django-defender

Release 0.9.7

Feb 27, 2023

Contents

1 Sites using django-defender **3**

2 Documentation **5**

3 Features **7**
3.1 Admin pages 8

4 Requirements **11**

5 Installation **13**
5.1 Migrations 14
5.2 Management commands 14

6 Long term goals **15**
6.1 Performance 15
6.2 Load testing 16
6.3 Results of load tests 16

7 Why not django-axes **17**

8 How django-defender works **19**

9 Cache backend **21**
9.1 Cache keys 21

10 Customizing django-defender **23**
10.1 Rationale for using DEFENDER_ATTEMPT_COOLOFF_TIME and DEFENDER_LOCKOUT_COOLOFF_TIME 24

11 Adapting to other authentication methods **27**

12 Adapting to other authentication methods :- django-rest-auth in djangorestframework **29**
12.1 Reference 29

13 Adapting for password reset forms **33**

14 Django signals **35**

15 Running tests **37**

16 Releasing **39**

17 Contributing **41**

18 Changes **43**

19 0.9.7 **45**

20 0.9.6 **47**

21 0.9.5 **49**
21.1 0.9.4 . . . 49
21.2 0.9.3 . . . 49
21.3 0.9.2 . . . 49
21.4 0.9.1 . . . 49
21.5 0.9.0 . . . 50
21.6 0.8.0 . . . 50
21.7 0.7.0 . . . 50
21.8 0.6.2 . . . 50
21.9 0.6.1 . . . 50
21.10 0.6.0 . . . 50
21.11 0.5.5 . . . 51
21.12 0.5.4 . . . 51
21.13 0.5.3 . . . 51
21.14 0.5.2 . . . 51
21.15 0.5.1 . . . 51
21.16 0.5.0 . . . 51
21.17 0.4.3 . . . 52
21.18 0.4.2 . . . 52
21.19 0.4.1 . . . 52
21.20 0.4.0 . . . 52
21.21 0.3.2 . . . 52
21.22 0.3.1 . . . 52
21.23 0.3 . . . 52
21.24 0.2.2 . . . 53
21.25 0.2.1 . . . 53
21.26 0.2 . . . 53
21.27 0.1 . . . 53

A simple Django reusable app that blocks people from brute forcing login attempts. The goal is to make this as fast as possible, so that we do not slow down the login attempts.

We will use a cache so that it doesn't have to hit the database in order to check the database on each login attempt. The first version will be based on Redis, but the goal is to make this configurable so that people can use whatever backend best fits their needs.

CHAPTER 1

Sites using django-defender

If you are using defender on your site, submit a PR to add to the list.

- https://hub.docker.com
- https://www.mycosbuilder.com

CHAPTER 2

Documentation

Documentation is available on Read the Docs:

https://django-defender.readthedocs.io

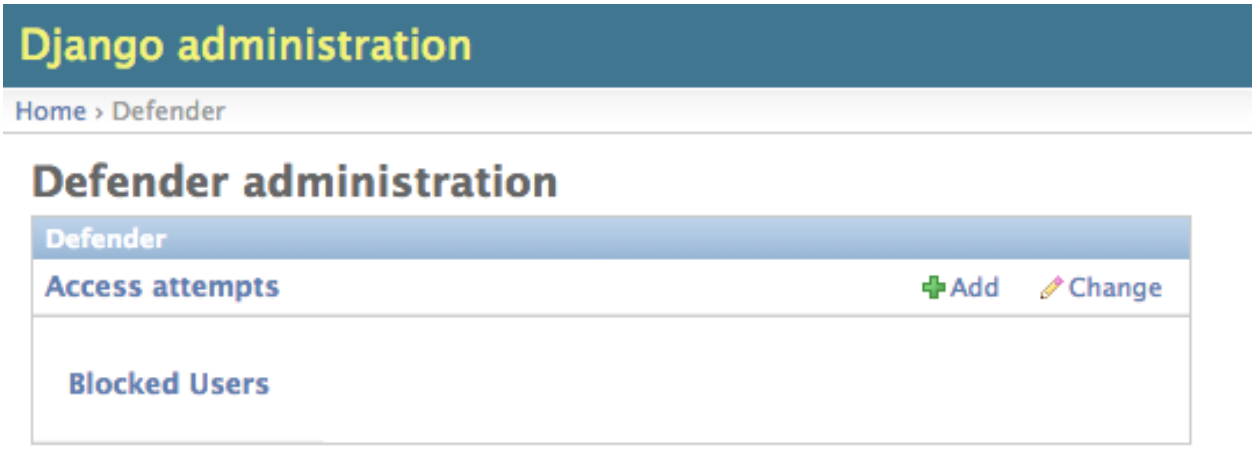
CHAPTER 3

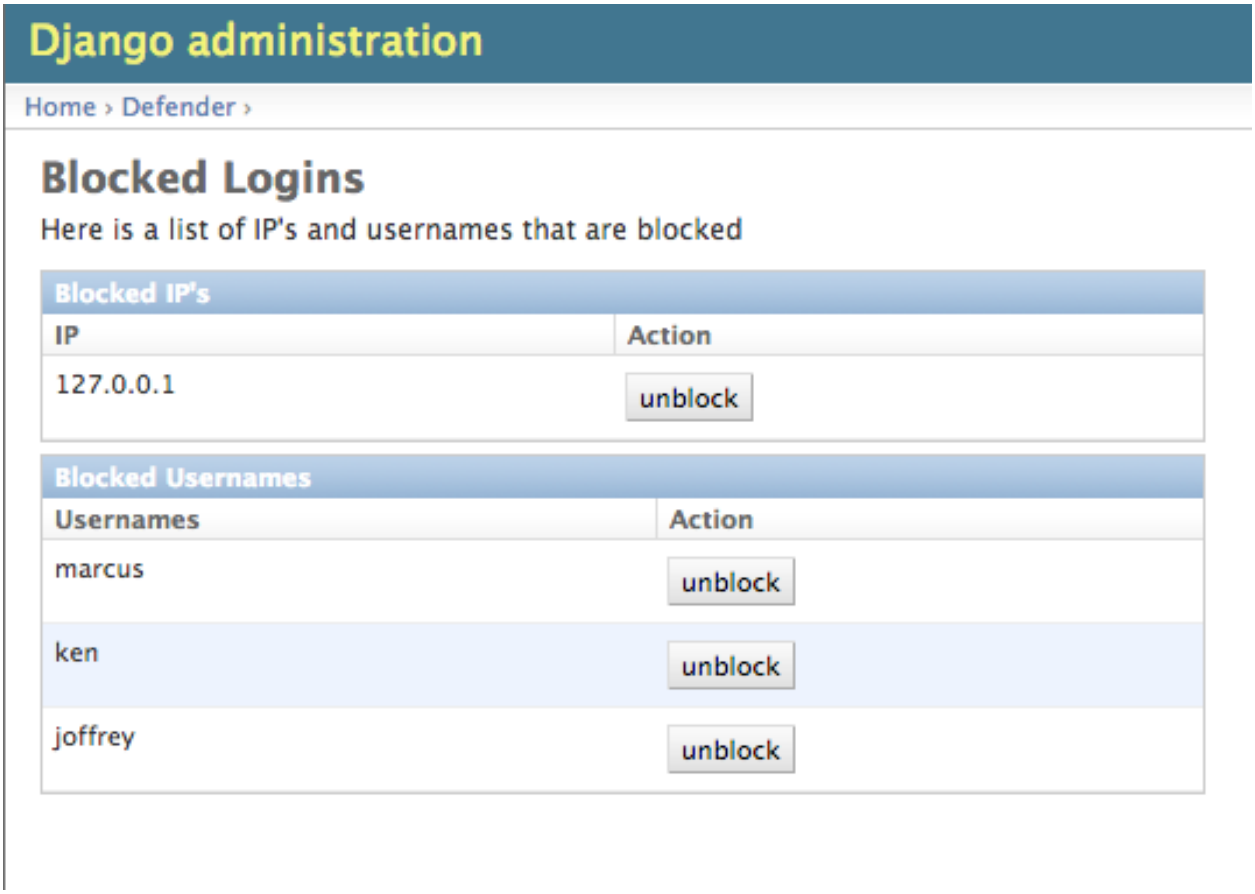
Features

- Log all login attempts to the database
- Support for reverse proxies with different headers for IP addresses
- Rate limit based on
  - Username
  - IP address
- Use Redis for the blacklist
- Configuration
  - Redis server
    - Host
    - Port
    - Database
    - Password
    - Key prefix
  - Block length
  - Number of incorrect attempts before block
- 95% code coverage
- Full documentation
- Ability to store login attempts to the database
- Management command to clean up login attempts database table
- Admin pages
  - List of blocked usernames and IP addresses
  - List of recent login attempts

  - Ability to unblock people
- Can be easily adapted to custom authentication method.
- Signals are sent when blocking username or IP

3.1 Admin pages




Django administration
Home › Defender ›
Blocked Logins
Here is a list of IP's and usernames that are blocked
Blocked IP's
IP
Action
127.0.0.1
unblock
Blocked Usernames
Usernames
Action
marcus
unblock
ken
unblock
joffrey
unblock

CHAPTER 4

Requirements

- Python: 3.7, 3.8, 3.9, 3.10, PyPy
- Django: 3.x, 4.x
- Redis: 5.x, 6.x, 7.x

CHAPTER 5

Installation

Download code, and run setup in one of the following ways depending on the method.

To install the production ready version from PyPI:

```
pip install django-defender
```

To install the development version from source code after download:

```
python setup.py install
```

To install the master branch development version from the GitHub repository:

```
pip install -e git+http://github.com/kencochran django-defender.git#egg=django_
↪defender-dev
```

First of all, you must add this project to your list of `INSTALLED_APPS` in `settings.py`

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    # ...
    'defender',
    # ...
]
```

Next, install the `FailedLoginMiddleware` middleware

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
```

(continues on next page)

(continued from previous page)

```
    'defender.middleware.FailedLoginMiddleware',
]
```

If you want to manage the blocked users via the Django admin, then add the following to your `urls.py`

```
urlpatterns = [
    path('admin/defender/', include('defender.urls')), # defender admin
    path('admin/', admin.site.urls), # normal admin
    # your own patterns follow...
]
```

5.1 Migrations

You will need to create tables in your database that are necessary for operation.

```
python manage.py migrate defender
```

5.2 Management commands

`cleanup_django_defender`

If you have a website with a lot of traffic, the AccessAttempts table will get full pretty quickly. If you don't need to keep the data for auditing purposes there is a management command to help you keep it clean.

It will look at your `DEFENDER_ACCESS_ATTEMPT_EXPIRATION` setting to determine which records will be deleted. Default if not specified, is 24 hours.

```
$ python manage.py cleanup_django_defender
```

You can set this up as a daily or weekly cron job to keep the table size down.

```
# run at 12:24 AM every morning.
24 0 * * * /usr/bin/python manage.py cleanup_django_defender >> /var/log/django_
↪defender_cleanup.log
```

CHAPTER 6

Long term goals

- Pluggable backends, so people can use something other than Redis
- Email users when their account is blocked
- Add a whitelist for username and ip's that we will never block (admin's, etc)
- Add a permanent black list for IP addresses
- Scan for known proxy IPs and do not block requests coming from those (improve the chances that a good IP is blocked)
- Add management command to prune old (configurable) login attempts.

6.1 Performance

The goal of defender is to make it as fast as possible so that it doesn't slow down the login process. In order to make sure our goals are met we need a way to test the application to make sure we are on the right track. The best way to do this is to compare how fast a normal Django login takes with defender and django-axes.

The normal django login, would be our baseline, and we expect it to be the fastest of the 3 methods, because there are no additional checks happening.

The defender login would most likely be slower then the django login, and hopefully faster then the django-axes login. The goal is to make it as little of a difference between the regular raw login, and defender.

The django-axes login speed, will probably be the slowest of the three since it does more checks and does a lot of database queries.

The best way to determine the speed of a login is to do a load test against an application with each setup, and compare the login times for each type.

6.2 Load testing

In order to make sure we cover all the different types of logins, in our load test we need to have more then one test.

1. All success: We will do a load test with nothing but successful logins.
2. Mixed: some success some failure: We will load test with some successful logins and some failures to see how the failure effect the performance.
3. All Failures: We will load test with all failure logins and see the difference in performance.

We will need a sample application that we can use for the load test, with the only difference is the configuration where we either load defender, axes, or none of them.

We can use a hosted load testing service, or something like jmeter. Either way we need to be consistent for all of the tests. If we use jmeter, we should have our jmeter configuration for others to run the tests on their own.

6.3 Results of load tests

We will post the results here. We will explain each test, and show the results along with some charts.

CHAPTER 7

Why not django-axes

django-axes is great but it puts everything in the database, and this causes a bottle neck when you have a lot of data. It slows down the auth requests by as much as 200-300ms. This might not be much for some sites, but for others it is too long.

This started out as a fork of django-axes, and is using as much of their code as possible, and removing the parts not needed, and speeding up the lookups to improve the login.

CHAPTER 8

How django-defender works

1. When someone tries to login, we first check to see if they are currently blocked. We check the username they are trying to use, as well as the IP address. If they are blocked, goto step 5. If not blocked go to step 2.
2. They are not blocked, so we check to see if the login was valid. If valid go to step 6. If not valid go to step 3.
3. Login attempt wasn't valid. Add their username and IP address for this attempt to the cache. If this brings them over the limit, add them to the blocked list, and then goto step 5. If not over the limit goto step 4.
4. Login was invalid, but not over the limit. Send them back to the login screen to try again.
5. User is blocked: Send them to the blocked page, telling them they are blocked, and give an estimate on when they will be unblocked.
6. Login is valid. Reset any failed login attempts, and forward to their destination.

CHAPTER 9

Cache backend

Defender uses the cache to save the failed attempts.

9.1 Cache keys

Counters:

- prefix:failed:ip:[ip] (count, TTL)
- prefix:failed:username:[username] (count, TTL)

Booleans (if present it is blocked):

- prefix:blocked:ip:[ip] (true, TTL)
- prefix:blocked:username:[username] (true, TTL)

CHAPTER 10

Customizing django-defender

You have a couple options available to you to customize `django-defender` a bit. These should be defined in your `settings.py` file.

- `DEFENDER_LOGIN_FAILURE_LIMIT`: Int: The number of login attempts allowed before a record is created for the failed logins. [Default: `3`]
- `DEFENDER_LOGIN_FAILURE_LIMIT_USERNAME`: Int: The number of login attempts allowed on a username before a record is created for the failed logins. [Default: `DEFENDER_LOGIN_FAILURE_LIMIT`]
- `DEFENDER_LOGIN_FAILURE_LIMIT_IP`: Int: The number of login attempts allowed from an IP before a record is created for the failed logins. [Default: `DEFENDER_LOGIN_FAILURE_LIMIT`]
- `DEFENDER_BEHIND_REVERSE_PROXY`: Boolean: Is defender behind a reverse proxy? [Default: `False`]
- `DEFENDER_REVERSE_PROXY_HEADER`: String: the name of the http header with your reverse proxy IP address [Default: `HTTP_X_FORWARDED_FOR`]
- `DEFENDER_LOCK_OUT_BY_IP_AND_USERNAME`: Boolean: Locks a user out based on a combination of IP and Username. This stops a user denying access to the application for all other users accessing the app from behind the same IP address. [Default: `False`]
- `DEFENDER_DISABLE_IP_LOCKOUT`: Boolean: If this is True, it will not lockout the users IP address, it will only lockout the username. [Default: False]
- `DEFENDER_DISABLE_USERNAME_LOCKOUT`: Boolean: If this is True, it will not lockout usernames, it will only lockout IP addresess. [Default: False]
- `DEFENDER_COOLOFF_TIME`: Int: If set, defines a period of inactivity after which old failed login attempts and username/ip lockouts will be forgotten. An integer, will be interpreted as a number of seconds. If 0, neither the failed login attempts nor the username/ip locks will expire. [Default: `300`]
- `DEFENDER_ATTEMPT_COOLOFF_TIME`: Int: If set, overrides the period of inactivity after which old failed login attempts will be forgotten set by DEFENDER_COOLOFF_TIME. An integer, will be interpreted as a number of seconds. If 0, the failed login attempts will not expire. [Default: `DEFENDER_COOLOFF_TIME`]
- `DEFENDER_LOCKOUT_COOLOFF_TIME`: Int or List: If set, overrides the period of inactivity after which username/ip lockouts will be forgotten set by DEFENDER_COOLOFF_TIME. An integer, will be interpreted as a number of seconds. A list of integers, will be interpreted as a number of seconds for users

with the integer's index being how many previous lockouts (up to some maximum) occurred in the last DEFENDER_ACCESS_ATTEMPT_EXPIRATION hours. If the property is set to 0 or [], the username/ip lockout will not expire. [Default: DEFENDER_COOLOFF_TIME]

- DEFENDER_LOCKOUT_TEMPLATE: String: [Default: None] If set, specifies a template to render when a user is locked out. Template receives the following context variables:
  - cooloff_time_seconds: The cool off time in seconds
  - cooloff_time_minutes: The cool off time in minutes
  - failure_limit: The number of failures before you get blocked.
- DEFENDER_USERNAME_FORM_FIELD: String: the name of the form field that contains your users usernames. [Default: username]
- DEFENDER_CACHE_PREFIX: String: The cache prefix for your defender keys. [Default: defender]
- DEFENDER_LOCKOUT_URL: String: The URL you want to redirect to if someone is locked out.
- DEFENDER_REDIS_URL: String: the redis url for defender. [Default: redis://localhost:6379/0] (Example with password: redis://:mypassword@localhost:6379/0)
- DEFENDER_REDIS_PASSWORD_QUOTE: Boolean: if special character in redis password (like '@'), we can quote password urllib.parse.quote("password!@#"), and set to True. [Default: False]
- DEFENDER_REDIS_NAME: String: the name of your cache client on the CACHES django setting. If set, DEFENDER_REDIS_URL will be ignored. [Default: None]
- DEFENDER_STORE_ACCESS_ATTEMPTS: Boolean: If you want to store the login attempt to the database, set to True. If False, it is not saved [Default: True]
- DEFENDER_USE_CELERY: Boolean: If you want to use Celery to store the login attempt to the database, set to True. If False, it is saved inline. [Default: False]
- DEFENDER_ACCESS_ATTEMPT_EXPIRATION: Int: Length of time in hours for how long to keep the access attempt records in the database before the management command cleans them up. [Default: 24]
- DEFENDER_GET_USERNAME_FROM_REQUEST_PATH: String: The import path of the function that access username from request. If you want to use custom function to access and process username from request - you can specify it here. [Default: defender.utils.username_from_request]

10.1 Rationale for using DEFENDER_ATTEMPT_COOLOFF_TIME and DEFENDER_LOCKOUT_COOLOFF_TIME

While using DEFENDER_COOLOFF_TIME alone is sufficent for most use cases, when using defender in some specific scenarios such as in a high security setting, developers may wish to have finer grained control over how long invalid login attempts are "remembered" while under consideration for lockout compared to the time those lockout keys are actually locked out from the system. DEFENDER_ATTEMPT_COOLOFF_TIME and DEFENDER_LOCKOUT_COOLOFF_TIME allow for this exact fine grained configuration.

We can also take a low security and low scale example like a high school's website. Such a website might be run on some of the school's computers and administrated by the school's IT staff and computer science teachers (if lucky enough to have any). In this scenario we can imagine that there are significant portions of the website accessible without authentication, but logging in to the website could provide access to some relatively privileged information such as the student's name, email, grades, and class schedule. Finally since there is an email linked with the account, we will assume that there is password reset functionality which unblocks the account when completed. In such a case, one could imagine that there is no need to remember failed logins for long periods of time since the application would simply wish to protect against potential denial of service

attacks. This could be accomplished keeping DEFENDER_ATTEMPT_COOLOFF_TIME low, say 30 seconds, and setting DEFENDER_LOCKOUT_COOLOFF_TIME to something much higher like 600 seconds. By keeping DEFENDER_ATTEMPT_COOLOFF_TIME low and locking out bad actors for significant periods of time by setting DEFENDER_LOCKOUT_COOLOFF_TIME high, rapid brute force login attacks will still be defeated and their small server will have more space in their cache for other data. And by providing password reset functionality as described above, these hypothetical administrators could limit their required involvement in unblocking real users while retaining the intended accessibility of their website.

While the previous example is somewhat contrived, the full power of these configurations is demonstrated with the following explanation and example.

When DEFENDER_STORE_ACCESS_ATTEMPTS is True, DEFENDER_LOCKOUT_COOLOFF_TIME can also be configured as a list of integers. When configured as a list, the number of previous failed login attempts for the configured lockout key is divided by DEFENDER_LOGIN_FAILURE_LIMIT to produce an intentionally overestimated count of the number of failed logins for the period defined by DEFENDER_ACCESS_ATTEMPT_EXPIRATION. This ends up being an overestimate because the time between the failed login attempts is not considered when doing this calculation. While this may seem harsh, in some specific scenarios the additional protection against slower attacks can be worth thepotentialinconvenience caused to real users of the system.

One such example of this could be a public web accessible web application that houses sensitive information of it's users (let's say personal financial records). The application and data therein should be accessible with minimal interruption, however security is integral so delays can be tolerated up to a point. Under these circumstances we may have a desire to simply set DEFENDER_COOLOFF_TIME to a very large integer or even 0 for maximum protection. But this would mean that if a real userdoesget locked out of the system, we will need an administrator to manually unblock them which of course is cumbersome and costly. By setting DEFENDER_ATTEMPT_COOLOFF_TIME to a large enough number, let's say 600 and setting DEFENDER_LOCKOUT_COOLOFF_TIME to a list of increasing integers (ie. [60, 120, 300, 600, 0]) we can protect our theoretical application comprably to if we had simply set DEFENDER_COOLOFF_TIME to 600 while disrupting our users significantly less.

CHAPTER 11

Adapting to other authentication methods

`defender` can be used for authentication other than `Django authentication system`. E.g. if `django-rest-framework` authentication has to be protected from brute force attack, a custom authentication method can be implemented.

There's sample `BasicAuthenticationDefender` class based on `djangorestframework.BasicAuthentication`:

```
import base64
import binascii

from django.utils.translation import gettext_lazy as _

from rest_framework import HTTP_HEADER_ENCODING, exceptions
from rest_framework.authentication import (
    BasicAuthentication,
    get_authorization_header,
)

from defender import utils
from defender import config

class BasicAuthenticationDefender(BasicAuthentication):

    def get_username_from_request(self, request):
        auth = get_authorization_header(request).split()
        return base64.b64decode(auth[1]).decode(HTTP_HEADER_ENCODING).partition(':
↪')[0]

    def authenticate(self, request):
        auth = get_authorization_header(request).split()

        if not auth or auth[0].lower() != b'basic':
            return None
```

(continues on next page)

(continued from previous page)

```
        if len(auth) == 1:
            msg = _('Invalid basic header. No credentials provided.')
            raise exceptions.AuthenticationFailed(msg)
        elif len(auth) > 2:
            msg = _('Invalid basic header. Credentials string should not contain␣
↪spaces.')
            raise exceptions.AuthenticationFailed(msg)

        if utils.is_already_locked(request, get_username=self.get_username_from_
↪request):
            detail = "You have attempted to login {failure_limit} times, with no␣
↪success." \
                     "Your account is locked for {cooloff_time_seconds} seconds" \
                     "".format(
                        failure_limit=config.FAILURE_LIMIT,
                        cooloff_time_seconds=config.LOCKOUT_COOLOFF_TIME[
                           defender_utils.get_lockout_cooloff_time(username=self.get_
↪username_from_request(request))
                        ]
                     )
            raise exceptions.AuthenticationFailed(_(detail))

        try:
            auth_parts = base64.b64decode(auth[1]).decode(HTTP_HEADER_ENCODING).
↪partition(':')
        except (TypeError, UnicodeDecodeError, binascii.Error):
            msg = _('Invalid basic header. Credentials not correctly base64 encoded.')
            raise exceptions.AuthenticationFailed(msg)

        userid, password = auth_parts[0], auth_parts[2]
        login_unsuccessful = False
        login_exception = None
        try:
            response = self.authenticate_credentials(userid, password)
        except exceptions.AuthenticationFailed as e:
            login_unsuccessful = True
            login_exception = e

        utils.add_login_attempt_to_db(request,
                                      login_valid=not login_unsuccessful,
                                      get_username=self.get_username_from_request)
        # add the failed attempt to Redis in case of a failed login or resets the␣
↪attempt count in case of success
        utils.check_request(request,
                            login_unsuccessful=login_unsuccessful,
                            get_username=self.get_username_from_request)
        if login_unsuccessful:
            raise login_exception

        return response
```

To make it work add `BasicAuthenticationDefender` to `DEFAULT_AUTHENTICATION_CLASSES` above all other authentication methods in your `settings.py`.

CHAPTER 12

Adapting to other authentication methods :- django-rest-auth in djangorestframework

`defender` can be incorporated with the combination of `django-rest-framework` and `django-rest-auth` which can be used to authenticate users.

12.1 Reference

- https://www.django-rest-framework.org/
- https://django-rest-auth.readthedocs.io/en/latest/

Below is a sample `BasicAuthenticationDefender` class based on `rest_framework.authentication.TokenAuthentication` which uses `django-rest-auth` library for user authentication.

```
import base64
import binascii

from django.conf import settings
from django.contrib.auth import get_user_model, authenticate
from django.contrib.auth.forms import PasswordResetForm, SetPasswordForm
from django.contrib.auth.tokens import default_token_generator
from django.utils.http import urlsafe_base64_decode as uid_decoder
from django.utils.translation import ugettext_lazy as _
from django.utils.encoding import force_text
from rest_framework import serializers, exceptions, HTTP_HEADER_ENCODING
from rest_framework.exceptions import ValidationError
from defender import utils as defender_utils
from defender import config
from rest_framework.authentication import (
    get_authorization_header,
)
```

(continues on next page)

(continued from previous page)

```
# Get the UserModel
UserModel = get_user_model()

class BasicAuthenticationDefender(serializers.Serializer):

   username = serializers.CharField(required=False, allow_blank=True)
   email = serializers.EmailField(required=False, allow_blank=True)
   password = serializers.CharField(style={'input_type': 'password'})

   def authenticate(self, **kwargs):
     request = self.context['request']

     if hasattr(settings, 'ACCOUNT_AUTHENTICATION_METHOD'):
         login_field = settings.ACCOUNT_AUTHENTICATION_METHOD
     else:
         login_field = 'username'
     userid = self.username_from_request(request, login_field)

     if defender_utils.is_already_locked(request, username=userid):
         detail = "You have attempted to login {failure_limit} times with no success.
↪"
                  .format(
                      failure_limit=config.FAILURE_LIMIT,
                      cooloff_time_seconds=config.LOCKOUT_COOLOFF_TIME[defender_utils.
↪get_lockout_cooloff_time(username=userid)]
                  )
         raise exceptions.AuthenticationFailed(_(detail))

     login_unsuccessful = False
     login_exception = None
     try:
         response = authenticate(request, **kwargs)
         if response == None:
             login_unsuccessful = True
             msg = _('Unable to log in with provided credentials.')
             # raise exceptions.ValidationError(msg)
             login_exception = exceptions.ValidationError(msg)
     except exceptions.AuthenticationFailed as e:
         login_unsuccessful = True
         login_exception = e

     defender_utils.add_login_attempt_to_db(request,
                                            login_valid=not login_unsuccessful,
                                            username=userid)

     user_not_blocked = defender_utils.check_request(request,
                                                     login_unsuccessful=login_
↪unsuccessful,
                                                     username=userid)
     if user_not_blocked and not login_unsuccessful:
         return response

     raise login_exception

   def _validate_email(self, email, password):
     user = None
```

(continues on next page)

(continued from previous page)

```
    if email and password:
        user = self.authenticate(email=email, password=password)
    else:
        msg = _('Must include "email" and "password".')
        raise exceptions.ValidationError(msg)

    return user

  def _validate_username(self, username, password):
    user = None

    if username and password:
        user = self.authenticate(username=username, password=password)
    else:
        msg = _('Must include "username" and "password".')
        raise exceptions.ValidationError(msg)

    return user

  def _validate_username_email(self, username, email, password):
    user = None

    if email and password:
        user = self.authenticate(email=email, password=password)
    elif username and password:
        user = self.authenticate(username=username, password=password)
    else:
        msg = _('Must include either "username" or "email" and "password".')
        raise exceptions.ValidationError(msg)

    return user

  def validate(self, attrs):
    username = attrs.get('username')
    email = attrs.get('email')
    password = attrs.get('password')

    user = None

    if 'allauth' in settings.INSTALLED_APPS:
        from allauth.account import app_settings

        # Authentication through email
        if app_settings.AUTHENTICATION_METHOD == app_settings.AuthenticationMethod.
↪EMAIL:
            user = self._validate_email(email, password)

        # Authentication through username
        elif app_settings.AUTHENTICATION_METHOD == app_settings.AuthenticationMethod.
↪USERNAME:
            user = self._validate_username(username, password)

        # Authentication through either username or email
        else:
            user = self._validate_username_email(username, email, password)

    else:
```

(continues on next page)

(continued from previous page)

```
        # Authentication without using allauth
        if email:
            try:
                username = UserModel.objects.get(
                    email__iexact=email).username()
            except UserModel.DoesNotExist:
                pass

        if username:
            user = self._validate_username_email(username, '', password)

    # Did we get back an active user?
    if user:
        if not user.is_active:
            msg = _('User account is disabled.')
            raise exceptions.ValidationError(msg)
    else:
        msg = _('Unable to log in with provided credentials.')
        raise exceptions.ValidationError(msg)

    # If required, is the email verified?
    if 'rest_auth.registration' in settings.INSTALLED_APPS:
        from allauth.account import app_settings
        if app_settings.EMAIL_VERIFICATION == app_settings.EmailVerificationMethod.
↪MANDATORY:
            email_address = user.emailaddress_set.get(email=user.email)
            if not email_address.verified:
                raise serializers.ValidationError(
                    _('E-mail is not verified.'))

    attrs['user'] = user
    return attrs

  def username_from_request(self, request, login_field):
    user_data = request._data
    return user_data[login_field]
```

To make it work add `BasicAuthenticationDefender` to `REST_AUTH_SERIALIZERS` dictionary in your `settings.py` under the key `LOGIN_SERIALIZER`. For example, in your settings.py add the below line,

```
REST_AUTH_SERIALIZERS = {
    'LOGIN_SERIALIZER': '<path to your basic authentication defender python file>.
↪BasicAuthenticationDefender',
}
```

CHAPTER 13

Adapting for password reset forms

`defender` can be adapted for Django's `PasswordResetView` to prevent too many submissions.

We need to create some new views that subclass Django's built-in `LoginView`, `PasswordResetView` & `PasswordResetConfirmView` — then use these views in our `urls.py` as replacements for Django's built-ins.

The views block based on email address submitted on the password reset view. This is different than the default implementation (which uses username), so we have to be careful to clean up after ourselves on sign-in & completed password reset.

```
from defender import utils as def_utils
from django.contrib.auth import views as auth_views

class UserSignIn(auth_views.LoginView):
    def form_valid(self, form):
        """Force clear all the cached Defender statues for the authenticated user's␣
↪email address."""
        super_valid = super().form_valid(form)
        def_utils.check_request(self.request, False, username=form.get_user().email)
        return super_valid

class PasswordResetBruteForceProtectedView(auth_views.PasswordResetView):
    def get(self, request, *args, **kwargs):
        """Confirm the user isn't already blocked by IP before showing the password␣
↪reset view."""
        if def_utils.is_already_locked(request):
            return def_utils.lockout_response(request)
        return super().get(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        """
        Confirm the user isn't already blocked by IP before allowing form POST.

        Also, force log this form POST as a single entry in the Defender cache,␣
↪against the submitted email address.
        """
```

(continues on next page)

(continued from previous page)

```
        if def_utils.is_already_locked(request):
            return def_utils.lockout_response(request)
        def_utils.check_request(
            request, login_unsuccessful=True, username=request.POST.get("email")
        )
        return super().post(request, *args, **kwargs)


class PasswordResetConfirmBruceForceProtectedView(auth_views.
↪PasswordResetConfirmView):
    def get(self, request, *args, **kwargs):
        """Confirm the user isn't already blocked by IP before showing the password␣
↪confirm view."""
        if def_utils.is_already_locked(request):
            return def_utils.lockout_response(request)
        return super().get(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        """Confirm the user isn't already blocked by IP before allowing form POST for␣
↪the password change confirmation."""
        if def_utils.is_already_locked(request):
            return def_utils.lockout_response(request)
        return super().post(request, *args, **kwargs)

    def form_valid(self, form):
        """Force clear all the cached Defender statues for the user's email address␣
↪after successfully changing their password."""
        super_valid = super().form_valid(form)
        def_utils.check_request(
            self.request, login_unsuccessful=False, username=self.user.email
        )
        return super_valid
```

CHAPTER 14

Django signals

`django-defender` will send signals when blocking a username or an IP address. To set up signal receiver functions:

```
from django.dispatch import receiver

from defender import signals

@receiver(signals.username_block)
def username_blocked(username, **kwargs):
    print("%s was blocked!" % username)

@receiver(signals.ip_block)
def ip_blocked(ip_address, **kwargs):
    print("%s was blocked!" % ip_address)
```

CHAPTER 15

Running tests

Tests can be run, after you clone the repository and having Django installed, like:

```
PYTHONPATH=$PYTHONPATH:$PWD django-admin test defender --settings=defender.test_
↪settings
```

With Code coverage:

```
PYTHONPATH=$PYTHONPATH:$PWD coverage run --source=defender $(which django-admin) test␣
↪defender --settings=defender.test_settings
```

CHAPTER 16

Releasing

1. `python setup.py sdist`
2. `twine upload dist/*`

CHAPTER 17

Contributing

This is a Jazzband project. By contributing you agree to abide by the Contributor Code of Conduct and follow the guidelines.

CHAPTER 18

Changes

CHAPTER 19

0.9.7

- Fix bug related to using a redis version less than 6 and not having a password. [@kencochrane]
- Fix bug in remove_prefix method [@dashgin]

CHAPTER 20

0.9.6

- Confirm support for Django 4.1
- Add `DEFENDER_ATTEMPT_COOLOFF_TIME` config to override `DEFENDER_COOLOFF_TIME` specifically for attempt lifespan [@djmore4]
- Add `DEFENDER_LOCKOUT_COOLOFF_TIME` config to override `DEFENDER_COOLOFF_TIME` specifically for lockout duration [@djmore4]

CHAPTER 21

0.9.5

- Add username support to Redis configuration. [@erdos4d]

21.1 0.9.4

- Remove port number from IP address string when behind reverse proxy [@ndrsn]

21.2 0.9.3

- Drop Python 3.6 support from package specifiers.

21.3 0.9.2

- Drop Python 3.6 support.
- Drop Django 3.1 support.
- Confirm support for Django 4.0
- Confirm support for Python 3.10
- Drop Django 2.2 support.

21.4 0.9.1

- Fix failing tests for Django main development branch (Django 4.0) [@JonathanWillitts]

21.5 0.9.0

- Move CI to GitHub Actions.
- Drop support for Django 3.0
- Add support for Django 3.2

21.6 0.8.0

- FIX: Change setup.py to allow for Django 3.1.x versions [@s4ke]
- FIX: dynamic load celery [@balsagoth]
- FIX: Redis requirement updated [@flaviomartins]
- FIX: if special character in redis password, we can set DEFENDER_REDIS_PASSWORD_QUOTE to True, and use quote password [@calmkart]

21.7 0.7.0

- Add support for Django 3.0 [@deeprave]
- Remove support from deprecated Python 3.4 and Django 2.0. [@aleksihakli]
- Add Read the Docs documentation. [@aleksihakli]
- Add support for Python 3.7, Python 3.8, PyPy3. [@aleksihakli]

21.8 0.6.2

- Add and test support for Django 2.2 [@chrisledet]
- Add support for redis client 3.2.1 [@softinio]

21.9 0.6.1

- Add redispy 3.2.0 compatibility [@nrth]

21.10 0.6.0

- Remove Python 3.3 [@fr0mhell]
- Remove Django 1.8-1.10 [@fr0mhell]
- Add Celery v4 [@fr0mhell]
- Update travis config [@fr0mhell]
- Update admin URL [@fr0mhell]

21.11 0.5.5

- Add new setting `DEFENDER_GET_USERNAME_FROM_REQUEST_PATH` for control how username is accessed from request [@andrewshkovskii]
- Add new argument `get_username` for `decorators.watch_login` to propagate `get_username` argument to other utils functions calls done in `watch_login` [@andrewshkovskii]

21.12 0.5.4

- Add 2 new setting variables for more granular failure limit control [@williamboman]
- Add ssl option when instantiating StrictRedis [@mjrimrie]
- Send signals when blocking username or ip [@williamboman]

21.13 0.5.3

- Remove mockredis as install requirement, make only test requirement [@blueyed]

21.14 0.5.2

- Fix regex in 'unblock_username_view' to handle special symbols [@ruthus18]
- Fix django requires version for 1.11.x [@kencochrane]
- Remove hiredis dependency [@ericbuckley]
- Correctly get raw client when using django_redis cache. [@cburger]
- Replace django.core.urlresolvers with django.urls For Django 2.0 [@s-wirth]
- Add username kwarg for providing username directly rather than via callback arg [@williamboman]
- Only use the username if it is actually provided [@cobusc]

21.15 0.5.1

- Middleware fix for django >- 1.10 #93 [@Temeez]
- Force the username to lowercase #90 [@MattBlack85]

21.16 0.5.0

- Better support for Django 1.11 [@dukebody]
- Add support to share redis config with django.core.cache [@Franr]
- Allow decoration of functions beyond the admin login [@MattBlack85]
- Doc improvements [@dukebody]

- Allow usernames with plus signs in unblock view [@dukebody]
- Code cleanup [@KenCochrane]

21.17 0.4.3

- Flex version requirements for dependencies
- Better support for Django 1.10

21.18 0.4.2

- Better support for Django 1.9

21.19 0.4.1

- Minor refactor to make it easier to retrieve username.

21.20 0.4.0

- Add `DEFENDER_DISABLE_IP_LOCKOUT` and added support for Python 3.5

21.21 0.3.2

- Add `DEFENDER_LOCK_OUT_BY_IP_AND_USERNAME`, and changed settings to support django 1.8.

21.22 0.3.1

- Fix the management command name

21.23 0.3

- Add management command `cleanup_django_defender` to clean up access attempt table.
- Add `DEFENDER_STORE_ACCESS_ATTEMPTS` config to say if you want to store attempts to DB or not.
- Add `DEFENDER_ACCESS_ATTEMPT_EXPIRATION` config to specify how long to store the access attempt records in the db, before the management command cleans them up.
- Change the Django admin page to remove some filters which were making the page load slow with lots of login attempts in the database.

21.24 0.2.2

- Another bug fix release for more missing files in distribution

21.25 0.2.1

- Bug fixes for packing missing files

21.26 0.2

- Add fixes to include possible security issue

21.27 0.1

- Initial Version