django-fernet-fields Documentation

Release 0.5.dev1

Carl Meyer

Sep 19, 2017

Contents

Fernet symmetric encryption for Django model fields, using the cryptography library.

CHAPTER 1

Prerequisites

`django-fernet-fields` supports Django 1.8.2 and later on Python 2.7, 3.3, 3.4, pypy, and pypy3.

Only PostgreSQL, SQLite, and MySQL are tested, but any Django database backend with support for `BinaryField` should work.

CHAPTER 2

Installation

`django-fernet-fields` is available on PyPI. Install it with:

```
pip install django-fernet-fields
```

CHAPTER 3

Usage

Just import and use the included field classes in your models:

```
from django.db import models
from fernet_fields import EncryptedTextField


class MyModel(models.Model):
    name = EncryptedTextField()
```

You can assign values to and read values from the `name` field as usual, but the values will automatically be encrypted before being sent to the database and decrypted when read from the database.

Encryption and decryption are performed in your app; the secret key is never sent to the database server. The database sees only the encrypted value of this field.

Field types

Several other field classes are included: `EncryptedCharField`, `EncryptedEmailField`, `EncryptedIntegerField`, `EncryptedDateField`, and `EncryptedDateTimeField`. All field classes accept the same arguments as their non-encrypted versions.

To create an encrypted version of some other custom field class, inherit from both `EncryptedField` and the other field class:

```
from fernet_fields import EncryptedField
from somewhere import MyField

class MyEncryptedField(EncryptedField, MyField):
    pass
```

Nullable fields

Nullable encrypted fields are allowed; a `None` value in Python is translated to a real `NULL` in the database column. Note that this trivially reveals the presence or absence of data in the column to an attacker. If this is a problem for your case, avoid using a nullable encrypted field; instead store some other sentinel "empty" value (which will be encrypted just like any other value) in a non-nullable encrypted field.

CHAPTER 4

Keys

By default, `django-fernet-fields` uses your `SECRET_KEY` setting as the encryption key.

You can instead provide a list of keys in the `FERNET_KEYS` setting; the first key will be used to encrypt all new data, and decryption of existing values will be attempted with all given keys in order. This is useful for key rotation: place a new key at the head of the list for use with all new or changed data, but existing values encrypted with old keys will still be accessible:

```
FERNET_KEYS = [
    'new key for encrypting',
    'older key for decrypting old data',
]
```

Warning: Once you start saving data using a given encryption key (whether your `SECRET_KEY` or another key), don't lose track of that key or you will lose access to all data encrypted using it! And keep the key secret; anyone who gets ahold of it will have access to all your encrypted data.

Disabling HKDF

Fernet encryption requires a 32-bit url-safe base-64 encoded secret key. By default, `django-fernet-fields` uses HKDF to derive such a key from whatever arbitrary secret key you provide.

If you wish to disable HKDF and provide your own Fernet-compatible 32-bit key(s) (e.g. generated with Fernet.generate_key()) directly instead, just set `FERNET_USE_HKDF = False` in your settings file. If this is set, all keys specified in the `FERNET_KEYS` setting must be 32-bit and url-safe base64-encoded bytestrings. If a key is not in the correct format, you'll likely get "incorrect padding" errors.

Warning: If you don't define a `FERNET_KEYS` setting, your `SECRET_KEY` setting is the fallback key. If you disable HKDF, this means that your `SECRET_KEY` itself needs to be a Fernet-compatible key.

CHAPTER 5

Indexes, constraints, and lookups

Because Fernet encryption is not deterministic (the same source text encrypted using the same key will result in a different encrypted token each time), indexing or enforcing uniqueness or performing lookups against encrypted data is useless. Every encrypted value will always be different, and every exact-match lookup will fail; other lookups' results would be meaningless.

For this reason, `EncryptedField` will raise `django.core.exceptions.ImproperlyConfigured` if passed any of `db_index=True`, `unique=True`, or `primary_key=True`, and any type of lookup on an `EncryptedField` except for `isnull` will raise `django.core.exceptions.FieldError`.

CHAPTER 6

Ordering

Ordering a queryset by an `EncryptedField` will not raise an error, but it will order according to the encrypted data, not the decrypted value, which is not very useful and probably not desired.

Raising an error would be better, but there's no mechanism in Django for a field class to declare that it doesn't support ordering. It could be done easily enough with a custom queryset and model manager that overrides `order_by()` to check the supplied field names. You might consider doing this for your models, if you're concerned that you might accidentally order by an `EncryptedField` and get junk ordering without noticing.

CHAPTER 7

Migrations

If migrating an existing non-encrypted field to its encrypted counterpart, you won't be able to use a simple `AlterField` operation. Since your database has no access to the encryption key, it can't update the column values correctly. Instead, you'll need to do a three-step migration dance:

1. Add the new encrypted field with a different name.
2. Write a data migration (using RunPython and the ORM, not raw SQL) to copy the values from the old field to the new (which automatically encrypts them in the process).
3. Remove the old field and (if needed) rename the new encrypted field to the old field's name.

CHAPTER 8

Note about deploying to Heroku

An important caveat when deploying an app dependent of *Fernet* to Heroku: you need to specify all requirements (even dependencies of dependencies) explicitly. In general, this is a good practice for version pinning purposes. But it's necessary for Fernet on Heroku because it depends on cryptography library, which in turn depends on libffi, a C library. When cryptography is explicitly defined on requirements.txt, Heroku knows it depends on libffi and installs it.

Therefore, an easy solution is to freeze your requirements after installing *Fernet*:

```
pip freeze > requirements.txt
```

CHAPTER 9

Contributing

See the contributing docs.