Authorization concepts and solutions for J2EE applications

Paul Ilechko
Mannie Kagan

July 19, 2006

Introduction

Authentication and privacy are security issues that have been largely standardized and productized. Authorization, on the other hand, is generally an area where every solution tends to become unique, depending on specific characteristics of the application in question. That said, there are some fairly common patterns that are repeated in many applications, and there are best practices in terms of how to effectively use the features provided by IBM WebSphere Application Server (and other application servers), even if the built-in authorization does not fully meet the business needs for a given scenario.

This article summarizes a few basic authorization patterns and the state of authorization technology in the J2EE™ space and lays out a common sense approach to solving authorization problems in the general case. Much of this information is product agnostic, although we will discuss specific features of WebSphere Application Server where applicable.

What is authorization?

Authorization is about managing access to protected system resources based on the rights of a user or class of users (in this context, "user" may refer to some foreign system, not necessarily an individual). Authorization, therefore, presupposes that authentication has occurred, as it is impossible to provide appropriate access controls if you do not know who the user is -- other than in the trivial case where all users have the same rights. We will detail a variety of patterns or styles of authorization later, but we will quickly summarize them here.

- **Role-based authorization**
  Role-based authorization provides access to resources based on the fact that the user is a member of some class of users. This typically depends on the user having group
identifications defined in a user registry. This methodology defines various components of the protected system and maps those components to the groups of users who can access them. This type of authorization is the standard of the J2EE world. It is generally a fairly coarse-grained form of security that deals with access to functional areas of the system, perhaps defined by URLs or method calls on an EJB component. This is not to say that you can't define more granular protections using roles; you can, but frequently it is not effective to do so.

- **Instance-based authorization**

  This pattern provides a more granular level of authorization than is typically provided by J2EE role-based authorization, getting down to the level of an individual object within a system. We are now dealing with protection of data, not just of functionality.

- **Relationship/Ownership-based authorization**

  This type of authorization is a specific case of the instance-based pattern where there is an owner relationship within the data structures of the application between the user and some other protected data. This typically means that the authorization rules have to be embedded into the data access logic of the application itself.

- **User interface customization**

  While not truly a type of authorization, a closely related problem is that of customizing a user interface to only show the specific functions that a specific user should be allowed to see. For example, a menu or portal page will not show links or portlets that the user is not authorized to access. This is really a form of pre-authorization; where authorization asks "can I do that?", pre-authorization asks the slightly different question of "if I were to ask if I could do this, what would the answer be?" However, as we will see later, the actual coding techniques can be identical in both cases.

**J2EE role-based authorization**

Role-based authorization protects resources by only making them available to users who have been assigned to the appropriate role. J2EE provides a standard way of implementing this, using both declarative and programmatic techniques. We will discuss the details of J2EE security later in the Authorization technologies section. However, it is important to understand the concept of role in the J2EE world. The key thing to remember is that J2EE roles are not groups of people. They are logical, application-specific names that can be mapped to users and groups at deployment time. Roles themselves generally represent collections of permissions in the J2EE application.

The actual definition of which users belong to which groups is managed in a user registry. This may be a corporate LDAP server or it may be a database that is only used by a single application, but the repository of user and group information is external to the J2EE environment. Within the J2EE application, you can define and name security roles and constrain resource access based on those roles. A role is therefore a collection of permissions; to be useful, you have to bind it to some set of users when you deploy the application. This mapping can be to one or more individual users by name, to one or more defined groups of users, or any combination of these. The term principal is used generically to refer to either a user or a group of users.

What you will see, then, is that while a J2EE role may closely mirror a registry group with a simple one-to-one mapping, it does not have to do so; you can have a role of "manager" that maps...
Authorization concepts and solutions for J2EE applications

To a group called "manager" in the simplest case, but the power of J2EE roles is that they give you the flexibility to change bindings when organizational change occurs without the need to make programming changes. So, for example, let's say that a specific application function is only available to employees in the legal department, and these employees are all in Department 102. A J2EE role called "legal" can be created in the application and bound to the registry group "Dept102". If at some later time that department is reorganized and half of the employees are moved into Department 507, then you can change the binding to map to both "Dept102" and "Dept507". Then, when any employee from either department authenticates to the system, they will be assigned the role of "legal", and the system will provide them with the appropriate access.

**Custom role-based authorization**

Even with the programmatic APIs, there will be situations where the J2EE role model is not sufficiently flexible to meet the business needs. However, before taking any rash action, you should investigate the possibility of building on, rather than replacing, the J2EE model. The advantages of this include:

- Leveraging native application server authentication is crucial to creating a properly secure system. (See [WebSphere Application Server V6 advanced security hardening](https://www.ibm.com/developerworks/library/specifications/ws-asv6advsec/)).
- Good security provides defense in depth; using more than one approach is not necessarily a bad thing.
- The possibility of performing basic declarative security using J2EE, then using a customized approach for more detailed authorization logic.
- It is not cost-effective to reinvent what you get for free!

Also be aware that sometimes you can obtain the needed flexibility using other pluggable aspects of the application server security runtime. For example, if the issue is that the role-to-group bindings need to be dynamic at run time, based on information provided when the user logs on, then this can be addressed using JAAS login extensions (see [Advanced authentication in WebSphere Application Server](https://www.ibm.com/developerworks/library/specifications/ws-wasp/)). In WebSphere Application Server V5.1.1 or later, you can create dynamic group memberships in either a Trust Association Interceptor or a JAAS login module. However, if roles are contextual, based on specific usage of the application, then there may be a need to buy or build an authorization framework. An example of contextual usage patterns can be found in a healthcare application: a doctor might login to the system once, but depending on the context of the patient or the healthcare entity being viewed (hospital or clinic, for example), the role of the doctor might change from referring physician to attending physician. This change is dynamic throughout the session, and is not limited to information available at authentication time.

Once you get to the point where you determine that J2EE authorization alone is insufficient, the next question becomes whether to build or purchase an authorization solution. Enterprise security products, such as IBM Tivoli® Access Manager, provide flexible, policy driven authorization capabilities. Integrating these capabilities with the application server, however, can be a challenging proposition. Some of the questions you need to address include:

- Do you use a proprietary API?
- What functionality will you provide?
• What is the performance penalty of externalizing authorization requests?

Java™ Authorization Contract for Containers (JACC) is a standards-based approach to integrating external security managers with an application server. JACC provides the ability to delegate the permissions checking of security authorization to an external provider. Since the authorization checking is performed before the container passes control to the application, JACC has the advantage of providing the ability to clearly distinguish custom authorization logic from application logic, thereby satisfying the separation of concerns requirement. However, there are caveats when considering the use of JACC, and so it is necessary to take a closer look at JACC to understand exactly what it does and where it may be useful. We will discuss JACC later in more detail.

Instance-based authorization

Instance-based authorization, as the name implies, is about authorizing access to a specific instance of some object. Instance-based authorization typically protects instances using access control lists (ACLs), which are stored in some type of policy store and can be used to make access decisions. It is possible, although potentially unwieldy, to use J2EE roles as ACLs. After all, as we discussed earlier, a J2EE role is merely a name, a logical construct that can be bound to any set of principals. This approach does not scale well to large numbers of ACLs and does not handle the situation where ACLs can be modified dynamically while the application is executing; remember that the deployment descriptors statically define J2EE roles when the application starts. Given these constraints, it is probably better in many cases to use either an external security solution or a custom framework for ACL instance-based security.

To make this a little easier to understand, let's use an example. Suppose that we have a new J2EE human resources system that lets users perform certain print-related tasks, such as sending a document to a printer and viewing and managing print queues. Certain restrictions need to be put in place due to the sensitive nature of the data that could be printed. For example, there could be rules about which users are authorized to use or manage which printers. There could also be rules about the use of printers at certain times of day; perhaps there are concerns about leaving sensitive materials on certain printers overnight.

We could potentially use standard J2EE role-based security to manage printer access. For example, we could define a role for each combination of printer and type of access required, and end up with roles like “Allowed_to_print_to_PrinterX”, or “Allowed_to_manage_queue_for_PrinterY”. We would then have to write code in the application to iterate through all the possible roles, using isUserInRole() calls, to validate that the current user was authorized to do whatever print-related action they were attempting, and we would have to bind each role to the appropriate user registry entries.

As you can see, this is not a very clean implementation of the authorization requirements. Any addition of new printers would require changes to the application; any changes in rules concerning who can do what would require re-binding the roles to principals and re-deploying the application. In addition, we haven’t even begun to consider the time of day question. We need to be able to store configurable information with each printer, but we don’t really have anywhere to put it within the standard J2EE authorization model.
Now, if we were to use a separate authorization service, this could simplify the solution greatly, as we would just ask the authorization service whether or not the current user can perform some action on some object. All of this would be externalized, greatly simplifying the programming model, and enable changes to be made to the external provider to be reflected real-time in the running application. We will take a look at this problem again later.

Ownership relationships

Direct ownership is a very common form of instance-based relationship between a user and some protected data. For example, in a brokerage application, it may well be that a financial advisor can see the accounts of their personal customers, but not those of other customers of the firm. Branch managers may be able to see all accounts for all customers of the financial advisors that work for them, but not for customers of other branch offices, and so on. In this case, the permissions are built into the very data structures of the application.

It would make very little sense for an application which is presenting an account list to a user interface to retrieve all the data, and then apply ACL based permissions to each row. The primary issue here is performance: it is generally much faster to filter out rows in the database engine than it ever could be in Java code. Although caching might alleviate this to some degree, another problem might arise if data is retrieved in blocks of a certain size to support user interface pagination. If the filtering happens in the application, there is no way of knowing how many rows to request, thus potentially leading to multiple calls to get a single page of data.

Unfortunately, this application-side filtering approach is used quite frequently despite its flaws. In particular, problems may be caused when applications are using object-relational mapping tools to perform data access and then try to apply security in some generic way to the classes that are instantiated. This is an incredibly inefficient use of computing resources, and it is highly unlikely that acceptable performance and scalability will ever be achieved.

The only sensible approach is to let the database do the filtering. This can be achieved by leveraging native database authorization, if the database knows the end user's identity, or by embedding the authorization logic in the application by modifying the filtering of the actual data access logic, typically by adding items to the `WHERE` clause of a SQL statement. Depending on the application complexity, it may be feasible to develop a custom framework that uses metadata to describe authorization rules and applies security changes to the SQL automatically. If stored procedures are used for data access, then these may well be optimal places to apply these rules.

User interface customization

The dynamic modification of a user interface to only show actions that the user is entitled to perform is often regarded as an aspect of authorization. While this may be debatable, there is no denying that this is a common problem; it is certainly a poor practice (and possibly an incitement to hacking) to show users links to tasks that they are not authorized to perform.

Customizing the UI to remove links and menu options is very analogous to role-based authorization. It is often simply a case of iterating through the items using the `isUserInRole()` call to validate whether the user should see it or not. It may also be possible to customize the actual
data on a form in the same way, although this can lead to long-winded programming without providing sufficient flexibility, depending on the business requirements. This can get particularly complex when you need to integrate access control customizations with other types, such as user preferences. If the user has access to some preference tooling that enables them to develop their own view of a particular UI component, then that tooling would also need awareness of the authorization rule metadata to prevent users from adding items to their personalized forms that they should not have access to. As in our earlier discussion of custom role-based authorization, there may be situations where the view of data is contextual such that it cannot be simply adjusted based on static roles.

Authorization technologies

J2EE provides both declarative and programmatic role-based authorization capabilities. Although it is not the purpose of this article to provide a tutorial on J2EE security, we will briefly summarize some of its major aspects.

J2EE declarative authorization

J2EE security roles and constraints are added to the deployment descriptor of the application. Web constraints are added to web.xml and are based on URL patterns. EJB constraints are added to ejb-jar.xml and define method level permissions. WebSphere Application Server provides tools to enable a deployer to define which users and groups (as defined in the User Registry) have access to which security roles. The application server runtime uses these bindings to determine if a specific user has a role assigned, and then decides whether or not the protected resource can be accessed.

J2EE programmatic authorization

While declarative authorization is simple and quite powerful, it frequently will not provide sufficient control. This is particularly true once you examine the issue of user interface customization. To provide additional flexibility, J2EE provides an API that lets the developer test whether or not the current user has access to a specific role, using these calls:

- `isUserInRole()` for Servlets
- `isCallerInRole()` for EJBs

Additionally, there are other API calls that provide access to the user's identity:

- `getUserPrincipal()` for Servlets
- `getCallerPrincipal()` for EJBs

Using these APIs, you can develop arbitrarily complex authorization models. In the extreme, if the J2EE role information is not useful, you can take the user identity information and use it to look up and execute arbitrarily complex authorization rules that are stored in some other source, such as a database or a rules engine.

Any code written to these latter APIs may have portability issues, as the format of the Principal object is not standardized and can vary across different application server platforms.
JACC

As mentioned earlier, JACC provides a standard approach to plugging external authorization providers into a J2EE container. In this way, third party authorization providers are able to make authorization decisions when a user accesses J2EE resources.

Prior to JACC, there was no specification to address the access decisions the application server makes. In the absence of JACC, vendor implementations and proprietary interfaces were used for third party vendor product integration. There was no standard way for third party authorization providers, such as Tivoli Access Manager, to plug in to application servers to make access decisions. To address these issues, Sun Microsystems™ introduced JACC in the J2EE 1.4 Specification, providing a standard mechanism for third party providers to collect the security policy information from the application and the application servers, and to implement policy decisions at run time.

These run time policy decisions are made through the java.security.Policy class. The Web and EJB containers enforce security policy at run time during method pre-dispatch by invoking the Java Security Manager checkPermission method, which delegates to the implies method on the abstract class java.security.Policy. The JACC provider is obliged to provide a concrete Policy sub-class, which implements the implies method. In addition, the JACC provider is obliged to implement the PolicyConfiguration interface, and to provide a concrete PolicyConfigurationFactory class.

At application deployment time, policy configuration permission objects add themselves to the provider. They contain the Web and EJB security-related J2EE deployment descriptor information configured in the container. This is the mechanism the container uses to pass security policy information to the provider.

At run time, when a user accesses a Web or EJB resource, the container creates the appropriate Web or EJB permission object and invokes the provider’s Policy object implies method. The permission object contains information on the resource being accessed: if the resource is a Web resource, then the permission object is the URL; if the resource is an EJB, then the permission object contains the names of the EJB object and method. The Protection Domain object encapsulating the Principal, if any, is also passed to the Policy object. Moreover, the provider is able to access fine grained resource attributes, such as the specific arguments of EJB calls via the getContext method of PolicyContext handlers registered by the container. The provider is then responsible for granting or denying access to the resource by returning true or false to the Policy.implies method. (See the JACC API Specification for more information on JACC.)

The original intent of JACC was to provide a standard interface to enable third parties to plug in authorization providers to make access decisions. Let us now consider the range of security requirements that JACC might fulfill.

Externalize the standard J2EE security model

The standard J2EE security model contains a description of roles, which encompasses the security-constrained resources that can be accessed by principals. As mentioned earlier, the
JACC provider has access to this information and is fully able to support externalizing these access decisions.

**Extend the standard J2EE security model**

The standard model is **static**. Constraints and roles are fixed when a secured application is deployed. There is no provision in the standard model for modifying the role-constraint relationship subsequent to deployment. However, the JACC provider is not limited by the static nature of the standard J2EE security model. The JACC provider has the autonomy and information necessary to also make dynamic, run time-based access decisions.

For example, we often want to consider: can a certain role perform a particular operation upon a specific object? Let us consider how this maps to standard J2EE security, and then to JACC usage before we look at the trade-offs that come with JACC.

Standard J2EE is partially able to satisfy this mapping, for example, by simply mapping the object to a session EJB class, and the operation to a particular secured EJB method. Also, as we mentioned before, this mapping is **static**.

JACC, on the other hand, is able to support the mapping described above, but also to support **dynamic, instance-based** access control as well. For example, the authorization check could be performed by passing operational parameters on an EJB method call, enabling the authorization check to be dynamically determined. We will discuss an example later.

Let us now consider some caveats in the consideration of JACC, relating to performance and implementation complexity:

- There might be performance implications in using the JACC method parameter-based access. For example, WebSphere Application Server supports all of the policy context handlers that are required by the JACC specification. However, due to performance impacts, the EJB arguments policy context handler does not activate unless it is specifically required by the provider. Performance impacts result if objects must be created for each argument of each EJB method.
- Implementing a JACC to support anything beyond standard J2EE security can be a complex task. JACC provides good support for implementing the standard J2EE security model. For example, there are very usable convenience methods on the permissions objects that check whether the incoming permission object satisfies the J2EE deployment descriptor policy requirements. On the other hand, JACC does not supply support for performing non-standard permission checking, as would be required in extending the J2EE security model. Implementation of a JACC provider that extends the J2EE security model is a complex task and should not be undertaken lightly.
- The JACC configuration is cell-wide. Any JACC provider must satisfy the authorization requirements for all the applications in a cell. In the case where some of these are standard J2EE in nature and some are special in some sense, then a provider that satisfies both sets of requirements must be provided -- adding to the complexity.
Let's take a quick look at how we might address our printer authorization problem with JACC. We could have a “Printer” session facade EJB with methods like:

- `print(printerName:string, itemToPrint:string)`
- `manageQueue(printerName:string, queueOperation: string)`

Within our JACC provider, we could provide a mapping table of EJBs, roles, operations, and objects; for example, using the “Printer” EJB, “Managers” are able to “ManageQueue” on “PrinterX”.

The facade EJB methods could be optionally secured with J2EE authorization constraints. However, whether secured or not, the JACC provider Policy.implies method is invoked. Within the provider, once we have completed the regular J2EE authorization checking, we would check whether the EJB is one of those used in mapping, and then proceed to check the EJB parameter values against our mapping table. If the call is authorized, we would return True from the Policy.implies method.

As an aside, within our JACC provider, since we are executing on the user’s calling thread, we are able to check whether the calling principal is in role by invoking the regular J2EE isCallerInRole method against the roles in our table (for example, "Managers").

**Enterprise security products**

As we discussed earlier, enterprise security products, such as Tivoli Access Manager, are typically designed to solve sets of classes of authorization problems. This is achieved by externalizing the authorization rules from the application and by providing an API to call either the authorization rules engine, or a transparent invocation of the rules engine by plugging into the J2EE container using the JACC interface, or both. When a product provides both JACC and proprietary API-based solutions, it is important to understand the capabilities available in each model. For example, in the Tivoli Access Manager case, JACC does not add a great deal of value, as all it really provides is delegation of J2EE security checks without the ability to pass additional information about the object of the access control check, such as the parameters to the method call. In this case, the API approach is much richer, leading to a tradeoff between portability and functionality.

Now, let’s go back and take a look at how we would solve our printer authorization problem using Tivoli Access Manager.

Tivoli Access Manager provides the ability to define a protected object space and add ACLs to the objects at any location in the object hierarchy. In our case, we can define a tree structure for the printers that we wish to protect. ACLs have certain default actions (or permissions) associated with them, but, if necessary, we can create a custom action group. In our scenario, then, we might decide to create an action group called “print_actions”, which has permission bits specifically defined to control printing, such as “q” for queue to printer, “p” to print, “v” to view print queue, “d” to delete from queue, and so on. We can then add ACLs at appropriate locations in the object space, and add users or groups to those ACLs to define the available permissions. Each user or group that is added to the ACL can have a different set of active permissions, enabling very granular control over exactly what type of access is available to any user. If necessary, we could
also add a Protected Object Policy (POP) to any object, which adds additional protection, such as time-of-day access rules.

Once this configuration has been completed, all that is left to do is add a simple API call to the application. This uses aznAPI and is used to ask Tivoli Access Manager if a certain action is allowed. In this call, we pass the user credential, the identity of the protected resource, and the operation that is being requested, and then Tivoli Access Manager evaluates the ACL and POP on the object and returns a Boolean. As you can see, this is clearly a more flexible solution to the problem than is trying to bend a role-based security model to fit a situation that it was not designed to handle.

**Futures: SAML and XACML**

Trying to determine the future direction of any technology is a risky business, but it certainly seems likely that SAML (Security Assertion Markup Language) and XACML (eXtensible Access Control Markup Language) will be important aspects of future security policy implementations. Although it is not the purpose of this article to describe these technologies in detail, here is a short summary and some ideas on how these may be relevant in the J2EE authorization space.

SAML is generally considered to be a Web services standard, although in fact it provides an overall structure for the management of federated identity that is platform and protocol independent. SAML introduces the concepts of Policy Decision Points (PDP) and Policy Enforcement Points (PEP). As the name implies, a PEP is a point in the security architecture where authentication and authorization rules are enforced, based upon the policy defined in the PDP. XACML builds on SAML by providing the actual semantics used to define access control policy and authorization request and response messages. What neither SAML nor XACML provide is an infrastructure to deliver and manage security services; it is up to the enterprise security vendor to provide this.

So the question is: how does this impact J2EE application server security, and in particular, how does it affect authorization in this space? As we have already discussed, authorization capabilities in the J2EE application come in two forms; an embedded security engine within the application server itself, and the ability to delegate authorization to an external provider using the JACC interface. While it is possible that a J2EE application server vendor could use XACML internally to define authorization policy created by the application of J2EE security constraints, this is typically a black box and is not very interesting. However, what is far more interesting is the potential for vendors of enterprise security solutions to support XACML -- in particular to support the request-response protocol. To use the JACC interface today, you need to install a specific provider for the vendor of the security service, as the protocol from the provider to the service is undefined. With XACML defining that protocol, this (at least in theory) makes it possible to have generic providers that can communicate with any XACML aware service.

**Layers of authorization**

One final point before closing. In layered applications, which J2EE applications frequently are, it is important to perform authorization at all layers-- but particularly at the lowest layer. For example, if
a user makes a URL request from a browser that accesses a servlet which, in turn, calls an EJB, it is all too often the case that authorization only happens at the servlet layer -- or even worse, in JavaScript code in the browser. This can lead to an insecure system, as hackers may be able to write code to call the EJB directly and bypass the Web tier. Similarly, if the EJB is protected but the database is not, then the hacker can bypass the application completely and go straight to the crown jewels, the corporate data. It is essential to consider the possible access paths to every layer of a system, and also to consider how authorization integrates with other security aspects, such as network level protection.

**Conclusion**

This article described some of the issues involved with authorization for J2EE applications, along with some of the existing standard capabilities of WebSphere Application Server and other J2EE application servers. It is in your best interest to use existing standards wherever possible. When they do not provide the complete solution, it is wiser to use them as far as they are useful and then extend their capabilities, rather than replace them entirely with custom solutions. Authorization is a complex and difficult topic, so think carefully about the issues.

**Acknowledgements**

Thanks to Keys Botzum for all his assistance in generating many of the original ideas included here, and for reviewing the paper several times while we polished it.
Related topics

- WebSphere Application Server V6 advanced security hardening
- Advanced authentication in WebSphere Application Server
- Java Authorization Contract for Containers (JACC) API Specification

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)