Security Policy Management in Federated Computing Environments

Daniela Inclezan and Michael Sobolewski
SORCER Research Group, Texas Tech University
http://sorcer.cs.ttu.edu

Abstract

The default Java implementation for security policies based on policy files doesn't comply with the specific needs of metacomputing environments. Managing a large number of policy files for all Java runtime systems in the metacomputing system doesn't scale. This paper presents a federated approach for security policy management in Java-based metacomputing systems. Security policies are stored in a policy base, which is managed by its policy service provider (Policer). The policy base and its Policer are replicated and the replicated policy bases are synchronized with each other in order to avoid a single point of failure. Any bootstrapping service provider gets its security policy from any available Policer on the network. The proposed solution ensures uniform policy-based authorization for all the services in the metacomputing environment through the use of the scalable policy management methodology. The Service ORiented Computing Environment (SORCER) is considered as a validation case for service-oriented security policy management solution presented in this paper.

1. Introduction

Built on the OO paradigm is the service-object oriented (SOO) paradigm, in which the objects are distributed, or more precisely they are remote (network) objects and play some predefined roles. A service provider is an object that accepts remote messages, called *exertions*, from service requestors to execute an elementary item of work (instruction) – a *service task*, or a composite item of work (procedure) – a *service job*.

The exertion becomes an SOO program that is dynamically bound to all relevant and currently available service providers on the network. This collection of providers dynamically participating in this federated remote invocation is called an exertion federation. This federation is also called a virtual metacomputer as federating services executing component exertions are located on multiple physical compute nodes held together by an SOO infrastructure so that, to the individual requestor submitting the exertion, it looks and acts like a single computer.

The SORCER environment [14] provides the means to create interactive SOO programs and execute them without writing a line of source code [13]. Exertions can be created using interactive user interfaces downloaded directly from service providers. Using these interfaces the user can execute and monitor the execution of exertions in the SOO metacomputer. The exertions can be persisted for later reuse. This feature allows the user quickly to create new applications or programs on the fly in terms of existing tasks and jobs.

In this paper a security policy management system is described to allow the exertion federation for a secure collaboration with presented policy services that enforce security permissions on all the federating providers.

2. Background review

The default implementation for security policy management in a Java runtime system relies on policy configuration files. These files have a simple hierarchical syntax composed of grant statements, each of which can be associated with a code base, a set of principals (optionally) and a set of permissions. A *grant* statement as a whole specifies the security permissions allowed to code downloaded from the code base, on the local Java runtime system. When a set of principals is included, the permissions are granted only to the entities corresponding to those principals. Since policy files have such a simple syntax and are saved in plaintext, they can be created manually using a text editor. Another option is to use the graphical utility called *Policy Tool* [16]. By default there is only one system policy file (`java.policy` saved in the `lib/security` directory in the Java runtime installation directory) and one (optional) user policy file (saved in the user's home directory) [15]. In order to enforce checking of the permissions stored in policy files, the security manager must be enabled at runtime. It ensures that a static Policy object is instantiated and populated based on the information coming from the system and user's policy files.

Such an approach is sufficient in the vast majority of cases, but being a default implementation, it may not be adequate for special types of applications. Policy files have a well-known syntax, are saved in plaintext and the location where they are stored is commonly known [7]. This means that even unauthenticated and unauthorized personnel can make harmful changes to their content when the policy file *write* access is not set correctly. Thus, policy files can create a breach in the security of a system and do not represent an adequately secure solution for applications that require high level security. As well, policy objects created from policy files are static objects and changes made to the policy files are not reflected by the Java runtime system unless it is restarted. In this case,

the default implementation with policy files shows a lack of flexibility that might be essential for some distributed applications.

Jini services [8], which employ the SOO paradigm, also use policy files to handle security permissions. In this case though, the policy object is dynamically created when the service is discovered [8]. The policy object created at bootstrapping is an instance of the AggregatePolicyProvider class, which supports the association of sub-policies with context class loaders. The sub-policy associated with the current context class loader or any of its parents is the active policy. If no such policy is found, then the fallback sub-policy (main policy) becomes the active one. An object instantiating a class implementing the DynamicPolicy interface is always defined as the main policy. A sub-policy instance of the LoaderSplitPolicyProvider class is associated with the current class loader. The LoaderSplitPolicyProvider sub-policy delegates permission queries and grants to either a sub-policy instance of the PolicyFileProvider class (if the current class loader, its children or the null class loader are involved in the query) or a sub-policy implementing the DynamicPolicy interface. All permission checks are handled by the currently active policy. In order to populate the sub-policy implementing the PolicyFileProvider class, the policy file specified in the service's configuration file is consulted.

Relying on policy files to enforce authorization on Jini services can cause a scalability problem. For example, if the same service is deployed on hundreds of hosts and later on some modification to the policy file is required, this change will need to be manually replicated on all related hosts. This can be not only time consuming and error-prone but as well difficult to perform in a quick and correct manner.

In federated computing environments in general, the scalability and security problems raised by enforcing authorization with policy files still exist. On top of that, there is another issue regarding the rigid syntax of policy files [7]. A security policy management based on roles may be needed for metacomputing systems, but it cannot currently be represented in a policy file. A more flexible syntax is required for such environments.

3. Security policy management framework

In the federated computing environments that use policy files to handle authorization, a large number of policy files reside on hosts. For each service provider there is one policy file (see Figure 1). If there are multiple service providers started on the same host, the host will contain one policy file for each type of started service provider. This means that on the whole federated environment there can be hundreds of policy files to manage. Modifying all these files and especially keeping the policy files synchronized is definitely cumbersome. Although policy files have a simple syntax, understanding which permissions should be allowed and to what downloaded code can be very difficult, especially in the case of metacomputing environments. This increases the possibility of untrained (and of course not authenticated or not authorized) personnel making changes that can affect the security of the system, even unintentionally.

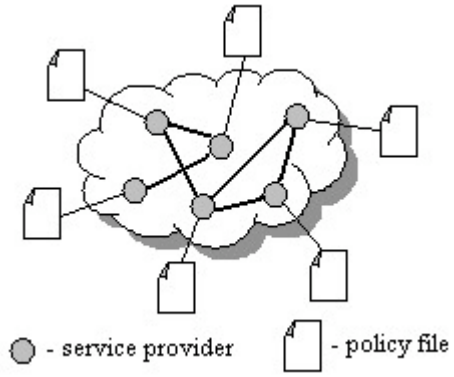

Figure 1. Federated computing environments using policy files

The solution to these security and scalability problems in the federated computing environment is to centrally manage the security policies. In order to do that we propose to store all the security permission information in a central base managed by a special service provider (Policer). Having all the permissions stored in the central base eases the management and updating operations, since they are performed in one place only, not on all the hosts. On the other hand, having all the security policy information stored in a single place can generate the one-point failure.

Having a centrally managed policy base and avoiding the one-point failure problem can be combined by replicating the Policer and its policy base in the federated computing environment. At all times it must be ensured that at least one Policer is active and able to enforce authorization in the environment. The policy bases of all the active policers must be synchronized. Thus, when a policer bootstraps, it contacts any other active policer and synchronizes its policy base to that of the active policer.

An authenticated and authorized administrator is able to modify the security information stored in policers' bases through the administrative user agent. This is a graphical interface that allows the administrator to modify, insert and delete security policies in the policy base. If all policers would provide the administrative user agent, this could create problems in the case of concurrent modifications in multiple policers. The same security policy should be present in all policy bases at all times for the same service provider. But, for example, if at the same time two different authorized administrators make conflicting changes on the same security policy, then the policers would not know which one of the two

modifications is the correct one, the one to persist in the policy base.

This concurrency problem is solved currently by allowing only one of the active replicated policers in the environment to provide the user agent at all times. If the active policer enabling policy administration (AdminPolicer) suddenly becomes inactive, a new AdminPolicer is elected from the group of already active policers. This master-slave approach (where the master is the AdminPolicer and the slaves are all the other policers) simplifies the synchronization of the policy bases. Whenever the AdminPolicer receives a modification from an authorized administrator through the user agent, it notifies all other active policers about the current update. Through this mechanism all active slave policers continually maintain their policy bases synchronized with that of the AdminPolicer. This also implies that two policers must be active in the environment all the time: the master policer and a slave policer that can take over the role of the master whenever the current master becomes inactive.

Any policer (master or slave) is able to provide the security policy object to a requesting service provider. At service bootstrapping, every service provider contacts dynamically an active policer and asks for its policy object. After mutual authentication between the bootstrapping service provider and the contacted policer, the policer retrieves the security policy information for the bootstrapping service provider from the policy base. It then creates a policy object, populates it with the security information retrieved from the policy base and passes this policy object to the service provider. On its side, the bootstrapping service provider reinforces all permissions contained in the received policy object.

Any change made by an authorized administrator to the security policy of a service provider is immediately enforced on all active service providers of that policy type. The AdminPolicer receiving the change through its user agent is responsible for notifying all the corresponding active service providers of the policy modification and for passing the new policy object on to them. The service providers receiving the modified policy object dynamically reinforce the new security policy on their side. Figure 2 briefly illustrates the main components of the proposed framework.

In the case of disconnected operations a strict policy object containing the minimal set of permissions that would allow normal functioning should be available to any service provider. Therefore, when the bootstrapping service provider fails to contact any of the policers due to intermittent lack of connectivity, the provider gets the default minimal policy object and reinforces it on itself. The provider will try to get its proper policy from any available policer later.

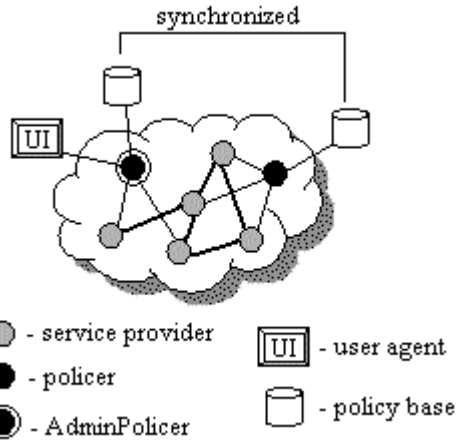

Figure 2. Proposed solution for federated computing environments

The federated policy management system is protected against outside intrusions by its own security solution. Administrators are authenticated and authorized before being allowed to access the administrative user agent and make any modifications to the existing policy base. Confidentiality and integrity is enforced on all remote communication channels. The actual schema of the policy base is hidden to the administrators and all other users.

4. Design of a policy management system

The architecture of the federated policy management system is described in Section 3 and design details of the security policy management system are presented in this Section. Different possible approaches and design decisions regarding the major components (see Figure 3) and their interaction are discussed as well.

4.1. Policy base

The policy base is represented by a relational database, which for now mimics the structure of a policy file (see Figure 4). Later on, the database schema can be easily modified to reflect any additional needs of a federated computing environment, for example role management.

The main table of the schema is *Policy*. This contains all the information needed to identify a service provider's security policy and to distinguish between different security policies stored in the database: service ID, service provider name, main published interface, location, host where the service is running and user directory where the service was started. A combination of these attributes is passed by the bootstrapping service provider to an active policer and is used by the contacted policer to retrieve the right security policy from the database (step 1 in the interaction diagram in Figure 3).

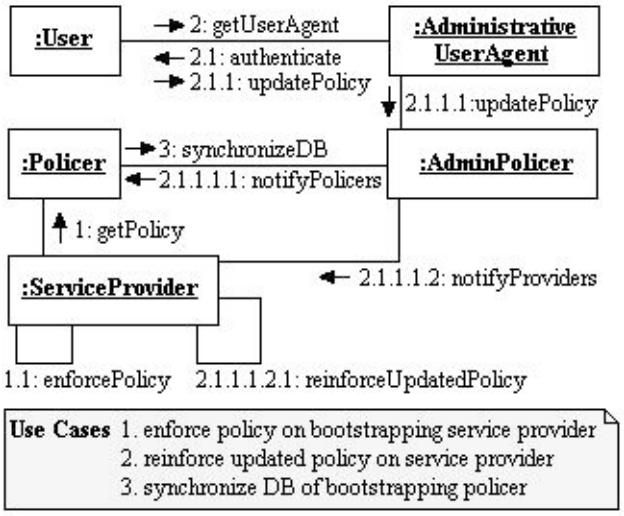

Figure 3. Interaction diagram of the policy management system

The information that would otherwise appear in a `grant` statement in a policy file is stored in the *Grant* table: code base and an optional keystore for the signer of that code base. The *User* table stores information about the (optional) principals. The *Permission* table contains records similar to the `permission` statements of a policy file: permission class, target name, and optionally action and signer.

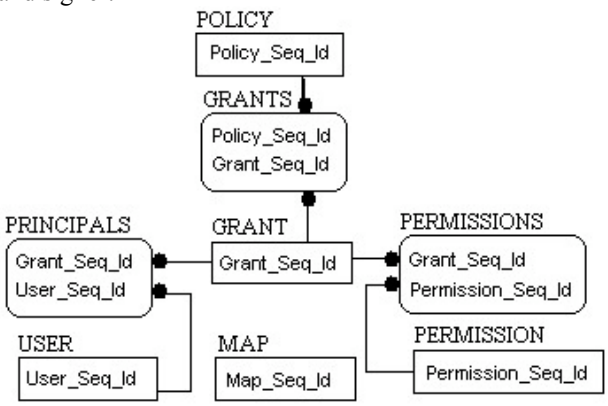

Figure 4. Policy database schema

All the strings that would appear repeatedly in different tables or records (such as class names, target names, etc.) are stored in the *Map* table. In the record where they should actually appear, the strings are replaced by a reference to the corresponding record in the *Map* table. The association tables *Grants*, *Permissions* and *Principals* allow for a flexible grouping of grants into policies, permissions into grants, and principals to grants respectively. Thus, the records in the *Grant*, *Permission* and *User* tables become highly reusable across different security policies.

There are two main choices regarding the relational database to be used: either an embedded or an enterprise database. Since our database is fairly small (only eight tables) and it requires secure and fast access to data and limited human administration or interruption, the best solution seems to be an embedded database [5]. An enterprise database would be more adequate for a large enterprise system, which is not the case here. An embedded relational database has yet another advantage over the enterprise solution: there is no need to secure the communication between the application and the database, as it would be required if a stand alone database were used.

On the other hand, in order to confer more flexibility to the framework both solutions might be implemented. Different embedded and enterprise databases could be used for storage and the requestor allowed choosing between these implementations. In case multiple storage solutions are provided, Hibernate might be considered as a portable middle object-oriented layer between the application and the various relational databases [10].

4.2. Administrative user agents

Only authenticated and authorized administrators are allowed to get the administrative user agent (steps 2, 2.1, and 2.1.1, Figure 3). There are different solutions for personnel authentication and authorization. The simplest one is based on login and password verification against security information stored on a policer. A more secure and advanced solution would require the use of smart cards, case in which keys are stored on the actual smart card only. A third solution worth considering would be the Kerberos protocol, which provides strong authentication by using secret-key cryptography [4, 6] and never passing the actual password through the network. In order to confer more flexibility to the environment, the best solution is to have all these approaches implemented and let the client select the suitable one for him.

The graphical interface of the Policy Tool utility [16] is used as a Service UI [17] model for the administrative user agent. Since the policy database follows a structure similar to that of a policy file and Policy Tool is designed to work with the same structure, many of the windows of this utility (such as the windows where grants, permissions, and principals are edited) are reused exactly as they are by the administrative user agent.

In order to comply with the specific needs of the database schema focused on federated metacomputing, some major changes have been made to the GUI inspired by Policy Tool. Fields identifying the service provider to which the edited policy belongs are added on the policy editing window: service ID, service provider name, implemented interface, etc. The first window to appear is a window listing all policies in the database. The policies displayed in a scroll box should be identifiable by all the components: service ID, service provider name, etc. The changes to the policy information are persisted in the policy base by the AdminPolicer instead into a file.

4.3. Policer replication and synchronization

The RIO framework [9] is used for policer provisioning in order to ensure that at least two active policers exist in the environment at all times. In case the master policer fails, a protocol for the election of the new master is applied: the remaining active slave policers send a message to the other active policers requiring to be elected as the new master. The first slave policer sending this message becomes the new master.

The databases of all active policers must contain the latest security policy information at all times. The synchronization of the databases of active policers is managed by the master policer. It remembers all the active slave policers in a continuously updated structure (hash table) and notifies all the policers in this structure of any policy modification coming from an administrator (step 2.1.1.1, Figure 3). The synchronization of bootstrapping policers (step 3, Figure 3) relies on the order of database events rather than on an unreliable real time clock. A vector clock timestamp is associated with every record in the database. The bootstrapping policer compares its latest vector clock timestamp with that of an active policer's to know how much behind it is. Then, either the modified records are updated or, if too many changes have occurred in the meantime, the whole database is copied from the active policer to the bootstrapping policer.

4.4. Policy reinforcement on service providers

The policy object coming from a contacted policer can be either statically or dynamically reinforced on the bootstrapping service provider (steps 1.1 and 2.1.1.1.2.1, Figure 3). A static enforcement would imply the replacement of whatever policy object was initially applied to the service provider by the new policy object coming from the policer. A dynamic enforcement on the other hand would imply adding the new policy object under the umbrella of the existing AggregatePolicyProvider object. In this case, the new policy object passed by the policer would add a new layer of restrictions on top of those already existing. Again, for flexibility reasons, both solutions should be implemented, with the dynamic approach considered the default behavior. The provider could choose otherwise by setting a required option in its startup configuration.

4.5. Security of the policy management framework

Authentication and authorization are enforced on the user agents of the AdminPolicer (step 2.1, Figure 3), which represent the only point of contact between the policy management framework and the exterior world in a design based on an embedded policy database. Confidentiality and integrity of the transferred data is ensured on all the communication channels between different policers, policers and service providers, and the AdminPolicer and the user agent by the use of Secure Socket Layer (SSL) [3]. Availability of security policy information is guaranteed through the automated replication and provisioning of policers. Modifications to policy information are not very frequent, thus only one AdminPolicer in the environment can handle all the workload.

5. Implementation

Parts of the proposed security policy management system have already been implemented. Static reinforcement of received policy object on the bootstrapping service provider's side is already available. The policer provider uses the Mckoi embedded database for policy persistence [1].

The SORCER [11-14] Grid (SGrid) was used as a validation case. An existing arithmetic service provider in the SGrid was modified to allow for static reinforcement of policy objects coming from a policer and its database. The goal of the arithmetic provider is to remotely execute various calculations requested by clients using the arithmetic user agent.

The testing scenarios relied on two different configuration files for the arithmetic service provider. In the first case, the service provider was *Policer Tester*, while in the second one the provider was *Restricted Policer Tester*. The policy stored in the database for the *Policer Tester* provider allowed for all permissions to code coming from any source. For the *Restricted Policer Tester* provider though, only the minimal set of permissions required by the service to start up was allowed. In the latter, it didn't include *read* permission for an "input.txt" file which was optionally used by the user agent to retrieve input numbers for the calculations.

The testing worked as follows: the policer, and then the two arithmetic testers named *Policer Tester* and *Restricted Policer Tester* were started. The two arithmetic providers dynamically contacted the policer, asked for their policy objects, which the policer identified using the service provider names, and both providers statically reinforced the policies received from the policer. The option to read the input numbers from a provided input file was required for both providers. In the first case the calculation succeeded as expected since all permissions were granted to *Policer Tester*. In the case of *Restricted Policer Tester* the input file was not read because an access control exception was thrown. This was expected since reading the input file was not granted by the policy stored in the database for this provider. The required calculation did not succeed in this case due to enforced security permissions.

6. Conclusions

Using policy files for authorization in Java-based metacomputing environments doesn't scale well. A scalable solution for security policy management in federated metacomputing environments is proposed here. It has the advantage of flexible database management of all security policies that say what system resources can be accessed, in what fashion, and under what circumstances. Thus, it provides for uniform authentication, authorization, and access control for all federating service providers. Confidentiality and integrity of policy information is guaranteed by securing all network communication channels. Consistency of policy data is ensured by policy base synchronization mechanisms. A friendly user agent is provided for the administrators to create and update policy information persisted in the database and then synchronized with other policer databases. Replication and autonomic provisioning of policers prevents service unavailability from occurring.

The presented framework's validity has been tested on the parts already implemented: static policy enforcement on the bootstrapping service provider; the policy object retrieval by policer from its policy base. The dynamic policy reinforcement remains to be implemented in the next project phase. The framework can be extended by adding multiple implementations to choose from for administrator authentication and policer's database solution (embedded or enterprise). In the future, the policy database can be restructured so that it reflects special needs of a particular metacomputing environment, such as role management.

This scalable methodology can be similarly applied to other aspects of security in metacomputing environments, for example federated authentication. A *KeyStorer* service provider persisting keys in a database can be designed following the same approach.

7. References

[1] Diehl and Associates, Inc., "Mckoi SQL Database", 2005. Retrieved December 27, 2006, from http://mckoi.com/database/

[2] L. Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation,* Prentice Hall PTR, 2nd edition, 2003.

[3] J. Grams, and D. Somerfield, *Professional Java Security*, Wrox Press, Birmingham, UK, 2001.

[4] Ch. Kaufman, R. Perlman, and M. Speciner, *Network Security: Private Communication in a Public World*, Second Edition, Prentice Hall PTR, 2 edition, 2002

[5] J.F. Koopmann, "Embedded Database Primer", 2005. Retrieved December 27, 2006, from: http://www.dbazine.com/ ofinterest /oi-articles/ koopmann5

[6] Massachusetts Institute of Technology, "Kerberos: The Network Authentication Protocol", 2003. Retrieved December 27, 2006, from http://web.mit.edu/Kerberos/

[7] T. Neward, "When "java.policy" Just Isn't Good Enough", 2001. Retrieved December 28, 2006 from: www.javageeks.com /Papers/JavaPolicy/ JavaPolicy.pdf

[8] J. Newmarch, *A Programmer's Guide to Jini Technology*, Apress, Berkley, CA, 2000.

[9] Project Rio, A Dynamic Service Architecture for Distributed Applications. Retrieved December 27, 2006, from https://rio.dev.java.net/

[10] Red Hat Middleware, "Hibernate: Relational Persistence for Java and .NET", 2006. Retrieved December 27, 2006, from http://www.hibernate .org

[11] M. Sobolewski, *Federated P2P services in CE Environments*, *Advances in Concurrent Engineering*, A.A. Balkema Publishers, 2002, pp. 13–22.

[12] M. Sobolewski, *FIPER: The Federated S2S Environment*, *JavaOne, Sun's 2002 Worldwide Java Developer Conference*, 2002. Retrieved December 27, 2006, from http://sorcer.cs.ttu. edu/publications/papers/2420.pdf

[13] M. Sobolewski, R. Kolonay, Federated Grid Computing with Interactive Service-oriented Programming, International Journal of Concurrent Engineering: Research & Applications, Vol. 14, No 1., pp. 55-66, 2006

[14] SORCER, Laboratory for Service-Oriented Computing Environment, Retrieved December 27, 2006, from http://sorcer.cs.ttu.edu.

[15] Sun Microsystems, Inc., "Default Policy Implementation and Policy File Syntax", 2002. Retrieved December 27, 2006, from http://java.sun.com/j2se/1.4.2/docs/gu ide/security/PolicyFiles.html

[16] Sun Microsystems, Inc., "Policy Tool – Policy File Creation and Management Tool", 2001. Retrieved December 27, 2006, from http://java.sun.com/j2se/1.3/docs/tool docs/win32/policytool.html

[17] The ServiceUI Project, Retrieved December 27, 2006, from http://www.artima.com/jini/serviceui/index.html