django-security Documentation

Release 1.0

Pawel Krawczyk

Aug 30, 2019

Contents

Contents:

CHAPTER 1

Security models

CHAPTER 2

Security views

security.views.**csp_report**(**args*, ***kwargs*)

Collect Content Security Policy reports from browsers. This view has two optional keyword arguments:

csp_save if True, reports will be saved as CspReport objects in database; this table is registered with Django Admin, so they can be later viewed in admin console.

csp_log if True, reports will be logged through Django logging facility under `security` class

By default only logging is enabled. To collect reports, this view needs to be added to project's urls.py. Examples:

Default mode, only logger enable, no database logging:

```
url(r'^csp-report/$', security.views.csp_report),
```

Logger and database enabled:

```
url(r'^csp-report/$', security.views.csp_report,
kwargs={'csp_save':True,'csp_log':True}),
```

security.views.**require_ajax**(*view*)

A view decorator which ensures that the request being proccessed by view is an AJAX request. We return a 403 error if the request is not an AJAX request.

CHAPTER 3

Security middleware

class security.middleware.**BaseMiddleware**(*get_response=None*)

Abstract class containing some functionality common to all middleware that require configuration.

load_setting(*setting*, *value*)

Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

class security.middleware.**ClearSiteDataMiddleware**(*get_response=None*)

Sends Clear-Site-Data HTTP response header on requests that match CLEAR_SITE_DATA_URL_WHITELIST.

Clears browsing data (cookies, storage, cache) associated with the requesting website. Allows web developers to have more control over the data stored locally by a browser for their origins.

Reference:

- Clear-Site-Data: “cache”, “cookies”, “storage”, “executionContexts”, “*” <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Clear-Site-Data>‘_

load_setting(*setting*, *value*)

Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

process_response(*request*, *response*)

Add `Clear-Site-Data` response header if request in CLEAR_SITE_DATA_URLS.

class security.middleware.**ContentNoSniff**(*get_response=None*)

Sends X-Content-Options HTTP header to disable autodetection of MIME type of files returned by the server in Microsoft Internet Explorer. Specifically if this flag is enabled, MSIE will not load external CSS and JavaScript files unless server correctly declares their MIME type. This mitigates attacks where web page would for example load a script that was disguised as an user- supplied image.

Note: As of 1.8, Django's SECURE_CONTENT_TYPE_NOSNIFF controls the X-Content-Type-Options header.

Reference:

- MIME-Handling Change: X-Content-Type-Options: nosniff

process_response(*request*, *response*)
Add `X-Content-Options: nosniff` to the response header.

class security.middleware.**ContentSecurityPolicyMiddleware**(*get_response=None*)
Adds Content Security Policy (CSP) header to HTTP response. CSP provides fine grained instructions to the browser on location of allowed resources loaded by the page, thus mitigating attacks based on loading of untrusted JavaScript code such as Cross-Site Scripting.

The policy can be set in two modes, controlled by `CSP_MODE` options:

`CSP_MODE='enforce'` browser will enforce policy settings and log violations (*default*)

`CSP_MODE='report-only'` browser will not enforce policy, only report violations

The policy itself is a dictionary of content type keys and values containing list of allowed locations. For example, `img-src` specifies locations of images allowed to be loaded by this page:

```
'img-src' :  [ 'img.example.com' ]
```

Content types and special location types (such as `none` or `self`) are defined in CSP draft (see *References*).

Example of policy dictionary (suitable for long, complex policies), with all supported content types (but not listing all supported locations):

```
CSP_DICT = {
    # arrays of allowed locations
    "default-src" : ["self", "https:" ],
    "script-src" : ["self", "http://js.example.com" ],
    "style-src" : ["self", "http://css.example.com" ],
    "img-src" : ["self", "https://img.example.com" ],
    "connect-src" : ["self" ],
    "font-src" : ["https://*.example.com" ],
    "object-src" : ["none" ],
    "media-src" : ["http://media.example.com" ],
    "frame-src" : ["self" ],

    # array of allowed sandbox features
    "sandbox" : [ "" ],

    # array of allowed MIME types
    "plugin-types" : [ "application/pdf" ],

    # these are **not** arrays
    "reflected-xss" : 'filter',
    "report-uri" : "http://example.com/csp-report",
}
```

You can also supply a raw policy string, which is more suitable for short policies:

```
CSP_STRING="default-src 'self'; script-src *.google.com"
```

If both `CSP_DICT` and `CSP_STRING` are set the middleware will throw an exception.

Notes:

- The special locations ('self', 'none', 'unsafe-eval', 'unsafe-inline') come **without** the additional single quotes in *CSP_DICT* and they will be automatically quoted by the middleware in the HTTP header.
- The `CSP_STRING` on the other hand should be a verbatim copy of the HTTP header contents. It's not going the be processed in any way.
- This middleware only sets the standard HTTP header variants (`Content-Security-Policy`). The experimental ones (`X-WebKit-CSP` and `Content-Security-Policy`) are now obsolete.
- Enabling CSP has significant impact on browser behaviour - for example inline JavaScript is disabled. Read Default Policy Restrictions to see how pages need to be adapted to work under CSP.
- Browsers will log CSP violations in JavaScript console and to a remote server configured by `report-uri` option. This package provides a view (*csp_report*) to collect these alerts in your application. They can be then viewed using Django admin interface. For more advanced analytics try CspBuilder.
- The middleware partially supports CSP 1.1 draft syntax.

References:

- Content Security Policy Level 2,
- HTML5.1 - Sandboxing

process_response(*request*, *response*)
Add Content Security Policy policy to the response header. Use either enforcement or report-only headers in all currently used variants.

class security.middleware.**CustomLogoutMixin**
If the CUSTOM_LOGOUT_MODULE is set in Django config, import and use that when performing a logout.

class security.middleware.**DoNotTrackMiddleware**(*get_response=None*)
When this middleware is installed Django views can access a new `request.dnt` parameter to check client's preference on user tracking as expressed by their browser configuration settings.

The parameter can take True, False or None values based on the presence of the `Do Not Track` HTTP header in client's request, which in turn depends on browser's configuration. The header indicates client's general preference to opt-out from behavioral profiling and third-party tracking.

The parameter does **not** change behaviour of Django in any way as its sole purpose is to pass the user's preference to application. It's then up to the owner to implement a particular policy based on this information. Compliant website should adapt its behaviour depending on one of user's preferences:

- Explicit opt-out (`request.dnt` is `True`): Disable third party tracking for this request and delete all previously stored tracking data.
- Explicit opt-in (`request.dnt` is `False`): Website may track user.
- Header not present (`request.dnt` is `None`): Website may track user, but should not draw any definite conclusions on user's preferences as the user has not expressed it.

For example, if `request.dnt` is '' True'' the website might respond by disabling template parts responsible for personalized statistics, targeted advertisements or switching to DNT aware ones.

Examples:

- Do Not Track (DNT) tutorial for Django
- Do Not Track - Web Application Templates
- Opt-out of tailoring Twitter

References:

- Web Tracking Protection

- Do Not Track: A Universal Third-Party Web Tracking Opt Out

process_request(*request*)
Read DNT header from browser request and create request attribute

process_response(*request*, *response*)
Echo DNT header in response per section 8.4 of draft-mayer-do-not- track-00

class security.middleware.**LoginRequiredMiddleware**(*get_response=None*)
Middleware that requires a user to be authenticated to view any page on the site that hasn't been white listed. (The middleware also ensures the user is 'active'. Disabled users will be logged out and redirected to the login page.

Exemptions to this requirement can optionally be specified in settings via a list of regular expressions in LOGIN_EXEMPT_URLS (which you can copy from your urls.py).

Requires authentication middleware and template context processors to be loaded. You'll get an error if they aren't.

By default this middleware will call the builtin Django logout function to perform the logout. You can customize which logout function will be called by specifying it in your django settings using the CUSTOM_LOGOUT_MODULE variable. The value should be the module path to the function, e.g. 'django.contrib.auth.logout'.

load_setting(*setting*, *value*)
Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

class security.middleware.**MandatoryPasswordChangeMiddleware**(*get_response=None*)
Redirects any request from an authenticated user to the password change form if that user's password has expired. Must be placed after `AuthenticationMiddleware` in the middleware list.

Configured by dictionary `MANDATORY_PASSWORD_CHANGE` with the following keys:

`URL_NAME` name of of the password change view

`EXEMPT_URL_NAMES` list of URLs that do not trigger password change request

`INCLUDE_SUPERUSERS` also check superusers for password change, default False

load_setting(*setting*, *value*)
Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

class security.middleware.**NoConfidentialCachingMiddleware**(*get_response=None*)
Adds No-Cache and No-Store headers to confidential pages. You can either whitelist non-confidential pages and treat all others as non-confidential, or specifically blacklist pages as confidential. The behaviour is configured in `NO_CONFIDENTIAL_CACHING` dictionary in settings file with the following keys:

WHITELIST_ON all pages are confifendialt, except for pages explicitly whitelisted in `WHITELIST_REGEXES`

WHITELIST_REGEXES list of regular expressions defining pages exempt from the no caching policy

BLACKLIST_ON only pages defined in BLACKLIST_REGEXES will have caching disabled

BLACKLIST_REGEXES list of regular expressions defining confidential pages for which caching should be prohibited

Note: Django's cache_control decorator allows more granular control of caching on individual view level.

Reference:

- HTTP/1.1 Header definitions - What is Cacheable

load_setting(*setting*, *value*)
Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

process_response(*request*, *response*)
Add the Cache control no-store to anything confidential. You can either whitelist non-confidential pages and treat all others as non- confidential, or specifically blacklist pages as confidential

class security.middleware.**P3PPolicyMiddleware**(*get_response=None*)
Adds the HTTP header attribute specifying compact P3P policy defined in P3P_COMPACT_POLICY setting and location of full policy defined in P3P_POLICY_URL. If the latter is not defined, a default value is used (/w3c/p3p.xml). The policy file needs to be created by website owner.

Note: P3P work stopped in 2002 and the only popular browser with **limited** P3P support is MSIE.

Reference:

- The Platform for Privacy Preferences 1.0 (P3P1.0) Specification - The Compact Policies

load_setting(*setting*, *value*)
Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

process_response(*request*, *response*)
Add P3P policy to the response header.

class security.middleware.**SessionExpiryPolicyMiddleware**(*get_response=None*)
The session expiry middleware will let you expire sessions on browser close, and on expiry times stored in the cookie itself. (Expiring a cookie on browser close means you don't set the expiry value of the cookie.) The middleware will read SESSION_COOKIE_AGE and SESSION_INACTIVITY_TIMEOUT from the settings.py file to determine how long to keep a session alive.

We will purge a session that has expired. This middleware should be run before the LoginRequired middleware if you want to redirect the expired session to the login page (if required).

Exemptions to this requirement can optionally be specified in settings via a list of regular expressions in SESSION_EXPIRY_EXEMPT_URLS (which you can copy from your urls.py).

By default this middleware will call the builtin Django logout function to perform the logout. You can customize which logout function will be called by specifying it in your django settings using the CUSTOM_LOGOUT_MODULE variable. The value should be the module path to the function, e.g. 'django.contrib.auth.logout'.

load_setting(*setting*, *value*)
Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

process_request(*request*)
Verify that the session should be considered active. We check the start time and the last activity time to determine if this is the case. We set the last activity time to now() if the session is still active.

class security.middleware.**StrictTransportSecurityMiddleware**(*get_response=None*)

Adds Strict-Transport-Security header to HTTP response that enforces SSL connections on compliant browsers. Two parameters can be set in settings file, otherwise reasonable defaults will be used:

- `STS_MAX_AGE` time in seconds to preserve host's STS policy (default: 1 year)
- `STS_INCLUDE_SUBDOMAINS` True if subdomains should be covered by the policy as well (default: True)
- `STS_PRELOAD` add `preload` flag to the STS header so that your website can be added to preloaded websites list

Note: As of 1.8, Django's SECURE_HSTS_SECONDS controls the HTTP Strict Transport Security header.

Reference:

- 'HTTP Strict Transport Security (HSTS)

  <https://datatracker.ietf.org/doc/rfc6797/>'_

- Preloaded HSTS sites

process_response(*request*, *response*)

Add Strict-Transport-Security header.

security.middleware.**XFrameOptionsDenyMiddleware**

alias of *security.middleware.XFrameOptionsMiddleware*

class security.middleware.**XFrameOptionsMiddleware**(*get_response=None*)

Emits `X-Frame-Options headers` in HTTP response. These headers will instruct the browser to limit ability of this web page to be framed, or displayed within a FRAME or IFRAME tag. This mitigates password stealing attacks like Clickjacking and similar.

Use `X_FRAME_OPTIONS` in settings file with the following values:

- `deny` prohibit any framing of this page
- `sameorigin` allow frames from the same domain (*default*)
- `allow-from URL` allow frames from specified *URL*

Note: Frames and inline frames are frequently used by ads, social media plugins and similar widgets so test these features after setting this flag.

You can exclude certain URLs from this header by setting `X_FRAME_OPTIONS_EXCLUDE_URLS` to a list of URL regexes like so:

```
X_FRAME_OPTIONS_EXCLUDE_URLS = (
    r'^/some/url/here$',        # Note the initial slash
    r'^/another/to/exclude$',
)
```

The header will be sent unless `request.path` matches any of the above list. For more granular control, use *ContentSecurityPolicyMiddleware*.

References:

- RFC 7034: HTTP Header Field X-Frame-Options

load_setting(*setting*, *value*)

Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the

new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

process_response(*request*, *response*)
Add X-Frame-Options and Frame-Options to the response header.

class security.middleware.**XssProtectMiddleware**(*get_response=None*)
Sends X-XSS-Protection HTTP header that controls Cross-Site Scripting filter on MSIE. Use XSS_PROTECT option in settings file with the following values:

`sanitize` enable XSS filter that tries to sanitize requests instead of blocking (*default*)

`on` enable full XSS filter blocking XSS requests (may leak document.referrer)

`off` completely disable XSS filter

Note: As of 1.8, Django's SECURE_BROWSER_XSS_FILTER controls the X-XSS-Protection header.

Reference:

- Controlling the XSS Filter

load_setting(*setting*, *value*)
Called initially for each of the keys in REQUIRED_SETTINGS and OPTIONAL_SETTINGS, and again whenever any of these settings change (from the setting_changed signal). Passed the setting key and the new value, which may be None for the keys in OPTIONAL_SETTINGS. If no setting keys are defined then this method is never called.

process_response(*request*, *response*)
Add X-XSS-Protection to the reponse header.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

R

S

X