Security in Django

1

THE THEME OF WEB SECURITY

- Never — under any circumstances — trust data from the browser.
- You *never* know who's on the other side of that HTTP connection. It might be one of your users, but it just as easily could be a nefarious cracker looking for an opening.
- Any data of any nature that comes from the browser needs to be treated with a healthy dose of paranoia.
- This includes data that's both "in band" (i.e., submitted from Web forms) and "out of band" (i.e., HTTP headers, cookies, and other request information).

2

Django's built in security features

- Ensuring that the sites you build are secure is of the utmost importance to a professional web applications developer.
- The Django framework is very mature and the majority of common security issues are addressed in some way by the framework itself, however no security measure is 100% guaranteed and there are new threats emerging all the time, so it's up to you as a web developer to ensure that your websites and applications are secure.
- This chapter includes an overview of Django's security features and advice on securing a Django powered site that will protect your sites 99% of the time, but it's up to you to keep abreast of changes in web security.

3

Cross Site Scripting (XSS)

- **Cross Site Scripting** (**XSS**) is found in Web applications that fail to escape user-submitted content properly before rendering it into HTML.
- This allows an attacker to insert arbitrary HTML into your Web page, usually in the form of <script> tags --- that is, inject client side scripts into the browsers of other users.
- Attackers often use XSS attacks to steal cookie and session information, or to trick users into giving private information to the wrong person (aka *phishing*).
- Here's a typical example. Consider this extremely simple "Hello, World" view

```
def say_hello(request):
    name = request.GET.get('name', 'world')
    return render_to_response("hello.html", {"name" : name})
```

- We might write a template for this view as follows: <h1>Hello, {{ name }}!</h1>

4

Cross Site Scripting (cont'd)

- So if we accessed http://example.com/hello/name=Jacob, the rendered page would contain this <h1>Hello, Jacob!</h1>
- But what happens if we access http://example.com/hello/name=Jacob? Then we get this <h1>Hello, Jacob!</h1>
- See, an attacker can easily insert <script> tags that will run some client-side script!
- The problem gets worse if you store this data (malicious scripts) in the database and later 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.
- Using Django templates protects you against the majority of XSS attacks. Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof.

5

Cross Site Request Forgery (CSRF)

- **Cross Site Request Forgery** (**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.
- CSRF protection works by checking for a nonce in each POST request. This ensures that a malicious user cannot simply 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 nonce, which is user specific (using a cookie).

6

CSRF protection

- Where a POST form includes the csrf_token tag, and the view concerned passes RequestContext to the template, requesting the page means Django includes a hidden form field which contains an alphanumeric string. Django also returns to the browser a cookie with the name set to csrftoken and value set to the same alphanumeric string.
- This ensures that only forms that have originated from your Web site can be used to POST data back

7

CSRF protection -- how it works

- A CSRF cookie is set to a random value (a session independent nonce), which other sites will not have access to.
- 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.
- 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.
- This ensures that only forms that have originated from your Web site can be used to POST data back.

8

SQL injection

- 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.
- This vulnerability most commonly crops up when constructing SQL "by hand" from user input. For example, imagine writing a function to gather a list of contact information from a contact search page. To prevent spammers from reading every single email in our system, we'll force the user to type in someone's username before providing her email address:

```
def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = '%s';" % username
    # execute the SQL here...
```

9

SQL injection (cont'd)

- At first this doesn't look dangerous, it really is.
- First, our attempt at protecting our entire email list will fail with a cleverly constructed query. Think about what happens if an attacker types "' OR 'a'='a" into the query box. In that case, the query that the string interpolation will construct will be:

```
SELECT * FROM user_contacts WHERE username = '' OR 'a' = 'a';
```

- Because we allowed unsecured SQL into the string, the attacker's added OR clause ensures that every single row is returned.
- Imagine what will happen if the attacker submits "'; DELETE FROM user_contacts WHERE 'a' = 'a'".

```
SELECT * FROM user_contacts WHERE username = ''; DELETE FROM
user_contacts WHERE 'a' = 'a';
```

10

SQL injection protection

- Although this problem is sometimes hard to spot, the solution is simple: *never* trust user-submitted data, and *always* escape it when passing it into SQL.
- By using Django's querysets, the resulting SQL will be properly escaped by the underlying database driver.
- However, 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.

11

SESSION HIJACKING

This is a general class of attacks on a user's session data. It can take a number of different forms:

- A *man-in-the-middle* attack, where an attacker snoops on session data as it travels over the wire (or wireless) network.
- *Session forging*, where an attacker uses a session ID (perhaps obtained through a man-in-the-middle attack) to pretend to be another user. An example of these first two would be an attacker in a coffee shop using the shop's wireless network to capture a session cookie. She could then use that cookie to impersonate the original user.

12

SESSION HIJACKING (cont'd)

- *Session fixation*, where an attacker tricks a user into setting or reseting the user's session ID. For example, PHP allows session identifiers to be passed in the URL (e.g., http://example.com/?PHPSESSID=fa90197ca25f6ab40bb1374c510d7. An attacker who tricks a user into clicking a link with a hard-coded session ID will cause the user to pick up that session.
  Session fixation has been used in phishing attacks to trick users into entering personal information into an account the attacker owns. He can later log into that account and retrieve the data.

13

SESSION HIJACKING -- The Solution

There are a number of general principles that can protect you from these attacks:

- Never allow session information to be contained in the URL. Django's session framework simply doesn't allow sessions to be contained in the URL.
- Don't store data in cookies directly; instead, store a session ID that maps to session data stored on the back-end.
- Remember to escape session data if you display it in the template
- Prevent attackers from spoofing session IDs whenever possible. Although it's nearly impossible to detect someone who's hijacked a session ID, Django does have built-in protection against a brute-force session attack.
- Session IDs are stored as hashes (instead of sequential numbers), which prevents a brute-force attack, and a user will always get a new session ID if she tries a nonexistent one, which prevents session fixation.

14

SESSION HIJACKING -- The Solution

- Notice that none of those principles and tools prevents man-in-the-middle attacks.
- These types of attacks are nearly impossible to detect. If your site allows logged-in users to see any sort of sensitive data, you should *always* serve that site over HTTPS.
- Additionally, if you have an SSL-enabled site, you should set the SESSION_COOKIE_SECURE setting to True; this will make Django only send session cookies over HTTPS.

15

Clickjacking protection

- Clickjacking is a type of attack where a malicious site wraps another site in a frame. This type of attack occurs when a malicious site tricks a user into clicking on a concealed element of another site which they have loaded in a hidden frame or iframe.
- 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.

16

SSL/HTTPS

- It is always better for security, though not always practical in all cases, 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.

17

EXPOSED ERROR MESSAGES

- During development, being able to see tracebacks and errors live in your browser is extremely useful. Django has “pretty” and informative debug messages specifically to make debugging easier.
- However, if these errors get displayed once the site goes live, they can reveal aspects of your code or configuration that could aid an attacker. Django’s philosophy is that site visitors should never see application-related error messages.
- Naturally, of course, developers need to see tracebacks to debug problems in their code. So the framework should hide all error messages from the public, but it should display them to the trusted site developers.
- **The Solution:** uses the simple flag that controls the display of these error messages, i.e. the DEBUG setting.

18

Archive of security issues

- Django's development team is strongly committed to responsible reporting and disclosure of security-related issues, as outlined in Django's security policies.
- As part of that commitment, they maintain an historical list of issues which have been fixed and disclosed.
- For the up to date list, see the archive of security issues
  https://docs.djangoproject.com/en/3.0/releases/security/