

# Content Security Policy Level 3

W3C Working Draft, 18 August 2016



## This version:

<https://www.w3.org/TR/2016/WD-CSP3-20160818/>

## 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-CSP3-20160801/>

## Version History:

<https://github.com/w3c/webappsec-csp/commits/master/index.src.html>

## Feedback:

[public-webappsec@w3.org](mailto:public-webappsec@w3.org) with subject line “ [CSP3] ... message topic ...” ([archives](#) )

## Editor:

[Mike West](#) (Google Inc. )

## Participate:

[File an issue](#) ([open issues](#) )

Copyright © 2016 W3C® ([MIT](#), [ERCIM](#), [Keio](#), [Beihang](#)). W3C [liability](#), [trademark](#) and [document use](#) rules apply.

---

## 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/](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](mailto: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 [5 February 2004 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 September 2015 W3C Process Document](#).

## Table of Contents

<b>1</b>	<b>Introduction</b>
1.1	Examples
1.1.1	Control Execution
1.2	Goals
1.3	Changes from Level 2
<b>2</b>	<b>Framework</b>
2.1	Policies
2.1.1	Parse a <i>serialized CSP</i> as <i>disposition</i>
2.1.2	Parse a <i>serialized CSP list</i> as <i>disposition</i>
2.2	Directives
2.2.1	Source Lists
2.3	Violations
2.3.1	Create a violation object for <i>global</i> , <i>policy</i> , and <i>directive</i>
2.3.2	Create a violation object for <i>request</i> , <i>policy</i> , and <i>directive</i>
<b>3</b>	<b>Policy Delivery</b>
3.1	The Content-Security-Policy HTTP Response Header Field
3.2	The Content-Security-Policy-Report-Only HTTP Response Header Field
3.3	The <meta> element
<b>4</b>	<b>Integrations</b>
4.1	Integration with Fetch
4.1.1	Set <i>response</i> 's <i>CSP list</i>
4.1.2	Report Content Security Policy violations for <i>request</i>
4.1.3	Should <i>request</i> be blocked by Content Security Policy?
4.1.4	Should <i>response</i> to <i>request</i> be blocked by Content Security Policy?
4.2	Integration with HTML

- 4.2.1 Initialize a Document's CSP *list*
- 4.2.2 Initialize a global object's CSP *list*
- 4.2.3 Should *element's* inline *type* behavior be blocked by Content Security Policy?
- 4.2.4 Should *navigation request* of *type* from *source* in *target* be blocked by Content Security Policy?
- 4.2.5 Should *navigation response* to *navigation request* of *type* from *source* in *target* be blocked by Content Security Policy?
- 4.3 Integration with ECMAScript
- 4.3.1 `EnsureCSPDoesNotBlockStringCompilation(callerRealm, calleeRealm)`

## 5 Reporting

- 5.1 Violation DOM Events
- 5.2 Obtain the deprecated serialization of *violation*
- 5.3 Report a *violation*

## 6 Content Security Policy Directives

### 6.1 Fetch Directives

- 6.1.1 *child-src*
  - 6.1.1.1 *child-src* Pre-request check
  - 6.1.1.2 *child-src* Post-request check
- 6.1.2 *connect-src*
  - 6.1.2.1 *connect-src* Pre-request check
  - 6.1.2.2 *connect-src* Post-request check
- 6.1.3 *default-src*
  - 6.1.3.1 *default-src* Pre-request check
  - 6.1.3.2 *default-src* Post-request check
- 6.1.4 *font-src*
  - 6.1.4.1 *font-src* Pre-request check
  - 6.1.4.2 *font-src* Post-request check
- 6.1.5 *frame-src*
  - 6.1.5.1 *frame-src* Pre-request check
  - 6.1.5.2 *frame-src* Post-request check
- 6.1.6 *img-src*
  - 6.1.6.1 *img-src* Pre-request check
  - 6.1.6.2 *img-src* Post-request check
- 6.1.7 *manifest-src*
  - 6.1.7.1 *manifest-src* Pre-request check
  - 6.1.7.2 *manifest-src* Post-request check
- 6.1.8 *media-src*
  - 6.1.8.1 *media-src* Pre-request check
  - 6.1.8.2 *media-src* Post-request check
- 6.1.9 *object-src*
  - 6.1.9.1 *object-src* Pre-request check
  - 6.1.9.2 *object-src* Post-request check
- 6.1.10 *script-src*

- 6.1.10.1 script-src Pre-request check
- 6.1.10.2 script-src Post-request check
- 6.1.10.3 script-src Inline Check
- 6.1.11 style-src
  - 6.1.11.1 style-src Pre-request Check
  - 6.1.11.2 style-src Post-request Check
  - 6.1.11.3 style-src Inline Check
- 6.1.12 worker-src
  - 6.1.12.1 worker-src Pre-request Check
  - 6.1.12.2 worker-src Post-request Check
- 6.2 Document Directives
  - 6.2.1 base-uri
    - 6.2.1.1 Is *base* allowed for *document*?
  - 6.2.2 plugin-types
    - 6.2.2.1 plugin-types Post-Request Check
    - 6.2.2.2 Should *plugin element* be blocked *a priori* by Content Security Policy?:
  - 6.2.3 sandbox
    - 6.2.3.1 sandbox Response Check
    - 6.2.3.2 sandbox Initialization
  - 6.2.4 disown-opener
    - 6.2.4.1 disown-opener Initialization
- 6.3 Navigation Directives
  - 6.3.1 form-action
    - 6.3.1.1 form-action Pre-Navigation Check
  - 6.3.2 frame-ancestors
    - 6.3.2.1 frame-ancestors Navigation Response Check
- 6.4 Reporting Directives
  - 6.4.1 report-uri
  - 6.4.2 report-to
- 6.5 Directives Defined in Other Documents
- 6.6 Matching Algorithms
  - 6.6.1 URL Matching
    - 6.6.1.1 Does *request* violate *policy*?
    - 6.6.1.2 Does *nonce* match *source list*?
    - 6.6.1.3 Does *request* match *source list*?
    - 6.6.1.4 Does *response* to *request* match *source list*?
    - 6.6.1.5 Does *url* match *source list* in *origin* with *redirect count*?
    - 6.6.1.6 Does *url* match *expression* in *origin* with *redirect count*?
    - 6.6.1.7 Get the effective directive for *request*
  - 6.6.2 Element Matching Algorithms
    - 6.6.2.1 Does *element* match *source list* for *type* and *source*?

## 7 Security and Privacy Considerations

### 7.1 Nonce Reuse

7.2	CSS Parsing
7.3	Violation Reports
<b>8</b>	<b>Authoring Considerations</b>
8.1	The effect of multiple policies
8.2	Usage of "'strict-dynamic'"
8.3	Usage of "'unsafe-hashed-attributes'"
8.4	Whitelisting external JavaScript with hashes
<b>9</b>	<b>Implementation Considerations</b>
9.1	Vendor-specific Extensions and Addons
<b>10</b>	<b>IANA Considerations</b>
10.1	Directive Registry
10.2	Headers
10.2.1	Content-Security-Policy
10.2.2	Content-Security-Policy-Report-Only
<b>11</b>	<b>Acknowledgements</b>
	<b>Conformance</b>
	Document conventions
	Conformant Algorithms
	<b>Index</b>
	Terms defined by this specification
	Terms defined by reference
	<b>References</b>
	Normative References
	Informative References
	<b>IDL Index</b>
	<b>Issues Index</b>

## 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 'none'
```

## 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 `frame-src` directive, which was deprecated in CSP Level 2, has been undeprecated, and a `worker-src` directive added. Both defer to `child-src` if not present (which defers to `default-src` in turn).
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 [\[OOB-REPORTING\]](#) 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-hashed-attributes'` source expression will now allow event handlers and style attributes to match hash source expressions. Details in [§8.3 Usage of "'unsafe-hashed-attributes'"](#).
9. The [source expression](#) matching has been changed to require explicit whitelisting of any non-[network scheme](#), rather than [local scheme](#), as described in [§6.6.1.6 Does url match expression in origin with redirect count?](#).
10. Hash-based source expressions may now whitelist external scripts if the `<script>` element that triggers the request specifies a set of integrity metadata which is whitelisted by the current policy. Details in [§8.4 Whitelisting external JavaScript with hashes](#).
11. The [disown-opener](#) directive ensures that a resource can't be opened in such a way as to give another browsing context control over its contents.

## 2. Framework§

### 2.1. Policies§

A **policy** defines a set of allowed and restricted behaviors, and may be applied to a [Window](#) or [WorkerGlobalScope](#) as described in [§4.2.2 Initialize a global object's CSP list](#).

Each policy has an associated **directive set**, which is a set of [directives](#) that define the policy's implications when applied.

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

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](#) \*( [OWS](#) ";" [ [OWS](#) [serialized-directive](#) ] )  
; [OWS](#) is defined in section 3.2.3 of RFC 7230

### 2.1.1. Parse a *serialized CSP as disposition*

Given a [serialized CSP](#) (*serialized CSP*), and a [disposition](#) (*disposition*), this algorithm will return a [policy](#) object. If the string cannot be parsed, the resulting [policy](#)'s [directive set](#) will be empty.

1. Let *policy* be a new [policy](#) with an empty [directive set](#), and a [disposition](#) of *disposition*.
  2. For each *token* returned by [strictly splitting](#) *serialized CSP* on the U+003B SEMICOLON character (;):
    1. [Strip leading and trailing whitespace](#) from *token*.
    2. If *token* is an empty string, skip the remaining substeps and continue to the next item.
    3. Let *directive name* be the result of [collecting a sequence of characters](#) from *token* which are not [space characters](#).
    4. If *policy*'s [directive set](#) contains a [directive](#) whose [name](#) is *directive name*, skip the remaining substeps and continue to the next item.
- The user agent SHOULD notify developers that a directive was ignored. A console warning might be appropriate, for example.
5. Let *directive value* be the result of [splitting token on spaces](#).
  6. Let *directive* be a new [directive](#) whose [name](#) is *directive name*, and [value](#) is *directive value*.
  7. Add *directive* to *policy*'s [directive set](#).
3. Return *policy*.

### 2.1.2. Parse a *serialized CSP list as disposition*

Given a string (*list*) and a [disposition](#) (*disposition*) which contains a comma-delimited series of [serialized CSP](#) strings, the following algorithm will return a list of [policy](#) objects:

1. Let *policies* be an empty list.
2. For each *token* returned by [splitting list on commas](#) :
  1. Let *policy* be the result of executing [§2.1.1 Parse a serialized CSP as disposition](#) on *token* with *disposition*.
  2. If *policy*'s [directive set](#) is empty, skip the remaining substeps, and continue to the next item.
  3. Add *policy* to *policies*.
3. Return *policies*.



## 2.2. Directives

Each [policy](#) contain a set of **directives**, 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](#) set 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 [ RWS directive-value ]
directive-name       = 1*( ALPHA / DIGIT / "-" )
directive-value      = *( %x09 / %x20-%x2B / %x2D-%x3A / %x3C-%7E )
                        ; Directive values may contain whitespace and VCHAR characters,
                        ; excluding ";" and ","

```

; [RWS](#) is defined in section 3.2.3 of RFC7230. [ALPHA](#), [DIGIT](#), and  
; [VCHAR](#) are defined in Appendix B.1 of RFC 5234.

[Directives](#) have six 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, and a source string as arguments, and is executed during [§4.2.3 Should element's inline type behavior be blocked by Content Security Policy?](#). 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](#), type string, and two [browsing contexts](#) as arguments, and is executed during [§4.2.4 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 [response](#) and two [browsing contexts](#) as arguments, and is executed during [§4.2.5 Should navigation response to navigation request of type from](#)

[source in target be blocked by Content Security Policy?](#). It returns "Allowed" unless otherwise specified.

## 2.2.1. Source Lists

Many [directives](#) [values](#) consist of **source lists**: sets of tokens which identify content that can be fetched and potentially embedded or executed. These tokens represent 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 or any of its subdomains (and any of its subdomains' subdomains, and so on))
5. Nonces such as `'nonce-qwertyu12345'` (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 space-delimited series of [source expressions](#), adhering to the following ABNF grammar [\[RFC5234\]](#):

```

serialized-source-list = ( source-expression *( RWS source-expression ) ) / "'none'"
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.js"

```

```

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

```

```

; path is defined in section 3.3 of RFC 3986.

```

```

; Keywords:

```

```

keyword-source = "'self'" / "'unsafe-inline'" / "'unsafe-eval'" / "'strict-dynamic'" / "'unsafe-
hashed-attributes'"

```

```

; 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 `üüüüü.de` MUST be represented as `xn--tdaaaaa.de`.

Note: Though IP address do match the grammar above, only `127.0.0.1` will actually match a URL when u in a source expression (see [§6.6.1.5 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.

## 2.3. 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 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.


### 2.3.1. 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 [\[ECMA262\]](#).

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*.

### 2.3.2. Create a violation object for *request*, *policy*, and *directive*<sup>§</sup>

Given a [request](#) (*request*), 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 the result of executing [§2.3.1 Create a violation object for global, policy, and directive](#) on *request*'s [client](#)'s [global object](#), *policy*, and *directive*.
2. 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.

3. Return *violation*.

## 3. Policy Delivery<sup>§</sup>

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<sup>§</sup>

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](#)

#### 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](#)

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 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.
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.

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](#) 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<sup>§</sup>

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 executing [§2.1.2 Parse a serialized CSP list as disposition](#) on the result of [parsing](#) Content-Security-Policy in *response*'s [header list](#), with a disposition of "enforce".
3. Append to *policies* the result of executing [§2.1.2 Parse a serialized CSP list as disposition](#) on the result of [parsing](#) Content-Security-Policy-Report-Only in *response*'s [header list](#), with 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*<sup>§</sup>

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 [§6.6.1.1 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 [§2.3.2 Create a violation object for request, policy, and directive](#) on *request*, *policy*, and *violates*.

**4.1.3. Should *request* be blocked by Content Security Policy?**<sup>§</sup>

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 [§6.6.1.1 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 [§2.3.2 Create a violation object for request, policy, and directive](#) on *request*, *policy*, and *violates*.
    2. Set *result* to "Blocked".
4. Return *result*.

**4.1.4. Should *response* to *request* be blocked by Content Security Policy?**<sup>§</sup>

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 [§2.3.2 Create a violation object for request, policy, and directive](#) on *request*, *policy*, and *directive*.
      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 §2.3.2 Create a violation object for request, policy, and directive on *request*, *policy*, and *directive*.
      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 and WorkerGlobalScope 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 algorithm.

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

2. A policy is **enforced** or **monitored** for a global object by inserting it into the global object's CSP list.
3. §4.2.2 Initialize a global object's CSP list is called during the initialising a new Document object and run a worker algorithms in order to bind a set of policy objects associated with a response to a newly created global object.
4. §4.2.3 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.
5. §4.2.3 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.
6. Policy is enforced during processing of the <meta> element's http-equiv.
7. A Document's **embedding document** is the Document through which the Document's browsing context is nested.
8. 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>](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>](https://github.com/whatwg/html/issues/968)

9. [§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.
10. [§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>](https://github.com/w3c/html/issues/547)

11. [§4.2.4 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 [§4.2.5 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.

**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>](https://github.com/w3c/html/issues/548)

## 4.2.1. Initialize a Document's CSP list<sup>§</sup>

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 an alias to *policy* in *document*'s [CSP list](#) .

Note: [local scheme](#) includes `about:`, and this algorithm will therefore alias 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 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<sup>§</sup>

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](#):
  1. Let *documents* be an empty list.
  2. Add each of *global*'s [document](#)s to *documents*.
  3. For each *document* in *documents*:
    1. For each *policy* in *document*'s [global object](#)'s [CSP list](#):
      1. Insert an alias to *policy* in *global*'s [CSP list](#).

Note: [local scheme](#) includes `about:`, and this algorithm will therefore alias the [embedding document](#)'s policies for [an iframe srcdoc Document](#).

2. For each *policy* in *response*'s [CSP list](#), insert *policy* into *global*'s [CSP list](#).

#### 4.2.3. Should *element*'s inline *type* behavior be blocked by Content Security Policy?<sup>§</sup>

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:

1. Let *result* be "Allowed".
2. For each *policy* in *element*'s [Document](#)'s [global object](#)'s [CSP list](#):

1. For each *directive* in *policy*:
  1. If *directive*'s [inline check](#) returns "Allowed" when executed upon *element*, *type*, and *source*, skip to the next *directive*.
  2. Otherwise, let *violation* be the result of executing [§2.3.1 Create a violation object for global, policy, and directive](#) on the [current settings object](#)'s [global object](#), *policy*, and "style-src" if *type* is "style" or "style-attribute", or "script-src" otherwise.
  3. Set *violation*'s [resource](#) to "inline".
  4. Execute [§5.3 Report a violation](#) on *violation*.
  5. If *policy*'s [disposition](#) is "enforce", then set *result* to "Blocked".
3. Return *result*.

**4.2.4. 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*, and *target*, skip to the next *directive*.
    2. Otherwise, let *violation* be the result of executing [§2.3.1 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. Return *result*.

**4.2.5. 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](#) :
  1. For each *directive* in *policy*:
    1. If *directive*'s [navigation response check](#) returns " Allowed" when executed upon *navigation request*, *type*, *navigation response*, *source*, and *target*, skip to the next *directive*.
    2. Otherwise, let *violation* be the result of executing [§2.3.1 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. 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 which examines the relevant [CSP list](#) to determine whether such compilation ought to be blocked.

### 4.3.1. EnsureCSPDoesNotBlockStringCompilation(*callerRealm*, *calleeRealm*)

Given two [realms](#) (*callerRealm* and *calleeRealm*), 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. 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 non-null, and does not contain a [source expression](#) which is an [ASCII case-](#)

insensitive match for the string "'unsafe-eval'", then throw an `EvalError` exception.

## 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§

```
[Constructor(DOMString type, optional SecurityPolicyViolationEventInit eventInitDict)]
interface SecurityPolicyViolationEvent : Event {
    readonly attribute DOMString documentURI;
    readonly attribute DOMString referrer;
    readonly attribute DOMString blockedURI;
    readonly attribute DOMString violatedDirective;
    readonly attribute DOMString effectiveDirective;
    readonly attribute DOMString originalPolicy;
    readonly attribute DOMString sourceFile;
    readonly attribute unsigned short statusCode;
    readonly attribute long lineNumber;
    readonly attribute long columnNumber;
};

dictionary SecurityPolicyViolationEventInit : EventInit {
    DOMString documentURI;
    DOMString referrer;
    DOMString blockedURI;
    DOMString violatedDirective;
    DOMString effectiveDirective;
    DOMString originalPolicy;
    DOMString sourceFile;
    unsigned short statusCode;
    long lineNumber;
    long columnNumber;
};
```

### 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](#)

**"status-code"**

*violation*'s [status](#)

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. 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 [global object](#).

1. [Fire an event](#) named `securitypolicyviolation` that uses the [SecurityPolicyViolationEvent](#) at *violation*'s [global object](#) with its attributes initialized as follows:

[documentURI](#)

*violation*'s [url](#)

[referrer](#)

*violation*'s [referrer](#)

[blockedURI](#)

*violation*'s [resource](#)

[effectiveDirective](#)

*violation*'s [effective directive](#)

[violatedDirective](#)

*violation*'s [effective directive](#)

[originalPolicy](#)

*violation*'s [policy](#)

sourceFile

*violation*'s source file

statusCode

*violation*'s status

lineNumber

*violation*'s line number

columnNumber

*violation*'s column number

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

2. 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. Let *endpoint* be the result of executing the URL parser on *directive*'s value.
  3. If *endpoint* is not a valid URL, skip the remaining substeps.
  4. Let *request* be a new request, initialized as follows:

method

"POST"

url

*violation*'s url

origin

*violation*'s global object's origin

window

"no-window"

client

*violation*'s global object's relevant settings object

destination

""

initiator

""

type

""

cache mode

"no-cache"

credentials mode

"same-origin"

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"

5. [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.

3. 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 [\[OOB-REPORTING\]](#)'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 whitelist 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://not-example.com"></iframe>
<script>
  var blockedWorker = new Worker("data:application/javascript,...");
</script>
```

### 6.1.1.1. *child-src Pre-request check*<sup>§</sup>

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 [§6.6.1.7 Get the effective directive for request](#) on *request*.
2. If *name* is not `frame-src` or `worker-src`, return "Allowed".
3. If *policy* contains a directive whose [name](#) is *name*, return "Allowed".
4. 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*<sup>§</sup>

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 [§6.6.1.7 Get the effective directive for request](#) on *request*.
2. If *name* is not `frame-src` or `worker-src`, return "Allowed".
3. If *policy* contains a directive whose [name](#) is *name*, return "Allowed".
4. 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*<sup>§</sup>

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://not-example.com">...
<script>
  var xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://not-example.com/');
  xhr.send();

  var ws = new WebSocket("https://not-example.com/");

  var es = new EventSource("https://not-example.com/");

  navigator.sendBeacon("https://not-example.com/", { ... });
</script>
```

### 6.1.2.1. *connect-src Pre-request check*<sup>§</sup>

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [initiator](#) is "fetch", or its [type](#) is "" and [destination](#) is "subresource":
  1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.2.2. *connect-src Post-request check*<sup>§</sup>

This directive's [post-request check](#) is as follows:

Given a [request](#) (*request*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [initiator](#) is "fetch", or its [type](#) is "" and [destination](#) is "subresource":

1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response, request*, and this directive's [value](#) is "Does Not Match", return "Blocked".


3. Return "Allowed".

6.1.3. **default-src**<sup>§</sup>

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';  
object-src 'self';  
script-src 'self';  
style-src '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 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';
                        object-src 'self';
                        script-src https://example.com;
                        style-src '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 [§6.6.1.7 Get the effective directive for request](#) on *request*.
2. If *name* is null, return " Allowed".
3. If *policy* contains a [directive](#) whose [name](#) is *name*, return " Allowed".
4. If *name* is "frame-src" or "worker-src", and *policy* contains a [directive](#) whose [name](#) is "child-src", return "Allowed".

Note: It would be lovely to remove this special case. Perhaps "effective directive" could return "child-src" and that could delegate out in the same way this algorithm does?

5. Otherwise, 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 [§6.6.1.7 Get the effective directive for request](#) on *request*.
2. If *name* is null, return " Allowed".
3. If *policy* contains a [directive](#) whose [name](#) is *name*, return "                   ".

Allowed

4. If *name* is "frame-src" or "worker-src", and *policy* contains a [directive](#) whose [name](#) is "child-src", return "Allowed".

Note: It would be lovely to remove this special case. Perhaps "effective directive" could return "child-src" and that could delegate out in the same way this algorithm does?

5. Otherwise, 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.

#### 6.1.4. 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://not-example.com/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. Assert: *policy* is unused.
2. If *request*'s [type](#) is "font":

1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. 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. Assert: *policy* is unused.
2. If *request*'s [type](#) is "font":
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. 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://not-example.com/">
</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. Assert: *policy* is unused.
2. If *request*'s *type* is "document" and *target browsing context* is a *nested browsing context*:
  1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's *value* is "Does Not Match", return "Blocked".
3. Return " Allowed".

#### 6.1.5.2. *frame-src Post-request check*<sup>§</sup>

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

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

1. Assert: *policy* is unused.
2. If *request*'s *type* is "document" and *target browsing context* is a *nested browsing context*:
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's *value* is "Does Not Match", return "Blocked".
3. Return " Allowed".

#### 6.1.6. *img-src*<sup>§</sup>

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 *type* 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* :

```

```

#### 6.1.6.1. *img-src Pre-request check*<sup>§</sup>

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "image":
  1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.6.2. *img-src* Post-request check<sup>§</sup>

This directive's [post-request check](#) is as follows:

Given a [request](#) (*request*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "image":
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.7. *manifest-src*<sup>§</sup>

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://not-example.com/manifest">
```

### 6.1.7.1. *manifest-src* Pre-request check<sup>§</sup>

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "", and its [initiator](#) is "manifest":
  1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.7.2. *manifest-src* Post-request check<sup>§</sup>

This directive's [post-request check](#) is as follows:

Given a [request](#) (*request*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "", and its [initiator](#) is "manifest":
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

## 6.1.8. *media-src*<sup>§</sup>

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://not-example.com/audio"></audio>
<video src="https://not-example.com/video">
  <track kind="subtitles" src="https://not-example.com/subtitles">
```

```
</video>
```

### 6.1.8.1. *media-src Pre-request check*<sup>§</sup>

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is one of " audio", "video", or "track":
  1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.8.2. *media-src Post-request check*<sup>§</sup>

This directive's [post-request check](#) is as follows:

Given a [request](#) (*request*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is one of " audio", "video", or "track":
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

## 6.1.9. *object-src*<sup>§</sup>

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 14

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://not-example.com/flash"></embed>
<object data="https://not-example.com/flash"></object>
<applet archive="https://not-example.com/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.

#### 6.1.9.1. *object-src Pre-request check*<sup>§</sup>

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "", and its [destination](#) is "unknown":
  1. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return "Allowed".

#### 6.1.9.2. *object-src Post-request check*<sup>§</sup>

This directive's [post-request check](#) is as follows:

Given a [request](#) (*request*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "", and its [destination](#) is "unknown":
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return "Allowed".

### 6.1.10. 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:

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

The script-src directive governs four 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 [§4.2.3 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 whitelisting "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".

#### 6.1.10.1. script-src Pre-request check§

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "script", and its [destination](#) is "subresource":
  1. If the result of executing [§6.1.2 Does nonce match source list?](#) on *request*'s [cryptographic nonce metadata](#) and this directive's [value](#) is "Matches", return "Allowed".
  2. If this directive's [value](#) contains one or more [source expressions](#) that match the [hash-source](#) grammar, and *request*'s [integrity metadata](#) is not the empty string, then:

1. Let *integrity sources* be the result of executing the [Subresource Integrity §parse-metadata](#) algorithm on *request*'s [integrity metadata](#) . [\[SRI\]](#)
2. Assert: *integrity sources* is not "no metadata".
3. Let *bypass due to integrity match* be true.
4. For each *source* in *integrity sources*:
  1. If this 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](#) whitelisted by this directive. We rely on the browser's enforcement of Subresource Integrity [\[SRI\]](#) to block non-matching resources upon response.

3. If this 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"](#) .

4. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".

3. Return " Allowed".

### 6.1.10.2. *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. Assert: *policy* is unused.
2. If *request*'s [type](#) is "script", and its [destination](#) is "subresource":
  1. If the result of executing [§6.6.1.2 Does nonce match source list?](#) on *request*'s [cryptographic nonce metadata](#) and this directive's [value](#) is "Matches", return "Allowed".
  2. Assert: This directive's [value](#) does not contain ["strict-dynamic"](#), or *request*'s [parser metadata](#) is not

"parser-inserted".

3. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".

3. Return " Allowed".

### 6.1.10.3. *script-src Inline Check*<sup>§</sup>

This directive's [inline check](#) algorithm is as follows:

Given an [Element](#) (*element*), a string ( *type*), and a string ( *source*):

1. If *type* is "script attribute":
  1. If *list* contains a [source expression](#) which is an [ASCII case-insensitive match](#) for the [keyword-source](#) "'strict-dynamic'", and does not contain a [source expression](#) which is an [ASCII case-insensitive match](#) for the [keyword-source](#) "'unsafe-hashed-attributes'", return " Blocked".
  2. If the result of executing [§6.6.2.1 Does element match source list for type and source?](#) on *element*, this directive's [value](#), *type*, and *source*, is "Does Not Match", return " Blocked".
2. If *type* is "script":
  1. If *list* contains a [source expression](#) which is an [ASCII case-insensitive match](#) for the [keyword-source](#) "'strict-dynamic'", return "Blocked".

Note: "'strict-dynamic'" is explained in more detail in [§8.2 Usage of "'strict-dynamic'"](#).

  2. If the result of executing [§6.6.2.1 Does element match source list for type and source?](#) on *element*, this directive's [value](#), *type*, and *source*, is "Does Not Match", return " Blocked".
3. Return " Allowed".

### 6.1.11. *style-src*<sup>§</sup>

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 \[RFC5988\]](#).
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 [§4.2.3 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 whitelisting "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 [\[CSSOM\]](#) [\[HTML5\]](#).

**ISSUE 8**  This needs to be better explained.

#### 6.1.11.1. *style-src Pre-request Check*<sup>§</sup>

This directive’s [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*’s [type](#) is "style":
  1. If the result of executing [§6.6.1.2 Does nonce match source list?](#) on *request*’s [cryptographic nonce metadata](#) and this directive’s [value](#) is "Matches", return "Allowed".
  2. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive’s [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

#### 6.1.11.2. *style-src Post-request Check*<sup>§</sup>

This directive’s [post-request check](#) is as follows:

Given a [request](#) (*request*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [type](#) is "style":
  1. If the result of executing [§6.6.1.2 Does nonce match source list?](#) on *request*'s [cryptographic nonce metadata](#) and this directive's [value](#) is "Matches", return "Allowed".
  2. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.11.3. *style-src Inline Check*<sup>§</sup>

This directive's [inline check](#) algorithm is as follows:

Given an [Element](#) (*element*), a string ( *type*), and a string ( *source*):

1. If *type* is "style" or "style attribute":
  1. If the result of executing [§6.6.2.1 Does element match source list for type and source?](#) on *element*, this directive's [value](#), *type*, and *source*, is "Does Not Match", return " Blocked".
2. Return " Allowed".

This directive's [initialization](#) algorithm is as follows:

**ISSUE 9** <sup>¶</sup> 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.

### 6.1.12. *worker-src*<sup>§</sup>

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
```

#### <sup>¶</sup>EXAMPLE 15

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://not-example.com/");
  navigator.serviceWorker.register('https://not-example.com/sw.js');
</script>
```

### 6.1.12.1. *worker-src Pre-request Check*§

This directive's [pre-request check](#) is as follows:

Given a [request](#) (*request*) and a [policy](#) (*policy*):

1. Assert: *policy* is unused.
2. If *request*'s [destination](#) is one of "serviceworker", "sharedworker", or "worker":
  4. If the result of executing [§6.6.1.3 Does request match source list?](#) on *request* and this directive's [value](#) is "Does Not Match", return "Blocked".
3. Return " Allowed".

### 6.1.12.2. *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. Assert: *policy* is unused.
2. If *request*'s [destination](#) is one of "serviceworker", "sharedworker", or "worker":
  1. If the result of executing [§6.6.1.4 Does response to request match source list?](#) on *response*, *request*, and this directive's [value](#) is "Does Not Match", return "Blocked".
3. 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?*<sup>§</sup>

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 [§6.6.1.5 Does url match source list in origin with redirect count?](#) on *base*, *source list*, *document*'s [relevant settings object](#)'s origin, and 0 is "Does Not Match":
    1. Let *violation* be the result of executing [§2.3.1 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".
2. Return "Allowed".

### 6.2.2. *plugin-types*<sup>§</sup>

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 *( RWS 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 16

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"></object>

<!-- 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"></object>
```

#### 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`<sup>§</sup>

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 \*( RWS 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*<sup>§</sup>

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.

### 6.2.4. *disown-opener*§

The *disown-opener* directive ensures that a resource will [disown its opener](#) when navigated to. The directive's syntax is described by the following ABNF grammar:

```
directive-name = "disown-opener"
directive-value = ""
```

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.

**ISSUE 10** Not sure this is the right model. We need to ensure that we take care of [the inverse](#) as well, and there might be a cleverer syntax that could encompass both a document's opener, and a document's openees. *disown-openees* is weird. Maybe *disown 'opener' 'openees'*? Do we need origin restrictions on either/both?

#### 6.2.4.1. *disown-opener Initialization*§

This directive's [initialization](#) algorithm is as follows:

Given a [Document](#) or [global object](#) (*context*), a [response](#) (*response*), and a [policy](#) (*policy*):

1. Assert: *response* and *policy* are unused.

2. If *context*'s [responsible browsing context](#) has an [opener browsing context](#), [disown its opener](#).

**ISSUE 11** What should this do in an `<iframe>`? Anything?

## 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 ( *type*, "form-submission" or "other") and two [browsing contexts](#) (*source* and *target*), 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 `form-action` directive's [pre-navigation check](#):

1. Assert: *source* and *target* are unused in this algorithm, as `form-action` is concerned only with details of the outgoing request.
2. If *type* is "form-submission":
  1. If the result of executing [§6.6.1.3 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 *( RWS ancestor-source ) ) / "'none'"
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 `response` (*navigation response*) and two `browsing contexts` (*source* and *target*), 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`'s `navigation response check`:

1. Assert: *request*, *navigation response*, and *source* are unused in this algorithm, as `frame-ancestors` is concerned only with *target*'s ancestors.
2. If *target* is not a `nested browsing context`, return "Allowed".
3. Let *current* be *target*.
4. 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 `unicode serialization` of *parent*'s `active document`'s `origin`.
  3. If §6.6.1.5 `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".
5. Return "Allowed".

## 6.4. Reporting Directives§

Various algorithms in this document hook into the reporting process by constructing a `violation` object via §2.3.2 `Create a violation object for request, policy, and directive` or §2.3.1 `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 17

`Content-Security-Policy: ...; report-uri https://endpoint.com; report-to groupname`

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 *( RWS 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 [\[OOB-REPORTING\]](#). 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. URL Matching

### 6.6.1.1. Does request violate policy?<sup>§</sup>

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*.

### 6.6.1.2. Does nonce match source list?<sup>§</sup>

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".

### 6.6.1.3. Does request match source list?<sup>§</sup>

Given a [request](#) (*request*), and a [source list](#) (*source list*), this algorithm returns the result of executing [§6.6.1.5 Does url match source list in origin with redirect count?](#) on *request*'s [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.

### 6.6.1.4. Does response to request match source list?<sup>§</sup>

Given a [request](#) (*request*), and a [source list](#) (*source list*), this algorithm returns the result of executing [§6.6.1.5 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.

#### 6.6.1.5. 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 [§6.6.1.6 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".

#### 6.6.1.6. 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' ", if instance, will have distinct meaning depending on that bit of context.

1. If *expression* is the string "\*", and *url*'s [scheme](#) is a [network scheme](#) , return "Matches".

Note: This logic means that in order to allow resource from non-[network scheme](#) , it has to be explicitly whitelisted: default-src \* data: custom-scheme-1: custom-scheme-2:. In other words, there is no semantic representation of most permissive *expression*.

2. If *expression* matches the [scheme-source](#) or [host-source](#) grammar:
  1. If *expression* has a [scheme-part](#) that is not an [ASCII case-insensitive match](#) for *url*'s [scheme](#), then

return "Does Not Match" unless one of the following conditions is met:

1. *expression*'s [scheme-part](#) is an [ASCII case-insensitive match](#) for "http" and *url*'s [scheme](#) is "https"
  2. *expression*'s [scheme-part](#) is an [ASCII case-insensitive match](#) for "ws" and *url*'s [scheme](#) is "wss", "http" or "https"
  3. *expression*'s [scheme-part](#) is an [ASCII case-insensitive match](#) for "wss" and *url*'s [scheme](#) is "https"
2. If *expression* matches the [scheme-source](#) grammar, return "Matches".

Note: This logic effectively means that `script-src http:` is equivalent to `script-src http: https:`, and `script-src http://example.com/` is equivalent to `script-src http://example.com https://example.com`. As well as WebSocket schemes are equivalent to corresponding HTTP schemes. In short, we always allow a secure upgrade from an explicitly insecure expression.

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](#), then return "Does Not Match" unless one of the following conditions is met:
    1. *origin*'s [scheme](#) is *url*'s [scheme](#)
    2. *origin*'s [scheme](#) is "http", and *url*'s [scheme](#) one of "https", "ws", or "wss".
    3. *origin*'s [scheme](#) is "https", and *url*'s [scheme](#) is "wss".

Note: As with [scheme-part](#) above, we allow schemeless [host-source](#) expressions to be upgraded from insecure schemes to secure schemes.

3. If the first character of *expression*'s [host-part](#) is an U+002A ASTERISK character (\*):
1. Let *remaining* be the result of removing the leading " \*" from *expression*.
  2. If *remaining* (including the leading U+002E FULL STOP character (.)) is not an [ASCII case-insensitive match](#) for the rightmost characters of *url*'s [host](#), then return "Does Not Match".
4. If the first character of *expression*'s [host-part](#) is not an U+002A ASTERISK character (\*), and *url*'s [host](#) is not an [ASCII case-insensitive match](#) for *expression*'s [host-part](#), return "Does Not Match".
5. If *expression*'s [host-part](#) matches the [IPv4address](#) rule from [\[RFC3986\]](#), and is not "127.0.0.1"; or if *expression*'s [host-part](#) 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.

6. If *expression* does not contain a [port-part](#), and *url*'s [port](#) is not the [default port](#) for *url*'s [scheme](#), return "Does Not Match".
7. If *expression* does contain a [port-part](#), return "Does Not Match" unless one of the following conditions is met:
  1. *expression*'s [port-part](#) is "\*".
  2. *expression*'s [port-part](#) is the same number as *url*'s [port](#).
  3. *expression*'s [port-part](#) is 80, and *url*'s [port](#) is 443.
8. If *expression* contains a non-empty [path-part](#), and *redirect count* is 0, then:
  1. Let *exact match* be false if the final character of *expression*'s [path-part](#) is the U+002F SOLIDUS character (/), and true otherwise.
  2. Let *path list* be the result of [strictly splitting](#) *expression*'s [path-part](#) on the U+002F SOLIDUS character (/).
  3. If *path list* has more items than *url*'s [path](#), return "Does Not Match".
  4. If *exact match* is true, and *path list* does not have the same number of items as *url*'s [path](#), return "Does Not Match".
  5. For each *expression piece* in *path list*:
    1. Let *url piece* be the next item in *url*'s [path](#).
    2. [Percent decode](#) *expression piece*.
    3. [Percent decode](#) *url piece*.
    4. If *expression piece* is not a [case-sensitive](#) match for *url piece*, return "Does Not Match".
9. 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"

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".

6.6.1.7. Get the effective directive for request§

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

- 1. Switch on *request*'s [type](#), and execute the associated steps:

```
""
    1. If the request's initiator is "fetch", return connect-src.
    2. If the request's initiator is "manifest", return manifest-src.
    3. If the request's destination is "subresource", return connect-src.
    4. If the request's destination is "unknown", return object-src.
    5. If the request's destination is "document" and 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 image-src.

"style"
    1. Return style-src.

"script"
    1. Switch on request's destination, and execute the associated steps:

        "script"
        "subresource"
            1. Return script-src.

        "serviceworker"
        "sharedworker"
        "worker"
            1. Return worker-src.

    2. Return null.
```

6.6.2. Element Matching Algorithms§

6.6.2.1. 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".

1. Assert: *source* contains the value of a [<script>](#) element's [text](#) IDL attribute, the value of a [<style>](#) element's [textContent](#) IDL attribute, or the value of one of a [<script>](#) element's [event handler IDL attribute](#).

Note: This means that *source* will be interpreted with the encoding of the page in which it is embedded. See the integration points in [§4.2 Integration with HTML](#) for more detail.

2. If *type element* has an attribute whose name is an [ASCII case-insensitive match](#) for the string "<script", or the string "<style", then return "Does Not Match".
3. Let *contains nonce or hash* and *hashes match attributes* be false.
4. For each *expression* in *list*:
  1. If *expression* matches the [nonce-source](#) or [hash-source](#) grammar, set *contains nonce or hash* to true.
  2. If *expression* is an [ASCII case-insensitive match](#) for the [keyword-source](#) "'unsafe-hashed-attributes'", set *hashes match attributes* to true.
5. If *contains nonce or hash* is false, and *list* contains a [source expression](#) which is an [ASCII case-insensitive match](#) for the string "unsafe-inline", then return "Matches".

Note: This logic means that if *list* contains both "unsafe-inline" and either [nonce-source](#) or [hash-source](#) "unsafe-inline" will have no effect.

6. If *type* is "script" or "style":
  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.

7. If *type* is "script" or "style", or *hashes match attributes* is true:
  1. 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. If *actual* is a [case-sensitive](#) match for *expression*'s [base64-value](#) part, return "Matches".

Note: Hashes apply to inline [<script>](#) and inline [<style>](#). If the ["unsafe-hashed-attributes"](#) source expression is present, they will also apply to event handlers and style attributes.

8. Return "Does Not Match".

## 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 whitelist 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. 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.

## 7.3. 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 whitelists `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.

## 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 18

```
Content-Security-Policy: default-src 'self' http://example.com http://example.net;
                        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'"§

Whitelists 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 whitelists can enable, and though CSP is capable of mitigating these bypasses via extensive whitelists, those 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 reasonably secure whitelist.

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-source](#) s" 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-abcdefg'` in browsers that support CSP2, and `'nonce-abcdefg' '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 19

Suppose MegaCorp, Inc. deploys the following policy:

[Content-Security-Policy](#): [script-src](#) 'nonce-abcdefg' ['strict-dynamic'](#)

And serves the following HTML with that policy active:

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

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-hashed-attributes'"§

*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 whitelisting '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-hashed-attributes'" source expression aims to make CSP deployment simpler and safer in these situations by allowing developers to whitelist specific handlers via hashes.

### EXAMPLE 20

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

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

Rather than whitelisting "'unsafe-inline'", they decide to use "'unsafe-hashed-attributes'" along with a hash source expression, as follows:

```
Content-Security-Policy: 'unsafe-hashed-attributes' 'sha256-
jzgBGA4UWFFmpOBq0JpdsySukE1FrEN5bUpoK8Z29fY='
```

## 8.4. Whitelisting external JavaScript with hashes§

*This section is not normative.*

In [CSP2], hash source expressions could only whitelist inlined script, but now that Subresource Integrity 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.

### EXAMPLE 21

MegaCorp, Inc. wishes to whitelist 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 whitelisted 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 be whitelisted because they contain 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>
```

## 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](#) )

#### disown-opener

This document (see [§6.2.4 disown-opener](#) )

#### 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 [§6.1.9 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 [§6.1.10 script-src](#) )

[style-src](#)

This document (see [§6.1.11 style-src](#) )

[worker-src](#)

This document (see [§6.1.12 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 22

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§

[ancestor-source](#), in §6.3.2

[ancestor-source-list](#), in §6.3.2

[base64-value](#), in §2.2.1

[base-uri](#), in §6.2.1

blockedURI

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[child-src](#), in §6.1.1

[column number](#), in §2.3

columnNumber

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[connect-src](#), in §6.1.2

[Content-Security-Policy](#), in §3.1



[Content Security Policy](#) , in §1

[Content-Security-Policy-Report-Only](#) , in §3.2

[CSP list](#) , in §4.2

[default-src](#) , in §6.1.3

[directive-name](#) , in §2.2

[directives](#) , in §2.2

[directive set](#) , in §2.1

[directive-value](#) , in §2.2

[disown-opener](#) , in §6.2.4

[disposition](#) , in §2.1

documentURI

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

effective directive

[dfn for violation](#), in §2.3

[dfn for request](#), in §6.6.1.7

effectiveDirective

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[embedding document](#) , in §4.2

[enforced](#) , in §4.2

[EnsureCSPDoesNotBlockStringCompilation\(callerRealm, calleeRealm\)](#), in §4.3

[Fetch directives](#) , in §6.1

[font-src](#) , in §6.1.4

[form-action](#) , in §6.3.1

[frame-ancestors](#) , in §6.3.2

[frame-src](#) , in §6.1.5

[global object](#) , in §2.3

[hash-algorithm](#) , in §2.2.1

[hash-source](#) , in §2.2.1

[host-char](#) , in §2.2.1

[host-part](#) , in §2.2.1

[host-source](#) , in §2.2.1

[img-src](#) , in §6.1.6

[initialization](#), in §2.2

[inline check](#), in §2.2

[keyword-source](#), in §2.2.1

[line number](#), in §2.3

lineNumber

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[manifest-src](#), in §6.1.7

[media-src](#), in §6.1.8

[media-type](#), in §6.2.2

[media-type-list](#), in §6.2.2

[monitored](#), in §4.2

[name](#), in §2.2

[navigation response check](#), in §2.2

[nonce-source](#), in §2.2.1

['none'](#), in §2.2.1

[object-src](#), in §6.1.9

originalPolicy

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[parse a serialized CSP](#), in §2.1

[path-part](#), in §2.2.1

[plugin-types](#), in §6.2.2

[plugin-types Post-Request Check](#), in §6.2.2

policy

[definition of](#), in §2.1

[dfn for violation](#), in §2.3

[port-part](#), in §2.2.1

[post-request check](#), in §2.2

[pre-navigation check](#), in §2.2

[pre-request check](#), in §2.2

referrer

[dfn for violation](#), in §2.3

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[report-to](#) , in §6.4.2

[report-uri](#) , in §6.4.1

[resource](#) , in §2.3

[response check](#) , in §2.2

[sandbox](#) , in §6.2.3

[scheme-part](#) , in §2.2.1

[scheme-source](#) , in §2.2.1

[script-src](#) , in §6.1.10

[SecurityPolicyViolationEvent](#) , in §5.1

[SecurityPolicyViolationEventInit](#) , in §5.1

[SecurityPolicyViolationEvent\(type\)](#) , in §5.1

[SecurityPolicyViolationEvent\(type, eventInitDict\)](#) , in §5.1

['self'](#) , in §2.2.1

[serialized CSP](#) , in §2.1

[serialized directive](#) , in §2.2

[serialized-directive](#) , in §2.2

[serialized-policy](#) , in §2.1

[serialized source list](#) , in §2.2.1

[serialized-source-list](#) , in §2.2.1

[Should plugin element be blocked a priori by Content Security Policy?:](#) , in §6.2.2.1

[source-expression](#) , in §2.2.1

[source expression](#) , in §2.2.1

[source file](#) , in §2.3

sourceFile

[attribute for SecurityPolicyViolationEvent](#) , in §5.1

[dict-member for SecurityPolicyViolationEventInit](#) , in §5.1

[source lists](#) , in §2.2.1

[status](#) , in §2.3

statusCode

[attribute for SecurityPolicyViolationEvent](#) , in §5.1

[dict-member for SecurityPolicyViolationEventInit](#) , in §5.1

['strict-dynamic'](#) , in §2.2.1

[style-src](#) , in §6.1.11

['unsafe-eval'](#) , in §2.2.1

['unsafe-hashed-attributes'](#) , in §2.2.1

['unsafe-inline'](#) , in §2.2.1

[url](#), in §2.3

[value](#) , in §2.2

violatedDirective

[attribute for SecurityPolicyViolationEvent](#), in §5.1

[dict-member for SecurityPolicyViolationEventInit](#), in §5.1

[violation](#) , in §2.3

[violation report](#) , in §5

[worker-src](#) , in §6.1.12

## Terms defined by reference§

[CSSOM] defines the following terms:

[insert a css rule](#)

[parse a css declaration block](#)

[parse a css rule](#)

[parse a group of selectors](#)

[ECMA262] defines the following terms:

[Function\(\)](#)

[HostEnsureCanCompileStrings\(\)](#)

[JSON.stringify\(\)](#)

[eval\(\)](#)

[realm](#)

[FETCH] defines the following terms:

[body](#)

[cache mode](#)

[client](#)

[credentials mode](#)

[cryptographic nonce metadata](#)

[csp list](#)

[current url](#)

[destination](#)

[extracting a mime type](#)

[fetch](#)

[header list](#)

[http fetch](#)

[http-network fetch](#)

- [initiator](#)
- [integrity metadata](#)
- [main fetch](#)
- [method](#)
- [network error](#)
- [origin](#)
- [parse a header value](#)
- [parser metadata](#)
- [redirect count](#)
- [redirect mode](#)
- [request](#)
- [response](#)
- [target browsing context](#)
- [type](#)
- [url](#)
- [window](#)

[HTML] defines the following terms:

- [content-security-policy http-equiv processing instructions](#)
- [initialising a new document object](#)
- [nonce](#)
- [ping](#)
- [process a navigate fetch](#)
- [process a navigate response](#)
- [realm's global object](#)
- [run a worker](#)
- [the worker's documents](#)
- [update a style block](#)

[HTML5] defines the following terms:

- [Window](#)
- [a](#)
- [active document](#)
- [an iframe srcdoc document](#)
- [applet](#)
- [ascii case-insensitive match](#)
- [base](#)
- [browsing context](#)
- [case-sensitive](#)
- [collect a sequence of characters](#)
- [content](#)
- [csp list](#)
- [current settings object](#)

- [disown its opener](#)
- [embed](#)
- [event handler idl attributes](#)
- [forced sandboxing flag set](#)
- [frame](#)
- [global object](#)
- [http-equiv](#)
- [iframe](#)
- [link](#)
- [meta](#)
- [nested browsing context](#)
- [nested through](#)
- [object](#)
- [opener browsing context](#)
- [parent browsing context](#)
- [parse a sandboxing directive](#)
- [parser-inserted](#)
- [prepare a script](#)
- [referrer](#)
- [relevant global object](#)
- [relevant settings object](#)
- [responsible browsing context](#)
- [sandboxed origin browsing context flag](#)
- [sandboxed scripts browsing context flag](#)
- [script](#)
- [set the frozen base url](#)
- [space characters](#)
- [split a string on commas](#)
- [split a string on spaces](#)
- [strictly split a string](#)
- [strip leading and trailing whitespace](#)
- [style](#)
- [unicode serialization](#)
- [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](#)

[scheme](#)

[uri-reference](#)

[rfc4648] defines the following terms:

[base64 encoding](#)

[RFC5234] defines the following terms:

[alpha](#)

[digit](#)

[vchar](#)

[rfc6454] defines the following terms:

[origin](#)

[rfc7230] defines the following terms:

[ows](#)

[rws](#)

[token](#)

[rfc7231] defines the following terms:

[representation](#)

[resource representation](#)

[service-workers] defines the following terms:

[ServiceWorker](#)

[SHA2] defines the following terms:

[sha-256](#)

[sha-384](#)

[sha-512](#)

[WHATWG-URL] defines the following terms:

[URL](#)

[default port](#)

[host](#)

[ipv6 address](#)

[local scheme](#)

[network scheme](#)

[origin](#)

[path](#)

[percent decode](#)

[port](#)

[scheme](#)

[url parser](#)

[url serializer](#)

[workers] defines the following terms:

[Worker](#)

[css-cascade-3] defines the following terms:

[@import](#)

[WHATWG-DOM] defines the following terms:

[Document](#)

[Element](#)

[Event](#)

[EventInit](#)

[ascii case-insensitive](#)

[fire an event](#)

[node document](#)

[textContent](#)

[HTML] defines the following terms:

[SharedWorker](#)

[WorkerGlobalScope](#)

[data](#)

[document](#)

[href](#)

[sandbox](#)

[setInterval\(\)](#)

[setTimeout\(\)](#)

[text](#)

[type](#)

## References§

### Normative References§

#### [CSS-CASCADE-3]

Elika Etemad; Tab Atkins Jr.. [CSS Cascading and Inheritance Level 3](#). 19 May 2016. CR. URL: <https://www.w3.org/TR/css-cascade-3/>

#### [CSSOM]

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

#### [ECMA262]

Brian Terlson; Allen Wirfs-Brock. [ECMAScript® Language Specification](#). URL: <https://tc39.github.io/ecma262/>

#### [FETCH]

Anne van Kesteren. [Fetch Standard](#). Living Standard. URL: <https://fetch.spec.whatwg.org/>

#### [HTML]

Ian Hickson. [HTML Standard](#). Living Standard. URL: <https://html.spec.whatwg.org/multipage/>



## [HTML5]

Ian Hickson; et al. [HTML5](#). 28 October 2014. REC. URL: <https://www.w3.org/TR/html5/>

## [OOB-REPORTING]

Ilya Gregorik; Mike West. [Out-of-band Reporting](#) . URL: <https://mikewest.github.io/error-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>

## [RFC5988]

M. Nottingham. [Web Linking](#) . October 2010. Proposed Standard. URL: <https://tools.ietf.org/html/rfc5988>

## [RFC6454]

A. Barth. [The Web Origin Concept](#) . December 2011. Proposed Standard. URL: <https://tools.ietf.org/html/rfc6454>

## [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>

## [SERVICE-WORKERS]

Alex Russell; Jungkee Song; Jake Archibald. [Service Workers](#) . 25 June 2015. WD. URL: <https://www.w3.org/TR/service-workers/>

## [SHA2]

FIPS PUB 180-4, Secure Hash Standard . URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>

## [SRI]

Devdatta Akhawe; et al. [Subresource Integrity](#) . 23 June 2016. REC. URL: <https://www.w3.org/TR/SRI/>

## [WHATWG-DOM]

Anne van Kesteren. [DOM Standard](#) . Living Standard. URL: <https://dom.spec.whatwg.org/>

## [WHATWG-URL]

Anne van Kesteren. [URL Standard](#) . Living Standard. URL: <https://url.spec.whatwg.org/>

## [WORKERS]

Ian Hickson. [Web Workers](#) . 24 September 2015. WD. URL: <https://www.w3.org/TR/workers/>

## Informative References§

### [APPMANIFEST]

Marcos Caceres; et al. [Web App Manifest](#) . 12 August 2016. WD. URL: <https://www.w3.org/TR/appmanifest/>

### [BEACON]

Ilya Grigorik; et al. [Beacon](#) . 29 June 2016. WD. URL: <https://www.w3.org/TR/beacon/>

### [CSP2]

Mike West; Adam Barth; Daniel Veditz. [Content Security Policy Level 2](#) . 21 July 2015. CR. 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/>

### [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](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: [http://www.contextis.com/documents/2/Browser\\_Timing\\_Attacks.pdf](http://www.contextis.com/documents/2/Browser_Timing_Attacks.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§

```
[Constructor(DOMString type, optional SecurityPolicyViolationEventInit eventInitDict)]
interface SecurityPolicyViolationEvent : Event {
    readonly attribute DOMString documentURI;
    readonly attribute DOMString referrer;
    readonly attribute DOMString blockedURI;
    readonly attribute DOMString violatedDirective;
    readonly attribute DOMString effectiveDirective;
    readonly attribute DOMString originalPolicy;
    readonly attribute DOMString sourceFile;
    readonly attribute unsigned short statusCode;
    readonly attribute long lineNumber;
    readonly attribute long columnNumber;
};

dictionary SecurityPolicyViolationEventInit : EventInit {
    DOMString documentURI;
    DOMString referrer;
    DOMString blockedURI;
    DOMString violatedDirective;
    DOMString effectiveDirective;
    DOMString originalPolicy;
    DOMString sourceFile;
    unsigned short statusCode;
    long lineNumber;
    long columnNumber;
};
```

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>](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>](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>](https://github.com/whatwg/html/issues/968)

**ISSUE 6** This hook is missing from W3C's HTML. [<https://github.com/w3c/html/issues/547>](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>](https://github.com/w3c/html/issues/548)

**ISSUE 8** This needs to be better explained. \_\_

**ISSUE 9** 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 10** Not sure this is the right model. We need to ensure that we take care of [the inverse](#) as well, and there might be a cleverer syntax that could encompass both a document's opener, and a document's openees. disown-openee is weird. Maybe disown 'opener' 'openees'? Do we need origin restrictions on either/both? \_\_

**ISSUE 11** What should this do in an [<iframe>](#)? Anything? \_\_