Django-CSP Documentation

Release 4.0

James Socol, Mozilla

Nov 14, 2025

CONTENTS

django-csp adds Content-Security-Policy headers to Django applications.

Version
4.0

Code
https://github.com/mozilla/django-csp

License
BSD; see LICENSE file

Issues
https://github.com/mozilla/django-csp/issues

Contents:

CHAPTER

ONE

INSTALLING DJANGO-CSP

First, install django-csp via pip or from source:

```
pip install django-csp
```

Add the csp app to your `INSTALLED_APPS` in your project's `settings` module:

```
INSTALLED_APPS = (
    # ...
    "csp",
    # ...
)
```

Now edit your project's `settings` module, to add the django-csp middleware to `MIDDLEWARE`, like so:

```
MIDDLEWARE = (
    # ...
    "csp.middleware.CSPMiddleware",
    # ...
)
```

Note

Middleware order does not matter unless you have other middleware modifying the CSP header, or requires CSP features like a nonce. See Using the generated CSP nonce for further advice on middleware order.

That should do it! Go on to *configuring CSP*.

CHAPTER

TWO

CONFIGURING DJANGO-CSP

Content-Security-Policy is a complicated header. There are many values you may need to tweak here.

It's worth reading the latest CSP spec and making sure you understand it before configuring django-csp.

> **Note**
>
> Many settings require a `tuple` or `list`. You may get very strange policies and even errors when mistakenly configuring them as a `string`.

2.1 Migrating from django-csp <= 3.8

Version 4.0 of django-csp introduces a new configuration format that breaks compatibility with previous versions. If you are migrating from django-csp 3.8 or lower, you will need to update your settings to the new format. See the *migration guide* for more information.

2.2 Configuration

All configuration of django-csp is done in your Django settings file with the `CONTENT_SECURITY_POLICY` setting or the `CONTENT_SECURITY_POLICY_REPORT_ONLY` setting. Each of these settings expects a dictionary representing a policy.

The `CONTENT_SECURITY_POLICY` setting is your enforceable policy.

The `CONTENT_SECURITY_POLICY_REPORT_ONLY` setting is your report-only policy. This policy is used to test the policy without breaking the site. It is useful when setting this policy to be slightly more strict than the default policy to see what would be blocked if the policy was enforced.

The following is an example of a policy configuration with a default policy and a report-only policy. The default policy is considered a "relaxed" policy that allows for the most flexibility while still providing a good level of security. The report-only policy is considered a step towards a more slightly strict policy and is used to test the policy without breaking the site.

```
from csp.constants import NONE, SELF

CONTENT_SECURITY_POLICY = {
    "EXCLUDE_URL_PREFIXES": ["/excluded-path/"],
    "DIRECTIVES": {
        "default-src": [SELF, "cdn.example.net"],
        "frame-ancestors": [SELF],
        "form-action": [SELF],
```

(continues on next page)

(continued from previous page)

```
        "report-uri": "/csp-report/",
    },
}

CONTENT_SECURITY_POLICY_REPORT_ONLY = {
    "EXCLUDE_URL_PREFIXES": ["/excluded-path/"],
    "DIRECTIVES": {
        "default-src": [NONE],
        "connect-src": [SELF],
        "img-src": [SELF],
        "form-action": [SELF],
        "frame-ancestors": [SELF],
        "script-src": [SELF],
        "style-src": [SELF],
        "upgrade-insecure-requests": True,
        "report-uri": "/csp-report/",
    },
}
```

Note

In the above example, the constant `NONE` is converted to the CSP keyword `"'none'"` and is distinct from Python's `None` value. The CSP keyword `'none'` is a special value that signifies that you do not want any sources for this directive. The `None` value is a Python keyword that represents the absence of a value and when used as the value of a directive, it will remove the directive from the policy.

This is useful when using the `@csp_replace` decorator to effectively clear a directive from the base configuration as defined in the settings. For example, if the Django settings the `frame-ancestors` directive is set to a list of sources and you want to remove the `frame-ancestors` directive from the policy for this view:

```
from csp.decorators import csp_replace


@csp_replace({"frame-ancestors": None})
def my_view(request): ...
```

2.3 Policy Settings

At the top level of the policy dictionary, these are the keys that can be used to configure the policy.

EXCLUDE_URL_PREFIXES
: A `tuple` or `list` of URL prefixes to exclude from CSP protection. URLs beginning with any of these strings will not get the Content-Security-Policy response headers at all.

Warning

Excluding any path on your site will eliminate the benefits of CSP everywhere on your site. The typical browser security model for JavaScript considers all paths alike. A Cross-Site Scripting flaw on, e.g., `excluded-page/` can therefore be leveraged to access everything on the same origin.

REPORT_PERCENTAGE

Percentage of requests that should see the `report-uri` directive. Use this to throttle the number of CSP violation reports made to your `report-uri`. A **float** between 0.0 and 100.0 (0.0 = no reports at all, 100.0 = always report). Ignored if `report-uri` isn't set.

> **Note**
>
> To allow rate limiting, `csp.contrib.rate_limiting.RateLimitedCSPMiddleware` must be used instead of `csp.middleware.CSPMiddleware`. See *violation reporting* for more details.

DIRECTIVES

A dictionary of policy directives. Each key in the dictionary is a directive and the value is a list of sources for that directive. The following is a list of all the directives that can be configured.

> **Note**
>
> The CSP keyword values of `'self'`, `'unsafe-inline'`, `'strict-dynamic'`, etc. must be quoted! e.g.: `"default-src": ["'self'"]`. Without quotes they will not work as intended.
>
> New in version 4.0 are CSP keyword constants. Use these to minimize quoting mistakes and typos.
>
> The following CSP keywords are available:
>
> - `NONE = "'none'"`
> - `REPORT_SAMPLE = "'report-sample'"`
> - `SELF = "'self'"`
> - `STRICT_DYNAMIC = "'strict-dynamic'"`
> - `UNSAFE_ALLOW_REDIRECTS = "'unsafe-allow-redirects'"`
> - `UNSAFE_EVAL = "'unsafe-eval'"`
> - `UNSAFE_HASHES = "'unsafe-hashes'"`
> - `UNSAFE_INLINE = "'unsafe-inline'"`
> - `WASM_UNSAFE_EVAL = "'wasm-unsafe-eval'"`
>
> Example usage:

```
from csp.constants import NONE, SELF, STRICT_DYNAMIC

CONTENT_SECURITY_POLICY = {
    "DIRECTIVES": {
        # No sources allowed for default-src by using `csp.constants.NONE`.
        "default-src": [NONE],
        "script-src": [SELF, STRICT_DYNAMIC],
        "style-src": [SELF],
        # Using Python's `None` will not include the directive in the header.␣
↪Useful
        # to override previous settings or when using the decorators.
        "base-uri": None,
    }
}
```

> **Note**
>
> The CSP keyword `csp.constants.NONE` is distinct from Python's `None` value. The CSP keyword `'none'` is a special value that signifies that you do not want any sources for the directive. The `None` value is a Python keyword that represents the absence of a value and when used as the value of a directive, it will remove the directive from the header.

> **Note**
>
> Deprecated features of CSP in general have been moved to the bottom of this list.

> **Warning**
>
> The `'unsafe-inline'` and `'unsafe-eval'` sources are considered harmful and should be avoided. They are included here for completeness, but should not be used in production.

default-src
: Set the `default-src` directive. A `tuple` or `list` of values, e.g.: `("'self'", "cdn.example.net")`. *default=[“‘self’”]*

script-src
: Set the `script-src` directive. A `tuple` or `list`. *default=None*

script-src-attr
: Set the `script-src-attr` directive. A `tuple` or `list`. *default=None*

script-src-elem
: Set the `script-src-elem` directive. A `tuple` or `list`. *default=None*

img-src
: Set the `img-src` directive. A `tuple` or `list`. *default=None*

object-src
: Set the `object-src` directive. A `tuple` or `list`. *default=None*

media-src
: Set the `media-src` directive. A `tuple` or `list`. *default=None*

frame-src
: Set the `frame-src` directive. A `tuple` or `list`. *default=None*

font-src
: Set the `font-src` directive. A `tuple` or `list`. *default=None*

connect-src
: Set the `connect-src` directive. A `tuple` or `list`. *default=None*

style-src
: Set the `style-src` directive. A `tuple` or `list`. *default=None*

style-src-attr
: Set the `style-src-attr` directive. A `tuple` or `list`. *default=None*

style-src-elem
: Set the `style-src-elem` directive. A `tuple` or `list`. *default=None*

base-uri
Set the `base-uri` directive. A `tuple` or `list`. *default=None*

Note: This doesn't use `default-src` as a fall-back.

child-src
Set the `child-src` directive. A `tuple` or `list`. *default=None*

frame-ancestors
Set the `frame-ancestors` directive. A `tuple` or `list`. *default=None*

Note: This doesn't use `default-src` as a fall-back.

navigate-to
Set the `navigate-to` directive. A `tuple` or `list`. *default=None*

Note: This doesn't use `default-src` as a fall-back.

form-action
Set the `form-action` directive. A `tuple` or `list`. *default=None*

Note: This doesn't use `default-src` as a fall-back.

sandbox
Set the `sandbox` directive. A `tuple` or `list`. *default=None*

Note: This doesn't use `default-src` as a fall-back.

report-uri
Set the `report-uri` directive. A `tuple` or `list` of URIs. Each URI can be a full or relative URI. *default=None*

Note: This doesn't use `default-src` as a fall-back.

report-to
Set the `report-to` directive. A `string` describing a reporting group. *default=None*

See Section 1.2: https://w3c.github.io/reporting/#group

Also see this MDN note on `report-uri` and `report-to`.

manifest-src
Set the `manifest-src` directive. A `tuple` or `list`. *default=None*

worker-src
Set the `worker-src` directive. A `tuple` or `list`. *default=None*

require-sri-for
Set the `require-sri-for` directive. A `tuple` or `list`. *default=None*

Valid values: a `list` containing `'script'`, `'style'`, or both.

Spec: require-sri-for-known-tokens

upgrade-insecure-requests
Include `upgrade-insecure-requests` directive. A `boolean`. *default=False*

Spec: upgrade-insecure-requests

require-trusted-types-for
Include `require-trusted-types-for` directive. A `tuple` or `list`. *default=None*

Valid values: `["'script'"]`

trusted-types
Include `trusted-types` directive. A `tuple` or `list`. *default=None*

Valid values: a `list` of allowed policy names that may include `default` and/or `'allow-duplicates'`

2.3.1 Deprecated CSP settings

The following `DIRECTIVES` settings are still configurable, but are considered deprecated in terms of the latest implementation of the relevant spec.

`block-all-mixed-content`
Include `block-all-mixed-content` directive. A `boolean`. *default=False*

Related note on MDN.

Spec: block-all-mixed-content

`plugin-types`
Set the `plugin-types` directive. A `tuple` or `list`. *default=None*

Note: This doesn't use `default-src` as a fall-back.

Related note on MDN.

`prefetch-src`
Set the `prefetch-src` directive. A `tuple` or `list`. *default=None*

Related note on MDN.

Changing the Policy

The policy can be changed on a per-view (or even per-request) basis. See the *decorator documentation* for more details.

CHAPTER

THREE

DJANGO-CSP 4.0 MIGRATION GUIDE

3.1 Overview

In the latest version of *django-csp*, the format for configuring Content Security Policy (CSP) settings has been updated are are backwards-incompatible with prior versions. The previous approach of using individual settings prefixed with `CSP_` for each directive is no longer supported. Instead, all CSP settings are now consolidated into one of two dict-based settings: `CONTENT_SECURITY_POLICY` or `CONTENT_SECURITY_POLICY_REPORT_ONLY`.

3.2 Migrating from the Old Settings Format

3.2.1 Update *django-csp*

First, update the *django-csp* package to the latest version that supports the new settings format. You can do this by running:

```
pip install -U django-csp
```

3.2.2 Add the *csp* app to *INSTALLED_APPS*

In your Django project's *settings.py* file, add the *csp* app to the `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # ...
    "csp",
    # ...
]
```

3.2.3 Run the Django check command

This is optional but can help kick start the new settings configuration for you. Run the Django check command which will look for old settings and output a configuration in the new format:

```
python manage.py check
```

This can help you identify the existing CSP settings in your project and provide a starting point for migrating to the new format.

3.2.4 Identify Existing CSP Settings

Locate all the existing CSP settings in your Django project. These settings start with the `CSP_` prefix, such as `CSP_DEFAULT_SRC`, `CSP_SCRIPT_SRC`, `CSP_IMG_SRC`, etc.

3.2.5 Create the New Settings Dictionary

In your Django project's *settings.py* file, create a new dictionary called `CONTENT_SECURITY_POLICY` or `CONTENT_SECURITY_POLICY_REPORT_ONLY`, depending on whether you want to enforce the policy or only report violations, or both. Use the output from the Django check command as a starting point to populate this dictionary.

3.2.6 Migrate Existing Settings

Migrate your existing CSP settings to the new format by populating the `DIRECTIVES` dictionary inside the `CONTENT_SECURITY_POLICY` setting. The keys of the `DIRECTIVES` dictionary should be the CSP directive names in lowercase, and the values should be lists containing the corresponding sources. The Django check command output can help you identify the directive names and sources.

For example, if you had the following old settings:

```
CSP_DEFAULT_SRC = ["'self'", "*.example.com"]
CSP_SCRIPT_SRC = ["'self'", "js.cdn.com/example/"]
CSP_IMG_SRC = ["'self'", "data:", "example.com"]
CSP_EXCLUDE_URL_PREFIXES = ["/admin"]
```

The new settings would be:

```
from csp.constants import SELF

CONTENT_SECURITY_POLICY = {
    "EXCLUDE_URL_PREFIXES": ["/admin"],
    "DIRECTIVES": {
        "default-src": [SELF, "*.example.com"],
        "script-src": [SELF, "js.cdn.com/example/"],
        "img-src": [SELF, "data:", "example.com"],
    },
}
```

Note

The keys in the `DIRECTIVES` dictionary, the directive names, are in lowercase and use dashes instead of underscores to match the CSP specification.

Note

If you were using the `CSP_INCLUDE_NONCE_IN` setting, this has been removed in the new settings format.

Previously: You could use the `CSP_INCLUDE_NONCE_IN` setting to specify which directives in your Content Security Policy (CSP) should include a nonce.

Now: You can include a nonce in any directive by adding the `NONCE` constant from the `csp.constants` module to the list of sources for that directive.

For example, if you had `CSP_INCLUDE_NONCE_IN = ["script-src"]`, this should be updated to include the *NONCE* sentinel in the *script-src* directive values:

```
from csp.constants import NONCE, SELF

CONTENT_SECURITY_POLICY = {
    "DIRECTIVES": {
        "script-src": [SELF, NONCE],
        # ...
    },
}
```

Note

If you were using the `CSP_REPORT_PERCENTAGE` setting, this should be updated to be a float percentage between 0.0 and 100.0. For example, if you had `CSP_REPORT_PERCENTAGE = 0.1`, this should be updated to `10.0` to represent 10% of CSP errors will be reported:

```
CONTENT_SECURITY_POLICY = {
    "REPORT_PERCENTAGE": 10.0,
    "DIRECTIVES": {
        "report-uri": "/csp-report/",
        # ...
    },
}
```

3.2.7 Remove Old Settings

After migrating to the new settings format, remove all the old `CSP_` prefixed settings from your *settings.py* file.

3.2.8 Update the CSP decorators

If you are using the CSP decorators in your views, those will need to be updated as well. The decorators now accept a dictionary containing the CSP directives as an argument. For example:

```
from csp.decorators import csp_update


@csp_update({"default-src": ["another-url.com"]})
def my_view(request): ...
```

Additionally, each decorator now takes an optional `REPORT_ONLY` argument to specify whether the policy should be enforced or only report violations. For example:

```
from csp.constants import SELF
from csp.decorators import csp


@csp({"default-src": [SELF]}, REPORT_ONLY=True)
def my_view(request): ...
```

Due to the addition of the `REPORT_ONLY` argument and for consistency, the `csp_exempt` decorator now requires parentheses when used with and without arguments. For example:

```
from csp.decorators import csp_exempt


@csp_exempt()
@csp_exempt(REPORT_ONLY=True)
def my_view(request): ...
```

Look for uses of the following decorators in your code: `@csp`, `@csp_update`, `@csp_replace`, and `@csp_exempt`.

3.3 Migrating Custom Middleware

The *CSPMiddleware* has changed in order to support easier extension via subclassing.

The *CSPMiddleware.build_policy* and *CSPMiddleware.build_policy_ro* methods have been deprecated in 4.0 and replaced with a new method *CSPMiddleware.build_policy_parts*.

> **Note**
>
> The deprecated methods will be removed in 4.1.

Unlike the old methods, which returned the built CSP policy header string, *build_policy_parts* returns a dataclass that can be modified and updated before the policy is built. This allows custom middleware to modify the policy whilst inheriting behaviour from the base classes.

An existing custom middleware, such as this:

```
from django.http import HttpRequest, HttpResponseBase

from csp.middleware import CSPMiddleware, PolicyParts


class ACustomMiddleware(CSPMiddleware):

    def build_policy(self, request: HttpRequest, response: HttpResponseBase) -> str:
        config = getattr(response, "_csp_config", None)
        update = getattr(response, "_csp_update", None)
        replace = getattr(response, "_csp_replace", {})
        nonce = getattr(request, "_csp_nonce", None)

        # ... do custom CSP policy logic ...

        return build_policy(config=config, update=update, replace=replace, nonce=nonce)

    def build_policy_ro(self, request: HttpRequest, response: HttpResponseBase) -> str:
        config = getattr(response, "_csp_config_ro", None)
        update = getattr(response, "_csp_update_ro", None)
        replace = getattr(response, "_csp_replace_ro", {})
        nonce = getattr(request, "_csp_nonce", None)

        # ... do custom CSP report-only policy logic ...

        return build_policy(config=config, update=update, replace=replace, nonce=nonce)
```

can be replaced with this:

```
from django.http import HttpRequest, HttpResponseBase

from csp.middleware import CSPMiddleware, PolicyParts


class ACustomMiddleware(CSPMiddleware):

    def get_policy_parts(
        self,
        request: HttpRequest,
        response: HttpResponseBase,
        report_only: bool = False,
    ) -> PolicyParts:
        policy_parts = super().get_policy_parts(request, response, report_only)

        if report_only:
            ...  # do custom CSP report-only policy logic
        else:
            ...  # do custom CSP policy logic

        return policy_parts
```

3.4 Conclusion

By following this migration guide, you should be able to successfully update your Django project to use the new dict-based CSP settings format introduced in the latest version of *django-csp*. This change aligns the package with the latest CSP specification and provides a more organized and flexible way to configure your Content Security Policy.

CHAPTER

FOUR

MODIFYING THE POLICY WITH DECORATORS

Content Security Policies should be restricted and paranoid by default. You may, on some views, need to expand or change the policy. django-csp includes four decorators to help.

All decorators take an optional keyword argument, `REPORT_ONLY`, which defaults to `False`. If set to `True`, the decorator will update the report-only policy instead of the enforced policy.

4.1 @csp_exempt

Using the `@csp_exempt` decorator disables the CSP header on a given view.

```
from csp.decorators import csp_exempt


# Will not have a CSP header.
@csp_exempt()
def myview(request):
    return render(...)


# Will not have a CSP report-only header.
@csp_exempt(REPORT_ONLY=True)
def myview(request):
    return render(...)
```

You can manually set this on a per-response basis by setting the `_csp_exempt` or `_csp_exempt_ro` attribute on the response to `True`:

```
# Also will not have a CSP header.
def myview(request):
    response = render(...)
    response._csp_exempt = True
    return response
```

4.2 @csp_update

The `@csp_update` header allows you to **append** values to the source lists specified in the settings. If there is no setting, the value passed to the decorator will be used verbatim.

> **Note**
>
> To quote the CSP spec: "There's no inheritance; … the default list is not used for that resource type" if it is set. E.g., the following will not allow images from 'self':
>
> default-src 'self'; img-src imgsrv.com

The arguments to the decorator are the same as the *settings*. The decorator expects a single dictionary argument, where the keys are the directives and the values are either strings, lists or tuples. An optional argument, `REPORT_ONLY`, can be set to `True` to update the report-only policy instead of the enforced policy.

```
from csp.decorators import csp_update


# Will append imgsrv.com to the list of values for `img-src` in the enforced policy.
@csp_update({"img-src": "imgsrv.com"})
def myview(request):
    return render(...)


# Will append cdn-img.com to the list of values for `img-src` in the report-only policy.
@csp_update({"img-src": "cdn-img.com"}, REPORT_ONLY=True)
def myview(request):
    return render(...)
```

4.3 @csp_replace

The `@csp_replace` decorator allows you to **replace** a source list specified in settings. If there is no setting, the value passed to the decorator will be used verbatim. (See the note under `@csp_update`.) If the specified value is None, the corresponding key will not be included.

The arguments and values are the same as `@csp_update`:

```
from csp.decorators import csp_replace


# Will allow images only from imgsrv2.com in the enforced policy.
@csp_replace({"img-src": "imgsrv2.com"})
def myview(request):
    return render(...)


# Will allow images only from cdn-img2.com in the report-only policy.
@csp_replace({"img-src": "cdn-img2.com"}, REPORT_ONLY=True)
def myview(request):
    return render(...)
```

The `csp_replace` decorator can also be used to remove a directive from the policy by setting the value to `None`. For example, if the `frame-ancestors` directive is set in the Django settings and you want to remove the `frame-ancestors` directive from the policy for this view:

```
from csp.decorators import csp_replace
```

(continues on next page)

(continued from previous page)

```
@csp_replace({"frame-ancestors": None})
def myview(request):
    return render(...)
```

4.4 @csp

If you need to set the entire policy on a view, ignoring all the settings, you can use the `@csp` decorator. This can be stacked to update both the enforced policy and the report-only policy if both are in use, as shown below.

```
from csp.constants import SELF, UNSAFE_INLINE
from csp.decorators import csp


@csp(
    {
        "default-src": [SELF],
        "img-src": ["imgsrv.com"],
        "script-src": ["scriptsrv.com", "googleanalytics.com", UNSAFE_INLINE],
    }
)
@csp(
    {
        "default-src": [SELF],
        "img-src": ["imgsrv.com"],
        "script-src": ["scriptsrv.com", "googleanalytics.com"],
        "frame-src": [SELF],
    },
    REPORT_ONLY=True,
)
def myview(request):
    return render(...)
```

CHAPTER

FIVE

USING THE GENERATED CSP NONCE

When `NONCE` is included in a directive, the nonce value is returned in the CSP headers **if it is used**, e.g. by evaluating the nonce in your template. To actually make the browser do anything with this value, you will need to include it in the attributes of the tags that you wish to mark as safe.

> **Note**
>
> Use view source on a page to see nonce values. **Nonce values are not visible in browser developer tools.** To prevent malicious CSS selectors leaking the values, they are not exposed to the DOM.

5.1 Middleware

Installing the middleware creates a lazily evaluated property `csp_nonce` and attaches it to all incoming requests.

```
MIDDLEWARE = (
    # ...
    "csp.middleware.CSPMiddleware",
    # ...
)
```

This value can be accessed directly on the request object in any view or template and manually appended to any script element like so -

```
<script nonce="{{request.csp_nonce}}">
        var hello="world";
</script>
```

Assuming the `NONCE` sentinel is included in the `script-src` directive, this will result in the above script being allowed.

> **Note**
>
> The nonce will only be included in the CSP header if:
>
> - `csp.constants.NONCE` is present in the `script-src` or `style-src` directives, **and**
> - `request.csp_nonce` is accessed during the request lifecycle, after the middleware processes the request but before it processes the response.
>
> The `csp.middleware.CSPMiddleware` will include the nonce in the CSP header as it processes the response, but only if it was generated by code reading `str(request.csp_nonce)`.

If code reads an un-generated `request.csp_nonce` after the middleware processes the response, it is probably a programming error. In this case, attempting to read the nonce (like `str(request.csp_nonce)`) will raise a `csp.exceptions.CSPNonceError`. If the nonce was generated and included in the CSP header, then reading `request.csp_nonce` is safe.

It is always safe to test `request.csp_nonce`, such as `bool(request.csp_nonce)` or in a conditional like `if request.csp_nonce: ...`. This will return `True` if the nonce was accessed and generated, and `False` if not acccesed or generated yet.

If other middleware or a later process needs to access `request.csp_nonce`, then there are a few options:

- The middleware can be placed after `csp.middleware.CSPMiddleware` in the `MIDDLEWARE` setting. This ensures that the middleware generates the nonce before `CSPMiddleware` writes the CSP header.
- Add a later middleware that accesses the nonce. For example, this function:

```
def init_csp_nonce_middleware(get_response):
    def middleware(request):
        str(getattr(request, "csp_nonce", None))
        return get_response(request)

    return middleware
```

could be added to the `MIDDLEWARE` list:

```
MIDDLEWARE = (
    "my.middleware.ThatUsesCSPNonce",
    # ...
    "csp.middleware.CSPMiddleware",
    # ...
    "my.middleware.init_csp_nonce_middleware",
)
```

5.2 Context Processor

This library contains an optional context processor, adding `csp.context_processors.nonce` to your configured context processors exposes a variable called `CSP_NONCE` into the global template context. This is simple shorthand for `request.csp_nonce`, but can be useful if you have many occurrences of script tags.

```
<script nonce="{{CSP_NONCE}}">
    var hello="world";
</script>
```

5.3 Django Template Tag/Jinja Extension

Note

If you're making use of `csp.extensions.NoncedScript` you need to have `jinja2>=2.9.6` installed, so please make sure to either use `django-csp[jinja2]` in your requirements or define it yourself.

It can be easy to forget to include the `nonce` property in a script tag, so there is also a `script` template tag available for both Django templates and Jinja environments.

This tag will output a properly nonced script every time. For the sake of syntax highlighting, you can wrap the content inside of the `script` tag in `<script>` html tags, which will be subsequently removed in the rendered output. Any valid script tag attributes can be specified and will be forwarded into the rendered html.

5.3.1 Django Templates

Add the CSP template tags to the TEMPLATES section of your settings file:

```
TEMPLATES = [
    {
        "OPTIONS": {
            "libraries": {
                "csp": "csp.templatetags.csp",
            }
        }
    }
]
```

Then load the `csp` template tags and use `script` in the template:

```
{% load csp %}
{% script type="application/javascript" async=False %}
        <script>
                var hello='world';
        </script>
{% endscript %}
```

5.3.2 Jinja

Add `csp.extensions.NoncedScript` to the TEMPLATES section of your settings file:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.jinja2.Jinja2",
        "OPTIONS": {
            "extensions": [
                "csp.extensions.NoncedScript",
            ],
        },
    }
]
```

```
{% script type="application/javascript" async=False %}
        <script>
                var hello='world';
        </script>
{% endscript %}
```

Both templates output the following with a different nonce:

```
<script nonce='123456' type="application/javascript" async=false>var hello='world';</
↪script>
```

CHAPTER

SIX

IMPLEMENTING TRUSTED TYPES WITH CSP

6.1 DOM Cross-site Scripting

Cross-site scripting (XSS) is one of the most prevalent vulnerabilities on the web. Nonce-based CSP is used to prevent server-side XSS. Trusted Types are used to prevent client-side or DOM-XSS. Trusted Types rely on the browser to enforce the policy that is provided to it. Currently, Trusted Types are supported on Chrome 83 and Android Webview. Many browsers are in the process of adding support. Check back for updated compatibility.

Follow the simple steps below to make your web application Trusted Types compliant.

6.2 Step 1: Enable Trusted Types and Report Only Mode

Trusted Types require data to be processed before being sent to a risky "sink" where DOM XSS might occur, such as when assigning to Element.innerHTML or calling document.write. When enforced, Trusted Types will tell the browser to block any data that is not properly processed. In order to avoid this, you must fix offending parts of your code. To see where adjustments will be required, turn on trusted types and report only mode.

Configure django-csp so that the directive `require-trusted-types-for` is set to *'script'*.

Configure the django-csp report-only policy.

Configure django-csp so that the directive `report-uri` is set to an app or CSP report processing service that you control.

Now trusted types violations will be reported to your `report-uri` without blocking any of your application's functionalities.

6.3 Step 2: Fixing Trusted Types Violations

There are four ways to resolve trusted types violations. They are explained here in order of preference.

6.3.1 Rewrite the Code

It may be possible for your code to be rewritten without using dangerous functions. For example, instead of dynamically placing an image using the dangerous `innerHTML` sink, the image could be created with `document.createElement` and placed using the `appendChild` function.

Rewriting may be possible for any of the dangerous sinks, which are listed here.

- **Script manipulation:**
  - **`<script src>` and setting text content of `<script>` elements.**
    - Tip: Avoid creating scripts at run time

        * Tip: Create a policy with a URL stringifier to verify scripts are from a trusted origin

- **Generating HTML from a string:**
    - **`innerHTML`, `outerHTML`, `insertAdjacentHTML`, `<iframe> srcdoc`, `document.write`, `document.writeln`, and `DOMParser.parseFromString`**
        * Tip: Use textContent instead of inner HTML
        * Tip: Use a templating library that supports Trusted Types
        * Tip: Use createElement and appendChild as explained above
- **Executing plugin content:**
    - **`<embed src>`, `<object data>` and `<object codebase>`**
        * Tip: Consider limiting plugin content by setting `object-src` to *none*
- **Runtime JavaScript code compilation:**
    - **`eval`, `setTimeout`, `setInterval`, and `new Function()`**
        * Tip: Avoid using eval entirely
        * Tip: Avoid passing strings to runtime compiled functions

6.3.2 Use a Library

When code cannot be rewritten to avoid dangerous sinks, Trusted Types require that data be processed before being passed to a dangerous sink. Processed data is wrapped in a TrustedHTML, TrustedScript, or TrustedScriptURL object to certify that it has been sanitized or otherwise assured to be safe in the given context. Some libraries will process data and return Trusted Types objects for you. For example, DOMPurify supports Trusted Types.

> **Note**
>
> Libraries are preferred to writing your own sanitation policies since they are generally more comprehensive, secure, and well reviewed.

6.3.3 Create Trusted Types Policies

Where code cannot be rewritten and an existing library cannot be used, you will have to create Trusted Types objects yourself. This is done using policies. Different policies can be created for use in different contexts. Policies produce Trusted Types after enforcing security rules on their input based on the sink context. Each policy should be given a distinct name.

Here is an example policy that sanitizes HTML by escaping the < character.

```
if (window.trustedTypes && trustedTypes.createPolicy) {
const escapeHTMLPolicy = trustedTypes.createPolicy('myEscapePolicy', {
        createHTML: string => string.replace(/\</g, '&lt;')
        });
}
```

Here is an example of how that policy can be used.

```
const escaped = escapeHTMLPolicy.createHTML('<img src=x onerror=alert(1)>');
console.log(escaped instanceof TrustedHTML);
el.innerHTML = escaped;
```

> **Note**
>
> Keep in mind that you are creating your own security rules with policies. Your application is only protected from DOM XSS if you use strict sanitation rules that consider which sink is accepting the data.

6.3.4 Use a Default Policy

In the event that you don't have control over the offending code, you can use a default policy. This may happen if you are loading a third party library that is not Trusted Types compliant. A default policy is defined the same way as any other Trusted Types policy. In order to be used by the browser as the default policy it must be named *default*.

The policy called *default* will be used wherever a string is sent to a dangerous sink that requires Trusted Types.

6.4 Step 3: Enforce Trusted Types

Once you have addressed all of the Trusted Types violations present in your application, you can begin enforcing Trusted Types to prevent DOM XSS.

Configure django-csp to remove the report-only policy from settings, if no longer needed.

> **Note**
>
> To learn more about trusted types or learn how to limit policy creation with `trusted-types` take a look at the complete spec or the article this guide is based on.

CHAPTER

SEVEN

CSP VIOLATION REPORTS

When something on a page violates the Content-Security-Policy, and the policy defines a `report-uri` directive, the user agent may POST a report. Reports are JSON blobs containing information about how the policy was violated.

Note: django-csp no longer handles report processing itself, so you will need to stand up your own app to receive them, or else make use of a third-party report processing service.

7.1 Throttling the number of reports

To throttle the number of requests made to your `report-uri` endpoint, you can use `csp.contrib.rate_limiting.RateLimitedCSPMiddleware` instead of `csp.middleware.CSPMiddleware` and set the `REPORT_PERCENTAGE` option:

`REPORT_PERCENTAGE`

Percentage of requests that should see the `report-uri` directive. Use this to throttle the number of CSP violation reports made to your `report-uri`. A **float** between 0.0 and 100.0 (0.0 = no reports at all, 100.0 = always report). Ignored if `report-uri` isn't set.

CHAPTER

EIGHT

CONTRIBUTING

Patches are more than welcome! You can find the issue tracker on GitHub and we'd love pull requests.

8.1 Setup

To install all the requirements (probably into a virtualenv):

```
pip install -e .
pip install -e ".[dev]"
```

This installs:

- All the text requirements
- All the typing requirements
- pre-commit, for checking styles
- tox, for running tests against multiple environments
- Sphinx and document building requirements

8.2 Style

Patches should follow PEP8 and should not introduce any new violations as detected by the ruff tool.

To help stay on top of this, install pre-commit, and then run `pre-commit install-hooks`. Now you'll be set up to auto-format your code according to our style and check for errors for every commit.

8.3 Tests

Patches fixing bugs should include regression tests (ideally tests that fail without the rest of the patch). Patches adding new features should test those features thoroughly.

To run the tests, install the requirements (probably into a virtualenv):

```
pip install -e .
pip install -e ".[tests]"
```

Then just pytest to run the tests:

```
pytest
```

To run the tests with coverage and get a report, use the following command:

```
pytest --cov=csp --cov-config=.coveragerc
```

To run the tests like Github Actions does, you'll need pyenv:

```
pyenv install 3.9 3.10 3.11 3.12 pypy3.9 pypy3.10
pyenv local 3.9 3.10. 3.11 3.12 pypy3.9 pypy3.10
pip install -e ".[dev]"  # installs tox
tox                # run sequentially
tox run-parallel   # run in parallel, may cause issues on coverage step
tox -e 3.12-4.2.x  # run tests on Python 3.12 and Django 4.x
tox --listenvs     # list all the environments
```

8.4 Type Checking

New code should have type annotations and pass mypy in strict mode. Use the typing syntax available in the earliest supported Python version 3.9.

To check types:

```
pip install -e ".[typing]"
mypy .
```

If you make a lot of changes, it can help to clear the mypy cache:

```
mypy --no-incremental .
```

8.5 Updating Documentation

To rebuild documentation locally:

```
pip install -e ".[dev]"
cd docs
make html
open _build/html/index.html  # On macOS
```