JACC Guide

Guide to JACC Providers

JACC (Java Authorization Contract for Containers) is a spec that allows plugging in providers to handle the security decisions for a j2ee app server. You can have simple self-contained systems all the way to rule engines or delegating to external security servers. This is an outline of what a JACC provider needs to implement, what deployment time info is provided, and what run time info is provided, and how to get Geronimo to use your JACC provider, including how to get geronimo to supply extra info to your provider.

Before we get started on the provider, notice that there are a bunch of new Permission classes for web and ejb that basically encapsulate the permissions you can specify in the spec deployment descriptors in classes. Geronimo implements these, and the Geronimo provider doesn't need to know anything specific about them, but you still might want to review them.

So far this is a brief first draft, so there are probably lots of errors and omissions.

Classes a Provider needs to implement.

JACC splits up the security decision making apparatus into bits per policyContextID, which basically corresponds to web or ejb module. For each policyContextID there's a PolicyConfiguration that makes the decisions for that module. The PolicyConfigurationFactory provides access to the PolicyConfigurations, both for configuration purposes and in geronimo for runtime decision delegation.

PolicyConfigurationFactory

This supplies the PolicyConfigurations and to some extent manages their lifecycles. You specify your implementation class with the system property javax.security.jacc.PolicyConfigurationFactory.provider and get your implementation with the static getPolicyConfigurationFactory method. The getPolicyConfiguration(String policyContextId, boolean replace) method gives you the PolicyConfiguration for the policyContextId, where the replace parameter indicates if you want all the content erased before you get it (so you can configure it). You can tell if a policyContextId is in available for runtime use with the inService(String policyContextId) method.

PolicyConfiguration

This provides a lot of methods for pushing the role to permissions mappings specified in the spec deployment descriptors into the PolicyConfigurations. In geronimo, this is done when the application is being started. There's also a link method that must be called by the container (geronimo) for every pair of modules in a j2ee application. The standard geronimo jacc implementation doesn't need to do anything to link, but other implementations might.

Policy

You also have to supply a Policy implementation that somehow uses the info supplied to the PolicyConfigurations to help with the security decisions. Geronimo extends PolicyConfiguration to GeronimoPolicyConfiguration and supplies a GeronimoPolicy that delegates the

public boolean implies(ProtectionDomain domain, Permission permission)

decision to the appropriate PolicyConfiguration. If you want to write a geronimo-specific JACC provider (or one that drags in bits of geronimo to work) you can resuse GeronimoPolicy and have your PolicyConfiguration implement GeronimoPolicyConfiguration, or you can copy GeronimoPolicy, or you can come up with an entirely different scheme.

Configuration time access

Before the app is started up and "commit" called on each PolicyConfiguration for the app, the container (geronimo) has to get the PolicyConfiguration for each policyContextId and stuff in the role-permission mapping defined in the spec deployment descriptors, using the methods

void addToRole(java.lang.String string, java.security.PermissionCollection permissionCollection) throws javax.security.jacc.PolicyContextException;

void addToRole(java.lang.String string, java.security.Permission permission) throws javax.security.jacc.PolicyContextException;

void addToUncheckedPolicy(java.security.PermissionCollection permissionCollection) throws javax.security.jacc.PolicyContextException;

void addToUncheckedPolicy(java.security.Permission permission) throws javax.security.jacc.PolicyContextException;

void addToExcludedPolicy(java.security.PermissionCollection permissionCollection) throws javax.security.jacc.PolicyContextException;

void addToExcludedPolicy(java.security.Permission permission) throws javax.security.jacc.PolicyContextException;

void removeRole(java.lang.String string) throws javax.security.jacc.PolicyContextException;

void removeUncheckedPolicy() throws javax.security.jacc.PolicyContextException;

void removeExcludedPolicy() throws javax.security.jacc.PolicyContextException;

(I don't know why there are 2 versions of each add method). After the app is deployed the spec seems to indicate that you can get a PolicyConfiguration from the PolicyConfigurationFactory (thus taking it out of service and blocking access to the app) and reconfigure these permissions, but it isn't clear to me from the spec when this could be called and what you can actually do with it, since the spec appears to require that exactly what is in the spec dd is fed into the PolicyConfigurations. Theoretically one could write a generic admin interface to change the role-permission mappings at runtime, but then they would get reset when the app is restarted (at least in geronimo) unless the dd was changed identically.

Geronimo also provides a way to stuff more info into your JACC implementation as the app starts up. The Geronimo JACC provider uses this to install an app-wide principal-role mapping specified somewhere in the geronimo application plan: this is combined with the spec role-permission mapping to provide a principal-permission mapping actually used for "implies" calculations.

To implement such a JACC-provider specific info-stuffer, you need to define a schema for the info you want to add, and register a NamespaceDrivenBuilder GBean for that namespace and change the references to SecurityBuilder so they point to your security builder. Typically your builder should install a gbean in the application configuration that holds the extra info and supplies it to your JACC provider (using a proprietary interface) when it starts. Look at the geronimo implementation for more details:

builder: GeronimoSecurityBuilderImpl

runtime gbean holding the extra info: ApplicationPolicyConfigurationManager

If you don't have any such extra info, you can leave these out.

TODO: need to discuss extra methods in SecurityBuilder. These don't belong there, but for now you can probably continue to use the built-in GeronimoSecurityBuilderImpl or copy it. Basically we should be getting default Subjects from logging in to the app's login configuration, not constructing them by hand.

Runtime permissions decisions

So now your JACC provider is installed and configured and your app is running and geronimo needs to make a security decision, so it calls implies on your policy. You might follow Geronimo's idea of delegating the decision to the PolicyConfiguration. In any case, lets look at the information available to your provider to make this decision. Some of this is available directly and some is available through PolicyContextHandlers that consult ThreadLocals that are filled in by geronimo as the request traverses the containers. You get this info from PolicyContext.getContext(String key) passing in a key for the info you want.

role-permission mapping from the specc dd

This is the unchecked and excluded permissions together with the role to permissions mapping installed into the PolicyConfiguration as the app was starting up.

The Principals in the Subject

Implies supplies you with the current ProtectionDomain, and you can get the Subject's principals using
Principal[] principals = domain.getPrincipals();
If you have a principal to role mapping you can use this to figure out the roles you are in and use the role permission mapping to decide on the permissions. This is what the default geronimo JACC provider does.

The Subject itself

This is available through the Subject PolicyContextHandler PolicyContextHandlerContainerSubject. You get this by calling

PolicyContext.getContext("javax.security.auth.Subject.container");

EJB object

PolicyContext.getContext("javax.ejb.EnterpriseBean");

returns the actual object instance of the ejb bean that the request is going to.

EJB method call arguments

PolicyContext.getContext("javax.ejb.arguments");

returns an Object[] containing the ejb method arguments. This is not available for ejb web service requests. The ejb method name is available from the permission you are checking against.

ServletRequest

PolicyContext.getContext("javax.servlet.http.HttpServletRequest");

returns the current HttpServletRequest.

SOAP request

PolicyContext.getContext("javax.xml.soap.SOAPMessage");

returns the current SOAP message. This is not yet implemented in geronimo! See GERONIMO-2622.