django-flashpolicies Documentation

Release 1.9

James Bennett

Jun 05, 2017

Contents

This application provides management of Flash cross-domain policies (which are required for Flash content to access information across domains) for Django-powered sites. Cross-domain policies are represented by an XML file format, and this application generates and serves the appropriate XML.

In many cases, the same policy file will also be understood by Microsoft's Silverlight browser plugin, which supports the Adobe Flash format as a fallback in the absence of a file in its own native cross-domain policy format.

In the most common case, you'll just set up one URL pattern, pointing the URL `/crossdomain.xml` to the view *`flashpolicies.views.allow_domains()`* and passing a list of domains from which you want to allow access. For example, to allow access from Flash content served from `media.example.com`, you could place the following in the root URLconf of your Django site:

```
from django.conf.urls import url

from flashpolicies.views import allow_domains

urlpatterns = [
    # ...your other URL patterns here...
    url(r'^crossdomain.xml$',
        allow_domains,
        {'domains': ['media.example.com']}),
]
```

CHAPTER 1

Documentation contents

Installation guide

Before installing django-flashpolicies, you'll need to have a copy of Django already installed. For information on obtaining and installing Django, consult the Django download page, which offers convenient packaged downloads and installation instructions.

The 1.9 release of django-flashpolicies supports Django 1.8, 1.10, and 1.11 on the following Python versions (matching the versions supported by Django itself):

- Django 1.8 suports Python 2.7, 3.3, 3.4, and 3.5.
- Django 1.10 supports Python 2.7, 3.4, and 3.5.
- Django 1.11 supports Python 2.7, 3.4, 3.5, and 3.6

Important: Python 3.2

Although Django 1.8 supported Python 3.2 at the time of its release, the Python 3.2 series has reached end-of-life, and as a result support for Python 3.2 has been dropped from django-flashpolicies.

Normal installation

The preferred method of installing django-flashpolicies is via `pip`, the standard Python package-installation tool. If you don't have `pip`, instructions are available for how to obtain and install it. If you're using Python 2.7.9 or later (for Python 2) or Python 3.4 or later (for Python 3), `pip` came bundled with your installation of Python.

Once you have `pip`, type:

```
pip install django-flashpolicies
```

Installing from a source checkout

If you want to work on django-flashpolicies, you can obtain a source checkout.

The development repository for django-flashpolicies is at <https://github.com/ubernostrum/django-flashpolicies>. If you have git installed, you can obtain a copy of the repository by typing:

```
git clone https://github.com/ubernostrum/django-flashpolicies.git
```

From there, you can use normal git commands to check out the specific revision you want, and install it using `pip install -e .` (the `-e` flag specifies an "editable" install, allowing you to change code as you work on django-flashpolicies, and have your changes picked up automatically).

Views for serving cross-domain policies

Included in django-flashpolicies are several views for generating and serving Flash cross-domain policies. Most sites will need no more than the *`allow_domains()`* policy-serving view.

Some of the other views here support more advanced use cases, but note that not all valid policy file options have direct support in these views. The *`Policy`* class does support all valid options, however, so instantiating a *`Policy`*, setting the desired options, and passing it to the *`serve()`* view will allow use of any options policy files can support.

`flashpolicies.views.`**`serve`**(*request*, *policy*)
Given a *`Policy`* instance, serializes it to UTF-8 and serve it.

Internally, this is used by all other included views as the mechanism which actually serves the policy file.

Parameters **`policy`** – The *`Policy`* to serve.

`flashpolicies.views.`**`allow_domains`**(*request*, *domains*)
Serves a cross-domain access policy allowing a list of domains.

Note that if this is returned from the URL `/crossdomain.xml` on a domain, it will act as a master policy and will not permit other policies to exist on that domain. If you need to set meta-policy information and allow other policies, use the *`metapolicy()`* view for the master policy instead.

Parameters **`domains`** – A list of domains from which to allow access. Each value may be either a domain name (e.g., `"example.com"`) or a wildcard (e.g., `"*.example.com"`). Due to serious potential security issues, it is strongly recommended that you not use wildcard domain values.

`flashpolicies.views.`**`metapolicy`**(*request*, *permitted*, *domains=None*)
Serves a cross-domain policy which can allow other policies to exist on the same domain.

Note that this view, if used, must be the master policy for the domain, and so must be served from the URL `/crossdomain.xml` on the domain: setting meta-policy information in other policy files is forbidden by the cross-domain policy specification.

Parameters

- **`permitted`** – A string indicating the extent to which other policies are permitted. *A set of constants is available, defining acceptable values for this argument*.
- **`domains`** – A list of domains from which to allow access. Each value may be either a domain name (e.g., `"example.com"`) or a wildcard (e.g., `"*.example.com"`). Due to serious potential security issues, it is strongly recommended that you not use wildcard domain values.

flashpolicies.views.**no_access**(*request*)

Serves a cross-domain policy which permits no access of any kind, via a meta-policy declaration disallowing all policy files.

Note that this view, if used, must be the master policy for the domain, and so must be served from the URL `/crossdomain.xml` on the domain. Setting meta-policy information in other policy files is forbidden by the cross-domain policy specification.

Internally, this view calls the *metapolicy()* view, passing *SITE_CONTROL_NONE* as the meta-policy.

Utilities for generating cross-domain policy files

Internally, all policy files generated by django-flashpolicies are represented by instances of *flashpolicies.policies.Policy*, which understands how to handle the various permitted options in policy files and can generate the correct XML. This documentation covers *Policy* objects and their API, but is not and should not be taken to be documentation on the format and options for cross-domain policy files; Adobe's cross-domain policy specification is the canonical source for that information.

Interaction with `Policy` objects

For most cases, instantiating a *Policy* object with one or more domains will accomplish the desired effect. The property *xml_dom* will yield an `xml.dom.minidom.Document` object representing the policy's XML; for information on working with these objects, consult the documentation for the xml.dom.minidom module in the Python standard library.

Serializing `Policy` objects

There are two similar but different ways to serialize the underlying XML. One is to use `str()` on a *Policy* instance, like so:

```
>>> from flashpolicies import policies
>>> my_policy = policies.Policy('media.example.com', 'api.example.com')
>>> print(str(my_policy))
<?xml version="1.0" ?>
<!DOCTYPE cross-domain-policy
  SYSTEM 'http://www.adobe.com/xml/dtds/cross-domain-policy.dtd'>
<cross-domain-policy>
        <allow-access-from domain="media.example.com"/>
        <allow-access-from domain="api.example.com"/>
</cross-domain-policy>
```

The other is to call the *serialize()* method. The difference between these options is:

1. `str()` will, as is required by Python's semantics, produce a result of type `str`. Which, on Python 3, is a Unicode string; this means the output is not in any particular encoding, and will omit the `encoding` declaration of the XML prolog.
2. *serialize()* will, on the other hand, always return a sequence of UTF-8-encoded bytes. This is the type `str` on Python 2, and the type `bytes` on Python 3. In accordance with this, the output of *serialize()* *will* include an `encoding` declaration in its XML prolog.

In general, `str()` should be used to inspect a *Policy* for debugging or educational purposes, while *serialize()* should be used any time the output will actually be treated as a policy file (i.e., if writing your own policy-serving view, or if serializing the policy to a file). The built-in *serve()* view uses *serialize()*.

API reference

class `flashpolicies.policies.`**`Policy`**

Wrapper object for creating and manipulating a Flash cross-domain policy.

In the most common case – specifying one or more domains from which to allow access – pass the domains when initializing. For example:

```
my_policy = Policy('media.example.com', 'api.example.com')
```

`xml_dom`

A read-only property which returns an XML representation of this policy, as an `xml.dom.minidom.Document` object.

`serialize``()`

Serialize this policy to a UTF-8-encoded byte string (i.e., `str` on Python 2, `bytes` on Python 3), suitable for serving over HTTP or writing to a file.

`allow_domain`(*domain*, *to_ports=None*, *secure=True*)

Allows access for Flash content served from a particular domain.

Parameters

- **`domain`** – The domain from which to allow access. May be either a full domain name (e.g., `"example.com"`) or a wildcard (e.g., `"example.com"`). Due to serious potential security concerns, it is strongly recommended that you avoid wildcard domain values.
- **`to_ports`** – (only for socket policy files) A list of ports the domain will be permitted to access. Each value in the list may be either a port number (e.g., `"80"`), a range of ports (e.g., `"80-120"`) or the wildcard value `"*"`, which will permit all ports.
- **`secure`** – If `True`, will require the security level of the HTTP protocol for Flash content to match that of this policy file; for example, if the policy file was retrieved via HTTPS, Flash content from `domain` must also be retrieved via HTTPS. If `False`, this matching of security levels will be disabled. It is strongly recommended that you not disable the matching of security levels.

`allow_headers`(*domain*, *headers*, *secure=True*)

Allows Flash content from a particular domain to push data via HTTP headers.

Parameters

- **`domain`** – The domain from which to allow access. May be either a full domain name (e.g., `"example.com"`) or a wildcard (e.g., `"example.com"`). Due to serious potential security concerns, it is strongly recommended that you avoid wildcard domain values.
- **`headers`** – A list of HTTP header names in which data may be submitted.
- **`secure`** – If `True`, will require the security level of the HTTP protocol for Flash content to match that of this policy file; for example, if the policy file was retrieved via HTTPS, Flash content from `domain` must also be retrieved via HTTPS. If `False`, this matching of security levels will be disabled. It is strongly recommended that you not disable the matching of security levels.

`allow_identity`(*fingerprint*)

Allows access from digitally-signed documents.

Parameters **`fingerprint`** – The fingerprint of the signing key to allow.

The XML resulting from use of this method will include both the key fingerprint and the name of an algorithm used to calculate the fingerprint. At the moment, `"sha-1"` is the only value defined in the cross-domain policy specification for the `fingerprint-algorithm` attribute of the `certificate`

element (which is the element produced by this method), and so an argument for this is omitted; if additional algorithms are added to the specification, support will be added in a backwards-compatible fashion (likely through an argument defaulting to SHA-1).

metapolicy (*permitted*)
Sets metapolicy information (only applicable to master policy files), determining which other policy files may be used on the same domain.

Parameters **permitted** – The metapolicy to use. Acceptable values are those listed in the cross-domain policy specification, and are also available as *a set of constants defined in this module*. Passing an invalid value will raise `TypeError`.

By default, Flash assumes a default metapolicy of `"master-only"` (except for socket policies, which assume a default of `"all"`), so if this is the desired metapolicy (and, for security reasons, it often is), this method does not need to be called.

Note that a metapolicy of `"none"` forbids **all** access, even if one or more domains, headers or identities have previously been specified as allowed. As such, setting the metapolicy to `"none"` will remove all access previously granted by *`allow_domain()`*, *`allow_identity()`* or *`allow_headers()`*. Additionally, attempting to grant access via *`allow_domain()`*, *`allow_identity()`* or *`allow_headers()`* will, when the metapolicy is `"none"`, raise `TypeError`.

Available constants

For ease of working with metapolicies, the following constants are defined, and correspond to the acceptable values for metapolicies as defined in the cross-domain policy specification.

`flashpolicies.policies.`**SITE_CONTROL_ALL**
All policy files available on the current domain are permitted. Actual value is the string `"all"`.

`flashpolicies.policies.`**SITE_CONTROL_BY_CONTENT_TYPE**
Only policy files served from the current domain with an HTTP `Content-Type` of `text/x-cross-domain-policy` are permitted. Actual value is the string `"by-content-type"`.

`flashpolicies.policies.`**SITE_CONTROL_BY_FTP_FILENAME**
Only policy files served from the current domain as files named `crossdomain.xml` are permitted. Actual value is the string `"by-ftp-filename"`.

`flashpolicies.policies.`**SITE_CONTROL_MASTER_ONLY**
Only the master policy file for this domain – the policy served from the URL `/crossdomain.xml` – is permitted. Actual value is the string `"master-only"`.

`flashpolicies.policies.`**SITE_CONTROL_NONE**
No policy files are permitted, including the master policy file. Actual value is the string `"none"`.

`flashpolicies.policies.`**VALID_SITE_CONTROL**
A tuple containing the above constants, for convenient validation of metapolicy values.

Feature and API deprecation cycle

The following features or APIs of django-flashpolicies are deprecated and scheduled to be removed in future releases. Please make a note of this and update your use of django-flashpolicies accordingly. When possible, deprecated features will emit a `DeprecationWarning` as an additional warning of pending removal.

The `flashpolicies.views.simple` view

Will be removed in: django-flashpolicies 2.0

This view has been renamed to *`allow_domains()`* to better communicate its purpose. The view `flashpolicies.views.simple` continues to exist for now as an alias for backwards compatibility, but will be removed (and emits a `DeprecationWarning`).

The `flashpolicies` combined package

Will be removed in: django-flashpolicies 2.0

For the 2.0 release, django-flashpolicies will be split into two packages. One – which will be named "flashpolicies" and provide a Python module named `flashpolicies` – will contain the *`Policy`* class and associated code, which are not dependent on Django in any way. The Django views for serving policies will become a separate package, retaining the name "django-flashpolicies" on the Python Package Index and providing a module named `django_flashpolicies`.

Frequently asked questions

The following notes answer common questions, and may be useful to you when installing, configuring or using django-flashpolicies.

Why do I need a cross-domain policy file?

Much like JavaScript, the Adobe Flash player by default has a same-origin policy; a Flash player instance on one domain cannot load data from another domain.

A cross-domain policy file allows you, as the owner of a domain, to specify exceptions to this, allowing loading of data from another domain (for example, if you have data hosted on a CDN).

In order to prevent security issues caused by loading data from untrusted domains, your cross-domain policy file should permit *only* those domains you know are trustworthy (i.e., because those domains are under your control, and you can prevent malicious content from being placed on them).

Why doesn't this application generate Silverlight's format?

The Microsoft Silverlight plugin has a same-origin sandbox like Flash, and its native format for cross-domain policies is a file called `clientaccesspolicy.xml`. However, if `clientaccesspolicy.xml` is not found on the target domain, or otherwise returns an error, Silverlight will fall back to requesting and obeying a Flash `crossdomain.xml` file.

This means that a single file – `crossdomain.xml` in the Flash format – suffices for both Flash and Silverlight. Additionally, Silverlight is no longer supported in current versions of Microsoft's own Edge browser, support for it is in the process of being dropped/disabled in other major browsers, and Microsoft has announced that Silverlight will reach end-of-life in 2021, meaning that the Silverlight-only format corresponds to an already-small and shrinking, and soon to be nonexistent, supported base.

What versions of Django are supported?

As of django-flashpolicies 1.9, Django 1.8 and 1.9 are supported.

Older versions of Django may work, but are not supported. In particular, the behavior of the `APPEND_SLASH` setting in some old Django versions may be problematic: on very old versions of Django, `APPEND_SLASH` always adds a trailing slash even if the URL would match without it. This makes it impossible to serve a master policy file, which must have *exactly* the URL `/crossdomain.xml`, with no trailing slash.

What versions of Python are supported?

As of django-flashpolicies 1.9, Django 1.8, 1.9, and 1.10 are supported, on Python 2.7, 3.3, 3.4 or 3.5. Although Django 1.8 supported Python 3.2 at initial release, Python 3.2 is now at its end-of-life and django-flashpolicies no longer supports it.

Why are the elements in a different order each time I serialize my policy?

Internally, a *`Policy`* stores information about permitted domains and headers in dictionaries, keyed by domain names. The resulting XML is generated by iterating over these dictionaries.

In older versions of Python, iteration over a dictionary would produce the same order of keys each time provided the set of keys was identical. Newer versions of Python include a feature, for security purposes, known as hash randomization; this means that two dictionaries with the same set of keys can and will at times iterate over those keys in different orders.

Hash randomization is enabled by default on Python 3.3, and can be enabled on older releases. If you are seeing inconsistent ordering for `allow-access-from` and `allow-http-request-headers-from` elements, it is due to hash randomization being enabled.

Since this does not affect the well-formedness or validity of the resulting XML document, it is not a bug, and you should not attempt to disable hash randomization in Python.

Why shouldn't I use wild-card (i.e., '*') domains in my policy?

Use of wild-card entries in a policy effectively negates much of the security gain that comes from explicitly specifying the permitted domains. Unless you can and do vigilantly control all possible domains/subdomains matching a wild-card entry, use of one will expose you to the possibility of loading malicious content.

How am I allowed to use this module?

I found a bug or want to make an improvement!

The canonical development repository for django-flashpolicies is online at <https://github.com/ubernostrum/django-flashpolicies>. Issues and pull requests can both be filed there.

See also:

- Overview of cross-domain policy files

- Policy file format specification
- Adobe's recommendations for use of Flash cross-domain policies
- Microsoft's documentation on support in Silverlight for cross-domain requests

Python Module Index

f

Index