pyramid_authsanity Documentation

Release 2.0.0

Bert JW Regeer

Mar 08, 2021

Contents

pyramid_authsanity is an authentication policy for the Pyramid Web Framework that strives to make it easier to write a secure authentication policy that follows web best practices.

- Uses tickets to allow sessions to be prematurely ended. Don't depend on the expiration of a cookie for example, instead have the ability to terminate sessions server side.
- Stops session fixation by automatically clearing the session upon login/logout. Sessions are also cleared if the new session is for a different userid than before.
- Automatically adds the Vary HTTP header if the authentication policy is used.

pyramid_authsanity uses Michael Merickel's absolutely fantastic pyramid_services to allow an application developer to easily plug in their own sources, and interact with their user database.

CHAPTER 1

API Documentation

Reference material for every public API exposed by pyramid_authsanity:

1.1 pyramid_authsanity

1.1.1 Authentication Service Policy

class pyramid_authsanity.**AuthServicePolicy**(*debug=False*)

authenticated_userid(*request*)
Returns the authenticated userid for this request.

effective_principals(*request*)
A list of effective principals derived from request.

forget(*request*)
A list of headers which will delete appropriate cookies.

remember(*request*, *principal*, ***kw*)
Returns a list of headers that are to be set from the source service.

unauthenticated_userid(*request*)
We do not allow the unauthenticated userid to be used.

1.2 pyramid_authsanity.interfaces

1.2.1 SourceService

interface pyramid_authsanity.interfaces.**IAuthSourceService**
Represents an authentication source.

vary
List of HTTP headers to Vary the response by.

get_value()
Returns the opaque value that was stored.

headers_remember(*value*)
Returns any and all headers for remembering the value, as a list. Value is a standard Python type that shall be serializable using JSON.

headers_forget()
Returns any and all headers for forgetting the current requests value.

1.2.2 AuthService

interface pyramid_authsanity.interfaces.**IAuthService**
Represents an authentication service. This service verifies that the users authentication ticket is valid and returns groups the user is a member of.

userid()
Return the current user id, None, or raise an error. Raising an error is used when no attempt to verify a ticket has been made yet and signifies that the authentication policy should attempt to call `verify_ticket`

groups()
Returns the groups for the current user, as a list. Including the current userid in this list is not required, as it will be implicitly added by the authentication policy.

verify_ticket(*principal*, *ticket*)
Verify that the principal matches the ticket given.

add_ticket(*principal*, *ticket*)
Add a new ticket for the principal. If there is a failure, due to a missing/non-existent principal, or failure to add ticket for principal, should raise an error

remove_ticket(*ticket*)
Remove a ticket for the current user. Upon success return True

1.3 pyramid_authsanity.sources

1.3.1 Session Authentication Source

pyramid_authsanity.sources.**SessionAuthSourceInitializer**(*value_key='sanity.'*)
An authentication source that uses the current session

1.3.2 Cookie Authentication Source

pyramid_authsanity.sources.**CookieAuthSourceInitializer**(*secret*, *cookie_name='auth'*, *secure=False*, *max_age=None*, *httponly=False*, *path='/'*, *domains=None*, *debug=False*, *hashalg='sha512'*)
An authentication source that uses a unique cookie.

1.3.3 Authorization Header Authentication Source

pyramid_authsanity.sources.**HeaderAuthSourceInitializer**(*secret*, *salt='sanity.header.'*)

An authentication source that uses the Authorization header.

CHAPTER 2

Narrative Documentation

Narrative documentation that describes how to use this library, with some examples.

2.1 The authentication policy

This authentication policy has two moving pieces, they work together to provide an easy to use authenitcation policy that provides more security by allowing the server to terminate an active authentication session.

2.1.1 Source Service

The first piece is called the authentication source service, this stores the principal and a ticket. There are two provided source services:

cookie

This is the default source and stores the information in a JSON encoded cookie that is signed using HMAC. This secures the information so long as the secret key for the HMAC is not made public.

session

This source stores the information required for the authentication in the Pyramid session, this requires that a session is available in the application as *request.session*. Since there is no requirement for a Pyramid application to have a registered session, pyramid_authsanity decided to not make this the default.

2.1.2 Authentication Service

The authentication service is defined by the user, the primary goal is to verify that the principal and ticket are both still valid.

CHAPTER 3

Other Matters

3.1 Frequently Asked Questions

3.1.1 Why tickets?

If you have a web application that uses a simple signed cookie that contains information about the signed in user, the login will not expire until the cookie's expiration. This can leave gaps in security.

Take the scenario of an employee that uses their own device for business, they log in in the morning before heading into the office and the cookie is set to authenticate them for 12 hours. They go buy some coffee and put their phone down. Walking out they leave the phone on a table. Once they find out they notify the company about the lost phone, however since the authentication cookie is their username, there is no way to terminate the existing session, and were an attacker able to use their phone they would be able to continue using the web application for the next 12 hours.

Tickets are stored server side, and for each device/login there will be a unique ticket. These can be individually removed, and as soon as it is removed the authentication is no longer valid.

Facebook/Google for example also allow the user to view their sessions, and terminate one, or all of them. This ticket based system allows for the same user interaction, thereby allowing more control over who is logged in or why.

If a user changes their password, tickets give the ability to log out all pre-existing sessions so that the user is required to login again on any and all devices.

3.1.2 What is session fixation?

Session fixation is an attack that permits an attacker to hijack a valid session. Generally this is done by going to the website and retrieving an session, that session is then given to the victim. As soon as the victim logs in, the attacker who still has the same session token is able to see what is being stored in the session which may potentially leak data. For authentication policies that store the authentication in the session this would give the attacker full control over the victim's account.

You stop session fixation by dropping the session when going across an authentication boundary (login/logout). This will recreate the session from scratch, which leaves the attacker with a session that is worthless.

3.1.3 Vary headers

When an HTTP request is made, the content is usually cached for as long as possible to avoid having to do more trips to the backend server (for reverse proxies) and more requests to the server for browsers. However proxies and browsers can't know that the page for example contains information that is dependent on a particular HTTP header, that is where the `vary` HTTP header comes in.

Using `vary` you can tell the proxies or web browser that this page is to be cached, but it is dependent on a particular header. For example `vary: cookie` means the cache is allowed to return the page without requesting information from the backend server so long the cookie the client sends is the exact same as at the time of the previous response generated.

For more take a look at RFC7231 section 7.1.4 which explains what this header does and means.

3.2 License

Python Module Index

Index