CSP and WASM

Current state and proposal

What is CSP (Content Security Policy)?

- Content Security Policy
  - Allows websites (servers) to restrict client-side loading of resources in order to mitigate cross-site scripting attacks (XSS)
  - Directives can restrict from where pages are allowed to load **scripts** and other resources
  - Can specify expected hashes to ensure resource integrity from 3rd party CDNs
- Relevance to WASM
  - CSP can be used to restrict the *source of code* running on a site (`script-src`)
  - Can disable JavaScript language features: **`eval`**
  - By default, disabling JavaScript's **`eval`** also disables WebAssembly compile APIs
  - Browsers should align to define the restrictions appropriate for WASM
    - Policy choices can and should differ for WASM (reasons)

A CSP Example

- Content Security Policy directives can be part of an HTTP response header
  - `Content-Security-Policy: script-src https://example.com/`
    - This restricts script tags within the document
    - `<script src="https://not-example.com/js/library.js"></script>` ✗ disallowed
  - CSP can also disable inline scripts in the document and JavaScript **`eval`**.
    - These features can be re-enabled with additional directives.
      - `Content-Security-Policy: script-src 'unsafe-inline'`
      - `Content-Security-Policy: script-src 'unsafe-eval'`

- There are lots of other directives that are out of scope

CSP `script-src` directive

- `Content-Security-Policy: script-src <source>;`

```
source ::= <host-source>
         |  <scheme-source>
         |  'self'
         |  'unsafe-inline’
         |  'unsafe-eval'
         |  'none'
         |  'nonce-<base64-value>'
         |  '<hash-algorithm>-<base64-value>'
         |  'strict-dynamic'
         |  'report-sample'
```

Weighing JavaScript and WebAssembly threats

- JavaScript **`eval`** has ambient authority
  - Code in (a sloppy) **`eval`** has access to the evaluator's ***scope***
  - A JS scope inevitability includes sensitive operations, representing a risk
- WebAssembly has capability-based security
  - Code in a module can only access that which is supplied at ***instantiation*** time of module
  - No risk that code can access *more* capabilities
  - But, man-in-the-middle type attacks can still substitute *different* code

The distinction is worth mentioning, yet CSP restrictions are still useful

What does CSP for Wasm accomplish?

- Prevent man-in-the-middle attacks (resource integrity)
- Prevent JavaScript vulnerabilities from becoming WASM vulnerabilities
    - The route is a bit more indirect.
    - E.g. JavaScript-generated URL ↠ fetch WASM module
    - JavaScript imports object ↠ WASM module

- To achieve its intended purpose, CSP is needed for *both* JavaScript and WASM on a page.

CSP “safe” WebAssembly operations

- Many uncontroversial operations need no CSP consideration.

Operation	✗ unsafe-eval	✓ unsafe-eval
`new WebAssembly.CompileError()`	✓	✓
`new WebAssembly.LinkError()`	✓	✓
`new WebAssembly.Table()`	✓	✓
`new WebAssembly.Global()`	✓	✓
`new WebAssembly.Memory()`	✓	✓

CSP “safe” WebAssembly operations

- There has been some discussion about instantiation.
  I argue it is “safe” when it *does not create* new code and propose to allow it.

Operation	✗ unsafe-eval	✓ unsafe-eval
`new WebAssembly.Instance(module)`	✓	✓
`WebAssembly.instantiate(module)`	✓	✓

CSP proposal addition (1) - `wasm-eval`

- Introduce a new `script-src` directive: `wasm-eval`
  - allows **`new WebAssembly.Module()`** and friends *but* no JavaScript **`eval`**
- Already implemented in Chromium for extension URLs (ChromeOS)
- Big hammer: effectively enables all WASM

Operation	✗ unsafe-eval	✓ wasm-eval	✓ unsafe-eval
JavaScript `eval()`	✗	✗	✓
`new WebAssembly.Module()`	✗	✓	✓
`WebAssembly.compile()`	✗	✓	✓
`WebAssembly.validate()`	✗	✓	✓

CSP proposal addition (2) - streaming operations

- WebAssembly streaming APIs accept `Response` objects
- Proposal: implementation `Response` has internal URL
- URL checked against CSP policy in streaming operations

Operation	✗ unsafe-eval	✓ wasm-eval	✓ unsafe-eval
`WebAssembly.compileStreaming()`	CSP	CSP	CSP
`WebAssembly.instantiateStreaming()`	CSP	CSP	CSP

- CSP checks: script origins and sub-resource integrity (e.g. hash)
- Enforce required mimetype: `application/wasm`

Open Issues

- #3 - Do we need to check the mimetype for streaming?
- #4 - Does `Response` really need a URL?
- #8 - Allow sub-resource integrity matching for compile APIs
- #9 - Do `WebAssembly.Module` objects need origin info?

#8 Allow compile-with-bytes if hash matches

- Allow compile APIs to work if bytes match a SRI hash?
  - `Content-Security-Policy: script-src 'sha256-123...456'’`

Operation	✗ unsafe-eval	✓ wasm-eval	✓ unsafe-eval
JavaScript `eval()`	✗	✗	✓
`new WebAssembly.Module()`	SRI	SRI	✓
`WebAssembly.compile()`	SRI	SRI	✓
`WebAssembly.validate()`	SRI	SRI	✓

#3 Do Mime Types need to match?

- Current proposal text suggestions that streaming APIs check mimetype
- Do we want that?
- That enforcement would be (yet another) addition to `Response`

#9 Should WebAssembly.Module objects have origin info?

- Objects in the Web platform do not, as a rule, contain origins
  - `postMessage()` on an object does not currently imply access permission checks
  - Cannot `postMessage()` a module to another worker that is not same origin
  - ⇒ `postMessage()` does not need permission checks here
    (and it would be weird if it did)
  - Workers that can post-message a WASM module share the same script sources
    - *Except* proposed CSP feature for more restrictive policies in workers
- Backroom discussion
  - CSP policy applies to an entire document
  - Not possible to have different CSP policies for shared workers, unless…
  - Browser honors *more restrictive* CSP policy for the script used to start a worker
    - This is currently being discussed by CSP groups; Google would prefer not
      ⇒ CSP policy for entire document is sufficient

WebAssembly CSP moving pieces

- CSP proposal (GitHub repo)
  - Overview
  - Issues being discussed
- Specifications
  - JavaScript API specification: hooks for web platform to disallow certain APIs
  - Web Embedding specification: CSP semantics
  - CSP specification: new `wasm-eval` directive
  - WebIDL specification (whatwg/fetch?): `Response` URL