Chapter 20: Security

The Internet can be a scary place.

These days, high-profile security gaffes seem to crop up on a daily basis. We've seen viruses spread with amazing speed, swarms of compromised computers wielded as weapons, a never-ending arms race against spammers, and many, many reports of identify theft from hacked Web sites.

As Web developers, we have a duty to do what we can to combat these forces of darkness. Every Web developer needs to treat security as a fundamental aspect of Web programming. Unfortunately, it turns out that implementing security is *hard* – attackers need to find only a single vulnerability, but defenders have to protect every single one.

Django attempts to mitigate this difficulty. It's designed to automatically protect you from many of the common security mistakes that new (and even experienced) Web developers make. Still, it's important to understand what these problems are, how Django protects you, and – most important – the steps you can take to make your code even more secure.

First, though, an important disclaimer: We do not intend to present a definitive guide to every known Web security exploit, and so we won't try to explain each vulnerability in a comprehensive manner. Instead, we'll give a short synopsis of security problems as they apply to Django.

The Theme of Web Security

If you learn only one thing from this chapter, let it be this:

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). It's trivial to spoof the request metadata that browsers usually add automatically.

Every one of the vulnerabilities discussed in this chapter stems directly from trusting data that comes over the wire and then failing to sanitize that data before using it. You should make it a general practice to continuously ask, "Where does this data come from?"

SQL Injection

SQL injection is a common exploit in which an attacker alters Web page parameters (such as `GET`/`POST` data or URLs) to insert arbitrary SQL snippets that a naive Web application executes in its database directly. It's probably the most dangerous – and, unfortunately, one of the most common – vulnerabilities out there.

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 e-mail in our system, we'll force the user to type in someone's username before providing her e-mail address:

```
def user_contacts(request):

    user = request.GET['username']

    sql = "SELECT * FROM user_contacts WHERE username =
'%s';" % username

    # execute the SQL here...
```

Note

In this example, and all similar "don't do this" examples that follow, we've deliberately left out most of the code needed to make the functions actually work. We don't want this code to work if someone accidentally takes it out of context.

Though at first this doesn't look dangerous, it really is.

First, our attempt at protecting our entire e-mail 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.

However, that's the *least* scary attack. Imagine what will happen if the attacker submits`"'; DELETE FROM user_contacts WHERE 'a' = 'a"`. We'll end up with this complete query:

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

Yikes! Our entire contact list would be deleted instantly.

The Solution

Although this problem is insidious and sometimes hard to spot, the solution is simple: *never* trust user-submitted data, and *always* escape it when passing it into SQL.

The Django database API does this for you. It automatically escapes all special SQL parameters, according to the quoting conventions of the database server you're using (e.g., PostgreSQL or MySQL).

For example, in this API call:

```
foo.get_list(bar__exact="' OR 1=1")
```

Django will escape the input accordingly, resulting in a statement like this:

```
SELECT * FROM foos WHERE bar = '\' OR 1=1'
```

Completely harmless.

This applies to the entire Django database API, with a couple of exceptions:

- The `where` argument to the `extra()` method. (See Appendix C.) That parameter accepts raw SQL by design.
- Queries done "by hand" using the lower-level database API. (See Chapter 10.)

In each of these cases, it's easy to keep yourself protected. In each case, avoid string interpolation in favor of passing in *bind parameters*. That is, the example we started this section with should be written as follows:

```
from django.db import connection


def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username =
%s"
    cursor = connection.cursor()
    cursor.execute(sql, [user])
    # ... do something with the results
```

The low-level `execute` method takes a SQL string with `%s` placeholders and automatically escapes and inserts parameters from the list passed as the second argument. You should *always* construct custom SQL this way.

Unfortunately, you can't use bind parameters everywhere in SQL; they're not allowed as identifiers (i.e., table or column names). Thus, if you need to, say, dynamically construct a list of tables from a `POST` variable, you'll need to escape that name in your code. Django provides a function,`django.db.connection.ops.quote_name`, which will escape the identifier according to the current database's quoting scheme.

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.

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*).

This type of attack can take a number of different forms and has almost infinite permutations, so we'll just look at a typical example. Consider this extremely simple "Hello, World" view:

```
from django.http import HttpResponse


def say_hello(request):
    name = request.GET.get('name', 'world')
    return HttpResponse('<h1>Hello, %s!</h1>' % name)
```

This view simply reads a name from a `GET` parameter and passes that name into the generated HTML. So, if we accessed `http://example.com/hello/?name=Jacob`, the page would contain this:

```
<h1>Hello, Jacob!</h1>
```

But wait – what happens if we access `http://example.com/hello/?name=<i>Jacob</i>`? Then we get this:

```
<h1>Hello, <i>Jacob</i>!</h1>
```

Of course, an attacker wouldn't use something as benign as `<i>` tags; he could include a whole set of HTML that hijacked your page with arbitrary content. This type of attack has been used to trick users into entering data into what looks like their bank's Web site, but in fact is an XSS-hijacked form that submits their back account information to an attacker.

The problem gets worse if you store this data in the database and later display it it on your site. For example, MySpace was once found to be vulnerable to an XSS attack of this nature. A user inserted JavaScript into his profile that automatically added him as your friend when you visited his profile page. Within a few days, he had millions of friends.

Now, this may sound relatively benign, but keep in mind that this attacker managed to get *his* code – not MySpace's – running on *your* computer. This violates the assumed trust that all the code on MySpace is actually written by MySpace.

MySpace was extremely lucky that this malicious code didn't automatically delete viewers' accounts, change their passwords, flood the site with spam, or any of the other nightmare scenarios this vulnerability unleashes.

The Solution

The solution is simple: *always* escape *any* content that might have come from a user before inserting it into HTML.

To guard against this, Django's template system automatically escapes all variable values. Let's see what happens if we rewrite our example using the template system:

```
# views.py


from django.shortcuts import render


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


# hello.html


<h1>Hello, {{ name }}!</h1>
```

With this in place, a request to `http://example.com/hello/name=<i>Jacob</i>` will result in the following page:

```
<h1>Hello, &lt;i&gt;Jacob&lt;/i&gt;!</h1>
```

We covered Django's auto-escaping back in Chapter 4, along with ways to turn it off. But even if you're using this feature, you should *still* get in the habit of asking yourself, at all times, "Where does this data come from?" No automatic solution will ever protect your site from XSS attacks 100% of the time.

Cross-Site Request Forgery

Cross-site request forgery (CSRF) happens when a malicious Web site tricks users into unknowingly loading a URL from a site at which they're already authenticated – hence taking advantage of their authenticated status.

Django has built-in tools to protect from this kind of attack. Both the attack itself and those tools are covered in great detail in Chapter 16.

Session Forging/Hijacking

This isn't a specific attack, but rather 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.

- A *cookie-forging* attack, where an attacker overrides the supposedly read-only data stored in a cookie.Chapter 14 explains in detail how cookies work, and one of the salient points is that it's trivial for browsers and malicious users to change cookies without your knowledge.

  There's a long history of Web sites that have stored a cookie like `IsLoggedIn=1` or even`LoggedInAsUser=jacob`. It's dead simple to exploit these types of cookies.

  On a more subtle level, though, it's never a good idea to trust anything stored in cookies. You never know who's been poking at them.

- *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=fa90197ca25f6ab40bb1374c510d7a32`). 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.

- *Session poisoning*, where an attacker injects potentially dangerous data into a user's session – usually through a Web form that the user submits to set session data.

  A canonical example is a site that stores a simple user preference (like a page's background color) in a cookie. An attacker could trick a user into clicking a link to submit a "color" that actually contains an XSS attack. If that color isn't escaped, the user could again inject malicious code into the user's environment.

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 (see Chapter 14) 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 backend.

If you use Django's built-in session framework (i.e., `request.session`), this is handled automatically for you. The only cookie that the session framework uses is a single session ID; all the session data is stored in the database.

- Remember to escape session data if you display it in the template. See the earlier XSS section, and remember that it applies to any user-created content as well as any data from the browser. You should treat session information as being user created.

- 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.

  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.

E-mail Header Injection

SQL injection's less well-known sibling, *e-mail header injection*, hijacks Web forms that send e-mail. An attacker can use this technique to send spam via your mail server. Any form that constructs e-mail headers from Web form data is vulnerable to this kind of attack.

Let's look at the canonical contact form found on many sites. Usually this sends a message to a hard-coded e-mail address and, hence, doesn't appear vulnerable to spam abuse at first glance.

However, most of these forms also allow the user to type in his own subject for the e-mail (along with a "from" address, body, and sometimes a few other fields). This subject field is used to construct the "subject" header of the e-mail message.

If that header is unescaped when building the e-mail message, an attacker could submit something like`"hello\ncc:spamvictim@example.com"` (where `"\n` is a newline character). That would make the constructed e-mail headers turn into:

```
To: hardcoded@example.com

Subject: hello

cc: spamvictim@example.com
```

Like SQL injection, if we trust the subject line given by the user, we'll allow him to construct a malicious set of headers, and he can use our contact form to send spam.

The Solution

We can prevent this attack in the same way we prevent SQL injection: always escape or validate user-submitted content.

Django's built-in mail functions (in `django.core.mail`) simply do not allow newlines in any fields used to construct headers (the from and to addresses, plus the subject). If you try to use`django.core.mail.send_mail` with a subject that contains newlines, Django will raise a `BadHeaderError`exception.

If you do not use Django's built-in mail functions to send e-mail, you'll need to make sure that newlines in headers either cause an error or are stripped. You may want to examine the `SafeMIMEText` class in`django.core.mail` to see how Django does this.

Directory Traversal

Directory traversal is another injection-style attack, wherein a malicious user tricks filesystem code into reading and/or writing files that the Web server shouldn't have access to.

An example might be a view that reads files from the disk without carefully sanitizing the file name:

```
def dump_file(request):

    filename = request.GET["filename"]

    filename = os.path.join(BASE_PATH, filename)

    content = open(filename).read()


    # ...
```

Though it looks like that view restricts file access to files beneath `BASE_PATH` (by using `os.path.join`), if the attacker passes in a `filename` containing `..` (that's two periods, a shorthand for "the parent directory"), she can access files "above" `BASE_PATH`. It's only a matter of time before she can discover the correct number of dots to successfully access, say, `../../../../../etc/passwd`.

Anything that reads files without proper escaping is vulnerable to this problem. Views that *write* files are just as vulnerable, but the consequences are doubly dire.

Another permutation of this problem lies in code that dynamically loads modules based on the URL or other request information. A well-publicized example came from the world of Ruby on Rails. Prior to mid-2006, Rails used URLs like `http://example.com/person/poke/1` directly to load modules and call methods. The result was that a carefully constructed URL could automatically load arbitrary code, including a database reset script!

The Solution

If your code ever needs to read or write files based on user input, you need to sanitize the requested path very carefully to ensure that an attacker isn't able to escape from the base directory you're restricting access to.

Note

Needless to say, you should *never* write code that can read from any area of the disk!

A good example of how to do this escaping lies in Django's built-in static content-serving view (in`django.views.static`). Here's the relevant code:

```
import os
import posixpath

# ...

path = posixpath.normpath(urllib.unquote(path))
newpath = ''
for part in path.split('/'):
    if not part:
        # strip empty path components
        continue

    drive, part = os.path.splitdrive(part)
    head, part = os.path.split(part)
    if part in (os.curdir, os.pardir):
        # strip '.' and '..' in path
        continue

    newpath = os.path.join(newpath, part).replace('\\',
'/')
```

Django doesn't read files (unless you use the `static.serve` function, but that's protected with the code just shown), so this vulnerability doesn't affect the core code much.

In addition, the use of the URLconf abstraction means that Django will *never* load code you've not explicitly told it to load. There's no way to create a URL that causes Django to load something not mentioned in a URLconf.

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.

Furthermore, errors and tracebacks aren't at all useful to end users. Django's philosophy is that site visitors should never see application-related error messages. If your code raises an unhandled exception, a site visitor should not see the full traceback – or *any* hint of code snippets or Python (programmer-oriented) error messages. Instead, the visitor should see a friendly "This page is unavailable" message.

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

As we covered in Chapter 12, Django's `DEBUG` setting controls the display of these error messages. Make sure to set this to `False` when you're ready to deploy.

Users deploying under Apache and mod_python (also see Chapter 12) should also make sure they have`PythonDebug Off` in their Apache conf files; this will suppress any errors that occur before Django has had a chance to load.

A Final Word on Security

We hope all this talk of security problems isn't too intimidating. It's true that the Web can be a wild world, but with a little bit of foresight, you can have a secure Web site.

Keep in mind that Web security is a constantly changing field; if you're reading the dead-tree version of this book, be sure to check more up to date security resources for any new vulnerabilities that have been discovered. In fact, it's always a good idea to spend some time each week or month researching and keeping current on the state of Web application security. It's a small investment to make, but the protection you'll get for your site and your users is priceless.