# django-debreach Documentation

***Release 1.4.1***

**Luke Pomfrey**

October 16, 2016

# Contents

**1 Installation** **3**

**2 Configuration** **5**
- 2.1 CSRF token masking (for Django < 1.10) . . . . . . 5
- 2.2 Content length modification . . . . . . 6

Basic/extra mitigation against the BREACH attack for Django projects.

When combined with rate limiting in your web-server, or by using something like django-ratelimit, the techniques here should provide at least some protection against the BREACH attack.

CHAPTER 1

# Installation

Install from PyPI using:

```
$ pip install django-debreach
```

Add to your *INSTALLED_APPS*:

```
INSTALLED_APPS = (
    ...
    'debreach',
    ...
)
```

CHAPTER 2

# Configuration

## 2.1 CSRF token masking (for Django < 1.10)

Django 1.10+ provides built-in support for masking CSRF tokens so you should use that. Including the middleware in a Django 1.10 project will raise an `ImproperlyConfigured` exception.

To mask CSRF tokens in the template add the `debreach.context_processors.csrf` context processor to the end of your *TEMPLATE_CONTEXT_PROCESSORS*:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'debreach.context_processors.csrf',
)
```

And add the `debreach.middleware.CSRFCryptMiddleware` to your middleware, *before* `django.middleware.csrf.CSRFMiddleware`:

```
MIDDLEWARE_CLASSES = (
    'debreach.middleware.CSRFCryptMiddleware',
    ...
    'django.middleware.csrf.CSRFMiddleware',
    ...
)
```

This works by xor-ing the CSRF token when it is added to the template, so that `{% csrf_token %}` now produces a hidden field with a value that is `"<random-string>$<actual-csrf-token-xor-ed-with-random-string>"`. Then, when the form is POSTed, the middleware xors the CSRF token back into it's original form. This ensures that the CSRF content is never the same between requests. If you are passing the token using the `X-CSRFToken` header (e.g. using XHR) that header will also be processed in the same way.

Note that values that are unchanged by django-debreach, or rather, don't contain a delimiting `$`, will be left unmodified. The middleware will also not operate on views marked as being exempt from CSRF protection using the `django.views.decorators.csrf.csrf_exempt` decorator.

### 2.1.1 CSRF protection using csrf_protect

If you don't use the CSRF middleware from django but, instead, apply the `django.views.decorators.csrf.csrf_protect` decorator to selected views, and don't want to use the `debreach.middleware.CSRFCryptMiddleware`, then you can use the `debreach.decorators.csrf_decrypt` decorator.

To use the `debreach.decorators.csrf_decrypt` decorator simply wrap your CSRF protected view with the decorator, like so:

```
@csrf_protect
@csrf_decrypt
def view(request, *args, **kwargs):
    return HttpResponse('')
```

## 2.2 Content length modification

django-debreach also enables you to counter the BREACH attack by randomising the content length of each response. This is acheived by adding a random string of between 12 and 25 characters as a comment to the end of the HTML content. Note that this will only be applied to responses with a content type of `text/html`.

To enable content length modification for all responses, add the `debreach.middleware.RandomCommentMiddleware` to the *start* of your middleware, but *after* the `GzipMiddleware` if you are using that.:

```
MIDDLEWARE_CLASSES = (
    'debreach.middleware.RandomCommentMiddleware',
    ...
)
```

or:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.gzip.GzipMiddleware',
    'debreach.middleware.RandomCommentMiddleware',
    ...
)
```

If you wish to disable this feature for selected views, simply apply the `debreach.decorators.random_comment_exempt` decorator to the view.

If you only want to protect a subset of views with content length modification then it may be easier to not use the middleware, but to selectively apply the `debreach.decorators.append_random_comment` decorator to the views you want protected.