Role-Based Access Control on the Web Using Java™

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

Abstract

This paper describes a new extension of the security features provided by the Java platform. This extension provides complex role-based access control mechanisms that take advantage of a new Java security service designed to enforce access controls based on who runs the code.

This extension will be utilized to define a new architecture that allows the design and implementation of role-based security policies for Web applications, using server-side Java technologies.

1 Introduction

Today, applications designed to run on the World Wide Web are becoming very important since they are very easy to deploy and they provide a common and familiar interface to the end user. Moreover, the huge popularity of the Internet is forcing companies to provide Web-based services to their customers. Due to this success, many technologies are competing to become leader in this field, and they are also becoming important for the development of enterprise applications based on Internet technologies (i.e. intranets).

On the other side, security problems that arises in this kind of situations are very serious, and security policies that must be enforced can be very complex. To give an acceptable solution to this problem, research and system vendors in the computer security area are considering *role-based access control* (RBAC) as a key security technology. This is probably the most interesting and promising technology 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 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. Moreover, RBAC has been included in the forthcoming ISO/SQL standard [GIU98a, SQL99].

The focus of this paper is on the Java platform and its extension for the support of Web-based server-side applications, i.e. Java Servlet. Within the Java platform, security has been considered as a key issue since the beginning of the project. 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. 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 [WAL97], the implementation of a secure multi-processing virtual machine [BAL97], and the stack inspection algorithm [WAL98]. An analysis of the security features provided by the Java platform in order to identify how it is possible to improve them using role-based access control mechanisms has been provided [GIU98b].

RBAC '99 10/99 Fairfax, VA, USA

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

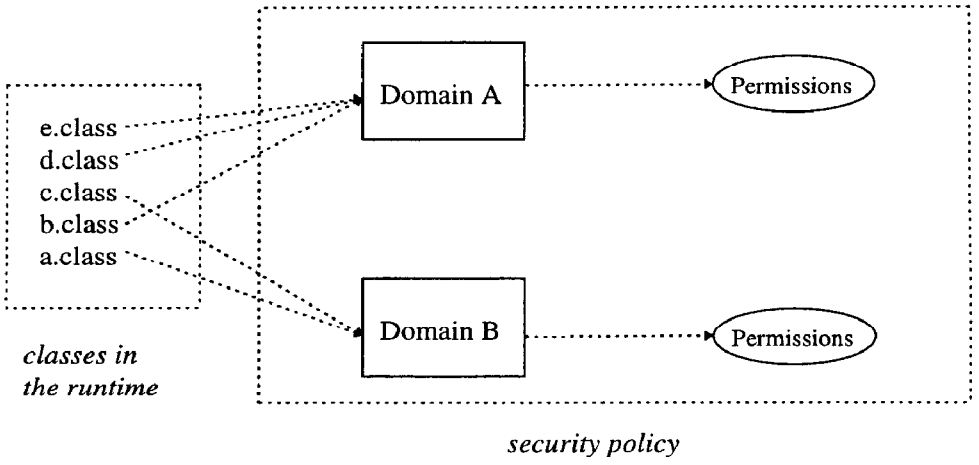

Figure 1. Protection domains in JDK 1.2.

The topic of this paper is the definition of a new Java RBAC extension that take advantage of a recently proposed Java security service designed to enforce access controls based on who runs the code, and its application to the design and implementation of security policies for Web applications.

The remainder of the paper is organized as follows. In Section 2 we provide the basic concepts of the security model implemented by the Java platform. Section 3 presents a new Java RBAC model that provides a role hierarchy and constraints. Section 4 shows how the new model can be used to implement RBAC policies for Java-based Web applications. Finally, Section 5 provides conclusions and suggestions for future work.

2 Java Security

2.1 JDK Security

The Java Development Kit (JDK) 1.2 provide a security model based on the 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 section 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].

Within the `java.security` package, 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;

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, to check if a permission is authorized at runtime, JDK 1.2 provides the new `AccessController` class. Within this class, The `checkPermission` static method determines whether the access request indicated by a specified permission should be granted or denied.

2.2 Java Authentication and Authorization Service

The JDK 1.2 security model enforces access controls based on where code came from and who signed it. To enforce similar access controls based on who runs the code, the JDK 1.2 requires additional support for user authentication, and requires extensions to the existing authorization components to enforce new access controls based on who was authenticated.

The Java Authentication and Authorization Service (JAAS) [JAA99] framework has been designed to augment the JDK with such support.

First of all, the JAAS framework provides the `Subject` class to represent the source of a request. A subject may be any entity, such as a person or service. Once authenticated, a subject is populated with associated identities, or principals, represented as instances of the `Principal` interface. A subject may have many principals. For example, a subject could have a principal that represents a user name, and another principal that represents a driver license.

To allow the implementation of different kinds of authentication technology, the JAAS framework requires applications to implement the `LoginModule` interface. For example, one particular `LoginModule` might verify a username and password, while another may interface to hardware devices such as smart cards or biometric devices.

Once a subject has been authenticated, access controls can be placed on it, based on the principals associated with that subject. The JAAS `Policy` class defines a means to grant permissions to principals. A sample implementation of a policy file (very similar to the JDK 1.2 policy file) is provided.

Finally, to check if a permission is authorized at runtime, the JAAS provides the `SecurityManager` class. Within this class, the `checkSubjectPermission` method performs subject-based access control checks.

3 The JRBAC-99 policy

3.1 Basic rules

In this section we provide a set of rules that specify what we call JRBAC-99 policy. First of all, we provide the rules that specify the concepts of user, role and role hierarchy:

- a user is a principal;
- a user is uniquely identified by a name;
- a role is a principal;
- a role is uniquely identified by a name;
- roles are organized into a (acyclic) usage hierarchy where permissions are inherited from junior to senior roles;
- roles can be assigned to users.

Since a role is a principal, permissions can be granted to a role. The set of permissions *included* by a role *r* is the set of permissions directly granted to *r* plus the set of permissions inherited by *r*.

A user is also a principal, so permissions can be granted to a user. Note that, within the JAAS framework, a subject can have many associated principals, so the following rule must be satisfied:

- a subject has at most one user principal.

Moreover, the following rule specifies how a user principal is associated to a subject:

- a user principal is associated to a subject through a login procedure.

The user principal in the JRBAC-99 framework has been introduced as a placeholder that allows the assignment of roles to real-world users. Implementations of the JRBAC-99 framework could allow security administrators to assign roles to other kinds of principals. This does not conflict with the last rule, since we only want to avoid that a subject acquires privileges that belong to different external entities, and it is possible that an entity has many associated principals. Anyway, the specification of such extension is out of the scope of this paper.

To honor the role semantics, at a given time, every permission included by a role is available to a subject protection domain if the role is *enabled* (or activated) in that protection domain. We provide the following general rule regarding role activation:

- the set of permissions available to a given subject protection domain is the set of permissions assigned to

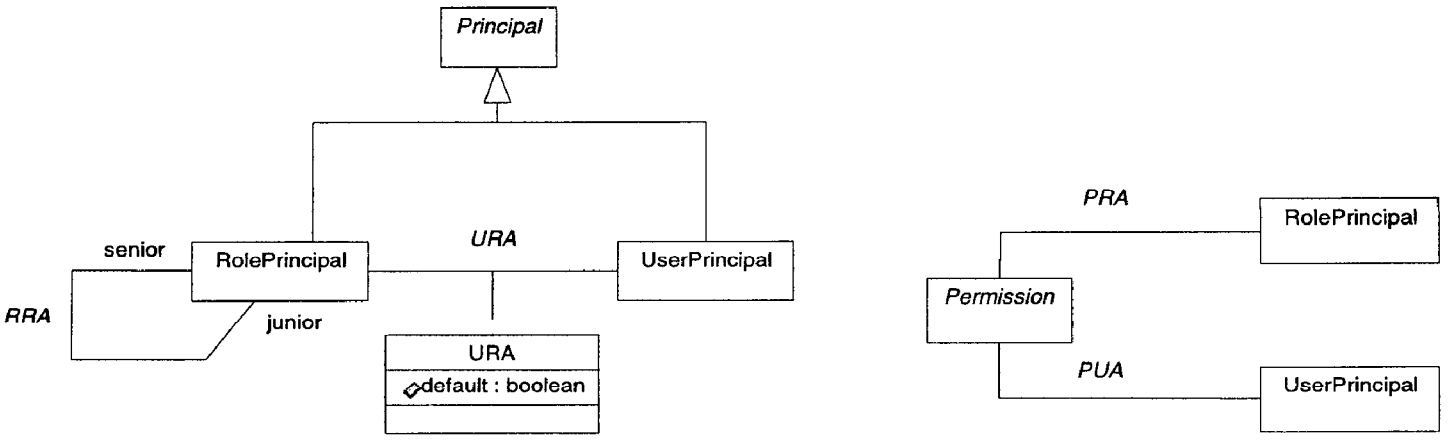

Figure 2. Basic JRBAC-99 policy.

the subject's principals plus the set of permissions included in its enabled roles;
- a role can be enabled in a subject protection domain only if the role is directly assigned to the corresponding user principal.

Note that we do not allow the activation of subroles of directly assigned roles in order to provide a sort of implementation hiding. However, this rule can be relaxed to provide more flexibility, thus allowing the activation of subroles of directly assigned roles.

Last rule is not sufficient to completely specify role activation, since it is necessary to take into account dynamic constraints (see next section).

Finally, it is necessary to specify which roles are activated when a user principal is associated to a subject (i.e., at login time). Four distinct possibilities are available:

- no roles are activated;
- all directly assigned roles are activated;
- a set of default roles are activated;
- a set of roles provided at login time are activated.

Practically, all the above rules provide the basis for a straightforward implementation as shown in figure 2. Two new `Principal` implementations are defined: `UserPrincipal` e `RolePrincipal`. The permission-role-assignment (PRA) and permission-user-assignment (PUA) relationships are directly implemented using the JAAS `Policy`. To implement the role-role-assignment (RRA) and the user-role-assignment (URA) relationships it is necessary to provide a new `RolePolicy` class. For example, a `RolePolicy` implementation could use a file (named *role policy file*) where the above relationships are represented with a syntax that is similar to the JAAS policy file syntax, that is:

```
grant [role "role-name" | user "user-name"]
{
   role "role-name1" [default];
   ...
   role "role-nameN" [default];
};
```

The `RolePolicy` class parses the role policy file and checks that there are no cycles in the RRA relationship.

To manage role activation, the `RoleController` class provides 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;
- `enabledRoles()`: retrieves the set of currently enabled roles.

Particularly, to implement permission inheritance, the `enableRole` method adds the enabled role and all its subroles to the set of subject's principals.

Finally, to associate a user principal to the subject, it is necessary to provide an implementation of the JAAS `LoginModule` interface that, after a successful authentication, add the corresponding user principal and role principals to the set of subject's principals. Practically, it is possible to define a `RoleLoginModule` abstract class that provides implementation of the part that associates authenticated principals with the subject, leaving the

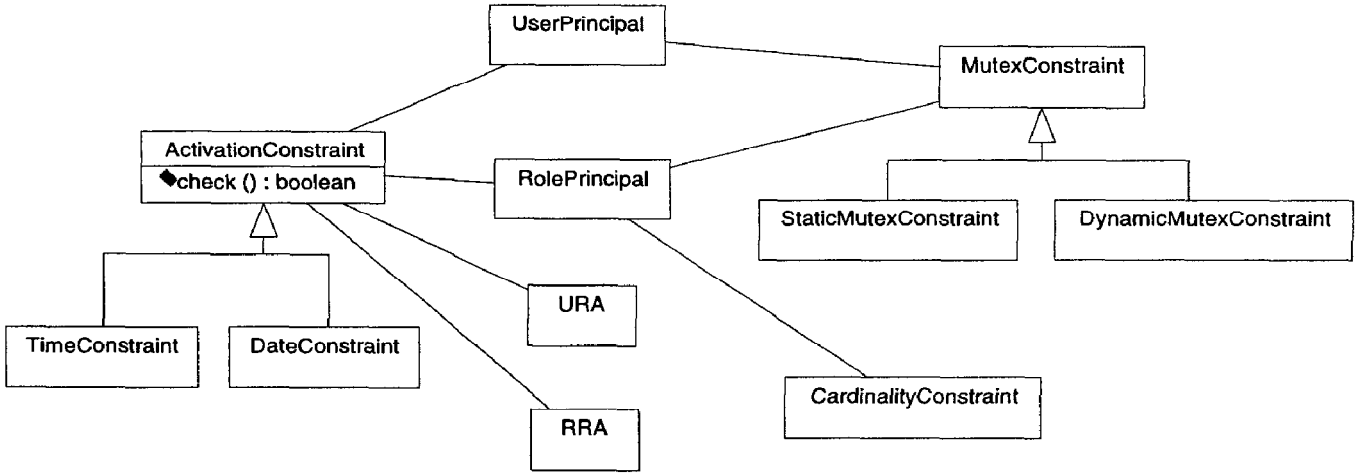

Figure 3. JRBAC-99 constraints.

authentication part unspecified. Later in this paper, the implementation of a `RoleLoginModule` that uses HTTP authentication will be described.

3.2 Constraints

The JRBAC-99 model allows the specification of constraints on users and roles. Two main categories of constraints can be identified:

- *static constraints*, that must be satisfied by the role policy;
- *dynamic constraints*, that limit the possible role activation configurations at runtime.

Figure 3 provides a schema of the actual constraint classes available.

An activation constraint is a dynamic constraint represented by a boolean condition associated to either a node or an edge of the role hierarchy, with the following semantics:

- a node constraint must evaluate to `true` in order to permit the activation of the associated node. A node constraint is associated to a user or to a role;
- an edge constraint must evaluate to `true` in order to permit the activation of the associated junior role as a child of the associated senior node. An edge constraint is associated to a URA or RRA instance.

A formal specification of a superset of this model can be found in [GIU96]. In this proposal we implement activation constraints using the `ActivationConstraint` interface that must be implemented by actual constraint classes. For example, the `TimeConstraint` class checks if the current time is within a specified interval. To specify activation constraints, the role policy file must be extended to accept the following syntax:

```
grant [role "role-name" | user "user-name"]
{
 role "role-name1" [default]
   constraint ConstraintClass "par1" ...;
 ...
 role "role-nameN" [default];
};

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

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

Note that this kind of activation constraints are different from those presented in [GIU98b] (named `RoleConstraint`) because `ActivationConstraints` are defined on principals, while `RoleConstraints` are defined on permissions. Thus, while the latter permit the specification of high granular constraints on single permissions, the former allow the specification of constraints on users and does not require modification of the standard JAAS policy. Moreover, the specification of a constraint on a single permission using `ActivationConstraints` can be obtained by defining a specific role for the considered permission and placing the constraint on that role.

Separation of duty constraints identifies set of roles that cannot be combined together. They are specified using the `MutexConstraint` class. As usual, a `StaticMutexConstraint` identifies roles that cannot be combined together within the role policy, while a `DynamicMutexConstraint` identifies roles that cannot be

combined together at run time. We also provide the possibility to specify that a `MutexConstraint` should be applied only to specific users.

To specify separation of duty constraints, the role policy file must be extended to accept the following syntax:

```
[static | dynamic] mutex
{
  role "role-name1";
  ...
  role "role-nameN";
 [user "user-name1";
  ...
  user "user-name1";]
};
```

Finally, a (static) role cardinality constraint can be associated to a role to specify the maximum number of users that can use (either directly or indirectly through role inheritance) that role. These constraints can be specified within the policy file using:

```
role "role-name"
  cardinality N;
```

Note that the implementation of static constraints is given by the `RolePolicy` class. In our example, the `RolePolicy` implementation must check that the role policy file is consistent with respect to the static constraints it specifies. Conversely, dynamic constraints are implemented by the `RoleController` class, particularly by the `enableRole()` method that must check that the situation after a role activation satisfies the set of dynamic constraints provided by the role policy.

4 The JRBAC-WEB architecture

JRBAC-WEB is a new architecture that provides RBAC on the World Wide Web using server-side Java technologies. Particularly, it is based on the Java Servlet standard Java extension. Briefly, a Java servlet is a module that extends request/response-oriented servers, such as Java-enabled Web servers. For example, a servlet might be responsible for taking data in an HTML order-entry form and applying the business logic used to update a company's order database. For more information on this topic, see [SER99].

At the heart of the JRBAC-WEB there are two servlets. The first, named `SecureHttpServlet`, uses HTTP authentication to perform a complete user login and to set the security characteristics for the execution of the user request. The second, named `SecureSessionHttpServlet`, provides the services of the previous servlet plus the capability to manage a secure session across many HTTP requests.

First of all, we add new `Permission` subclass, named `ServletPermission`, that represents a new permission used to guard the execution of HTTP requests through a secure servlet, i.e. it provides means to control access to secure servlets' services. This permission contains a name (generally referred to as a *target name*), that represents the complete name of the servlet, and an action which can be one of the following:

```
GET
HEAD
POST
PUT
DELETE
OPTIONS
TRACE
*
```

i.e., actions represent HTTP methods and * represents any method.

`SecureHttpServlet` is an abstract class that is actually a wrapper of the standard `HttpServlet` class of the Java Servlet framework. To force the execution of application code within a secured environment, this class provides a final implementation of the `service` method (figure 4). Since this method is called every time an external request is made to the servlet, to set the secure environment it is sufficient to perform a login operation. Then, a permission check (using an appropriate `ServletPermission` instance) is made before executing the standard servlet code for the requested services.

Particularly, the `login` method uses a special `RoleLoginModule`, named `HttpLoginModule`, that does not perform any special authentication since it relies on HTTP authentication, so it uses the HTTP-provided name of the user making this request to perform a login and set the subject characteristics. Since HTTP is not RBAC-aware, the login procedure starts with an empty set of enabled roles. Subsequently, application code can use the `RoleController` to enable roles, possibly on a user-driven basis (i.e., using data submitted by the user through HTTP).

The permission check simply consists in a check of the `ServletPermission` associated to the requested HTTP method.

Thus, to create an application-specific secure servlet it is sufficient to define that servlet as a subclass of `SecureHttpServlet` instead of the standard `HttpServlet`.

Unlike the approach of [FER99], we neither associate a subject to a user, nor we maintain information on enabled roles on the server. Our approach tends to preserve the basic semantics of underlying technologies, so we do not maintain security state information for a stateless protocol.

However, complex web-based applications (like electronic commerce systems), do require that some information is used across multiple HTTP requests. To support this kind of requirements, the Java Servlet API

```
public abstract class SecureHttpServlet
  extends javax.servlet.http.HttpServlet
{
  public final void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
  {
    login(request, response);
    checkPermission(request, response);

    super(request, response);
  }

  ...
}

public abstract class SecureSessionHttpServlet
  extends javax.servlet.http.HttpServlet
{
  public final void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
  {
    sessionLogin(request, response);
    checkPermission(request, response);

    super(request, response);

    setSessionRoles(request, response);
  }

  ...
}
```

Figure 4. JRBAC-WEB secure servlets.

provides the `HttpSession` class that permits to store data whose life spawns multiple requests. Actually, session data is stored into the client using cookies or an URL-rewriting technique [SER99].

To support secure sessions, we provide the new `SecureSessionHttpServlet` class. In this class, the method `setSessionRoles` stores into the `HttpSession` the set of enabled roles associated with the subject. The method `sessionLogin` perform a login using the HTTP authenticated user name and the roles stored into the `HttpSession` (the first time it creates an empty session). Thus, the service method combines the two previous methods obtaining a complete security session management (figure 4).

To achieve a higher level of security, session data could be cryptographically signed by `setSessionRoles` and verified by `sessionLogin`. Anyway, this is out of the scope of this paper.

5 Conclusions and Future Works

Today, it is very important to have means that allow the design and implementation of complex security policies for Web-enabled applications. This paper shows that mature technologies like Java and RBAC, combined on the server side, have good chances to achieve a leadership position in the information technology area.

Further work could be done in order to extend the proposed framework, for example, to provide an extensible constraint system. Another interesting topic is the specification of an alternative framework that uses, instead of the JAAS extension, the approach named *security-passing style* [WAL99], that could also be applied to previous versions of the Java platform.

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.

[FER99] Ferraiolo D. F., Barkley J. F., Kuhn D. R., "A Role Based Access Control Model and Reference Implementation within a Corporate Intranet", ACM Transactions on Information and System Security, Volume 2, Number 1, February 1999.

[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.

[GIU98a] 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.

[GIU98b] Giuri L., "Role-Based Access Control in Java", in Proceedings of Third ACM Workshop on Role-Based Access Control, ACM Press, 1998.

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

[JAA99] Java Authentication and Authorization Service, http://www.javasoft.com/security/jaas/.

[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.

[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.

[SER99] Java Servlet API, http://www.javasoft.com/products/servlet/

[SQL99] Jim Melton (ed.), "ISO Final Draft International Standard (FDIS) Database Language SQL - Part 2: Foundation (SQL/Foundation)", ISO/IEC JTC1/SC32 N00223

[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.

[WAL99] Wallach D. S., "A New Approach to Mobile Code Security", Ph.D. dissertation, January 1999.