RFP-165 Authorization

Draft

9 Pages

Abstract

This RFP requests a service that can be used to provide the information to authorize actions in any service implementation code based on a user identity. The authorizations must be usable in a type safe way in Java as well as useable in other languages like Javascript so that the user interface can adapt to the possible authorizations. This service must be aligned with User Admin.

0 Document Information

0.1 Table of Contents

0 Document Information........ 2

0.1 Table of Contents........ 2

0.2 Terminology and Document Conventions........ 2

0.3 Revision History........ 3

1 Introduction........ 3

2 Application Domain........ 3

2.1.1 Basic Authentication........ 4

2.1.2 Kerberos........ 4

2.1.3 Mozilla Persona........ 4

2.1.4 OAuth 2.0........ 5

2.1.5 JAAS & Login Module........ 6

2.1.6 SASL........ 7

2.1.7 Signed Requests........ 7

2.2 Terminology + Abbreviations........ 8

3 Problem Description........ 8

4 Use Cases........ 9

4.1.1 Persona........ 9

4.1.2 Signed Requests........ 10

5 Requirements........ 11

5.1.1 General........ 11

6 Document Support........ 11

6.1 References........ 11

6.2 Author's Address........ 12

6.3 End of Document........ 12

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	*25-11-14*	*Initial* *Peter.Kriens@aQute.biz*

1 Introduction

This RFP originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

2 Application Domain

Authorization is the process of requesting permission to an *authority* to execute an *action*, which generally means executing some code with a number of parameters that can influence the decision of the authority to grant permission. The authority maintains a current *execution context* usually associated with a *user*, either a human being or another program. The execution context maintains a set of *permissions*. A permission contains an assertion on the action and its parameters that allow the action to be executed in its execution context.

In general, authentication takes place during a *request*. A *request processor* then executes this request in a execution context that is established by the authority. An example of a request processors is a servlet. In the servlet model, authentication is often done with a Servlet Filter.

Since a program can have thousands of actions creating permissions for each of these actions creates large configurations. It is therefore standard in Java that permissions are grouped in a *role* and a user then is associated with several roles, minimizing the amount of needed configuration. It is popular in Java security libraries to shortcut the grouping and to let the code directly check the role name instead, requiring code changes if the action-role mapping has to be changed.

2.1.1 User Admin

The OSGi developed User Admin as one of the earliest specification. It provides a service to store user information (both open profile properties as well as secret credentials), as well as an authorization model, all based on a user name. User Admin has users and named groups, both called roles. Roles can be assigned to groups, providing for a recursive model. Group names can be used as action names and User Admin can provide an Authorization object for a given user that can act as the authority.

2.1.2 Java Security Model

In Java, and therefore OSGi, the security model is based around Permission classes. When a method wants to check if it is allowed to execute an action, it creates and instance of a Permission subclass that represents the requested action. The constructor can be used to pass parameters where the general format is a resource name and one or more verbs. For example `new FilePermission("/tmp", "read")`. This permission is then verified using the Security Manager. In the standard setup each code base (class loader, JAR, directory) can have its own Protection Domain. Each domain can be associated with permissions grouped by their actual type. The Security Manager requests the current Access Control Context to verify that all methods on the stack come from a Protection Domain that implies the requested permission. The semantic of what implies means is left to the actual Permission subtype; it is fully legal to allow certain actions only when it is full moon. That is, the results of the implies cannot be cached nor inferred in any other way. Methods can block the stack crawl by executing in a privileged block so that they can increase their privilege above any caller.

JAAS added the capability to add a user based context to the stack crawling. This way, the checks used for code based checking would automatically check the user permissions by requiring their conjunction.

2.1.3 Spring

Spring added the highly popular Spring security subsystem [4]. It provides comprehensive support in one large library that supports many popular standards and integrates heavily with the servlet specifications. It includes filters to protect against external attacks as well as authentication and authorization facilities. It relies heavily on the Spring XML to initialize and configure an application. A very popular aspect of Spring security is the use of annotations on methods to minimize the code that needs to check for permissions. Spring places interceptors before the actual code to process these annotations and do the proper checking. Spring heavily relies on statics and singletons.

2.1.4 Apache Shiro

Apache Shiro [3]. is a relative newcomer and provides a comprehensive library for any security need for web applications. It is very flexible and provides solutions for authentication, authorization, cryptography, web integration, etc. It has several models for authorization including Java Permissions, in code, annotations, etc. It supports

2.2 Terminology + Abbreviations

3 Problem Description

There are a number of problems with the current popular Java solutions:

- DTOs. It is impossible to provide another process the permissions so that it can adapt to the possible actions. For example, show/hide buttons that can/cannot be used. Most Java security solutions heavily rely on generating the user interface code in the server while the current trend is to execute the GUI in the browser since this scales better and provides a much better user experience.
- Annotations. Annotation based security checking has a number of drawbacks:
    - Annotations can by their nature not check any runtime parameters
    - Annotations require an interceptor. Interceptors muddle the concept of this since 'this' calls to other methods on the same class will bypass the interceptor.
    - Annotations are magic, the check code cannot be stepped through in a debugger.
    - It is impossible to react based on the absence or presence of authorization, in many cases an alternative path of an unauthorized user is available. In certain cases the default response of throwing a security exception might not be applicable since this leaks the existing of the function.
    - Security annotations today mostly use role based checks where permission based checks are better.
- Uncohesive libraries. Both Apache Shiro and Spring security are one stop solutions. Not only do they drag in a lot of unwanted code, they also bring their own configuration and dependency injection frameworks that often require singletons and statics. In general, the API of these libraries is coupling too many unrelated things.
- Overly flexible. Existing solutions cater to support any security model under the sun. This is ok for monolithic applications since it allows software architects to model the system exactly as they want. However, this does not work very well in a component based system since security is a horizontal aspect of the system; most components will have a security part. Different models for these kind of horizontal services tend to split the available components. For OSGi, we must have a single authorization model so that components can verify the execution of their actions without having to work with many different models.
- Not service based. None of the libraries has a proper service model. (Although all libraries can be used to implement a proper authorization service.)
- Type safety. Though Java security models permissions as types, other libraries tend to become string based. E.g. the annotations uses a string for the role name. Though this is not always inevitable, the life

of a programmer can be significantly improved if the permissions are expressed as Java names, for example method names. A method like `security.widget_modify(String name, int value)` contains all information necessary to check the authorization for the given action `('widget.modify')` and the parameters `('xyz', '3')`. The advantage of expressing this as a Java name are:

- IDE navigation. That is, it is trivial to find where a given permission is checked.
- No typos or missing definitions found in runtime
- Refactoring
- Typed parameters

This RFP searches an OSGi service specification so that any component can find out if a component defined action can be executed in the current execution context.

4 Use Cases

4.1.1 The Cloth Company

The Cloth Company has a large distributed application for financial applications. The management application that controls the cluster of computers can control the operations in detail but is also used by departments to install applications in the cluster. Since the Cloth company has been successfully around since the gilded age, they have thousands of employees. They have an Active Directory server farm where they maintain the authorizations of all employees. It is a company rule that any computer system in the company consults this directory, authenticates the user, and is given a set of allowed role names.

The Management application is based on OSGi and a single page web app. When starting and not logged in, the user is requested for a user id and password using basic authorization (the server returns a NOT AUTHORIZED return code when there is no user set in the session). On the server side, a Servlet Filter consults an RFP 164 Authenticator.

This authenticator consults the Active Directory and successfully authenticates the user. It therefore receives a set of role names for the trusted user id: DEVOP and APP_FOO and APP_BAR. The authenticator has a local configuration that allows him to translate the role names to permissions:

- machine.edit *
- machine.view *
- application.view *
- application.edit “foo”

- application.edit “bar”
- manage

The authenticator now stores collaborates with the local authority to set the permissions for the given user id. The authenticator then returns the trusted user id to the servlet filter. The filter then sets the user id in the session.

On the next request the user wants to install an application “Xyz”. In the security filter, the user id is detected in the session and the downstream filters/servlets are now executed in a proper execution context.

Downstream, there is a service that can install an application. When this service is called, it verifies with the local authority if the current execution context allows the installation of application “Xyz”. Since this is not allowed, a security exception is thrown and the user will get a FORBIDDEN result.

Disappointed, the user goes somewhere else on the net. The security filter detects the timeout of the session and tells the authenticator that authenticated the user that the user id is no longer used. The authenticator then withdraws the permissions for that user from the authority.

4.1.2 OSGi enRoute

The OSGi enRoute project has developed an Authentication API [5]. that should follow most of the requirements and ideas in this RFP.

5 Requirements

5.1.1 General

- G0010 – Provide an OSGi authorization service that can establish an execution context for a user id that can be consulted by any component to check if an action is authorized.
- G0020 – Request processors must be able to establish an execution context associated with a user id for a call.
- G0030 – It must be possible to provide a set of permissions to a single execution context
- G0040 – It must be possible to have an anonymous user
- G0050 – The execution context must be accessible anywhere in a call stack without having to pass it as a parameter in all the intermediate calls.
- G0060 – It must be possible to have multiple execution contexts for the same user simultaneously.
- G0070 – Any method must be able to verify if an action is authorized in the execution context by consulting the authority.

- G0080 – It must be possible to define the permissions in an interface definition that the authority service user can use in runtime to check the permission.
- G0090 – A permission check must be able to take parameters
- G0100 – Parameters may be any type but the check can be limited to their toString value.
- G0110 – Parameter checks must at least support globbing
- G0120 – It must be possible to get the user id of the current execution context.
- G0130 – It must be possible to get the permissions for an execution context in a DTO form
- G0140 – The permission DTOs must provide sufficient information for a pure Javascript library to verify the assertions against a set of parameters. The rationale is that the browser can adapt the GUI based on the execution context.

5.1.2 Management

- M0010 – The service must provide a way to manage the active permissions for a user id (create/read/update/delete).
- M0020 – It must be possible to list the user ids available to the authority with a filter (to support large number of users).
- M0030 – Any updates to the permissions for a user id must be visible to future checks of the execution context of that user id.
- M0040 – It must be possible to get events of the management operations in real time

5.1.3 Non Functional

- N0010 – The API must be able to support 1 billion users. The rationale is that no API call assumes it can return all users for example.
- N0020 – The API must leverage Java 8 (stream API, lambda) if possible
- N0030 – The solution must be self contained, it must not require other services or any of their types.
- N0040 – Implementations must be able to leverage the authorization functions in Apache Shiro and/or Spring security.
- N0050 – It must be possible to create implementations that work over large clusters.
- N0060 – It must be possible to use User Admin to store the permissions using the User Admin group model so role expansion is supported.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. http://shiro.apache.org/
[4]. http://projects.spring.io/spring-security/
[5]. http://enroute.osgi.org/services/osgi.enroute.authorization.api.html

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	\+33467542167
e-mail	Peter.kriens@aQute.biz

6.3 End of Document