This document is public

Chrome Translate running under Content-Security-Policy

Master issue: ***crbug/164547***

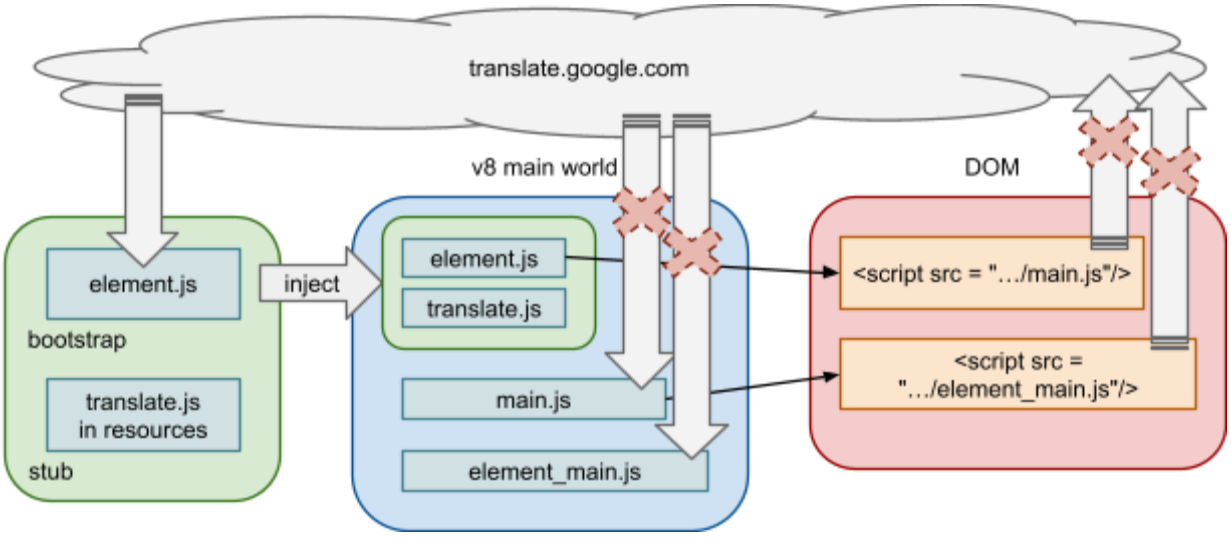
Created: 2013-03-11
Last modified: 2013-04-19
toyoshim@chromium.org

Background and Objective

CSP: Content-Security-Policy is a new scheme to protect a web from XSS attacks. Simply said, the spec allows a server to request a browser to limit injecting JavaScript from external security origin.
This is fine for typical cases. But Chrome Extensions and some embedded features are implemented by JavaScripts and it needs to inject external JavaScripts which may break server providing CSP.
Chrome Extensions uses an isolated world and runs Extensions' JavaScripts inside the world. They expand the isolated world to own independent CSP and it make it possible Extensions works under strict CSP. In this proposal, I try to apply a similar scheme to an embedded feature, Chrome Translate though it requires some technical challenges.

Current Design

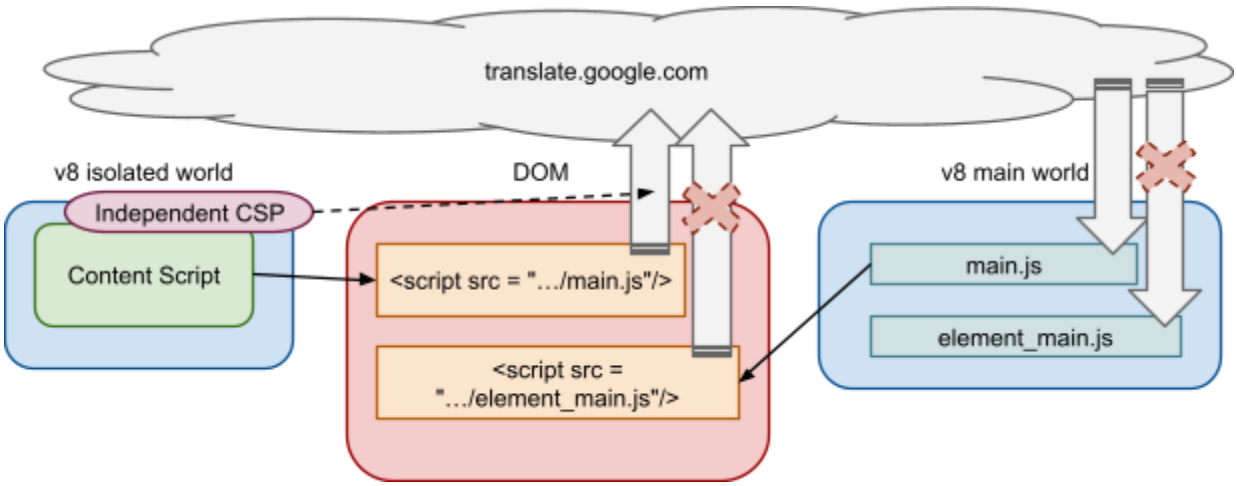
Chrome embedded Translate



1. *Chrome injects boot scripts into v8 main world, then run them*
    a. *Fetch element.js script from https://translate.google.com/translate_a/*
    b. *Merge it with an internal resource translate.js*

    c. *Inject them into v8 main world*
2. *Injected element.js appends a script tag with src="<translate server>/main.js"*
    a. *Browser fetch the specified main.js script from the translate server*
    b. *Run main.js in v8 main world context*
3. *main.js appends a script tag with src="<translate server>/element_main.js"*
    a. *<translate server> contains random prefix which balances server loads*
    b. *Browser fetch the specified element_main.js from the server*
    c. *Run element_main.js in v8 main world context*
4. *element_main.js performs actual web contents translation*

Chrome Translate Extension



Problems under CSP restriction in Chrome embedded Translate

If a server requests to apply strict CSP rules, Chrome can inject boot scripts directly into page contents, but script tags appended by element.js and main.js do not work.

- *Chrome fails to load main.js from a translate server due to CSP restriction*
- *Of course, element_main.js can not be loaded due to lack of main.js as its loader*

Problems under CSP restriction in Chrome Translate Extension

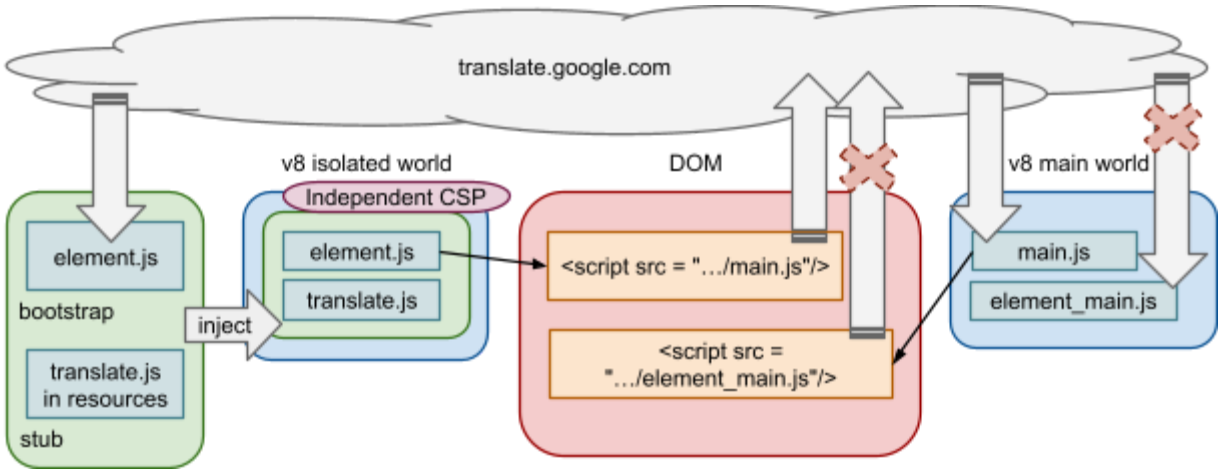
If a server requests to apply strict CSP rules, the first appended script tag can load main.js under independent CSP defined for the Extension. But main.js can not work to load element_main.js.

- *Chrome fails to load element_main.js from a translate server due to main world CSP*

Note: This is a story where the Extension is deployed with proper manifest v2 which defines proper content_security_policy field. Currently, the Extension is deployed without it, so loading main.js also fails.

New design proposals

Straightforward way (still doesn't work like the Extension)



1. *Chrome injects boot scripts into v8 isolated world, then run them*
    a. *Fetch element.js script from https://translate.google.com/translate_a/*
    b. *Merge it with an internal resource translate.js*
    c. *Create new v8 isolated world for Chrome embedded Translate*
    d. *Inject them into v8 isolated world*
2. *Injected element.js appends a script tag with src="<translate server>/main.js"*
    a. *Browser fetch the specified main.js script from the translate server*
    b. *Run main.js in v8 main world context*
3. *main.js appends a script tag with src="<translate server>/element_main.js"*
    a. *<translate server> contains random prefix which balances server loads*
    b. *Browser fetch the specified element_main.js from the server*
    c. *Run element_main.js in v8 main world context*
4. *element_main.js performs actual web contents translation*

This scheme is not enough to work under CSP restriction. The same problem of the Extension with own CSP exists. We should find out right way to avoid this problem.

Discussion

Plan A)

1. *Integrate element.js and main.js as a part of boot scripts*
2. *Chrome injects them as a boot scripts into an isolated world directly, then load element_main.js into main world via script tag under independent CSP*

    This will work if element_main.js doesn't load other scripts via script tags any more.

    *Note: After discussion with Translate team, I decide **not to adopt this plan**. Fetching main.js is important to shutdown Translate service immediately on a sudden surge in demand. Embedding element.js into Chrome looks safe and make the design simpler.*

Plan B)

1. *Integrate element.js into translate.js in Chrome resources*
2. *Inject integrated translate.js into isolated world, and run it there*

a. *translate.js should be modified to fetch main.js by using XHR*
b. *eval main.js in isolated world. It will append script tag for element_main.js, and it works under independent CSP*

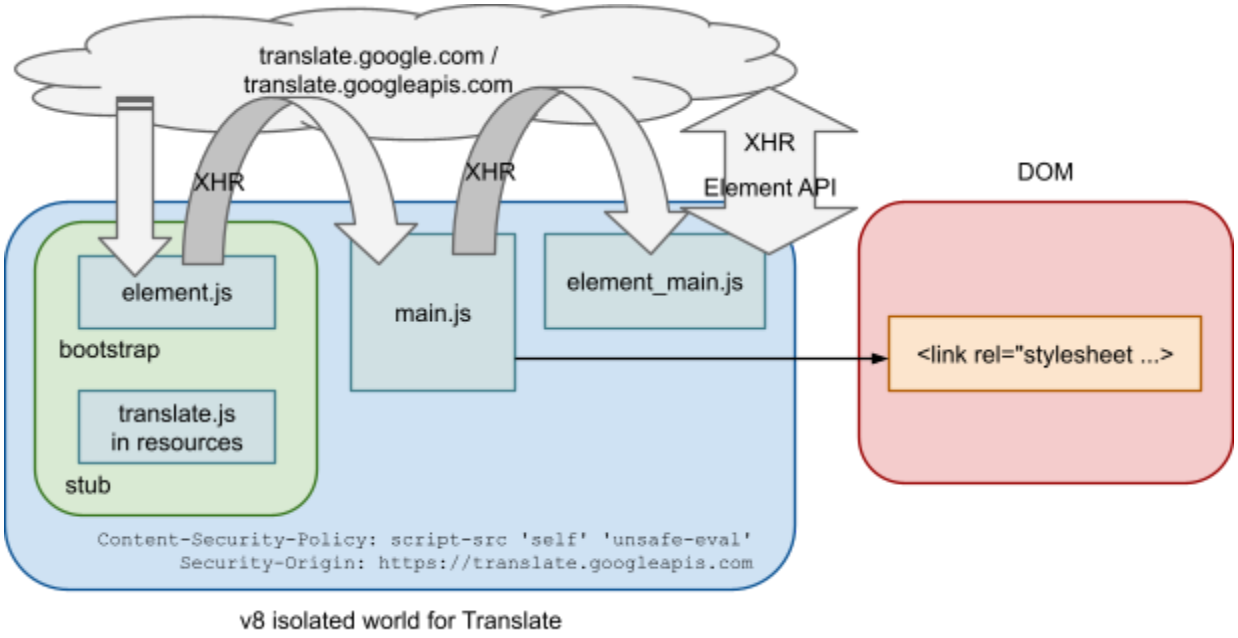
Firstly, I'm going to adopt Plan B for Chrome embedded Translate. But it also doesn't work because of two reasons.

The first reason is JavaScript isolation itself. main.js defines various variables for error messages which depends on user locale. But it is in the isolated world. On the other hand, element_main.js runs in the main world. So it can not use defined variables. Of course, it is not impossible to communicate between the two world, but it needs a kind of tricky code.

The second reason is critical. element_main.js still needs XHR connections to translate page contents. But, XHR connections can be disabled by CSP, too. If the page owner specify connect-src 'self', it doesn't work at all.

Plan C) - Run everything in an isolated world -

1. *Hook appendChild method of the first head element in the isolated world*
   a. *to insert passed CSS links into DOM tree to inject it in the main world*
   b. *to load and run passed JavaScripts in the isolated world by using XHR*



v8 isolated world for Translate

Chrome Implementation Details

Isolated World ID Management

Currently, isolated world ID management depends on Chrome Extensions. To use it outside Chrome Extensions, I should move management logic outside extensions directory.

place:

src/chrome/renderer/extensions/user_script_slave.cc

src/chrome/renderer/reserved_isolated_world_ids.h [added]

method and enum:

```
int UserScriptSlave::GetIsolatedWorldIdForExtension(
                const Extension* extension,
                WebFrame* frame);
enum IsolatedWorldIDs {
        ISOLATED_WORLD_ID_GLOCAL = 0,
        ISOLATED_WORLD_ID_TRANSLATE,
        ISOLATED_WORLD_ID_EXTENSIONS
};
```

logic:

If an isolated world ID is already assigned for specified extension's Extension ID, it returns the assigned ID. Otherwise, assign new incremented ID which starts from the fixed value. Assigned ID and Extension ID are binded together, then stored to a map. The fixed value was 1, but now IsolatedWorldIDs defines the fixed number as ISOLATED_WORLD_ID_EXTENSIONS so that ISOLATED_WORLD_ID_TRANSLATE is reserved for Chrome Translate.

CL:

https://codereview.chromium.org/12583016

Adopt an Isolated World for Chrome Translate

Extension Groups

place:

src/chrome/renderer/extensions/extension_groups.h

logic:

Use a specific ExtensionGroups ID and no extension APIs should be available in the group.

CL:

https://codereview.chromium.org/14022005

Content Security Policy

place:

src/chrome/renderer/translate/translate_helper.cc

logic:

Apply an independent content security policy `script-src 'self' 'unsafe-eval'` *for the isolated world. 'unsafe-eval' is needed to emulate script tag injection which is needed to load and run scripts fetched via https from a Google Translate server.*

CL:

https://codereview.chromium.org/14022005

Security Origin

place:

src/chrome/renderer/translate/translate_helper.cc

logic:

Apply an independent security origin `https://translate.googleapis.com`*. Originally, external scripts are provided from the server and run with the origin. It is*

needed to use XHR from the script for emulating script tag injection and performing translation.

CL:

https://codereview.chromium.org/14022005

Adopt Isolated World

Just call WebFrame::executeScriptInIsolatedWorld() instead of WebFrame::executeScript() with proper arguments, chrome::ISOLATED_WORLD_ID_TRANSLATE and extensions::EXTENSION_GROUP_INTERNAL_TRANSLATE_SCRIPTS. Independent content security policy and security origin are assigned on starting page translation by calling WebFrame::setIsolatedWorldContentSecurityPolicy() and WebFrame::setIsolatedWorldSecurityOrigin().

Alternate Ideas

Component Extensions

In a review, mpcomplete@ let me know interesting approach which uses Component Extensions. The Component Extensions is a kind of a Chrome Extensions which has a special privilege to access private Extension APIs. Using this scheme, we can use all benefits of Chrome Extensions, and collaborate with native implementation through private APIs.
This approach sounds good, but needs more reconstruction of Translate feature. So, I give up to adopt this approach for now.