aiohttp_security Documentation

Release 0.4.0-

Andrew Svetlov

Mar 16, 2022

Contents

The library provides security for aiohttp.web.

The current version is 0.4

CHAPTER 1

Contents

1.1 Usage

First of all, what is *aiohttp_security* about?

aiohttp-security is a set of public API functions as well as a reference standard for implementation details for securing access to assets served by a wsgi server.

Assets are secured using authentication and authorization as explained below. *aiohttp-security* is part of the aio-libs project which takes advantage of asynchronous processing using Python's asyncio library.

1.1.1 Public API

The API is agnostic to the low level implementation details such that all client code only needs to implement the endpoints as provided by the API (instead of calling policy code directly (see explanation below)).

Via the API an application can:

(i) remember a user in a local session (`remember()`),

(ii) forget a user in a local session (`forget()`),

(iii) retrieve the *userid* (`authorized_userid()`) of a remembered user from an *identity* (discussed below), and

(iv) check the *permission* of a remembered user (`permits()`).

The library internals are built on top of two concepts:

1) *authentication*, and

2) *authorization*.

There are abstract base classes for both types as well as several pre-built implementations that are shipped with the library. However, the end user is free to build their own implementations.

The library comes with two pre-built identity policies; one that uses cookies, and one that uses sessions[1]. It is envisioned that in most use cases developers will use one of the provided identity policies (Cookie or Session) and implement their own authorization policy.

The workflow is as follows:

1) User is authenticated. This has to be implemented by the developer.
2) Once user is authenticated an identity string has to be created for that user. This has to be implemented by the developer.
3) The identity string is passed to the Identity Policy's remember method and the user is now remembered (Cookie or Session if using built-in). *Only once a user is remembered can the other API methods:* `permits()`, `forget()`, *and* `authorized_userid()` *be invoked* .
4) If the user tries to access a restricted asset the `permits()` method is called. Usually assets are protected using the `check_permission()` helper. This should return True if permission is granted.

The `permits()` method is implemented by the developer as part of the `AbstractAuthorizationPolicy` and passed to the application at runtime via setup.

In addition a `check_authorized()` also exists that requires no permissions (i.e. doesn't call `permits()` method) but only requires that the user is remembered (i.e. authenticated/logged in).

1.1.2 Authentication

Authentication is the process where a user's identity is verified. It confirms who the user is. This is traditionally done using a user name and password (note: this is not the only way).

A authenticated user has no access rights, rather an authenticated user merely confirms that the user exists and that the user is who they say they are.

In *aiohttp_security* the developer is responsible for their own authentication mechanism. *aiohttp_security* only requires that the authentication result in a identity string which corresponds to a user's id in the underlying system.

Note: *identity* is a string that is shared between the browser and the server. Therefore it is recommended that a random string such as a uuid or hash is used rather than things like a database primary key, user login/email, etc.

1.1.3 Identity Policy

Once a user is authenticated the *aiohttp_security* API is invoked for storing, retrieving, and removing a user's *identity*. This is accommplished via AbstractIdentityPolicy's `remember()`, `identify()`, and `forget()` methods. The Identity Policy is therefore the mechanism by which a authenticated user is persisted in the system.

aiohttp_security has two built in identity policy's for this purpose. `CookiesIdentityPolicy` that uses cookies and `SessionIdentityPolicy` that uses sessions via `aiohttp-session` library.

1.1.4 Authorization

Once a user is authenticated (see above) it means that the user has an *identity*. This *identity* can now be used for checking access rights or *permission* using a *authorization* policy.

The authorization policy's `permits()` method is used for this purpose.

[1] jwt - json web tokens in the works

When `aiohttp.web.Request` has an *identity* it means the user has been authenticated and therefore has an *identity* that can be checked by the *authorization* policy.

As noted above, *identity* is a string that is shared between the browser and the server. Therefore it is recommended that a random string such as a uuid or hash is used rather than things like a database primary key, user login/email, etc.

1.2 Reference

1.2.1 Public API functions

`aiohttp_security.`**`setup`**`(`*app*, *identity_policy*, *autz_policy*`)`

Setup `aiohttp` application with security policies.

Parameters

- **`app`** – aiohttp `aiohttp.web.Application` instance.
- **`identity_policy`** – indentification policy, an *`AbstractIdentityPolicy`* instance.
- **`autz_policy`** – authorization policy, an *`AbstractAuthorizationPolicy`* instance.

`coroutine` `aiohttp_security.`**`remember`**`(`*request*, *response*, *identity*, ***kwargs*`)`

Remember *identity* in *response*, e.g. by storing a cookie or saving info into session.

The action is performed by registered *`AbstractIdentityPolicy.remember()`*.

Usually the *identity* is stored in user cookies somehow for using by *`authorized_userid()`* and *`permits()`*.

Parameters

- **`request`** – `aiohttp.web.Request` object.
- **`response`** – `aiohttp.web.StreamResponse` and descendants like `aiohttp.web.Response`.
- **`identity`** (*`str`*) – `aiohttp.web.Request` object.
- **`kwargs`** – additional arguments passed to *`AbstractIdentityPolicy.remember()`*.

  They are policy-specific and may be used, e.g. for specifiying cookie lifetime.

`coroutine` `aiohttp_security.`**`forget`**`(`*request*, *response*`)`

Forget previously remembered *identity*.

The action is performed by registered *`AbstractIdentityPolicy.forget()`*.

Parameters

- **`request`** – `aiohttp.web.Request` object.
- **`response`** – `aiohttp.web.StreamResponse` and descendants like `aiohttp.web.Response`.

`coroutine` `aiohttp_security.`**`check_authorized`**`(`*request*`)`

Checker that doesn't pass if user is not authorized by *request*.

Parameters **`request`** – `aiohttp.web.Request` object.

Return str authorized user ID if success

Raise `aiohttp.web.HTTPUnauthorized` for anonymous users.

Usage:

```
async def handler(request):
    await check_authorized(request)
    # this line is never executed for anonymous users
```

coroutine `aiohttp_security.`**`check_permission`**`(`*request*, *permission*`)`

Checker that doesn't pass if user has no requested permission.

Parameters **`request`** – `aiohttp.web.Request` object.

Raise `aiohttp.web.HTTPUnauthorized` for anonymous users.

Raise `aiohttp.web.HTTPForbidden` if user is authorized but has no access rights.

Usage:

```
async def handler(request):
    await check_permission(request, 'read')
    # this line is never executed if a user has no read permission
```

coroutine `aiohttp_security.`**`authorized_userid`**`(`*request*`)`

Retrieve *userid*.

The user should be registered by *`remember()`* before the call.

Parameters **`request`** – `aiohttp.web.Request` object.

Returns `str` *userid* or `None` for session without signed in user.

coroutine `aiohttp_security.`**`permits`**`(`*request*, *permission*, *context=None*`)`

Check user's permission.

Return `True` if user remembered in *request* has specified *permission*.

Allowed permissions as well as *context* meaning are depends on *`AbstractAuthorizationPolicy`* implementation.

Actually it's a wrapper around *`AbstractAuthorizationPolicy.permits()`* coroutine.

The user should be registered by *`remember()`* before the call.

Parameters

- **`request`** – `aiohttp.web.Request` object.
- **`permission`** – Requested *permission*. `str` or `enum.Enum` object.
- **`context`** – additional object may be passed into `AbstractAuthorizationPolicy.permission()` coroutine.

Returns `True` if registered user has requested *permission*, `False` otherwise.

coroutine `aiohttp_security.`**`is_anonymous`**`(`*request*`)`

Checks if user is anonymous user.

Return `True` if user is not remembered in request, otherwise returns `False`.

Parameters **`request`** – `aiohttp.web.Request` object.

`@aiohttp_security.`**`login_required`**

Decorator for handlers that checks if user is authorized.

Raises `aiohttp.web.HTTPUnauthorized` if user is not authorized.

Deprecated since version 0.3: Use `check_authorized()` async function.

`@aiohttp_security.`**`has_permission`**`(`*permission*`)`

Decorator for handlers that checks if user is authorized and has correct permission.

Raises `aiohttp.web.HTTPUnauthorized` if user is not authorized.

Raises `aiohttp.web.HTTPForbidden` if user is authorized but has no access rights.

Parameters **`permission`** (`str`) – requested *permission*.

Deprecated since version 0.3: Use `check_authorized()` async function.

1.2.2 Abstract policies

***aiohttp_security* is built on top of two *abstract policies* –** `AbstractIdentityPolicy` and `AbstractAuthorizationPolicy`.

The first one responds on remembering, retrieving and forgetting *identity* into some session storage, e.g. HTTP cookie or authorization token.

The second is responsible to return persistent *userid* for session-wide *identity* and check user's permissions.

Most likely sofware developer reuses one of pre-implemented *identity policies* from *aiohttp_security* but build *authorization policy* from scratch for every application/project.

Identification policy

`class` `aiohttp_security.`**`AbstractIdentityPolicy`**

`coroutine identify``(`*request*`)`

Extract *identity* from *request*.

Abstract method, should be overriden by descendant.

Parameters **`request`** – `aiohttp.web.Request` object.

Returns the claimed identity of the user associated request or `None` if no identity can be found associated with the request.

`coroutine remember``(`*request*, *response*, *identity*, ***kwargs*`)`

Remember *identity*.

May use *request* for accessing required data and *response* for storing *identity* (e.g. updating HTTP response cookies).

kwargs may be used by concrete implementation for passing additional data.

Abstract method, should be overriden by descendant.

Parameters

- **`request`** – `aiohttp.web.Request` object.
- **`response`** – `aiohttp.web.StreamResponse` object or derivative.
- **`identity`** – *identity* to store.
- **`kwargs`** – optional additional arguments. An individual identity policy and its consumers can decide on the composition and meaning of the parameter.

coroutine forget(*request*, *response*)

Forget previously stored *identity*.

May use *request* for accessing required data and *response* for dropping *identity* (e.g. updating HTTP response cookies).

Abstract method, should be overriden by descendant.

Parameters

- **request** – `aiohttp.web.Request` object.
- **response** – `aiohttp.web.StreamResponse` object or derivative.

Authorization policy

class aiohttp_security.**AbstractAuthorizationPolicy**

coroutine authorized_userid(*identity*)

Retrieve authorized user id.

Abstract method, should be overriden by descendant.

Parameters **identity** – an *identity* used for authorization.

Returns the *userid* of the user identified by the *identity* or `None` if no user exists related to the identity.

coroutine permits(*identity*, *permission*, *context=None*)

Check user permissions.

Abstract method, should be overriden by descendant.

Parameters

- **identity** – an *identity* used for authorization.
- **permission** – requested permission. The type of parameter is not fixed and depends on implementation.

1.3 How to Make a Simple Server With Authorization

Simple example:

```
from aiohttp import web
from aiohttp_session import SimpleCookieStorage, session_middleware
from aiohttp_security import check_permission, \
    is_anonymous, remember, forget, \
    setup as setup_security, SessionIdentityPolicy
from aiohttp_security.abc import AbstractAuthorizationPolicy


# Demo authorization policy for only one user.
# User 'jack' has only 'listen' permission.
# For more complicated authorization policies see examples
# in the 'demo' directory.
class SimpleJack_AuthorizationPolicy(AbstractAuthorizationPolicy):
    async def authorized_userid(self, identity):
```

(continues on next page)

(continued from previous page)

```
        """Retrieve authorized user id.
        Return the user_id of the user identified by the identity
        or 'None' if no user exists related to the identity.
        """
        if identity == 'jack':
            return identity

    async def permits(self, identity, permission, context=None):
        """Check user permissions.
        Return True if the identity is allowed the permission
        in the current context, else return False.
        """
        return identity == 'jack' and permission in ('listen',)


async def handler_root(request):
    is_logged = not await is_anonymous(request)
    return web.Response(text='''<html><head></head><body>
            Hello, I'm Jack, I'm {logged} logged in.<br /><br />
            <a href="/login">Log me in</a><br />
            <a href="/logout">Log me out</a><br /><br />
            Check my permissions,
            when i'm logged in and logged out.<br />
            <a href="/listen">Can I listen?</a><br />
            <a href="/speak">Can I speak?</a><br />
        </body></html>'''.format(
            logged='' if is_logged else 'NOT',
        ), content_type='text/html')


async def handler_login_jack(request):
    redirect_response = web.HTTPFound('/')
    await remember(request, redirect_response, 'jack')
    raise redirect_response


async def handler_logout(request):
    redirect_response = web.HTTPFound('/')
    await forget(request, redirect_response)
    raise redirect_response


async def handler_listen(request):
    await check_permission(request, 'listen')
    return web.Response(body="I can listen!")


async def handler_speak(request):
    await check_permission(request, 'speak')
    return web.Response(body="I can speak!")


async def make_app():
    #
    # WARNING!!!
    # Never use SimpleCookieStorage on production!!!
    # It’s highly insecure!!!
```

(continues on next page)

(continued from previous page)

```
    #

    # make app
    middleware = session_middleware(SimpleCookieStorage())
    app = web.Application(middlewares=[middleware])

    # add the routes
    app.add_routes([
        web.get('/', handler_root),
        web.get('/login', handler_login_jack),
        web.get('/logout', handler_logout),
        web.get('/listen', handler_listen),
        web.get('/speak', handler_speak)])

    # set up policies
    policy = SessionIdentityPolicy()
    setup_security(app, policy, SimpleJack_AuthorizationPolicy())

    return app


if __name__ == '__main__':
    web.run_app(make_app(), port=9000)
```

1.4 Permissions with PostgreSQL-based storage

Make sure that you have PostgreSQL and Redis servers up and running. If you want the full source code in advance or for comparison, check out the demo source.

1.4.1 Database

Launch these sql scripts to init database and fill it with sample data:

```
psql template1 < demo/sql/init_db.sql
```

and

```
psql template1 < demo/sql/sample_data.sql
```

Now you have two tables:

- for storing users

users
id
login
passwd
is_superuser
disabled

- for storing their permissions

permissions
id
user_id
permission_name

1.4.2 Writing policies

You need to implement two entities: *IdentityPolicy* and *AuthorizationPolicy*. First one should have these methods: *identify*, *remember* and *forget*. For second one: *authorized_userid* and *permits*. We will use built-in *SessionIdentityPolicy* and write our own database-based authorization policy.

In our example we will lookup database by user login and if presents then return this identity:

```
async def authorized_userid(self, identity):
    async with self.dbengine as conn:
        where = sa.and_(db.users.c.login == identity,
                        sa.not_(db.users.c.disabled))
        query = db.users.count().where(where)
        ret = await conn.scalar(query)
        if ret:
            return identity
        else:
            return None
```

For permission checking we will fetch the user first, check if he is superuser (all permissions are allowed), otherwise check if permission is explicitly set for that user:

```
async def permits(self, identity, permission, context=None):
    if identity is None:
        return False

    async with self.dbengine as conn:
        where = sa.and_(db.users.c.login == identity,
                        sa.not_(db.users.c.disabled))
        query = db.users.select().where(where)
        ret = await conn.execute(query)
        user = await ret.fetchone()
        if user is not None:
            user_id = user[0]
            is_superuser = user[3]
            if is_superuser:
                return True

            where = db.permissions.c.user_id == user_id
            query = db.permissions.select().where(where)
            ret = await conn.execute(query)
            result = await ret.fetchall()
            if ret is not None:
                for record in result:
                    if record.perm_name == permission:
                        return True

        return False
```

1.4.3 Setup

Once we have all the code in place we can install it for our application:

```
from aiohttp_session.redis_storage import RedisStorage
from aiohttp_security import setup as setup_security
from aiohttp_security import SessionIdentityPolicy
from aiopg.sa import create_engine
from aioredis import create_pool

from .db_auth import DBAuthorizationPolicy


async def init(loop):
    redis_pool = await create_pool(('localhost', 6379))
    dbengine = await create_engine(user='aiohttp_security',
                                   password='aiohttp_security',
                                   database='aiohttp_security',
                                   host='127.0.0.1')
    app = web.Application()
    setup_session(app, RedisStorage(redis_pool))
    setup_security(app,
                   SessionIdentityPolicy(),
                   DBAuthorizationPolicy(dbengine))
    return app
```

Now we have authorization and can decorate every other view with access rights based on permissions. There are already implemented two helpers:

```
from aiohttp_security import check_authorized, check_permission
```

For each view you need to protect - just apply the decorator on it:

```
class Web:
    async def protected_page(self, request):
        await check_permission(request, 'protected')
        response = web.Response(body=b'You are on protected page')
        return response
```

or:

```
class Web:
    async def logout(self, request):
        await check_authorized(request)
        response = web.Response(body=b'You have been logged out')
        await forget(request, response)
        return response
```

If someone try to access that protected page he will see:

```
403: Forbidden
```

The best part of it - you can implement any logic you want until it follows the API conventions.

1.4.4 Launch application

For working with passwords there is a good library passlib. Once you've created some users you want to check their credentials on login. Similar function may do what you are trying to accomplish:

```
from passlib.hash import sha256_crypt

async def check_credentials(db_engine, username, password):
    async with  db_engine as conn:
        where = sa.and_(db.users.c.login == username,
                        sa.not_(db.users.c.disabled))
        query = db.users.select().where(where)
        ret = await conn.execute(query)
        user = await ret.fetchone()
        if user is not None:
            hash = user[2]
            return sha256_crypt.verify(password, hash)
    return False
```

Final step is to launch your application:

```
python demo/database_auth/main.py
```

Try to login with admin/moderator/user accounts (with **password** password) and access **/public** or **/protected** endpoints.

1.5 Glossary

aiohttp *asyncio* based library for making web servers.

asyncio The library for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives.

Reference implementation of **PEP 3156**

https://pypi.python.org/pypi/asyncio/

authentication Actions related to retrieving, storing and removing user's *identity*.

Authenticated user has no access rights, the system even has no knowledge is there the user still registered in DB.

If `Request` has an *identity* it means the user has some ID that should be checked by *authorization* policy.

authorization Checking actual permissions for identified user along with getting *userid*.

identity Session-wide `str` for identifying user.

Stored in local storage (client-side cookie or server-side storage).

Use `remember()` for saving *identity* (sign in) and `forget()` for dropping it (sign out).

identity is used for getting *userid* and *permission*.

permission Permission required for access to resource.

Permissions are just strings, and they have no required composition: you can name permissions whatever you like.

userid User's ID, most likely his *login* or *email*

CHAPTER 2

License

`aiohttp_security` is offered under the Apache 2 license.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index