Django-CSP Documentation

Release 3.7

James Socol, Mozilla

Jun 28, 2022

Contents

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

Version 3.7

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

License BSD; see LICENSE file

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

Contents:

CHAPTER 1

Installing django-csp

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

```
# pip
$ pip install django-csp
```

```
# source
$ git clone https://github.com/mozilla/django-csp.git
$ cd django-csp
$ python setup.py install
```

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.

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

CHAPTER 2

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 Policy Settings

These settings affect the policy in the header. The defaults are in *italics*.

Note: The "special" source values of `'self'`, `'unsafe-inline'`, `'unsafe-eval'`, `'none'` and hash-source (`'sha256-...'`) must be quoted! e.g.: `CSP_DEFAULT_SRC = ("'self'",)`. Without quotes they will not work as intended.

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

CSP_SCRIPT_SRC Set the `script-src` directive. A `tuple` or `list`. *None*

CSP_SCRIPT_SRC_ATTR Set the `script-src-attr` directive. A `tuple` or `list`. *None*

CSP_SCRIPT_SRC_ELEM Set the `script-src-elem` directive. A `tuple` or `list`. *None*

CSP_IMG_SRC Set the `img-src` directive. A `tuple` or `list`. *None*

CSP_OBJECT_SRC Set the `object-src` directive. A `tuple` or `list`. *None*

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

CSP_MEDIA_SRC Set the `media-src` directive. A `tuple` or `list`. *None*

CSP_FRAME_SRC Set the `frame-src` directive. A `tuple` or `list`. *None*

CSP_FONT_SRC Set the `font-src` directive. A `tuple` or `list`. *None*

CSP_CONNECT_SRC Set the `connect-src` directive. A `tuple` or `list`. *None*

CSP_STYLE_SRC Set the `style-src` directive. A `tuple` or `list`. *None*

CSP_STYLE_SRC_ATTR Set the `style-src-attr` directive. A `tuple` or `list`. *None*

CSP_STYLE_SRC_ELEM Set the `style-src-elem` directive. A `tuple` or `list`. *None*

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

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

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

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

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

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

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

CSP_FORM_ACTION Set the `FORM_ACTION` directive. A `tuple` or `list`. *None*

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

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

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

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

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

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

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

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

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

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

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

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

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

See: require-sri-for-known-tokens

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

See: upgrade-insecure-requests

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

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

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

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

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

See: block-all-mixed-content

CSP_INCLUDE_NONCE_IN Include dynamically generated nonce in all listed directives. A `tuple` or `list`, e.g.: `CSP_INCLUDE_NONCE_IN = ['script-src']` will add `'nonce-<b64-value>'` to the `script-src` directive. *['default-src']*

Note: The nonce value will only be generated if `request.csp_nonce` is accessed during the request/response cycle.

2.1.1 Changing the Policy

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

2.2 Other Settings

These settings control the behavior of django-csp. Defaults are in *italics*.

CSP_REPORT_ONLY Send "report-only" headers instead of real headers. A `boolean`. *False*

See the spec and the chapter on *reports* for more info.

CSP_EXCLUDE_URL_PREFIXES A `tuple` (*not* a `list`) of URL prefixes. URLs beginning with any of these will not get the CSP headers. *()*

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.

CHAPTER 3

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.

3.1 @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(...)
```

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

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

3.2 @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 the same as the *settings* without the `CSP_` prefix, e.g. `IMG_SRC`. (They are also case-insensitive.) The values are either strings, lists or tuples.

```
from csp.decorators import csp_update

# Will allow images from imgsrv.com.
@csp_update(IMG_SRC='imgsrv.com')
def myview(request):
    return render(...)
```

3.3 @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`.)

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

```
from csp.decorators import csp_replace

# settings.CSP_IMG_SRC = ['imgsrv.com']
# Will allow images from imgsrv2.com, but not imgsrv.com.
@csp_replace(IMG_SRC='imgsrv2.com')
def myview(request):
    return render(...)
```

3.4 @csp

If you need to set the entire policy on a view, ignoring all the settings, you can use the `@csp` decorator. The arguments and values are as above:

```
from csp.decorators import csp

@csp(DEFAULT_SRC=["'self'"], IMG_SRC=['imgsrv.com'],
     SCRIPT_SRC=['scriptsrv.com', 'googleanalytics.com'])
def myview(request):
    return render(...)
```

CHAPTER 4

Using the generated CSP nonce

When `CSP_INCLUDE_NONCE_IN` is configured, 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.

4.1 Middleware

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

```
    MIDDLEWARE_CLASSES = (
    #...
    '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 `CSP_INCLUDE_NONCE_IN` list contains the `script-src` directive, this will result in the above script being allowed.

Note: The nonce will only be added to the CSP headers if it is used.

4.2 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>
```

4.3 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.

4.3.1 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 %}
```

4.3.2 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 5

Implementing Trusted Types with CSP

5.1 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.

5.2 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 `CSP_REQUIRE_TRUSTED_TYPES_FOR` is set to *'script'*.

Configure django-csp so that `CSP_REPORT_ONLY` is set to *True*.

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

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

5.3 Step 2: Fixing Trusted Types Violations

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

5.3.1 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.w`**
    - 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‘‘CSP_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

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.4 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 so that `CSP_REPORT_ONLY` is set to *False*.

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

CHAPTER 6

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.

6.1 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 `CSP_REPORT_PERCENTAGE` option:

`CSP_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 `CSP_REPORT_URI`. A **float** between 0 and 1 (0 = no reports at all). Ignored if `CSP_REPORT_URI` isn't set.

CHAPTER 7

Contributing

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

7.1 Style

Patches should follow PEP8 and should not introduce any new violations as detected by the flake8 tool.

7.2 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 py.test to run the tests:

```
py.test
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search