README.md

vuejs-browser-extensions

What to keep in mind if you want to use Vue.js for a WebExtension

Anatomy of a browser extension

Browser extensions consist of 4 parts which are mostly just regular web apps:

- Popup page - the main application (restricted)
- Options page - a config page (restricted)
- Background page - an invisible tab that runs as long as your browser is open (restricted, no frontend)
- Inject script(s) - scripts that can be optionally or automatically run within the context of any website you visit. (unrestricted)

The first three parts have some restrictions...

- Content Security Policy defaults to `script-src 'self'; object-src 'self'`
- `eval` (and friends) won't work (ooh... ahh...)
- All inline JavaScript will not be executed.

More details: https://developer.chrome.com/extensions/contentSecurityPolicy

...and some superpowers. For example:

- they can inject code and assets into any page you visit.
- they can interact with some system hardware, like USB devices.
- they can add new tabs and functions to the debug console.

MANY more details: https://developer.chrome.com/extensions/api_index

Can we get around these restrictions?

The answer is *Yes, but don't*.

Because extensions have invasive access to modify browser behavior, XSS attacks are a greater threat than they would be in a regular browser tab.

For perspective, because electron has a node.js runtime, XSS attacks can directly impact your operating system. Here's A recent electron CVE that demonstrates how XSS could result in Remote code execution (RCE).

Security best practices

- Use the default CSP
- Minimize third party library use
- Avoid distributing un-invoked code

We also have to pick a "stack". Here's mine:

- Webpack & Yarn/NPM
- Vue.js & vue-loader

- SASS & sass-loader

Pitfalls: vue-cli

You may be tempted to `vue-cli init webpack my-extension` and try to shoehorn the result into a working browser extension.

This will result in:

- A rat's nest of Webpack config you don't need
- A dependency list heavier than a neutron star
- A development server on `http://localhost:8080` we can't do anthing with.

A from-scratch approach

Use the un-minified development version of `vue.js` from https://github.com/vuejs/vue/releases and create a simple hello-world vue.js app will look something like this:

```
// index.html

<!DOCTYPE html>
<html lang="en">
  <body>

    <div id="elm">
      <h1>{{ msg }}</h1>
    </div>
    <script src="vue.js"></script>
    <script src="index.js"></script>

  </body>
</html>

// index.js

new Vue({
        el: "#elm",
        data() {
          return {
            msg: "Hello World!"
          }
        }
});

// manifest.json

{
        "name": "Example",
        "short_name": "Example",
        "version": "2018.5.9",
        "manifest_version": 2,
        "minimum_chrome_version": "48",
        "description": "Example",
        "browser_action": {
                "default_popup": "index.html",
                "default_title": "Example"
        }
}
```

If you browse to `file:///path/to/index.html` everything looks peachy.

If you import the app as a browser extension you get a console error:

```
Error compiling template:

<div id="elm">
        <h1>{{ msg }}</h1>
</div>

- invalid expression: Refused to evaluate a string as JavaScript because 'unsafe-eval' is not an allowed so
 in

    _s(msg)

  Raw expression: {{ msg }}

(found in <Root>)
```

The offending lines from `vue.js` reveal themselves to be:

```
function createFunction (code, errors) {
  try {
    return new Function(code)
  } catch (err) {
    errors.push({ err: err, code: code });
    return noop
  }
}
```

Setting a breakpoint shows the `code` string arg:

```
"with(this){return _c('div',{attrs:{"id":"elm"}},[_c('h1',[_v(_s(msg))])])}"
```

Solving the eval problem

Here, I'll borrow some points from a great article on the subject.

Write your own render functions

`vue-template-compiler` is actually a component seperable from the vue runtime. It is invoked at runtime when you either:

1. you use a template string in your vue source code; or
2. you mount to a template using `el` à la example1

You can avoid invoking the template compiler by writing your own render functions, described in the Vue.js docs but this won't scale.

From the docs:

> Vue recommends using templates to build your HTML in the vast majority of cases. There are situations however, where you really need the full programmatic power of JavaScript. That's where you can use the render function, a closer-to-the-compiler alternative to templates.

Render functions are impractical for writing your entire application.

```
// An example render function
// No need to eval because this function explicitly creates my new element.

render: function (createElement) {
  return createElement(
    s'h1',                              // the element to creat,
    {},                                 // A data object corresponding to the attributess
    [ this.blogTitle ]  // the list of children to populate this new element
```

```
  )
}
```

You'll notice this is exactly the same as the `code` argument from above. The template compiler turned our template into a string representation of a render function and tried to `eval()` it.

Use Single File Components (SFCs)

From the same article:

> When you use vue-loader to process your .vue file, one of the things it does is use vue-template-compiler to turn your component's template into a render function.

You can precompile your entire application. For this we need.... **webpack!** (and Babel and vue-loader)

Luckily, the webpack configuration much more terse than `vue-cli init` gives you.

```
.
├── build
│   └── index.build.js
├── index.html
├── index.js
├── manifest.json
├── package.json
├── Popup.vue
├── vue.js
├── webpack.config.js
├── yarn-error.log
└── yarn.lock
```

The important parts are:

```
// index.html:

<div id="app">
    <popup></popup>
</div>
<script src="build/index.build.js"></script>

// index.js:

import Popup from './Popup.vue'
import Vue from 'vue';

new Vue({
    el: "#app",
    components: {
        Popup
    }
})

// Popup.vue is boundary of wierdness, where all your current Vue code will work normally.
```

Loading `/path/to/index.html` as a static file produces a broken page. In console, you can see:

```
[Vue warn]: You are using the runtime-only build of Vue where the template compiler is not available.
Either pre-compile the templates into render functions, or use the compiler-included build.
```

It turns out that `import Vue from 'vue'` grabs a runtime-only build by default. While there are no templates to explicitly cause `new Function(stringn)` to be executed, vue wants to run `<div id="app">...</div>` from `index.html` through the compiler and cannot find it.

Even if the compiler were present, we would be right back to the `eval` problem.

Solution: Use a render function for *only* the top-level template so `vue-template-loader` is never needed.

`index.js` becomes:

```
import Popup from './Popup.vue' /* compiled by webpack */
import Vue from 'vue'           /* runtime only */
new Vue({
    el: "#app",
    render: createElement => createElement(Popup)
})
```

Viola, you're running a Vue.js app as a browser extension!

So how do I set up my development environment?

We can't use a development server, and we need to load fully compiled static assets from disk.

`webpack --watch` is perfect for us because during development, the browser fetches all assets from disk every time the application is opened. For example, when you click the Popup icon, all necessary files are fetched directly from disk.

Webpack Watch docs

`webpack --watch` is *very* fast because it only re-compiles the files that change. The rest of the compilation is kept in process memory.

Your development flow is now as easy as:

1. Make a code change.
2. Reload the popup (or options) window. I do this with `window.location.reload()` in the debug console.

How come I've never had to do that render function jazz?

You might be using `vue.runtime.js` if you load vue from script tags.

It's more likely that you started a project with a tool like `vue-cli init`, and the `webpack.conf` it creates contains the following section, aliasing `vue` as the full build:

```
resolve: {
    alias: {
        'vue$': 'vue/dist/vue.esm.js'
    },
    extensions: ['*', '.js', '.vue', '.json']
},
```

This is why I'm not a fan of boilerplate project generators.

Other problems

1. Expensive(ish) source maps. As with before, we cannot use `eval` in our source maps. According to https://webpack.js.org/configuration/devtool/ `cheap-source-map` is the best we can do.
2. Long initial build times. Avoid compiling static assets with webpack if you can avoid it (images and fonts are the worst), and try `DLLPlugin`.
3. Hot Reload is tricky but not impossible. Webpack Chrome Extension Loader is brilliant middleware that translates WebSocket reload notifications to `chrome.runtime` events that your extension can listen for.

A living example project

`example2_sfc` is great for understanding the basic setup.

My browser extension, Tusk will provide guidance for the gritty details, such as handling static resources and using `DLLPlugin`. It's also a decent example of how to organize a large browser extension project with Vue.js (maybe?)