Role-Based Access Control in Java™

Luigi Giuri
Fondazione Ugo Bordoni
Roma, Italy
e-mail: giuri@fub.it

Abstract

As the Java platform is becoming attractive and convenient for the construction of cross-platform client-server applications, the problem of developing and managing effective security policies in that environment becomes critical.

This paper analyzes the security features provided by the new Java platform in order to identify how it is possible to improve them by providing state-of-the-art role-based access control mechanisms.

1 Introduction

The fact that government agencies, commerce operators, etc. use automated information systems for almost every activity makes the problem of designing, deploying and administering an access control policy an overwhelming task.

The research in the computer security area is toward in many directions, but one of the most promising is the so-called *role-based access control* (RBAC). This is probably the most interesting and promising technique recently proposed for design and implementation of modern system security policies. It is based on the common practice in organizations of assigning duties and responsibilities to the employees on the basis of their role within the organization itself. In this way the computer system security policy resembles the corporate security policy and all the other higher-level security policies on which it depends. The result is an increase in security comprehensibility and manageability for the entire organization, that is, an improvement of the global degree of security.

In the last few years, researchers and vendors have proposed many enhancements of RBAC models, and some RBAC implementations are currently available. The fundamentals of RBAC policies have been clearly identified [SAN96], and many RBAC models have been proposed to satisfy security requirements in different information technology domains. For example, different RBAC models have been developed for object-oriented databases [BER94], collaborative and workflow systems [JAE95, BER97], etc.

However, the considerable scientific results in the RBAC area are not entirely considered by commercial software producers. Actually, it is possible to find a lot of products that implement some kind of RBAC mechanisms, but generally they are not inspired by a common model of RBAC. This implies, for example, that it is very difficult to build a conceptual security modeling tool that can be used to target different systems.

In conclusion, it seems that there is a need of convergence to a standard model in the RBAC products area. To help satisfy this need, we have started the RBAC Implementation Project (RIP)[RIP]. RIP is an activity mainly devoted to the study and implementation of extensions for currently available systems to provide affordable state-of-the-art role-based access control mechanisms. As an example, we are working in the database field for the extension and improvement of the RBAC model required by the forthcoming ISO SQL/3 standard [GIU98].

The topic of this paper is the RIP task that is oriented to the analysis and implementation of RBAC mechanisms for the Java platform. This is a very hot topic since Java is becoming a practicable platform for both server-side and client-side computing. Moreover, since Java programs can virtually run on every hardware/OS platform and can be automatically downloaded and executed from the Internet, they can be the source of serious security problems. However, a lot of work has been done in this field (for example, see [MAR97], [MCG97], [MEH98]). As far as access control is regarded, there are interesting works about the definition of an extensible security architecture

3rd ACM Workshop on Role-Based Access Fairfax VA

Java is a trademark of Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

[WAL97], the implementation of a secure multi-processing virtual machine [BAL97], and the stack inspection algorithm [WAL98].

The remainder of the paper is organized as follows. In Section 2 we provide the basic concepts of the security model implemented by the current Java platform. Section 3 presents a hierarchical RBAC model and two possible implementations within Java, one as an extension and the other as an evolution. Section 4 shows how the provided solution can be used to implement complex RBAC models that permit to specify security constraints. Finally, Section 5 provides conclusions and suggestions for future work.

2 JDK Security

At the time this paper is being written, the security model provided by the current (non-beta) Java platform, i.e. the Java Development Kit (JDK) 1.1, is based on the so-called *sandbox* model. The basic idea behind this model is very simple. Java programs are partitioned into trusted and untrusted programs. Trusted programs run in an environment where they don't have special restrictions, thus they have full access to system resources (e.g., the file system, the network, etc.). On the contrary, untrusted programs run in a special environment (the sandbox) that allows them to access a very limited subset of system resources. For example, a program downloaded from the network (i.e. an *applet*) running in the sandbox cannot access files on the local file system, can open network connections only if the target host is the host where the applet was downloaded from, etc.

The final step for completing the sandbox model is to specify how the set of Java programs is partitioned into trusted and untrusted programs. The JDK 1.0 security model considers every local program as trusted and every remote program (i.e. every applet) as untrusted. The JDK 1.1 (which represents the latest non-beta version of the Java platform) introduces the concept of *signed applet*, which is simply an applet digitally signed using a cryptographic mechanism. When an applet is downloaded the system checks its signature. If the signature is correct then the applet is considered as a trusted program, so it is allowed to run outside the sandbox. Otherwise, the applet runs in the sandbox as an untrusted program.

Starting from this basic model, various implementations have provided some extensions. For example, Microsoft Internet Explorer™ provides four areas that represent different groups of sites (namely Internet sites, Internet sites with restrictions, trusted Internet sites, and intranet sites), and for each area a different set of allowed action (i.e. permissions) can be defined for signed and unsigned applets. Moreover, a set of denied permissions can be specified for signed applets for each area. Another example, Netscape Navigator™ provides the Capability API that allows an applet to request particular permissions that can be explicitly approved by the user that started the browser.

The basic need that pushed vendors to provide extensions to the basic sandbox model is flexibility. Actually, the objective of the above extensions is to provide more than one sandbox where different applets can perform different actions on an application-driven basis.

To satisfy these needs, the future JDK 1.2 will provide a new security model that replaces the old sandbox with the new concept of *protection domain* [GON98].

In JDK 1.2, a protection domain is a set of permissions that is associated with every program that comes from a particular origin and is signed with a specified set of public keys. The origin of a program is specified through a URL location, and the association between the origin and the set of public keys is called `CodeSource` (and represented by the corresponding class). In brief, the protection domain represents a customized sandbox associated with every Java program that belongs to a particular `CodeSource` (figure 1).

The model requires the Java runtime to provide a *policy*, that is a set of rules that permits one to calculate the set of permissions associated to a given `CodeSource`. A policy is implemented by subclassing the `java.security.Policy` abstract class. In particular, the `evaluate` method must be implemented to return a `Permissions` object for a given `CodeSource`. The JDK 1.2 provides a default policy through the `PolicyFile` class, but everyone can provide his or her own policy. The `PolicyFile` default policy provides a way to specify a policy using a set of policy entries. A policy entry grants a set of permissions to a specified `CodeSource` using the following syntax:

```
grant [SignedBy "signer-name"]
      [, CodeBase "URL"]
{
  Permission1;
  ...
  PermissionN;
};
```

Moreover, since a URL can be used to specify, for example, a directory or an entire host, then a single policy entry can represent the assignment of permissions to multiple `CodeSources`.

Note that the new security model does not make any distinction between local programs and remote programs, applying them the same policy. That is, an origin URL can refer to both local and remote origins.

The rest of this chapter will introduce some details of the JDK 1.2 security model and API that will be useful in this paper. For a complete description of the JDK 1.2 security model, see [GON98].

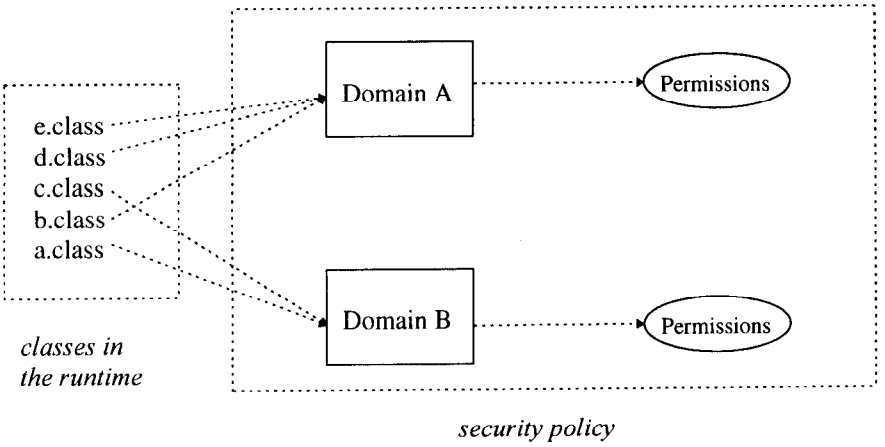

Figure 1. Protection domains in JDK 1.2

Within the `java.security` package, the following classes constitute the basis for the specification of sets of permissions:

- the `Permission` abstract class defines the basic features required for permissions, i.e. every actual permission class will be derived from this class. It represents the authorization to access a particular system resource or to execute a particular operation;
- the `PermissionCollection` abstract class represents a homogeneous collection of `Permission` objects, i.e. it holds permissions of the same type;
- the `Permissions` class, which contains a collection of `Permission` objects organized into a collection of `PermissionCollection` objects. This class is very important since, as we will see, it makes design choices on how the access control check is performed that will have a strong influence on how the RBAC extension will be defined.

For example, the `FilePermission` class is used to allow a Java program to access files and directories, and the corresponding `FilePermissionCollection` class is used to hold `FilePermission` objects.

An interesting feature of the JDK 1.2 is that it is possible to add new permission classes (eventually with the corresponding permission collection classes) in order to define application specific security policies. To do so, it is only necessary to define the new classes as subclasses of the corresponding base classes, i.e. by correctly implementing the required methods.

Finally, the access control algorithm utilizes the `implies(Permission)` method, provided by the `Permissions` class and by the subclasses of `PermissionCollection` and `Permission` classes, to verify that a particular permission is authorized by the set of permissions of a protection domain. Particularly, the `implies` method of the `Permissions` class looks like the following:

```
public boolean implies(Permission p)
{
  PermissionCollection pc =
    getPermissionCollection(p);
  return pc.implies(p);
}
```

That is, it chooses the appropriate permission collection based on the permission type, and then calls the `implies` method on it.

Note that, since the `Permissions` class is a final class and there is no way to replace it as for the `PolicyFile` class, then the behavior of its `implies` method cannot be customized for a new policy.

On the other hand, the `implies` method of new subclasses of `PermissionCollection` and `Permission` classes can be freely defined. However, since these classes are utilized under the rigid rules of the `Permissions` class, it does not make sense to define, for example, implications that consider different permission types since the `Permissions` class will only check a permission collection for one type.

3 Implementing RBAC

The new security model provided by the JDK 1.2 represents a considerable improvement with respect to sandbox models. It provides a basic access control mechanism that is simple and sufficiently general, so it can be easily extended (and, within certain limits, does not

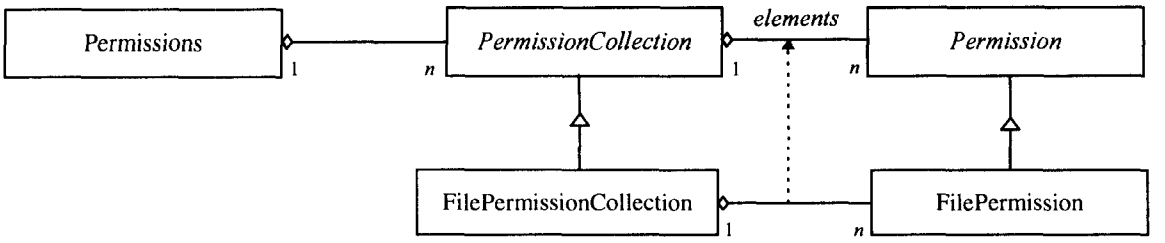

Figure 2. Permissions in JDK 1.2

preclude extensions) to provide more complex security policies than the `PolicyFile` default implementation.

In this section we propose an extension of the JDK 1.2 security model in order to provide role-based access control mechanisms. In particular, Section 3.1 specifies the requirements of what we name "basic JRBAC policy". In Section 3.2 we show how RBAC mechanisms can be provided as a "legal" extension to JDK 1.2, whereas in Section 3.3 we describe a set of (very simple) changes that could be made to JDK 1.2 to provide a direct support for RBAC.

3.1 Basic JRBAC Policy

In this proposal, roles are defined and structured as a simple role hierarchy [SAN96]. With the following rules we specify the basic requirements that must be fulfilled by an implementation of a basic Java RBAC (JRBAC) policy:

- a role is a permission;
- a role is uniquely identified by a name;
- a role includes permissions of any permission type;
- the inclusion relationship is transitive;
- cycles in the role inclusion relationship are not allowed.

That is, roles represent abstract subjects organized into a hierarchy where permissions are assigned to them and are inherited from included to including roles.

Since a role is also a permission, it can be granted to a `CodeSource` just like every other permission. However, to honor the role semantics, at a given time, every permission granted to a role is available to (i.e. is implied by) a protection domain if the role is *enabled* (or activated) in that protection domain. Since in this paper we will provide several implementation proposals of JRBAC policies, in this section we only provide the following general rules regarding role activation:

- a role can be enabled in a protection domain only if the role is granted to the corresponding `CodeSource`;
- the set of permissions available to a given protection domain is the set of permissions directly assigned to that protection domain plus the set of permissions included in its enabled roles.

A graphical representation of the basic JRBAC policy is shown in figure 3. A role hierarchy is represented by a directed graph where each node represents a role and an edge from a role r_1 to a role r_2 represents the fact that r_2 is a subrole of r_1.

Note that a role can also be used as a simple permission, i.e. a program can perform an `AccessController.checkPermission` operation simply to verify if the role is available (implied) by the current protection domain.

In the previous definitions we have chosen to use the new term "includes" even if, at first glance, it seems natural to use the term "implies" already used by the JDK to identify derivation of permissions from other permissions. The problem is that there is a semantic difference between implication and inclusion concepts, because the first is restricted to directly assigned permissions, whereas the second also comprises inherited permissions.

Practically, a role permission is implemented by the `RolePermission` class, which simply represents the role and identifies it through its name. Moreover a `RolePermissionCollection` class is defined to hold `RolePermission` objects.

From the basic rules, it follows that a basic JRBAC-compliant policy must provide a way to specify the assignment of permissions to roles. In our sample implementation, we provide an extension of the `PolicyFile` that also accepts the following syntax:

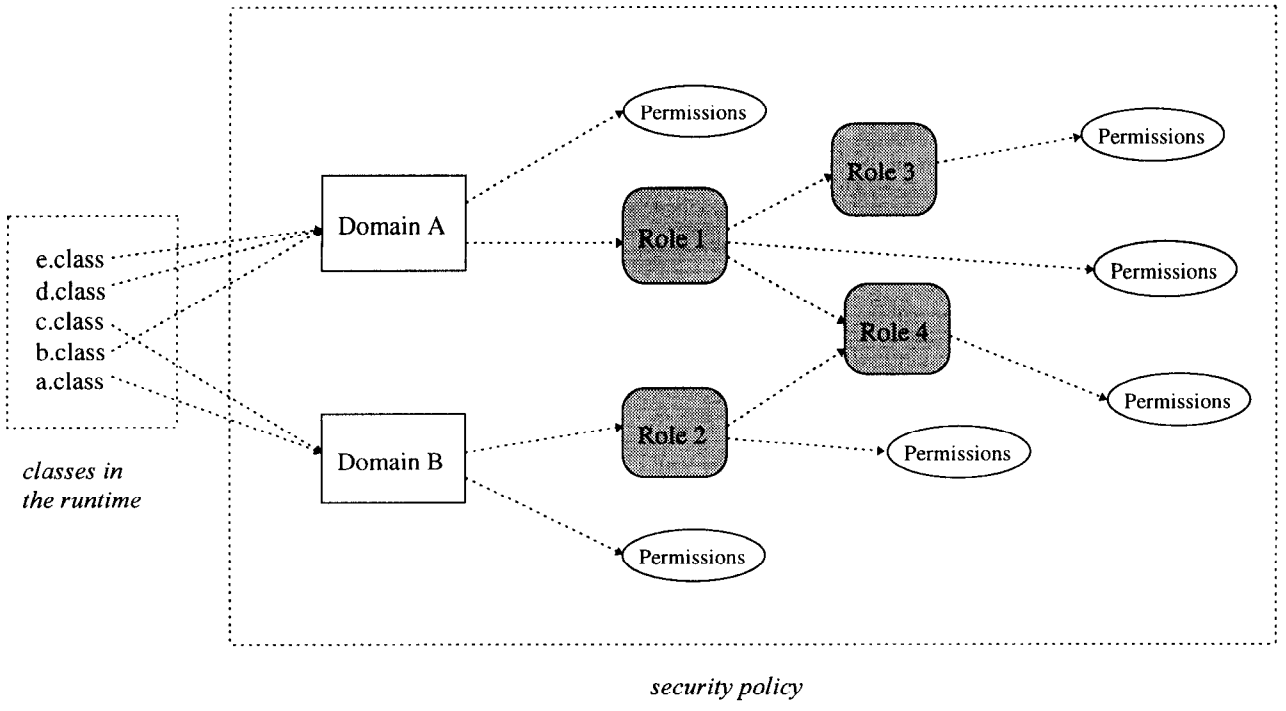

Figure 3. Basic JRBAC policy

```
grant role "role-name"
{
  Permission1;
  ...
  PermissionN;
};
```

The policy class that implements the basic JRBAC policy is responsible for verifying the consistency of the role hierarchy. That is, the policy must verify that there are no cycles in the role inclusion relationship.

The final issue that must be considered is the permission checking algorithm. The last rule of the basic JRBAC policy means that the `implies(permission)` method of the `Permissions` class must return `true` if `permission` is either directly assigned to the protection domain or it is included in one of the enabled roles. Since the JDK 1.2 defines the `Permissions` class as a final class, the `implies` method cannot be customized to realize a different behavior, so two different approaches can be considered to satisfy the last requirement:

- the first does not require modifications of the JDK 1.2 source code, i.e. it is a "legal" extension;
- the second requires modifications of the JDK 1.2 source code, so it can be employed only if it is officially accepted into the JDK as an evolution.

Next sections will provide details about the two approaches.

3.2 JDK Extension

Since the JDK 1.2 does not allow any customization regarding the `Permissions` class, to implement the JRBAC policy without modifications to the JDK, we calculate the permissions that are included by the protection domain's roles and add them directly to the protection domain itself. Moreover, since there is no way to modify the set of permissions of a protection domain at runtime, the following activation rule must be added:

- the set of enabled roles for a protection domain corresponds to the set of roles granted to the corresponding `CodeSource`.

Practically, when a protection domain is created, each role granted to the corresponding `CodeSource` is expanded into its included permissions. This operation must be performed by the `evaluate` method of the policy class.

In our sample implementation, the role policy is represented by the `RolePolicyFile` class, which actually is a subclass of the JDK `PolicyFile` class. The role policy maintains an internal representation of the role hierarchy through the following classes:

- `RolePermissions` is the class that contains permissions granted to roles. That is, an instance of this class contains all the permissions granted to a given role. This class is a very simplified version of the JDK `Permissions` class.
- `Role` is the class that represents a role definition. It contains a role name and a `RolePermissions` object. The policy will maintain only one `Role` object for each role.

Since a `Role` object contains the set of its subroles as a `RolePermissionCollection` object (contained within the `RolePermissions` object), there is no need to provide additional classes to represent the role hierarchy.

The code for the `evaluate` method is defined as shown:

```
public Permissions evaluate(CodeSource cs)
{
    Permissions perms = super.evaluate(cs);
    return expandRoles(perms);
}
```

When the `evaluate` method is called, first the `evaluate` method of the `PolicyFile` class is called to obtain a `Permissions` object, named `perms`, that contains all the permissions directly granted to the specified `CodeSource`. Then the `expandRoles` private method of the `RolePolicyFile` class is applied to `perms` in order to add every permission inherited from every role directly granted to the specified `CodeSource`, i.e. from every role identified by a `RolePermission` object contained in `perms`.

In conclusion, it is possible to add RBAC features to the current JDK, but they are limited to policies where it is sufficient that the set of permissions for a given `CodeSource` is fixed, and is statically computed at object creation time. This also causes a proliferation of permission-related structures within the system, with a possible reduction of the overall system performance.

Nevertheless, the possibility of structuring the security policy using this RBAC implementation improves the comprehensibility and manageability of the policy implementation. This is a typical advantage of every hierarchically structured RBAC mechanism.

3.3 JDK Evolution

As previously shown, the solution presented as a JDK extension correctly implements the basic JRBAC policy, but has some drawbacks. A significant problem is that, after the policy evaluates the `CodeSource` of a `ProtectionDomain` to calculate its `Permissions`, the system treats every permission as if it was assigned directly to the `CodeSource`, with the consequence that the complex structure of the policy as shown in figure 3 is practically lost.

This side-effect can be acceptable only if the application security requirements do not exceed the ones fulfilled by that implementation. If the application requires the enforcement of a complex security constraint as, for example, dynamic separation of duties, the implementation of a complex RBAC policy as a JDK extension could be impossible or at least unacceptable.

In this section we outline a possible JDK evolution that takes into account RBAC as a core component. We will show how little modifications to the source code of the current JDK can provide a simple implementation of the basic JRBAC policy that explicitly utilizes role structures for both policy definition and run-time access control checking.

This implementation can be used as a basis for the specification and implementation of more complex JRBAC policies. As stated in [SAN96], constraints are one of the reasons that pushed research in the RBAC area. With explicit role structures, it is possible to effectively and efficiently implement complex security constraints. As an example, in this paper we will describe extensions that provide the capability to specify separation of duties constraints and user-defined constraints.

In the rest of this section we describe the above mentioned JDK evolution.

The objective is to define an access control checking mechanism that is RBAC-aware, i.e. the system maintains a set of structures that represent roles, their permissions and the role hierarchy, and the check is performed by directly traversing the role structure, instead of expanding the role hierarchy into its included permissions and assigning them to the protection domain as in the case of JDK extension. Moreover, the evolution will provide a way to perform explicit role activation.

The requirements of the basic JRBAC policy are not modified, and only the following rule concerning role activation is added:

- when an object is created, its default roles are enabled,

where the default roles for an object of a given class are those defined as such within the policy for the corresponding `CodeSource`. For example, the policy file could be extended with a new `default` keyword that indicates which role permissions are default roles for a given `CodeSource`.

First of all, we extend the `RolePermissions` class introduced in the previous section in order to add the capability to check if a permission is implied by its contained permissions. This new functionality is obviously provided by the new `implies` method. This implication is defined according to the original definition of implication of the JDK 1.2, i.e. it checks only for permissions directly

granted to the role and does not consider permissions inherited from subroles. This functionality will be useful to check the permissions directly available from a single node of the role hierarchy.

The `Permissions::implies()` issue is another important problem that must be considered. As previously explained in Section 3.1, since this method cannot be "legally" customized, the only practicable way is to propose a modification of the `Permissions` class, particularly of its `implies` method, that allows a more complex and customizable definition of the permission checking algorithm. We put customizability as a requirement since we want to propose a RBAC architecture that can be extended in order to implement different, arbitrarily complex role semantics, without worsening the overall system performance.

To provide a framework for the definition and implementation of a JRBAC policy, we provide the following abstract classes:

- `RoleChecker` is the class that provides a container for a set of roles that must be considered for permission checking. This class provides the abstract method `includes(permission)` that checks if `permission` is included by the roles represented by a `RoleChecker` instance. Every protection domain has an associated `RoleChecker` object (contained within its `Permissions` object) that represents the set of its granted and enabled roles;
- `RoleIterator` is the class that provides a specialized way to navigate within the role hierarchy in order to perform a correct and efficient permission check. Every `RoleChecker` object organizes its enabled roles with a corresponding `RoleIterator` object;
- `RoleController` is the class that provides the means to enable roles.

Next, the code of the modified version of the `Permissions::implies()` method looks like the following:

```
public boolean implies(Permission p)
{
  PermissionCollection pc =
    getPermissionCollection(p);

  if (pc.implies(p))
    return true;
  else
    return currentRoles.includes(p);
}
```

where `currentRoles` is private member variable that is an instance of the `RoleChecker` class. As shown, the `implies` method checks if the permission is implied by the permissions that directly belongs to the protection domain. If this is not the case then the method checks if the `RoleChecker` object includes the permission.

Finally, the means to enable roles must be supplied by implementations of the new `RoleController` public abstract class, that requires the implementation of the following methods:

- `reset()`: disables every role;
- `resetDefaults()`: disables every role and enables default roles only;
- `enableRole(String roleName)`: adds the role identified by `roleName` to the set of enabled roles;
- `grantedRoles()`: retrieves the names of roles granted to the `CodeSource`.

Since roles that must be controlled are contained within a `RoleChecker` object, this object has the responsibility to provide a role controller through the method `getRoleController`. Then, the role controller is propagated up to the JDK's `AccessController` so that applications can perform role activation.

To summarize, the only modifications required to the JDK 1.2 source code are primarily related to the `Permissions` class and are the following:

- a new private field: `RoleChecker currentRoles;`
- a modification to the `implies` method as just shown;
- a new `setCurrentRoles` set method that allows a policy to set the correct role checker within the `evaluate` method;
- a new `getRoleController` method that retrieves the role controller from the role checker.

Moreover, a `getRoleController` method is also added to `AccessController`, `AccessControlContext` and `ProtectionDomain` classes.

These are very simple modifications that allow a whole new range of security policies to be implemented, providing complete backward compatibility.

In the rest of this section we describe an implementation of the basic JRBAC policy based on the framework.

From the definition of permission inclusion, to check if a permission is included by a role it is sufficient to find one of its subroles whose `RolePermissions` implies the given permission. This means that it is sufficient to implement a `RoleIterator` that simply enumerates every subrole of the enabled roles. This is actually implemented by the `BasicRoleIterator` class. Moreover, the `BasicRoleController` class implements a simple role controller that adds to the role iterator every subrole of the enabled roles, on the basis of the role hierarchy provided by the policy (in this case, a `RolePolicyFile` object).

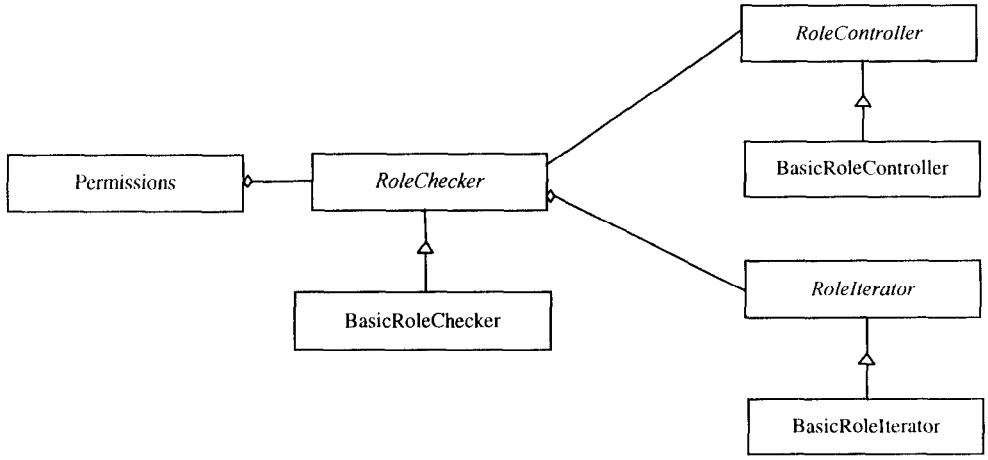

Figure 4. JDK evolution

The last step is to define the role checker, which is represented by the `BasicRoleChecker` class whose `includes` method is defined as shown:

```
public boolean includes(Permission p)
{
  RoleIterator ri = getRoleIterator();

  for(ri.reset(); ri.hasNext(); )
  {
    (Role) r = ri.next();

    if (r.getRolePermissions().includes(p))
      return true;
  }

  return false;
}
```

4 Complex policy examples

4.1 Separation of duties

In this section we will utilize the framework presented in Section 3.3 to define a complex JRBAC policy that provides the ability to specify constraints for mutually exclusive roles, i.e. it is possible to specify which roles cannot be enabled together. Actually, we allow the specification of the following entries within the policy file:

```
mutex
{
  role "role-name1";
  ...
  role "role-nameN";
};
```

and the semantics is that a role contained in a `mutex` entry cannot be enabled if another role contained in the same `mutex` entry is already enabled. This way, it is possible to specify dynamic separation of duties constraints [FER95].

The implementation of this new policy is very simple. If we start from the implementation of the basic JRBAC policy, we only need to write a checking algorithm that verifies the validity of a `setRole` with respect to the currently enabled roles, or the validity of a `resetDefaults` at all. If the operation is valid, i.e. no `mutex` entry is violated, the specified role (or roles) is enabled, otherwise the enabled roles remain unchanged and an exception is thrown.

4.2 Constrained roles

Another interesting example is the definition of a role hierarchy with activation constraints. An activation constraint is a boolean condition associated to either a node or an edge of the hierarchy, with the following semantics:

- a node (or role) constraint must evaluate to `true` in order to permit the activation of the associated role;
- an edge (or role assignment) constraint must evaluate to `true` in order to permit the activation of the associated assigned role as a subrole of the receiving role.

A formal specification of a superset of this model can be found in [GIU96].

To implement this model within the framework, we need to give a way to specify constraints code and to associate them to elements of the hierarchy. To specify the code, we provide the following interface:

```
interface RoleConstraint
{
  boolean check();
}
```

that must be implemented by every constraint, For example, a `TimeConstraint` can be implemented to check that a role is activated within a specified interval of time as shown:

```
class TimeConstraint
implements RoleConstraint
{
  private Time fromTime, toTime;

  public TimeConstraint(String from,
                        String to)
  { ... }

  public boolean check()
  {
    return (fromTime <= getCurrentTime()) &&
           (getCurrentTime() <= toTime);
  }

  private Time getCurrentTime()
  { ... }
}
```

Next, the policy file must be extended to accept the following syntax:

```
grant role "role-name"
{
  Permission1
    constraint ConstraintClass "par1" ...;
  ...
  PermissionN;
}

role "role-name"
  constraint ConstraintClass "par1" ...;
```

Finally, the `setRole` and `resetDefaults` methods must perform the correct checks while they traverse the role hierarchy in order to enable only roles whose associated `RoleConstraints` evaluates to `true`.

For example, the following definitions:

```
role "Clerk"
  constraint TimeConstraint "08:00" "17:00";

role "Clerk"
  constraint DateConstraint "Mon" "Fri";
```

means that the Clerk role can be enabled only during working days, from 8:00 to 17:00.

5 Conclusions and Future Works

RBAC is an access control model that is increasingly gaining acceptance in several information technology areas, so it is very important for the popular Java platform to be ready to support it.

Direct RBAC support by the JDK should provide a complete set of basic mechanisms that would satisfy the requirements of a large part of application developers, diminishing the need for proprietary extension. In this paper we have analyzed the latest JDK security architecture in order to identify how it is possible to provide such RBAC mechanisms. We provided both a simple extension of JDK security and an evolution of the JDK aimed to provide a base framework that is capable to be customized for different, arbitrarily complex RBAC policies.

Further work should be done in order to extend the proposed framework to other interesting policy issues like the specification of explicit denials of authorization and the activation of roles within privileged security regions (like the regions identified by `beginPrivileged()` and `endPrivileged()` JDK primitives).

References

[BAL97] Balfanz D., Gong L., "Experience with Secure Multi-Processing in Java", Technical Report 560-97, Department of Computer Science, Princeton University, September, 1997.

[BER94] Bertino E., Origgi F., Samarati P., "A New Authorization Model for Object-Oriented Databases", in Proceedings of the IFIP WG 11.3 Eight Annual Working Conference on Database Security, August 1994.

[BER97] Bertino E., Ferrari E., Atluri V., "A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems", in Proceedings of Second ACM Workshop on Role-Based Access Control, ACM Press, 1997

[FER95] Ferraiolo D., Cugini J., Kuhn R., "Role-Based Access Control (RBAC): Features and Motivations", in Proceedings of 11th Annual Computer Security Application Conference, New Orleans, LA, December 13-15, 1995.

[GIU96] Giuri L., Iglio P., "A Formal Model For Role-Based Access Control with Constraints", in Proceedings of 9th IEEE Computer Security Foundation Workshop, County Kerry, Ireland, June 10-12, 1996.

[GIU98] Giuri L., "An extension of the SQL/3 security model for a better support of role-based access control", Document ISO/IEC JTC1/SC21 WG3/DBL, n. CWB013, `ftp://jerry.ece.umassd.edu/isowg3/dbl/CWBdocs/cwb013.pdf`.

[GON98] Gong L., "Java™ Security Architecture (JDK 1.2)", draft document (revision 0.8), Sun Microsystems Inc., March 9, 1998.

[JAE95] Jaeger T., Prakash A., "Requirements of Role-based Access Control for Collaborative Systems", in Proceedings of First ACM Workshop on Role-Based Access Control, ACM Press, 1996

[MAR97] Martin D. M., Rajagopalan S., Rubin A. D., "Blocking Java Applets at the Firewall", in Proceedings of IEEE Symposium on Network and Distributed System Security, IEEE Computer Society Press, 1997.

[MCG97] McGraw G., Felten W. F., Java Security: Hostile Applets, Holes and Antidotes, Jon Wiley & Sons, 1997.

[MEH98] Mehta N., "Expanding and Extending the Security Features of Java", 7th USENIX Security Symposium Proceedings, San Antonio (Texas), Jan 1998.

[RIP] The RBAC Implementation Project, `http://www.fub.it/compsec/rip/`

[SAN96] Sandhu R. S., Coyne E. J., Feinstein H., Youman C. E., "Role-Based Access Control Models", ACM *Computer*, Vol. 29, No. 2, February 1996.

[WAL97] Wallach D. S., Balfanz D., Dean D., Felten E. W., "Extensible Security Architectures for Java", in Proceedings of 16th Symposium on Operating System Principles, Saint-Malo, France, October 1997.

[WAL98] Wallach D. S., Felten E. W., "Understanding Java Stack Inspection", in Proceedings of 1998 IEEE Symposium on Security and Privacy, Oakland, CA, May 1998.