ROLE-BASED DELEGATION AND REVOCATION IN XORBAC - IMPLEMENTATION EXPERIENCES

Sigrid Schefer-Wenzl
University of Applied Sciences Campus Vienna, Austria
Competence Center for IT-Security
sigrid.schefer-wenzl@fh-campuswien.ac.at

ABSTRACT
Delegation has been repeatedly identified as an important concept to increase flexibility in access control management. In case of an employee’s temporary absence it may be useful that a user can delegate parts or all of his/her permissions to another user who subsequently will act on behalf of the delegator. Several delegation models exist to ensure a systemic delegation and revocation of access rights for secure software-based systems. In this paper, we present our experiences when implementing two delegation and revocation models into the access control management system xoRBAC.

KEY WORDS
Access control; Delegation; RBAC; Revocation; xoRBAC;

1 Introduction
Access control deals with the elicitation, specification, maintenance, and enforcement of access control policies in software-based systems [8]. In software systems management, the definition of access control policies is a useful way to cope with the increasing complexity and rapid changes in distributed systems by automating activities which are usually done manually.

In general, a policy is a statement that aims to influence or guide behaviour of subjects in information systems [11]. Each information system may be controlled by a number of different policies which are related or interdependent. The behavioural rules defined by policies support information systems in deciding autonomously how to act when certain events occur. Thus, the definition and utilization of policies significantly eases software systems administration. Moreover, policies can also be used to synchronize software-based systems with the operational goals and scenarios of a specific organization by defining relationships between subjects and targets [11, 12]. Hereby, the subject is the active entity invoking an operation on another entity, called target [12]. In access control management, policies define access rights for users on system resources.

Organizations usually apply access control models to define access control policies [8]. In recent years, role-based access control (RBAC) [2] has developed into a de facto standard for access control in both, research and industry. In RBAC, roles correspond to different job-profiles and scopes of duty within a particular organization or information system [12]. Access permissions are assigned to roles according to the tasks each role has to accomplish. Human users and other active entities are assigned to roles with respect to their work profile. Thereby, each subject acquires all permissions necessary to fulfill its duties via his/her role memberships.

Using roles as an abstraction mechanism for users facilitates the administration of access rights and the admin-
The xoRBAC component offers a multi-policy access control service, which can enforce RBAC as well as MAC- and DAC-based policies. Some of the main features of xoRBAC are: many-to-many user-role and permission-role assignment (and revocation); user-role review and permission-role review; definition of arbitrary role-hierarchies; definition of access control constraints, such as static separation of duty (SSD), dynamic separation of duty (DSD), and context constraints for roles and permissions; definition of maximum and minimum cardinalities for roles and permissions.

xoRBAC is implemented via the programming language eXtended Object Tcl (XOTcl, see, e.g., [4, 6]). XOTcl is an object-oriented extension of the scripting language Tcl [7]. XOTcl is a C-library that can be dynamically loaded into Tcl-compatible environments and is embeddable into C programs. Moreover, it offers language support for object-oriented concepts, such as design patterns and aspect orientation. XOTcl is based on the class concept of OTcl [14] and supports meta classes, filters, per-object mixins, per-class mixins, and dynamic object aggregation (see, e.g., [15]). XOTcl mixin classes are a dynamic message interception technique. They allow to flexibly define extension classes in addition to the inheritance hierarchy of the object a mixin is registered for. For method resolution, mixin classes are searched prior to searching the object’s class itself.

In xoRBAC, this feature is useful to give an object or a classes’ instances access to several different supplemental classes, which may be changed dynamically. In particular, XOTcl supports per-object- and per-class mixins. Per-object mixins are classes that are applied as mixins for an individual instance of a class, while per-class mixins are classes that are applied as mixins for a class (see [15] for details). Moreover, the dynamic object aggregation mechanism supports the dynamic aggregation and disaggregation of objects during runtime. The per-object-mixin (POM) as well as the dynamic object aggregation language constructs are used in xoRBAC to dynamically activate or deactivate certain behaviour for a class or object.

The xoRBAC component consists of the following eight main classes: RightsManager, Role, Permission, User, Audit, as well as Sensor, Condition, and ContextConstraint (see [5, 13]). Several design patterns are used in the implementation of xoRBAC. For example, the RightsManager class serves as Facade for the xoRBAC component. This is because it hides xoRBAC internal structures from other components that use xoRBAC. Therefore, every external component uses xoRBAC through a well-defined API offered by the RightsManager class. Furthermore, the user-role assignment and the permission-role assignment relations are implemented using the Decorator pattern. For further details, please refer to [5].

In order to assign a certain permission to a specific role, the respective Permission object is registered as POM for a particular Role object. Similarly, a Role instance is registered as POM for a certain Subject instance. At runtime, a RightsManager object contains the instances of the Subject, Role, Permission, ContextConstraint and Condition classes via dynamic object aggregation. These instances are encapsulated in a common name space.

### 3 Implementing a Basic Delegation Model

In this paper, the software component xoRBAC is extended to support the enforcement of delegation policies following primarily the basic delegation models introduced in [3, 16]. The main features of the delegation extension presented in this Section include:

- creation of delegation roles for single- and multi-step delegation which can be further delegated arbitrary times;
- activation of multi-step delegation and change of the maximum number of further delegations for delegation roles at any point of time;
- simple and cascading revocation of multi-step delegation;
- permission-based delegation and revocation;
- role-based delegation and revocation;
- assignment of subjects to delegation roles;
- simple and cascading revocation of delegation role-subject assignments;
- defining delegation constraints on permissions and roles;
- introspection mechanisms for delegation roles;
- deletion of delegation roles and all associated relations.

The delegation model supports user-to-user delegation. Accordingly, each action, such as creating, changing or deleting a delegation role can only be performed by the creator of this role. The creator of a delegation role is also called delegator and cannot be changed.

Subsequently, we describe several implementation experiences from implementing our delegation model based on the formal models presented in [3, 16]. Due to the page restrictions, all code visualizations are reduced to the most relevant parts.

#### 3.1 Single- and Multi-Step Delegation Roles

Our method createDelegationRole (delegator role {multiStepDelegation 0} {levels -1}) enables a subject to create a delegation role. Subsequently, roles and permissions can be delegated to this delegation role. In particular, we create an object of the class DelegationRole, which is a subclass of Role. According to the prerequisites of user-to-user delegation, the method requires to declare the subject which creates the delegation role as well as a unique name for the new delegation role.

After a delegation role has been successfully created, it is included into the array createdDelegationRoles of the respective subject, which has created the delegation role. This subject will subsequently act as the delegator of this delegation role. The delegator can delegate to a delegation...
role, assign subjects, and revoke single permissions of role from a delegation role. Moreover, no other regular or delegation role can be created with the same name.

By default, a delegatee is not allowed to further delegate a delegation role (single-step delegation). However, the delegator can activate multi-step delegation already when creating a delegation role. The boolean instance variable multiStepDelegatable is then set to 1 (i.e., true) for the corresponding delegation role object. In addition to activating multi-step delegation, the delegator can define how often the delegation role can be delegated further. The instance variables levelsOfDelegation and remainingLevelsOfDelegation either get the specified value or, if no value is defined, are set to 1, i.e. the corresponding delegation role can be further delegated once. The value of the levelsOfDelegation variable remains constant, while the value of remainingLevelsOfDelegation is changed according to the number of currently possible further delegations. The value -1 leads to an infinite number of further delegations for a delegation role.

A delegator can also activate multi-step delegation or change the number of maximum further delegations after creating a delegation role via the method enableMultiStepDelegation (delegator delegationRole {levels -1}). Via the method incrLevelsOfDelegation (delegator delegationRole levels) the delegator can arbitrarily increase the number of further delegations for a delegation role.

The method disableMultiStepDelegation (delegator delegationRole {revocationType cascading}) enables a delegator to deactivate multi-step delegation. The parameter revocationType allows for simple or cascading revocation of multi-step delegation. By default, multi-step delegation is revoked cascadingly. Hereby, all existing relations of a particular delegation role are revoked. If simple revocation is selected, only future further delegations of a delegation role are prohibited.

A delegator can delete a delegation role via the method revokeDelegationRole (delegator delegationRole). Before deleting a delegation role, first all delegation role-subject relations are removed and subsequently the subject is revoked as the delegator of the delegation role.

The method transferDelegationRoleOwnership {old new} transfers the creatorship of all delegation roles of subject old to subject new. As a result, new can now act as the delegator of all delegation roles created by old.

3.2 Delegating and Revoking Roles and Permissions

A subject can delegate all permissions it is assigned to directly or via the role-hierarchy by assigning them to one of its delegation roles and assigning the delegation roles to another subject. The method delegatePermToDelegationRole (delegator perm delegationRole) checks if the delegating subject is the creator, i.e. delegator, of this delegation role (see Figure 1). In addition, it checks if any delegation constraints prohibit the delegation of the specified permission. After several further consistency checks, the respective Permission object is registered as POM for the corresponding DelegationRole object.

A whole role is delegated to a delegation role by using the method delegateRoleToDelegationRole (delegator role delegationRole). Amongst others, the method first checks if the subject invoking this method is the creator of this delegation role and if the role to be delegated is not already defined as junior role of delegationRole. In addition, it checks for potential delegation constraints prohibiting the delegation. If all checks are successful, delegationRole is defined as senior role of role and thereby inherits all permissions of the delegated role. Note that a regular role must not be defined as senior role of a delegation role in order to prohibit illegal permission combinations (see, e.g., [16]). If the role to be delegated is a delegation role, multi-step delegation needs to be activated for role. In addition, the maximum number of further delegations allowed for role must not be exceeded for role or any of its junior roles. The number of further delegations when delegating a delegation role is initially increased by 1 and subsequently increased by the number of senior roles which are assigned to delegationRole, to which role is to be delegated.

Consider an example where delegation role D1, which has the junior roles D2 and D3, is delegated to delegation role D8, which has the senior roles D9, D10, and D11. Subsequently, the number of further delegations for D1, D2, and D3 is reduced by 4.

The delegation of a permission can be revoked via the method permDelegationRoleRevoke (delegator perm delegationRole). One prerequisite of a successful revocation...
cation is that the delegation role owns the specified permission directly. The delegation of a role can be revoked via the method roleDelegationRoleRevoke {delegator role delegationRole}. The role to be revoked needs to be a direct junior role of the delegation role.

3.3 Delegation-Role Subject Relations

A delegator can assign a delegation role to a subject by invoking the method delegationRoleSubjectAssign {delegator delegationRole delegatee} (see Figure 2). The subject receives all permissions and roles that are assigned to the delegation role and can subsequently further delegate this role to its delegation roles if multi-step delegation is activated and the maximum number of further delegations has not been reached yet. It is also important to consider corresponding delegation constraints.

The creator of a delegation role can also assign himself/herself to his/her delegation roles, if this is not prohibited by delegation constraints. This makes sense if a subject wants to make a backup of its current role memberships by assigning all permissions owned by the subject to its delegation roles.

The assignment of a subject to a delegation role can be revoked via the method delegationRoleSubjectRevoke {delegator delegationRole delegatee} (see Figure 3). By default, revoking the role membership is cascadingly. The delegatee, all delegation roles of the delegatee as well as all subjects assigned to the delegatee’s delegation roles subsequently loose their direct or transitive assignment to the specified delegationRole. When choosing simple revocation only the specified subject looses its membership to the delegation role. All existing relations between delegation roles of the subject and delegationRole remain.

3.4 Defining Delegation Constraints

Delegation constraints can be defined on permission level (delegatePermConstraints) or on role level (delegateRoleConstraints). In the basic model, delegation constraints are the only way to restrict a user-to-user delegation. For each delegation of a role or permission and for each assignment of a subject to a delegation role, it is checked, if a delegation constraint prohibits this action. In our extension, two types of delegation constraints can be defined. First, the method setDelegatePermConstraint {role1 role2 perm} specifies a delegation constraint on permission level, which prohibits that members of role1 and all members of its junior roles can delegate the permission perm to members of role2 or to members of its junior roles.

Second, the method setDelegateRoleConstraint {role1 role2 role} specifies a delegation constraint on role level, which prohibits that members of role1 and all members of its junior roles can delegate the role role to members of role2 or to members of its junior roles.

The methods delegatePermConstraintFor {role1 role2 perm} and delegateRoleConstraintFor {role1 role2 role} enable to specifically search for certain delegation constraints on permission or role level. The method hasRoleDelegateRoleConstraint {role1 role2 role} checks, if any delegation constraint is defined.
for role1 or any of its senior roles, which prohibits the delegation of role or one of its senior roles to members of role2 or members of its senior roles. The methods `getAllDelegatePermConstraintsForRole {role}` and `getAllDelegateRoleConstraintsForRole {role}` return all delegation constraints for role and its junior roles. Furthermore, the methods `getDirectDelegatePermConstraintsForRole {role}` and `getDirectDelegateRoleConstraintsForRole {role}` only return the delegation constraints directly defined for role (without considering its junior roles).

### 3.5 Introspection Mechanisms for Delegation Roles

Several introspection mechanisms already exist in xoRBAC. The following introspection functions were added into xoRBAC specifically for the purpose of delegation:

- `isDelegationRole role` checks if the specified role is a delegation role.
- `getAllDelegationRolesCreatedBySubject {delegator}` returns a list of all delegation roles created by a particular subject.
- `isSubjectCreatorOfDelegationRole {subject role}` checks if a certain subject has created a particular delegation role.
- `getDelegationRoleList {}` returns a list of all currently available delegation roles.
- `isMultiStepDelegationEnabled {delegationRole}` checks if a certain delegation role is multi-step delegatable.
- `getLevelsOfDelegation {delegationRole}` returns how often a delegation role can currently be delegated further.

### 3.6 Example

The following example shows the sequence of methods which need to be invoked in order to delegate a role or permission and to revoke a delegation. In our example, subject Alice owns the role Professor, which is a senior role of the roles Secretary, AssistantProfessor and Student. Professor is assigned to the permissions PUT_Document and DEL_Document. Alice is allowed to delegate her role Professor as well as all junior roles and all assigned permissions to other subjects if there does not exist any delegation constraint prohibiting this delegation.

For delegation purposes, Alice creates a delegation role ExtendedSecretary for single-step delegation via the following command:

```
rm createDelegationRole Alice ExtendedSecretary
```

From now on, a subject cannot create a delegation role with the same name any more. Similarly, it is not possible to create a regular role with the same name. The following method invocation returns if ExtendedSecretary actually is a delegation role:

```
rmi isDelegationRole ExtendedSecretary
```

We can also check, which delegation roles Alice already has created as well as if Alice has created a particular delegation role:

```
rmi getAllDelegationRolesForSubject Alice
rn isSubjectCreatorOfDelegationRole Alice ExtendedSecretary
```

The following command returns the list of all delegation roles that currently exist:

```
rmi getDelegationRoleList
```

Alice can delegate permissions which she is assigned directly or via the role-hierarchy to her delegation role ExtendedSecretary:

```
rmi delegatePermToDelegationRole Alice DEL_Document
ExtendedSecretary
```

Similarly, Alice can delegate roles which she owns directly or via the role-hierarchy. Afterwards, ExtendedSecretary is a senior role of the delegated role:

```
rmi delegateRoleToDelegationRole Alice Secretary ExtendedSecretary
```

Subject Bob is assigned to the delegation role and thereby acquires all permissions which were delegated to ExtendedSecretary:

```
rmi delegationRoleSubjectAssign Alice ExtendedSecretary Bob
```

In order for Bob to be able to further delegate his delegation role ExtendedSecretary, Alice needs to activate multi-step delegation for ExtendedSecretary:

```
rmi enableMultiStepDelegation Alice ExtendedSecretary 5
```

ExtendedSecretary can now be further delegated five times. Using the following method invocations, we can check if and how often a delegation role can be delegated further:

```
rmi hasMultiStepDelegationEnabled ExtendedSecretary
```
Alice can change the number of further delegations. After incrementing the value by 2, ExtendedSecretary can be further delegated seven times:

```
rm remainingDelegationLevelsForRole ExtendedSecretary
```

Furthermore, delegation constraints can be defined for roles and permissions. In our example, a role-level delegation constraint defines that members of role AssistantProfessor are not allowed to delegate the role AssistantProfessor to members of the role Student:

```
rm setDelegateRoleConstraint AssistantProfessor Student
```

A permission-level delegation constraint may define that members of role Professor are not allowed to delegate permission PUT_Document to members of role Student:

```
rm setDelegatePermConstraint Professor Student
```

Figure 4 visualizes the example RightsManager object rm in xoRBAC after all those method invocations.

Alice can deactivate multi-step delegation for ExtendedSecretary. She can choose between simple or cascading revocation. The following method invocation does not revoke existing role relations of ExtendedSecretary (simple revocation):

```
rm disableMultiStepDelegation Alice ExtendedSecretary 0
```

At any time, Alice can revoke the assignment of a subject to her delegation role. Again, she can choose between simple and cascading revocation. After the following method invocation not only Bob looses his role membership to ExtendedSecretary, but also all relations between delegation roles of Bob and ExtendedSecretary are revoked (cascading revocation):

```
rm delegationRoleSubjectRevoke Alice ExtendedSecretary Bob
```

The delegation of single permissions or roles can also be revoked. Hereby, the delegation role needs to be directly assigned to the delegated permission or the delegated role needs to be a direct junior role in order to undo the delegation:

```
rm permDelegationRoleRevoke Alice DEL_Document ExtendedSecretary
```

```
rm roleDelegationRoleRevoke Alice Secretary ExtendedSecretary
```

Alice can also completely delete the delegation role ExtendedSecretary including all its subject and role relations. Note that if subject Alice is removed, all her delegation roles are also deleted:

```
rm deleteDelegationRole Alice ExtendedSecretary
```

- creation of delegatable roles in parallel to creating regular roles;
- defining delegatable and non-delegatable permissions;
- restricting the delegation of roles and permissions to delegatable roles and subjects;
- introspection mechanisms for delegatable roles.

We implemented several alternative versions in order to efficiently realize the three different role types — regular roles, delegatable roles, and delegation roles — and their (inter-)relations with respect to simple administration. In comparison to all other designs, we consider the version implementing a parallel approach for creating, changing, and deleting regular and delegatable roles as well as their junior and senior role relations to be the most intuitive one:

This implementation guarantees a 1:1 relation between regular and delegatable roles as recommended in [16]. Moreover, administrating the new relations for delegatable roles does only require a minimum of additional effort compared to the basic model introduced in Section 3. The owner of a regular role is automatically also the owner of a corresponding delegatable role. However, he/she can only delegate the permissions which are assigned to the delegatable role to a delegation role. Furthermore, delegatable roles are easy to distinguish from regular and delegation roles via their names by using the prefix “delegatable.”.

## 4 Extending the Delegation Model

In this Section, we describe an extension of the delegation model described in Section 3 which restricts user-to-user delegation to a limited set of delegatable roles and permissions. The following additional features are available when using this package:

### 4.1 Creating and Deleting Delegatable Roles

We adapted the original xoRBAC method `createRole` (name `{juniorRoles `''`} {seniorRoles `''`}) in a way that as soon as a regular role is defined the corresponding delegatable role of type `DelegatableRole` is also created. In addition, the delegatable role is also automatically assigned to all its junior and senior roles. As mentioned above, every delegatable role has the prefix “delegatable_”. For example, when creating the role `role1` having a junior role `role2` and a senior role `role3`, the corresponding delegatable role `delegatable_role1` of type `DelegatableRole` is also created as well as the assignments to the junior role `delegatable_role2` and to the senior role `delegatable_role3`.

Thus, junior and senior role relations are not only defined between the specified regular roles but also between the respective delegatable roles. The relevant xoRBAC methods `addJuniorRoleRelation {role junior {type `''junior''`}}` as well as `removeJuniorRoleRelation {role junior {type `''junior''`}}` were adapted accordingly. Similarly, besides removing the regular role `role` the method `deleteRole {role}` does also remove the associated delegatable role `delegatable_role`.

### 4.2 Delegating Delegatable Roles and Permissions

In this extended delegation model, subjects can only delegate permissions which they own via an assignment to a delegatable role. Furthermore, a subject can not delegate its regular roles but only the respective delegatable roles to their delegation roles.
By using the method `roleSubjectAssign {role subject}` a subject is simultaneously assigned to the regular role `role` as well as to the corresponding delegatable role. Accordingly, the method `roleSubjectRevoke {role subject}` removes the relation between `subject` and the regular role `role` as well as between `subject` and the delegatable role.

A permission is transitively defined as being delegatable by assigning it to a delegatable role. A permission is assigned to a delegatable role if the parameter `delegatable` gets the value `true` when invoking the method `permRoleAssign {perm role {delegatable 0}}`. By default, a permission is defined as being not delegatable and thus is only assigned to the regular role `role`. A permission can also be defined as delegatable at a later point of time by invoking the method `permDelegatableRoleAssign {perm role}`.

Via the method `permRoleRevoke {perm role}` the relation between `perm` and `role` as well as between `perm` and `delegatable role` is removed. A delegatable role is not allowed to own any permission which is not assigned to the corresponding regular role.

For delegating a permission, the method `delegatePermToDelegationRole {delegator perm delegationRole}` introduced in Section 3 was adapted. Permission `perm` which shall be delegatable needs to be delegatable. `perm` is delegatable if it is assigned to a delegatable or delegation role owned by subject `delegator`.

For delegating a role, the method `delegateRoleToDelegationRole {delegator role delegationRole}` first checks if `role` is a regular role. In this case `role` itself but the corresponding delegatable role is delegated to `delegationRole`. If `role` is a delegation role it can be delegated unless the maximum number of further delegations has not yet been reached.

### 4.3 Introspection Mechanisms for Delegatable Roles

The following introspection functions were added for delegatable roles:

- `isDelegatableRole {role}` checks if the specified role is a delegatable role of type `DelegatableRole`.
- `getDelegatableRoleList {}` returns a list of all currently available delegatable roles.
- `getAllDelegatableRolesAssignedToSubject {subject}` returns a list of all delegatable roles which are assigned to `subject`.

### 4.4 Example

The following example shows the sequence of methods which need to be invoked in order to delegate a role or permission and to revoke a delegation via our extended delegation model. Subject `Alice` owns role `Professor` like defined in Section 3.6. When creating the regular role the corresponding delegatable role is also created. Every subject owning a regular role is also assigned to the respective delegatable role. Thus, `Alice` also owns the role `delegatableProfessor`.

We can now retrieve a list of all available delegatable roles, check which delegatable roles are assigned to `Alice`, and if a particular role is a delegatable role via the following method invocations:

- `rm getAllDelegatableRoleList` 
- `rm getAllDelegatableRolesAssignedToSubject Alice` 
- `rm isDelegatableRole Secretary` 
- `rm isDelegatableRole delegatableSecretary` 

When assigning a permission to a regular role one needs to decide if it is delegatable or not. In our example, permission `DEL_Document` is delegatable, while `PUT_Document` is not delegatable for owners of the regular role `Professor`.

We can now delegate the permission `DEL_Document` to her delegation role `ExtendedSecretary`.

In order to delegate a permission it needs to be assigned to a delegatable role of the delegating subject. Therefore, `Alice` can delegate the permission `DEL_Document` to her delegation role `ExtendedSecretary`:

When delegating a role, the delegatable role is delegated despite of entering the name of the regular role. Thus, `Alice` delegates the permissions assigned to `delegatableSecretary` (i.e. all delegatable permissions assigned to the role `Secretary`) to her delegation role `ExtendedSecretary`:

![Figure 5. Example RightsManager object](image-url)
Figure 5 visualizes the example RightsManager object rm in xoRBAC after assigning subject Bob to the delegation role ExtendedSecretary.

5 Conclusion

In this paper, we documented our experiences when implementing two complementary delegation models into an access control management system. In particular, we extended the software component xoRBAC to support flexible role- and permission-based delegation and revocation. The basic delegation package offers straightforward user-to-user delegation, while the extended delegation package allows security administrators to limit the delegation possibilities. Thus, the models add elements of discretionary access control into the role-based access control management system. Moreover, the main features of our delegation extensions also include the creation of delegation roles for single- and multi-step delegation, simple and cascading revocation of multi-step delegation for delegation roles, the delegation and revocation of single permissions and complete roles, as well as the definition of delegation constraints on permission- and role-level. Our second delegation extension adds the feature to distinguish between delegatable and non-delegatable permissions and roles. However, both delegation models provide great freedom to the users which can be exploited by malicious users.

In future work, we aim to reduce the conceptual complexity of our implementations to make our delegation models feasible also for large role collections. Moreover, our delegation models only cover a part of potential delegation scenarios. For example, both models require that a subject can only delegate permissions owned by this subject. Furthermore, only the creator can delegate roles and permissions to a delegation role which is not possible in case of an unexpected absence of the corresponding creator. This problem could be addressed by transferring the delegation authority to a deputy who can perform the delegation actions instead of a particular delegator. In addition, we plan to provide further administration and control facilities for security administrators.

References


