SECURE WEB DEVELOPMENT

with **django**

James Bennett · DjangoCon US · July 17th 2016

THERE'S NO SUCH THING AS "SECURE"

Let's just get that out of the way right now. It's an important enough idea that this slide is YELLING *about it in* ALL CAPS.

OK, SMART GUY, SO WHAT'S THIS TALK ABOUT, THEN?

And why is it still YELLING*?*

- Useful ways to think and talk about security, and bring it into your development process
- Security issues in web applications
- How to deal with those issues (and how Django and Python will help you)
- Django's security history, and learning from our mistakes

LET'S TALK ABOUT TALKING ABOUT SECURITY

So meta.

SECURITY IS IMPORTANT

```
'); DROP TABLE slides;--
```

SECURITY IS NOT AN ABSOLUTE

Only a Sith deals in absolutes. And their bases keep getting blown up. You don't want your base blown up. So don't be like the Sith, is what I'm saying.

SECURITY IS ABOUT TRADEOFFS

The sun becoming a red giant and consuming the world is a very effective denial-of-service attack, but you probably shouldn’t worry about it.

Probably.

SECURITY ISN'T ONLY FOR EXPERTS

What do they know, anyway?

SECURITY CAN'T BE AN AFTERTHOUGHT

We'll deal with that next quarter. Wait, why is our bank account suddenly empty?

THE OWASP TOP TEN

O wasp, where art thou?

A list of the top ten security issues in web applications:

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

INJECTION ATTACKS

Injection attacks occur when an application inappropriately trusts data from an untrustworthy source.

```
username = request.GET['username']
my_query = """
SELECT *
FROM users
WHERE username = '%s'
""" % username
result = db_cursor.execute(my_query)
```

HI! MY NAME IS

```
''; UPDATE users SET
is_superuser = true WHERE
username = 'haxor'; --
```

This example is a *SQL injection* attack. SQL injection vulnerabilities are common any time queries are constructed with user input as a parameter.

OTHER FORMS OF INJECTION

- *Mail header injection* attacks: an email-sending form can be tricked into sending to unintended recipients
- *Command injection* attacks: an application which triggers shell commands with user input as a parameter can be tricked into executing other, arbitrary shell commands
- *XML injection* attacks: an XML processor is tricked into handling unsafe input (perhaps containing scripts, entity definitions which expand to perform network requests or reveal contents of system files, etc.)
- Almost anything which takes user input and does something with that input

AUTHENTICATION AND SESSION MANAGEMENT

Authentication and sessions are *extremely* *hard* to get right.

JUST A FEW OF THE MANY WAYS IT CAN GO WRONG

- Credentials (such as passwords) are insufficiently protected on either the server or the client side
- Credentials are transmitted over unencrypted connections
- Credentials can be reset/overwritten too easily
- Identifiers are exposed to public view (i.e., session ID in a GET parameter)
- Session hijacking/fixation: attacker can get a valid session ID and use it or force it to be reused

CROSS-SITE SCRIPTING

But my friends call me “XSS”.

An application constructs HTML by concatenating or interpolating strings that include user input (either the current user, or a previous user's stored input), allowing unsafe content — such as JavaScript code — to wind up in the output.

```
name = request.GET['name']
my_html = '<p>Hello, there, %s</p>' % name
```

HI! MY NAME IS

```
<script type="text/javascript">
    alert("Oops");</script>
```

INSECURE DIRECT OBJECT REFERENCE

http://example.com/accounts/manage/1

Hmm...

Wonder what happens at

http://example.com/accounts/manage/2

or

http://example.com/accounts/manage/528

This issue is more subtle, because it's usually exploited in combination with something else (such as lack of appropriate access controls), though by itself it can leak information you might not have wanted to leak.

It’s also tricky to point out and fix, since the obvious “solution” is to introduce security through obscurity (for example, through randomly-generated instead of sequential identifiers).

MISCONFIGURATION

The default password is “admin”. Remember to change it after you log in the first time!

PLACES TO CHECK ON

- Default accounts/credentials or authentication bypasses
- Debugging modes for all pieces of software (not just Django)
- Security-related or security-relevant settings for all software you use
- Default error-handling behaviors — stack traces are gold mines of information about your application

SENSITIVE DATA EXPOSURE

Unsalted MD5 was good enough for our ancestors, and it's good enough for me!

There’s subtlety here as well. Applications can leak data in unexpected ways.

THINGS THAT CAN LEAK

- *Anything* which transmits or stores information, not just the database
- Logging systems: do they store in plain text? ship to a third-party log services?
- Error handlers: do they alert through a third-party service? is a secure connection used?
- Credentials: do you require authentication to occur over secure connections?

MISSING FUNCTION-LEVEL ACCESS CONTROL

I wonder what happens if I click this button?

Every function which can create, delete or modify data should be appropriately protected by authentication or authorization controls

CROSS-SITE REQUEST FORGERY

That's "CSRF" to you.

In a CSRF attack, a legitimate user of your application is tricked or deceived into submitting a request to your application. XSS vulnerabilities can provide one avenue to create CSRF vulnerabilities, but are not the only method.

COMPONENTS WITH KNOWN VULNERABILITIES

Version `0.0.1-pre-alpha` *is probably safe enough for production, right?*

Keeping track of components and libraries you use, and issues in them, is difficult. But it’s also necessary: an issue anywhere in your stack can expose everything.

UNVALIDATED REDIRECTS AND FORWARDS

http://example.com/login/?next=/profile/

I wonder what happens if I pass in `next=http://evilsite.com/` ...

Unfortunately, validating redirection targets is a hard problem.

SO WHAT CAN WE DO ABOUT IT?

"Give up and become a potato farmer" is looking more tempting every day.

INJECTION ATTACKS

The simplest and most reliable way to prevent SQL injection is to use parameterized queries.

```
name = request.GET['name']
my_query = """
SELECT *
FROM users
WHERE name = %s
"""
result = db_cursor.execute(
    my_query, (name,)
)
```

Django's ORM uses parameterized queries by default, so you don't need to worry about this most of the time.

You *do* need to worry about it any time you're supplying raw SQL or bits of SQL to Django's ORM, though.

The `extra()` and `raw()` methods, and the `RawSQL` query expression, all take a `params` argument. Use it.

OTHER INJECTION ATTACKS

- Mail header injection: reject any input value with a newline in it. Django's mail-sending functions do this for you automatically, raising `BadHeaderError`
- Command injection: use Python's `subprocess` module and *never* invoke anything with `shell=True`
- XML injection: use the `defusedxml` Python library for XML handling

AUTHENTICATION AND SESSION MANAGEMENT

Django's authentication framework does its best to protect you, but there's some extra work required to cover your bases.

BASIC STEPS FOR MORE SECURE AUTH AND SESSIONS

- Serve your site over a secure connection
- Turn on HSTS to be sure
- Mark important cookies secure and inaccessible to JavaScript
- Never expose a session ID
- Use password validation (new in Django 1.9) to avoid easily-guessed credentials

CROSS-SITE SCRIPTING

By default Django applies HTML escaping to the output of all template variables.

But that’s just a start: Django won’t generate all your HTML. Audit everything else, including JavaScript, for unsafe uses (especially of `innerHTML` — or better yet, don’t use it!)

Also, make sure *not* to use the `escapejs` template filter for security — all it does is perform backslash escaping to make strings be syntactically valid for JavaScript. It *does not* perform any type of sanitization.

INSECURE DIRECT OBJECT REFERENCES

Whenever possible, avoid exposing internal object IDs publicly; instead prefer natural keys. Django's URL routing makes this easy, since you decide which parameters to put in your URLs.

Bad:

http://example.com/users/23/

Good:

http://example.com/users/janedoe/

SECURITY MISCONFIGURATION

Use Django's system check framework and run the deployment check before moving to production:

```
python manage.py check --deploy
```

You can also run only the security-related checks:

```
python manage.py check --tag security
```

This *only* checks your Django applications and configuration. For other components of your stack you'll need to read documentation to familiarize yourself with best practices and secure configuration.

SENSITIVE DATA EXPOSURE

Much of the advice here is similar to auth and sessions: use secure connections, etc.

But that is, as always, just a start.

Django also provides decorators to let you specify security-sensitive request parameters and view-local variables. If you do, they'll be scrubbed from logging and error reporting handlers within Django.

They live in `django.views.decorators.debug`:

- `sensitive_post_parameters`
- `sensitive_variables`

```
from django.views.decorators.debug import \
  sensitive_variables

@sensitive_post_parameters('username', 'password')
def my_login_function(username, password):
    # If an error occurs in this function, the
    # username and password variables will be
    # scrubbed from any reported traceback.
```

Any tracebacks generated by Django will also scrub the values of any settings whose names match common sensitive patterns (such as ‘API’, ‘SECRET’, ‘PASS’, etc.).

You should avoid ever receiving sensitive values in `GET` parameters; Django can't help you with this, because they'll be logged automatically by your web server and possibly other parts of your stack.

MISSING FUNCTION-LEVEL ACCESS CONTROL

Django's authentication system provides the tools to let you control access down to the view level.

For function-based views, decorators live in `django.contrib.auth.decorators`:

- `login_required`
- `permission_required`
- `user_passes_test`

For class-based views, mixins live in `django.contrib.auth.mixins`:

- `LoginRequiredMixin`
- `PermissionRequiredMixin`
- `UserPassesTestMixin`

You can also control which HTTP methods are permitted on a view. Decorators (for function-based views) live in `django.views.decorators.http`:

- `require_GET`
- `require_POST`
- `require_http_methods`
- `require_safe`

On class-based views, you can set the attribute `http_method_names` to a list of accepted HTTP methods.

```
from django.views.generic import View

# This view only allows POST and PUT
class PostPutView(View):
    http_method_names = ['POST', 'PUT']
```

For more fine-grained control you can build logic into your view (for example, to have per-object control).

On generic class-based views you can often override the method which performs the database query and do the checks there.

If you raise `django.core.exceptions.PermissionDenied` anywhere in your code, Django will convert it to an HTTP 403 Forbidden response.

CROSS-SITE REQUEST FORGERY

CSRF protection is on by default in Django. Don’t disable it, but be aware of what it requires you to do.

Using Django templates, always put `{% csrf_token %}` just after the opening `<form>` tag for anything which will use an “unsafe” HTTP method like `POST`.

Using Jinja with Django’s built-in Jinja template backend, use `{{ csrf_input }}` there instead.

For AJAX form submissions the instructions are slightly more involved:

https://docs.djangoproject.com/en/1.9/ref/csrf/#ajax

COMPONENTS WITH KNOWN VULNERABILITIES

Django can’t directly help you with this, because Django’s code has no knowledge of these types of security issues. So you’ll have to do this manually.

USEFUL RESOURCES

- Subscribe to the `django-announce` mailing list to get announcements of new Django releases (including security releases and advisories).
- Regularly run your operating system's package/software updater.
- Use a service like `https://requires.io/` (free for open-source projects) to track and be notified of the status of your Python dependencies.

UNVALIDATED REDIRECTS AND FORWARDS

If at all possible, don’t rely on a user-controllable parameter to determine where to redirect.

If you *do* need to rely on such a parameter, validate it before you issue a redirect.

`django.utils.http.is_safe_url()` can help you with this, but it isn't perfect — validating URLs is notoriously difficult.

WE SOLVED SECURITY! YAY!

We did solve it, right?

…right?

THE OWASP TOP TEN IS JUST THE BEGINNING

YOU CAN LEAD A USER TO A SECURE CONNECTION...

And you can make them use it.

```
#Add
#django.middleware.security.SecurityMiddleware
#to your MIDDLEWARE_CLASSES setting.

#Then this:
SECURE_SSL_REDIRECT = True
```

But there's still a risk: the initial connection will be done over HTTP before the redirect to HTTPS happens. Can you close that off, too?

HTTP Strict Transport Security (HSTS) uses a header to tell browsers to always force a secure (HTTPS) connection to your site. It does require that *everything* on your site — including all included images, stylesheets, JavaScript, etc. — be served over HTTPS. But you want to do that anyway, right?

```
# More SecurityMiddleware fun!
SECURE_HSTS_SECONDS = 31536000
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
```

MY CONTENT SMELLS BAD

Why do you keep sniffing it?

Content sniffing is an unfortunate "feature" where web browsers read the first part of your response to try to guess its content type rather than follow the `Content-Type` header you sent.

This can result in files being interpreted as the wrong type — including perhaps as executable code.

```
# Still got SecurityMiddleware enabled? Good:
SECURE_CONTENT_TYPE_NOSNIFF = True
```

WHO STOLE THE COOKIE FROM THE COOKIE JAR?

It definitely wasn't Cookie Monster.

Cookies are useful, but:

- JavaScript can access them
- They get sent on both secure and insecure requests

This can lead to cookie values being exposed to people with less-than-good intentions.

```
# Make CSRF token and session cookie only get
# sent over secure connections:

CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True
```

```
# Make CSRF token and session cookie use the
# HttpOnly flag, denying JavaScript access:

CSRF_COOKIE_HTTPONLY = True
SESSION_COOKIE_HTTPONLY = True

# Just remember HttpOnly is a “better than
# nothing” approach, not a full solution to
# protecting cookie values.
```

I WAS FRAMED!

It probably still wasn't Cookie Monster.

Clickjacking attacks deceive a user into making a legitimate request, by overlaying a hidden frame — with your site's form controls — on something the user is tempted into clicking.

```
# django.middleware.clickjacking.XFrameOptionsMiddleware
# in your MIDDLEWARE_CLASSES

X_FRAME_OPTIONS = 'DENY'

# or to allow your own site to frame itself:

X_FRAME_OPTIONS = 'SAMEORIGIN'

# django.views.decorators.clickjacking also contains
# decorators to let you do this on a per-view basis:
#
# xframe_options_exempt, xframe_options_deny, and
# xframe_options_sameorigin
```

WHERE DID THIS JAVASCRIPT COME FROM?

Cookie Monster wanted for questioning.

Autoescaping HTML is a good *start* for preventing cross-site scripting. But there are still ways to sneak JavaScript into unexpected places.

```
# SecurityMiddleware again!

SECURE_BROWSER_XSS_FILTER = True

# Though like HttpOnly on cookies, this is a
# “probably better than nothing” rather than a
# “slam dunk win”.
```

But what you really want is a way to allow *only* the scripts you personally put on your site. How can you do that?

Content Security Policy (CSP) is a browser-supported HTTP header specifying valid sources for JavaScript, stylesheets, images and more.

Browsers will refuse to load/execute any resource not permitted by a CSP header, including inline JavaScript if the policy disallows it.

CSP also allows you to specify a callback URL where supporting browsers will `POST` a summary of any violations of your policy they encounter while rendering your site.

`django-csp` *is a third-party package maintained by Mozilla, providing configurable CSP support for Django:*

http://django-csp.readthedocs.io/

STAY IN THE SANDBOX

And don’t kick over anyone’s sand castle.

JavaScript has a *same-origin sandbox*: by default, it can only issue requests to the domain the JavaScript was served from. This can be overridden using *Cross-Origin Resource Sharing (CORS)*, which uses an HTTP header to specify an access-control policy.

Flash and Silverlight *also* have a same-origin sandbox, and *also* let you override it, but they use XML policy files instead of HTTP headers.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE cross-domain-policy
  SYSTEM 'http://www.adobe.com/xml/dtds/
cross-domain-policy.dtd'>
<cross-domain-policy>
  <site-control permitted-cross-domain-
policies="none"/>
</cross-domain-policy>
```

A Flash cross-domain policy needs to be served from the URL `/crossdomain.xml` on the domain (but can specify additional policy files found elsewhere).

Silverlight uses a file called `clientaccesspolicy.xml` with a different format, but also understands and looks for Flash's `crossdomain.xml` file and format.

`django-flashpolicies` *is a package providing support for generating and serving cross-domain policy files in Django:*

http://django-flashpolicies.readthedocs.io/

A BRIEF HISTORY OF DJANGO AND SECURITY

Spoiler alert: we got some things wrong.

2007:

Pre-1.0 Django

Issues reported haphazardly, fixed in SVN trunk. Advice sometimes to download and overwrite files with new versions.

2008

Django 1.0 released

Template system now HTML-escapes variable output.

2010

Django 1.2 released

Django now includes a CSRF prevention tool in core.

2012

Django 1.4 released

Better password storage, vetted crypto, signed cookies, clickjacking protection, error scrubbing, and a formal security process.

2013

Django 1.5 and 1.6 released

Host header hardening, more password-storage improvements.

2014

Django 1.7 released

System check framework introduced to verify configuration automatically.

Early 2015

Django 1.8 released

System check framework now has a security-oriented “deployment check”, security middleware introduced.

Late 2015

Django 1.9 released

Password validation framework, new permission mixins for class-based views.

WE FALL DOWN.

August 16th, 2006: Django's first vulnerability

CVE-2007-0404

Filename validation issue in translation framework

March 1st, 2016: Django's latest vulnerability

CVE-2016-2513

Username enumeration through timing difference on password hasher work factor upgrade

Between those came *fifty-five* other security issues and advisories.

Here are all of them:

https://docs.djangoproject.com/en/dev/releases/security/

WE FALL DOWN A LOT.

AND WE GET BACK UP.

Django's full security policy is always available online at:

https://www.djangoproject.com/security/

The primary goals of this process are to protect Django's users by encouraging responsible reporting and disclosure of security issues.

Django's security process begins with an email address: `security@djangoproject.com`

If you think you've found a security issue, please email that address.

Once an issue has been reported, Django's security team will verify the issue with the reporter, then begin tracking it in a private issue repository.

Once a patch has been developed, a CVE identifier is requested for the issue, and our security-prenotification list receives the issue description and patch.

One week after pre-notification, we go public, issuing new releases of Django and publishing a full description of the problem and direct links to the commit(s) that fixed it.

And then we wait for the next one.

PATTERNS IN SECURITY ISSUES

It’s déjà vu all over again!

> Parsing the `Accept-Language` header is expensive to do on every request. Let's do it once per unique value and cache the results!
>
> *-The Django team, circa 2007*

> Let's use a one-time base36 token to do password resets!

-The Django team, circa 2010

“

Formsets need to be able to dynamically grow the number of forms they use!

-The Django team, circa 2013

“

Restrictions on password length are dumb! Everybody knows long passwords are better!

-The Django team, circa 2013

CVE-2007-5712

Denial-of-service via arbitrarily-large `Accept-Language` header

CVE-2010-4535

Denial-of-service in password-reset mechanism

CVE-2013-0306

Denial-of-service via formset `max_num` bypass

CVE-2013-1443

Denial-of-service via large passwords

Python doesn’t have some of the vulnerabilities common in other languages, but you can still DoS yourself if you’re not careful.

STOP DOS'ING YOURSELF!

- Sanity-check all your inputs for length *before* you start processing them.
- Yes, even passwords (where appropriate)!
- Configure your web server to cap the length of HTTP headers and request bodies

“

`URLField` should really check whether the URL exists before accepting the value!

-The Django team, circa 2006

“

`URLField` should accept anything that matches the format of a valid URL!

-The Django team, circa 2006

“

`EmailField` should accept anything that matches the format of a valid email address!

-The Django team, circa 2006

> Checking for corrupt images is easy, we can just use PIL/Pillow's routines for that!
>
> *-The Django team, circa 2012*

> Most image formats store metadata in a header, let's find it by only reading a few bytes at a time!
>
> *-The Django team, circa 2012*

CVE-2011-4137

Denial-of-service via `URLField.verify_exists`

CVE-2009-3965

Denial-of-service via pathological regular-expression performance

CVE-2012-3443

Denial-of-service via compressed image files

CVE-2012-3444

Denial-of-service via large image files

“

What’s the worst that could happen?

-A really good question to ask!

NO REALLY, STOP DOS'ING YOURSELF!

- Figure out how much work your code should do
- Then figure out whether you can make it do more
- Then figure out ways to ensure it does less
- Some issues, like compressed formats, pathological regex, etc. have been around forever — read up on them!

“

Values of cookies we’ve set can be trusted!

-The Django team, circa 2010

“

Admin users can be trusted with a bit of the lookup API!

-The Django team, circa 2010

“

We can trust the browser same-origin sandbox!

-The Django team, circa 2011

“

We can trust admin users with the history log!

-The Django team, circa 2013

“

Once we’ve validated a value and stored it, we can trust it!

-The Django team, circa 2013

CVE-2010-3082

XSS via trusting unsafe cookie values

CVE-2010-4534

Information leakage in administrative interface

CVE-2011-0696

CSRF via forged HTTP headers

CVE-2013-0305

Information leakage via admin history log

No CVE identifier

XSS via admin trusting `URLField` values

“

We can trust the HTTP `Host` header now!

-The Django team, over and over again…

CVE-2011-4139

`Host` header cache poisoning

CVE-2011-4140

Potential CSRF via `Host` header

CVE-2012-4520

`Host` header poisoning

Advisory, 2012-12-10

Additional `Host` header hardening

Advisory, 2013-02-19

Additional hardening of `Host` header handling

TRUST NO ONE

THIS IS ONLY THE TIP OF THE ICEBERG

And unlike “Titanic”, the iceberg doesn’t have an award-winning soundtrack.

THERE'S NO SUCH THING AS "SECURE"

It's an important enough idea that this slide is **YELLING** *about it in* **ALL CAPS**. *Again.*

QUESTIONS?