django-security-logger Documentation

Release 1.4

Luboš Mátl

Jan 25, 2023

Contents

Django-security-logger is library for logging input, output request and Django commands. Library can be used with django-reversion to log which data was changed in a request. The library provides throttling security mechanism.

CHAPTER 1

Project Home

https://github.com/druids/django-security

CHAPTER 2

Documentation

https://django-security-logger.readthedocs.org/en/latest

2.1 Content

2.1.1 Installation

Using PIP

You can install django-security-logger via pip:

```
$ pip install django-security-logger
```

2.1.2 Configuration

After installation you must go through these steps:

Required Settings

The following variables have to be added to or edited in the project's `settings.py`:

For using the library you just add `security` to `INSTALLED_APPS` variable:

```
INSTALLED_APPS = (
    ...
    'security',
    ...
)
```

Next you muse select which logging backend you want to use and add it into `INSTALLED_APPS` variable:

```
INSTALLED_APPS = (
    ...
    'security.backends.sql',  # log is stored into SQL DB with Django ORM
    'security.backends.elasticsearch',  # log is stored into Elasticsearch DB with␣
↪Elasticsearch-DSL
    'security.backends.logging',  # standard python log
    ...
)
```

Next you must add `security.middleware.LogMiddleware` to list of middlewares, the middleware should be added after authentication middleware:

```
MIDDLEWARE = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'security.middleware.LogMiddleware',
    ...
)
```

SQL backend

For SQL backend if your database configuration uses atomic requests, it's highly recommended to use second non atomic connection to your database for security logs. Possible rollback will not remove logs.

Example:

```
DATABASES = {
    'default': {
        'NAME': 'db_name',
        'USER': 'db_user',
        'PASSWORD': 'db_password',
        'HOST': 'postgres',
        'PORT': 5432,
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'ATOMIC_REQUESTS': True,
    },
    'log': {
        'NAME': 'db_name',
        'USER': 'db_user',
        'PASSWORD': 'db_password',
        'HOST': 'postgres',
        'PORT': 5432,
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'ATOMIC_REQUESTS': False,
        'TEST': {
            'MIRROR': 'default', # Test purposes
        },
    },
}
DATABASE_ROUTERS = ['security.backends.sql.db_router.MirrorSecurityLoggerRouter']  #␣
↪DB router which defines connection for logs
SECURITY_DB_NAME = 'log'
```

For test purposes you will need to configure both databases to be tested:

```
from django.test.testcases import TestCase

class YourTestCase(TestCase):
    databases = ('default', 'log')
```

The second solution is have a independent database for logs in this case you can use `MultipleDBSecurityLoggerRouter`:

```
DATABASES = {
    'default': {
        'NAME': 'db_name',
        'USER': 'db_user',
        'PASSWORD': 'db_password',
        'HOST': 'postgres',
        'PORT': 5432,
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'ATOMIC_REQUESTS': True,
    },
    'log': {
        'NAME': 'log_db_name',
        'USER': 'log_db_user',
        'PASSWORD': 'log_db_password',
        'HOST': 'log_db_postgres',
        'PORT': 5432,
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'ATOMIC_REQUESTS': False,
    },
}
DATABASE_ROUTERS = ['security.backends.sql.dbr_router.MultipleDBSecurityLoggerRouter
↪']  # DB router which defines connection for logs
SECURITY_DB_NAME = 'log'
```

Elasticsearch backend

Elasticsearch backend can be configured via `SECURITY_ELASTICSEARCH_DATABASE` variable:

```
SECURITY_ELASTICSEARCH_DATABASE = {
    'host': 'localhost',
}
```

For elasticsearch database initialization you must run `./manage.py init_elasticsearch_log` command to create indexes in the database.

There are two ways how to store logs in the elasticsearch: direct connection or via logstash. Direct connection is defined by default and no extra configuration is not required. For the logstash solution you need to allow configuration `SECURITY_ELASTICSEARCH_LOGSTASH_WRITER`:

```
SECURITY_ELASTICSEARCH_LOGSTASH_WRITER = True
```

Now you have to run logstash with configuration defined in `logstash.example.conf`.

Django will send data to the logstash via logger with this settings:

```
LOGGING.update({
    'handlers': {
        ...
```

(continues on next page)

(continued from previous page)

```
        'logstash': {
            'level': 'INFO',
            'class': 'security.backends.elasticsearch.logstash.handler_tcp.
↪TCPLogstashHandler',
            'host': 'logstash',
            'port': 5044,
            'formatter': 'logstash',
        },
        ...
    },
    'loggers': {
        ...
        'security.logstash': {
            'handlers': ['logstash'],
            'level': 'INFO',
            'propagate': False,
        },
    }
```

Testing backend

For testing purposes you can use *'security.backends.testing'* and turn off log writers:

```
SECURITY_BACKEND_WRITERS = []  # Turn off log writers
```

Your test you can surround with *security.backends.testing.capture_security_logs* decorator/context processor:

```
def your_test():
   with capture_security_logs() as logged_data:
        ...
        assert_length_equal(logged_data.input_request, 1)
        assert_length_equal(logged_data.output_request, 1)
        assert_length_equal(logged_data.command, 1)
        assert_length_equal(logged_data.celery_task_invocation, 1)
        assert_length_equal(logged_data.celery_task_run, 1)
        assert_equal(logged_data.input_request[0].request_body, 'test')
```

Readers

Some `elasticsearch`, `sql` and `testing` backends can be used as readers too. You can use these helpers to get data from these backends (no matter which wan is set):

- `security.backends.reader.get_count_input_requests(from_time, ip=None, path=None, view_slug=None, slug=None, method=None, exclude_log_id=None)` - to get count input requests with input arguments
- `security.backends.reader.get_logs_related_with_object(logger_name, related_object)` - to get list of logs which are related with object

Setup

SECURITY_DEFAULT_THROTTLING_VALIDATORS_PATH
: Path to the file with configuration of throttling validators. Default value is `'security.default_validators'`.

SECURITY_THROTTLING_FAILURE_VIEW
Path to the view that returns throttling failure. Default value is `'security.views.throttling_failure_view'`.

SECURITY_LOG_REQUEST_IGNORE_URL_PATHS
Set of URL paths that are omitted from logging.

SECURITY_LOG_REQUEST_IGNORE_IP
Tuple of IP addresses that are omitted from logging.

SECURITY_LOG_REQUEST_BODY_LENGTH
Maximal length of logged request body. More chars than defined are truncated. Default value is `1000`. If you set `None` value the request body will not be truncated.

SECURITY_LOG_RESPONSE_BODY_LENGTH
Maximal length of logged response body. More chars than defined are truncated. Default value is `1000`. If you set `None` value the response body will not be truncated.

SECURITY_LOG_RESPONSE_BODY_CONTENT_TYPES
Tuple of content types which request/response body are logged for another content types body are removed. Default value is `('application/json', 'application/xml', 'text/xml', 'text/csv', 'text/html', 'application/xhtml+xml')`.

SECURITY_LOG_JSON_STRING_LENGTH
If request/response body are in JSON format and body is longer than allowed the truncating is done with a smarter way. String JSON values longer than value of this setting are truncated. Default value is `250`. If you set `None` value this method will not be used.

SECURITY_COMMAND_LOG_EXCLUDED_COMMANDS
Because logger supports Django command logging too this setting contains list of commands that are omitted from logging. Default value is `('runserver', 'makemigrations', 'migrate', 'sqlmigrate', 'showmigrations', 'shell', 'shell_plus', 'test', 'help', 'reset_db', 'compilemessages', 'makemessages', 'dumpdata', 'loaddata')`.

SECURITY_HIDE_SENSITIVE_DATA_PATTERNS
Setting contains patterns for regex function that goes through body and headers and replaces sensitive data with defined replacement.

SECURITY_HIDE_SENSITIVE_DATA
If set to True enables replacing of sensitive data with defined replacement *SECURITY_HIDE_SENSITIVE_DATA_PATTERNS* inside body and headers. Default value is `True`.

SECURITY_SENSITIVE_DATA_REPLACEMENT
Setting contains sensitive data replacement value. Default value is `'[Filtered]'`.

SECURITY_APPEND_SLASH
Setting same as Django setting `APPEND_SLASH`. Default value is `True`.

SECURITY_CELERY_STALE_TASK_TIME_LIMIT_MINUTES
Default wait timeout to set not triggered task to the failed state. Default value is `60`.

SECURITY_LOG_OUTPUT_REQUESTS
Enable logging of output requests via logging module. Default value is `True`.

SECURITY_AUTO_GENERATE_TASKS_FOR_DJANGO_COMMANDS
List or set of Django commands which will be automatically transformed into celery tasks.

SECURITY_LOG_DB_NAME
Name of the database which security uses to log events.

SECURITY_BACKENDS
With this setting you can select which backends will be used to store logs. Default value is `None` which means all installed backends are used.

SECURITY_ELASTICSEARCH_DATABASE
Setting can be used to set Elasticsearch database configuration.

SECURITY_ELASTICSEARCH_AUTO_REFRESH
Every write to the Elasticsearch database will automatically call auto refresh.

SECURITY_LOG_STRING_IO_FLUSH_TIMEOUT
Timeout which set how often will be stored output stream to the log. Default value is `5` (s).

SECURITY_LOG_STRING_OUTPUT_TRUNCATE_LENGTH
Max length of log output string. Default value is `10000`.

SECURITY_LOG_STRING_OUTPUT_TRUNCATE_OFFSET
Because too frequent string truncation can cause high CPU load, log string is truncated by more characters. This setting defines this value which is by default `1000`.

2.1.3 Commands

purge_logs

Remove old request, command or celery logs that are older than defined value, parameters:

- `expiration` - timedelta from which logs will be removed. Units are h - hours, d - days, w - weeks, m - months, y - years
- `noinput` - tells Django to NOT prompt the user for input of any kind
- `backup` - tells Django where to backup removed logs in JSON format
- `type` - tells Django what type of requests should be removed (input-request/output-request/command/celery-task-invocation/celery-task-run)

Logs can be removed only for `elasticsearch` and `sql` backends.

set_celery_task_log_state

Set celery tasks which are in WAITING state. Tasks which were not started more than `SECURITY_CELERY_STALE_TASK_TIME_LIMIT_MINUTES` (by default 60 minutes) to the failed state. Task with succeeded/failed task run is set to succeeded/failed state.

2.1.4 Logger

Input requests

Input requests are logged automatically with `security.middleware.LogMiddleware`. The middleware creates `security.models.InputLoggedRequest` object before sending request to next middleware. Response data to the logged requests are completed in the end. You can found logged request in the Django request objects with that way `request.input_logged_request`.

View decorators

There are several decorators for views and generic views that can be used for view logging configuration:

- `security.decorators.hide_request_body` - decorator for view that removes request body from logged request
- `security.decorators.hide_request_body_all` - decorator for generic view class that removes request body from logged request
- `security.decorators.log_exempt` - decorator for view that exclude all requests to this view from logging
- `security.decorators.log_exempt_all` - decorator for generic view class that exclude all requests to this view from logging

Output requests

Logging of output requests is a little bit complicated and is related to the way how output requests are performed. You can enable logging of output requests to stdout via `SECURITY_LOG_OUTPUT_REQUESTS` (default `True`) in following format: `"{request_timestamp}" "{response_timestamp}" "{response_time}" "{http_code}" "{http_host}" "{http_path}" "{http_method}" "{slug}"`. Security provides two ways how to log output requests:

requests

The first method is used for logging simple HTTP requests using `requests` library. The only change necessary is to import `from security import requests` instead of `import requests`. Same methods (get, post, put, ..) are available as in the requests library. Every method has two extra optional parameters:

- `slug` - text slug that is stored with the logged request to tag concrete logged value
- `related_objects` - list or tuple of related objects that will be related with output logged request

Example where user is stored in the related objects and log slug is set to the value `'request'`:

```
from security import requests
from users.models import User

user = User.objects.first()
requests.get('https:///github.com/druids/', slug='request', related_objects=[user])
```

suds

For SOAP based clients there are extensions to the `suds` library. You must only use `security.suds.Client` class without standard suds client or `security.suds.SecurityRequestsTransport` with standard suds client object. As init data of `security.suds.SecurityRequestsTransport` you can send `slug` and `related_objects`. The `security.suds.Client` has `slug` and `related_objects` input parameter:

```
from security.suds import Client
from users.models import User

user = User.objects.first()
client = Client('http://your.service.url, slug='suds', related_objects=[user])
```

Decorators/context processors

`security.decorators.log_with_data` - because logged requests are stored in models, they are subject to rollback, if you are using transactions. To solve this problem you can use this decorator before Django `transaction.atomic` decorator. The logs are stored on the end of the transaction (even with raised exception). Decorator can be nested, logs are saved only with the last decorator. If you want to join a object with output request log you can use this decorator too. In the example user is logged with output request:

```
from security.decorators import atomic_log
from security import requests

user = User.objects.first()
with log_with_data(slug='github-request', output_requests_related_objects=[user],␣
↪extra_data={'extra': 'data'}):
    requests.get('https://github.com/druids/')
```

Sensitive data

Because some sensitive data inside requests and responses should not be stored (for example password, authorization token, etc.) `django-security-logger` uses regex to find these cases and replace these values with information about hidden value. Patterns are set with `SECURITY_HIDE_SENSITIVE_DATA_PATTERNS` which default setting is:

```
SECURITY_HIDE_SENSITIVE_DATA_PATTERNS = {
    'BODY': (
        r'"password"\s*:\s*"((?:\\"|[^"])*)',
        r'<password>([^<]*)',
        r'password=([^&]*)',
        r'csrfmiddlewaretoken=([^&]*)',
        r'(?i)content-disposition: form-data; name="password"\r\n\r\n.*',
        r'"access_key": "([^"]*)',
    ),
    'HEADERS': (
        r'Authorization',
        r'X_Authorization',
        r'Cookie',
        r'.*token.*',
    ),
    'QUERIES': (
        r'.*token.*',
    ),
}
```

Patterns are split to two groups `BODY`, `HEADERS` and `QUERIES`. There are names of HTTP headers and queries, whose values will be replaced by the replacement. The search is case insensitive. `BODY` is a little bit complicated. If regex groups are used in the pattern only these groups will be replaced with the replacement. If no groups are used, the whole pattern will be replaced.

Commands log

If you want to log commands you must only modify your `mangage.py` file:

```
if __name__ == '__main__':
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'settings')
```

(continues on next page)

(continued from previous page)

```
    # Used function for security to log commands
    from security.management import execute_from_command_line

    sys.path.append(os.path.join(PROJECT_DIR, 'libs'))

    execute_from_command_line(sys.argv)
```

If you want to call command from code, you should use `security.management.call_command` instead of standard Django `call_command` function.

Celery tasks log

If you want to log celery tasks you must install celery library (`celery>=5`). Then you must use `security.task import LoggedTask` as a base class of your celery task, example:

```
from security.task import LoggedTask

@celery_app.task(
    base=LoggedTask,
    bind=True,
    name='sum_task')
def sum_task(self, task_id, a, b):
    return a + b
```

Task result will be automatically logged to the log.

2.1.5 Throttling

In terms of `django-security-logger` throttling is a process responsible for regulating the rate of incoming HTTP requests. There are many ways how to restrict number of requests that may depend on a concrete view. The simplest throttling is to restrict maximum number of request from one IP address per unit of time.

Default configuration

Default throttling configuration is set with `SECURITY_DEFAULT_THROTTLING_VALIDATORS_PATH`. The setting contains path to the file with throttling configuration. Default configuration is `'security.default_validators'` and the config file content is:

```
from .throttling import PerRequestThrottlingValidator


default_validators = (
    PerRequestThrottlingValidator(3600, 1000),  # 1000 per an hour
    PerRequestThrottlingValidator(60, 20),  # 20 per an minute
)
```

Only backends which support reading (`sql`, `elasticsearch` and `testing`) can be used with throttling validators.

Validators

There are only three predefined throttling validators:

- `security.throttling.validators.PerRequestThrottlingValidator` - init parameters are `timeframe` throttling timedelta in seconds, `throttle_at` number of request per one IP address per timeframe and error message.
- `security.throttling.validators.UnsuccessfulLoginThrottlingValidator` - validator with same input parameters as `PerRequestThrottlingValidator` but counts only unsuccessful login request.
- `security.throttling.validators.SuccessfulLoginThrottlingValidator` - validator with same input parameters as `PerRequestThrottlingValidator` but counts only successful login requests.

Custom validator

Creating custom validator is very simple, you only create class with validate method that receives request and if request must be regulated the method raises `security.exception.ThrottlingException`:

```
class CustomValidator:

    def validate(self, request):
      if should_regulate(request):
          raise  ThrottlingException('Your custom message')
```

Decorators

Because throttling can be different per view, there are decorators for changing default validators for concrete view:

- `security.decorators.throttling_exempt()` - marks a view function as being exempt from the throttling protection.
- `security.decorators.throttling_exempt_all()` - marks a view class as being exempt from the throttling protection.
- `security.decorators.throttling(*validators, keep_default=True)` - add throttling validators for view function. You can remove default throttling validators with set `keep_default` to the `False` value.
- `security.decorators.throttling_all(*validators, keep_default=True)` - add throttling validators for view class. You can remove default throttling validators with set `keep_default` to the `False` value.

View

If `security.throttling.exception.ThrottlingException` is raised the specific error view is returned. You can change it with only overriding template named 429.html in your templates. With setting `SECURITY_THROTTLING_FAILURE_VIEW` you can change view function which default code is:

```
from django.shortcuts import render
from django.utils.encoding import force_text


def throttling_failure_view(request, exception):
    response = render(request, '429.html', {'description': force_text(exception)})
    response.status_code = 429
    return response
```

2.1.6 Extra

Django-security-logger provides extra features to improve your logged data.

security.contrib.reversion_log

If you have installed `django-reversion` it is possible to relate input logged requests with concrete object change. Firstly you must add extension to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    ...
    'security.contrib.reversion_log',
    ...
)
```

For `django-reversion` version older than 2.x you must add middleware `security.contrib.reversion_log.middleware.RevisionLogMiddleware` too:

```
MIDDLEWARE = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'security.middleware.LogMiddleware',
    'security.contrib.reversion_log.middleware.RevisionLogMiddleware',
    ...
)
```

Input logged requests and reversion revision objects are related via m2m model `security.contrib.reversion_log.models.InputRequestRevision`

security.contrib.debug_toolbar_log

If you are using `django-debug-toolbar` you can log toolbar results with logged request. You only add extension to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    ...
    'security.contrib.reversion_log',
    ...
)
```

And add `security.contrib.debug_toolbar_log.middleware.DebugToolbarLogMiddleware` on the first place:

```
MIDDLEWARE = (
    'security.contrib.debug_toolbar_log.middleware.DebugToolbarLogMiddleware',
    ...
)
```

Finally you can start log debug toolbar settings with your logged requests by turning on settings:

```
SECURITY_DEBUG_TOOLBAR = True
```

Do not forget turn on django DEBUG.

To show results in `django-is-core` you must set setting:

```
SECURITY_SHOW_DEBUG_TOOLBAR = True
```

django-is-core

Backends `elasticsearch` and `sql` provide prepared django-is-core administration. If you are using django-is-core library you can find admin core classes in: * elasticsearch - `security.elasticsearch.is_core.cores`

- `InputRequestLogCore`
- `OutputRequestLogCore`
- `CommandLogCore`
- `CeleryTaskRunLogCore`
- `CeleryTaskInvocationLogCore`
- **sql - `security.sql.is_core.cores`**
  - `InputRequestLogCore`
  - `OutputRequestLogCore`
  - `CommandLogCore`
  - `CeleryTaskRunLogCore`
  - `CeleryTaskInvocationLogCore`

2.1.7 django-security-logger changelog

1.2.0 - 10/20/2020

- purge migrations because of splitting log to the extra database
- used new version of generic m2m relation which uses relations without FK
- added multiple database router

1.0.6 - 02/06/2020

- Added DB index to celery log task name.
- Celery log *state* is field on model now (is not dynamically computed).
- Celery log *state* is set in *LoggedTask* with methods *on_start_task*, *on_success_task*, *on_failure_task* and *on_retry_task*.
- Command *set_staletasks_to_error_state was* replaced with *set_celery_task_log_state* command.

Index

S