Security in Django

This document is an overview of Django's security features. It includes advice on securing a Django-powered site.

Cross site scripting (XSS) protection

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

```
<style class={{ var }}>...</style>
```

If `var` is set to `'class1 onmouseover=javascript:func()'`, this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML. (Quoting the attribute value would fix this case.)

It is also important to be particularly careful when using `is_safe` with custom template tags, the `safe` template tag, `mark_safe`, and when autoescape is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.

Django has built-in protection against most types of CSRF attacks, providing you have enabled and used it where appropriate. However, as with any mitigation technique, there are limitations. For example, it is possible to disable the CSRF module globally or for particular views. You should only do this if you know what you are doing. There are other limitations if your site has subdomains that are outside of your control.

CSRF protection works by checking for a secret in each POST request. This ensures that a malicious user cannot "replay" a form POST to your website and have another logged in user unwittingly submit that form. The malicious user would have to know the secret, which is user specific (using a cookie).

When deployed with HTTPS, `CsrfViewMiddleware` will check that the HTTP referer header is set to a URL on the same origin (including subdomain and port). Because HTTPS provides additional security, it is imperative to ensure connections use HTTPS where it is available by forwarding insecure connection requests and using HSTS for supported browsers.

Be very careful with marking views with the `csrf_exempt` decorator unless it is absolutely necessary.

SQL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

Django's querysets are protected from SQL injection since their queries are constructed using query parameterization. A query's SQL code is defined separately from the query's parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.

Django also gives developers power to write raw queries or execute custom sql. These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using `extra()` and `RawSQL`.

Clickjacking protection

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site.

Django contains clickjacking protection in the form of the `X-Frame-Options middleware` which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.

The middleware is strongly recommended for any site that does not need to have its pages wrapped in a frame by third party sites, or only needs to allow that for a small section of the site.

SSL/HTTPS

It is always better for security to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information transferred between client and server, and in some cases – **active** network attackers – to alter data that is sent in either direction.

If you want the protection that HTTPS provides, and have enabled it on your server, there are some additional steps you may need:

- If necessary, set `SECURE_PROXY_SSL_HEADER`, ensuring that you have understood the warnings there thoroughly. Failure to do this can result in CSRF vulnerabilities, and failure to do it correctly can also be dangerous!
- Set `SECURE_SSL_REDIRECT` to `True`, so that requests over HTTP are redirected to HTTPS.

  Please note the caveats under `SECURE_PROXY_SSL_HEADER`. For the case of a reverse proxy, it may be easier or more secure to configure the main web server to do the redirect to HTTPS.
- Use 'secure' cookies.

  If a browser connects initially via HTTP, which is the default for most browsers, it is possible for existing cookies to be leaked. For this reason, you should set your `SESSION_COOKIE_SECURE` and `CSRF_COOKIE_SECURE` settings to `True`. This instructs the browser to only send these cookies over HTTPS connections. Note that this will mean that sessions will not work over HTTP, and the CSRF protection will prevent any POST data being accepted over HTTP (which will be fine if you are redirecting all HTTP traffic to HTTPS).
- Use HTTP Strict Transport Security (HSTS)

  HSTS is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS. Combined with redirecting requests over HTTP to HTTPS, this will ensure that connections always enjoy the added security of SSL provided one successful connection has occurred. HSTS may either be configured with `SECURE_HSTS_SECONDS`, `SECURE_HSTS_INCLUDE_SUBDOMAINS`, and `SECURE_HSTS_PRELOAD`, or on the web server.

Host header validation

Django uses the `Host` header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, a fake `Host` value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails.

Because even seemingly-secure web server configurations are susceptible to fake `Host` headers, Django validates `Host` headers against the `ALLOWED_HOSTS` setting in the `django.http.HttpRequest.get_host()` method.

This validation only applies via `get_host()`; if your code accesses the `Host` header directly from `request.META` you are bypassing this security protection.

For more details see the full `ALLOWED_HOSTS` documentation.

Warning: Previous versions of this document recommended configuring your web server to ensure it validates incoming HTTP `Host` headers. While this is still recommended, in many common web servers a configuration that seems to validate the `Host` header may not in fact do so. For instance, even if Apache is configured such that your Django site is served from a non-default virtual host with the `ServerName` set, it is still possible for an HTTP request to match this virtual host and supply a fake `Host` header. Thus, Django now requires that you set `ALLOWED_HOSTS` explicitly rather than relying on web server configuration.

Additionally, Django requires you to explicitly enable support for the `X-Forwarded-Host` header (via the `USE_X_FORWARDED_HOST` setting) if your configuration requires it.

Referrer policy

Browsers use the `Referer` header as a way to send information to a site about how users got there. By setting a *Referrer Policy* you can help to protect the privacy of your users, restricting under which circumstances the `Referer` header is set. See the referrer policy section of the security middleware reference for details.

Cross-origin opener policy

The cross-origin opener policy (COOP) header allows browsers to isolate a top-level window from other documents by putting them in a different context group so that they cannot directly interact with the top-level window. If a document protected by COOP opens a cross-origin popup window, the popup's `window.opener` property will be `null`. COOP protects against cross-origin attacks. See the cross-origin opener policy section of the security middleware reference for details.

Session security

Similar to the CSRF limitations requiring a site to be deployed such that untrusted users don't have access to any subdomains, `django.contrib.sessions` also has limitations. See the session topic guide section on security for details.

User-uploaded content

Note: Consider serving static files from a cloud service or CDN to avoid some of these issues.

- If your site accepts file uploads, it is strongly advised that you limit these uploads in your web server configuration to a reasonable size in order to prevent denial of service (DOS) attacks. In Apache, this can be easily set using the LimitRequestBody directive.
- If you are serving your own static files, be sure that handlers like Apache's `mod_php`, which would execute static files as code, are disabled. You don't want users to be able to execute arbitrary code by uploading and requesting a specially crafted file.
- Django's media upload handling poses some vulnerabilities when that media is served in ways that do not follow security best practices. Specifically, an HTML file can be uploaded as an image if that file contains a valid PNG header followed by malicious HTML. This file will pass verification of the library that Django uses for `ImageField` image processing (Pillow). When this file is subsequently displayed to a user, it may be displayed as HTML depending on the type and configuration of your web server.

  No bulletproof technical solution exists at the framework level to safely validate all user uploaded file content, however, there are some other steps you can take to mitigate these attacks:

  1. One class of attacks can be prevented by always serving user uploaded content from a distinct top-level or second-level domain. This prevents any exploit blocked by same-origin policy protections such as cross site scripting. For example, if your site runs on `example.com`, you would want to serve uploaded content (the `MEDIA_URL` setting) from something like `usercontent-example.com`. It's *not* sufficient to serve content from a subdomain like `usercontent.example.com`.
  2. Beyond this, applications may choose to define a list of allowable file extensions for user uploaded files and configure the web server to only serve such files.

Additional security topics

While Django provides good security protection out of the box, it is still important to properly deploy your application and take advantage of the security protection of the web server, operating system and other components.

- Make sure that your Python code is outside of the web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).
- Take care with any user uploaded files.
- Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or web server module to throttle these requests.
- Keep your `SECRET_KEY`, and `SECRET_KEY_FALLBACKS` if in use, secret.
- It is a good idea to limit the accessibility of your caching system and database using a firewall.
- Take a look at the Open Web Application Security Project (OWASP) Top 10 list which identifies some common vulnerabilities in web applications. While Django has tools to address some of the issues, other issues must be accounted for in the design of your project.
- Mozilla discusses various topics regarding web security. Their pages also include security principles that apply to any system.