Java Enterprise Edition Security Explained

MUNI Brno, May 9, 2019

Peter Škopek, pskopek@redhat.com, twitter: @pskopek

Abstract

This lecture will guide you through various aspects of security in Java Enterprise Edition Applications.
It will start with a bit of history of JAAS and continue with Java EE security concepts and explanation of their usage in your application.
Next comes Keycloak introduction and ways to secure applications running in WildFly 14.

Java Authentication and Authorization Service

History

> The Java Authentication and Authorization Service (JAAS) was introduced as an optional package (extension) to the Java 2 SDK, Standard Edition (J2SDK), v 1.3. JAAS was integrated into the J2SDK 1.4.

What is that?

> API for authentication and authorization services in Java application.
> Hopefully become history soon.

JAAS

JAAS implements a Java version of the standard Pluggable Authentication Module (**PAM**) framework as we know it from Linux/UNIX environment.

This permits applications to remain independent from underlying authentication technologies.

New or updated authentication technologies can be plugged under an application without requiring modifications to the application itself.

JAAS

Applications enable the authentication process by instantiating a **LoginContext** object, which in turn references a Configuration to determine the authentication technology(ies), or **LoginModule**(s), to be used in performing the authentication.

Typical LoginModules may prompt for and verify a username and password. Others may read and verify a voice or fingerprint sample.

Common JAAS interfaces

Principals

- Must implement java.security.Principal interface.

Credentials

- Public and private credential classes are not part of the core JAAS class library.
- Any class can represent a credential.
- Developers, however, may elect to have their credential classes implement two interfaces related to credentials: javax.security.auth.**Refreshable** and javax.security.auth.**Destroyable**

Java EE Security - Overview

What are the aspects of secure applications?

Authentication - The means by which communicating entities (for example, client and server) prove to one another that they are acting on behalf of specific identities that are authorized for access.

Access control for resources - The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

Java EE Security - Overview

Data integrity - The means used to prove that information has not been modified by a third party (some entity other than the source of the information).
For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent.

Java EE Security - Overview

Another security aspects of good application

Confidentiality - The means used to ensure that information is made available only to users who are authorized to access it.

Non-repudiation - The means used to prove that a user performed some action such that the user cannot reasonably deny having done so.

Auditing - The means used to capture a tamper-resistant record of security related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms.

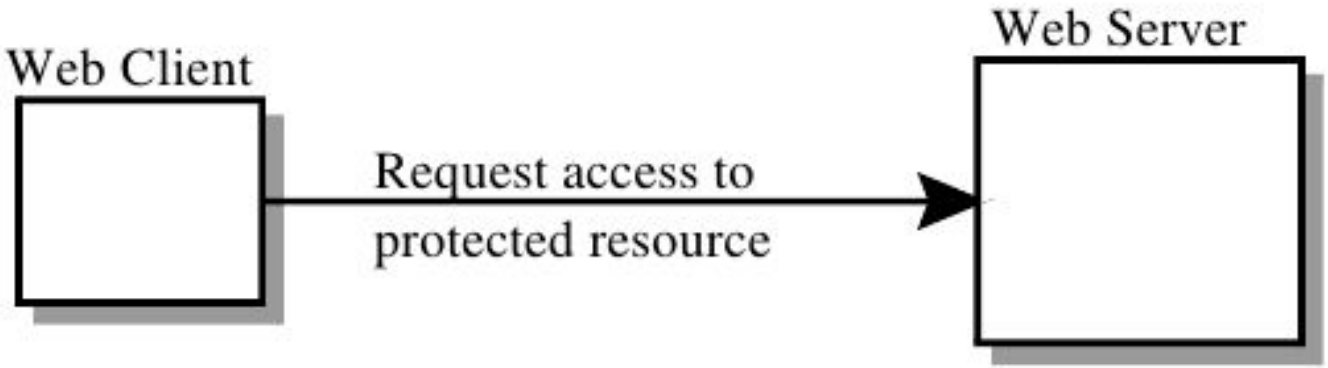
Java EE Security - HTTP Auth Mechanisms

There is more than one authentication mechanism

- Basic
- Digest
- Certificate
- SPNEGO/Kerberos
- JWT

Simple Java EE example

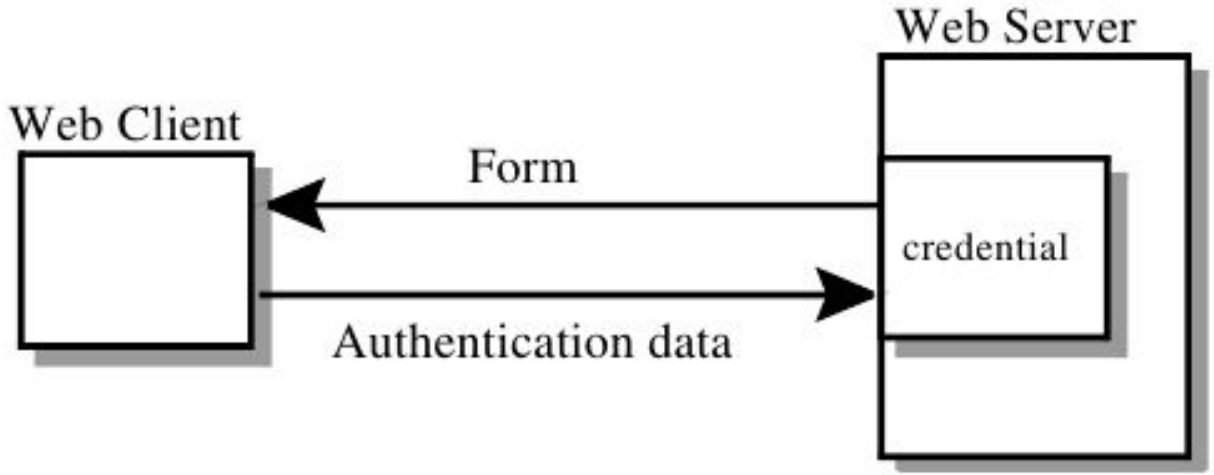
Step 1: Initial request



The web client requests the main application URL

Simple Java EE example

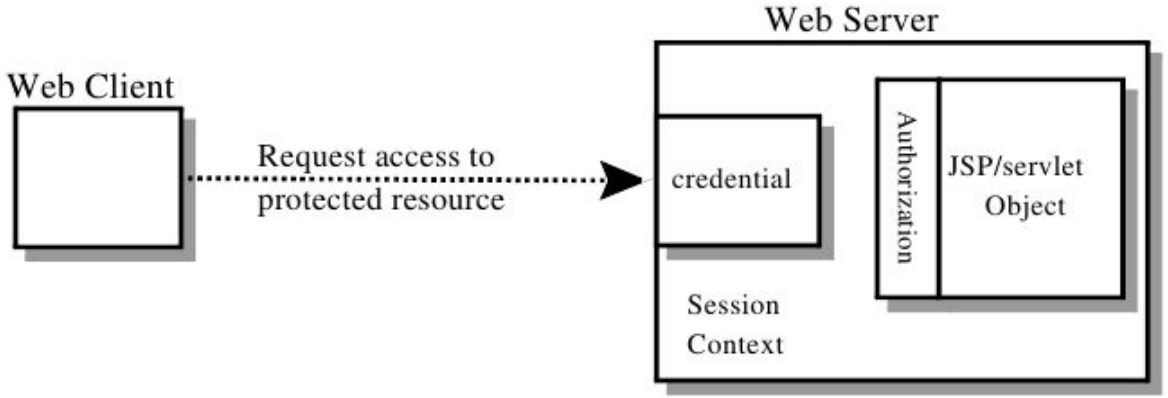
Step 2: Initial Authentication



The web server returns a form that the web client uses to collect authentication data (for example, username and password) from the user. The web client forwards the authentication data to the web server, where it is validated by the web server.

Simple Java EE example

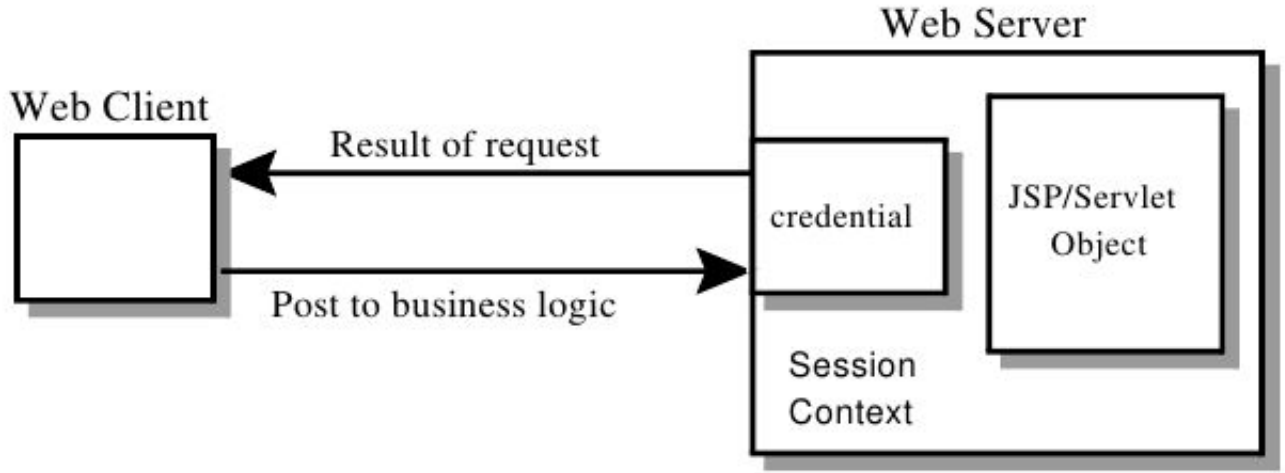
Step 3: URL Authorization



The web container then tests the user's credential against each role to determine if it can map the user to the role.

Simple Java EE example

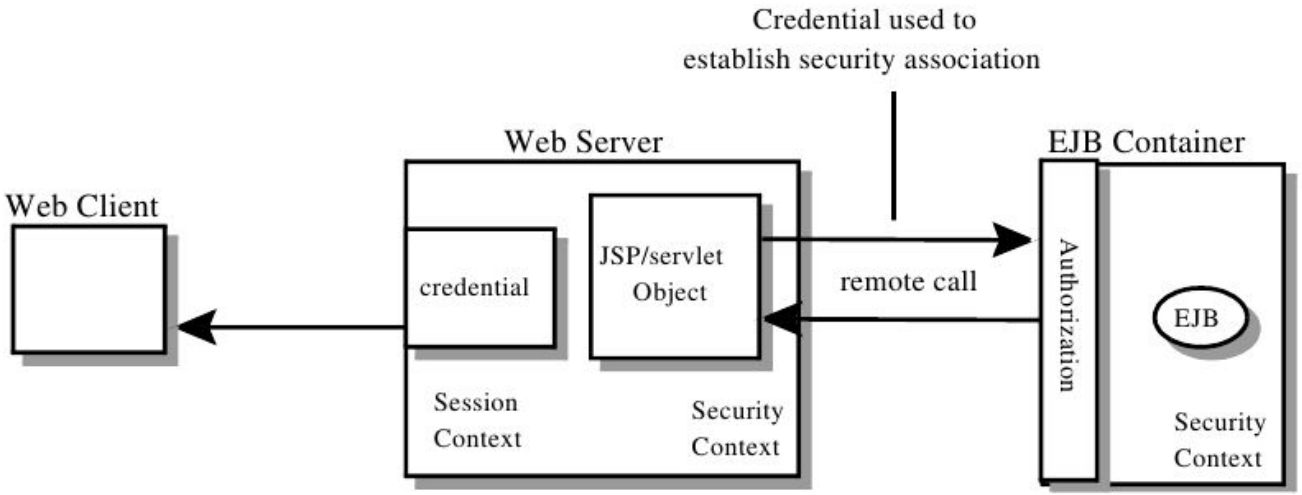
Step 4: Fulfilling the Original Request



If the user is authorized, the web server returns the result of the original URL request.

Simple Java EE example

Step 5: Invoking Enterprise Bean Business Methods



The servlet performs the remote method call to the enterprise bean, using the user's credential to establish a secure association between the servlet and the enterprise bean. The association is implemented as two related security contexts, one in the web server and one in the EJB container.

Goals of Java EE Security Architecture

Transparency: Application Component Providers should not have to know anything about security to write an application.

Isolation: Divorcing the application from responsibility for security ensures greater portability of Java EE applications.

Flexibility: The security mechanisms and declarations used by applications under this specification should not impose a particular security policy, but facilitate the implementation of security policies specific to the particular Java EE installation or application.

Abstraction: An application component's security requirements will be logically specified using deployment descriptors.

Goals of Java EE Security Architecture

Independence: Required security behaviors and deployment contracts should be implementable using a variety of popular security technologies.

Secure interoperability: Application components executing in a Java EE product must be able to invoke services provided in a Java EE product from a different vendor.

Terminology

Principal - is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise.

Security Policy Domain - is a scope over which a common security policy is defined and enforced by the security administrator of the security service (also known as security domain or realm).

Security Attributes - a set of security attributes is associated with every principal.

Credential - contains or references information (security attributes) used to authenticate a principal for Java EE product services.

Container Based Security

Security for components is provided by their containers in order to achieve the goals for security specified above in a Java EE environment. A container provides two kinds of security:

1. Declarative security

   Declarative security refers to the means of expressing an application's security model or requirements, including roles, access control, and authentication requirements in a form external to the application. The deployment descriptor is the primary vehicle for declarative security in web applications.

2. Programmatic security

   Programmatic security is used by security aware applications when declarative security alone is not sufficient to express the security model of the application.

Declarative Security

Following annotations are part of Servlet 3.0 specification and provide alternative to defining access control via declarative deployment descriptor.

- **@ServletSecurity** - Security constraints to be enforced by a Servlet container on HTTP protocol messages
- **@HttpConstraint** - The annotation is used within the @ServletSecurity annotation to represent the security constraint to be applied to all HTTP protocol methods for which a corresponding @HttpMethodConstraint does NOT occur within the @ServletSecurity annotation.
- **@HttpMethodConstraint** - The annotation is used within the @ServletSecurity annotation to represent security constraints on specific HTTP protocol messages/verbs.

Servlet Security Annotation Reference

Detailed descriptions could be found at:

http://jcp.org/en/jsr/detail?id=315

https://javaee.github.io/javaee-spec/javadocs/javax/servlet/annotation/package-summary.html

Example

For all HTTP methods, no constraints

```
@ServletSecurity
public class Example1 extends HttpServlet {
   ...
}
```

Example

For all HTTP methods, no auth-constraint, confidential transport required

```
@ServletSecurity(@HttpConstraint(transportGuarantee =
          TransportGuarantee.CONFIDENTIAL))
public class Example2 extends HttpServlet {
  ...
}
```

Example

For all HTTP methods, all access denied

```
@ServletSecurity(@HttpConstraint(EmptyRoleSemantic.DENY))
public class Example3 extends HttpServlet {
   ...
}
```

Example

For all HTTP methods, auth-constraint requiring membership in Role R1

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))
public class Example4 extends HttpServlet {
   ...
}
```

Example

For All HTTP methods except GET and POST, no constraints; for methods GET and POST, auth-constraint requiring membership in Role R1; for POST, confidential transport required

```
@ServletSecurity((httpMethodConstraints = {
 @HttpMethodConstraint(value = "GET", rolesAllowed = "R1"),
 @HttpMethodConstraint(value = "POST", rolesAllowed = "R1",
    transportGuarantee = TransportGuarantee.CONFIDENTIAL)
})
public class Example5 extends HttpServlet {
  ...
}
```

Example

For all HTTP methods except GET auth-constraint requiring membership in Role R1; for GET, no constraints

```
@ServletSecurity(
   value = @HttpConstraint(rolesAllowed = "R1"),
   httpMethodConstraints = @HttpMethodConstraint("GET"))
public class Example6 extends HttpServlet {
  ...
}
```

Example

For all HTTP methods except TRACE, auth-constraint requiring membership in Role R1; for TRACE, all access denied

```
@ServletSecurity(
 value = @HttpConstraint(rolesAllowed = "R1"),
 httpMethodConstraints = @HttpMethodConstraint(
      value="TRACE",
      emptyRoleSemantic = EmptyRoleSemantic.DENY))
public class Example7 extends HttpServlet {
  ...
}
```

Mapping to web.xml - example

Mapping @ServletSecurity with no contained @HttpMethodConstraint

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))
```

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>...</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <security-role-name>R1</security-role-name>
   </auth-constraint>
</security-constraint>
```

Mapping to web.xml - example

Mapping @ServletSecurity with contained @HttpMethodConstraint

```
@ServletSecurity(value=@HttpConstraint(rolesAllowed = "Role1"),
   httpMethodConstraints = @HttpMethodConstraint(value = "TRACE",
   emptyRoleSemantic = EmptyRoleSemantic.DENY))
```

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>...</url-pattern>
      <http-method-omission>TRACE</http-method-omission>
   </web-resource-collection>
   <auth-constraint>
      <security-role-name>Role1</security-role-name>
   </auth-constraint>
</security-constraint>
<security-constraint>
   <web-resource-collection>
      <url-pattern>...</url-pattern>
      <http-method>TRACE</http-method>
   </web-resource-collection>
   <auth-constraint/>
</security-constraint>
```

Roles

A servlet container enforces declarative or programmatic security for the principal associated with an incoming request based on the security attributes of the principal.

- A deployer has mapped a security role to a **user group** in the operational environment.
- A deployer has mapped a security role to a **principal name** in a security policy domain.

Programmatic Security

Programmatic security consists of the following methods of the **HttpServletRequest** interface:

- authenticate
- login
- logout
- getRemoteUser
- isUserInRole
- getUserPrincipal

EJB Container Security - Basic idea of EJB security

- Business methods of Enterprise Java Beans contain no security-related logic.
- Security policies for the application can be configured in a way that is most appropriate for the operational environment of the enterprise.
- A **security role** is a semantic grouping of permissions that a given type of users of the application must have in order to successfully use the application.

Security Permission Specification

The Bean Provider can use metadata **annotations** or the **deployment descriptor** to specify whether the caller's security identity or a run-as security identity should be used for the execution of the bean's methods.

- By default, the caller **principal** will be propagated as the **caller identity**. The Bean Provider can use the **RunAs** annotation to specify that a security principal that has been assigned to a specified security role be used instead.
- If the **deployment descriptor** is used to specify the security principal, the Bean Provider or the Application Assembler can use the security-identity deployment descriptor element to specify or override the security identity.

Programmatic Access to Caller's Security Context

The javax.ejb.EJBContext interface provides two methods (plus two deprecated methods that were defined in EJB 1.0) that allow the Bean Provider to access security information about the enterprise bean's caller.

```
public class MyBusinessBean {

   @Resource
   EJBContext ctx;

   ...
}
```

Security Related Annotations

Annotation	Corresponding DD Element
@DeclareRoles	security-role
@RolesAllowed	method-permission
@PermitAll	unchecked
@DenyAll	exclude-list
@RunAs	security-identity run-as

Security Related Annotations

Annotation	Class	Method
@DeclareRoles	Yes	No
@RolesAllowed	Yes	Yes
@PermitAll	Yes	Yes
@DenyAll	Yes	Yes
@RunAs	Yes	Yes

Java EE and Java Security Manager

Java EE application components are able to run with Java Security Manager.

Permission declarations must be stored in **META-INF/permissions.xml** file within an EJB, web, application client, or resource adapter archive in order for them to be located and subsequently processed by the deployment machinery of the Java EE Application server.

Permissions Allowed in Web, EJB, and Resource Adapter Components

java.lang.RuntimePermission loadLibrary.*

java.lang.RuntimePermission queuePrintJob

java.net.SocketPermission * connect

java.io.FilePermission * read,write

java.io.FilePermission file:${javax.servlet.context.temdir} read, write

java.util.PropertyPermission

Keycloak Features

- Single-Sign On
- Kerberos bridge
- Identity Brokering and Social Login
- User Federation
- Client Adapters
- Admin Console
- Account Management Console
- Standard Protocols
  - OpenID Connect, OAuth 2.0, and SAML
- Authorization Services

Introduction to Keycloak

What is Keycloak?

> Keycloak is an open source **Identity and Access Management** solution aimed at modern applications and services. It makes it easy to secure applications and services with little to no code.

Where to get it?

> http://www.keycloak.org/

Keycloak

What to download?

For the sake of simplicity we will use separate Keycloak server known as “Standalone server distribution”.

For the application server we need to download WildFly Client Adapter for appropriate server version and intended application. (WildFly 13 client adapter is the most recent one, but we will use one from snapshot builds. This is at the moment only way to get support for WildFly 14)

Keycloak Setup

1. Unzip keycloak distribution package
2. Run ./bin/standalone.sh -Djboss.socket.binding.port-offset=100
   This will start instance of keycloak server at port 8180 (= 8080 + 100)
3. Open http://localhost:8180/ to finish initial setup by creating admin user
4. Open Keycloak Admin Console at http://localhost:8180/auth/
5. Create new realm called “CTU”
6. Then create couple of test users (test1, test2)
   a. Go to “Credentials” tab and enter new password and password confirmation
   b. Switch off temporary password flag
   c. Press <Reset Password> button

Keycloak Client Adapter

Install Keycloak Client Adapter for WildFly 14 as we are using this application server.

1. Download and install WildFly (you should have one already installed)
2. Download Keycloak Client Adapter and unzip this file into the root directory of your WildFly distribution (WF14 Adapter)
3. Finish installation with:

   ```
   cd bin
   ./jboss-cli.sh --file=adapter-install-offline.cli
   ```
4. Previous command will do necessary changes to WildFly config file and we can start the server using ./bin/standalone.sh

Securing Java EE application with Keycloak

Application needs to be registered first with “CTU” Keycloak realm we have created earlier.

Follow the steps to do it.

1. Use Keycloak Admin Console open CTU realm
2. Select “Clients” and press <Create> button at the right side.
3. Register your client using this data

    Client ID: app-profile-vanilla

    Client Protocol: openid-connect

    Root URL: http://localhost:8080/vanilla

Securing Java EE application with Keycloak

To make it simple we will use keycloak-quickstarts namely “app-profile-jee-vanilla” app. Edit src/main/webapp/WEB-INF/web.xml to use authentication method KEYCLOAK instead of BASIC.

```
mvn clean wildfly:deploy -DskipTests
```

Login to application http://localhost:8080/vanilla/profile.jsp (not working)

Login is not working properly, one needs to modify Keycloak subsystem to recognize the client application we have just deployed.

Securing Java EE application with Keycloak

Get the installation template from installation tab in client record in Keycloak Console.

It looks something like this:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
    <secure-deployment name="WAR MODULE NAME.war">
        <realm>CTU</realm>
        <auth-server-url>http://localhost:8180/auth</auth-server-url>
        <public-client>true</public-client>
        <ssl-required>EXTERNAL</ssl-required>
        <resource>vanilla</resource>
    </secure-deployment>
</subsystem>
```

Change war name to vanilla.war

Update standalone.xml in your wildfly instance under section