Content Security Policy Level 3

W3C Working Draft, 15 October 2018

This version:
https://www.w3.org/TR/2018/WD-CSP3-20181015/

Latest published version:
https://www.w3.org/TR/CSP3/

Editor's Draft:
https://w3c.github.io/webappsec-csp/

Previous Versions:
https://www.w3.org/TR/2016/WD-CSP3-20160913/

Version History:
https://github.com/w3c/webappsec-csp/commits/master/index.src.html

Feedback:
public-webappsec@w3.org with subject line "`[CSP3] … message topic …`" (archives)

Editor:
Mike West (Google Inc.)

Participate:
File an issue (open issues)

Tests:
web-platform-tests content-security-policy/ (ongoing work)

Abstract

This document defines a mechanism by which web developers can control the resources which a particular page can fetch or execute, as well as a number of security-relevant policy decisions.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical

report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the Web Application Security Working Group as a Working Draft. This document is intended to become a W3C Recommendation.

The (archived) public mailing list public-webappsec@w3.org (see instructions) is preferred for discussion of this specification. When sending e-mail, please put the text “CSP3” in the subject, preferably like this: “[CSP3] *...summary of comment...*”

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by the Web Application Security Working Group.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 February 2018 W3C Process Document.

The following features are at-risk, and may be dropped during the CR period:

- The §6.6.3.1 Is element nonceable? algorithm.

“At-risk” is a W3C Process term-of-art, and does not necessarily imply that the feature is in danger of being dropped or delayed. It means that the WG believes the feature may have difficulty being interoperably implemented in a timely manner, and marking it as such allows the WG to drop the feature if necessary when transitioning to the Proposed Rec stage, without having to publish a new Candidate Rec without the feature first.

Table of Contents

§ 1. Introduction

This section is not normative.

This document defines ***Content Security Policy*** (CSP), a tool which developers can use to lock down their applications in various ways, mitigating the risk of content injection vulnerabilities such as cross-site scripting, and reducing the privilege with which their applications execute.

CSP is not intended as a first line of defense against content injection vulnerabilities. Instead, CSP is

best used as defense-in-depth. It reduces the harm that a malicious injection can cause, but it is not a replacement for careful input validation and output encoding.

This document is an iteration on Content Security Policy Level 2, with the goal of more clearly explaining the interactions between CSP, HTML, and Fetch on the one hand, and providing clear hooks for modular extensibility on the other. Ideally, this will form a stable core upon which we can build new functionality.

§ 1.1. Examples

§ 1.1.1. Control Execution

> EXAMPLE 1
> MegaCorp Inc's developers want to protect themselves against cross-site scripting attacks. They can mitigate the risk of script injection by ensuring that their trusted CDN is the only origin from which script can load and execute. Moreover, they wish to ensure that no plugins can execute in their pages' contexts. The following policy has that effect:
>
> ```
> Content-Security-Policy: script-src https://cdn.example.com/scripts/; object-src
> ```

§ 1.2. Goals

Content Security Policy aims to do to a few related things:

1. Mitigate the risk of content-injection attacks by giving developers fairly granular control over
    - The resources which can be requested (and subsequently embedded or executed) on behalf of a specific `Document` or `Worker`
    - The execution of inline script
    - Dynamic code execution (via `eval()` and similar constructs)
    - The application of inline style
2. Mitigate the risk of attacks which require a resource to be embedded in a malicious context (the "Pixel Perfect" attack described in [TIMING], for example) by giving developers granular control over the origins which can embed a given resource.
3. Provide a policy framework which allows developers to reduce the privilege of their applications.
4. Provide a reporting mechanism which allows developers to detect flaws being exploited in the

wild.

§ 1.3. Changes from Level 2

This document describes an evolution of the Content Security Policy Level 2 specification [CSP2]. The following is a high-level overview of the changes:

1. The specification has been rewritten from the ground up in terms of the [FETCH] specification, which should make it simpler to integrate CSP's requirements and restrictions with other specifications (and with Service Workers in particular).
2. The `child-src` model has been substantially altered:
    1. The `frame-src` directive, which was deprecated in CSP Level 2, has been undeprecated, but continues to defer to `child-src` if not present (which defers to `default-src` in turn).
    2. A `worker-src` directive has been added, deferring to `child-src` if not present (which likewise defers to `script-src` and eventually `default-src`).
    3. Dedicated workers now always inherit their creator's policy.
3. The URL matching algorithm now treats insecure schemes and ports as matching their secure variants. That is, the source expression `http://example.com:80` will match both `http://example.com:80` and `https://example.com:443`.

    Likewise, `'self'` now matches `https:` and `wss:` variants of the page's origin, even on pages whose scheme is `http`.
4. Violation reports generated from inline script or style will now report "`inline`" as the blocked resource. Likewise, blocked `eval()` execution will report "`eval`" as the blocked resource.
5. The `manifest-src` directive has been added.
6. The `report-uri` directive is deprecated in favor of the new `report-to` directive, which relies on [REPORTING] as infrastructure.
7. The `'strict-dynamic'` source expression will now allow script which executes on a page to load more script via non-"parser-inserted" `<script>` elements. Details are in §8.2 Usage of "'strict-dynamic'".
8. The `'unsafe-hashes'` source expression will now allow event handlers, style attributes and `javascript:` navigation targets to match hashes. Details in §8.3 Usage of "'unsafe-hashes'".
9. The source expression matching has been changed to require explicit presence of any non-network scheme, rather than local scheme, unless that non-network scheme is the same as the scheme of protected resource, as described in §6.6.2.6 Does url match expression in origin with

redirect count?.

10. Hash-based source expressions may now match external scripts if the `<script>` element that triggers the request specifies a set of integrity metadata which is listed in the current policy. Details in §8.4 Allowing external JavaScript via hashes.
11. The `navigate-to` directive gives a resource control over the endpoints to which it can initiate navigation.
12. Reports generated for inline violations will contain a sample attribute if the relevant directive contains the `'report-sample'` expression.

§ 2. Framework

§ 2.1. Infrastructure

This document uses ABNF grammar to specify syntax, as defined in [RFC5234]. It also relies on the `#rule` ABNF extension defined in Section 7 of [RFC7230], with the modification that OWS is replaced with optional-ascii-whitespace. That is, the `#rule` used in this document is defined as:

```
1#element => element *( optional-ascii-whitespace "," optional-ascii-whitespace ele
```

and for n >= 1 and m > 1:

```
<n>#<m>element => element <n-1>*<m-1>( optional-ascii-whitespace "," optional-ascii
```

This document depends on the Infra Standard for a number of foundational concepts used in its algorithms and prose [INFRA].

The following definitions are used to improve readability of other definitions in this document.

```
optional-ascii-whitespace = *( %x09 / %x0A / %x0C / %x0D / %x20 )
required-ascii-whitespace = 1*( %x09 / %x0A / %x0C / %x0D / %x20 )
; These productions match the definition of ASCII whitespace from the INFRA standar
```

§ 2.2. Policies

A ***policy*** defines allowed and restricted behaviors, and may be applied to a `Document`, `WorkerGlobalScope`, or `WorkletGlobalScope` as described in §4.2.2 Initialize a global object's CSP list and in §4.2.1 Initialize a Document's CSP list.

Each policy has an associated ***directive set***, which is an ordered set of directives that define the

policy's implications when applied.

Each policy has an associated ***disposition***, which is either "`enforce`" or "`report`".

Each policy has an associated ***source***, which is either "`header`" or "`meta`".

Multiple policies can be applied to a single resource, and are collected into a list of policies known as a ***CSP list***.

A CSP list ***contains a header-delivered Content Security Policy*** if it contains a policy whose source is "`header`".

A ***serialized CSP*** is an ASCII string consisting of a semicolon-delimited series of serialized directives, adhering to the following ABNF grammar [RFC5234]:

```
serialized-policy =
    serialized-directive *( optional-ascii-whitespace ";" [ optional-ascii-whitespa
```

A ***serialized CSP list*** is an ASCII string consisting of a comma-delimited series of serialized CSPs, adhering to the following ABNF grammar [RFC5234]:

```
serialized-policy-list = 1#serialized-policy
                       ; The '#' rule is the one defined in section 7 of RFC 7230
                       ; but it incorporates the modifications specified
                       ; in section 2.1 of this document.
```

§ 2.2.1. Parse a serialized CSP

To ***parse a serialized CSP***, given a serialized CSP (*serialized*), a source (*source*), and a disposition (*disposition*), execute the following steps.

This algorithm returns a Content Security Policy object. If *serialized* could not be parsed, the object's directive set will be empty.

1. Let *policy* be a new policy with an empty directive set, a source of *source*, and a disposition of *disposition*.
2. For each *token* returned by strictly splitting *serialized* on the U+003B SEMICOLON character (`;`):

    1. Strip leading and trailing ASCII whitespace from *token*.
    2. If *token* is an empty string, continue.

3. Let *directive name* be the result of collecting a sequence of code points from *token* which are not ASCII whitespace.
4. Set *directive name* to be the result of running ASCII lowercase on *directive name*.

> Note: Directive names are case-insensitive, that is: `script-SRC 'none'` and `ScRiPt-sRc 'none'` are equivalent.

5. If *policy*'s directive set contains a directive whose name is *directive name*, continue.

> Note: In this case, the user agent SHOULD notify developers that a duplicate directive was ignored. A console warning might be appropriate, for example.

6. Let *directive value* be the result of splitting *token* on ASCII whitespace.
7. Let *directive* be a new directive whose name is *directive name*, and value is *directive value*.
8. Append *directive* to *policy*'s directive set.

3. Return *policy*.

§ 2.2.2. Parse a serialized CSP list

To ***parse a serialized CSP list***, given a serialized CSP list (*list*), a source (*source*), and a disposition (*disposition*), execute the following steps.

This algorithm returns a list of Content Security Policy objects. If *list* cannot be parsed, the returned list will be empty.

1. Let *policies* be an empty list.
2. For each *token* returned by splitting *list* on commas:
    1. Let *policy* be the result of parsing *token*, with a source of *source*, and disposition of *disposition*.
    2. If *policy*'s directive set is empty, continue.
    3. Append *policy* to *policies*.
3. Return *policies*.

§ 2.3. Directives

Each policy contains an ordered set of ***directives*** (its directive set), each of which controls a specific behavior. The directives defined in this document are described in detail in §6 Content Security Policy Directives.

Each directive is a ***name*** / ***value*** pair. The name is a non-empty string, and the value is a set of non-empty strings. The value MAY be empty.

A ***serialized directive*** is an ASCII string, consisting of one or more whitespace-delimited tokens, and adhering to the following ABNF [RFC5234]:

```
serialized-directive = directive-name [ required-ascii-whitespace directive-value ]
directive-name       = 1*( ALPHA / DIGIT / "-" )
directive-value      = *( required-ascii-whitespace / ( %x21-%x2B / %x2D-%x3A / %x3
                       ; Directive values may contain whitespace and VCHAR characte
                       ; excluding ";" and ",". The second half of the definition
                       ; above represents all VCHAR characters (%x21-%x7E)
                       ; without ";" and "," (%x3B and %x2C respectively)

; ALPHA, DIGIT, and VCHAR are defined in Appendix B.1 of RFC 5234.
```

Directives have a number of associated algorithms:

1. A ***pre-request check***, which takes a request and a policy as an argument, and is executed during §4.1.3 Should request be blocked by Content Security Policy?. This algorithm returns "`Allowed`" unless otherwise specified.
2. A ***post-request check***, which takes a request, a response, and a policy as arguments, and is executed during §4.1.4 Should response to request be blocked by Content Security Policy?. This algorithm returns "`Allowed`" unless otherwise specified.
3. A ***response check***, which takes a request, a response, and a policy as arguments, and is executed during §4.1.4 Should response to request be blocked by Content Security Policy?. This algorithm returns "`Allowed`" unless otherwise specified.
4. An ***inline check***, which takes an `Element` a type string, a policy, and a source string as arguments, and is executed during §4.2.4 Should element's inline type behavior be blocked by Content Security Policy? and during §4.2.5 Should navigation request of type from source in target be blocked by Content Security Policy? for `javascript:` requests. This algorithm returns "`Allowed`" unless otherwise specified.
5. An ***initialization***, which takes a `Document` or global object, a response, and a policy as arguments. This algorithm is executed during §4.2.1 Initialize a Document's CSP list, and has no effect unless otherwise specified.
6. A ***pre-navigation check***, which takes a request, a navigation type string ("`form-submission`" or

"other"), two browsing contexts, and a policy as arguments, and is executed during §4.2.5 Should navigation request of type from source in target be blocked by Content Security Policy?. It returns "Allowed" unless otherwise specified.

7. A ***navigation response check***, which takes a request, a navigation type string ("form-submission" or "other"), a response, two browsing contexts, a check type string ("source" or "response"), and a policy as arguments, and is executed during §4.2.6 Should navigation response to navigation request of type from source in target be blocked by Content Security Policy?. It returns "Allowed" unless otherwise specified.

§ 2.3.1. Source Lists

Many directives' values consist of ***source lists***: sets of strings which identify content that can be fetched and potentially embedded or executed. Each string represents one of the following types of ***source expression***:

1. Keywords such as `'none'` and `'self'` (which match nothing and the current URL's origin, respectively)
2. Serialized URLs such as `https://example.com/path/to/file.js` (which matches a specific file) or `https://example.com/` (which matches everything on that origin)
3. Schemes such as `https:` (which matches any resource having the specified scheme)
4. Hosts such as `example.com` (which matches any resource on the host, regardless of scheme) or `*.example.com` (which matches any resource on the host's subdomains (and any of its subdomains' subdomains, and so on))
5. Nonces such as `'nonce-ch4hvvbHDpv7xCSvXCs3BrNggHdTzxUA'` (which can match specific elements on a page)
6. Digests such as `'sha256-abcd...'` (which can match specific elements on a page)

A ***serialized source list*** is an ASCII string, consisting of a whitespace-delimited series of source expressions, adhering to the following ABNF grammar [RFC5234]:

```
serialized-source-list = ( source-expression *( required-ascii-whitespace source-ex
source-expression      = scheme-source / host-source / keyword-source
                       / nonce-source / hash-source

; Schemes: "https:" / "custom-scheme:" / "another.custom-scheme:"
scheme-source = scheme-part ":"

; Hosts: "example.com" / "*.example.com" / "https://*.example.com:12/path/to/file.j
```

```
host-source = [ scheme-part "://" ] host-part [ ":" port-part ] [ path-part ]
scheme-part = scheme
              ; scheme is defined in section 3.1 of RFC 3986.
host-part   = "*" / [ "*." ] 1*host-char *( "." 1*host-char )
host-char   = ALPHA / DIGIT / "-"
port-part   = 1*DIGIT / "*"
path-part   = path-absolute (but not including ";" or ",")
              ; path-absolute is defined in section 3.3 of RFC 3986.

; Keywords:
keyword-source = "'self'" / "'unsafe-inline'" / "'unsafe-eval'"
                 / "'strict-dynamic'" / "'unsafe-hashes'" /
                 / "'report-sample'" / "'unsafe-allow-redirects'"

ISSUE: Bikeshed unsafe-allow-redirects.

; Nonces: 'nonce-[nonce goes here]'
nonce-source  = "'nonce-" base64-value "'"
base64-value  = 1*( ALPHA / DIGIT / "+" / "/" / "-" / "_" )*2( "=" )

; Digests: 'sha256-[digest goes here]'
hash-source    = "'" hash-algorithm "-" base64-value "'"
hash-algorithm = "sha256" / "sha384" / "sha512"
```

The host-char production intentionally contains only ASCII characters; internationalized domain names cannot be entered directly as part of a serialized CSP, but instead MUST be Punycode-encoded [RFC3492]. For example, the domain üüüüüü.de MUST be represented as xn--tdaaaaaa.de.

> Note: Though IP address do match the grammar above, only 127.0.0.1 will actually match a URL when used in a source expression (see §6.6.2.5 Does url match source list in origin with redirect count? for details). The security properties of IP addresses are suspect, and authors ought to prefer hostnames whenever possible.

> Note: The base64-value grammar allows both base64 and base64url encoding. These encodings are treated as equivalant when processing hash-source values. Nonces, however, are strict string matches: we use the base64-value grammar to limit the characters available, and reduce the complexity for the server-side operator (encodings, etc), but the user agent doesn't actually care about any underlying value, nor does it do any decoding of the nonce-source value.

§ 2.4. Violations

A ***violation*** represents an action or resource which goes against the set of policy objects associated with a global object.

Each violation has a ***global object***, which is the global object whose policy has been violated.

Each violation has a ***url*** which is its global object's `URL`.

Each violation has a ***status*** which is a non-negative integer representing the HTTP status code of the resource for which the global object was instantiated.

Each violation has a ***resource***, which is either `null`, `"inline"`, `"eval"`, or a `URL`. It represents the resource which violated the policy.

Each violation has a ***referrer***, which is either `null`, or a `URL`. It represents the referrer of the resource whose policy was violated.

Each violation has a ***policy***, which is the policy that has been violated.

Each violation has a ***disposition***, which is the disposition of the policy that has been violated.

Each violation has an ***effective directive*** which is a non-empty string representing the directive whose enforcement caused the violation.

Each violation has a ***source file***, which is either `null` or a `URL`.

Each violation has a ***line number***, which is a non-negative integer.

Each violation has a ***column number***, which is a non-negative integer.

Each violation has a ***element***, which is either `null` or an element.

Each violation has a ***sample***, which is a string. It is the empty string unless otherwise specified.

> Note: A violation's sample will be populated with the first 40 characters of an inline script, event handler, or style that caused an violation. Violations which stem from an external file will not include a sample in the violation report.

§ 2.4.1. Create a violation object for *global*, *policy*, and *directive*

Given a global object (*global*), a policy (*policy*), and a string (*directive*), the following algorithm creates a new violation object, and populates it with an initial set of data:

1. Let *violation* be a new violation whose global object is *global*, policy is *policy*, effective directive is *directive*, and resource is `null`.
2. If the user agent is currently executing script, and can extract a source file's URL, line number, and column number from the *global*, set *violation*'s source file, line number, and column number accordingly.

   ISSUE 1 Is this kind of thing specified anywhere? I didn't see anything that looked useful in [ECMA262].

   Note: User agents need to ensure that the source file is the URL requested by the page, pre-redirects. If that's not possible, user agents need to strip the URL down to an origin to avoid unintentional leakage.
3. If *global* is a `Window` object, set *violation*'s referrer to *global*'s `document`'s `referrer`.
4. Set *violation*'s status to the HTTP status code for the resource associated with *violation*'s global object.

   ISSUE 2 How, exactly, do we get the status code? We don't actually store it anywhere.
5. Return *violation*.

§ 2.4.2. Create a violation object for *request*, and *policy*.

Given a request (*request*), a policy (*policy*), the following algorithm creates a new violation object, and populates it with an initial set of data:

1. Let *directive* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. Let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on *request*'s client's global object, *policy*, and *directive*.
3. Set *violation*'s resource to *request*'s url.

   Note: We use *request*'s url, and *not* its current url, as the latter might contain information about redirect targets to which the page MUST NOT be given access.
4. Return *violation*.

§ 3. Policy Delivery

A server MAY declare a policy for a particular resource representation via an HTTP response header field whose value is a serialized CSP. This mechanism is defined in detail in §3.1 The Content-Security-Policy HTTP Response Header Field and §3.2 The Content-Security-Policy-Report-Only HTTP Response Header Field, and the integration with Fetch and HTML is described in §4.1 Integration with Fetch and §4.2 Integration with HTML.

A policy may also be declared inline in an HTML document via a `<meta>` element's `http-equiv` attribute, as described in §3.3 The <meta> element.

§ 3.1. The `Content-Security-Policy` HTTP Response Header Field

The ***`Content-Security-Policy`*** HTTP response header field is the preferred mechanism for delivering a policy from a server to a client. The header's value is represented by the following ABNF [RFC5234]:

```
Content-Security-Policy = 1#serialized-policy
                    ; The '#' rule is the one defined in section 7 of RFC 7230
                    ; but it incorporates the modifications specified
                    ; in section 2.1 of this document.
```

> EXAMPLE 2
>
> ```
> Content-Security-Policy: script-src 'self';
>                          report-to csp-reporting-endpoint
> ```

A server MAY send different `Content-Security-Policy` header field values with different representations of the same resource.

A server SHOULD NOT send more than one HTTP response header field named "`Content-Security-Policy`" with a given resource representation.

When the user agent receives a `Content-Security-Policy` header field, it MUST parse and enforce each serialized CSP it contains as described in §4.1 Integration with Fetch, §4.2 Integration with HTML.

§ 3.2. The `Content-Security-Policy-Report-Only` HTTP Response Header Field

The ***`Content-Security-Policy-Report-Only`*** HTTP response header field allows web developers to experiment with policies by monitoring (but not enforcing) their effects. The header's value is

represented by the following ABNF [RFC5234]:

```
Content-Security-Policy-Report-Only = 1#serialized-policy
                    ; The '#' rule is the one defined in section 7 of RFC 7230
                    ; but it incorporates the modifications specified
                    ; in section 2.1 of this document.
```

This header field allows developers to piece together their security policy in an iterative fashion, deploying a report-only policy based on their best estimate of how their site behaves, watching for violation reports, and then moving to an enforced policy once they've gained confidence in that behavior.

EXAMPLE 3

```
Content-Security-Policy-Report-Only: script-src 'self';
                                     report-to csp-reporting-endpoint
```

A server MAY send different `Content-Security-Policy-Report-Only` header field values with different representations of the same resource.

A server SHOULD NOT send more than one HTTP response header field named "`Content-Security-Policy-Report-Only`" with a given resource representation.

When the user agent receives a `Content-Security-Policy-Report-Only` header field, it MUST parse and monitor each serialized CSP it contains as described in §4.1 Integration with Fetch and §4.2 Integration with HTML.

Note: The `Content-Security-Policy-Report-Only` header is **not** supported inside a `<meta>` element.

§ 3.3. The `<meta>` element

A `Document` may deliver a policy via one or more HTML `<meta>` elements whose `http-equiv` attributes are an ASCII case-insensitive match for the string "`Content-Security-Policy`". For example:

EXAMPLE 4

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

Implementation details can be found in HTML's Content Security Policy state `http-equiv` processing instructions [HTML].

Note: The `Content-Security-Policy-Report-Only` header is *not* supported inside a `<meta>` element. Neither are the `report-uri`, `frame-ancestors`, and `sandbox` directives.

Authors are *strongly encouraged* to place `<meta>` elements as early in the document as possible, because policies in `<meta>` elements are not applied to content which precedes them. In particular, note that resources fetched or prefetched using the `Link` HTTP response header field, and resources fetched or prefetched using `<link>` and `<script>` elements which precede a `<meta>`-delivered policy will not be blocked.

Note: A policy specified via a `<meta>` element will be enforced along with any other policies active for the protected resource, regardless of where they're specified. The general impact of enforcing multiple policies is described in §8.1 The effect of multiple policies.

Note: Modifications to the `content` attribute of a `<meta>` element after the element has been parsed will be ignored.

§ 4. Integrations

This section is non-normative.

This document defines a set of algorithms which are used in other specifications in order to implement the functionality. These integrations are outlined here for clarity, but those external documents are the normative references which ought to be consulted for detailed information.

§ 4.1. Integration with Fetch

A number of directives control resource loading in one way or another. This specification provides algorithms which allow Fetch to make decisions about whether or not a particular request should be blocked or allowed, and about whether a particular response should be replaced with a network error.

1. §4.1.3 Should request be blocked by Content Security Policy? is called as part of step #5 of its Main Fetch algorithm. This allows directives' pre-request checks to be executed against each request before it hits the network, and against each redirect that a request might go through on its way to reaching a resource.
2. §4.1.4 Should response to request be blocked by Content Security Policy? is called as part of step #13 of its Main Fetch algorithm. This allows directives' post-request checks and response checks to be executed on the response delivered from the network or from a Service Worker.

A policy is generally enforced upon a global object, but the user agent needs to parse any policy delivered via an HTTP response header field before any global object is created in order to handle directives that require knowledge of a response's details. To that end:

1. A response has an associated CSP list which contains any policy objects delivered in the response's header list.
2. §4.1.1 Set response's CSP list is called in the HTTP fetch and HTTP-network fetch algorithms.

> Note: These two calls should ensure that a response's CSP list is set, regardless of how the response is created. If we hit the network (via HTTP-network fetch, then we parse the policy before we handle the `Set-Cookie` header. If we get a response from a Service Worker (via HTTP fetch, we'll process its CSP list before handing the response back to our caller.

§ 4.1.1. Set *response*'s `CSP list`

Given a response (*response*), this algorithm evaluates its header list for serialized CSP values, and populates its CSP list accordingly:

1. Set *response*'s CSP list to the empty list.
2. Let *policies* be the result of parsing the result of extracting header list values given `Content-Security-Policy` and *response*'s header list, with a source of `"header"`, and a disposition of `"enforce"`.
3. Append to *policies* the result of parsing the result of extracting header list values given `Content-Security-Policy-Report-Only` and *response*'s header list, with a source of `"header"`, and a disposition of `"report"`.
4. For each *policy* in *policies*:
    1. Insert *policy* into *response*'s CSP list.

§ 4.1.2. Report Content Security Policy violations for *request*

Given a request (*request*), this algorithm reports violations based on client's "report only" policies.

1. Let *CSP list* be *request*'s client's global object's CSP list.
2. For each *policy* in *CSP list*:
    1. If *policy*'s disposition is `"enforce"`, then skip to the next *policy*.
    2. Let *violates* be the result of executing §6.6.2.1 Does request violate policy? on *request* and *policy*.
    3. If *violates* is not `"Does Not Violate"`, then execute §5.3 Report a violation on the result of executing §2.4.2 Create a violation object for request, and policy. on *request*, and *policy*.

§ 4.1.3. Should *request* be blocked by Content Security Policy?

Given a request (*request*), this algorithm returns `Blocked` or `Allowed` and reports violations based on *request*'s client's Content Security Policy.

1. Let *CSP list* be *request*'s client's global object's CSP list.
2. Let *result* be `"Allowed"`.
3. For each *policy* in *CSP list*:
    1. If *policy*'s disposition is `"report"`, then skip to the next *policy*.
    2. Let *violates* be the result of executing §6.6.2.1 Does request violate policy? on *request* and *policy*.
    3. If *violates* is not `"Does Not Violate"`, then:
        1. Execute §5.3 Report a violation on the result of executing §2.4.2 Create a violation object for request, and policy. on *request*, and *policy*.
        2. Set *result* to `"Blocked"`.
4. Return *result*.

§ 4.1.4. Should *response* to *request* be blocked by Content Security Policy?

Given a response (*response*) and a request (*request*), this algorithm returns `Blocked` or `Allowed`, and reports violations based on *request*'s client's Content Security Policy.

1. Let *CSP list* be *request*'s client's global object's CSP list.
2. Let *result* be "`Allowed`".
3. For each *policy* in *CSP list*:
    1. For each *directive* in *policy*:
        1. If the result of executing *directive*'s post-request check is "`Blocked`", then:
            1. Execute §5.3 Report a violation on the result of executing §2.4.2 Create a violation object for request, and policy. on *request*, and *policy*.
            2. If *policy*'s disposition is "`enforce`", then set *result* to "`Blocked`".

> Note: This portion of the check verifies that the page can load the response. That is, that a Service Worker hasn't substituted a file which would violate the page's CSP.

4. For each *policy* in *response*'s CSP list:
    1. For each *directive* in *policy*:
        1. If the result of executing *directive*'s response check on *request*, *response*, and *policy* is "`Blocked`", then:
            1. Execute §5.3 Report a violation on the result of executing §2.4.2 Create a violation object for request, and policy. on *request*, and *policy*.
            2. If *policy*'s disposition is "`enforce`", then set *result* to "`Blocked`".

> Note: This portion of the check allows policies delivered with the response to determine whether the response is allowed to be delivered.

5. Return *result*.

§ 4.2. Integration with HTML

1. The `Document`, `WorkerGlobalScope`, and `WorkletGlobalScope` objects have a `CSP list`, which holds all the policy objects which are active for a given context. This list is empty unless otherwise specified, and is populated via the §4.2.2 Initialize a global object's CSP list and §4.2.1 Initialize a Document's CSP list algorithms.

> ISSUE 3 This concept is missing from W3C's Workers. <https://github.com/w3c/html/issues/187>

2. A global object's ***CSP list*** is the result of executing §4.2.3 Retrieve the CSP list of an object with the global object as the `object`.
3. A policy is ***enforced*** or ***monitored*** for a global object by inserting it into the global object's CSP list.
4. §4.2.2 Initialize a global object's CSP list is called during the run a worker algorithm in order to bind a set of policy objects associated with a response `WorkerGlobalScope` or `WorkletGlobalScope`.
5. §4.2.1 Initialize a Document's CSP list is called during the initializing a new `Document` object algorithm in order to bind a set of policy objects associated with a response to a newly created `Document`.
6. §4.2.4 Should element's inline type behavior be blocked by Content Security Policy? is called during the prepare a script and update a `style` block algorithms in order to determine whether or not an inline script or style block is allowed to execute/render.
7. §4.2.4 Should element's inline type behavior be blocked by Content Security Policy? is called during handling of inline event handlers (like `onclick`) and inline `style` attributes in order to determine whether or not they ought to be allowed to execute/render.
8. policy is enforced during processing of the `<meta>` element's `http-equiv`.
9. A `Document`'s ***embedding document*** is the `Document` through which the `Document`'s browsing context is nested.
10. HTML populates each request's cryptographic nonce metadata and parser metadata with relevant data from the elements responsible for resource loading.

> ISSUE 4 Stylesheet loading is not yet integrated with Fetch in W3C's HTML. <https://github.com/whatwg/html/issues/198>

> ISSUE 5 Stylesheet loading is not yet integrated with Fetch in WHATWG's HTML. <https://github.com/whatwg/html/issues/968>

11. §6.2.1.1 Is base allowed for document? is called during `<base>`'s set the frozen base URL algorithm to ensure that the `href` attribute's value is valid.
12. §6.2.2.2 Should plugin element be blocked a priori by Content Security Policy?: is called during the processing of `<object>`, `<embed>`, and `applet` elements to determine whether they may trigger a fetch.

> Note: Fetched plugin resources are handled in §4.1.4 Should response to request be blocked by Content Security Policy?.

> ISSUE 6 This hook is missing from W3C's HTML. <https://github.com/w3c/html/issues/547>

13. §4.2.5 Should navigation request of type from source in target be blocked by Content Security Policy? is called during the process a navigate fetch algorithm, and §4.2.6 Should navigation response to navigation request of type from source in target be blocked by Content Security Policy? is called during the process a navigate response algorithm to apply directive's navigation checks, as well as inline checks for navigations to `javascript:` URLs.

> ISSUE 7 W3C's HTML is not based on Fetch, and does not have a process a navigate response algorithm into which to hook. <https://github.com/w3c/html/issues/548>

§ 4.2.1. Initialize a `Document`'s `CSP list`

Given a `Document` (*document*), and a response (*response*), the user agent performs the following steps in order to initialize *document*'s CSP list:

1. If *response*'s url's scheme is a local scheme:
    1. Let *documents* be an empty list.
    2. If *document* has an embedding document (*embedding*), then add *embedding* to *documents*.
    3. If *document* has an opener browsing context, then add its active document to *documents*.
    4. For each *doc* in *documents*:
        1. For each *policy* in *doc*'s CSP list:
            1. Insert a copy of *policy* into *document*'s CSP list.

> Note: local scheme includes `about:`, and this algorithm will therefore copy the embedding document's policies for an iframe `srcdoc` `Document`.

> Note: We do all this to ensure that a page cannot bypass its policy by embedding a frame or popping up a new window containing content it controls (`blob:` resources, or `document.write()`).

2. For each *policy* in *response*'s CSP list, insert *policy* into *document*'s CSP list.
3. For each *policy* in *document*'s CSP list:
    1. For each *directive* in *policy*:
        1. Execute *directive*'s initialization algorithm on *document* and *response*.

§ 4.2.2. Initialize a global object's `CSP list`

Given a global object (*global*), and a response (*response*), the user agent performs the following steps in order to initialize *global*'s CSP list:

1. If *response*'s url's scheme is a local scheme, or if *global* is a `DedicatedWorkerGlobalScope`:
    1. Let *owners* be an empty list.
    2. Add each of the items in *global*'s owner set to *owners*.
    3. For each *owner* in *owners*:
        1. For each *policy* in *owner*'s CSP list:
            1. Insert a copy of *policy* into *global*'s CSP list.

> Note: local scheme includes `about:`, and this algorithm will therefore copy the embedding document's policies for an iframe `srcdoc Document`.

2. If *global* is a `SharedWorkerGlobalScope` or `ServiceWorkerGlobalScope`:
    1. For each *policy* in *response*'s CSP list, insert *policy* into *global*'s CSP list.
3. If *global* is a `WorkletGlobalScope`:
    1. Let *owner* be *global*'s owner document.
    2. For each *policy* in *owner*'s CSP list:
        1. Insert a copy of *policy* into *global*'s CSP list.

§ 4.2.3. Retrieve the CSP list of an *object*

To obtain *object*'s CSP list:

1. If *object* is a `Document` return *object*'s CSP list.
2. If *object* is a `Window` return *object*'s associated `Document`'s CSP list.
3. If *object* is a `WorkerGlobalScope`, return *object*'s CSP list.
4. If *object* is a `WorkletGlobalScope`, return *object*'s CSP list.
5. Return `null`.

§ 4.2.4. Should *element*'s inline *type* behavior be blocked by Content Security Policy?

Given an `Element` (*element*), a string (*type*), and a string (*source*) this algorithm returns `"Allowed"` if the element is allowed to have inline definition of a particular type of behavior (script execution, style application, event handlers, etc.), and `"Blocked"` otherwise:

> Note: The valid values for *type* are `"script"`, `"script attribute"`, `"style"`, and `"style attribute"`.

1. Assert: *element* is not `null`.
2. Let *result* be `"Allowed"`.
3. For each *policy* in *element*'s `Document`'s global object's CSP list:
    1. For each *directive* in *policy*'s directive set:
        1. If *directive*'s inline check returns `"Allowed"` when executed upon *element*, *type*, *policy* and *source*, skip to the next *directive*.
        2. Let *directive-name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
        3. Otherwise, let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on the current settings object's global object, *policy*, and *directive-name*.
        4. Set *violation*'s resource to `"inline"`.
        5. Set *violation*'s element to *element*.
        6. If *directive*'s value contains the expression `"'report-sample'"`, then set *violation*'s sample to the substring of *source* containing its first 40 characters.
        7. Execute §5.3 Report a violation on *violation*.
        8. If *policy*'s disposition is `"enforce"`, then set *result* to `"Blocked"`.

4. Return *result*.

§ 4.2.5. Should *navigation request* of *type* from *source* in *target* be blocked by Content Security Policy?

Given a request (*navigation request*), a string (*type*, either "`form-submission`" or "`other`"), and two browsing contexts (*source* and *target*), this algorithm return "`Blocked`" if the active policy blocks the navigation, and "`Allowed`" otherwise:

1. Let *result* be "`Allowed`".
2. For each *policy* in *source*'s active document's CSP list:
    1. For each *directive* in *policy*:
        1. If *directive*'s pre-navigation check returns "`Allowed`" when executed upon *navigation request*, *type*, *source*, *target*, and *policy* skip to the next *directive*.
        2. Otherwise, let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on *source*'s relevant global object, *policy*, and *directive*'s name.
        3. Set *violation*'s resource to *navigation request*'s URL.
        4. Execute §5.3 Report a violation on *violation*.
        5. If *policy*'s disposition is "`enforce`", then set *result* to "`Blocked`".
3. If *result* is "`Allowed`", and if *navigation request*'s current URL's scheme is `javascript`:
    1. For each *policy* in *source*'s active document's CSP List:
        1. For each *directive* in *policy*:
            1. If *directive*'s inline check returns "`Allowed`" when executed upon `null`, "`navigation`" and *navigation request*'s current URL, skip to the next *directive*.
            2. Let *directive-name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
            3. Otherwise, let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on *source*'s relevant global object, *policy*, and *directive-name*.
            4. Set *violation*'s resource to *navigation request*'s URL.

5. Execute §5.3 Report a violation on *violation*.
6. If *policy*'s disposition is "enforce", then set *result* to "Blocked".

4. Return *result*.

§ 4.2.6. Should *navigation response* to *navigation request* of *type* from *source* in *target* be blocked by Content Security Policy?

Given a request (*navigation request*), a string (*type*, either "form-submission" or "other"), a response *navigation response*, and two browsing contexts (*source* and *target*), this algorithm returns "Blocked" if the active policy blocks the navigation, and "Allowed" otherwise:

1. Let *result* be "Allowed".
2. For each *policy* in *navigation response*'s CSP list:

> Note: Some directives (like frame-ancestors) allow a *response*'s Content Security Policy to act on the navigation.

1. For each *directive* in *policy*:
    1. If *directive*'s navigation response check returns "Allowed" when executed upon *navigation request*, *type*, *navigation response*, *source*, *target*, "response", and *policy* skip to the next *directive*.
    2. Otherwise, let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on null, *policy*, and *directive*'s name.

        > Note: We use null for the global object, as no global exists: we haven't processed the navigation to create a Document yet.

    3. Set *violation*'s resource to *navigation response*'s URL.
    4. Execute §5.3 Report a violation on *violation*.
    5. If *policy*'s disposition is "enforce", then set *result* to "Blocked".
3. For each *policy* in *source*'s active document's CSP List:

> Note: Some directives in the *source* context (like navigate-to) need the *response* before acting on the navigation.

1. For each *directive* in *policy*:

1. If *directive*'s navigation response check returns "`Allowed`" when executed upon *navigation request*, *type*, *navigation response*, *source*, *target*, "`source`", and *policy* skip to the next *directive*.
2. Otherwise, let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on *source*'s relevant global object, *policy*, and *directive*'s name.
3. Set *violation*'s resource to *navigation request*'s URL.
4. Execute §5.3 Report a violation on *violation*.
5. If *policy*'s disposition is "`enforce`", then set *result* to "`Blocked`".

4. Return *result*.

§ 4.3. Integration with ECMAScript

ECMAScript defines a `HostEnsureCanCompileStrings()` abstract operation which allows the host environment to block the compilation of strings into ECMAScript code. This document defines an implementation of that abstract operation thich examines the relevant CSP list to determine whether such compilation ought to be blocked.

§ 4.3.1. EnsureCSPDoesNotBlockStringCompilation(*callerRealm*, *calleeRealm*, *source*)

Given two realms (*callerRealm* and *calleeRealm*), and a string (*source*), this algorithm returns normally if string compilation is allowed, and throws an "`EvalError`" if not:

1. Let *globals* be a list containing *callerRealm*'s global object and *calleeRealm*'s global object.
2. For each *global* in *globals*:
    1. Let *result* be "`Allowed`".
    2. For each *policy* in *global*'s CSP list:
        1. Let *source-list* be `null`.
        2. If *policy* contains a directive whose name is "`script-src`", then set *source-list* to that directive's value.

           Otherwise if *policy* contains a directive whose name is "`default-src`", then set *source-list* to that directive's value.
        3. If *source-list* is not `null`, and does not contain a source expression which is an ASCII

case-insensitive match for the string "`'unsafe-eval'`", then:

1. Let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on *global*, *policy*, and "`script-src`".
2. Set *violation*'s resource to "`inline`".
3. If *source-list* contains the expression "`'report-sample'`", then set *violation*'s sample to the substring of *source* containing its first 40 characters.
4. Execute §5.3 Report a violation on *violation*.
5. If *policy*'s disposition is "`enforce`", then set *result* to "`Blocked`".

3. If *result* is "`Blocked`", throw an `EvalError` exception.

ISSUE 8 `HostEnsureCanCompileStrings()` does not include the string which is going to be compiled as a parameter. We'll also need to update HTML to pipe that value through to CSP. <https://github.com/tc39/ecma262/issues/938>

§ 5. Reporting

When one or more of a policy's directives is violated, a ***violation report*** may be generated and sent out to a reporting endpoint associated with the policy.

§ 5.1. Violation DOM Events

```
enum SecurityPolicyViolationEventDisposition {
  "enforce", "report"
};

[Constructor(DOMString type, optional SecurityPolicyViolationEventInit eventInitD
 Exposed=(Window,Worker)]
interface SecurityPolicyViolationEvent : Event {
    readonly    attribute USVString      documentURI;
    readonly    attribute USVString      referrer;
    readonly    attribute USVString      blockedURI;
    readonly    attribute DOMString      violatedDirective;
    readonly    attribute DOMString      effectiveDirective;
    readonly    attribute DOMString      originalPolicy;
    readonly    attribute USVString      sourceFile;
```

```
    readonly    attribute DOMString      sample;
    readonly    attribute SecurityPolicyViolationEventDisposition      dispositic
    readonly    attribute unsigned short statusCode;
    readonly    attribute unsigned long  lineNumber;
    readonly    attribute unsigned long  columnNumber;
};

dictionary SecurityPolicyViolationEventInit : EventInit {
    required USVString      documentURI;
             USVString      referrer = "";
             USVString      blockedURI = "";
    required DOMString      violatedDirective;
    required DOMString      effectiveDirective;
    required DOMString      originalPolicy;
             USVString      sourceFile = "";
             DOMString      sample = "";
    required SecurityPolicyViolationEventDisposition disposition;
    required unsigned short statusCode;
             unsigned long  lineNumber = 0;
             unsigned long  columnNumber = 0;
};
```

§ 5.2. Obtain the deprecated serialization of *violation*

Given a violation (*violation*), this algorithm returns a JSON text string representation of the violation, suitable for submission to a reporting endpoint associated with the deprecated `report-uri` directive.

1. Let *object* be a new JavaScript object with properties initialized as follows:

   `"document-uri"`
   : The result of executing the URL serializer on *violation*'s url, with the `exclude fragment` flag set.

   `"referrer"`
   : The result of executing the URL serializer on *violation*'s referrer, with the `exclude fragment` flag set.

   `"blocked-uri"`
   : The result of executing the URL serializer on *violation*'s resource, with the `exclude fragment` flag set.

   `"effective-directive"`
   : *violation*'s effective directive

"violated-directive"
violation's effective directive

"original-policy"
The serialization of *violation*'s policy

"disposition"
The disposition of *violation*'s policy

"status-code"
violation's status

"script-sample"
violation's sample

> Note: The name `script-sample` was chosen for compatibility with an earlier iteration of this feature which has shipped in Firefox since its initial implementation of CSP. Despite the name, this field will contain samples for non-script violations, like stylesheets. The data contained in a `SecurityPolicyViolationEvent` object, and in reports generated via the new `report-to` directive, is named in a more encompassing fashion: `sample`.

2. If *violation*'s source file is not `null`:
    1. Set *object*'s "`source-file`" property to the result of executing the URL serializer on *violation*'s source file, with the `exclude fragment` flag set.
    2. Set *object*'s "`line-number`" property to *violation*'s line number.
    3. Set *object*'s "`column-number`" property to *violation*'s column number.
3. Assert: If *object*'s "`blocked-uri`" property is not "`inline`", then its "`sample`" property is the empty string.
4. Return the result of executing `JSON.stringify()` on *object*.

§ 5.3. Report a *violation*

Given a violation (*violation*), this algorithm reports it to the endpoint specified in *violation*'s policy, and fires a `SecurityPolicyViolationEvent` at *violation*'s element, or at *violation*'s global object as described below:

1. Let *global* be *violation*'s global object.
2. Let *target* be *violation*'s element.
3. Queue a task to run the following steps:

> Note: We "queue a task" here to ensure that the event targeting and dispatch happens after JavaScript completes execution of the task responsible for a given violation (which might manipulate the DOM).

1. If *target* is not `null`, and *global* is a `Window`, and *target*'s shadow-including root is not *global*'s associated `Document`, set *target* to `null`.

   > Note: This ensures that we fire events only at elements connected to *violation*'s policy's `Document`. If a violation is caused by an element which isn't connected to that document, we'll fire the event at the document rather than the element in order to ensure that the violation is visible to the document's listeners.

2. If *target* is `null`:

   1. Set *target* be *violation*'s global object.
   2. If *target* is a `Window`, set *target* to *target*'s associated `Document`.

3. Fire an event named `securitypolicyviolation` that uses the `SecurityPolicyViolationEvent` interface at *target* with its attributes initialized as follows:

   `documentURI`
   : The result of executing the URL serializer on *violation*'s url, with the `exclude fragment` flag set.

   `referrer`
   : The result of executing the URL serializer on *violation*'s referrer, with the `exclude fragment` flag set.

   `blockedURI`
   : The result of executing the URL serializer on *violation*'s resource, with the `exclude fragment` flag set.

   `effectiveDirective`
   : *violation*'s effective directive

   `violatedDirective`
   : *violation*'s effective directive

   `originalPolicy`
   : The serialization of *violation*'s policy

   `disposition`
   : *violation*'s disposition

   `sourceFile`

The result of executing the URL serializer on *violation*'s source file, with the `exclude fragment` flag set if the *violation*'s source file it not `null` and the empty string otherwise.

`statusCode`
violation's status

`lineNumber`
violation's line number

`columnNumber`
violation's column number

`sample`
violation's sample

`bubbles`
`true`

`composed`
`true`

> Note: Both `effectiveDirective` and `violatedDirective` are the same value. This is intentional to maintain backwards compatibility.

> Note: We set the `composed` attribute, which means that this event can be captured on its way into, and will bubble its way out of a shadow tree. `target`, et al will be automagically scoped correctly for the main tree.

4. If *violation*'s policy's directive set contains a directive named "`report-uri`" (*directive*):
    1. If *violation*'s policy's directive set contains a directive named "`report-to`", skip the remaining substeps.
    2. For each *token* returned by splitting a string on ASCII whitespace with *directive*'s value as the `input`.
        1. Let *endpoint* be the result of executing the URL parser with *token* as the input, and *violation*'s url as the base URL.
        2. If *endpoint* is not a valid URL, skip the remaining substeps.
        3. Let *request* be a new request, initialized as follows:

            method
            `"POST"`

            url

violation's url

origin
violation's global object's relevant settings object's origin

window
"no-window"

client
violation's global object's relevant settings object

destination
"report"

initiator
""

credentials mode
"same-origin"

keepalive flag
"true"

header list
A header list containing a single header whose name is "Content-Type", and value is "application/csp-report"

body
The result of executing §5.2 Obtain the deprecated serialization of violation on *violation*

redirect mode
"error"

> Note: *request*'s mode defaults to "no-cors"; the response is ignored entirely.

4. Fetch *request*. The result will be ignored.

> Note: All of this should be considered deprecated. It sends a single request per violation, which simply isn't scalable. As soon as this behavior can be removed from user agents, it will be.

> Note: report-uri only takes effect if report-to is not present. That is, the latter overrides the former, allowing for backwards compatibility with browsers that don't support the new mechanism.

5. If *violation*'s policy's directive set contains a directive named "report-to" (*directive*):

1. Let *group* be *directive*'s value.
2. Let *settings object* be *violation*'s global object's relevant settings object.
3. Execute [REPORTING]'s Queue *data* as *type* for *endpoint group* on *settings* algorithm with the following arguments:

   data
   violation

   type
   "CSP"

   endpoint group
   group

   settings
   settings object

§ 6. Content Security Policy Directives

This specification defines a number of types of directives which allow developers to control certain aspects of their sites' behavior. This document defines directives which govern resource fetching (in §6.1 Fetch Directives), directives which govern the state of a document (in §6.2 Document Directives), directives which govern aspects of navigation (in §6.3 Navigation Directives), and directives which govern reporting (in §6.4 Reporting Directives). These form the core of Content Security Policy; other directives are defined in a modular fashion in ancillary documents (see §6.5 Directives Defined in Other Documents for examples).

To mitigate the risk of cross-site scripting attacks, web developers SHOULD include directives that regulate sources of script and plugins. They can do so by including:

- Both the script-src and object-src directives, or
- a default-src directive

In either case, developers SHOULD NOT include either `'unsafe-inline'`, or `data:` as valid sources in their policies. Both enable XSS attacks by allowing code to be included directly in the document itself; they are best avoided completely.

§ 6.1. Fetch Directives

Fetch directives control the locations from which certain resource types may be loaded. For instance,

script-src allows developers to allow trusted sources of script to execute on a page, while font-src controls the sources of web fonts.

§ 6.1.1. `child-src`

The ***`child-src`*** directive governs the creation of nested browsing contexts (e.g. `<iframe>` and `<frame>` navigations) and Worker execution contexts. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "child-src"
directive-value = serialized-source-list
```

This directive controls requests which will populate a frame or a worker. More formally, requests falling into one of the following categories:

- destination is "`document`", and whose target browsing context is a nested browsing context (e.g. requests which will populate an `<iframe>` or `<frame>` element)
- destination is either "`serviceworker`", "`sharedworker`", or "`worker`" (which are fed to the run a worker algorithm for `ServiceWorker`, `SharedWorker`, and `Worker`, respectively).

> EXAMPLE 5
> Given a page with the following Content Security Policy:
>
> `Content-Security-Policy: child-src https://example.com/`
>
> Fetches for the following code will all return network errors, as the URLs provided do not match `child-src`'s source list:
>
> ```
> <iframe src="https://example.org"></iframe>
> <script>
>   var blockedWorker = new Worker("data:application/javascript,...");
> </script>
> ```

§ *6.1.1.1. `child-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `child-src` and *policy* is "`No`", return "`Allowed`".
3. Return the result of executing the pre-request check for the directive whose name is *name* on *request* and *policy*, using this directive's value for the comparison.

§ *6.1.1.2. `child-src` Post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `child-src` and *policy* is "`No`", return "`Allowed`".
3. Return the result of executing the post-request check for the directive whose name is *name* on *request*, *response*, and *policy*, using this directive's value for the comparison.

§ 6.1.2. `connect-src`

The ***connect-src*** directive restricts the URLs which can be loaded using script interfaces. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "connect-src"
directive-value = serialized-source-list
```

This directive controls requests which transmit or receive data from other origins. This includes APIs like `fetch()`, [XHR], [EVENTSOURCE], [BEACON], and `<a>`'s `ping`. This directive *also* controls WebSocket [WEBSOCKETS] connections, though those aren't technically part of Fetch.

EXAMPLE 6

JavaScript offers a few mechanisms that directly connect to an external server to send or receive information. `EventSource` maintains an open HTTP connection to a server in order to receive push notifications, `WebSockets` open a bidirectional communication channel between your browser and a server, and `XMLHttpRequest` makes arbitrary HTTP requests on your behalf. These are powerful APIs that enable useful functionality, but also provide tempting avenues for data exfiltration.

The `connect-src` directive allows you to ensure that these and similar sorts of connections are only opened to origins you trust. Sending a policy that defines a list of source expressions for this directive is straightforward. For example, to limit connections to only `https://example.com`, send the following header:

```
Content-Security-Policy: connect-src https://example.com/
```

Fetches for the following code will all return network errors, as the URLs provided do not match `connect-src`'s source list:

```
<a ping="https://example.org">...
<script>
  var xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://example.org/');
  xhr.send();

  var ws = new WebSocket("wss://example.org/");

  var es = new EventSource("https://example.org/");

  navigator.sendBeacon("https://example.org/", { ... });
</script>
```

§ *6.1.2.1. `connect-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `connect-src` and *policy*

is "No", return "Allowed".

3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "Does Not Match", return "Blocked".
4. Return "Allowed".

§ *6.1.2.2. `connect-src` Post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `connect-src` and *policy* is "No", return "Allowed".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "Does Not Match", return "Blocked".
4. Return "Allowed".

§ 6.1.3. `default-src`

The ***default-src*** directive serves as a fallback for the other fetch directives. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "default-src"
directive-value = serialized-source-list
```

If a default-src directive is present in a policy, its value will be used as the policy's default source list. That is, given `default-src 'none'; script-src 'self'`, script requests will use `'self'` as the source list to match against. Other requests will use `'none'`. This is spelled out in more detail in the §4.1.3 Should request be blocked by Content Security Policy? and §4.1.4 Should response to request be blocked by Content Security Policy? algorithms.

EXAMPLE 7

The following header:

```
Content-Security-Policy: default-src 'self'
```

will have the same behavior as the following header:

```
Content-Security-Policy: connect-src 'self';
                         font-src 'self';
                         frame-src 'self';
                         img-src 'self';
                         manifest-src 'self';
                         media-src 'self';
                         prefetch-src 'self';
                         object-src 'self';
                         script-src-elem 'self';
                         script-src-attr 'self';
                         style-src-elem 'self';
                         style-src-attr 'self';
                         worker-src 'self'
```

That is, when `default-src` is set, every fetch directive that isn't explicitly set will fall back to the value `default-src` specifies.

EXAMPLE 8

There is no inheritance. If a `script-src` directive is explicitly specified, for example, then the value of `default-src` has no influence on script requests. That is, the following header:

```
Content-Security-Policy: default-src 'self'; script-src-elem https://example.com
```

will have the same behavior as the following header:

```
Content-Security-Policy: connect-src 'self';
                         font-src 'self';
                         frame-src 'self';
                         img-src 'self';
                         manifest-src 'self';
                         media-src 'self';
                         prefetch-src 'self';
                         object-src 'self';
                         script-src-elem https://example.com;
                         script-src-attr 'self';
                         style-src-elem 'self';
                         style-src-attr 'self';
                         worker-src 'self'
```

Given this behavior, one good way to build a policy for a site would be to begin with a `default-src` of `'none'`, and to build up a policy from there which allowed only those resource types which are necessary for the particular page the policy will apply to.

§ *6.1.3.1. `default-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `default-src` and *policy* is `"No"`, return `"Allowed"`.
3. Return the result of executing the pre-request check for the directive whose name is *name* on *request* and *policy*, using this directive's value for the comparison.

§ *6.1.3.2. `default-src` Post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `default-src` and *policy* is `"No"`, return `"Allowed"`.
3. Return the result of executing the post-request check for the directive whose name is *name* on *request*, *response*, and *policy*, using this directive's value for the comparison.

6.1.3.3. `default-src` Inline Check

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `default-src` and *policy* is `"No"`, return `"Allowed"`.
3. Otherwise, return the result of executing the inline check for the directive whose name is *name* on *element*, *type*, *policy* and *source*, using this directive's value for the comparison.

6.1.4. `font-src`

The ***font-src*** directive restricts the URLs from which font resources may be loaded. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "font-src"
directive-value = serialized-source-list
```

EXAMPLE 9

Given a page with the following Content Security Policy:

```
Content-Security-Policy: font-src https://example.com/
```

Fetches for the following code will return a network errors, as the URL provided do not match `font-src`'s source list:

```
<style>
  @font-face {
    font-family: "Example Font";
    src: url("https://example.org/font");
  }
  body {
    font-family: "Example Font";
  }
</style>
```

6.1.4.1. `font-src` Pre-request check

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `font-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

6.1.4.2. `font-src` Post-request check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `font-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ 6.1.5. `frame-src`

The ***frame-src*** directive restricts the URLs which may be loaded into nested browsing contexts. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "frame-src"
directive-value = serialized-source-list
```

> EXAMPLE 10
> Given a page with the following Content Security Policy:
>
> `Content-Security-Policy: frame-src https://example.com/`
>
> Fetches for the following code will return a network errors, as the URL provided do not match `frame-src`'s source list:
>
> ```
> <iframe src="https://example.org/">
> </iframe>
> ```

§ *6.1.5.1. `frame-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `frame-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's

value is "Does Not Match", return "Blocked".

4. Return "Allowed".

§ 6.1.5.2. frame-src Post-request check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, frame-src and *policy* is "No", return "Allowed".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "Does Not Match", return "Blocked".
4. Return "Allowed".

§ 6.1.6. img-src

The ***img-src*** directive restricts the URLs from which image resources may be loaded. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "img-src"
directive-value = serialized-source-list
```

This directive controls requests which load images. More formally, this includes requests whose destination is "image" [FETCH].

> EXAMPLE 11
> Given a page with the following Content Security Policy:
>
> ```
> Content-Security-Policy: img-src https://example.com/
> ```
>
> Fetches for the following code will return a network errors, as the URL provided do not match img-src's source list:
>
> ```
> <img src="https://example.org/img">
> ```

§ *6.1.6.1. `img-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `img-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ *6.1.6.2. `img-src` Post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `frame-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ 6.1.7. `manifest-src`

The ***manifest-src*** directive restricts the URLs from which application manifests may be loaded [APPMANIFEST]. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "manifest-src"
directive-value = serialized-source-list
```

EXAMPLE 12

Given a page with the following Content Security Policy:

```
Content-Security-Policy: manifest-src https://example.com/
```

Fetches for the following code will return a network errors, as the URL provided do not match `manifest-src`'s source list:

```
<link rel="manifest" href="https://example.org/manifest">
```

§ 6.1.7.1. `manifest-src` Pre-request check

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `manifest-src` and *policy* is `"No"`, return `"Allowed"`.
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is `"Does Not Match"`, return `"Blocked"`.
4. Return `"Allowed"`.

§ 6.1.7.2. `manifest-src` Post-request check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `manifest-src` and *policy* is `"No"`, return `"Allowed"`.
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is `"Does Not Match"`, return `"Blocked"`.
4. Return `"Allowed"`.

6.1.8. `media-src`

The ***media-src*** directive restricts the URLs from which video, audio, and associated text track resources may be loaded. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "media-src"
directive-value = serialized-source-list
```

> EXAMPLE 13
> Given a page with the following Content Security Policy:
>
> ```
> Content-Security-Policy: media-src https://example.com/
> ```
>
> Fetches for the following code will return a network errors, as the URL provided do not match `media-src`'s source list:
>
> ```
> <audio src="https://example.org/audio"></audio>
> <video src="https://example.org/video">
>     <track kind="subtitles" src="https://example.org/subtitles">
> </video>
> ```

6.1.8.1. `media-src` Pre-request check

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `media-src` and *policy* is `"No"`, return `"Allowed"`.
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is `"Does Not Match"`, return `"Blocked"`.
4. Return `"Allowed"`.

6.1.8.2. `media-src` Post-request check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `media-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ 6.1.9. `prefetch-src`

The ***prefetch-src*** directive restricts the URLs from which resources may be prefetched or prerendered. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "prefetch-src"
directive-value = serialized-source-list
```

> EXAMPLE 14
> Given a page with the following Content Security Policy:
>
> `Content-Security-Policy: prefetch-src https://example.com/`
>
> Fetches for the following code will return network errors, as the URLs provided do not match `prefetch-src`'s source list:
>
> ```
> <link rel="prefetch" src="https://example.org/"></link>
> <link rel="prerender" src="https://example.org/"></link>
> ```

§ *6.1.9.1. `prefetch-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.

2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `prefetch-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

6.1.9.2. `prefetch-src` Post-request check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `prefetch-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

6.1.10. `object-src`

The ***object-src*** directive restricts the URLs from which plugin content may be loaded. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "object-src"
directive-value = serialized-source-list
```

EXAMPLE 15

Given a page with the following Content Security Policy:

```
Content-Security-Policy: object-src https://example.com/
```

Fetches for the following code will return a network errors, as the URL provided do not match `object-src`'s source list:

```
<embed src="https://example.org/flash"></embed>
<object data="https://example.org/flash"></object>
<applet archive="https://example.org/flash"></applet>
```

If plugin content is loaded without an associated URL (perhaps an `<object>` element lacks a `data` attribute, but loads some default plugin based on the specified `type`), it MUST be blocked if `object-src`'s value is `'none'`, but will otherwise be allowed.

Note: The `object-src` directive acts upon any request made on behalf of an `<object>`, `<embed>`, or `applet` element. This includes requests which would populate the nested browsing context generated by the former two (also including navigations). This is true even when the data is semantically equivalent to content which would otherwise be restricted by another directive, such as an `<object>` element with a `text/html` MIME type.

Note: When a plugin resource is navigated to directly (that is, as a plugin document in the top-level browsing context or a nested browsing context, and not as an embedded subresource via `<embed>`, `<object>`, or `applet`), any policy delivered along with that resource will be applied to the plugin document. This means, for instance, that developers can prevent the execution of arbitrary resources as plugin content by delivering the policy `object-src 'none'` along with a response. Given plugins' power (and the sometimes-interesting security model presented by Flash and others), this could mitigate the risk of attack vectors like Rosetta Flash.

§ *6.1.10.1. `object-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.

2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `object-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ *6.1.10.2. `object-src` Post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `object-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ 6.1.11. `script-src`

The ***script-src*** directive restricts the locations from which scripts may be executed. This includes not only URLs loaded directly into `<script>` elements, but also things like inline script blocks and XSLT stylesheets [XSLT] which can trigger script execution. The syntax for the directive's name and value is described by the following ABNF:

The `script-src` directive acts as a default fallback for all script-like destinations (including worker-specific destinations if `worker-src` is not present). Unless granularity is desired `script-src` should be used in favor of `script-src-attr` and `script-src-elem` as in most situations there is no particular reason to have separate lists of permissions for inline event handlers and `<script>` elements.

```
directive-name  = "script-src"
directive-value = serialized-source-list
```

The `script-src` directive governs five things:

1. Script requests MUST pass through §4.1.3 Should request be blocked by Content Security Policy?.

2. Script responses MUST pass through §4.1.4 Should response to request be blocked by Content Security Policy?.
3. Inline `<script>` blocks MUST pass through §4.2.4 Should element's inline type behavior be blocked by Content Security Policy?. Their behavior will be blocked unless every policy allows inline script, either implicitly by not specifying a `script-src` (or `default-src`) directive, or explicitly, by specifying "`unsafe-inline`", a nonce-source or a hash-source that matches the inline block.
4. The following JavaScript execution sinks are gated on the "`unsafe-eval`" source expression:
   - `eval()`
   - `Function()`
   - `setTimeout()` with an initial argument which is not callable.
   - `setInterval()` with an initial argument which is not callable.

   > Note: If a user agent implements non-standard sinks like `setImmediate()` or `execScript()`, they SHOULD also be gated on "`unsafe-eval`". Note: Since "`unsafe-eval`" acts as a global page flag, `script-src-attr` and `script-src-elem` are not used when performing this check, instead `script-src` (or it's fallback directive) is always used.
5. Navigation to `javascript:` URLs MUST pass through §6.1.11.3 script-src Inline Check.

§ *6.1.11.1. `script-src` Pre-request check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src` and *policy* is "`No`", return "`Allowed`".
3. Return the result of executing §6.6.1.1 Script directives pre-request check on *request* and this directive.

§ *6.1.11.2. `script-src` Post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src` and *policy* is "`No`", return "`Allowed`".
3. Return the result of executing §6.6.1.2 Script directives post-request check on *request*, *response* and this directive.

§ *6.1.11.3. `script-src` Inline Check*

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Assert: *element* is not `null` or *type* is "`navigation`".
2. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
3. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src` and *policy* is "`No`", return "`Allowed`".
4. If the result of executing §6.6.3.3 Does element match source list for type and source? on *element*, this directive's value, *type*, and *source*, is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

§ 6.1.12. `script-src-elem`

The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "script-src-elem"
directive-value = serialized-source-list
```

The ***script-src-elem*** directive applies to all script requests and script blocks. Attributes that execute script (inline event handlers) are controlled via `script-src-attr`.

As such, the following differences exist when comparing to `script-src`:

- `script-src-elem` applies to inline checks whose `|type|` is "`script`" and "`navigation`" (and is ignored for inline checks whose `|type|` is "`script attribute`").
- `script-src-elem`'s value is not used for JavaScript execution sink checks that are gated on the

"unsafe-eval" check.

- `script-src-elem` is not used as a fallback for the `worker-src` directive. The `worker-src` checks still fall back on the `script-src` directive.

6.1.12.1. `script-src-elem` Pre-request check

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src-elem` and *policy* is "No", return "Allowed".
3. Return the result of executing §6.6.1.1 Script directives pre-request check on *request* and this directive.

6.1.12.2. `script-src-elem` Post-request check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src-elem` and *policy* is "No", return "Allowed".
3. Return the result of executing §6.6.1.2 Script directives post-request check on *request*, *response* and this directive.

6.1.12.3. `script-src-elem` Inline Check

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Assert: *element* is not `null` or *type* is "navigation".
2. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.

3. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src-elem`, and *policy* is "`No`", return "`Allowed`".
4. If the result of executing §6.6.3.3 Does element match source list for type and source? on *element*, this directive's value, *type*, and *source* is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

§ 6.1.13. `script-src-attr`

The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "script-src-attr"
directive-value = serialized-source-list
```

The ***script-src-attr*** directive applies to event handlers and, if present, it will override the `script-src` directive for relevant checks.

§ *6.1.13.1. `script-src-attr` Inline Check*

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Assert: *element* is not `null` or *type* is "`navigation`".
2. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
3. If the result of executing §6.7.4 Should fetch directive execute on *name*, `script-src-attr` and *policy* is "`No`", return "`Allowed`".
4. If the result of executing §6.6.3.3 Does element match source list for type and source? on *element*, this directive's value, *type*, and *source*, is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

§ 6.1.14. `style-src`

The ***style-src*** directive restricts the locations from which style may be applied to a `Document`. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "style-src"
directive-value = serialized-source-list
```

The `style-src` directive governs several things:

1. Style requests MUST pass through §4.1.3 Should request be blocked by Content Security Policy?. This includes:
    1. Stylesheet requests originating from a `<link>` element.
    2. Stylesheet requests originating from the '`@import`' rule.
    3. Stylesheet requests originating from a `Link` HTTP response header field [RFC8288].
2. Responses to style requests MUST pass through §4.1.4 Should response to request be blocked by Content Security Policy?.
3. Inline `<style>` blocks MUST pass through §4.2.4 Should element's inline type behavior be blocked by Content Security Policy?. The styles will be blocked unless every policy allows inline style, either implicitly by not specifying a `style-src` (or `default-src`) directive, or explicitly, by specifying "`unsafe-inline`", a nonce-source or a hash-source that matches the inline block.
4. The following CSS algorithms are gated on the `unsafe-eval` source expression:
    1. insert a CSS rule
    2. parse a CSS rule,
    3. parse a CSS declaration block
    4. parse a group of selectors

    This would include, for example, all invocations of CSSOM's various `cssText` setters and `insertRule` methods [CSSOM] [HTML].

    > ISSUE 9 This needs to be better explained. <https://github.com/w3c/webappsec-csp/issues/212>

§ *6.1.14.1. `style-src` Pre-request Check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src` and *policy* is "`No`", return "`Allowed`".

3. If the result of executing §6.6.2.2 Does nonce match source list? on *request*'s cryptographic nonce metadata and this directive's value is "`Matches`", return "`Allowed`".
4. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

§ *6.1.14.2. `style-src` Post-request Check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.2 Does nonce match source list? on *request*'s cryptographic nonce metadata and this directive's value is "`Matches`", return "`Allowed`".
4. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

§ *6.1.14.3. `style-src` Inline Check*

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.3.3 Does element match source list for type and source? on *element*, this directive's value, *type*, and *source*, is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

This directive's initialization algorithm is as follows:

> ISSUE 10 Do something interesting to the execution context in order to lock down interesting CSSOM algorithms. I don't think CSSOM gives us any hooks here, so let's work with them to put something reasonable together.

6.1.15. `style-src-elem`

The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "style-src-elem"
directive-value = serialized-source-list
```

The ***style-src-elem*** directive governs the behaviour of styles except for styles defined in inline attributes.

6.1.15.1. `style-src-elem` Pre-request Check

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src-elem` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.2 Does nonce match source list? on *request*'s cryptographic nonce metadata and this directive's value is "`Matches`", return "`Allowed`".
4. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

6.1.15.2. `style-src-elem` Post-request Check

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.

2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src-elem` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.2.2 Does nonce match source list? on *request*'s cryptographic nonce metadata and this directive's value is "`Matches`", return "`Allowed`".
4. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

6.1.15.3. `style-src-elem` Inline Check

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src-elem` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.3.3 Does element match source list for type and source? on *element*, this directive's value, *type*, and *source*, is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

6.1.16. `style-src-attr`

The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "style-src-attr"
directive-value = serialized-source-list
```

The ***style-src-attr*** directive governs the behaviour of style attributes.

6.1.16.1. `style-src-attr` Inline Check

This directive's inline check algorithm is as follows:

Given an `Element` (*element*), a string (*type*), a policy (*policy*) and a string (*source*):

1. Let *name* be the result of executing §6.7.2 Get the effective directive for inline checks on *type*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `style-src-attr` and *policy* is "`No`", return "`Allowed`".
3. If the result of executing §6.6.3.3 Does element match source list for type and source? on *element*, this directive's value, *type*, and *source*, is "`Does Not Match`", return "`Blocked`".
4. Return "`Allowed`".

§ 6.1.17. `worker-src`

The ***worker-src*** directive restricts the URLs which may be loaded as a `Worker`, `SharedWorker`, or `ServiceWorker`. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "worker-src"
directive-value = serialized-source-list
```

> EXAMPLE 16
> Given a page with the following Content Security Policy:
>
> ```
> Content-Security-Policy: worker-src https://example.com/
> ```
>
> Fetches for the following code will return a network errors, as the URL provided do not match `worker-src`'s source list:
>
> ```
> <script>
>   var blockedWorker = new Worker("data:application/javascript,...");
>   blockedWorker = new SharedWorker("https://example.org/");
>   navigator.serviceWorker.register('https://example.org/sw.js');
> </script>
> ```

§ *6.1.17.1. `worker-src` Pre-request Check*

This directive's pre-request check is as follows:

Given a request (*request*) and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `worker-src` and *policy*

is "No", return "Allowed".

3. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "Does Not Match", return "Blocked".
4. Return "Allowed".

§ *6.1.17.2. `worker-src` Post-request Check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Let *name* be the result of executing §6.7.1 Get the effective directive for request on *request*.
2. If the result of executing §6.7.4 Should fetch directive execute on *name*, `worker-src` and *policy* is "No", return "Allowed".
3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and this directive's value is "Does Not Match", return "Blocked".
4. Return "Allowed".

§ 6.2. Document Directives

The following directives govern the properties of a document or worker environment to which a policy applies.

§ 6.2.1. `base-uri`

The ***base-uri*** directive restricts the URLs which can be used in a Document's `<base>` element. The syntax for the directive's name and value is described by the following ABNF:

```
directive-name  = "base-uri"
directive-value = serialized-source-list
```

The following algorithm is called during HTML's set the frozen base url algorithm in order to monitor and enforce this directive:

§ *6.2.1.1. Is base allowed for document?*

Given a URL (*base*), and a Document (*document*), this algorithm returns "Allowed" if *base* may be used as the value of a <base> element's href attribute, and "Blocked" otherwise:

1. For each *policy* in *document*'s global object's csp list:
    1. Let *source list* be null.
    2. If a directive whose name is "base-uri" is present in *policy*'s directive set, set *source list* to that directive's value.
    3. If *source list* is null, skip to the next *policy*.
    4. If the result of executing §6.6.2.5 Does url match source list in origin with redirect count? on *base*, *source list*, *document*'s fallback base URL's origin, and 0 is "Does Not Match":
        1. Let *violation* be the result of executing §2.4.1 Create a violation object for global, policy, and directive on *document*'s global object, *policy*, and "base-uri".
        2. Set *violation*'s resource to "inline".
        3. Execute §5.3 Report a violation on *violation*.
        4. If *policy*'s disposition is "enforce", return "Blocked".

    > Note: We compare against the fallback base URL in order to deal correctly with things like an iframe srcdoc Document which has been sandboxed into an opaque origin.

2. Return "Allowed".

§ 6.2.2. plugin-types

The ***plugin-types*** directive restricts the set of plugins that can be embedded into a document by limiting the types of resources which can be loaded. The directive's syntax is described by the following ABNF grammar:

```
directive-name  = "plugin-types"
directive-value = media-type-list

media-type-list = media-type *( required-ascii-whitespace media-type )
media-type = type "/" subtype
; type and subtype are defined in RFC 2045
```

If a plugin-types directive is present, instantiation of an <embed> or <object> element will fail if any of the following conditions hold:

1. The element does not explicitly declare a valid MIME type via a `type` attribute.
2. The declared type does not match one of the items in the directive's value.
3. The fetched resource does not match the declared type.

EXAMPLE 17

Given a page with the following Content Security Policy:

```
Content-Security-Policy: plugin-types application/pdf
```

Fetches for the following code will all return network errors:

```
<!-- No 'type' declaration -->
<object data="https://example.com/flash"></object>

<!-- Non-matching 'type' declaration -->
<object data="https://example.com/flash" type="application/x-shockwave-flash">

<!-- Non-matching resource -->
<object data="https://example.com/flash" type="application/pdf"></object>
```

If the page allowed Flash content by sending the following header:

```
Content-Security-Policy: plugin-types application/x-shockwave-flash
```

Then the second item above would load successfully:

```
<!-- Matching 'type' declaration and resource -->
<object data="https://example.com/flash" type="application/x-shockwave-flash">
```

§ *6.2.2.1. `plugin-types` Post-Request Check*

This directive's post-request check algorithm is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s destination is either "`object`" or "`embed`":

1. Let *type* be the result of extracting a MIME type from *response*'s header list.
2. If *type* is not an ASCII case-insensitive match for any item in this directive's value, return "Blocked".

3. Return "Allowed".

6.2.2.2. Should plugin element be blocked a priori by Content Security Policy?:

Given an `Element` (*plugin element*), this algorithm returns "Blocked" or "Allowed" based on the element's `type` attribute and the policy applied to its document:

1. For each *policy* in *plugin element*'s node document's CSP list:
    1. If *policy* contains a directive (*directive*) whose name is `plugin-types`:
        1. Let *type* be "application/x-java-applet" if *plugin element* is an `applet` element, or *plugin element*'s `type` attribute's value if present, or "null" otherwise.
        2. Return "Blocked" if any of the following are true:
            1. *type* is `null`.
            2. *type* is not a valid MIME type.
            3. *type* is not an ASCII case-insensitive match for any item in *directive*'s value.
2. Return "Allowed".

§ 6.2.3. sandbox

The ***sandbox*** directive specifies an HTML sandbox policy which the user agent will apply to a resource, just as though it had been included in an `<iframe>` with a `sandbox` property.

The directive's syntax is described by the following ABNF grammar, with the additional requirement that each token value MUST be one of the keywords defined by HTML specification as allowed values for the `<iframe>` `sandbox` attribute [HTML].

```
directive-name  = "sandbox"
directive-value = "" / token *( required-ascii-whitespace token )
```

This directive has no reporting requirements; it will be ignored entirely when delivered in a `Content-Security-Policy-Report-Only` header, or within a `<meta>` element.

§ *6.2.3.1. `sandbox` Response Check*

This directive's response check algorithm is as follows:

Given a request (*request*), a response (*response*), and a policy (*policy*):

1. Assert: *response* is unused.
2. If *policy*'s disposition is not "`enforce`", then return "`Allowed`".
3. If *request*'s destination is one of "`serviceworker`", "`sharedworker`", or "`worker`":
    1. If the result of the Parse a sandboxing directive algorithm using this directive's value as the input contains either the sandboxed scripts browsing context flag or the sandboxed origin browsing context flag flags, return "`Blocked`".

    > Note: This will need to change if we allow Workers to be sandboxed into unique origins, which seems like a pretty reasonable thing to do.

4. Return "`Allowed`".

§ *6.2.3.2. `sandbox` Initialization*

This directive's initialization algorithm is responsible for adjusting a `Document`'s forced sandboxing flag set according to the `sandbox` values present in its policies, as follows:

Given a `Document` or global object (*context*), a response (*response*), and a policy (*policy*):

1. Assert: *response* is unused.
2. If *policy*'s disposition is not "`enforce`", or *context* is not a `Document`, then abort this algorithm.

    > Note: This will need to change if we allow Workers to be sandboxed, which seems like a pretty reasonable thing to do.

3. Parse a sandboxing directive using this directive's value as the input, and *context*'s forced sandboxing flag set as the output.

§ 6.3. Navigation Directives

§ **6.3.1. `form-action`**

The ***form-action*** directive restricts the URLs which can be used as the target of a form submissions from a given context. The directive's syntax is described by the following ABNF grammar:

```
directive-name  = "form-action"
directive-value = serialized-source-list
```

§ 6.3.1.1. `form-action` Pre-Navigation Check

Given a request (*request*), a string *navigation type* ("`form-submission`" or "`other`"), two browsing contexts (*source* and *target*), and a policy (*policy*) this algorithm returns "`Blocked`" if a form submission violates the `form-action` directive's constraints, and "`Allowed`" otherwise. This constitutes the `form-action` directive's pre-navigation check:

1. Assert: *source*, *target*, and *policy* are unused in this algorithm, as `form-action` is concerned only with details of the outgoing request.
2. If *navigation type* is "`form-submission`":
    1. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
3. Return "`Allowed`".

§ 6.3.2. `frame-ancestors`

The ***frame-ancestors*** directive restricts the URLs which can embed the resource using `<frame>`, `<iframe>`, `<object>`, `<embed>`, or `applet` element. Resources can use this directive to avoid many UI Redressing [UISECURITY] attacks, by avoiding the risk of being embedded into potentially hostile contexts.

The directive's syntax is described by the following ABNF grammar:

```
directive-name  = "frame-ancestors"
directive-value = ancestor-source-list

ancestor-source-list = ( ancestor-source *( required-ascii-whitespace ancestor-sour
ancestor-source      = scheme-source / host-source / "'self'"
```

The `frame-ancestors` directive MUST be ignored when contained in a policy declared via a `<meta>` element.

> Note: The `frame-ancestors` directive's syntax is similar to a source list, but `frame-ancestors` will not fall back to the `default-src` directive's value if one is specified. That is, a policy that declares `default-src 'none'` will still allow the resource to be embedded by anyone.

§ *6.3.2.1. `frame-ancestors` Navigation Response Check*

Given a request (*request*), a string *navigation type* ("`form-submission`" or "`other`"), a response (*navigation response*) two browsing contexts (*source* and *target*), a string *check type* ("`source`" or "`response`"), and a policy (*policy*) this algorithm returns "`Blocked`" if one or more of the ancestors of *target* violate the `frame-ancestors` directive delivered with the response, and "`Allowed`" otherwise. This constitutes the `frame-ancestors` directive's navigation response check:

1. Assert: *request*, *navigation response*, *navigation type*, *source*, and *policy* are unused in this algorithm, as `frame-ancestors` is concerned only with *navigation response*'s frame-ancestors directive.
2. If *check type* is "`source`", return "`Allowed`".

   > Note: The 'frame-ancestors' directive is relevant only to the *target* browsing context and it has no impact on the *source* browsing context.

3. If *target* is not a nested browsing context, return "`Allowed`".
4. Let *current* be *target*.
5. While *current* has a parent browsing context (*parent*):

   1. Set *current* to *parent*.
   2. Let *origin* be the result of executing the URL parser on the ASCII serialization of *parent*'s active document's relevant settings object's origin.
   3. If §6.6.2.5 Does url match source list in origin with redirect count? returns `Does Not Match` when executed upon *origin*, this directive's value, *navigation response*'s url's origin, and `0`, return "`Blocked`".

6. Return "`Allowed`".

§ *6.3.2.2. Relation to `X-Frame-Options`*

This directive is similar to the `X-Frame-Options` header that several user agents have implemented.

The 'none' source expression is roughly equivalent to that header's DENY, 'self' to SAMEORIGIN, and so on. The major difference is that many user agents implement SAMEORIGIN such that it only matches against the top-level document's location, while the frame-ancestors directive checks against each ancestor. If _any_ ancestor doesn't match, the load is cancelled. [RFC7034]

In order to allow backwards-compatible deployment, the frame-ancestors directive _obsoletes_ the X-Frame-Options header. If a resource is delivered with an policy that includes a directive named frame-ancestors and whose disposition is "enforce", then the X-Frame-Options header MUST be ignored.

> ISSUE 11 Spell this out in more detail as part of defining X-Frame-Options integration with the process a navigate response algorithm. <https://github.com/whatwg/html/issues/1230>

§ 6.3.3. navigate-to

The ***navigate-to*** directive restricts the URLs to which a document can initiate navigations by any means (<a>, <form>, window.location, window.open, etc.). This is an enforcement on what navigations this document initiates not on what this document is allowed to navigate to. If the form-action directive is present, the navigate-to directive will not act on navigations that are form submissions.

> EXAMPLE 18
> A document *initiator* has the following Content-Security-Policy:
>
> ```
> Content-Security-Policy: navigate-to example.com
> ```
>
> A document *target* has the following Content-Security-Policy:
>
> ```
> Content-Security-Policy: navigate-to not-example.com
> ```
>
> If the *initiator* attempts to navigate the *target* to example.com, the navigation is allowed by the navigate-to directive.
>
> If the *initiator* attempts to navigate the *target* to not-example.com, the navigation is blocked by the navigate-to directive.

The directive's syntax is described by the following ABNF grammar:

```
directive-name  = "navigate-to"
directive-value = serialized-source-list
```

§ 6.3.3.1. `navigate-to` Pre-Navigation Check

Given a request (*request*), a string *navigation type* ("`form-submission`" or "`other`"), two browsing contexts (*source* and *target*), and a policy (*policy*), this algorithm returns "`Blocked`" if the navigation violates the `navigate-to` directive's constraints, and "`Allowed`" otherwise. This constitutes the `navigate-to`' directive's pre-navigation check:

1. Assert: *source* and *target* are unused as 'navigate-to' is concerned with the details of the request.
2. If *navigation type* is "`form-submission`" and *policy* contains a directive named "`form-action`", return "`Allowed`".
3. If this directive's value contains a source expression that is an ASCII case-insensitive match for the "`'unsafe-allow-redirects'`" keyword-source, return "`Allowed`".

   > Note: If the 'unsafe-allow-redirects' flag is present we have to wait for the response and take into account the response's status in §6.3.3.2 navigate-to Navigation Response Check.

4. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
5. Return "`Allowed`".

§ 6.3.3.2. `navigate-to` Navigation Response Check

Given a request (*request*), a string *navigation type* ("`form-submission`" or "`other`"), a response (*navigation response*) two browsing contexts (*source* and *target*), a string *check type* ("`source`" or "`response`"), and a policy (*policy*), this algorithm returns "`Blocked`" if the navigation violates the `navigate-to` directive's constraints, and "`Allowed`" otherwise. This constitutes the `navigate-to` directive's navigation response check:

1. Assert: *source*, and *target* are unused.
2. If *check type* is "`response`", return "`Allowed`".

   > Note: The 'navigate-to' directive is relevant only to the *source* browsing context and it has no impact on the *target* browsing context.

3. If *navigation type* is "`form-submission`" and *policy* contains a directive named "`form-action`", return "`Allowed`".
4. If this directive's value does not contain a source expression that is an ASCII case-insensitive match for the "`'unsafe-allow-redirects'`" keyword-source, return "`Allowed`".

> Note: If the 'unsafe-allow-redirects' flag is not present we have already checked the navigation in §6.3.3.1 navigate-to Pre-Navigation Check.

5. If *navigation response*'s status is a redirect status, return "`Allowed`".
6. If the result of executing §6.6.2.3 Does request match source list? on *request* and this directive's value is "`Does Not Match`", return "`Blocked`".
7. Return "`Allowed`".

§ 6.4. Reporting Directives

Various algorithms in this document hook into the reporting process by constructing a violation object via §2.4.2 Create a violation object for request, and policy. or §2.4.1 Create a violation object for global, policy, and directive, and passing that object to §5.3 Report a violation to deliver the report.

§ 6.4.1. `report-uri`

> Note: The `report-uri` directive is deprecated. Please use the `report-to` directive instead. If the latter directive is present, this directive will be ignored. To ensure backwards compatibility, we suggest specifying both, like this:
>
> EXAMPLE 19
>
> `Content-Security-Policy: ...; report-uri https://endpoint.com; report-to grou`

The ***`report-uri`*** directive defines a set of endpoints to which violation reports will be sent when particular behaviors are prevented.

```
directive-name  = "report-uri"
directive-value = uri-reference *( required-ascii-whitespace uri-reference )

; The uri-reference grammar is defined in Section 4.1 of RFC 3986.
```

The directive has no effect in and of itself, but only gains meaning in combination with other directives.

6.4.2. `report-to`

The ***`report-to`*** directive defines a reporting group to which violation reports ought to be sent [REPORTING]. The directive's behavior is defined in §5.3 Report a violation. The directive's name and value are described by the following ABNF:

```
directive-name  = "report-to"
directive-value = token
```

6.5. Directives Defined in Other Documents

This document defines a core set of directives, and sets up a framework for modular extension by other specifications. At the time this document was produced, the following stable documents extend CSP:

- [MIX] defines `block-all-mixed-content`
- [UPGRADE-INSECURE-REQUESTS] defines `upgrade-insecure-requests`
- [SRI] defines `require-sri-for`

Extensions to CSP MUST register themselves via the process outlined in [RFC7762]. In particular, note the criteria discussed in Section 4.2 of that document.

New directives SHOULD use the pre-request check, post-request check, response check, and initialization hooks in order to integrate themselves into Fetch and HTML.

6.6. Matching Algorithms

6.6.1. Script directive checks

6.6.1.1. Script directives pre-request check

Given a request (*request*) and a directive (*directive*):

1. If *request*'s destination is script-like:
    1. If the result of executing §6.6.2.2 Does nonce match source list? on *request*'s cryptographic nonce metadata and this directive's value is `"Matches"`, return `"Allowed"`.
    2. Let *integrity expressions* be the set of source expressions in *directive*'s value that match the hash-source grammar.

3. If *integrity expressions* is not empty:

    1. Let *integrity sources* be the result of executing the algorithm defined in Subresource Integrity §parse-metadata on *request*'s integrity metadata. [SRI]
    2. If *integrity sources* is "`no metadata`" or an empty set, skip the remaining substeps.
    3. Let *bypass due to integrity match* be `true`.
    4. For each *source* in *integrity sources*:

        1. If *directive*'s value does not contain a source expression whose hash-algorithm is a case-sensitive match for *source*'s `hash-algo` component, and whose base64-value is a case-sensitive match for *source*'s `base64-value`, then set *bypass due to integrity match* to `false`.

    5. If *bypass due to integrity match* is `true`, return "`Allowed`".

    > Note: Here, we verify only that the *request* contains a set of integrity metadata which is a subset of the hash-source source expressions specified by *directive*. We rely on the browser's enforcement of Subresource Integrity [SRI] to block non-matching resources upon response.

4. If *directive*'s value contains a source expression that is an ASCII case-insensitive match for the "`'strict-dynamic'`" keyword-source:

    1. If the *request*'s parser metadata is "parser-inserted", return "`Blocked`".

        Otherwise, return "`Allowed`".

        > Note: "`'strict-dynamic'`" is explained in more detail in §8.2 Usage of "'strict-dynamic'".

5. If the result of executing §6.6.2.3 Does request match source list? on *request* and *directive*'s value is "`Does Not Match`", return "`Blocked`".

2. Return "`Allowed`".

§ *6.6.1.2. Script directives post-request check*

This directive's post-request check is as follows:

Given a request (*request*), a response (*response*), and a directive (*directive*):

1. If *request*'s destination is script-like:
    1. If the result of executing §6.6.2.2 Does nonce match source list? on *request*'s cryptographic nonce metadata and this directive's value is "`Matches`", return "`Allowed`".
    2. If *directive*'s value contains "`'strict-dynamic'`", and *request*'s parser metadata is not "parser-inserted", return "`Allowed`".
    3. If the result of executing §6.6.2.4 Does response to request match source list? on *response*, *request*, and *directive*'s value is "`Does Not Match`", return "`Blocked`".
2. Return "`Allowed`".

§ 6.6.2. URL Matching

§ *6.6.2.1. Does request violate policy?*

Given a request (*request*) and a policy (*policy*), this algorithm returns the violated directive if the request violates the policy, and "`Does Not Violate`" otherwise.

1. Let *violates* be "`Does Not Violate`".
2. For each *directive* in *policy*:
    1. Let *result* be the result of executing *directive*'s pre-request check on *request* and *policy*.
    2. If *result* is "`Blocked`", then let *violates* be *directive*.
3. Return *violates*.

§ *6.6.2.2. Does nonce match source list?*

Given a request's cryptographic nonce metadata (*nonce*) and a source list (*source list*), this algorithm returns "`Matches`" if the nonce matches one or more source expressions in the list, and "`Does Not Match`" otherwise:

1. Assert: *source list* is not `null`.
2. If *nonce* is the empty string, return "`Does Not Match`".
3. For each *expression* in *source list*:
    1. If *expression* matches the `nonce-source` grammar, and *nonce* is a case-sensitive match for *expression*'s `base64-value` part, return "`Matches`".

4. Return "`Does Not Match`".

§ *6.6.2.3. Does request match source list?*

Given a request (*request*), and a source list (*source list*), this algorithm returns the result of executing §6.6.2.5 Does url match source list in origin with redirect count? on *request*'s current url, *source list*, *request*'s origin, and *request*'s redirect count.

> Note: This is generally used in directives' pre-request check algorithms to verify that a given request is reasonable.

§ *6.6.2.4. Does response to request match source list?*

Given a request (*request*), and a source list (*source list*), this algorithm returns the result of executing §6.6.2.5 Does url match source list in origin with redirect count? on *response*'s url, *source list*, *request*'s origin, and *request*'s redirect count.

> Note: This is generally used in directives' post-request check algorithms to verify that a given response is reasonable.

§ *6.6.2.5. Does url match source list in origin with redirect count?*

Given a `URL` (*url*), a source list (*source list*), an origin (*origin*), and a number (*redirect count*), this algorithm returns "`Matches`" if the URL matches one or more source expressions in *source list*, or "`Does Not Match`" otherwise:

1. Assert: *source list* is not `null`.
2. If *source list* is an empty list, return "`Does Not Match`".
3. If *source list* contains a single item which is an ASCII case-insensitive match for the string "`'none'`", return "`Does Not Match`".

   > Note: An empty source list (that is, a directive without a value: `script-src`, as opposed to `script-src host1`) is equivalent to a source list containing `'none'`, and will not match any URL.

4. For each *expression* in *source list*:

    1. If §6.6.2.6 Does url match expression in origin with redirect count? returns "`Matches`" when executed upon *url*, *expression*, *origin*, and *redirect count*, return "`Matches`".

5. Return "`Does Not Match`".

§ *6.6.2.6. Does url match expression in origin with redirect count?*

Given a `URL` (*url*), a source expression (*expression*), an origin (*origin*), and a number (*redirect count*), this algorithm returns "`Matches`" if *url* matches *expression*, and "`Does Not Match`" otherwise.

> Note: *origin* is the origin of the resource relative to which the *expression* should be resolved. "`'self'`", for instance, will have distinct meaning depending on that bit of context.

1. If *expression* is the string "*", return "`Matches`" if one or more of the following conditions is met:

    1. *url*'s scheme is a network scheme.
    2. *url*'s scheme is the same as *origin*'s scheme.

    > Note: This logic means that in order to allow a resource from a non-network scheme, it has to be either explicitly specified (e.g. `default-src * data: custom-scheme-1: custom-scheme-2:`), or the protected resource must be loaded from the same scheme.

2. If *expression* matches the `scheme-source` or `host-source` grammar:

    1. If *expression* has a `scheme-part`, and it does not `scheme-part` match *url*'s scheme, return "`Does Not Match`".
    2. If *expression* matches the `scheme-source` grammar, return "`Matches`".

3. If *expression* matches the `host-source` grammar:

    1. If *url*'s `host` is `null`, return "`Does Not Match`".
    2. If *expression* does not have a `scheme-part`, and *origin*'s scheme does not `scheme-part` match *url*'s scheme, return "`Does Not Match`".

        > Note: As with `scheme-part` above, we allow schemeless `host-source` expressions to be upgraded from insecure schemes to secure schemes.

    3. If *expression*'s `host-part` does not `host-part` match *url*'s `host`, return "`Does Not`

`Match`".

4. Let *port-part* be *expression*'s `port-part` if present, and `null` otherwise.
5. If *port-part* does not `port-part` match *url*'s port and *url*'s scheme, return "`Does Not Match`".
6. If *expression* contains a non-empty `path-part`, and *redirect count* is 0, then:
    1. Let *path* be the resulting of joining *url*'s path on the U+002F SOLIDUS character (`/`).
    2. If *expression*'s `path-part` does not `path-part` match *path*, return "`Does Not Match`".
7. Return "`Matches`".

4. If *expression* is an ASCII case-insensitive match for "`'self'`", return "`Matches`" if one or more of the following conditions is met:
    1. *origin* is the same as *url*'s origin
    2. *origin*'s `host` is the same as *url*'s `host`, *origin*'s `port` and *url*'s `port` are either the same or the default ports for their respective schemes, and one or more of the following conditions is met:
        1. *url*'s scheme is "`https`" or "`wss`"
        2. *origin*'s scheme is "`http`" and *url*'s scheme is "`http`" or "`ws`"

> Note: Like the `scheme-part` logic above, the "`'self'`" matching algorithm allows upgrades to secure schemes when it is safe to do so. We limit these upgrades to endpoints running on the default port for a particular scheme or a port that matches the origin of the protected resource, as this seems sufficient to deal with upgrades that can be reasonably expected to succeed.

5. Return "`Does Not Match`".

§ *6.6.2.7. `scheme-part` matching*

An ASCII string ***`scheme-part` matches*** another ASCII string if a CSP source expression that contained the first as a `scheme-part` could potentially match a URL containing the latter as a scheme. For example, we say that "http" `scheme-part` matches "https".

> Note: The matching relation is asymmetric. For example, the source expressions `https:` and `https://example.com/` do not match the URL `http://example.com/`. We always allow a secure upgrade from an explicitly insecure expression. `script-src http:` is treated as equivalent to `script-src http: https:`, `script-src http://example.com` to `script-src http://example.com https://example.com`, and `connect-src ws:` to `connect-src ws: wss:`.

More formally, two ASCII strings (*A* and *B*) are said to `scheme-part` match if the following algorithm returns "`Matches`":

1. If one of the following is true, return "`Matches`":
    1. *A* is an ASCII case-insensitive match for *B*.
    2. *A* is an ASCII case-insensitive match for "`http`", and *B* is an ASCII case-insensitive match for "`https`".
    3. *A* is an ASCII case-insensitive match for "`ws`", and *B* is an ASCII case-insensitive match for "`wss`", "`http`", or "`https`".
    4. *A* is an ASCII case-insensitive match for "`wss`", and *B* is an ASCII case-insensitive match for "`https`".
2. Return "`Does Not Match`".

§ *6.6.2.8. `host-part` matching*

An ASCII string ***`host-part` matches*** another ASCII string if a CSP source expression that contained the first as a `host-part` could potentially match a URL containing the latter as a host. For example, we say that "www.example.com" host-part matches "www.example.com".

More formally, two ASCII strings (*A* and *B*) are said to `host-part` match if the following algorithm returns "`Matches`":

> Note: The matching relation is asymmetric. That is, *A* matching *B* does not mean that *B* will match *A*. For example, `*.example.com` `host-part` matches `www.example.com`, but `www.example.com` does not `host-part` match `*.example.com`.

1. If the first character of *A* is an U+002A ASTERISK character (*):
    1. Let *remaining* be the result of removing the leading ("*") from *A*.

> 2. If *remaining* (including the leading U+002E FULL STOP character (`.`)) is an ASCII case-insensitive match for the rightmost characters of *B*, then return "`Matches`". Otherwise, return "`Does Not Match`".

2. If *A* is not an ASCII case-insensitive match for *B*, return "`Does Not Match`".
3. If *A* matches the IPv4address rule from [RFC3986], and is not "`127.0.0.1`"; or if *A* is an IPv6 address, return "`Does Not Match`".

> Note: A future version of this specification may allow literal IPv6 and IPv4 addresses, depending on usage and demand. Given the weak security properties of IP addresses in relation to named hosts, however, authors are encouraged to prefer the latter whenever possible.

4. Return "`Matches`".

§ *6.6.2.9. `port-part` matching*

An ASCII string (*port A*) ***`port-part` matches*** two other ASCII strings (*port B* and *scheme B*) if a CSP source expression that contained the first as a `port-part` could potentially match a URL containing the latter as port and scheme. For example, "80" `port-part` matches matches "80"/"http".

1. If *port A* is empty:

> 1. If *port B* is the default port for *scheme B*, return "`Matches`". Otherwise, return "`Does Not Match`".

2. If *port A* is equal to "*", return "`Matches`".
3. If *port A* is a case-sensitive match for *port B*, return "`Matches`".
4. If *port B* is empty:

> 1. If *port A* is the default port for *scheme B*, return "`Matches`". Otherwise, return "`Does not Match`".

5. Return "`Does Not Match`".

§ *6.6.2.10. `path-part` matching*

An ASCII string (*path A*) ***`path-part` matches*** another ASCII string (*path B*) if a CSP source expression that contained the first as a `path-part` could potentially match a URL containing the latter

as a path. For example, we say that "/subdirectory/" `path-part` matches "/subdirectory/file".

> Note: The matching relation is asymmetric. That is, *path A* matching *path B* does not mean that *path B* will match *path A*.

1. If *path A* is empty, return "`Matches`".
2. If *path A* consists of one character that is equal to the U+002F SOLIDUS character (`/`) and *path B* is empty, return "`Matches`".
3. Let *exact match* be `false` if the final character of *path A* is the U+002F SOLIDUS character (`/`), and `true` otherwise.
4. Let *path list A* and *path list B* be the result of strictly splitting *path A* and *path B* respectively on the U+002F SOLIDUS character (`/`).
5. If *path list A* has more items than *path list B*, return "`Does Not Match`".
6. If *exact match* is `true`, and *path list A* does not have the same number of items as *path list B*, return "`Does Not Match`".
7. If *exact match* is `false`:
    1. Assert: the final item in *path list A* is the empty string.
    2. Remove the final item from *path list A*.
8. For each *piece A* in *path list A*:
    1. Let *piece B* be the next item in *path list B*.
    2. Percent decode *piece A*.
    3. Percent decode *piece B*.
    4. If *piece A* is not a case-sensitive match for *piece B*, return "`Does Not Match`".
9. Return "`Matches`".

§ 6.6.3. Element Matching Algorithms

§ *6.6.3.1. Is element nonceable?*

Given an `Element` (*element*), this algorithm returns "`Nonceable`" if a `nonce-source` expression can match the element (as discussed in §7.2 Nonce Stealing), and "`Not Nonceable`" if such expressions should not be applied.

1. If *element* does not have an attribute named "`nonce`", return "`Not Nonceable`".
2. If *element* is a `<script>` element, then for each *attribute* in *element*:
    1. If *attribute*'s name is an ASCII case-insensitive match for the string "`<script`" or the string "`<style`", return "`Not Nonceable`".
    2. If *attribute*'s value contains an ASCII case-insensitive match the string "`<script`" or the string "`<style`", return "`Not Nonceable`".
3. If *element* had a duplicate-attribute parse error during tokenization, return "`Not Nonceable`".

> ISSUE 12 We need some sort of hook in HTML to record this error if we're planning on using it here. <https://github.com/whatwg/html/issues/3257>

4. Return "`Nonceable`".

> ISSUE 13 This processing is meant to mitigate the risk of dangling markup attacks that steal the nonce from an existing element in order to load injected script. It is fairly expensive, however, as it requires that we walk through all attributes and their values in order to determine whether the script should execute. Here, we try to minimize the impact by doing this check only for `<script>` elements when a nonce is present, but we should probably consider this algorithm as "at risk" until we know its impact. <https://github.com/w3c/webappsec-csp/issues/98>

§ *6.6.3.2. Does a source list allow all inline behavior for type?*

A source list ***allows all inline behavior*** of a given *type* if it contains the `keyword-source` expression `'unsafe-inline'`, and does not override that expression as described in the following algorithm:

Given a source list (*list*) and a string (*type*), the following algorithm returns "`Allows`" if all inline content of a given *type* is allowed and "`Does Not Allow`" otherwise.

1. Let *allow all inline* be `false`.
2. For each *expression* in *list*:
    1. If *expression* matches the `nonce-source` or `hash-source` grammar, return "`Does Not Allow`".
    2. If *type* is "`script`", "`script attribute`" or "`navigation`" and *expression* matches the keyword-source "`'strict-dynamic'`", return "`Does Not Allow`".

> Note: `'strict-dynamic'` only applies to scripts, not other resource types. Usage is explained in more detail in §8.2 Usage of "'strict-dynamic'".

3. If *expression* is an ASCII case-insensitive match for the `keyword-source` "`'unsafe-inline'`", set *allow all inline* to `true`.

3. If *allow all inline* is `true`, return "`Allows`". Otherwise, return "`Does Not Allow`".

> EXAMPLE 20
> Source lists that allow all inline behavior:
>
> ```
> 'unsafe-inline' http://a.com http://b.com
> 'unsafe-inline'
> ```
>
> Source lists that do not allow all inline behavior due to the presence of nonces and/or hashes, or absence of 'unsafe-inline':
>
> ```
> 'sha512-321cba' 'nonce-abc'
> http://example.com 'unsafe-inline' 'nonce-abc'
> ```
>
> Source lists that do not allow all inline behavior when *type* is 'script' or 'script attribute' due to the presence of 'strict-dynamic', but allow all inline behavior otherwise:
>
> ```
> 'unsafe-inline' 'strict-dynamic'
> http://example.com 'strict-dynamic' 'unsafe-inline'
> ```

§ *6.6.3.3. Does element match source list for type and source?*

Given an `Element` (*element*), a source list (*list*), a string (*type*), and a string (*source*), this algorithm returns "`Matches`" or "`Does Not Match`".

> Note: Regardless of the encoding of the document, *source* will be converted to `UTF-8` before applying any hashing algorithms.

1. If §6.6.3.2 Does a source list allow all inline behavior for type? returns "`Allows`" given *list* and *type*, return "`Matches`".
2. If *type* is "`script`" or "`style`", and §6.6.3.1 Is element nonceable? returns "`Nonceable`" when executed upon *element*:
    1. For each *expression* in *list*:

1. If *expression* matches the `nonce-source` grammar, and *element* has a `nonce` attribute whose value is a case-sensitive match for *expression*'s `base64-value` part, return "`Matches`".

> Note: Nonces only apply to inline `<script>` and inline `<style>`, not to attributes of either element or to `javascript:` navigations.

3. Let *unsafe-hashes flag* be `false`.
4. For each *expression* in *list*:
    1. If *expression* is an ASCII case-insensitive match for the `keyword-source` "`'unsafe-hashes'`", set *unsafe-hashes flag* to `true`. Break out of the loop.
5. If *type* is "`script`" or "`style`", or *unsafe-hashes flag* is `true`:
    1. Set *source* to the result of executing UTF-8 encode on the result of executing JavaScript string converting on *source*.
    2. For each *expression* in *list*:
        1. If *expression* matches the `hash-source` grammar:
            1. Let *algorithm* be `null`.
            2. If *expression*'s `hash-algorithm` part is an ASCII case-insensitive match for "sha256", set *algorithm* to SHA-256.
            3. If *expression*'s `hash-algorithm` part is an ASCII case-insensitive match for "sha384", set *algorithm* to SHA-384.
            4. If *expression*'s `hash-algorithm` part is an ASCII case-insensitive match for "sha512", set *algorithm* to SHA-512.
            5. If *algorithm* is not `null`:
                1. Let *actual* be the result of base64 encoding the result of applying *algorithm* to *source*.
                2. Let *expected* be *expression*'s `base64-value` part, with all '-' characters replaced with '+', and all '_' characters replaced with '/'.

                > Note: This replacement normalizes hashes expressed in base64url encoding into base64 encoding for matching.

                3. If *actual* is a case-sensitive match for *expected*, return "`Matches`".

> Note: Hashes apply to inline `<script>` and inline `<style>`. If the "`'unsafe-hashes'`" source expression is present, they will also apply to event handlers, style attributes and `javascript:` navigations.

6. Return "`Does Not Match`".

§ 6.7. Directive Algorithms

§ 6.7.1. Get the effective directive for *request*

Each fetch directive controls a specific destination of request. Given a request (*request*), the following algorithm returns either `null` or the name of the request's ***effective directive***:

1. If *request*'s initiator is "`fetch`" or its destination is "", return `connect-src`.
2. If *request*'s initiator is "`prefetch`" or "`prerender`", return `prefetch-src`.
3. Switch on *request*'s destination, and execute the associated steps:

"manifest"

1. Return `manifest-src`.

"object"
"embed"

1. Return `object-src`.

"document"

1. If the *request*'s target browsing context is a nested browsing context, return `frame-src`.

"audio"
"track"
"video"

1. Return `media-src`.

"font"

1. Return `font-src`.

"image"

1. Return `img-src`.

"style"

1. Return `style-src-elem`.

"script"
"xslt"

1. Return `script-src-elem`.

"serviceworker"
"sharedworker"
"worker"

1. Return `worker-src`.

4. Return `null`.

§ 6.7.2. Get the effective directive for inline checks

Given a string (*type*), this algorithm returns the name of the effective directive.

> Note: While the effective directive is only defined for requests, in this algorithm it is used similarly to mean the directive that is most relevant to a particular type of inline check.

1. Switch on *type*:

"script"
"navigation"

1. Return `script-src-elem`.

"script attribute"

1. Return `script-src-attr`.

"style"

1. Return `style-src-elem`.

"style attribute"

1. Return `style-src-attr`.

2. Return `null`.

§ 6.7.3. Get fetch directive fallback list

Will return an ordered set of the fallback directives for a specific directive. The returned ordered set is sorted from most relevant to least relevant and it includes the effective directive itself.

Given a string (*directive name*):

1. Switch on *directive name*:

"script-src-elem"

1. Return `<< "script-src-elem", "script-src", "default-src" >>`.

"script-src-attr"

1. Return << "script-src-attr", "script-src", "default-src" >>.

"style-src-elem"

1. Return << "style-src-elem", "style-src", "default-src" >>.

"style-src-attr"

1. Return << "style-src-attr", "style-src", "default-src" >>.

"worker-src"

1. Return << "worker-src", "child-src", "script-src", "default-src" >>.

"connect-src"

1. Return << "connect-src", "default-src" >>.

"manifest-src"

1. Return << "manifest-src", "default-src" >>.

"prefetch-src"

1. Return << "prefetch-src", "default-src" >>.

"object-src"

1. Return << "object-src", "default-src" >>.

"frame-src"

1. Return << "frame-src", "child-src", "default-src" >>.

"media-src"

1. Return << "media-src", "default-src" >>.

"font-src"

1. Return << "font-src", "default-src" >>.

"image-src"

1. Return << "image-src", "default-src" >>.

2. Return << >>.

§ 6.7.4. Should fetch directive execute

This algorithm is used for fetch directives to decide whether a directive should execute or defer to a different directive that is better suited. For example: if the *effective directive name* is `worker-src` (meaning that we are currently checking a worker request), a `default-src` directive should not execute if a `worker-src` or `script-src` directive exists.

Given a string (*effective directive name*), a string (*directive name*) and a policy (*policy*):

1. Let *directive fallback list* be the result of executing §6.7.3 Get fetch directive fallback list on

effective directive name.

2. For each *fallback directive* in *directive fallback list*:

    1. If *directive name* is *fallback directive*, Return "`Yes`".

    2. If *policy* contains a directive whose name is *fallback directive*, Return "`No`".

3. Return "`No`".

§ 7. Security and Privacy Considerations

§ 7.1. Nonce Reuse

Nonces override the other restrictions present in the directive in which they're delivered. It is critical, then, that they remain unguessable, as bypassing a resource's policy is otherwise trivial.

If a server delivers a nonce-source expression as part of a policy, the server MUST generate a unique value each time it transmits a policy. The generated value SHOULD be at least 128 bits long (before encoding), and SHOULD be generated via a cryptographically secure random number generator in order to ensure that the value is difficult for an attacker to predict.

> Note: Using a nonce to allow inline script or style is less secure than not using a nonce, as nonces override the restrictions in the directive in which they are present. An attacker who can gain access to the nonce can execute whatever script they like, whenever they like. That said, nonces provide a substantial improvement over 'unsafe-inline' when layering a content security policy on top of old code. When considering 'unsafe-inline', authors are encouraged to consider nonces (or hashes) instead.

§ 7.2. Nonce Stealing

Dangling markup attacks such as those discussed in [FILEDESCRIPTOR-2015] can be used to repurpose a page's legitimate nonces for injections. For example, given an injection point before a `<script>` element:

```
<p>Hello, [INJECTION POINT]</p>
<script nonce=abc src=/good.js></script>
```

If an attacker injects the string "`<script src='https://evil.com/evil.js' `", then the browser

will receive the following:

```
<p>Hello, <script src='https://evil.com/evil.js' </p>
<script nonce=abc src=/good.js></script>
```

It will then parse that code, ending up with a `<script>` element with a `src` attribute pointing to a malicious payload, an attribute named `</p>`, an attribute named `"<script"`, a `nonce` attribute, and a second `src` attribute which is helpfully discarded as duplicate by the parser.

The §6.6.3.1 Is element nonceable? algorithm attempts to mitigate this specific attack by walking through `<script>` or `<style>` element attributes, looking for the string `"<script"` or `"<style"` in their names or values.

§ 7.3. Nonce Retargeting

Nonces bypass host-source expressions, enabling developers to load code from any origin. This, generally, is fine, and desirable from the developer's perspective. However, if an attacker can inject a `<base>` element, then an otherwise safe page can be subverted when relative URLs are resolved. That is, on `https://example.com/` the following code will load `https://example.com/good.js`:

```
<script nonce=abc src=/good.js></script>
```

However, the following will load `https://evil.com/good.js`:

```
<base href="https://evil.com">
<script nonce=abc src=/good.js></script>
```

To mitigate this risk, it is advisable to set an explicit `<base>` element on every page, or to limit the ability of an attacker to inject their own `<base>` element by setting a `base-uri` directive in your page's policy. For example, `base-uri 'none'`.

§ 7.4. CSS Parsing

The style-src directive restricts the locations from which the protected resource can load styles. However, if the user agent uses a lax CSS parsing algorithm, an attacker might be able to trick the user agent into accepting malicious "stylesheets" hosted by an otherwise trustworthy origin.

These attacks are similar to the CSS cross-origin data leakage attack described by Chris Evans in 2009 [CSS-ABUSE]. User agents SHOULD defend against both attacks using the same mechanism: stricter CSS parsing rules for style sheets with improper MIME types.

§ 7.5. Violation Reports

The violation reporting mechanism in this document has been designed to mitigate the risk that a malicious web site could use violation reports to probe the behavior of other servers. For example, consider a malicious web site that allows `https://example.com` as a source of images. If the malicious site attempts to load `https://example.com/login` as an image, and the `example.com` server redirects to an identity provider (e.g. `identityprovider.example.net`), CSP will block the request. If violation reports contained the full blocked URL, the violation report might contain sensitive information contained in the redirected URL, such as session identifiers or purported identities. For this reason, the user agent includes only the URL of the original request, not the redirect target.

Note also that violation reports should be considered attacker-controlled data. Developers who wish to collect violation reports in a dashboard or similar service should be careful to properly escape their content before rendering it (and should probably themselves use CSP to further mitigate the risk of injection). This is especially true for the "`script-sample`" property of violation reports, and the `sample` property of `SecurityPolicyViolationEvent`, which are both completely attacker-controlled strings.

§ 7.6. Paths and Redirects

To avoid leaking path information cross-origin (as discussed in Egor Homakov's Using Content-Security-Policy for Evil), the matching algorithm ignores the path component of a source expression if the resource being loaded is the result of a redirect. For example, given a page with an active policy of `img-src example.com example.org/path`:

- Directly loading `https://example.org/not-path` would fail, as it doesn't match the policy.
- Directly loading `https://example.com/redirector` would pass, as it matches `example.com`.
- Assuming that `https://example.com/redirector` delivered a redirect response pointing to `https://example.org/not-path`, the load would succeed, as the initial URL matches `example.com`, and the redirect target matches `example.org/path` if we ignore its path component.

This restriction reduces the granularity of a document's policy when redirects are in play, a necessary

compromise to avoid brute-forced information leaks of this type.

The relatively long thread "Remove paths from CSP?" from public-webappsec@w3.org has more detailed discussion around alternate proposals.

§ 7.7. Secure Upgrades

To mitigate one variant of history-scanning attacks like Yan Zhu's Sniffly🔒, CSP will not allow pages to lock themselves into insecure URLs via policies like `script-src http://example.com`. As described in §6.6.2.7 scheme-part matching, the scheme portion of a source expression will always allow upgrading to a secure variant.

§ 7.8. CSP Inheriting to avoid bypasses

As described in §4.2.1 Initialize a Document's CSP list and §4.2.2 Initialize a global object's CSP list, documents loaded from local schemes will inherit a copy of the policies in the CSP list of the embedding document or opener browsing context. The goal is to ensure that a page can't bypass its policy by embedding a frame or opening a new window containing content that is entirely under its control (`srcdoc` documents, `blob:` or `data:` URLs, `about:blank` documents that can be manipulated via `document.write()`, etc).

> EXAMPLE 21
> If this would not happen a page could execute inline scripts even without `unsafe-inline` in the page's execution context by simply embedding a `srcdoc iframe`.
>
> ```
> <iframe srcdoc="<script>alert(1);</script>"></iframe>
> ```

Note that we create a copy of the CSP list which means that the new `Document`'s CSP list is a snapshot of the relevant policies at its creation time. Modifications in the CSP list of the new `Document` won't affect the embedding document or opener browsing context's CSP list or vice-versa.

EXAMPLE 22

In the example below the image inside the iframe will not load because it is blocked by the policy in the `meta` tag of the iframe. The image outside the iframe will load (assuming the main page policy does not block it) since the policy inserted in the iframe will not affect it.

```
<iframe srcdoc='<meta http-equiv="Content-Security-Policy" content="img-src ex
                  <img src="not-example.com/image">'></iframe>

<img src="not-example.com/image">
```

8. Authoring Considerations

8.1. The effect of multiple policies

This section is not normative.

The above sections note that when multiple policies are present, each must be enforced or reported, according to its type. An example will help clarify how that ought to work in practice. The behavior of an `XMLHttpRequest` might seem unclear given a site that, for whatever reason, delivered the following HTTP headers:

EXAMPLE 23

```
Content-Security-Policy: default-src 'self' http://example.com http://example.ne
                         connect-src 'none';
Content-Security-Policy: connect-src http://example.com/;
                         script-src http://example.com/
```

Is a connection to example.com allowed or not? The short answer is that the connection is not allowed. Enforcing both policies means that a potential connection would have to pass through both unscathed. Even though the second policy would allow this connection, the first policy contains `connect-src 'none'`, so its enforcement blocks the connection. The impact is that adding additional policies to the list of policies to enforce can *only* further restrict the capabilities of the protected resource.

To demonstrate that further, consider a script tag on this page. The first policy would lock scripts down to `'self'`, `http://example.com` and `http://example.net` via the `default-src` directive. The second, however, would only allow script from `http://example.com/`. Script will only load if it

meets both policy's criteria: in this case, the only origin that can match is `http://example.com`, as both policies allow it.

§ 8.2. Usage of "'strict-dynamic'"

Host- and path-based policies are tough to get right, especially on sprawling origins like CDNs. The solutions to Cure53's H5SC Minichallenge 3: "Sh*t, it's CSP!" [H5SC3] are good examples of the kinds of bypasses which such policies can enable, and though CSP is capable of mitigating these bypasses via exhaustive declaration of specific resources, those lists end up being brittle, awkward, and difficult to implement and maintain.

The "`'strict-dynamic'`" source expression aims to make Content Security Policy simpler to deploy for existing applications who have a high degree of confidence in the scripts they load directly, but low confidence in their ability to provide a reasonable list of resources to load up front.

If present in a `script-src` or `default-src` directive, it has two main effects:

1. host-source and scheme-source expressions, as well as the "`'unsafe-inline'`" and "`'self'` keyword-sources will be ignored when loading script.

   hash-source and nonce-source expressions will be honored.

2. Script requests which are triggered by non-"parser-inserted" `<script>` elements are allowed.

The first change allows you to deploy "`'strict-dynamic'` in a backwards compatible way, without requiring user-agent sniffing: the policy `'unsafe-inline' https: 'nonce-abcdefg' 'strict-dynamic'` will act like `'unsafe-inline' https:` in browsers that support CSP1, `https: 'nonce-DhcnhD3khTMePgXwdayK9BsMqXjhguVV'` in browsers that support CSP2, and `'nonce-DhcnhD3khTMePgXwdayK9BsMqXjhguVV' 'strict-dynamic'` in browsers that support CSP3.

The second allows scripts which are given access to the page via nonces or hashes to bring in their dependencies without adding them explicitly to the page's policy.

EXAMPLE 24

Suppose MegaCorp, Inc. deploys the following policy:

```
Content-Security-Policy: script-src 'nonce-DhcnhD3khTMePgXwdayK9BsMqXjhguVV' 'st
```

And serves the following HTML with that policy active:

```
...
<script src="https://cdn.example.com/script.js" nonce="DhcnhD3khTMePgXwdayK9Bs
...
```

This will generate a request for `https://cdn.example.com/script.js`, which will not be blocked because of the matching `nonce` attribute.

If `script.js` contains the following code:

```
var s = document.createElement('script');
s.src = 'https://othercdn.not-example.net/dependency.js';
document.head.appendChild(s);

document.write('<scr' + 'ipt src="/sadness.js"></scr' + 'ipt>');
```

`dependency.js` will load, as the `<script>` element created by `createElement()` is not "parser-inserted".

`sadness.js` will *not* load, however, as `document.write()` produces `<script>` elements which are "parser-inserted".

§ 8.3. Usage of "'unsafe-hashes'"

This section is not normative.

Legacy websites and websites with legacy dependencies might find it difficult to entirely externalize event handlers. These sites could enable such handlers by allowing `'unsafe-inline'`, but that's a big hammer with a lot of associated risk (and cannot be used in conjunction with nonces or hashes).

The "`'unsafe-hashes'`" source expression aims to make CSP deployment simpler and safer in these situations by allowing developers to enable specific handlers via hashes.

EXAMPLE 25

MegaCorp, Inc. can't quite get rid of the following HTML on anything resembling a reasonable schedule:

```
<button id="action" onclick="doSubmit()">
```

Rather than reducing security by specifying `"'unsafe-inline'"`, they decide to use `"'unsafe-hashes'"` along with a hash source expression, as follows:

```
Content-Security-Policy:  script-src 'unsafe-hashes' 'sha256-jzgBGA4UWFFmpOBq0Jp
```

The capabilities `'unsafe-hashes'` provides is useful for legacy sites, but should be avoided for modern sites. In particular, note that hashes allow a particular script to execute, but do not ensure that it executes in the way a developer intends. If an interesting capability is exposed as an inline event handler (say `<a onclick="transferAllMyMoney()">Transfer</a>`), then that script becomes available for an attacker to inject as `<script>transferAllMyMoney()</script>`. Developers should be careful to balance the risk of allowing specific scripts to execute against the deployment advantages that allowing inline event handlers might provide.

§ 8.4. Allowing external JavaScript via hashes

In [CSP2], hash source expressions could only match inlined script, but now that Subresource Integrity [SRI] is widely deployed, we can expand the scope to enable externalized JavaScript as well.

If multiple sets of integrity metadata are specified for a `<script>`, the request will match a policy's hash-sources if and only if *each* item in a `<script>`'s integrity metadata matches the policy.

Note: The CSP spec specifies that the contents of an inline `<script>` element or event handler needs to be encoded using UTF-8 encode before computing its hash. [SRI] computes the hash on the raw resource that is being fetched instead. This means that it is possible for the hash needed to whitelist an inline script block to be different that the hash needed to whitelist an external script even if they have identical contents.

EXAMPLE 26

MegaCorp, Inc. wishes to allow two specific scripts on a page in a way that ensures that the content matches their expectations. They do so by setting the following policy:

```
Content-Security-Policy: script-src 'sha256-abc123' 'sha512-321cba'
```

In the presence of that policy, the following `<script>` elements would be allowed to execute because they contain only integrity metadata that matches the policy:

```
<script integrity="sha256-abc123" ...></script>
<script integrity="sha512-321cba" ...></script>
<script integrity="sha256-abc123 sha512-321cba" ...></script>
```

While the following `<script>` elements would not execute because they contain valid metadata that does not match the policy (even though other metadata does match):

```
<script integrity="sha384-xyz789" ...></script>
<script integrity="sha384-xyz789 sha512-321cba" ...></script>
<script integrity="sha256-abc123 sha384-xyz789 sha512-321cba" ...></script>
```

Metadata that is not recognized (either because it's entirely invalid, or because it specifies a not-yet-supported hashing algorithm) does not affect the behavior described here. That is, the following elements would be allowed to execute in the presence of the above policy, as the additional metadata is invalid and therefore wouldn't allow a script whose content wasn't listed explicitly in the policy to execute:

```
<script integrity="sha256-abc123 sha1024-abcd" ...></script>
<script integrity="sha512-321cba entirely-invalid" ...></script>
<script integrity="sha256-abc123 not-a-hash-at-all sha512-321cba" ...></script
```

§ 9. Implementation Considerations

§ 9.1. Vendor-specific Extensions and Addons

Policy enforced on a resource SHOULD NOT interfere with the operation of user-agent features like addons, extensions, or bookmarklets. These kinds of features generally advance the user's priority over page authors, as espoused in [HTML-DESIGN].

Moreover, applying CSP to these kinds of features produces a substantial amount of noise in violation reports, significantly reducing their value to developers.

Chrome, for example, excludes the `chrome-extension:` scheme from CSP checks, and does some work to ensure that extension-driven injections are allowed, regardless of a page's policy.

§ 10. IANA Considerations

§ 10.1. Directive Registry

The Content Security Policy Directive registry should be updated with the following directives and references [RFC7762]:

`base-uri`
: This document (see §6.2.1 base-uri)

`child-src`
: This document (see §6.1.1 child-src)

`connect-src`
: This document (see §6.1.2 connect-src)

`default-src`
: This document (see §6.1.3 default-src)

`font-src`
: This document (see §6.1.4 font-src)

`form-action`
: This document (see §6.3.1 form-action)

`frame-ancestors`
: This document (see §6.3.2 frame-ancestors)

`frame-src`
: This document (see §6.1.5 frame-src)

`img-src`
: This document (see §6.1.6 img-src)

`manifest-src`
: This document (see §6.1.7 manifest-src)

`media-src`
: This document (see §6.1.8 media-src)

`object-src`
: This document (see §6.1.10 object-src)

`plugin-types`
: This document (see §6.2.2 plugin-types)

`report-uri`
: This document (see §6.4.1 report-uri)

`report-to`
: This document (see §6.4.2 report-to)

`sandbox`
: This document (see §6.2.3 sandbox)

`script-src`
: This document (see §6.1.11 script-src)

`script-src-attr`
: This document (see §6.1.13 script-src-attr)

`script-src-elem`
: This document (see §6.1.12 script-src-elem)

`style-src`
: This document (see §6.1.14 style-src)

`style-src-attr`
: This document (see §6.1.16 style-src-attr)

`style-src-elem`
: This document (see §6.1.15 style-src-elem)

`worker-src`
: This document (see §6.1.17 worker-src)

§ 10.2. Headers

The permanent message header field registry should be updated with the following registrations: [RFC3864]

§ 10.2.1. Content-Security-Policy

Header field name
: Content-Security-Policy

Applicable protocol
: http

Status
: standard

Author/Change controller
W3C

Specification document
This specification (See §3.1 The Content-Security-Policy HTTP Response Header Field)

§ 10.2.2. Content-Security-Policy-Report-Only

Header field name
Content-Security-Policy-Report-Only

Applicable protocol
http

Status
standard

Author/Change controller
W3C

Specification document
This specification (See §3.2 The Content-Security-Policy-Report-Only HTTP Response Header Field)

§ 11. Acknowledgements

Lots of people are awesome. For instance:

- Mario and all of Cure53.
- Artur Janc, Michele Spagnuolo, Lukas Weichselbaum, Jochen Eisinger, and the rest of Google's CSP Cabal.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 27
>
> This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

§ Conformant Algorithms

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant. Implementers are encouraged to optimize.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[csp-3] defines the following terms:

content-security-policy

parse a serialized csp

[css-cascade-4] defines the following terms:

@import

[CSSOM] defines the following terms:

- insert a css rule
- parse a css declaration block
- parse a css rule
- parse a group of selectors

[DOM] defines the following terms:

- Document
- Element
- Event
- EventInit
- bubbles
- composed
- connected
- document
- fire an event
- node document
- shadow-including root
- target

[ECMA262] defines the following terms:

- Function()
- HostEnsureCanCompileStrings()
- JSON.stringify()
- eval()
- realm

[ENCODING] defines the following terms:

- utf-8 encode

[FETCH] defines the following terms:

- body
- client
- credentials mode
- cryptographic nonce metadata
- csp list
- current url
- destination
- extract a mime type
- extracting header list values
- fetch
- header list (for response)
- http fetch
- http-network fetch
- initiator
- integrity metadata
- keepalive flag
- local scheme
- main fetch
- method
- mode
- network error
- network scheme
- origin
- parser metadata
- redirect count
- redirect mode
- redirect status
- request
- response
- script-like
- status
- target browsing context
- url (for response)
- window

[HTML] defines the following terms:

- "parser-inserted"
- DedicatedWorkerGlobalScope
- SharedWorker
- SharedWorkerGlobalScope
- Window
- Worker
- WorkerGlobalScope
- a
- active document
- an iframe srcdoc document
- applet
- ascii serialization of an origin
- associated document
- base
- browsing context
- case-sensitive
- content
- content security policy state
- csp list
- current settings object
- data
- document
- duplicate-attribute
- embed
- fallback base url
- forced sandboxing flag set
- form
- frame
- global object (for environment settings object)
- href
- http-equiv
- iframe
- initializing a new document object
- link
- meta
- nested browsing context
- nested through
- nonce
- object
- opener browsing context
- origin (for environment settings object)
- owner set
- parent browsing context
- parse a sandboxing directive
- parse error
- ping
- plugin document
- prepare a script
- process a navigate fetch
- process a navigate response
- queue a task
- referrer
- relevant global object
- relevant settings object
- run a worker
- sandbox
- sandboxed origin browsing context flag
- sandboxed scripts browsing context flag
- scheme
- script
- set the frozen base url
- setInterval()
- setTimeout()
- style
- top-level browsing context
- type
- update a style block

[INFRA] defines the following terms:

- append (for set)
- ascii case-insensitive
- ascii lowercase
- ascii string
- ascii whitespace
- collecting a sequence of code points
- contain
- continue
- convert
- infra
- is empty
- list
- ordered set
- set
- split a string on ascii whitespace
- split a string on commas
- strictly split a string
- string
- strip leading and trailing ascii whitespace

[MIMESNIFF] defines the following terms:

- valid mime type

[REPORTING] defines the following terms:

- group
- queue report

[rfc2045] defines the following terms:

- subtype
- type

[RFC3986] defines the following terms:

- ipv4address
- path-absolute
- scheme
- uri-reference

[rfc4648] defines the following terms:

- base64 encoding
- base64url encoding

[RFC5234] defines the following terms:

- alpha
- digit
- vchar

[RFC7230] defines the following terms:

- ows
- token

[rfc7231] defines the following terms:

- representation
- resource representation

[service-workers-1] defines the following terms:

- ServiceWorker
- ServiceWorkerGlobalScope

[sha2] defines the following terms:

- sha-256
- sha-384
- sha-512

[URL] defines the following terms:

- URL
- base url
- default port
- host (for url)
- ipv6 address
- origin
- path
- percent decode
- port (for url)
- scheme
- url parser
- url serializer

[WebIDL] defines the following terms:

- DOMString
- Exposed
- USVString
- unsigned long
- unsigned short

[worklets-1] defines the following terms:

- WorkletGlobalScope
- owner document

§ References

§ Normative References

[CSP-3]
Content Security Policy Level 3 URL: https://www.w3.org/TR/CSP3/

[CSS-CASCADE-4]
Elika Etemad; Tab Atkins Jr.. CSS Cascading and Inheritance Level 4. 28 August 2018. CR. URL: https://www.w3.org/TR/css-cascade-4/

[CSSOM]
Simon Pieters; Glenn Adams. CSS Object Model (CSSOM). 17 March 2016. WD. URL: https://www.w3.org/TR/cssom-1/

[DOM]
Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMA262]
Brian Terlson; Allen Wirfs-Brock. ECMAScript® Language Specification. URL: https://tc39.github.io/ecma262/

[ENCODING]
Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]
Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[HTML]
Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[MIMESNIFF]
Gordon P. Hemsley. MIME Sniffing Standard. Living Standard. URL: https://mimesniff.spec.whatwg.org/

[REPORTING]
Ilya Gregorik; Mike West. Reporting API. URL: https://wicg.github.io/reporting/

[RFC2045]
N. Freed; N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. November 1996. Draft Standard. URL: https://tools.ietf.org/html/rfc2045

[RFC2119]
S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3492]
A. Costello. Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). March 2003. Proposed Standard. URL: https://tools.ietf.org/html/rfc3492

[RFC3864]
G. Klyne; M. Nottingham; J. Mogul. Registration Procedures for Message Header Fields. September 2004. Best Current Practice. URL: https://tools.ietf.org/html/rfc3864

[RFC3986]
T. Berners-Lee; R. Fielding; L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[RFC4648]
S. Josefsson. The Base16, Base32, and Base64 Data Encodings. October 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4648

[RFC5234]
D. Crocker, Ed.; P. Overell. Augmented BNF for Syntax Specifications: ABNF. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234

[RFC7034]
D. Ross; T. Gondrom. HTTP Header Field X-Frame-Options. October 2013. Informational. URL: https://tools.ietf.org/html/rfc7034

[RFC7230]
R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7230

[RFC7231]
R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7231

[RFC7762]
M. West. Initial Assignment for the Content Security Policy Directives Registry. January 2016. Informational. URL: https://tools.ietf.org/html/rfc7762

[RFC8288]
M. Nottingham. Web Linking. October 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8288

[SERVICE-WORKERS-1]
Alex Russell; et al. Service Workers 1. 2 November 2017. WD. URL: https://www.w3.org/TR/service-workers-1/

[SRI]
Devdatta Akhawe; et al. Subresource Integrity. 23 June 2016. REC. URL: https://www.w3.org/TR/SRI/

[URL]
Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

[WORKLETS-1]
Ian Kilpatrick. Worklets Level 1. 7 June 2016. WD. URL: https://www.w3.org/TR/worklets-1/

§ Informative References

[APPMANIFEST]
Marcos Caceres; et al. Web App Manifest. 6 September 2018. WD. URL: https://www.w3.org/TR/appmanifest/

[BEACON]
Ilya Grigorik; et al. Beacon. 13 April 2017. CR. URL: https://www.w3.org/TR/beacon/

[CSP2]
Mike West; Adam Barth; Daniel Veditz. Content Security Policy Level 2. 15 December 2016. REC. URL: https://www.w3.org/TR/CSP2/

[CSS-ABUSE]
Chris Evans. Generic cross-browser cross-domain theft. 28 December 2009. URL: https://scarybeastsecurity.blogspot.com/2009/12/generic-cross-browser-cross-domain.html

[EVENTSOURCE]
Ian Hickson. Server-Sent Events. 3 February 2015. REC. URL: https://www.w3.org/TR/eventsource/

[FILEDESCRIPTOR-2015]
filedescriptor. CSP 2015. 23 November 2015. URL: https://blog.innerht.ml/csp-2015/#danglingmarkupinjection

[H5SC3]

Mario Heiderich. H5SC Minichallenge 3: "Sh*t, it's CSP!". URL: https://github.com/cure53/XSSChallengeWiki/wiki/H5SC-Minichallenge-3:-%22Sh*t,-it%27s-CSP!%22

[HTML-DESIGN]

Anne Van Kesteren; Maciej Stachowiak. HTML Design Principles. URL: https://www.w3.org/TR/html-design-principles/

[MIX]

Mike West. Mixed Content. 2 August 2016. CR. URL: https://www.w3.org/TR/mixed-content/

[TIMING]

Paul Stone. Pixel Perfect Timing Attacks with HTML5. URL: https://www.contextis.com/media/downloads/Pixel_Perfect_Timing_Attacks_with_HTML5_Whitepaper.pdf

[UISECURITY]

Brad Hill. User Interface Security and the Visibility API. 7 June 2016. WD. URL: https://www.w3.org/TR/UISecurity/

[UPGRADE-INSECURE-REQUESTS]

Mike West. Upgrade Insecure Requests. 8 October 2015. CR. URL: https://www.w3.org/TR/upgrade-insecure-requests/

[WEBSOCKETS]

Ian Hickson. The WebSocket API. 20 September 2012. CR. URL: https://www.w3.org/TR/websockets/

[XHR]

Anne van Kesteren. XMLHttpRequest Standard. Living Standard. URL: https://xhr.spec.whatwg.org/

[XSLT]

James Clark. XSL Transformations (XSLT) Version 1.0. 16 November 1999. REC. URL: https://www.w3.org/TR/xslt/

§ IDL Index

```
enum SecurityPolicyViolationEventDisposition {
  "enforce", "report"
};

[Constructor(DOMString type, optional SecurityPolicyViolationEventInit eventInitD
 Exposed=(Window,Worker)]
interface SecurityPolicyViolationEvent : Event {
    readonly    attribute USVString      documentURI;
    readonly    attribute USVString      referrer;
```

```
    readonly    attribute USVString      blockedURI;
    readonly    attribute DOMString      violatedDirective;
    readonly    attribute DOMString      effectiveDirective;
    readonly    attribute DOMString      originalPolicy;
    readonly    attribute USVString      sourceFile;
    readonly    attribute DOMString      sample;
    readonly    attribute SecurityPolicyViolationEventDisposition      dispositic
    readonly    attribute unsigned short statusCode;
    readonly    attribute unsigned long  lineNumber;
    readonly    attribute unsigned long  columnNumber;
};

dictionary SecurityPolicyViolationEventInit : EventInit {
    required USVString      documentURI;
             USVString      referrer = "";
             USVString      blockedURI = "";
    required DOMString      violatedDirective;
    required DOMString      effectiveDirective;
    required DOMString      originalPolicy;
             USVString      sourceFile = "";
             DOMString      sample = "";
    required SecurityPolicyViolationEventDisposition disposition;
    required unsigned short statusCode;
             unsigned long  lineNumber = 0;
             unsigned long  columnNumber = 0;
};
```

§ Issues Index

ISSUE 1 Is this kind of thing specified anywhere? I didn't see anything that looked useful in [ECMA262]. ↲

ISSUE 2 How, exactly, do we get the status code? We don't actually store it anywhere. ↲

ISSUE 3 This concept is missing from W3C's Workers. <https://github.com/w3c/html/issues/187> ↲

ISSUE 4 Stylesheet loading is not yet integrated with Fetch in W3C's HTML. <https://github.com/whatwg/html/issues/198> ↲

ISSUE 5 Stylesheet loading is not yet integrated with Fetch in WHATWG's HTML. <https://github.com/whatwg/html/issues/968> ↲

ISSUE 6 This hook is missing from W3C's HTML. <https://github.com/w3c/html/issues/547> ↲

ISSUE 7 W3C's HTML is not based on Fetch, and does not have a process a navigate response algorithm into which to hook. <https://github.com/w3c/html/issues/548> ↲

ISSUE 8 `HostEnsureCanCompileStrings()` does not include the string which is going to be compiled as a parameter. We'll also need to update HTML to pipe that value through to CSP. <https://github.com/tc39/ecma262/issues/938> ↲

ISSUE 9 This needs to be better explained. <https://github.com/w3c/webappsec-csp/issues/212> ↲

ISSUE 10 Do something interesting to the execution context in order to lock down interesting CSSOM algorithms. I don't think CSSOM gives us any hooks here, so let's work with them to put something reasonable together. ↲

ISSUE 11 Spell this out in more detail as part of defining `X-Frame-Options` integration with the process a navigate response algorithm. <https://github.com/whatwg/html/issues/1230> ↲

ISSUE 12 We need some sort of hook in HTML to record this error if we're planning on using it here. <https://github.com/whatwg/html/issues/3257> ↲

ISSUE 13 This processing is meant to mitigate the risk of dangling markup attacks that steal the nonce from an existing element in order to load injected script. It is fairly expensive, however, as it requires that we walk through all attributes and their values in order to determine whether the script should execute. Here, we try to minimize the impact by doing this check only for `<script>` elements when a nonce is present, but we should probably consider this algorithm as "at risk" until we know its impact. <https://github.com/w3c/webappsec-csp/issues/98> ↲