Django

Documentation

Cross Site Request Forgery protection

The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries. This type of attack occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, 'login CSRF', where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.

The first defense against CSRF attacks is to ensure that GET requests (and other 'safe' methods, as defined by **RFC 7231#section-4.2.1**) are side effect free. Requests via 'unsafe' methods, such as POST, PUT, and DELETE, can then be protected by following the steps below.

How to use it

To take advantage of CSRF protection in your views, follow these steps:

1. The CSRF middleware is activated by default in the **`MIDDLEWARE_CLASSES`** setting. If you override that setting, remember that **`'django.middleware.csrf.CsrfViewMiddleware'`** should come before any view middleware that assume that CSRF attacks have been dealt with.

   If you disabled it, which is not recommended, you can use **`csrf_protect()`** on particular views you want to protect (see below).

2. In any template that uses a POST form, use the **`csrf_token`** tag inside the **`<form>`** element if the form is for an internal URL, e.g.:

```
<form action="" method="post">{% csrf_token %}
```

   This should not be done for POST forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability.

3. In the corresponding view functions, ensure that **`RequestContext`** is used to render the response so that **`{% csrf_token %}`** will work properly. If you're using the **`render()`** function, generic views, or contrib apps, you are covered already since these all use **`RequestContext`**.

AJAX

While the above method can be used for AJAX POST requests, it has some inconveniences: you have to remember to pass the CSRF token in as POST data with every POST request. For this reason, there is an alternative method: on each XMLHttpRequest, set a custom **`X-CSRFToken`** header to the value of the CSRF token. This is often easier, because many JavaScript frameworks provide hooks that allow headers to be set on every request.

As a first step, you must get the CSRF token itself. The recommended source for the token is the **`csrftoken`** cookie, which will be set if you've enabled CSRF protection for your views as outlined above.

Note

The CSRF token cookie is named **`csrftoken`** by default, but you can control the cookie name via the **`CSRF_COOKIE_NAME`** setting.

The CSRF header name is **`HTTP_X_CSRFTOKEN`** by default, but you can customize it using the **`CSRF_HEADER_NAME`** setting.

Acquiring the token is straightforward:

```
// using jQuery
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie != '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) == (name + '=')) {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}
var csrftoken = getCookie('csrftoken');
```

The above code could be simplified by using the JavaScript Cookie library to replace **`getCookie`**:

```
var csrftoken = Cookies.get('csrftoken');
```


Note

The CSRF token is also present in the DOM, but only if explicitly included using **`csrf_token`** in a template. The cookie contains the canonical token; the **`CsrfViewMiddleware`** will prefer the cookie to the token in the DOM. Regardless, you're guaranteed to have the cookie if the token is present in the DOM, so you should use the cookie!

Warning

If your view is not rendering a template containing the **`csrf_token`** template tag, Django might not set the CSRF token cookie. This is common in cases where forms are dynamically added to the page. To address this case, Django provides a view decorator which forces setting of the cookie: **`ensure_csrf_cookie()`**.

Finally, you'll have to actually set the header on your AJAX request, while protecting the CSRF token from being sent to other domains using settings.crossDomain in jQuery 1.5.1 and newer:

```
function csrfSafeMethod(method) {
    // these HTTP methods do not require CSRF protection
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken);
        }
    }
});
```

If you're using AngularJS 1.1.3 and newer, it's sufficient to configure the **`$http`** provider with the cookie and header names:

```
$httpProvider.defaults.xsrfCookieName = 'csrftoken';
$httpProvider.defaults.xsrfHeaderName = 'X-CSRFToken';
```

Other template engines

When using a different template engine than Django's built-in engine, you can set the token in your forms manually after making sure it's available in the template context.

For example, in the Jinja2 template language, your form could contain the following:

```
<div style="display:none">
    <input type="hidden" name="csrfmiddlewaretoken" value="{{ csrf_token }}">
</div>
```

You can use JavaScript similar to the AJAX code above to get the value of the CSRF token.

The decorator method

Rather than adding **`CsrfViewMiddleware`** as a blanket protection, you can use the **`csrf_protect`** decorator, which has exactly the same functionality, on particular views that need the protection. It must be used **both** on views that insert the CSRF token in the output, and on those that accept the POST form data. (These are often the same view function, but not always).

Use of the decorator by itself is **not recommended**, since if you forget to use it, you will have a security hole. The 'belt and braces' strategy of using both is fine, and will incur minimal overhead.

`csrf_protect`(*view*)

Decorator that provides the protection of **`CsrfViewMiddleware`** to a view.

Usage:

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

If you are using class-based views, you can refer to Decorating class-based views.

Rejected requests

By default, a '403 Forbidden' response is sent to the user if an incoming request fails the checks performed by **`CsrfViewMiddleware`**. This should usually only be seen when there is a genuine Cross Site Request Forgery, or when, due to a programming error, the CSRF token has not been included with a POST form.

The error page, however, is not very friendly, so you may want to provide your own view for handling this condition. To do this, simply set the **`CSRF_FAILURE_VIEW`** setting.

How it works

The CSRF protection is based on the following things:

1. A CSRF cookie that is set to a random value (a session independent nonce, as it is called), which other sites will not have access to.

This cookie is set by **`CsrfViewMiddleware`**. It is meant to be permanent, but since there is no way to set a cookie that never expires, it is sent with every response that has called **`django.middleware.csrf.get_token()`** (the function used internally to retrieve the CSRF token).

For security reasons, the value of the CSRF cookie is changed each time a user logs in.

2. A hidden form field with the name 'csrfmiddlewaretoken' present in all outgoing POST forms. The value of this field is the value of the CSRF cookie.

   This part is done by the template tag.

3. For all incoming requests that are not using HTTP GET, HEAD, OPTIONS or TRACE, a CSRF cookie must be present, and the 'csrfmiddlewaretoken' field must be present and correct. If it isn't, the user will get a 403 error.

   This check is done by **`CsrfViewMiddleware`**.

4. In addition, for HTTPS requests, strict referer checking is done by **`CsrfViewMiddleware`**. This means that even if a subdomain can set or modify cookies on your domain, it can't force a user to post to your application since that request won't come from your own exact domain.

   This also addresses a man-in-the-middle attack that's possible under HTTPS when using a session independent nonce, due to the fact that HTTP **`Set-Cookie`** headers are (unfortunately) accepted by clients even when they are talking to a site under HTTPS. (Referer checking is not done for HTTP requests because the presence of the **`Referer`** header isn't reliable enough under HTTP.)

   If the **`CSRF_COOKIE_DOMAIN`** setting is set, the referer is compared against it. This setting supports subdomains. For example, **`CSRF_COOKIE_DOMAIN = '.example.com'`** will allow POST requests from **`www.example.com`** and **`api.example.com`**. If the setting is not set, then the referer must match the HTTP **`Host`** header.

   Expanding the accepted referers beyond the current host or cookie domain can be done with the **`CSRF_TRUSTED_ORIGINS`** setting.

This ensures that only forms that have originated from trusted domains can be used to POST data back.

It deliberately ignores GET requests (and other requests that are defined as 'safe' by **RFC 7231**). These requests ought never to have any potentially dangerous side effects , and so a CSRF attack with a GET request ought to be harmless. **RFC 7231** defines POST, PUT, and DELETE as 'unsafe', and all other methods are also assumed to be unsafe, for maximum protection.

The CSRF protection cannot protect against man-in-the-middle attacks, so use HTTPS with HTTP Strict Transport Security. It also assumes validation of the HOST header and that there aren't any cross-site scripting vulnerabilities on your site (because XSS vulnerabilities already let an attacker do anything a CSRF vulnerability allows and much worse).

Changed in Django 1.9:

Checking against the **`CSRF_COOKIE_DOMAIN`** setting was added.

Caching

If the **`csrf_token`** template tag is used by a template (or the **`get_token`** function is called some other way), **`CsrfViewMiddleware`** will add a cookie and a **`Vary: Cookie`** header to the response. This means that the middleware will play well with the cache middleware if it is used as instructed (**`UpdateCacheMiddleware`** goes before all other middleware).

However, if you use cache decorators on individual views, the CSRF middleware will not yet have been able to set the Vary header or the CSRF cookie, and the response will be cached without either one. In this case, on any views that will require a CSRF token to be inserted you should use the **`django.views.decorators.csrf.csrf_protect()`** decorator first:

```
from django.views.decorators.cache import cache_page
from django.views.decorators.csrf import csrf_protect

@cache_page(60 * 15)
@csrf_protect
def my_view(request):
    ...
```

If you are using class-based views, you can refer to Decorating class-based views.

Testing

The **`CsrfViewMiddleware`** will usually be a big hindrance to testing view functions, due to the need for the CSRF token which must be sent with every POST request. For this reason, Django's HTTP client for tests has been modified to set a flag on requests which relaxes the middleware and the **`csrf_protect`** decorator so that they no longer rejects requests. In every other respect (e.g. sending cookies etc.), they behave the same.

If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks:

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

Limitations

Subdomains within a site will be able to set cookies on the client for the whole domain. By setting the cookie and using a corresponding token, subdomains will be able to circumvent the CSRF protection. The only way to avoid this is to ensure that subdomains are controlled by trusted users (or, are at least unable to set cookies). Note that even without CSRF, there are other vulnerabilities, such as session fixation, that make giving subdomains to untrusted parties a bad idea, and these vulnerabilities cannot easily be fixed with current browsers.

Edge cases

Certain views can have unusual requirements that mean they don't fit the normal pattern envisaged here. A number of utilities can be useful in these situations. The scenarios they might be needed in are described in the following section.

Utilities

The examples below assume you are using function-based views. If you are working with class-based views, you can refer to Decorating class-based views.

`csrf_exempt`(*view*)[source]

This decorator marks a view as being exempt from the protection ensured by the middleware. Example:

```
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse

@csrf_exempt
def my_view(request):
    return HttpResponse('Hello world')
```

`requires_csrf_token`(*view*)

Normally the **`csrf_token`** template tag will not work if **`CsrfViewMiddleware.process_view`** or an equivalent like **`csrf_protect`** has not run. The view decorator **`requires_csrf_token`** can be used to ensure the template tag does work. This decorator works similarly to **`csrf_protect`**, but never rejects an incoming request.

Example:

```
from django.views.decorators.csrf import requires_csrf_token
from django.shortcuts import render

@requires_csrf_token
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

`ensure_csrf_cookie`(*view*)

This decorator forces a view to send the CSRF cookie.

Scenarios

CSRF protection should be disabled for just a few views

Most views requires CSRF protection, but a few do not.

Solution: rather than disabling the middleware and applying **`csrf_protect`** to all the views that need it, enable the middleware and use **`csrf_exempt()`**.

CsrfViewMiddleware.process_view not used

There are cases when **`CsrfViewMiddleware.process_view`** may not have run before your view is run - 404 and 500 handlers, for example - but you still need the CSRF token in a form.

Solution: use **`requires_csrf_token()`**

Unprotected view needs the CSRF token

There may be some views that are unprotected and have been exempted by **`csrf_exempt`**, but still need to include the CSRF token.

Solution: use **`csrf_exempt()`** followed by **`requires_csrf_token()`**. (i.e. **`requires_csrf_token`** should be the innermost decorator).

View needs protection for one path

A view needs CSRF protection under one set of conditions only, and mustn't have it for the rest of the time.

Solution: use **`csrf_exempt()`** for the whole view function, and **`csrf_protect()`** for the path within it that needs protection. Example:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def my_view(request):

    @csrf_protect
    def protected_path(request):
        do_something()

    if some_condition():
       return protected_path(request)
    else:
       do_something_else()
```

Page uses AJAX without any HTML form

A page makes a POST request via AJAX, and the page does not have an HTML form with a **`csrf_token`** that would cause the required CSRF cookie to be sent.

Solution: use **`ensure_csrf_cookie()`** on the view that sends the page.

Contrib and reusable apps

Because it is possible for the developer to turn off the **`CsrfViewMiddleware`**, all relevant views in contrib apps use the **`csrf_protect`** decorator to ensure the security of these applications against CSRF. It is recommended that the developers of other reusable apps that want the same guarantees also use the **`csrf_protect`** decorator on their views.

Settings

A number of settings can be used to control Django's CSRF behavior:

- **CSRF_COOKIE_AGE**
- **CSRF_COOKIE_DOMAIN**
- **CSRF_COOKIE_HTTPONLY**
- **CSRF_COOKIE_NAME**
- **CSRF_COOKIE_PATH**
- **CSRF_COOKIE_SECURE**
- **CSRF_FAILURE_VIEW**
- **CSRF_HEADER_NAME**
- **CSRF_TRUSTED_ORIGINS**

Frequently Asked Questions

Is posting an arbitrary CSRF token pair (cookie and POST data) a vulnerability?

No, this is by design. Without a man-in-the-middle attack, there is no way for an attacker to send a CSRF token cookie to a victim's browser, so a successful attack would need to obtain the victim's browser's cookie via XSS or similar, in which case an attacker usually doesn't need CSRF attacks.

Some security audit tools flag this as a problem but as mentioned before, an attacker cannot steal a user's browser's CSRF cookie. "Stealing" or modifying *your own* token using Firebug, Chrome dev tools, etc. isn't a vulnerability.

Is the fact that Django's CSRF protection isn't linked to a session a problem?

No, this is by design. Not linking CSRF protection to a session allows using the protection on sites such as a *pastebin* that allow submissions from anonymous users which don't have a session.

Why not use a new token for each request?

Generating a new token for each request is problematic from a UI perspective because it invalidates all previous forms. Most users would be very unhappy to find that opening a new tab on your site has invalidated the form they had just spent time filling out in another tab or that a form they accessed via the back button could not be filled out.

Why might a user encounter a CSRF validation failure after logging in?

For security reasons, CSRF tokens are rotated each time a user logs in. Any page with a form generated before a login will have an old, invalid CSRF token and need to be reloaded. This might happen if a user uses the back button after a login or if they log in in a different browser tab.

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List