Password management in Django

Password management is something that should generally not be reinvented unnecessarily, and Django endeavors to provide a secure and flexible set of tools for managing user passwords. This document describes how Django stores passwords, how the storage hashing can be configured, and some utilities to work with hashed passwords.

See also: Even though users may use strong passwords, attackers might be able to eavesdrop on their connections. Use HTTPS to avoid sending passwords (or any other sensitive data) over plain HTTP connections because they will be vulnerable to password sniffing.

How Django stores passwords

Django provides a flexible password storage system and uses PBKDF2 by default.

The `password` attribute of a `User` object is a string in this format:

```
<algorithm>$<iterations>$<salt>$<hash>
```

Those are the components used for storing a User's password, separated by the dollar-sign character and consist of: the hashing algorithm, the number of algorithm iterations (work factor), the random salt, and the resulting password hash. The algorithm is one of a number of one-way hashing or password storage algorithms Django can use; see below. Iterations describe the number of times the algorithm is run over the hash. Salt is the random seed used and the hash is the result of the one-way function.

By default, Django uses the PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST. This should be sufficient for most users: it's quite secure, requiring massive amounts of computing time to break.

However, depending on your requirements, you may choose a different algorithm, or even use a custom algorithm to match your specific security situation. Again, most users shouldn't need to do this – if you're not sure, you probably don't. If you do, please read on:

Django chooses the algorithm to use by consulting the `PASSWORD_HASHERS` setting. This is a list of hashing algorithm classes that this Django installation supports.

For storing passwords, Django will use the first hasher in `PASSWORD_HASHERS`. To store new passwords with a different algorithm, put your preferred algorithm first in `PASSWORD_HASHERS`.

For verifying passwords, Django will find the hasher in the list that matches the algorithm name in the stored password. If a stored password names an algorithm not found in `PASSWORD_HASHERS`, trying to verify it will raise `ValueError`.

The default for `PASSWORD_HASHERS` is:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
]
```

This means that Django will use PBKDF2 to store all passwords but will support checking passwords stored with PBKDF2SHA1, argon2, and bcrypt.

The next few sections describe a couple of common ways advanced users may want to modify this setting.

Using Argon2 with Django

Argon2 is the winner of the 2015 Password Hashing Competition, a community organized open competition to select a next generation hashing algorithm. It's designed not to be easier to compute on custom hardware than it is to compute on an ordinary CPU. The default variant for the Argon2 password hasher is Argon2id.

Argon2 is not the default for Django because it requires a third-party library. The Password Hashing Competition panel, however, recommends immediate use of Argon2 rather than the other algorithms supported by Django.

To use Argon2id as your default storage algorithm, do the following:

1. Install the argon2-cffi package. This can be done by running `python -m pip install django[argon2]`, which is equivalent to `python -m pip install argon2-cffi` (along with any version requirement from Django's `pyproject.toml`).
2. Modify `PASSWORD_HASHERS` to list `Argon2PasswordHasher` first. That is, in your settings file, you'd put:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
        "django.contrib.auth.hashers.ScryptPasswordHasher",
    ]
```

Keep and/or add any entries in this list if you need Django to upgrade passwords.

Using `bcrypt` with Django

Bcrypt is a popular password storage algorithm that's specifically designed for long-term password storage. It's not the default used by Django since it requires the use of third-party libraries, but since many people may want to use it Django supports bcrypt with minimal effort.

To use Bcrypt as your default storage algorithm, do the following:

1. Install the bcrypt package. This can be done by running `python -m pip install django[bcrypt]`, which is equivalent to `python -m pip install bcrypt` (along with any version requirement from Django's `pyproject.toml`).
2. Modify `PASSWORD_HASHERS` to list `BCryptSHA256PasswordHasher` first. That is, in your settings file, you'd put:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
]
```

Keep and/or add any entries in this list if you need Django to upgrade passwords.

That's it – now your Django install will use Bcrypt as the default storage algorithm.

Using `scrypt` with Django

scrypt is similar to PBKDF2 and bcrypt in utilizing a set number of iterations to slow down brute-force attacks. However, because PBKDF2 and bcrypt do not require a lot of memory, attackers with sufficient resources can launch large-scale parallel attacks in order to speed up the attacking process. scrypt is specifically designed to use more memory compared to other password-based key derivation functions in order to limit the amount of parallelism an attacker can use, see **RFC 7914** for more details.

To use scrypt as your default storage algorithm, do the following:

1. Modify `PASSWORD_HASHERS` to list `ScryptPasswordHasher` first. That is, in your settings file:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.ScryptPasswordHasher",
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
]
```

Keep and/or add any entries in this list if you need Django to upgrade passwords.

Note: `scrypt` requires OpenSSL 1.1+.

Increasing the salt entropy

Most password hashes include a salt along with their password hash in order to protect against rainbow table attacks. The salt itself is a random value which increases the size and thus the cost of the rainbow table and is currently set at 128 bits with the `salt_entropy` value in the `BasePasswordHasher`. As computing and storage costs decrease this value should be raised. When implementing your own password hasher you are free to override this value in order to use a desired entropy level for your password hashes. `salt_entropy` is measured in bits.

Implementation detail: Due to the method in which salt values are stored the `salt_entropy` value is effectively a minimum value. For instance a value of 128 would provide a salt which would actually contain 131 bits of entropy.

Increasing the work factor

PBKDF2 and bcrypt

The PBKDF2 and bcrypt algorithms use a number of iterations or rounds of hashing. This deliberately slows down attackers, making attacks against hashed passwords harder. However, as computing power increases, the number of iterations needs to be increased. We've chosen a reasonable default (and will increase it with each release of Django), but you may wish to tune it up or down, depending on your security needs and available processing power. To do so, you'll subclass the appropriate algorithm and override the `iterations` parameter (use the `rounds` parameter when subclassing a bcrypt hasher). For example, to increase the number of iterations used by the default PBKDF2 algorithm:

1. Create a subclass of `django.contrib.auth.hashers.PBKDF2PasswordHasher`

   ```
   from django.contrib.auth.hashers import PBKDF2PasswordHasher


   class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
       """
       A subclass of PBKDF2PasswordHasher that uses 100 times more iterations.
       """

       iterations = PBKDF2PasswordHasher.iterations * 100
   ```

   Save this somewhere in your project. For example, you might put this in a file like `myproject/hashers.py`.
2. Add your new hasher as the first entry in PASSWORD_HASHERS:

   ```
   PASSWORD_HASHERS = [
       "myproject.hashers.MyPBKDF2PasswordHasher",
       "django.contrib.auth.hashers.PBKDF2PasswordHasher",
       "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
       "django.contrib.auth.hashers.Argon2PasswordHasher",
       "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
       "django.contrib.auth.hashers.ScryptPasswordHasher",
   ]
   ```

That's it – now your Django install will use more iterations when it stores passwords using PBKDF2.

Note: bcrypt `rounds` is a logarithmic work factor, e.g. 12 rounds means `2 ** 12` iterations.

Argon2

Argon2 has the following attributes that can be customized:

1. `time_cost` controls the number of iterations within the hash.
2. `memory_cost` controls the size of memory that must be used during the computation of the hash.
3. `parallelism` controls how many CPUs the computation of the hash can be parallelized on.

The default values of these attributes are probably fine for you. If you determine that the password hash is too fast or too slow, you can tweak it as follows:

1. Choose `parallelism` to be the number of threads you can spare computing the hash.
2. Choose `memory_cost` to be the KiB of memory you can spare.
3. Adjust `time_cost` and measure the time hashing a password takes. Pick a `time_cost` that takes an acceptable time for you. If `time_cost` set to 1 is unacceptably slow, lower `memory_cost`.

`memory_cost` **interpretation:** The argon2 command-line utility and some other libraries interpret the `memory_cost` parameter differently from the value that Django uses. The conversion is given by `memory_cost == 2 ** memory_cost_commandline`.

`scrypt`

scrypt has the following attributes that can be customized:

1. `work_factor` controls the number of iterations within the hash.
2. `block_size`
3. `parallelism` controls how many threads will run in parallel.
4. `maxmem` limits the maximum size of memory that can be used during the computation of the hash. Defaults to `0`, which means the default limitation from the OpenSSL library.

We've chosen reasonable defaults, but you may wish to tune it up or down, depending on your security needs and available processing power.

Estimating memory usage: The minimum memory requirement of scrypt is:

```
work_factor * 2 * block_size * 64
```

so you may need to tweak `maxmem` when changing the `work_factor` or `block_size` values.

Password upgrading

When users log in, if their passwords are stored with anything other than the preferred algorithm, Django will automatically upgrade the algorithm to the preferred one. This means that old installs of Django will get automatically more secure as users log in, and it also means that you can switch to new (and better) storage algorithms as they get invented.

However, Django can only upgrade passwords that use algorithms mentioned in `PASSWORD_HASHERS`, so as you upgrade to new systems you should make sure never to *remove* entries from this list. If you do, users using unmentioned algorithms won't be able to upgrade. Hashed passwords will be updated when increasing (or decreasing) the number of PBKDF2 iterations, bcrypt rounds, or argon2 attributes.

Be aware that if all the passwords in your database aren't encoded in the default hasher's algorithm, you may be vulnerable to a user enumeration timing attack due to a difference between the duration of a login request for a user with a password encoded in a non-default algorithm and the duration of a login request for a nonexistent user (which runs the default hasher). You may be able to mitigate this by upgrading older password hashes.

Password upgrading without requiring a login

If you have an existing database with an older, weak hash such as MD5, you might want to upgrade those hashes yourself instead of waiting for the upgrade to happen when a user logs in (which may never happen if a user doesn't return to your site). In this case, you can use a "wrapped" password hasher.

For this example, we'll migrate a collection of MD5 hashes to use PBKDF2(MD5(password)) and add the corresponding password hasher for checking if a user entered the correct password on login. We assume we're using the built-in `User` model and that our project has an `accounts` app. You can modify the pattern to work with any algorithm or with a custom user model.

First, we'll add the custom hasher:

`accounts/hashers.py`

```
from django.contrib.auth.hashers import (
    PBKDF2PasswordHasher,
    MD5PasswordHasher,
)


class PBKDF2WrappedMD5PasswordHasher(PBKDF2PasswordHasher):
    algorithm = "pbkdf2_wrapped_md5"

    def encode_md5_hash(self, md5_hash, salt, iterations=None):
        return super().encode(md5_hash, salt, iterations)

    def encode(self, password, salt, iterations=None):
        _, _, md5_hash = MD5PasswordHasher().encode(password, salt).split("$", 2)
        return self.encode_md5_hash(md5_hash, salt, iterations)
```

The data migration might look something like:

`accounts/migrations/0002_migrate_md5_passwords.py`

```
from django.db import migrations

from ..hashers import PBKDF2WrappedMD5PasswordHasher


def forwards_func(apps, schema_editor):
    User = apps.get_model("auth", "User")
    users = User.objects.filter(password__startswith="md5$")
    hasher = PBKDF2WrappedMD5PasswordHasher()
    for user in users:
        algorithm, salt, md5_hash = user.password.split("$", 2)
        user.password = hasher.encode_md5_hash(md5_hash, salt)
        user.save(update_fields=["password"])


class Migration(migrations.Migration):
    dependencies = [
        ("accounts", "0001_initial"),
        # replace this with the latest migration in contrib.auth
        ("auth", "####_migration_name"),
    ]

    operations = [
        migrations.RunPython(forwards_func),
    ]
```

Be aware that this migration will take on the order of several minutes for several thousand users, depending on the speed of your hardware.

Finally, we'll add a `PASSWORD_HASHERS` setting:

`mysite/settings.py`

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "accounts.hashers.PBKDF2WrappedMD5PasswordHasher",
]
```

Include any other hashers that your site uses in this list.

Included hashers

The full list of hashers included in Django is:

```
[
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
    "django.contrib.auth.hashers.BCryptPasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
    "django.contrib.auth.hashers.MD5PasswordHasher",
]
```

The corresponding algorithm names are:

- `pbkdf2_sha256`
- `pbkdf2_sha1`
- `argon2`
- `bcrypt_sha256`
- `bcrypt`
- `scrypt`
- `md5`

Writing your own hasher

If you write your own password hasher that contains a work factor such as a number of iterations, you should implement a `harden_runtime(self, password, encoded)` method to bridge the runtime gap between the work factor supplied in the `encoded` password and the default work factor of the hasher. This prevents a user enumeration timing attack due to difference between a login request for a user with a password encoded in an older number of iterations and a nonexistent user (which runs the default hasher's default number of iterations).

Taking PBKDF2 as example, if `encoded` contains 20,000 iterations and the hasher's default `iterations` is 30,000, the method should run `password` through another 10,000 iterations of PBKDF2.

If your hasher doesn't have a work factor, implement the method as a no-op (`pass`).

Manually managing a user's password

The `django.contrib.auth.hashers` module provides a set of functions to create and validate hashed passwords. You can use them independently from the `User` model.

`check_password(password, encoded, setter=None, preferred='default')` [source]

`acheck_password(password, encoded, asetter=None, preferred='default')`

Asynchronous version: `acheck_password()`

If you'd like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the convenience function check_password(). It takes two mandatory arguments: the plain-text password to check, and the full value of a user's `password` field in the database to check against. It returns `True` if they match, `False` otherwise. Optionally, you can pass a callable `setter` that takes the password and will be called when you need to regenerate it. You can also pass `preferred` to change a hashing algorithm if you don't want to use the default (first entry of `PASSWORD_HASHERS` setting). See Included hashers for the algorithm name of each hasher.

Changed in Django 5.0:
`acheck_password()` method was added.

```
make_password(password, salt=None, hasher='default')
```
[source]

Creates a hashed password in the format used by this application. It takes one mandatory argument: the password in plain-text (string or bytes). Optionally, you can provide a salt and a hashing algorithm to use, if you don't want to use the defaults (first entry of `PASSWORD_HASHERS` setting). See Included hashers for the algorithm name of each hasher. If the password argument is `None`, an unusable password is returned (one that will never be accepted by check_password()).

```
is_password_usable(encoded_password)
```
[source]

Returns `False` if the password is a result of User.set_unusable_password().

Password validation

Users often choose poor passwords. To help mitigate this problem, Django offers pluggable password validation. You can configure multiple password validators at the same time. A few validators are included in Django, but you can write your own as well.

Each password validator must provide a help text to explain the requirements to the user, validate a given password and return an error message if it does not meet the requirements, and optionally define a callback to be notified when the password for a user has been changed. Validators can also have optional settings to fine tune their behavior.

Validation is controlled by the AUTH_PASSWORD_VALIDATORS setting. The default for the setting is an empty list, which means no validators are applied. In new projects created with the default startproject template, a set of validators is enabled by default.

By default, validators are used in the forms to reset or change passwords and in the createsuperuser and changepassword management commands. Validators aren't applied at the model level, for example in `User.objects.create_user()` and `create_superuser()`, because we assume that developers, not users, interact with Django at that level and also because model validation doesn't automatically run as part of creating models.

Note: Password validation can prevent the use of many types of weak passwords. However, the fact that a password passes all the validators doesn't guarantee that it is a strong password. There are many factors that can weaken a password that are not detectable by even the most advanced password validators.

Enabling password validation

Password validation is configured in the AUTH_PASSWORD_VALIDATORS setting:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME": "django.contrib.auth.password_validation.UserAttributeSimilarityValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.MinimumLengthValidator",
        "OPTIONS": {
            "min_length": 9,
        },
    },
    {
        "NAME": "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.NumericPasswordValidator",
    },
]
```

This example enables all four included validators:

- `UserAttributeSimilarityValidator`, which checks the similarity between the password and a set of attributes of the user.
- `MinimumLengthValidator`, which checks whether the password meets a minimum length. This validator is configured with a custom option: it now requires the minimum length to be nine characters, instead of the default eight.
- `CommonPasswordValidator`, which checks whether the password occurs in a list of common passwords. By default, it compares to an included list of 20,000 common passwords.

- `NumericPasswordValidator` , which checks whether the password isn't entirely numeric.

For `UserAttributeSimilarityValidator` and `CommonPasswordValidator` , we're using the default settings in this example. `NumericPasswordValidator` has no settings.

The help texts and any errors from password validators are always returned in the order they are listed in `AUTH_PASSWORD_VALIDATORS`.

Included validators

Django includes four validators:

`class MinimumLengthValidator(min_length=8)` [source]

Validates that the password is of a minimum length. The minimum length can be customized with the `min_length` parameter.

`class UserAttributeSimilarityValidator(user_attributes=DEFAULT_USER_ATTRIBUTES, max_similarity=0.7)` [source]

Validates that the password is sufficiently different from certain attributes of the user.

The `user_attributes` parameter should be an iterable of names of user attributes to compare to. If this argument is not provided, the default is used: `'username', 'first_name', 'last_name', 'email'` . Attributes that don't exist are ignored.

The maximum allowed similarity of passwords can be set on a scale of 0.1 to 1.0 with the `max_similarity` parameter. This is compared to the result of `difflib.SequenceMatcher.quick_ratio()`. A value of 0.1 rejects passwords unless they are substantially different from the `user_attributes` , whereas a value of 1.0 rejects only passwords that are identical to an attribute's value.

`class CommonPasswordValidator(password_list_path=DEFAULT_PASSWORD_LIST_PATH)` [source]

Validates that the password is not a common password. This converts the password to lowercase (to do a case-insensitive comparison) and checks it against a list of 20,000 common password created by Royce Williams.

The `password_list_path` can be set to the path of a custom file of common passwords. This file should contain one lowercase password per line and may be plain text or gzipped.

`class NumericPasswordValidator` [source]

Validate that the password is not entirely numeric.

Integrating validation

There are a few functions in `django.contrib.auth.password_validation` that you can call from your own forms or other code to integrate password validation. This can be useful if you use custom forms for password setting, or if you have API calls that allow passwords to be set, for example.

`validate_password(password, user=None, password_validators=None)` [source]

Validates a password. If all validators find the password valid, returns `None` . If one or more validators reject the password, raises a `ValidationError` with all the error messages from the validators.

The `user` object is optional: if it's not provided, some validators may not be able to perform any validation and will accept any password.

`password_changed(password, user=None, password_validators=None)` [source]

Informs all validators that the password has been changed. This can be used by validators such as one that prevents password reuse. This should be called once the password has been successfully changed.

For subclasses of `AbstractBaseUser`, the password field will be marked as "dirty" when calling `set_password()` which triggers a call to `password_changed()` after the user is saved.

`password_validators_help_texts(password_validators=None)` [source]

Returns a list of the help texts of all validators. These explain the password requirements to the user.

`password_validators_help_text_html(password_validators=None)`

Returns an HTML string with all help texts in an `<ul>` . This is helpful when adding password validation to forms, as you can pass the output directly to the `help_text` parameter of a form field.

`get_password_validators(validator_config)` [source]

Returns a set of validator objects based on the `validator_config` parameter. By default, all functions use the validators defined in `AUTH_PASSWORD_VALIDATORS`, but by calling this function with an alternate set of validators and then passing the result into the `password_validators` parameter of the other functions, your custom set of validators will be used instead. This is useful when you have a typical set of validators to use for most scenarios, but also have a special situation that requires a custom set. If you always use the same set of validators, there is no need to use this function, as the configuration from `AUTH_PASSWORD_VALIDATORS` is used by default.

The structure of `validator_config` is identical to the structure of `AUTH_PASSWORD_VALIDATORS`. The return value of this function can be passed into the `password_validators` parameter of the functions listed above.

Note that where the password is passed to one of these functions, this should always be the clear text password - not a hashed password.

Writing your own validator

If Django's built-in validators are not sufficient, you can write your own password validators. Validators have a fairly small interface. They must implement two methods:

- `validate(self, password, user=None)` : validate a password. Return `None` if the password is valid, or raise a `ValidationError` with an error message if the password is not valid. You must be able to deal with `user` being `None` - if that means your validator can't run, return `None` for no error.
- `get_help_text()` : provide a help text to explain the requirements to the user.

Any items in the `OPTIONS` in `AUTH_PASSWORD_VALIDATORS` for your validator will be passed to the constructor. All constructor arguments should have a default value.

Here's a basic example of a validator, with one optional setting:

```python
from django.core.exceptions import ValidationError
from django.utils.translation import gettext as _


class MinimumLengthValidator:
    def __init__(self, min_length=8):
        self.min_length = min_length

    def validate(self, password, user=None):
        if len(password) < self.min_length:
            raise ValidationError(
                _("This password must contain at least %(min_length)d characters."),
                code="password_too_short",
                params={"min_length": self.min_length},
            )

    def get_help_text(self):
        return _(
            "Your password must contain at least %(min_length)d characters."
            % {"min_length": self.min_length}
        )
```

You can also implement `password_changed(password, user=None)`, which will be called after a successful password change. That can be used to prevent password reuse, for example. However, if you decide to store a user's previous passwords, you should never do so in clear text.