django-secure Documentation

Release 1.0.1

Carl Meyer and contributors

Sep 27, 2017

Contents

1 Quickstart **3**
- 1.1 Dependencies . . . 3
- 1.2 Installation . . . 3
- 1.3 Usage . . . 3

2 The Details **5**
- 2.1 Design Goals . . . 5
- 2.2 SecurityMiddleware . . . 5
- 2.3 The `checksecure` management command . . . 8
- 2.4 Settings Reference . . . 10
- 2.5 CHANGES . . . 13
- 2.6 TODO . . . 13
- 2.7 Contributors . . . 13

3 Indices and tables **15**

Helping you remember to do the stupid little things to improve your Django site's security.

Inspired by Mozilla's Secure Coding Guidelines, and intended for sites that are entirely or mostly served over SSL (which should include anything with user logins).

CHAPTER 1

Quickstart

Dependencies

Tested with Django 1.4 through trunk, and Python 2.6, 2.7, 3.2, and 3.3. Quite likely works with older versions of both, though; it's not very complicated.

Installation

Install from PyPI with `pip`:

```
pip install django-secure
```

or get the in-development version:

```
pip install django-secure==dev
```

Usage

- Add `"djangosecure"` to your `INSTALLED_APPS` setting.
- Add `"djangosecure.middleware.SecurityMiddleware"` to your `MIDDLEWARE_CLASSES` setting (where depends on your other middlewares, but near the beginning of the list is probably a good choice).
- Set the `SECURE_SSL_REDIRECT` setting to `True` if all non-SSL requests should be permanently redirected to SSL.
- Set the `SECURE_HSTS_SECONDS` setting to an integer number of seconds and `SECURE_HSTS_INCLUDE_SUBDOMAINS` to `True`, if you want to use HTTP Strict Transport Security.

- Set the `SECURE_FRAME_DENY` setting to `True`, if you want to prevent framing of your pages and protect them from clickjacking.
- Set the `SECURE_CONTENT_TYPE_NOSNIFF` setting to `True`, if you want to prevent the browser from guessing asset content types.
- Set the `SECURE_BROWSER_XSS_FILTER` setting to `True`, if you want to enable the browser's XSS filtering protections.
- Set `SESSION_COOKIE_SECURE` and `SESSION_COOKIE_HTTPONLY` to `True` if you are using `django.contrib.sessions`. These settings are not part of `django-secure`, but they should be used if running a secure site, and the `checksecure` management command will check their values.
- Ensure that you're using a long, random and unique `SECRET_KEY`.
- Run `python manage.py checksecure` to verify that your settings are properly configured for serving a secure SSL site.

Warning: If `checksecure` gives you the all-clear, all it means is that you're now taking advantage of a small selection of easy security wins. That's great, but it doesn't mean your site or your codebase is secure: only a competent security audit can tell you that.

CHAPTER 2

The Details

Design Goals

Django-secure does not make your site secure. It does not audit code, or do intrusion detection, or really do anything particularly interesting or complicated.

Django-secure is an automated low-hanging-fruit checklist. Django-secure helps you remember the stupid simple things that improve your site's security, reminds you to do those easy things, and makes them as easy as possible to do.

SecurityMiddleware

The `djangosecure.middleware.SecurityMiddleware` performs six different tasks for you. Each one can be independently enabled or disabled with a setting.

- *X-Frame-Options: DENY*
- *HTTP Strict Transport Security*
- *X-Content-Type-Options: nosniff*
- *X-XSS-Protection: 1; mode=block*
- *SSL Redirect*
- *Detecting proxied SSL*

X-Frame-Options: DENY

Note: Django 1.4+ provides its own middleware and setting to set the `X-Frame-Options` header; you can use either this or Django's, there's no value in using both.

Clickjacking attacks use layered frames to mislead users into clicking on a different link from the one they think they are clicking on. Fortunately, newer browsers support an `X-Frame-Options` header that allows you to limit or prevent the display of your pages within a frame. Valid options are "DENY" or "SAMEORIGIN" - the former prevents all framing of your site, and the latter allows only sites within the same domain to frame.

Unless you have a need for frames, your best bet is to set "X-Frame-Options: DENY" – and this is what `SecurityMiddleware` will do for all responses, if the *SECURE_FRAME_DENY* setting is `True`.

If you have a few pages that should be frame-able, you can set the "X-Frame-Options" header on the response to "SAMEORIGIN" in the view; `SecurityMiddleware` will not override an already-present "X-Frame-Options" header. If you don't want the "X-Frame-Options" header on this view's response at all, decorate the view with the `frame_deny_exempt` decorator:

```
from djangosecure.decorators import frame_deny_exempt

@frame_deny_exempt
def my_view(request):
    # ...
```

HTTP Strict Transport Security

For sites that should only be accessed over HTTPS, you can instruct newer browsers to refuse to connect to your domain name via an insecure connection (for a given period of time) by setting the "Strict-Transport-Security" header. This reduces your exposure to some SSL-stripping man-in-the-middle (MITM) attacks.

`SecurityMiddleware` will set this header for you on all HTTPS responses if you set the *SECURE_HSTS_SECONDS* setting to a nonzero integer value.

Additionally, if you set the *SECURE_HSTS_INCLUDE_SUBDOMAINS* setting to `True`, `SecurityMiddleware` will add the `includeSubDomains` tag to the `Strict-Transport-Security` header. This is recommended, otherwise your site may still be vulnerable via an insecure connection to a subdomain.

Warning: The HSTS policy applies to your entire domain, not just the URL of the response that you set the header on. Therefore, you should only use it if your entire domain is served via HTTPS only.

Warning: Browsers properly respecting the HSTS header will refuse to allow users to bypass warnings and connect to a site with an expired, self-signed, or otherwise invalid SSL certificate. If you use HSTS, make sure your certificates are in good shape and stay that way!

Note: If you are deployed behind a load-balancer or reverse-proxy server, and the Strict-Transport-Security header is not being added to your responses, it may be because Django doesn't realize when it's on a secure connection; you may need to set the *SECURE_PROXY_SSL_HEADER* setting.

X-Content-Type-Options: nosniff

Some browsers will try to guess the content types of the assets that they fetch, overriding the `Content-Type` header. While this can help display sites with improperly configured servers, it can also pose a security risk.

If your site serves user-uploaded files, a malicious user could upload a specially-crafted file that would be interpreted as HTML or Javascript by the browser when you expected it to be something harmless.

To learn more about this header and how the browser treats it, you can read about it on the IE Security Blog.

To prevent the browser from guessing the content type, and force it to always use the type provided in the `Content-Type` header, you can pass the `X-Content-Type-Options: nosniff` header. `SecurityMiddleware` will do this for all responses if the *SECURE_CONTENT_TYPE_NOSNIFF* setting is `True`.

X-XSS-Protection: 1; mode=block

Some browsers have to ability to block content that appears to be an XSS attack. They work by looking for Javascript content in the GET or POST parameters of a page. If the Javascript is replayed in the server's response the page is blocked from rendering and a error page is shown instead.

The X-XSS-Protection header is used to control the operation of the XSS filter.

To enable the XSS filter in the browser, and force it to always block suspected XSS attacks, you can pass the `X-XSS-Protection: 1; mode=block` header. `SecurityMiddleware` will do this for all responses if the *SECURE_BROWSER_XSS_FILTER* setting is `True`.

Warning: The XSS filter does not prevent XSS attacks on your site, and you should ensure that you are taking all other possible mesaures to prevent XSS attacks. The most obvious of these is validating and sanitizing all input.

SSL Redirect

If your site offers both HTTP and HTTPS connections, most users will end up with an unsecured connection by default. For best security, you should redirect all HTTP connections to HTTPS.

If you set the *SECURE_SSL_REDIRECT* setting to True, `SecurityMiddleware` will permanently (HTTP 301) redirect all HTTP connections to HTTPS.

Note: For performance reasons, it's preferable to do these redirects outside of Django, in a front-end loadbalancer or reverse-proxy server such as nginx. In some deployment situations this isn't an option - *SECURE_SSL_REDIRECT* is intended for those cases.

If the *SECURE_SSL_HOST* setting has a value, all redirects will be sent to that host instead of the originally-requested host.

If there are a few pages on your site that should be available over HTTP, and not redirected to HTTPS, you can list regular expressions to match those URLs in the *SECURE_REDIRECT_EXEMPT* setting.

Note: If you are deployed behind a load-balancer or reverse-proxy server, and Django can't seem to tell when a request actually is already secure, you may need to set the *SECURE_PROXY_SSL_HEADER* setting.

Detecting proxied SSL

Note: Django 1.4+ offers the same functionality built-in. The Django setting works identically to this version.

In some deployment scenarios, Django's `request.is_secure()` method returns `False` even on requests that are actually secure, because the HTTPS connection is made to a front-end loadbalancer or reverse-proxy, and the internal proxied connection that Django sees is not HTTPS. Usually in these cases the proxy server provides an alternative header to indicate the secured external connection.

If this is your situation, you can set the *SECURE_PROXY_SSL_HEADER* setting to a tuple of ("header", "value"); if "header" is set to "value" in `request.META`, django-secure will tell Django to consider it a secure request (in other words, `request.is_secure()` will return `True` for this request). The "header" should be specified in the format it would be found in `request.META` (e.g. "HTTP_X_FORWARDED_PROTOCOL", not "X-Forwarded-Protocol"). For example:

```
SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTOCOL", "https")
```

Warning: If you set this to a header that your proxy allows through from the request unmodified (i.e. a header that can be spoofed), you are allowing an attacker to pretend that any request is secure, even if it is not. Make sure you only use a header that your proxy sets unconditionally, overriding any value from the request.

The `checksecure` management command

The `checksecure` management command is a "linter" for simple improvements you could make to your site's security configuration. It just runs a list of check functions. Each check function can return a set of warnings, or the empty set if it finds nothing to warn about.

- *When to run it*
- *Built-in checks*
- *Modifying the list of check functions*
- *Writing custom check functions*

When to run it

You can run it in your local development checkout. Your local dev settings module may not be configured for SSL, so you may want to point it at a different settings module, either by setting the `DJANGO_SETTINGS_MODULE` environment variable, or by passing the `--settings` option:

```
django-admin.py checksecure --settings=production_settings
```

Or you could run it directly on a production or staging deployment to verify that the correct settings are in use.

You could even make it part of your integration test suite, if you want. The `djangosecure.check.run_checks()` function runs all configured checks and returns the complete set of warnings; you could write a simple test that asserts that the returned value is empty.

Built-in checks

The following check functions are built-in to django-secure, and will run by default:

`djangosecure.check.djangosecure.`**`check_security_middleware`**`()`
Warns if *SecurityMiddleware* is not in your `MIDDLEWARE_CLASSES`.

`djangosecure.check.djangosecure.`**`check_sts`**`()`
Warns if *SECURE_HSTS_SECONDS* is not set to a non-zero value.

`djangosecure.check.djangosecure.`**`check_sts_include_subdomains`**`()`
Warns if *SECURE_HSTS_INCLUDE_SUBDOMAINS* is not `True`.

`djangosecure.check.djangosecure.`**`check_frame_deny`**`()`
Warns if *SECURE_FRAME_DENY* is not `True`.

`djangosecure.check.djangosecure.`**`check_content_type_nosniff`**`()`
Warns if *SECURE_CONTENT_TYPE_NOSNIFF* is not `True`.

`djangosecure.check.djangosecure.`**`check_xss_filter`**`()`
Warns if *SECURE_BROWSER_XSS_FILTER* is not `True`.

`djangosecure.check.djangosecure.`**`check_ssl_redirect`**`()`
Warns if *SECURE_SSL_REDIRECT* is not `True`.

`djangosecure.check.djangosecure.`**`check_secret_key`**`()`
Warns if SECRET_KEY is empty, missing, or has a very low number of different characters.

`djangosecure.check.sessions.`**`check_session_cookie_secure`**`()`
Warns if you appear to be using Django's session framework and the SESSION_COOKIE_SECURE setting is not `True`. This setting marks Django's session cookie as a secure cookie, which instructs browsers not to send it along with any insecure requests. Since it's trivial for a packet sniffer (e.g. Firesheep) to hijack a user's session if the session cookie is sent unencrypted, there's really no good excuse not to have this on. (It will prevent you from using sessions on insecure requests; that's a good thing).

`djangosecure.check.sessions.`**`check_session_cookie_httponly`**`()`
Warns if you appear to be using Django's session framework and the SESSION_COOKIE_HTTPONLY setting is not `True`. This setting marks Django's session cookie as "HTTPOnly", meaning (in supporting browsers) its value can't be accessed from client-side scripts. Turning this on makes it less trivial for an attacker to escalate a cross-site scripting vulnerability into full hijacking of a user's session. There's not much excuse for leaving this off, either: if your code depends on reading session cookies from Javascript, you're probably doing it wrong.

`djangosecure.check.csrf.`**`check_csrf_middleware`**`()`
Warns if you do not have Django's built-in CSRF protection enabled globally via the CSRF view middleware. It's important to CSRF protect any view that modifies server state; if you choose to do that piecemeal via the csrf_protect view decorator instead, just disable this check.

Suggestions for additional built-in checks (or better, patches implementing them) are welcome!

Modifying the list of check functions

By default, all of the *built-in checks* are run when you run `./manage.py checksecure`. However, some of these checks may not be appropriate for your particular deployment configuration. For instance, if you do your HTTP->HTTPS redirection in a loadbalancer, it'd be irritating for `checksecure` to constantly warn you about not having enabled *SECURE_SSL_REDIRECT*. You can customize the list of checks by setting the *SECURE_CHECKS* setting; you can just copy the default value and remove a check or two; you can also write your own *custom checks*.

Writing custom check functions

A `checksecure` check function can be any Python function that takes no arguments and returns a Python iterable of warnings (an empty iterable if it finds nothing to warn about).

Optionally, the function can have a `messages` attribute, which is a dictionary mapping short warning codes returned by the function (which will be displayed by `checksecure` if run with `--verbosity=0`) to longer explanations which will be displayed by `checksecure` when running at its default verbosity level. For instance:

```
from django.conf import settings

def check_dont_let_the_bad_guys_in():
    if settings.LET_THE_BAD_GUYS_IN:
        return ["BAD_GUYS_LET_IN"]
    return []

check_dont_let_the_bad_guys_in.messages = {
    "BAD_GUYS_LET_IN": (
        "Longer explanation of why it's a bad idea to let the bad guys in, "
        "and how to correct the situation.")
}
```

Settings Reference

- *SECURE_CHECKS*
- *SECURE_FRAME_DENY*
- *SECURE_HSTS_SECONDS*
- *SECURE_HSTS_INCLUDE_SUBDOMAINS*
- *SECURE_CONTENT_TYPE_NOSNIFF*
- *SECURE_BROWSER_XSS_FILTER*
- *SECURE_PROXY_SSL_HEADER*
- *SECURE_REDIRECT_EXEMPT*
- *SECURE_SSL_HOST*
- *SECURE_SSL_REDIRECT*

SECURE_CHECKS

A list of strings. Each string should be a Python dotted path to a function implementing a configuration check that will be run by the *checksecure management command*.

Defaults to:

```
[
    "djangosecure.check.csrf.check_csrf_middleware",
    "djangosecure.check.sessions.check_session_cookie_secure",
    "djangosecure.check.sessions.check_session_cookie_httponly",
    "djangosecure.check.djangosecure.check_security_middleware",
```

```
    "djangosecure.check.djangosecure.check_sts",
    "djangosecure.check.djangosecure.check_frame_deny",
    "djangosecure.check.djangosecure.check_content_type_nosniff",
    "djangosecure.check.djangosecure.check_xss_filter",
    "djangosecure.check.djangosecure.check_ssl_redirect",
]
```

SECURE_FRAME_DENY

Note: Django 1.4+ provides the same functionality via the X_FRAME_OPTIONS setting and XFrameOptionsMiddleware. You can use either this setting or Django's, there's no value in using both.

If set to `True`, causes *SecurityMiddleware* to set the *X-Frame-Options: DENY* header on all responses that do not already have that header (and where the view was not decorated with the `frame_deny_exempt` decorator).

Defaults to `False`.

SECURE_HSTS_SECONDS

If set to a non-zero integer value, causes *SecurityMiddleware* to set the *HTTP Strict Transport Security* header on all responses that do not already have that header.

Defaults to `0`.

SECURE_HSTS_INCLUDE_SUBDOMAINS

If `True`, causes *SecurityMiddleware* to add the `includeSubDomains` tag to the *HTTP Strict Transport Security* header.

Has no effect unless *SECURE_HSTS_SECONDS* is set to a non-zero value.

Defaults to `False` (only for backwards compatibility; in most cases if HSTS is used it should be set to `True`).

SECURE_CONTENT_TYPE_NOSNIFF

If set to `True`, causes *SecurityMiddleware* to set the *X-Content-Type-Options: nosniff* header on all responses that do not already have that header.

Defaults to `False`.

SECURE_BROWSER_XSS_FILTER

If set to `True`, causes *SecurityMiddleware* to set the *X-XSS-Protection: 1; mode=block* header on all responses that do not already have that header.

Defaults to `False`.

SECURE_PROXY_SSL_HEADER

Note: This setting is built-in to Django 1.4+. The Django setting works identically to this version.

A tuple of ("header", "value"); if "header" is set to "value" in `request.META`, django-secure will tell Django to consider this a secure request. For example:

```
SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTOCOL", "https")
```

See *Detecting proxied SSL* for more details.

Defaults to `None`.

Warning: If you set this to a header that your proxy allows through from the request unmodified (i.e. a header that can be spoofed), you are allowing an attacker to pretend that any request is secure, even if it is not. Make sure you only use a header that your proxy sets unconditionally, overriding any value from the request.

SECURE_REDIRECT_EXEMPT

Should be a list of regular expressions. Any URL path matching a regular expression in this list will not be redirected to HTTPS, if *SECURE_SSL_REDIRECT* is `True` (if it is `False` this setting has no effect).

Defaults to `[]`.

SECURE_SSL_HOST

If set to a string (e.g. `secure.example.com`), all SSL redirects will be directed to this host rather than the originally-requested host (e.g. `www.example.com`). If *SECURE_SSL_REDIRECT* is `False`, this setting has no effect.

Defaults to `None`.

SECURE_SSL_REDIRECT

If set to `True`, causes *SecurityMiddleware* to *redirect* all non-HTTPS requests to HTTPS (except for those URLs matching a regular expression listed in *SECURE_REDIRECT_EXEMPT*).

Note: If turning this to `True` causes infinite redirects, it probably means your site is running behind a proxy and can't tell which requests are secure and which are not. Your proxy likely sets a header to indicate secure requests; you can correct the problem by finding out what that header is and configuring the *SECURE_PROXY_SSL_HEADER* setting accordingly.

Defaults to `False`.

CHANGES

1.0.1 (2014.10.23)

- Hide django-secure tests from pre-1.6 Django test runners, to avoid breaking project tests.

1.0 (2013.04.17)

- BACKWARDS INCOMPATIBLE: Dropped tested support for Python 2.5, Django 1.2, and Django 1.3.
- Added support and testing for Python 3 (though all non-test code worked fine under Python 3 previously.)

0.1.3 (2013.04.17)

- Added check for `SECRET_KEY`. Thanks Ram Rachum.

0.1.2 (2012.04.13)

- Added the `SECURE_HSTS_INCLUDE_SUBDOMAINS` setting. Thanks Paul McMillan for the report and Donald Stufft for the patch. Fixes #13.
- Added the `X-XSS-Protection: 1; mode=block` header. Thanks Johannas Heller.

0.1.1 (2011.11.23)

- Added the `X-Content-Type-Options: nosniff` header. Thanks Johannas Heller.
- `SECURE_PROXY_SSL_HEADER` setting now patches `request.is_secure()` so it respects proxied SSL, to avoid redirects to http that should be to https.

0.1.0 (2011.05.29)

- Initial release.

TODO

Contributors

- Carl Meyer <carl@oddbird.net>
- Donald Stufft <donald.stufft@gmail.com>
- Johannas Heller <johann@phyfus.com>
- Ram Rachum <ram@rachum.com>

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

C

check_content_type_nosniff() (in module djangosecure.check.djangosecure), 9
check_csrf_middleware() (in module djangosecure.check.csrf), 9
check_frame_deny() (in module djangosecure.check.djangosecure), 9
check_secret_key() (in module djangosecure.check.djangosecure), 9
check_security_middleware() (in module djangosecure.check.djangosecure), 9
check_session_cookie_httponly() (in module djangosecure.check.sessions), 9
check_session_cookie_secure() (in module djangosecure.check.sessions), 9
check_ssl_redirect() (in module djangosecure.check.djangosecure), 9
check_sts() (in module djangosecure.check.djangosecure), 9
check_sts_include_subdomains() (in module djangosecure.check.djangosecure), 9
check_xss_filter() (in module djangosecure.check.djangosecure), 9