A Generic XACML Based Declarative Authorization Scheme for Java

Architecture and Implementation

Rajeev Gupta and Manish Bhide

IBM India Research Lab, Block 1, IIT Delhi, India, +91-11-26861100
{grajeev, abmanish}@in.ibm.com

Abstract. Security and authorization play a very important role in the development, deployment and functioning of software systems. Java being the most popular platform for component-based software and systems, Java security is playing a key role in enterprise systems. The major drawback in the security support provided by J2EE and J2SE is the absence of a standard way to support instance level access control. JAAS does provide some help, but it is not without its share of problems. The newest standard related to security - XACML, provides a standard simple way to represent security policies. In the paper we propose a unique way to extend JAAS technology so that it can support class-instance level access control in a declarative manner. We then showcase how this extension can be molded in the XACML architecture, thereby providing an end-to-end standard based access control specification and implementation for J2SE and J2EE applications. The major advantage of our technique is that, being declarative it does not require any change to the security code when - either the security policies are changed or the security infrastructure is deployed in a new environment.

1 Introduction

The exponential growth of e-commerce in the recent past has lead to a proportional increase in the complexity of software systems. This complexity has also lead to an increase in security and authorization needs of enterprise applications. In order to deal with this complexity, various proprietary and application specific languages [1, 2, 3] that help in specifying access control policies of enterprise systems have been proposed. XACML is one such general-purpose access control policy language, which in addition to being a standard, is generic, distributed and powerful [4]. It provides an XML based access control policy language as well as an access control decision request/response language, which can be used by applications and systems to fulfill their access control needs. The XACML specification deals with the framework and the exact implementation details of the access control engine are left for the implementers.

Java is the most popular platform for component based software and has played a key role in the popularity of e-commerce applications. Java has its own standardized mechanism to provide user-based security and access control called, Java Authentication and Authorization service (JAAS). JAAS has played a key role in securing these enterprise applications. The advancements in enterprise applications have lead to rapid changes in the requirements and needs of the software developer. The Java language has tried to keep pace with these needs by adding new features such as Data Access Objects [6], remote monitoring and management of JVM [5], class data sharing, generic types etc. But fulfilling the security needs of Java applications is still closely tied with application code [13], leading to an ad-hoc, application specific development of security and access control implementations.

In the J2EE architecture, providing authentication and access control is delegated to the application server. A declarative XML based mechanism is used to specify the access control needs of the J2EE applications. But such an access control can be provided only at a method-level granularity. The state or logic of the software object/component does not factor into the access control decision. This is very restrictive and policies such as: *"Employees can only view their own salaries from the salary database";* are implemented programmatically.

This paper tries to bridge the gap between XACML based specification of access control needs and standard security implementation for Java and outlines an XACML implementation for Java applications. The XACML implementation proposed in this paper provides a generic and declarative mechanism for providing access control in Java applications. Our technique uses an innovative extension of JAAS to attain our objectives. Thus, the contributions of this paper include:

– We propose a standard based implementation of XACML for Java using an innovative extension of JAAS. In other words, we show how XACML and JAAS can co-exist thereby providing end-to-end standards based access control specification and implementation for Java language.
– Our technique provides a mechanism for supporting instance level authorization in Java applications using declarative specifications.
– We provide a method of writing declarative security policies for Java applications

The rest of the paper is organized as follows: A brief introduction to XACML architecture is given in section 2. Section 3 gives an overview of Java security and JAAS, explaining their deficiencies in providing fine-grained access control. Section 4 outlines how JAAS can be extended so as to provide declarative authorization support in Java while using XACML standards. Related work is discussed in Section 5 and Section 6 concludes the paper.

2 XACML

One of the basic reasoning for the development of XACML was the need to have a standard, generic and powerful access control specification language. Existence of various proprietary languages provided piecemeal solutions to security issues

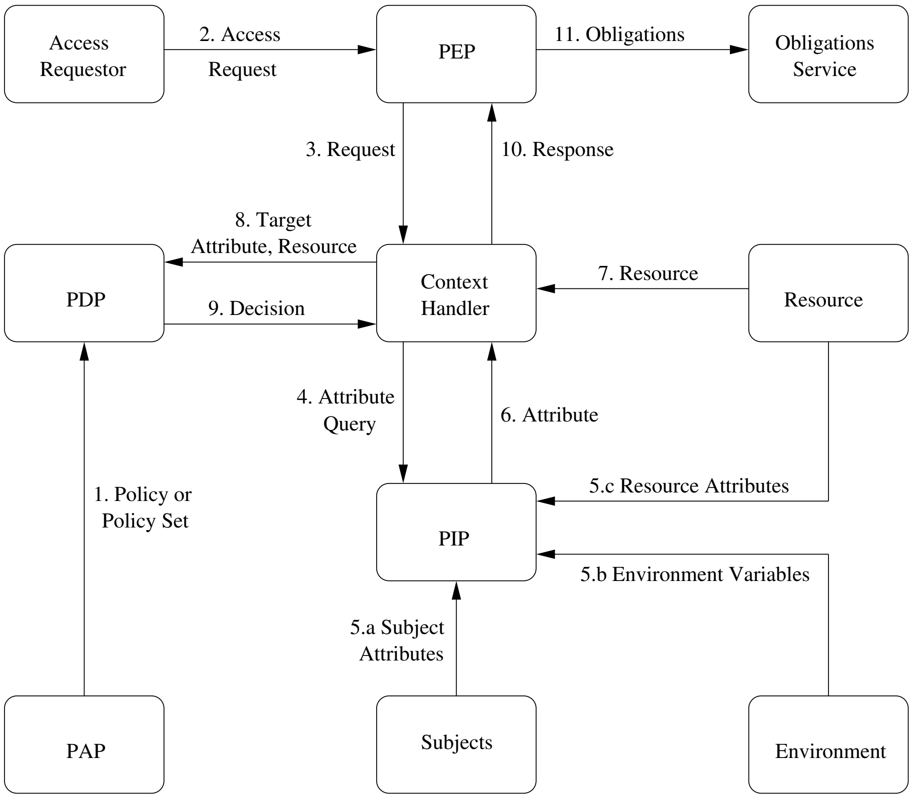

Fig. 1. Data flow diagram for XACML

of the enterprise. XACML tries to bridge this gap and provides a common language for expressing security policy across the enterprise. It allows the enterprise to manage the enforcement of all the elements of its security policy in all the components of its information systems. XACML is generic thus it can be used in various environments. As XACML is used across various components of the enterprise application, management of the access control policy becomes easier.

The data flow diagram of Figure 1 shows the major actors in the XACML domain, which are important for our approach. Main components of this architecture are:

1. **PEP:** The Policy Enforcement Point is the system entity that performs access control by making decision request and enforcing authorization decisions. This is the entry point for access control infrastructure. Consider an example of an e-commerce application, which supports auctions. In this application, each request (to bid, to create an auction etc.) made by the user will be routed through a software component, which will send it out to an entity responsible for making the access decision (allow/deny). Such an entity, which makes a callout to the access control component, is the PEP of the application.

2. **PDP:** This is the core of the framework and is responsible for evaluating the applicable access control policies and to render the authorization decision as one of four values: *permit, deny, indeterminate or not applicable.* In the above example, the entity which makes the decision to allow or deny the request is the PDP. The PDP will evaluate the access control rules, which could include policies like *"Only a user who has created the auction should be able to modify it"*.
3. **PIP:** This component acts as a source of attribute values. The responsibility of this component is to provide all the information that might be required by the PDP to make the access decision. Attributes are characteristics of the resource being managed (e.g. *who is the owner of auction bid*), the user making the request, action (*read or modify*) or environment (*office hours*).
4. **Context Handler:** This is a system entity that converts decision requests in the native request format to the XACML canonical form and converts authorization decisions in the XACML canonical form to the native response format. The context handler allows the use of XACML in a variety of application environments. This component is one of the key components of our solution, as it allows the JAAS specific data to be converted into decision requests, which are understood by the PDP.

The context handler forms the access control request based on the attributes of the requester, action, resource and environment. This information is provided to the PDP to find the access control policy applicable for the request. The access control policy is defined in terms of the attributes of the requester, action, environment and resource. The policy can also include functions defined on these attributes. The PDP performs the following two operations to arrive at a decision: (1) It first tries to find all the policies applicable for the request and (2) then it evaluates these policies and returns the decision back to the PEP (via the context handler). PAP is the policy administrative point, responsible for managing access control policies. The curious reader is requested to refer to [4] for further details about XACML. We now explain why standard JAAS based access control is not sufficient for enterprise applications, and how JAAS can be extended to overcome its drawbacks.

3 Java Security and JAAS

3.1 Java 2 Security

Java 2 uses policy based security architecture. The security policy (Figure 2) is defined by a set of permissions for code in various locations (*codeBase*) and by various signers. These permissions allow certain actions on a certain resource. Resource names and their associated actions are enumerated in a policy file. In Java 2, *AccessController* is used as the security enforcer. The example policy file in Figure 2 gives all permissions to the jar files present in the *${java.home}/lib/ext* directory, whereas read permissions on some system properties, are given to all the other code bases.

```
grant codeBase "file:${java.home}/lib/ext/*" {
   permission java.security.AllPermission;
};

grant {
   permission java.util.PropertyPermission "os.name", "read";
   permission java.util.PropertyPermission "os.version", "read";
   permission java.util.PropertyPermission "os.arch", "read";
 };
```

Fig. 2. Policy file in Java

```
SafeFileWriter() {
   Permission perm = new java.io.FilePermission("foo.txt", "write");
   AccessController.checkPermission(perm);
   // Write to foo.txt
}
```

Fig. 3. Protecting a method using AccessController

The example method in Figure 3 shows the typical way in which a protected resource is accessed using Java methods. Before performing the operation, the method calls the *AccessController* with the permissions required to perform operations on the resource being accessed. The *AccessController* checks the requested permission with the application's current authorization policy. If any permission defined in the policy file implies the requested permission, the method *checkPermission* simply returns; otherwise an *AccessControlException* is thrown. The *SafeFileWriter* is used for writing in the file *foo.txt.* Thus before writing, the *AccessController* is called to check whether writing to the file is allowed. The major drawback of Java 2 security is that it does not have user, role or object based permissions. The user based access control is added in Java 2 using JAAS, which is explained next.

3.2 Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) is a set of APIs that enable Java applications to authenticate and enforce access controls upon users. JAAS reliably and securely determines who is currently executing the Java code and whether the user is allowed to do so. JAAS adds subject-based policies to the Java 2 security model. For this the user is first authenticated and the *javax.security.auth.Subject* class is used to encapsulate the credentials of the authenticated user. A *Subject* can have multiple identities called *Principals.* In a JAAS policy file, each grant statement is associated with a *Principal.* For each *Principal* associated with the *Subject*, the *AccessController* gets permissions

```
grant codebase "file:./MyAction.jar",
     Principal sample.principal.ExamplePrincipal "Bob" {
          permission java.io.FilePermission "max.txt", "read";
};
```

Fig. 4. Principal based authorization in JAAS

from the policy file and checks whether any permission implies the requested permission. Otherwise, it throws an *AccessControlException.* Figure 4 shows a typical authorization policy file in JAAS. It allows Java classes in *MyAction.jar* to read *max.txt* if the particular class is accessed by *Bob.* All other users are not allowed to access the resource.

The standard policy file format of JAAS does not support security policies that are based on the properties of the application object on which the policy is defined. This precludes the definition of policies such as *"A manager is allowed to edit the salary information only of his direct reports"*. Using JAAS, if a user (manager) is allowed to call method *editSalary* then the user is allowed to edit salaries of all the employees irrespective of whether the employee is his/her direct report or not. Such policies are very common in any application and custom code is required to enforce such policies. However, JAAS does recognize such needs and it provides mechanisms to extend its standard interfaces to suite the client needs. Possible extensions to JAAS are explained next.

3.3 JAAS Extensions

JAAS is build on top of the pre-existing security model of Java, which depends on the *codeBase* accessing the resource and uses the plaintext format policy file implementation. JAAS makes authorization decision based on the *Subject* who is performing the action, the action being performed and the resource being accessed. Thanks to pluggable-features of JAAS, writing custom authentication and authorization sub-modules can change its default behavior. In this section we explain possible extensions which can be used for XACML implementation using JAAS. In order to support instance level access control in JAAS the following JAAS artifacts can be changed:

- *java.security.Principal*: The *Principal* interface represents the abstract notion used to represent an entity such as an individual, an organization, a group or a login id. By extending the *Principal* one can add custom properties which can be used for authorization.
- *java.security.Permission*: The *Permission* class is used in two places namely, (1) The policy file where it represents the permissions given to a user on a *codeBase* and (2) the permission object which is constructed in the code before accessing a resource. In the code the object represents the permission required by the code for accessing a resource. If the policy file grants the requested permission to the *codeBase*, then the action is allowed by JAAS.

As explained in Figure 3, the *AccessController* calls *checkPermission* to know whether the caller has authority to perform the requested action. By default the *Permission* object can specify things like name of the permission (which may indicate the resource on which access is required), action for which the resource is accessed, etc. The *Permission* class implements the *implies* method, which takes as input another *Permission* object. This method is called by the *AccessController* to know whether the requested permission (present in the code) is implied by any permission present in the policy file.

- o *java.security.PermissionCollection*: This abstract class is used for representing a collection of *Permission* objects. This class can be implemented to have the desired way of storing the granted *Permission's* and comparing them with the requested *Permission.*
- o *java.security.Policy*: It is an abstract class for storing security policies in Java application environment. The *AccessController* contacts the *Policy* implementation to get the set of permissions defined in the policy file for an authenticated *Subject* on a *codeBase.* By-default the *Policy* class is extended by the *PolicyFile* class to read the policy file as depicted in Figure 4. The *Policy* class has *getPermissions* method, which parses the policy file and returns an appropriate *PermissionCollection* object enumerating the permissions of the *codeBase* for the calling *Subject.*

From the above discussion it is fairly clear that JAAS can be extended in a variety of ways to attain various authentication and authorization objectives. But for these extensions one needs to write code to implement or extend various JAAS interfaces and classes respectively. Writing new code, whenever there is change in security requirements, is cumbersome and makes the code difficult to maintain. Further, it precludes the possibility of changing the security settings at deployment time thereby preventing the reuse of code across different domains.

Hence there is clearly a need for a security approach, which is flexible, standard based and which gracefully handles the extension or changes in the security policy without requiring changes to the security code. The key points to be considered while addressing these problems are:

1. JAAS being a Java security standard, the solution should adhere to JAAS security framework
2. The solution should provide fine grained (instance based) access control
3. The solution should enable the modification of the security policy without requiring any change to the security code.

XACML provides a representation of fine-grained security policies across the enterprise. This motivates us to explore whether the marriage of these two technologies/standards can be a solution to our problems? As it turns out, this indeed is the case. The next section outlines our proposed extension to JAAS, which allows XACML to be used with Java applications while having standard based implementation.

4 Extending JAAS for XACML Implementation

In the last section we listed the requirements which JAAS based implementation of XACML should meet. In this section we present our XACML implementation for Java, which is generic and declarative. It enables changes in security settings without writing any new code. It is assumed here that the resource being protected is accessed through various Java methods with different methods performing different actions on the resource. In Section 4.1 we present our extensions to JAAS which are required to support generic authorization. Section 4.2 explains the code-flow between a user making an access request and the *AccessController* returning a response. Section 4.3 deals with the mapping between XACML and our unique JAAS extension.

4.1 Generic Authorization Using JAAS

To attain the objectives of providing generic and declarative authorization we propose a technique that modifies JAAS in a unique way so that its extension can be written in a declarative manner rather than the conventional programmatic way. Following are the extensions that we have implemented to the standard JAAS classes/interfaces described in the previous section:

1. *GenericPermission*: It extends the standard *Permission* class of JAAS. For implementing attribute level authorization, as mandated by XACML, the class instance (object) representing the resource on which access is requested, needs to be passed to the *Permission* object. The *GenericPermission* has a constructor that takes the object (on which access is requested) as input. The *implies* method of *GenericPermission* is written in such a way that it takes into consideration the attributes of the action, the attributes of the object and the environment variables for deciding whether the granted permission implies the requested permission. The attributes of the resource object required for XACML implementation can be obtained by calling the getter methods on the object instance using Java reflection. This unique extension of the implies method acts as PDP component of the XACML architecture.
2. *GenericPolicy*: The core of our technique lies in the representation of the authorization framework in an XACML policy file. Our *GenericPolicy*, which is an extension of *java.security.Policy* class of JAAS, interprets the authorization policies, written in XACML language. The *getPermissions* method of *GenericPolicy* parses the XACML based policy file and retrieves all the *GenericPermission*'s granted to the specified *Subject* and *codeBase* in the policy file. The permissions are returned in the form of a *GenericPermissionCollection* which is explained next.
3. *GenericPermissionCollection*: This class is used to represent a collection of *GenericPermission* objects.

The access control policy is represented in terms of getter methods (for getting attributes of the resource object) defined on the application objects. The policy allows the use of expressions, which operate on the values returned by the getter methods. If the getter method(s) used in XACML policy are not implemented, the context handler (which is responsible for invoking the getter methods and calling the PDP) throws an exception. For example consider the policy – "*Only a user who has created the auction should be able to modify it*". In this policy, access control is based on *whether the caller is the owner of the auction object.* Thus, we pass the *auction* object to the *GenericPermission* constructor so that the *owner* of the auction can be obtained using *getOwner* method of the auction object.

4.2 Code Flow

Here we explain the steps required to protect a resource as well as the steps followed when a user wants to perform some action on the protected resource. Our technique assumes that all resource actions are implemented as methods, and hence whenever a user wants to perform some action on a resource, the corresponding method is accessed. Thus protecting a resource is equivalent to protecting methods performing some action on a Java object representing the resource. Each method, which is required to be protected, needs to start with the construction of *GenericPermission* object having three parameters:

1. The class to which the method belongs,
2. The action which the method wants to perform and
3. The resource object on which the method is called.

This *GenericPermission* object represents the permissions necessary to execute the method. At run-time, if the policy file grants this permission to the code and the user invoking the method, then the access will be allowed by JAAS. Figure 5 shows a method, which updates an auction object, creating the *GenericPermission* object. Then a call is made to the *AccessController* provided by Java. The *GenericPermission* object is passed as a parameter to the *checkPermission* call of the *AccessController.* This ensures that any user who does not have the permissions as indicated by the *GenericPermission* object will be thrown an *AccessControlException.* What follows next are the steps through which the *AccessController* determines permissions for an authenticated user.

```
updateAction() {
     Permission perm = new GenericPermission(String auction,
                              String update, Object auction1);
     AccessController.checkPermission(perm);
     // perform action
}
```

Fig. 5. Method protection using by GenericPermission

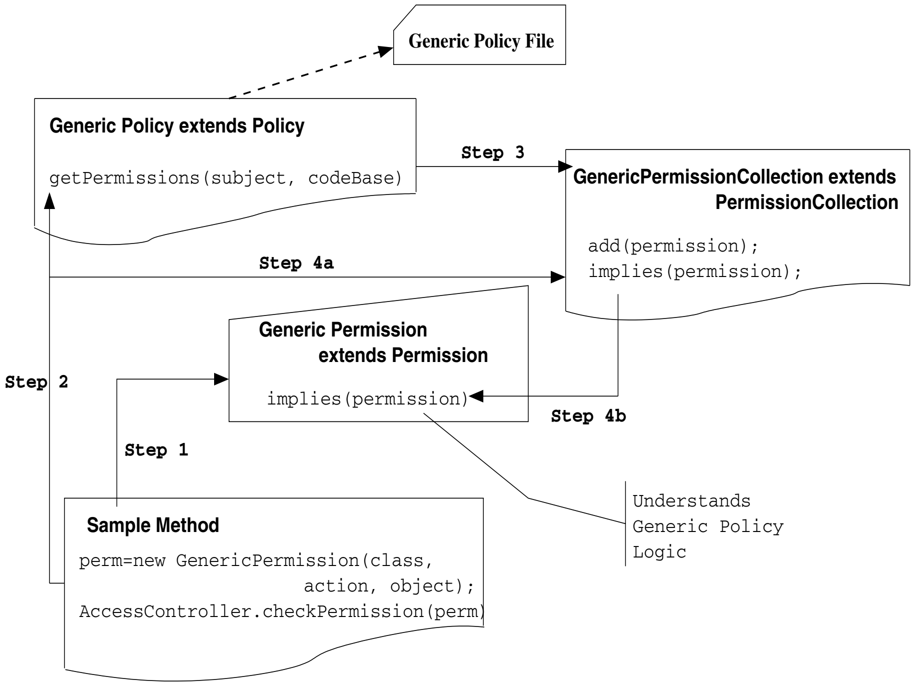
Generic Policy File
Generic Policy extends Policy
getPermissions(subject, codeBase)
Step 3
GenericPermissionCollection extends PermissionCollection
add(permission);
implies(permission);
Step 4a
Generic Permission extends Permission
implies(permission)
Step 2
Step 4b
Step 1
Understands Generic Policy Logic
Sample Method
perm=new GenericPermission(class, action, object);
AccessController.checkPermission(perm)

Fig. 6. Code Flow for Generic Authorization

Figure 6 shows the code flow of an authorization decision using our generic authorization scheme. In Step 1, a *GenericPermission* object is created. As mentioned earlier, this *GenericPermission* object represents the permission that is necessary to execute the method i.e., to perform the action on the resource. This permission object is then passed as a parameter to the *checkPermission* call of the *AccessController*. The *AccessController* uses the Policy implementation of JAAS to make the authorization decision. The standard Policy implementation of JAAS (*PolicyFile*) cannot understand the XACML policy file. We have extended the *Policy* implementation (called *GenericPolicy*) and our implementation is equipped to handle the XACML format. The Policy implementation to be used by the *AccessController* is specified using the *auth.policy.provider* parameter in the *java.security* file. Changing this parameter setting ensures that the JVM uses the *GenericPolicy* for evaluating the authorization decision instead of the normal PolicyFile provided by JAAS.

The *GenericPolicy* class finds the granted permissions for the given *Subject* and *codeBase* by parsing the XACML file in the *getPermission* method (Step 2). In other words, if a user authenticated as "*Foo*" is accessing some code (in *"abc.jar"*), which is trying to perform an action on an object (resource), then the *getPermissions* method will try to find all the permissions that are given to *"Foo"* for *codeBase "abc.jar"* in the XACML policy file. The *getPermission* method returns *GenericPermissionCollection* which is a set of *GenericPermis-*

sion objects given to the *Subject* ("Foo") for the given *codeBase* ("abc.jar") in the XACML policy file (Step 3).

After getting the *GenericPermissionCollection* corresponding to the calling *Subject* and *codeBase*, the *AccessController* calls the implies method of the *GenericPermissionCollection* (Step 4a). This method has one parameter which is the *GenericPermission* Object constructed in Step 1. The logical meaning of this is to find if any one of the permissions in the *GenericPermissionCollection* implies the requested permission (which is constructed in Step 1). In order to do this, the *implies* method of the *GenericPermissionCollection* iterates through each of the constituent *GenericPermission*'s in the collection. For each of these *GenericPermission*'s it calls the *implies* method of the *GenericPermission* class (Step 4b). The *implies* method of *GenericPermission* understands the semantics of the declarative authorization policy specified in the XACML policy file. Based on the authorization policy specified in the XACML file, it checks if the requested permission (constructed in Step 1) can be implied from the granted permission (which is represented by the *GenericPermission* object itself on which the implies is called). If the *AccessController* finds that any one of the *GenericPermission.implies* returns true, then it simply returns. If the requested permission is not granted by any policy present in the XACML file or if is not implied by any of the policies, then an *AccessControlException* is thrown.

The *implies* method of *GenericPermission* uses the resource object instance to get the values of attributes by calling the getter method using Java reflection technology. These values are used in the expressions defined on the object attributes as well as environment variables to decide on the imply relationships. For e.g., consider an authorization policy "*Only a gold user is allowed to access critical data between 10AM and 4 PM*". In this policy, the application object, say an entity bean, will have a getter method that returns the criticality (high/low etc.) of the data and the policy will be expressed in terms of the return value of this getter method. If this policy is later changed to "*Only gold and silver customers are allowed to access stock history data*", then such a change will only require a minor change in the XACML policy and no change in the code.

```
grant codebase "file:./MyAction.jar",
   Principal sample.principal.GenericPrincipal "GoldCustomer" {
        permission com.ibm.jaas.GenericPermission
                                Object="StockInfo" action="read";
        CompoundCondition operator="AND" name="C1" name="C2";
        Condition name="C1" type="method" mName="getType"
                         operator="equals" value="confidential";
        Condition name="C2" type="environmentValue"
             mName="getTime" operator="between" lower="10AM"
                                                  upper="4PM";
 };
```

Fig. 7. XACML based sample JAAS extension policy file

However, using conventional JAAS implementation, even this minor change will require a change in the security code of the application. Thus a clever use of Java reflection and the unique representation of the XACML based authorization policy file, allow us to support fine-grained access control, which can be changed without warranting any change to the authorization code. A sample XACML based JAAS policy is given in the appendix. In order to highlight the difference between the normal JAAS and our extended JAAS implementation, Figure 7 shows the JAAS generic policy represented in non-XACML format. It should be noted that we use an XACML based JAAS policy file (given in appendix), but the non-XACML format (Figure 7) is given for illustrating the differences between the normal and extended version of JAAS.

4.3 JAAS and XACML

In this section we explain how our implementation fits into XACML architecture. The policy in XACML can either deny or permit an action on a resource. The parallel of a policy in JAAS is a *permission*, which as the name suggests, only provides closed policy authorizations [16]. Hence our framework consists of permit policies only. Extending the architecture for other response alternatives of XACML (open authorization policy) needs a different implementation of *Policy* which is part of our future work. In Java, the *AccessController* handles access requests, thus it acts as the PEP for Java applications. When the PEP makes an evaluation request to the PDP of the XACML architecture, the PDP first tries to find out the policies that apply for the given target. Then it evaluates the applicable policies for making access decisions. The *getPermissions* method of *GenericPolicy* is similar to the initial work done by the PDP. The implies method of the *GenericPermission* class consists of an engine that can do evaluation of logic expressions. It uses the values of the various (resource object, environment etc.) attributes provided by the context handler (using Java reflection) to evaluate the authorization decisions. Thus it does the second function of PDP. The context handler is responsible for converting application specific objects to an XML format which is understood by the PDP. In our extension of JAAS, the context handler does the reflection on the application object. Figure 8 summarizes how our extension fits into XACML architecture. The major steps of the flow are:

– When an access request is made, the PEP calls the *GenericPolicy.getPermissions* method. As mentioned earlier, one of the functionality of the PDP is to find all the policies relevant to the given decision request. In the standard XACML framework, this functionality is not independently accessible from outside of PDP. In JAAS, the *GenericPolicy.getPermissions* does a similar work (of finding the relevant policies). Hence in our framework the *GenericPolicy.getPermissions* is part of the PDP and the *AccessController* (which is part of the PEP) calls this method. Thus we have externalized some of the functionality of the PDP. This is one of the extensions, which is required to the XACML framework to support our JAAS extension.

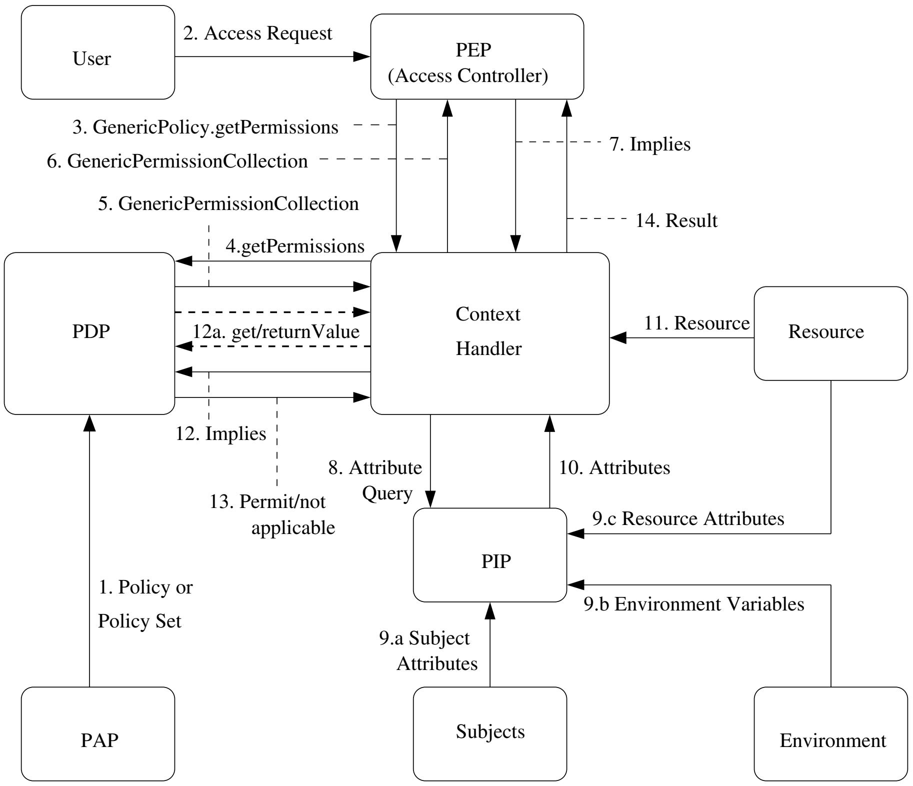

Fig. 8. XACML implementation using JAAS extensions

– Once the *GenericPermissionCollection* is received by the PEP, it calls the *implies* method on the *GenericPermissionCollection*. This is implemented in the PEP and it iterates over the *GenericPermssion*'s and calls the *implies* method of each of the constituent *GenericPermission* objects (Step 7).
– The context handler uses reflection to find out the values returned by the getter methods. It then constructs the decision request (which is an XML document) and sends it to the PDP (Step 12). The *GenericPermission.implies* method is the core of the PDP. It evaluates the policy/rule and returns its decision to the PEP.
– The XACML policy is represented in terms of expressions on the values returned by the getter methods of the resource object. Hence we need to represent the resource (Java) object in terms of an XML document. There are known techniques such as XStream [15] and SYS-CON [14] for converting Java objects as XML documents. Using these techniques, each property of the resource object can be easily represented in terms of an XPATH expression on an XML document. The values of the getter methods defined on the resource are obtained by the context handler (using Java reflection)

and sent out as part of the decision request. Invoking all the getter methods can have a lot of overhead. Hence the other option is for the PDP to call back the context handler (Step 12a) to obtain the value of the necessary getter methods. The information about the getter methods will be present in the policy file. The PDP will send this information to the context handler in Step 12a. E.g. the method to obtain the value of *owner-id* will be represented as *auctionEntry/auctionInfo/owner-id* in the policy file. This will be used by the context handler which will call the getter methods in a sequence (i.e. call *getAuctionInfo().getOwnerID()*) and will return the results to the PDP. For more information, please refer to the appendix.

It is to be noted that we can afford some flexibility in the XACML framework, but doing so in JAAS is not possible as the architecture and its data flow is coded inside the JVM. Our framework requires a minor change of externalizing the *getPermissions* API present in the PDP but it does not jeopardize the XACML architecture. Although we have given examples of resources being protected by Java methods, our scheme can be easily extended to J2EE environment. The various J2EE methods can be classified as actions on a resource (in which case the action name can be replaced by the method name). Whenever those methods are called on a Java object, the J2EE application server would have to create the *GenericPermission* object corresponding to the intended action on the intended Java object. The container would then invoke the *AccessController* before the actual method call. Thus our technique can fit seamlessly into a J2EE as well as J2SE environment.

5 Related Works

There are various attempts to represent the access control policies in XML format and for providing granular authorization. [1] deals with XML based access control specification for dynamic web services using role based access control (X-RBAC). [3] deals with a security mechanism that can support a wide range of security models and policies in an efficient and unified manner. These models include ACLs, lattice based access control models, etc. But their implementation is ad-hoc without focusing on any particular language. [7] provides a naive mechanism for management of security policies using XML in a distributed environment. It proposes a schema, which represents the Role based access control (RBAC) policies in XML. This XML policy file is interpreted using a standard API. The paper is very general and does not provide instance level access control, nor does it provide authorization using any standard such as JAAS. [12] proposes *dynFAF*, a constraint logic programming based approach for expressing and enforcing constraints. These constraints are evaluated at run-time. Our instance based access control also fits in that definition. We give the implementation of such a framework using Java. [8] explains how meta-programming can help in expressing and implementing security policies. It presents three different types of meta-object protocols (MOP). Compile time

MOP's reflect language constructs available at compile time. Load-time MOP's reflect on the byte-code using a modified class loader. Run-time MOP's use a modified version of JVM. Out of these three, the first two cannot support instance level access control, which is the central theme of our paper. Run time MOP's can provide instance level access control, but it requires changing the JVM, which is not required in our approach. In [9], a Java secure execution framework (JSEF) is presented which introduces higher-level abstraction for defining security policies. Using JSEF one can define permit as well as deny policies. It also provides support for security negotiation in the case of insufficient permissions at runtime. This paper is related to Java security but it does not give mechanism a to express conditional authorization based on object instance. It does not follow any standard as we do using JAAS and XACML. A UML based modeling language for specifying security requirement of an application is presented in [10]. The model is used to automatically generate access control infrastructure. Authorization constraints are represented using Object Constrained Language (OCL). Besides this, there are various other Java security implementations, which cater to some particular kinds of applications or platforms. [11] describes Java based security model used in IBM's WebSphere Commerce Suite (WCS). This paper deals with policy based access control by modeling relationship between business objects and users. But, the authorization code is embedded in the application and the implementation is not JAAS based. Our work is unique in the sense that using our standards based extensions, Java applications can get flexibility of programmatic authorization using declarative specification.

6 Conclusions

Java is one of the most popular languages for developing e-commerce and web applications. Java has evolved over time, but the surprising fact of the day is that Java security still leaves a lot to be desired. Even the basic class instance level access control cannot be supported in a standard way using Java. This has lead to the use of custom security code which is difficult to maintain and is prone to security errors. In this paper we have proposed an innovative technique that: (1) provides a declarative access control support for Java applications using an extension of JAAS, and (2) shows how the XACML framework can be used to cater to the needs of JAAS security policies. Our innovative extension of JAAS enables the declarative specification of the Java security, which can complement the security provided in J2EE as well as J2SE applications. This mechanism has the potential to reduce the re-engineering work, which is in the order of months to the order of hours and will also allow the specification of Java security policies using the XACML representation, thereby providing a consistent and standard way of representing security policies across the enterprise.

Acknowledgments. We would like to thank Neeran Karnik and Vishal Batra for their helpful comments and discussions.

References

[1] R. Bhatti, J. B. Joshi, E. Bertino, and, A. Ghafoor: "Access Control in Dynamic XML-based Web-Services with X-RBAC". First International Conference on Web Services, Las Vegas, June-2003.

[2] T. Fink, M. Koch, and C. Oancea: "Specification and Enforcement of Access Control in Heterogeneous Distributed Applications". International Conference on Web Services (ICWS), Germany, Sept-2003.

[3] Ungureanu, V. and N. H. Misnky: "Unified Support for Heterogeneous Security Polices in Distributed Systems". 7^{th} USENIX Security Symposium, Texas, Jan-1998.

[4] OASIS extensible Access Control Markup language (XACML). `http://www.oasis-open.org/committees`

[5] J2SE 5.0 in a nutshell. `http://java.sun.com/developer/technicalArticles/releases/j2se15`

[6] Core J2EE patterns, Data Access Object. `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`

[7] N. Vuong, G. Smith, and Y. Deng: "Managing security policies in a distributed environment using eXtensible markup language". The 2001 ACM Symposium on Applied Computing, Las Vegas, March-2001.

[8] J. Vayssiere: "Security and Meta Programming in Java". European Conference Object Oriented Programming - Workshop on Reflection and Meta-Level Architectures, France, May-2000.

[9] M. Hauswirth, C. Kerer, and R. Kurmanowytsch: "A Secure Exceution Framework for Java". 7^{th} ACM Conference on Computer and Communications Security, Greece, Nov-2000.

[10] T. Lodderstedt, D. Basin, and J. Doser: "SecureUML: A UML based Modeling Language for Model-Driven Security". Proceedings of UML'2002 - Unified Modelling Language, 5^{th} International Conference, Germany, Sept-2002.

[11] R. Goodwin, S.F. Goh, and F.Y. Wu: "Instance-level access control for business-to-business electronic commerce". IBM Systems Journal. Vol. 41, Number 2, 2002.

[12] S. Chen, D. Wijesekera and Sushil Jajodia: "Incorporating Dynamic Constraints in the Flexible Authorization Framework". 9^{th} European Symposium on Research in Computer Security (ESORICS'2004), France, Sept-2004.

[13] D. Wallach, D. Balfanz, D. Dean and E. Felten: "Extensible Security Architectures for Java". 16^{th} Symposium on Operating Systems Principles, France, Oct-1997.

[14] XML Serialization of Java Objects (SYS-CON). `http://www.sys-con.com/story/?storyid=37550\&DE=1`

[15] XStream: Java to XML Serialization and back again. `http://joe.truemesh.com/blog/000261.html`

[16] S. De Capitani di Vimercati, P. Samarati, and S. Jajodia: "Policies, Models, and Languages for Access Control". Workshop on Databases in Networked Information Systems, Japan, March-2005.

A Appendix

In this section we provide a sample security policy of JAAS represented in XACML format. The policy is for an e-auction site and is to ensure the following rule: "*Only the owner of an auction is allowed to modify the closing-date* *of an auction*". This is mapped to executing a method *"updateClosingDate"* on the Auction object. We first present the policy stated in XACML format. It consists of the following parts:

– The Policy
– The Rule

The policy can consist of multiple rules, which apply to the e-commerce site. In our case the policy consists of only a single rule, which is of "Permit" type. This means that if the rule fires then the subject is permitted to do the requested operation. The "Target" part of the policy decides the conditions under which this policy (and the rules in this policy) will be applicable.

The target of the rule decides the conditions under which the rule will be applicable. In this case the rule target states that it will apply to any subject who is trying to execute the *updateClosingDate* method of the Auction object. The condition part of the rule states the conditions that should hold true for the action to be permitted. In this rule, it is stated that the customer-id given in the request context should be equal to the owner-id of the auction that the user is trying to update.

```
<?xml version="1.0" encoding=‘‘UTF-8"?>
<Policy xmlns="urn:ibm:names:tc:xacml:1.0:policy"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:ibm:names:tc:xacml:1.0:policy
 http://www.ibm.com/irl/xacml/1.0/cs-xacml-schema-policy-01.xsd"
 PolicyId="identifier:example:JaasPolicy1"
 RuleCombiningAlgId="identifier:rule-combining-algorithm:permit-overrides">
<Description>
   JAAS based Access Control policies for an e-Auction Site.
</Description>
<Target>
   <Subjects>
       <AnySubject/>
   </Subjects>
   <Resources>
       <Resource>
           <!-- match document target namespace -->
           <ResourceMatch MatchId=
              "urn:ibm:names:tc:xacml:1.0:function:string-equal">
                <AttributeValue
                   DataType="http://www.w3.org/2001/XMLSchema#string">
                    file:./Auction.jar
                </AttributeValue>
                <ResourceAttributeDesignator AttributeId=
                   "urn:ibm:names:tc:xacml:1.0:resource:target-namespace"
                   DataType="http://www.w3.org/2001/XMLSchema#string"/>
           </ResourceMatch>
       </Resource>
    </Resources>
    <Actions>
```

```
        <AnyAction/>
    </Actions>
</Target>
<Rule RuleId="urn:ibm:names:tc:xacml:examples:ruleid:1" Effect="Permit">
<Description>
    Only the owner of an auction is allowed to modify the
    closing date of an auction.
</Description>
<Target>
    <Subjects>
        <AnySubject/>
    </Subjects>
    <Resources>
        <Resource>
           <ResourceMatch MatchId=
              "urn:ibm:names:tc:xacml:1.0:function:xpath-node-match">
                 <AttributeValue
                    DataType="http://www.w3.org/2001/XMLSchema#string">
                     Auction
                 </AttributeValue>
                 <ResourceAttributeDesignator
                   AttributeId="urn:ibm:names:tc:xacml:1.0:resource:xpath"
                   DataType="http://www.w3.org/2001/XMLSchema#string"/>
           </ResourceMatch>
        </Resource>
    </Resources>
    <Actions>
        <Action>
           <!-- Match ‘‘updateClosingDate" action -->
           <ActionMatch MatchId=
              "urn:ibm:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue
                 DataType="http://www.w3.org/2001/XMLSchema#string">
                   updateClosingDate
               </AttributeValue>
               <ActionAttributeDesignator AttributeId=
                  "urn:ibm:names:tc:xacml:1.0:action:action-id"
                  DataType="http://www.w3.org/2001/XMLSchema#string"/>
           </ActionMatch>
        </Action>
    </Actions>
</Target>
<Condition FunctionId="urn:ibm:names:tc:xacml:1.0:function:and">
   <!-- compare customer-id subject attribute with the
                                     owner-id value in the document -->
   <Apply FunctionId="urn:ibm:names:tc:xacml:1.0:function:string-equal">
      <Apply FunctionId=
         "urn:ibm:names:tc:xacml:1.0:function:string-one-and-only">
          <!-- customer-id subject attribute -->
          <SubjectAttributeDesignator AttributeId=
             "urn:ibm:names:tc:xacml:1.0:examples:attribute:customer-id"
             DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </Apply>
      <Apply FunctionId=
         "urn:ibm:names:tc:xacml:1.0:function:string-one-and-only">
          <!-- owner-id element in the document -->
             <AttributeSelector RequestContextPath=
               "//ac:auctionEntry/ac:ownerInfo/ac:owner-id/text()"
               DataType="http://www.w3.org/2001/XMLSchema#string">
             </AttributeSelector>
       </Apply>
    </Apply>
</Condition>
</Rule>
</Policy>
```

Request Context

What follows next is an example of the request context that is constructed by the Context handler. This is created when a user makes a request to update the closing-date of an auction. The context handler creates this document by using Java reflection on the auction object which is passed by the PEP to the context handler. This request context states that a subject with the name “Joe” and with customer-id *jh1234* is trying to update the closing date of an auction whose owner-id is *jh1234*.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request xmlns="urn:ibm:names:tc:xacml:1.0:context"
  Xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:ibm:names:tc:xacml:1.0:context
  http://www.ibm.com/irl/xacml/1.0/cs-xacml-schema-context-01.xsd">
    <Subject>
        <Attribute AttributeId=
           "urn:ibm:names:tc:xacml:1.0:subject:subjectid"
           DataType="urn:ibm:names:tc:xacml:1.0:data-type:rfc822Name">
            <AttributeValue>Joe</AttributeValue>
        </Attribute>
        <Attribute AttributeId=
           "urn:ibm:names:tc:xacml:1.0:example:attribute:customer-id"
           DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>jh1234</AttributeValue>
        </Attribute>
    </Subject>
    <Resource>
        <Attribute AttributeId=
           "urn:ibm:names:tc:xacml:1.0:resource:ufspath"
           DataType="http://www.w3.org/2001/XMLSchema#anyURI">
            <AttributeValue>
                Auction
            </AttributeValue>
        </Attribute>
        <Attribute AttributeId=
           "urn:ibm:names:tc:xacml:1.0:example:attribute:owner-id"
           DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>jh1234</AttributeValue>
        </Attribute>
    </Resource>
    <Action>
        <Attribute AttributeId=
           "urn:ibm:names:tc:xacml:1.0:action:action-id"
           DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>updateClosingDate</AttributeValue>
        </Attribute>
    </Action>
</Request>
```

Response Context
The PDP evaluates the rule applicable for the decision request and constructs a response context. In this example, the user "Joe" is also the owner of the auction and hence is permitted to update the closing-date of the auction as per the access control rule. Hence the PDP returns a result of "Permit". The syntax of the response context is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response xmlns="urn:ibm:names:tc:xacml:1.0:context"
 xsi:schemaLocation="urn:ibm:names:tc:xacml:1.0:context
 http://www.ibm.com/irl/xacml/1.0/cs-xacml-schema-context-01.xsd">
   <Result>
       <Decision>Permit</Decision>
   </Result>
</Response>
```