔6〕 Masato Noguchi and Yosuke Kurami, "Electron Application Development", In sung Yoon Trans., Freelec, pp. 278-279, 2018

〔7〕 Discord bugbounty Homepage (https://discord.com/security)

〔8〕 "Electronegativity - A study of electron security", Blackhat USA 2017, Jul. 2017

〔9〕 "Electron: Abusing the lack of context isolation", CureCon 2018, Aug. 2018

〔10〕 "Preloading Insecurity In Your Electron", Blackhat Asia 2019, Mar. 2019

〔11〕 "app setAsDefaultRCE Client: Electron, scheme handlers and stealthy security patches", ZeroNights 2019, Nov. 2019

〔12〕 Luca Carettoni, "Electron Security Checklist - A guide for developers and auditors", 2017

〔13〕 "Democratizing Electron.js Security", Covalence Conference 2020, Jan. 2020

〔14〕 Common Vulnerabilites and Exposures, "CVE-2020-15926"(Internet), https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-15926

〔15〕 Common Vulnerabilites and Exposures, "CVE-2018-15685"(Internet), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15685

〔16〕 Common Vulnerabilites and Exposures, "CVE-2020-25019"(Internet), https://cve.mitre.org/cgigksrnin/cvename.cgi?name=CVE-2020-25019

〈저 자 소 개〉

이 효 민 (Hyomin Lee) 학생회원
2021년 3월~현재: 고려대학교 스마트보안학부
〈관심분야〉 모의해킹, 시스템 보안, 역공학

장 연 석 (Yeonseok Jang) 정회원
2020년 8월: 고려대학교 컴퓨터학과 졸업
2021년 3월~현재: 고려대학교 일반대학원 석사과정
〈관심분야〉 악성코드, 시스템 보안, 모의해킹

권 용 희 (Yonghee Kwon) 학생회원
2021년 2월: 고려대학교 정보보호학부 졸업
〈관심분야〉 시스템 보안, 네트워크 보안, 모의해킹

임 은 지 (Eunji Lim) 학생회원
2021년 2월: 성신여자대학교 융합보안공학과 졸업
〈관심분야〉 정보보호, 인공지능, 블록체인, 취약점 분석

김 종 민 (Jongmin Kim) 학생회원
2021년 3월~현재: 고려대학교 스마트보안학부
〈관심분야〉 악성코드, 리버스 엔지니어링, 모바일 해킹

박 진 우 (Jinwoo Park) 학생회원
2019년 3월~현재: 가톨릭대학교 컴퓨터정보공학부
〈관심분야〉 웹해킹, 모의해킹