Conditional Permissions

Final

22 Pages

Abstract

This RFC adds support for conditional permissions in PermissionAdmin. It allows user defined conditions to be added to Permissions managed by Permission Admin such that the permission evaluation will only take place if the condition is satisfied.

0 Document Information

0.1 Table of Contents

All Page Within This Box

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	2004-08-05	Initial Revision. Benjamin Reed.
0.1	2004-09-02	Added requirements, Condition evaluation as a set, and the transient state.
0.2	2004-09-09	Made encoded conditions the key to the location table in PermissionAdmin.
0.3	2004-10-04	Added language to handle multiple possible conditions.

All Page Within This Box

Revision	Date	Comments
0.4	2004-10-13	Added text about state preservation and recursive checks. Added note about default permissions.
0.5	2004-11-04	Changed to ConditionalPermissionInfo and clarified matching terminology.
0.6	2004-11-15	Added BundleSymbolicNameCondition.
0.7	2004-12-01	Added getAccessControlContext to ConditionalPermissionAdmin and made section on full bundle signing to be signature method agnostic.
0.8	2004-12-07	Minor editing changes.
0.9	2005-01-08	Fixed formatting and some typos.
0.10	2005-01-16	Fixed typos. Added clarifying language to section 5.3. Removed filtering for BundleSignerCondition and added wildcards for DN and RDN.
0.11	2005-02-01	Added the ability to use static getInstance() methods as well as Constructors to instantiate Conditions per Nokia's request. Added the ability to wildcard the value of an RDN and added text about recursive Condition checks per the CPEG F2F meeting. Added consequences of a missing permissions.perm file to fix a security hole.
0.12	2005-03-20	Changed META-INF to OSGI-INF BJH
Final	2005-05-27	Renamed getInstance method to getCondition. BJ Hargrave, hargrave@us.ibm.com

1 Introduction

This RFC introduces the concept of conditional permissions into OSGi and into Java. Currently permissions are associated with a bundle through PermissionAdmin. The permission evaluation for a bundle is currently based solely on the permissions assigned to a bundle in PermissionAdmin. In this RFC we introduce the concept of associating conditions with permissions such that the permission evaluation will be based on the permissions associated with a bundle and the conditions required the enable the permissions.

A simple example of a scenario enabled by this RFC is the case of two bundles: BrowserBundle and NetworkBundle. BrowserBundle has permission to access port 80 of any host in the *.com domain. In OSGi this would be represented by a PermissionInfo of the form (java.net.SocketPermission "*.com:80" "connect"). NetworkBundle can connect to any port on any host. The PermissionInfo in this case would look like

(java.net.SocketPermission "*" "connect"). In both cases we only want to enable the permissions if the user answers affirmatively to the prompt "Connect to the Internet?". So, if the user opens the browser in the BrowserBundle and tries to access a page on the Internet, the "Connect to the Internet?" prompt will appear and the connection will only succeed if the user answers positively.

If the user causes the BrowserBundle to connect to http://www.ibm.com using the services of the BrowserBundle, the user will get prompted "Connect to the Internet?", answer positively, and view the page. However, if the user causes the BrowserBundle to try to connect to http://www.congress.gov using the services of the BrowserBundle, the request should fail since the BrowseBundle does not have permission to connect to the *.gov domain. So, to provide a reasonable user experience the conditions should only be evaluated if the full permission check can succeed.

2 Application Domain

In the mobile phone environment the phone is very much under the control of the user. Even though a management system, be it the service provider or phone manufacturer, sets up permissions to control what code from different sources can do, often it is the user that actually enables the permission. An example of this kind of user control is the case where the user will be charged for certain operations. For example, if an application is going to connect to the Internet, the user may want to be prompted before allowing applications to gain needed permission to connect.

Other example of conditional permissions is a permission that is granted only if a certain SIM card is installed in the phone. An example of a context dependent conditional permission is a permission that is granted only if the cost of the operation is low.

3 Problem Description

MEG Policy RFP [3] contains some requirements when policy rules depend on some condition. Examples of such conditions are user confirmation, dependency on operator or device characteristics and transfer cost dependency. During the MEG Policy design it was decided that MEG policy requirements would be mapped to Java permissions. Java2 permission system does not have a notion of conditional permissions. The solution described in this document aims to close this feature gap.

4 Requirements

The conditional permission requirements were derived from the following requirements in RFP-55 [3].

REQ-POL-06-05. It MUST be possible to restrict the cost incurred by the install/update action by policy. *For example it is possible to state that expensive wireless network cannot be used for the install action.*

REQ-POL-06-08. It MUST be possible to describe that user must be prompted for confirmation as policy condition. If the policy rule matches, the user will be prompted for confirmation and the [install or upgrade] action is executed only if the user accepted the operation.

REQ-POL-07-05. It MUST be possible to describe that user must be prompted for confirmation as policy condition. If the policy rule matches, the user will be prompted for confirmation and the [uninstall] action is executed only if the user accepted the operation.

REQ-POL-10-11. It MUST be possible to describe that user must be prompted for confirmation as policy condition. If the policy rule matches, the user will be prompted for confirmation and the [configuration management] action is executed only if the user accepted the operation.

REQ-POL-13-10. It MUST be possible to describe that user must be prompted for confirmation as policy condition. If the policy rule matches, the user will be prompted for confirmation and the [execution state management] action is executed only if the user accepted the operation.

REQ-POL-16-01. It MUST be possible to use an operator-specific subscriber identifier (like IMSI) as policy condition.

REQ-POL-16-02. It SHOULD be possible to use an operator identification code as policy condition. *For example it is possible to state that certain policy rule is applicable only if the smart card identification module in the device belongs to certain operator.*

REQ-POL-16-03. It MAY be possible to use a device identification code (like IMEI) as policy condition. *For example it is possible to state that certain policy rule is applicable only if the device is the one identified by the IMEI.*

During the MEG Policy design these general requirements yielded the following, more implementation-specific requirements for the conditional permission mechanism.

REQ-CP-01. It MUST be possible to assign user prompt, operator or device identifier and transfer cost conditions to management-related permissions.

REQ-CP-02. It SHOULD be possible to assign programmer-defined condition elements to any permission.

REQ-CP-03. The conditional permission mechanism SHOULD be similar to the user confirmation mechanism in MIDP2.0 as much as it is reasonably possible (considering that MIDP2.0 is based on CLDC and MEG is based on CDC).

REQ-CP-04. When user confirmation conditions are used, the conditional permission framework SHOULD provide the necessary interfaces so that the user is not bothered with unnecessary prompting.

All Page Within This Box

5 Technical Solution

Our solution consists of two parts. First, we define a method of specifying conditions that must be met before a set of permissions will be assigned to a bundle. Second, we define three conditions, BundleLocationCondition, BundleSymbolicNameCondition and BundleSignerCondition, that can be used to indicate the bundle location or signature needed before a set of permissions will be assigned to a bundle.

Conditions are specified using a series of strings where the first string is a condition type and the following strings are a variable number of constructor arguments. Their string form of Conditions is very much like the string form of Permission info except that square braces, [], are used rather than parenthesis. This is similar to the encoding of PermissionInfo objects. Here are some examples of encoded Conditions:

```
[org.osgi.framework.BundleLocationCondition “http://www.ibm.com/bundles/superBundle.jar”]

[org.osgi.framework.BundleSignerCondition “CN=Super Man, O=IBM, C=US; OU=ACME Cert Authority,
O=ACME Inc, C=US”]

[org.osgi.meg.UserPrompt “Connect to the Internet?” “blanket”]
```

To construct a Condition object, the framework uses reflection to find a static method called “getCondition” that takes a Bundle object and the appropriate number of String arguments and returns the result of the method invocation. If the “getCondition” method cannot be found, the framework will try to find a constructor with that signature and use that constructor to create the Condition. Each bundle has a single ProtectionDomain. Conditions are constructed using a public constructor with a Bundle object as the first parameter and a variable number of String arguments according to the number of arguments given. A Condition object will be unique to a bundle’s ProtectionDomain. Thus, any queries made on a Condition object will be with regard to the bundle to which the Condition object corresponds.

Permissions are assigned to a bundle through the ConditionalPermissionAdmin service, which is a special service registered by the framework. It allows ConditionalPermissionInfos to be specified that the framework will use to assign permissions to bundles. A ConditionalPermissionInfo consists of an array of Conditions and a corresponding array of PermissionInfos that describe permissions that will be assigned to bundles that satisfy all of the Conditions in the array of Conditions.

The permissions for a ProtectionDomain (and therefore the bundle the ProtectionDomain represents) are found by instantiating sets of Conditions with the Bundle object that corresponds to the ProtectionDomain and populating the ProtectionDomain with permissions whose Condition objects are immutable and satisfied or may be satisfied when the unevaluated Conditions are evaluated.

Because the framework evaluates the Conditions, only Condition classes from the system classpath and from the framework’s classpath, if any, will be used to resolve Conditions.

5.1 Framework defined Conditions

There are three Condition types that are specified by the framework. Each of these types are in the “org.osgi.service.condpermadmin” package. All of these Conditions are immutable since their values cannot chain after bundle installation. The first Condition type is the BundleLocationCondition. It takes a single parameter: the bundle location, which is a glob expression using the expression syntax of FilePermission.

All Page Within This Box

The second framework Condition is the BundleSymbolicNameCondition. Its first parameter is a BundleSymbolicName that may have '*' to do wildcarding as defined in LDAP queries [5]. The second parameter is the version range as defined in section 5.4 of RFC 79 [4]. If the second parameter is not set, the Condition will be satisfied by any version of a bundle with a BundleSymbolicName that matches the first parameter. Because the BundleSymbolicName is set by the developer and not validated by the framework the BundleSymbolicNameCondition should only be used with another validated Condition such as BundleSignerCondition. In that case, the joint use implies that for the permission set under evaluation the signer is authorized to use the BundleSymbolicName.

The final Condition is the BundleSignerCondition which is a chain of X.500 Distinguished Names (DNs). The X.509 certificates that correspond to the signers of a bundle will be matched against this chain to determine if the BundleSignerCondition is satisfied. Only valid X.509 certificates that are rooted in a trusted certificate authority or that are installed in the framework will be used. Any other certificates will be ignored.

Each DN in the chain is encoded according to RFC 2253 and separated by a semicolon. Using the terminology of RFC 2253, each DN in the chain is made of an ordered set of Relative Distinguished Names (RDNs) separated by commas. For example, an X.509 certificate with the DN chain of "CN=Super Man, O=IBM, C=US; OU=ACME Cert Authority, O=ACME Inc, C=US" is used to indicate that the signer is Super Man from IBM in the US, and his identity is certified by ACME Cert Authority of ACME Inc. in the US.

Wildcards ('*') can be used to allow greater flexibility in specifying the DN chains. Wildcards can be used in place of DNs, RDNs, or the value in an RDN. If a wildcard is used for a value of an RDN, the value must be exactly "*" and will match any value for the corresponding type in that RDN. If a wildcard is used for a RDN, it must be the first RDN and will match any number of RDNs (including zero RDNs). For example,

[org.osgi.framework.BundleSignerCondition "*, O=IBM, C=US; OU=ACME Cert Authority, O=ACME Inc, C=US"]

will be satisfied by any X.509 certificate chain from someone in IBM in the US that is certified by ACME Cert Authority.

[org.osgi.framework.BundleSignerCondition "*, O=IBM, C=*; *"]

will be satisfied by any X.509 certificate chain from someone in IBM (no matter which country) that is certified by a trusted certificate authority. Because the wildcard indicates one or more DNs, this Condition will be satisfied by a DN chain of "CN=Super Man, O=IBM, C=US; OU=ACME Cert Authority, O=ACME Inc, C=US" as well as "CN=Super Man, O=IBM, C=US; OU=Internal CA, O=IBM, C=US; OU=ACME Cert Authority, O=ACME Inc, C=US".

[org.osgi.framework.BundleSignerCondition "*; OU=ACME Cert Authority, O=ACME Inc, C=US"]

will be satisfied by any X.509 certificate chain from someone certified by ACME Cert Authority.

How the actual X.509 validation is done and how the signature algorithms are implemented are outside the scope of this specification.

5.2 Satisfying Conditions

The most important method of a Condition is the isSatisfied() method, which tells the state of the Condition. Unfortunately, some Conditions (e.g. UserPrompt) require a large effort to determine whether a Condition is satisfied. We use isEvaluated() to distinguish between Conditions that can be quickly checked and those which need to extra processing to do the check.

This distinction between an evaluated Condition and a satisfied Condition is an important one to ensure efficient processing of the Conditions and correct behavior of the Condition implementations. A Condition is considered

evaluated the Condition object already knows whether it is satisfied or not, or in other words, the object can answer whether it is satisfied using its internal state. The isEvaluated() and isSatisfied() methods of Condition are used to interrogate these two states. We will see exactly how evaluated Conditions are used in the next section.

Checking an instance of a Condition object to see if it is satisfied does not always cause the Condition to be evaluated. For example, in the case of UserPrompt with the “blanket” parameter once the user answers affirmatively, we consider the Condition to be satisfied for the lifetime of the bundle. So until the affirmative answer, we must continue to reevaluate the Condition, but once the user answers affirmatively the Condition will be satisfied and further evaluation is not needed.

To allow further optimizations we also have a mutable state to indicate whether the Condition will ever change state for the lifetime of the ProtectionDomain. For example, if the UserPrompt Condition is used with the “blanket” parameter, once the user answers affirmatively to the prompt, the Condition will always be satisfied; therefore, it becomes non mutable. The isMutable() method is used to check whether a Condition is mutable. Note, if isMutable() is false, isEvaluated() MUST also return true.

5.3 SecurityManager and ProtectionDomain implementation

Since there is no notion of conditional permissions in Java 2 permissions, protection domains, and policy, we need to add semantics to the existing Java 2 permissions infrastructure. Our only hook for adding such semantics is the java.lang.SecurityManager. Java lets us add our own SecurityManager and it is possible to construct one in such a way that we can implement conditional permissions.

The framework implements the ProtectionDomain for a bundle by categorizing the permissions available to a bundle according to the mutable state of the Conditions that may be satisfied by a bundle.

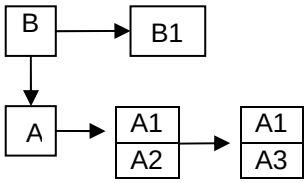
The first category is the non-mutable set of permissions. This set is made up of permissions that are enabled by non mutable Conditions that are satisfied for the ProtectionDomain. All of the permissions in this category are put into a single PermissionCollection to be evaluated together.

The second category is the mutable sets of possible permissions. These sets are made up of permissions whose corresponding Conditions are made up of at least one mutable Condition and none of the other Conditions, if any, are evaluated and unsatisfied. Unlike the non mutable set of permissions, the mutable sets are evaluated individually rather than as a unified PermissionCollection.

During a permission check, if permissions from this second category are used, the unevaluated Conditions need to be tracked for later evaluation before permission check can succeed. These unevaluated Conditions are first grouped by ProtectionDomains. Within the ProtectionDomain grouping, the Conditions are grouped by the ConditionalPermissionInfos they come from. Because the same Condition can be in different ConditionalPermissionInfos in the same ProtectionDomain, the same Condition can be in the unevaluated Conditions list multiple times. In order for a check to succeed all the Conditions must be satisfied in at least one group of Conditions for each ProtectionDomain.

The details of how the unevaluated, but needed Conditions, are tracked are an implementation detail. Since the framework implements the ProtectionDomain and SecurityManager private data structures can be used to do the tracking.

For example, if we have two bundles A and B, where the needed permission is enabled for A if the Conditions A1 or A2 are satisfied or A1 and A2 are satisfied. B has the needed permission if Condition B1 is satisfied. A calls B causing the permission check, the unevaluated Conditions set would have the following structure:



For the check to succeed both B1 and A1 must be satisfied and either A2 or A3 must be satisfied.

The framework SecurityManager does a permission check in the following form:

1. It gets the AccessControlContext in effect. This means that it does a AccessController.getContext() if it is not passed one.

2. AccessControlContext.checkPermission() is called, which causes the call stack to be walked. At each level the ProtectionDomain of the calling class is evaluated for the permission using the ProtectionDomain.implies() method, by evaluating the following rules in order:

    a. If the non-mutable set of permissions implies the permission, the check is considered successful and the implies() method returns true.

    b. For all of the mutable sets of PermissionCollections that imply the permission, any unevaluated Conditions are added to the list unevaluated Conditions for the ProtectionDomain, and the implies() method returns true. If at least one of the mutable sets imply the permission, the check is considered successful and the implies() method returns true.

    c. If neither of the previous rules are successful, the implies() method returns false.

3. If the previous step completes successfully (everything returns true), the unevaluated Conditions are checked to see if they are satisfied. Evaluation is done by Condition types. The SecurityManager determines a set of Conditions of a given type that all must be satisfied by processing the list of unevaluated Conditions for each ProtectionDomain (using implementation specific methods and data structure). The Condition.isSatisfied(Condition[], Dictionary) method is used to allow the Conditions to be evaluated as a group. If isSatisfied(Condition[], Dictionary) returns false, another set of unevaluated sets will be passed to isSatisfied(Condition[], Dictionary). Each possible set will be tried until one returns true or all possible sets are exhausted at which time the check will fail. As the various possible sets of Conditions are evaluated the Condition implementation needs to be able to remember the evaluated states between invocations of isSatisifed(Conditon[], Dictionary). The SecurityManager aids in preserving this information by creating a Dictionary object on the first invocation of Condition.isSatisfied(Condition[], Dictionary) for a give Condition type and passes it as the second parameter. All subsequent invocations of isSatisfied(Conditon[], Dictionary) in the same permission check will receive the same Dictionary object.

4. If at least one combination of Conditions were satisfied the permission check succeeds, otherwise the check fails. The SecurityManager will not hold any references to the Dictionary objects that were created in the previous check to allow them to garbage collected.

Using the Java API bundle programmers can also do security checks using the AccessController.getContext() directly. This style of programming should be discouraged, but unfortunately, it may not always be avoidable when dealing with legacy code. If a permission check is done outside the SecurityManager, the evaluation will take place just as outlined above, except that in step 3.b any unevaluated Conditions would be evaluated and false would be returned if the Condition is unsatisfied. This evaluation must take place during step 3.b because there would be no thread local Vector of Conditions setup.

5.4 Considerations when implementing Conditions

When implementing Conditions it is the implementer's responsibility to preserve any persistent state. The Condition objects will get reconstructed when the framework is restarted or the ProtectionDomain is reconstructed. Framework implementations may also use optimizations that cause Conditions to get created and destroyed multiple times within the lifetime of an instance of a ProtectionDomain, so implementers must not make assumptions that tie the lifetime of a Condition to a ProtectionDomain.

Implementers must also take care of recursive permission checks. Implementers of Permission classes have the same issue, but Condition classes in general have much more sophisticated logic than Permissions classes. For example, a UserPrompt Condition has the potential to cause many permission checks as it interacts with the UI API. If there is a chance that permissions will be checked, the implementer of the Condition should use AccessController.doPrivileged() to ensure needed permissions. Even when using AccessController.doPrivileged() there is still the possibility that a chain of method invocations can cause a Condition to get reevaluated. Since the framework cannot detect all such recursive calls, it is up to the implementer of the Condition class to detect a recursive call to a Condition and take action to avoid infinite recursion.

The framework can detect nested Condition checks when they occur on the same thread. When the framework detects nested conditions, unevaluated Conditions will not be evaluated. Instead, they will be treated as unsatisfied conditions.

5.5 Context Dependent Conditions

In the introduction we mentioned the need for context dependent Conditions. For these types of Conditions to be evaluated correctly, we must be able to pass a description of the current context along with the permission to be checked. Since the bundle checking the permission does not know the Conditions for the permissions of the bundles being checked, it is very possible that the context is not actually used.

To implement context dependent Conditions we take advantage of the knowledge that the permission check is done on the same thread as the caller of SecurityManager.checkPermission(). The programmer uses a thread local variable to store the context for the specific Condition class and then must reset it after the check is done. (As pointed out in the previous paragraph, we cannot always count on the context being used, thus we cannot reset it automatically.)

To illustrate how this is done, we will show the implementation of a skeleton of the TransferCostCondition:

```
public class TransferCostCondition implements Condition {
      int maxCost;
      public TransferCostCondition(String cost) {
           maxCost = Integer.parseInt(cost);
      }
      private static ThreadLocal context = new ThreadLocal();
      public static void setTransferCost(int cost) {
           context.set(new Integer(cost));
      }
      public static void resetTransferCost() {context.set(null);}
      public boolean isSatisfied() {
           Integer i = (Integer)context.get();
           return i!=null ? i.intValue()<=maxCost : false;
      }
}
```

This Condition is constructed with a maximum transfer cost. It is considered satisfied when the context transfer cost is less that the maximum transfer cost. Since there the state of the Condition is fully contained in the class it

All Page Within This Box

is always considered evaluated. When the transfer cost is know it should be set using the static setTransferCost() and resetTransferCost() methods. Here is a code snippet of how this class would be used:

```
TransferCostCondition.setTranferCost(2);
… Other Processing …
sm.checkPermission(new SocketPermission("www.ibm.com:80", "connect"));
… Other Processing …
TransferCostCondition.resetTransferCost();
```

In this example if the caller had a permission of the form

(java.net.SocketPermission "*" "connect" [org.osg.meg.TransferCost 3])

the snippet would run successfully. It would fail if the caller had the following permission

(java.net.SocketPermission "*" "connect" [org.osg.meg.TransferCost 1])

5.6 ConditionalPermissionAdmin and PermissionAdmin interaction

If the framework provides both a ConditionalPermissionAdmin service and a PermissionAdmin service, the permissions of the bundle will be defined using only one of the following rules evaluated in this order:

- If there is an entry for a bundle's location in PermissionAdmin, only the permissions assigned to that location will be given to the bundle and any permissions that would be assigned to the bundle by ConditionalPermissionAdmin will be ignored.
- If the bundle's location is not in PermissionAdmin and the Conditions of at least one ConditionalPermissionInfo could be satisfied by the bundle, only the permissions defined in ConditionalPermissionAdmin will be assigned to the bundle. Note that a ConditionalPermissionInfo with a zero length array of Conditions will apply to all bundles.
- If above rules do not apply to a bundle, the default permissions as defined by PermissionAdmin will be assigned to the bundle.

5.7 Permission Checking from Bundles

In order for conditional permissions to work properly bundle programmers must use the SecurityManager to do permission checks. As mention in **Error! Reference source not found.**, conditional permissions are only guaranteed to work when the SecurityManager is used to do permission checks. The Java API documentation should be consulted for correct use of the SecurityManager. We do not require any extra effort on the part of the Java programmer except as noted in the previous section when using context dependent Conditions.

5.8 Full Bundle Signing

Only fully signed bundles are supported by the framework. This means that the manifest must have a valid digest in all the signature files for the bundle to be installed. It also means that if there is a signature file, any classes or resources that are not signed by all the signers of the bundle will be ignored. A resource or class will also be ignored if its digest is different from that of the signature file. If the certificate chain of a signature cannot be checked, e.g. the certificate authority of the certificate is not in the trusted list, the signature will be ignored.

The above text is not meant to limit the ways a bundle can be signed. If a bundle is signed using some other method, such as JAD signing, the whole bundle must have the same signature.

5.9 Bundle Permission Information

Bundles can convey the permissions they require using the file OSGI-INF/permissions/bundle.perm. Each permission is listed on its own line using the encoded form of PermissionInfo. Any blank lines and lines beginning with '# 'or '//' will be ignored. If this file is present, the ProtectionDomain of the bundle will be the intersection of the permissions listed in this file and the permissions assigned to the bundle through the PermissionAdmin and ConditionalPermissionAdmin services.

When a permissions.perm file is present, the permission evaluation will take place in two stages:

1. The permission check will first be made against the permissions listed in the permissions.perm.
2. Assuming the previous check succeeds, the checks listed in 5.7 will be evaluated.

An attacker can circumvent the above logic by simply removing the permissions.perm file from the bundle. This would remove any permission scoping that may be required by a signer of the bundle. To prevent this type of attack the framework detects the condition by checking if the permissions.perm file was signed. If it was signed and the permissions.perm file does not exist, the framework knows that the permissions.perm file was removed after signing. If the bundle is being installed when this condition is detected, the install will fail with a BundleException. Otherwise, this condition will result in a framework error event and the bundle will not move into RESOLVED state.

6 Java API

All the classes listed in this section will be in the org.osgi.service.condpermadmin.

6.1 Interface Condition

public interface **Condition**

This interface is used to implement Conditions that are bound to Permissions using ConditionalPermissionInfo. The Permissions of the ConditionalPermissionInfo can only be used if the associated Condition is satisfied.

Method Summary

`boolean`	**`isEvaluated`**`()` This method returns true if the Condition has already been evaluated, and its satisfiability can be determined from its internal state.
`boolean`	**`isMutable`**`()` This method returns true if the satisfiability may change.
`boolean`	**`isSatisfied`**`()`

All Page Within This Box

	This method returns true if the Condition is satisfied.
boolean	**isSatisfied**(Condition[] cond, Dictionary context) This method returns true if the set of Conditions are satisfied.

Method Detail

6.1.1 isEvaluated

`public boolean` **`isEvaluated`**`()`

This method returns true if the Condition has already been evaluated, and its satisfiability can be determined from its internal state. In other words, isSatisfied() will return very quickly since no external sources, such as users, need to be consulted.

6.1.2 isSatisfied

`public boolean` **`isSatisfied`**`()`

This method returns true if the Condition is satisfied.

6.1.3 isMutable

This method returns true if the satisfiability may change. If this method returns false, isEvaluated() must return true.

6.1.4 isSatisfied

`public boolean` **`isSatisfied`**`(Condition[] conds, Dictionary context)`

This method returns true if the set of Conditions are satisfied. Although this method is not static, it should be implemented as if it were static. All of the passed Conditions will have the same type and will correspond to the class type of the object on which this method is invoked.

6.2 Interface ConditionalPermissionAdmin

---

public interface **ConditionalPermissionAdmin**

This is a framework service that allows ConditionalPermissionInfos to be added to, retrieved from, and removed from the framework.

Method Summary

ConditionalPermissionInfo	**addCollection**(ConditionInfo[] conds, org.osgi.service.permissionadmin.PermissionInfo[] perms) Adds a set of permissions to the framework that are enabled by the corresponding Conditions.
java.util.Enumeration	**getCollections**() Returns the PermissionInfos for the Permission in this ConditionalPermissionInfo.

All Page Within This Box

java.security.AccessControlContext	**getAccessControlContext**(String[] signatures) Returns the AccessControlContext that corresponds to the signature chain.

Method Detail

6.2.1 addCollection

public ConditionalPermissionInfo **addCollection**(ConditionInfo[] conds,

org.osgi.service.permissionadmin.PermissionInfo[] perms)

Adds a set of permissions to the framework that are enabled by the corresponding Conditions.

6.2.2 getCollections

public java.util.Enumeration **getCollections**()

Returns the PermissionInfos for the Permission in this ConditionalPermissionInfo.

6.2.3 getAccessControlContext

public java.security.AccessControlContext **getAccessControlContext**(String[] sigchain)

Returns the AccessControlContext that corresponds to the signature chain. This method will find the ConditionalPermissionInfos that consist solely of BundleSignerConditions. It will then evaluate those Conditions as if it was for a bundle signed with the given signature chain. The returned AccessControlContext will consist of the permissions that correspond to the satisfied BundleSignerConditions

---

6.3 public interface ConditionalPermissionInfo

This interface describes a binding of a set of Conditions to a set of Permissions. Instances of this interface are obtained from the ConditionalPermissionAdmin service. This interface is also used to remove ConditionalPermissionInfos from ConditionPermissionAdmin.

Method Summary

void	**delete**() Removes the ConditionalPermissionInfo from the ConditionalPermissionAdmin.
ConditionInfo[]	**getConditionInfos**() Returns the ConditionInfos for the Conditions that must be satisfied to enable this ConditionalPermissionInfo.
PermissionInfo[]	**getPermissionInfos**() Returns the PermissionInfos for the Permission in this ConditionalPermissionInfo.

All Page Within This Box

Method Detail

6.3.1 getConditionInfos

`public ConditionInfo[] getConditionInfos()`

Returns the ConditionInfos for the Conditions that must be satisfied to enable this ConditionalPermissionInfo.

6.3.2 getPermissionInfos

`public org.osgi.service.permissionadmin.PermissionInfo[] getPermissionInfos()`

Returns the PermissionInfos for the Permission in this ConditionalPermissionInfo.

6.3.3 delete

`public void delete()`

Removes the ConditionalPermissionInfo from the ConditionalPermissionAdmin.

6.4 Class ConditionInfo

```
java.lang.Object
  |
  +--org.osgi.service.condpermadmin.ConditionInfo
```

---

public class **ConditionInfo**
extends java.lang.Object

Condition representation used by the Conditional Permission Admin service.

This class encapsulates two pieces of information: a Condition *type* (class name), which must implement `Condition`, and the arguments passed to its constructor.

In order for a Condition represented by a `ConditionInfo` to be instantiated and considered during a permission check, its Condition class must be available from the system classpath.

Constructor Summary

ConditionInfo(java.lang.String encodedCondition) Constructs a ConditionInfo object from the given encoded ConditionInfo string.	
ConditionInfo(java.lang.String type, java.lang.String[] args) Constructs a ConditionInfo from the given type and args.	

Method Summary

boolean	**equals**(java.lang.Object obj) Determines the equality of two ConditionInfo objects.

All Page Within This Box

java.lang.String[]	**getArgs**() Returns arguments of this ConditionInfo.
java.lang.String	**getEncoded**() Returns the string encoding of this ConditionInfo in a form suitable for restoring this ConditionInfo.
java.lang.String	**getType**() Returns the fully qualified class name of the Condition represented by this ConditionInfo.
int	**hashCode**() Returns the hash code value for this object.
java.lang.String	**toString**() Returns the string representation of this ConditionInfo.

Methods inherited from class java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Constructor Detail

6.4.1 ConditionInfo

```
public ConditionInfo(java.lang.String type,
                     java.lang.String[] args)
```

Constructs a ConditionInfo from the given type and args.

Parameters:

type - The fully qualified class name of the Condition represented by this ConditionInfo. The class must implement Condition and must define a constructor that takes a Bundle and the correct number of argument strings.

args - The arguments that will be passed to the constructor of the Conditon class identified by type.

Throws:

java.lang.NullPointerException - if type is null.

6.4.2 ConditionInfo

```
public ConditionInfo(java.lang.String encodedCondition)
```

Constructs a ConditionInfo object from the given encoded ConditionInfo string.

Parameters:

encodedCondition - The encoded ConditionInfo.

Throws:

java.lang.IllegalArgumentException - if encodedCondition is not properly formatted.

See Also:

getEncoded()

Method Detail

6.4.3 getEncoded

`public final java.lang.String` **`getEncoded`**`()`

Returns the string encoding of this `ConditionInfo` in a form suitable for restoring this `ConditionInfo`.

The encoding format is:

```
[type "arg0" "arg1" ...]
```

where *argX* are strings that are encoded for proper parsing. Specifically, the `"`, `\`, carriage return, and linefeed characters are escaped using `\"`, `\\`, `\r`, and `\n`, respectively.

The encoded string must contain no leading or trailing whitespace characters. A single space character must be used between type and "*arg0*" and between all arguments.

Returns:
The string encoding of this `ConditionInfo`.

6.4.4 toString

`public java.lang.String` **`toString`**`()`

Returns the string representation of this `ConditionInfo`. The string is created by calling the `getEncoded` method on this `ConditionInfo`.

Overrides:
`toString` in class `java.lang.Object`

Returns:
The string representation of this `ConditionInfo`.

6.4.5 getType

`public final java.lang.String` **`getType`**`()`

Returns the fully qualified class name of the Condition represented by this `ConditionInfo`.

Returns:
The fully qualified class name of the Condition represented by this `ConditionInfo`.

6.4.6 getArgs

`public final java.lang.String[]` **`getArgs`**`()`

Returns arguments of this `ConditionInfo`.

Returns:
The arguments of this `ConditionInfo`. have a name.

6.4.7 equals

`public boolean` **`equals`**`(java.lang.Object obj)`

Determines the equality of two `ConditionInfo` objects. This method checks that specified object has the same type and args as this `ConditionInfo` object.

Overrides:
`equals` in class `java.lang.Object`

All Page Within This Box

Parameters:
`obj` - The object to test for equality with this `ConditionInfo` object.
Returns:
`true` if `obj` is a `ConditionInfo`, and has the same type and args as this `ConditionInfo` object; `false` otherwise.

6.4.8 hashCode

`public int` **`hashCode`**`()`

Returns the hash code value for this object.
Overrides:
`hashCode` in class `java.lang.Object`
Returns:
A hash code value for this object.

7 Considered Alternatives

In this section we discuss solutions that were considered, but reject after discussion.

7.1 Partial Bundle Signing

Java2 Jar file signing allows for bundles to be partially signed. This means that classes in a same bundle may have different protection domains. Unfortunately, as pointed out earlier, the framework makes many access decisions based on bundle level permission checks rather than class level checks. More details on this can be found in the security section.

We decided against partial bundle signing because of the complications that it introduced. The biggest complication is that of breaking the assumption of a single protection domain per bundle. We also ran into security and implementation issues. This section outlines some of the things we were thinking.

Another security exposure is that trusted code may be mixed with untrusted code in the same bundle. A malicious bundle programmer can extract signed classes signed by a trusted signer and add them to his bundle. Even if the bundle is run with no permissions, when the signed classes are run, they will have the more trusted permissions that correspond to the signer. Unfortunately, this trusted class is controlled by the malicious bundle. It's possible that bugs or poor design decisions made by the programmer of the trusted class can be exploited by the malicious bundle to coerce it into doing privileged operations.

To prevent this exposure we require that the permissions of the classes of a bundle be a subset of the permissions of the Manifest.

Signatures of elements of an embedded Jar file come from two sources. First, if the Jar file itself is signed, then the signature will apply to all of the elements of the Jar file. Second, if there are signature files in the embedded jar file, those signatures will also be evaluated and assigned.

7.1.1 Framework changes

We also need to clarify how some of the framework methods are affected. First, we also need to redefine the Bundle.hasPermission() method. Since a bundle may be partially signed, we need to figure out which ProtectionDomain to use to represent the bundle. The obvious candidates are the ProtectionDomain indicated by the bundle location and signer of the BundleActivator and the Manifest. Since not every bundle may have a BundleActivator, we propose that the Manifest be used.

7.1.2 Exploiting bundle permissions

If partial bundle signing is allowed, code that doesn't have permission to access certain things that are evaluated on a bundle level, such as package and bundle import, will still be able to access them. When the code in the unsigned class file runs, it will not run with the same permissions as the rest of the bundle, but it will still have access to resources and classes imported by the bundle.

This attack requires Trojan type behavior since it would need to fool a bundle programmer into including the untrusted class into the bundle. Thus programmers will need to be aware that even untrusted code will get some extra permissions by virtue of being in a bundle.

7.2 Code Source

Rather than using our own PermissionQualifier class, we could have used CodeSource. Unfortunately, even though CodeSource is part of the minimum execution environment, it would drag a lot of other classes with it if it was used because if the many classes associated with the Cerficate class and X509 certificates. In reality all we need is a string to represent the signers, so dragging in the extra classes would just be a waste. We also would not get the ability to use filters to select signers for permission assignment. Also, CodeSource does not have a concept of a Principal, so subclass to extend that concept would be needed in the future to incorporate Principals. The biggest problem is that CodeSource also requires the location information to be in URL format. Even though the location string is often a URL, there is no requirement for the location string to be a valid URL.

7.3 Signature verification and certificate provisioning

Signatures and certificates are useful outside of PermissionAdmin, so it would be nice to come up with a more general solution. The Java2 solution is to use a Keystore, but that pulls in a large API. In the most basic scenario, we simply need to provision Certificate Authority certificates. Signature verification could then be done by the framework using these certificates. The provisioning could be accomplished with the addition of two simple methods:

```
void setCACertificates(String certs);

String getCACertificates();
```

The Strings used in each method could be Base64 encoded certificates.

While such an API would provision the certificates, it doesn't fit with some of the deployment scenarios that are possible. For example, some deployments will use smart cards. These cards may have built-in certificates and often do signature verification on the card. Other deployments will have specific certificate distribution schemes that may not fit with such a simple API.

All Page Within This Box

8 Security Considerations

This RFC specifies how permissions are assigned in the framework. Therefore, the mechanisms outlined must be implemented and managed with care to prevent permission assignments by untrusted bundles. Condition classes come from the system and framework classpath. We are making the assumption that these classes are trusted and therefore will behave correctly. Malicious classes on the system or framework classpath can compromise the system if used as Conditions in any ConditionalPermissionInfo.

Because the BundleSymbolicName is set by the developer and not validated by the framework the BundleSymbolicNameCondition should only be used with another validated Condition such as BundleSignerCondition. In that case, the joint use implies that for the permission set under evaluation the signer is authorized to use the BundleSymbolicName.

9 Document Support

9.1 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]. MEG Policy requirements – RFP 55

[4]. Enhanced Modularity Support – RFC 79

[5]. Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names, RFC 2253, IETF.

9.2 Author's Address

All Page Within This Box

Name	Benjamin Reed
Company	IBM
Address	650 Harry Rd, San Jose, CA 95037
Voice	408 927 1811
e-mail	breed@almaden.ibm.com

9.3 Acronyms and Abbreviations

9.4 End of Document