Separation of Duty in Role-Based Environments

Richard T. Simon
The Open Group Research Institute
Eleven Cambridge Center
Cambridge, MA 02142
r.simon@opengroup.org

Mary Ellen Zurko
The Open Group Research Institute
Eleven Cambridge Center
Cambridge, MA 02142
m.zurko@opengroup.org

Abstract

Separation of Duty is a principle that has a long history in computer security research. Many computing systems provide rudimentary support for this principle, but often the support is inconsistent with the way the principle is applied in non-computing environments. Furthermore, there appears to be no single accepted meaning of the term. We examine the ways in which Separation of Duty has been used, adding the notion of History-based Separation of Duty. We assess ways in which computing systems may support Separation of Duty. We discuss the mechanisms we are implementing to support Separation of Duty and roles in Adage, a general-purpose authorization language and toolkit.

1. Introduction

Separation of Duty is a security principle used to formulate multi-person control policies, requiring that two or more different people be responsible for the completion of a task or set of related tasks. The purpose of this principle is to discourage fraud by spreading the responsibility and authority for an action or task over multiple people, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual. A frequently used example is the process of creating and approving purchase orders. If a single person creates and approves purchase orders, it is easy and tempting for them to create and approve a phony order and pocket the money. If different people must create and approve orders, then committing fraud requires a conspiracy of at least two, which raises the risk of disclosure and capture significantly.

Although Separation of Duty is easy to understand, it is hard to express this principle in computer security systems. Early work in computer-supported Separation of Duty focused on mechanisms that were easy to implement and turned out to be rigid and unrealistic. Mechanisms that support flexible commercial policies embodying Separation of Duty must incorporate the principle of user-centered security [17], a principle that has only recently started to be examined in depth. Our work on the Adage toolkit for authorization in distributed applications [15] is grounded in the user centered security philosophy. In supporting the policies that users want, it provides support for a variety of Separation of Duty variations.

In this paper, we will discuss prior work done on Separation of Duty and similar concepts in computing systems, starting with the earliest references and ending with current research in the context of role-based access control. We then characterize role-based environments with an emphasis on those concepts needed to define Separation of Duty variations. Next we outline different kinds of Separation of Duty variations, and discuss the mechanisms needed to implement those policies. We describe the mechanisms in Adage that support these variations at both the user interface and the architectural level and tie together the variations and mechanisms with some examples of how Separation of Duty variations are implemented in Adage. We summarize the results of this paper in the final section.

2. Prior work in separation of duty

Separation of Duty is a foundational principle in computer security. In 1975, Saltzer and Schroeder [9] defined "separation of privilege" as one of the eight design principles for the protection of information in computer systems. They credit R. Needham with making the following observation in 1973: a protection mechanism that requires two keys to unlock it is more robust and flexible than one that requires only a single key. No single accident, deception, or breach of trust is sufficient to compromise the system.

Clark and Wilson's commercial security policy for integrity [4] identified Separation of Duty as one of the two major mechanisms to counter fraud and error while ensuring the correspondence between data objects within a sys-

0-8186-7990-5/97 $10.00

tem and the real world objects they represent. At the policy level, processes were divided into steps, with each step being performed by a different person. Thus Separation of Duty is tightly tied to application semantics or commands. Clark and Wilson suggested further safeguarding against collusion by random selection of the sets of people to perform some operation, so that any proposed collusion is only safe by chance.

At the formal model level, Clark and Wilson tasked the security administrator with the job of maintaining Separation of Duty requirements while granting users the ability to run Transformation Procedures against Constrained Data Items. Since this difficult task was done by hand, it was rarely repeated and encoded a static representation of who may perform which actions. Moreover, the model specified that an agent who can certify the use of a Transformation Procedure (TP) may not execute that TP — but the identity of the certifier was not part of the model, so this was another Separation of Duty constraint that the security administrator had to remember and enforce.

Baldwin's Named Protection Domains (NPDs) [1] implemented many of the concepts used in role-based systems today. An NPD was a named, hierarchical grouping of database privileges and users. To help enforce Separation of Duty, a user could have only one of these NPDs activated at any time. The security administrator determined which NPDs could be activated, but there were no further restrictions on the graph of NPDs (other than it be acyclic). Thus, one activatable NPD could contain multiple activatable and non-activatable NPDs. While the activation restriction meant that a user could be in only one role (NPD) at a time, the security administrator could set up arbitrarily complex roles.

Sandhu's work on Transaction Control Expressions [10][11] introduced notation for Dynamic Separation of Duty. Roles were used to specify who can issue which transaction steps (much like NPDs). However, in Sandhu's model each user executing a step in a transaction had to be different. To enforce this, the history of the execution of each transaction was maintained. The constraints specifying the roles that could execute each step were associated with an object. These constraints turned into the history specifying which user executed each step on that object. Hierarchical roles were specified either on an object or via a global notation. A weighted voting syntax allowed the specification of multiple person authorizations on a particular step on a particular object.

Nash and Poland's study [8] of a portable security device used in the commercial world raised a number of new issues around Separation of Duty. The system defined two disjoint groups of authorizing officers, and each day one or two officers from each of those groups were chosen as officers of the day. This was the first example of the utility of specifying cardinality for a particular role and of time-based roles. Each transaction had to be authorized by one of those officers from each of the two groups. While the device enforced Separation of Duty between the manager who chooses officers and the officers themselves, the documentation suggested these roles could be shared in small companies.

In their general discussion of Separation of Duty, Nash and Poland stated that neither TCSEC [5] mechanisms nor Clark and Wilson's original proposal allowed implementation of common commercial schemes for Separation of Duty. Nash and Poland proposed the notion of "object based Separation of Duty," which forced every transaction against an object to be by a different user. They suggested using Sandhu's Transaction Control Expressions [10] to maintain the history of an object's transactions.

Work by Sandhu and others on defining role-based access controls (RBAC) with more precision [13][14] acknowledged that there might be a need for users to hold multiple roles (that are not connected hierarchically) at the same time. They recognized a need for a way to place limits on how more powerful roles are combined with less powerful ones. They also suggested mechanisms such as time constraints on roles to limit the amount of damage that might be done with misuse or intrusion.

Ferraiolo, Cugini, and Kuhn's paper on RBAC [7] presented the beginnings of a formal model of RBAC. They defined three kinds of Separation of Duty. The first two were Static Separation of Duty and Dynamic Separation of Duty. These variants were presented in previous work. The third kind was Operational Separation of Duty which introduced the notion of a "business function" and the set of operations required for that function; a business function resembles the notion of task and task unit in [16]. The formal definition of Operational Separation of Duty stated that no role can contain the permissions to execute all of the operations necessary to a single business function. This forces all business functions to require at least two roles to be used for their completion. The informal description of Operational Separation of Duty assumes the roles involved have disjoint memberships (Static Separation of Duty), so that no single person has access to all the operations in a business function.

3. Role-based environments

In a role-based environment [7][13][14], a security administrator controls access by assigning users (or their authenticated counterparts *principals*) to security policy constructs called *roles*. Defining a role includes defining three groups:

- a group of users or principals who may act in the role
- a group of operations or actions comprising what may be done in the role
- a group of objects or targets that may be acted upon

The first level of access control is the definition of these groups. Defining the group of actions and the group of targets is a first approximation to an on-line representation of a job. Placing a user in the group of users designates that user as one who does that job and, in the absence of other constraints, may perform all the designated actions on all the designated targets. It is these additional constraints that capture the full flavor of human organizations and their security policies. Three kinds of constraints may be identified:

- **Constraints on role membership**: overlap in membership is constrained (usually to be null)
- **Constraints on role activation**: legitimate users of a role may be prevented from assuming the role (e.g., a cardinality constraint may prohibit more than 2 users from being in the role at once)
- **Constraints on role use**: users who have assumed a role may be restricted in how it is used (e.g., history constraints may restrict which objects may be operated on, or what may be done to the objects)

Constraints on *role membership* concern rules about how the group definitions of different roles may overlap. Constraints on memberships are used to define the simplest Separation of Duty policy, Static Separation of Duty.

Constraints on *role activation* concern when a user may actually begin acting in the role (*assuming* the role), and when a user leaves a role (*role deactivation*). Often some form of explicit authentication or role-assumption dialogue is required, though explicit switching can be burdensome in many cases [13]. A user who is a member of the group permitted to assume a role may still be denied entry to the role if some constraint is not satisfied. Control over activation is used in some more flexible Separation of Duty policies.

Constraints on *role use* concern what a user may do in a role once the role has been assumed, beyond the coarser control provided by defining the action and target groups. The finer controls involve the specific history of a given user with a given target. Constraints on role use are used in the most flexible (and generally realistic) Separation of Duty policies.

In summary, role-based environments group users, actions, and targets into policy constructs called roles. These environments control access by controlling membership in, activation of, and use of roles. In the next section, we categorize different kinds of Separation of Duty policies based on their use of these kinds of controls.

4. Separation of duty: theme and variations

Many authors have discussed and categorized different forms of Separation of Duty [7][8][10][11], but no single source has yet enumerated all the various forms. In this section we describe all the variations of Separation of Duty that have been called out in one source or another.

The two broadest categories of Separation of Duty variations are *strong exclusion* or *Static Separation of Duty* and *weak exclusion* or *Dynamic Separation of Duty.* Strong exclusion represents a single variation whereas the more realistic and useful weak exclusion contains several variations.

4.1. Strong exclusion (Static Separation of Duty)

Strong exclusion is also called Static Separation of Duty and is the simplest variation of Separation of Duty. Two roles are strongly exclusive if no one person is ever allowed to perform both of these roles. In other words, the two roles have no shared principals. If Order Creator and Order Approver are strongly exclusive roles, then no one who may assume the Creator role would be allowed to assume the Approver role, and no one who may assume the Approver role would be allowed to assume the Creator role. Strong exclusion can be implemented using only controls over the membership of roles.

Though it has the advantage of simplicity, Static Separation of Duty is not a practical or realistic variation of Separation of Duty, because it does not reflect the actual functioning of human organizations [8]. Users often have legitimate reasons for wanting or needing to act in two strongly exclusive roles, and careful construction of a security policy can ensure that these "violations" are secure.

Because Static Separation of Duty is too rigid to be a satisfactory control, a number of other variations have been defined that more closely mimic the functioning of human organizations.

4.2. Weak exclusion (Dynamic Separation of Duty)

Weak exclusion, or Dynamic Separation of Duty, provides a richer set of possible policies by controlling the activation and use of roles. Weak exclusion allows users to act in roles that would be strongly exclusive in static sys-

tems, as long as constraints are satisfied that eliminate or reduce the possibility of fraud. Because of its usefulness and because it reflects the functioning of human organizations (which tend to be more fluid than strong exclusion permits), weak exclusion has several variations.

In describing the variations of weak exclusion we will use the term *restricted roles* to refer to roles that have constraints on their membership, activation, or use. The actual constraints used or allowed depends on the variation of weak exclusion being discussed.

Simple Dynamic Separation of Duty. In the simplest variation of Dynamic Separation of Duty, restricted roles may have common members, but users may not assume both roles at the same time. This variation by itself is sometimes called Dynamic Separation of Duty [7].

Object-based Separation of Duty. Restricted roles may have common members, and those members may assume both roles at the same time, but no user may act upon a target that that user has previously acted upon. This is called *Object-based Separation of Duty* [8].

Operational Separation of Duty. Restricted roles may have common members as long as the union of all the groups of actions in the roles does not contain all the actions in a complete business task (like the processing of an order). This prevents any one person from performing all of the actions in the business task. This is called *Operational Separation of Duty* [7].

History-based Separation of Duty. While both Object-based and Operational Separation of Duty respond to deficiencies in the simpler variations, they still do not allow the expression of some desirable Separation of Duty policies. Object-based Separation of Duty does not allow a user to perform a second action on an object when this makes sense and is allowed by (human) policy, and Operational Separation of Duty does not allow one user to perform all the actions in a task to different objects when this makes sense and is allowed by (human) policy. To permit the most complete flexibility in describing a Separation of Duty policy, these two variations should be combined.

Two or more restricted roles may have common members and the union of the actions granted by those roles may span the actions in a business task, but no role member is allowed to perform all the actions in a business task on the same target or collection of targets. We call this *History-based Separation of Duty* to emphasize the central part played by the individual histories of users in determining access. Sandhu [10] emphasizes the importance of history in defining Separation of Duty policies and provides some examples of policies that use the full generality of History-based Separation of Duty (though he does not use that term).

To make our definition of History-based Separation of Duty as general as possible, we note that such policies define sequences of allowed or required steps and that each step may consist of order-dependent or order-independent actions:

Order-dependent. In some situations, the fraud preventive nature of Separation of Duty controls is not implemented simply by splitting duties between roles; the roles also must perform their actions in a particular order. For example, a purchase should be approved only if it has been *previously* created properly (all the blanks are filled in). These are order-dependent actions.

Order-independent. Sometimes order is not important. For example, suppose the policy states that two different approvals are required to make a purchase transaction valid. The order of approvals is not important (so long as both happen after the purchase has been created), but they must both happen. These are order-independent actions.

A single Separation of Duty policy may include both order-dependent and order-independent parts, as in the example just given in which the order-independent approvals must both be given after an order creation. Sandhu [10] also gives examples of policies that mix order-dependence and independence.

5. Required mechanisms for separation of duty

The survey of published literature on Separation of Duty and the categorization of the variations of Separation of Duty reveal a number of mechanisms essential to the formulation of a general-purpose policy builder in a role-based environment.

5.1. Groupings

Defining roles for Separation of Duty requires that principals, actions, and targets be grouped in meaningful ways. Groupings should be hierarchical so that levels of meaning can be captured. For instance, it should be possible to group all the principals belonging to a single user and then to use that group as part of the definition of a role's membership. This grouping embodies the notion that duties or tasks must be separated between users, not just principals. The ability to group things meaningfully is the first level of control in role-based environments.

5.2. Membership controls

Membership controls limit how the groups (in particular, principal groups) in different roles may overlap. These are the first of the finer grained controls in role-based environments. A common requirement is that the principal groups of two roles contain no common members.

5.3. Activation controls

Activation controls limit when a principal may assume a role. These controls generally concern how many principals may simultaneously be in the role, or which roles may be active simultaneously. We categorize an activation control as either a mutual exclusion control or a cardinality control.

A *mutual exclusion activation control* prevents a principal from activating two roles simultaneously. The distinction between this kind of activation control and the membership control that prohibits overlap is that the membership control prevents a user from even being placed in two excluded roles while the activation control permits a user to be in the two roles but not at the same time. The membership controls are static and are applied only during role definition; the activation controls are dynamic and applied during system operation.

A *cardinality activation control* prevents more than N different principals from being active in a role simultaneously or requires that at least N principals be active in the role.

5.4. History controls

History controls limit how activated roles may be used based on the history of the principal. The specific history a principal has with a target may limit the actions that principal may perform on that target or some other target.

History is a three-dimensional bit matrix (conceptually). The dimensions of the matrix are principals, actions, and targets; an entry in the matrix indicates whether that principal performed that action on that target. This three dimensional matrix is a generalization of the matrix used in defining the Chinese Wall policy [3].

5.5. Labels

Labels are used to mark entities with the information required to perform an access decision. In role-based environments, labels can be used to hold the names of roles that are pertinent. For actions and targets, these are the roles in whose groups they have been placed. For principals, two kinds of labels seem appropriate: one containing all the roles in which the corresponding user may act and one containing all the roles that are currently activated.

Sandhu [12] points out the importance of keeping distinct the notions of user, principal, and process (called a subject in [12]) to avoid unnecessarily restrictive restraints on information flow. In his model principals and processes have fixed labels, while user labels may float "up" a lattice. Our description above generally follows this suggestion, though we suggest allowing principal labels to float, placing the onus of containing information flow on the process level. We will elaborate on these points later when we describe the Adage architecture.

5.6. Other mechanisms

Two other mechanisms, auditing and authentication, are required to support separation of duty in role-based environments, but they are ancillary to the main topic of this paper and will not be discussed further.

6. Adage mechanisms

Adage [15] is a project at the Open Group Research Institute exploring authorization support for distributed applications. Its primary goal is to make the design and implementation of computer-based authorization policy easier and richer for both distributed applications and the security administrators that manage them. It provides a visual authorization language for security administrators and a textual authorization language for more advanced administrators and application writers. The authorization languages are designed to make it easy to implement a wide variety of policies found in both the security literature and real life. Adage interoperates with external authentication services (such as DCE, Kerberos, or public key certificates), and can use identity, delegation, and user attribute information from these authorities in its policy definitions and authorization decisions. In this section, we discuss those features of Adage at the policy (user interface) level and at the architectural level that are designed to support a variety of Separation of Duty policies.

The descriptions of Adage mechanisms that follow are high-level platform-independent descriptions. They conform to the top two tiers of the three-tier model suggested by [13]. The policy level mechanisms are the external or user view of Separation of Duty policies, expressed in terms of security policy. The architecture level mechanisms are the common or composite view of Separation of Duty policies. We do not discuss the third tier, implementation view, in this paper.

6.1. Policy level

Adage provides a Visual Policy Builder (VPB) and textual Authorization Language (AL) designed to support security administrators' implementation of their site security policy [15]. The VPB and AL are designed to be flexible enough to specify a wide variety of real and useful authorization policies. Using the principle of user-centered security [17], they present administrators with familiar building blocks and a context that allows them to build up and query their policy definitions incrementally. Adage is policy-neutral and allows many kinds of policies including Role-Based Access Control (RBAC) to be specified. In this section, we discuss the user interface primitives that security administrators may use to implement Separation of Duty and role-based policies in Adage.

Groupings. Users, who may be associated with more than one principal, are represented at the policy level by *actors*. Actors are necessary for capturing Separation of Duty policies that rely on more than one user participating in an action or task. There is no way within the system to be certain that all recognized principals belonging to a single user are associated with that user's actor. Security administrators must assure themselves of this out-of-band. However, the notion of actor gives administrators a simple primitive for applying consistent policies to multi-principal users.

Adage extends the notion of groups in other ways. Groups traditionally contain users (and other groups of users). We apply the power of meaningful groupings to the three fundamental building blocks of high-level policy definition: actors, actions, and targets. Groups may be either homogeneous or heterogeneous at the policy level.

Role-based policies in Adage. A role-based policy can easily be built on top of Adage groups. The core of role-based policies defines a role as a means of associating a group of users with a group of permissions that they are granted. Separation of Duty semantics are then applied to the defined roles, as described in Section 3. A role is a heterogeneous group in the Adage high level policy language. It has two groups associated with it; a homogeneous group for the actors who are granted the role (called a *team*), and a heterogeneous group for the permissions that the role grants. Separation of Duty constraints are applied to the homogeneous group of actors.

The permissions granted by the role may be specified at any granularity by grouping actions and targets in that permission grouping. There may be a group containing a single action and one or more specific targets that action can be applied to, as in the Clark and Wilson model. Or the permission may be an application action that can be applied to all targets (for which that action is legal). These general permissions may be grouped to form even more general permissions. For example, a generic READ permission can be constructed by grouping all the permissions for application actions that only read their targets.

Separation of Duty features in Adage. Figure 1 outlines the features at the user interface level of Adage that support various kinds of Separation of Duty policies.

Adage Mechanism	Category of Separation of Duty
NoOverlap	Static
CantHoldSimultaineously	Dynamic
AtMost	Dynamic
HasDone	Order dependent history-based
NeverDid	History-based
NeverUsed	History-based
DifferentOneFromEach	Object-based
SomeoneFromEach	Order-independent history-based

Fig. 1 Adage Mechanisms for Separation of Duty

Teams may have constraints applied to them. Any set of teams can be marked as mutually exclusive (using the **NoOverlap** constraint), so that no actor can be a member of more than one team in that set (either directly, or indirectly through the hierarchy of team memberships). This constraint supports Static Separation of Duty. Alternatively, a set of teams can be defined such that no actor may be active in more than one team of the set (using the **CantHoldSimultaineously** constraint). This constraint supports Simple Dynamic Separation of Duty. The maximum number of members of a team may also be constrained (**AtMost**) to support cardinality constraints for Separation of Duty policies.

To support a variety of history-based policies, Adage supports constraints on groups of actions that gate the use of those actions based on previous authorizations. Validity times can be attached to action groups, so that they may only be exercised at certain times of the day, month, or year, or their use may expire after a certain interval. Adage's **HasDone** constraint allows an authorization if the other action specified in the constraint has been executed by the specified user on the specified target (some initial authorization must be allowed regardless of history to start the process). The user specified may be a particular user (a system administrator, the same user requesting the authorization) or it may be any user with a particular relationship to the requesting user. The target may be specified in the same way. The **NeverDid** constraint is similar to the **HasDone** constraint, only it allows the authorization if the specified event has never occurred. The **NeverUsed** constraint is the same as the **NeverDid** constraint, but it checks that some actor never executed any action on some target.

These historical constraints allow the administrator to build up a simple notion of a task sequence by specifying that the authorization for each action within a task depends on the previous action occurring. For instance, for a task consisting of preparing a check, approving the check, and issuing a check, the authorization of the approval will not be allowed unless the check has been prepared, and the issuing will not be authorized unless the approval of the same check has occurred, both using the **HasDone** constraint.

This simple form of task sequence does not capture the richer if/then/else and looping structure of task-based authorization [16]. This is because Adage is designed to present a fully general, policy neutral, high level authorization language. The richer task-based authorization requires the notion of task to be shared between the authorization system and the application, and is currently tied to a particular workflow product [6][16]. The Adage authorization language may be used with any distributed application, with a minimal amount of application-specific work.

Some forms of Separation of Duty require approvals or authorizations from more than one person (as in Nash and Poland's commercial security device example). Actions that require multiple actors to authorize them can be formed in Adage with the constraints **DifferentOneFromEach** and **SomeoneFromEach**. The **DifferentOneFromEach** constraint will require a different actor from each team specified. The most straightforward policy formed from this constraint would require a different actor from each of two teams (team A and team B) to approve a transaction. The **SomeoneFromEach** constraint relaxes the requirement that the actor representing each of the teams must be different allowing, for instance, the manager who selects the officers of the day to come from the group of all candidate officers, as discussed in Nash and Poland.

Enabling and disabling privileges. Separation of Duty policies need the ability to enable and disable privileges. For example, if a user can act in a number of roles, the system needs to determine under which the user is acting. While it is most convenient for users to have all of their groups and roles enabled at all times, this can increase the damage a trojan horse can do. In addition, some roles (such as security administrator, manager, and personnel representative) are more powerful than others.

The trade-off between usability and security is a difficult one for dynamic groups and roles. Adage allows security administrators to mark the roles that are powerful with an activation constraint. Adage will automatically activate roles as they are needed by the user. However, activating constrained roles must be verified by the user via a trusted path interaction with Adage. Roles that must be explicitly activated cannot unknowingly be enabled or used by a user because of a trojan horse. Security administrators should apply this activation to particularly powerful roles. Adage allows three kinds of acknowledgment requirement:

- Acknowledge always — the user must acknowledge every time an action in the role is performed
- Acknowledge on activation — the user must acknowledge the first time an action in the role is performed
- Acknowledge never — this is the default situation in which a user assumes the role automatically upon performing any action in the role's action set

For example, Nash and Poland [8] documented a policy where clerks entered transaction data, but transactions were not final until authorizing officers accepted them. Clerks were not even authenticated (although the necessity of entering data on a hand held device may have provided some physical security). The damage a clerk running a trojan horse could do was considered negligible, so assuming a Clerk role would not require any explicit acknowledgment from the user. On the other hand, the damage an authorizing officer running a trojan horse could do would be great, since it could easily submit the two authorizations required by the policy. In Adage terms, the Authorizing Officer role might require acknowledgment whenever it was used for an authorization.

In addition, Adage provides a tool that allows users to monitor which teams are enabled, in a compact visual format. Users are not expected to track exactly which teams are being used so much as notice when something surprising happens (a change in the pattern or a whole new pattern). This enlists users in monitoring their own security, which provides greater assurance of trojan horses being noticed. Adage also audits all actions on the authorization policy state and authorization requests from applications, allowing traditional centralized monitoring of audit information as well.

6.2. Architecture level

The user-visible policy building mechanisms in the VPB and AL are implemented using platform-independent architectural mechanisms. The architectural constructs discussed in this section support the VPB and the AL and are also platform-independent.

Basic entities. The basic entities at the implementation level are the principal (authenticated entity), the action (operation), and the target (object). Actors are not explic-

itly supported at the architectural level because the architectural level deals only with authenticated entities in making access decisions.

Groups. Each of the three basic entities may be placed in groups of like entities. A group of principals is called a *team*, a group of actions is called an *action set*, and a group of targets is called a *collection*. Groups may contain other groups of the same kind; cycles are not allowed.

A role has one group of each kind: a team, an action set, and a collection.

Labels. Labels are used to mark principals, targets, and actions with the roles that are relevant to access decisions. In most cases, the roles in the label are just those to which the entity belongs. The roles form a non-hierarchical set of names like those used in labels for policies like Bell and LaPadula [2]. (A complete Adage label also contains hierarchical confidentiality and integrity levels. We do not consider those parts of the label in this paper.)

Target and action labels are straightforward, while principal labels are more involved. We discuss each in turn.

1) Target labels

Targets are labeled with the names of the roles in whose collections the target resides. These labels are static and change only when an administrator changes role definitions. New targets receive the label of the action that creates them.

2) Action labels

Action labels are like target labels; they have the names of the roles in whose action sets the action resides, and they change only when an administrator changes role definitions. New actions may be added by changing the existing policy to place them in appropriate roles.

3) Principal labels

Principals have two kinds of labels: a *label closure* and a *current label*. The label closure is exactly like the target and action labels; it contains all the roles to which the principal belongs. The current label contains the roles that the user has active; this is a subset of the roles in the label closure. Allowing the principal's current label to contain more than a single role is an important usability feature since it mirrors how people use roles in actual job situations [8][12].

The Adage label design is similar to that suggested in [12] which points out the importance of keeping distinct the notions of user, principal, and process (called a subject in the cited paper). The problem is avoiding unnecessary constraints on how people use their roles while still preventing insecure information flow. In the cited paper, user's labels are allowed to float "up" the label hierarchy, gathering greater privilege, while principal and process labels remain fixed.

At the VPB level, actors are the surrogates of users and each user has one or more principals, each principal being associated with one user. An administrator defines the set of labels over which a user may range by assigning the user's actor or individual principals to roles. While the actor may range over all permissible values (since it represents all the user's principals), principals may be more constrained by assigning them to only a few or one role. Assigning a principal to one and only one role would be equivalent to a policy in which principals have fixed labels. In such a policy, a user would have to authenticate as a different principal for each role. The ability to allow principals to have more than one permissible label opens up more flexible and easier to use policies.

At the architectural level, the set of principals is the user surrogate; actors are not supported at this level. A principal's label floats up as more roles are activated, within the permissible range defined by the policy for each principal. The containment of information flows using fixed labels is achieved at the implementation level where processes (which are the entities that actually contain information) inherit the current label of the principal and use that as a fixed label. We do not describe the implementation level details further in this paper.

One problem with allowing principal labels to float is that targets created by the principals would have labels that contain too many role names if they inherited their labels from the creating principal. Adage applies the label of the action that creates the target to the new target. This means that any role having access to that action has (potentially) access to the new target, and this set of roles will be some subset of the roles in the principal's current label.

Rules. Rules contain the expression of policy at the architectural level. Rules compare the principal, target, and action labels to determine if access is allowed.

Rules have five parts:

- The *target scope* determines the targets to which the rule applies. The scope is an expression involving role names and any target whose label satisfies the expression is within the scope of the rule.
- The *principal scope* determines the principals who may access the targets, if the constraints specified in the relation (below) are satisfied.
- The *action scope* determines what actions the principals are allowed to perform on the targets.

- The *relation* expresses any additional constraints that must be met to be granted access, beyond being included within the principal scope. These include activation and history controls.
- The *date* determines when a rule is in force

Role creation. Creating a role creates three groups (team, action set, collection), and a default rule that allows any member of the team to assume the role at any time and perform any action in the role's action set on any target in the role's collection. Finer grained control is applied by creating rules that modify activation and use of the roles.

Role activation and deactivation. In the simplest case, a user activates a role by performing an action in the role's action set. In more complex cases, the activation may require other constraints to be satisfied. These constraints would be expressed in rules about the mutual exclusion or cardinality of the roles, or the history of the principal with the given target.

Adage does not deactivate roles until a user logs out, authenticates as a new principal, or explicitly requests deactivation. This means that roles are allowed to accumulate as a user does more and more things. This is reflected in the current label of the user (actually, the user's current principal); the label floats "up" the label hierarchy as more roles are activated. This approach does not lead to the usual problems associated with accumulating power because the history-based rules allow the individual access of each user to be constrained based on what they have done. So, a user who has all their roles active will be no more powerful than they would be if we required them to (laboriously) activate and deactivate roles. They are permitted to perform the same actions on the same targets in either case.

However, the issue of least privilege does pertain to risks of user error or trojan horses. The possibility of a trojan horse (one that, say, approves an order created by the trojan horse's creator, without the knowledge of the supposed approver) can be contained by requiring acknowledgment of role activation or role actions. For instance, one may require that all Approver actions be acknowledged. This is an instance where the user-centered principle must be balanced against other security principles. Automatically activating roles increases the potential damage of user error or trojan horses, but forcing explicit activation and deactivation of every role increases the likelihood that users will seek out and use methods and tools to turn off or circumvent onerous security procedures. No one answer will satisfy the needs of all sites. Any given policy maker must choose a level of (in)convenience that is appropriate for their environment and its risks and level of user discipline.

Deactivating roles under the same principal must be done properly to avoid a flow violation. In Adage this is handled at the implementation level by restricting processes to a fixed label. When a principal goes "down" the label hierarchy, the processes it has spawned do not follow it; their labels remain where they were and new processes will be used to do work at the new, lower level.

How access is computed. In this section we give a brief overview of how Adage makes an access decision.

Adage first determines the roles involved in the decision. An access decision involves a principal, a target, and an action. Each of these has a label containing the names of roles to which each of the entities belongs. Adage can determine the roles involved by examining the labels of each entity. We will call the role names in a label the *role set.*

If the requested action can never be performed on the target, then the target's label will not contain the name of any role in which the action resides. So, if the intersection of the role sets for the target and action labels is null, then the access is not allowed.

Similarly, if the intersection of the role sets of the principal's label *closure* and the action's label is null, then the principal is not a member of any team that can perform the action and the access is not allowed.

At this point, Adage knows that in some cases the principal may perform the desired action, and in some cases the desired action may be performed on the target. However, it must now check rules to determine if the detailed policy of the system permits the specific access attempted.

Adage determines which rules affect the access decision by using the target scope to select rules that apply to the target in question, and in turn using the principal scope to determine if any of these rules apply to the principal. In addition, the date portion of the rules are examined to eliminate any that are inactive at the time of the request. If no rules are found, access is denied.

Once the applicable set of rules is known, the relation in each of them is checked. A rule fails if the principal fails to meet the constraints in the rule's relation. If all rules succeed, then access is granted. If the access is permitted, then the principal's current label is augmented with the role name if that name is not already present in the label.

7. Examples

In this section, we present some Separation of Duty examples encoded in our rule format. We present our rules in the following way:

Users in **Team** <Team-name>
may perform **Action** <Action-name>

on **Targets** in **Scope** <Scope-name>
if <**Relationship**> (1)

This format was chosen for readability. In all the example rules, the date component is omitted and assumed to be void (indicating the rule is always applied).

7.1. Strong exclusion

Strong exclusion may be implemented using the **NoOverlap** constraint in the authorization language (see Figure 1) to prevent actors and their corresponding principals from being placed in more than one of a set of strongly excluded roles. This constraint is checked at the time the policy is constructed in the VPB (or the AL), so it is not checked later during the decision process.

7.2. Weak exclusion

Strong exclusion is usually too rigid to reflect real-world security policies. Weak exclusion is more realistic. Weak exclusion permits the same person to act in two restricted roles, as long as certain constraints are met.

Simple Dynamic Separation of Duty. In Simple Dynamic Separation of Duty the same person may act in restricted roles but only at different times.

Simple Dynamic Separation of Duty may be implemented by a rule that permits a principal to enter a role if the principal is not already in the restricted role.

Users in **Team** <Teller>
may perform **Action** <TellerActions>
on **Targets** in **Scope** <TellerTargets>
if <Auditor **NOT IN CurrentLabel**> (2)

Users in **Team** <Auditor>
may perform **Action** <AuditorActions>
on **Targets** in **Scope** <AuditorTargets>
if <Teller **NOT IN CurrentLabel**> (3)

Rule 2 simply states that a user who is authorized to perform Teller actions (contained in the action set TellerActions) may do so if they do not already have the Auditor role active (indicated by the presence of "Auditor" in the user's current label). Rule 3 says the same thing about the Auditor role.

Operational Separation of Duty. In Operational Separation of Duty the same person may act in restricted roles as long as the complete set of actions permitted by the roles does not span an entire business task.

Adage supports the notion of business task by allowing the administrator to make the authorizations for each sequential action in a task depend on the previous action having been authorized and executed. Example 7 shows a rule for sequencing one task action after another one. Administrators can encode Operational Separation of Duty by dividing the actions in a task among different mutually exclusive roles.

Object-based Separation of Duty. In Object-based Separation of Duty the same person may act in restricted roles as long as that person does not act on the same object in each role.

Object-based Separation of Duty may be implemented by modifying the rules for Simple Dynamic Separation of Duty.

Users in **Team** <Teller>
may perform **Action** <TellerActions>
on **Targets** in **Scope** <TellerTargets>
if **<THIS USER NEVERDID** <AuditorActions> **TO THIS TARGET>** (4)

Users in **Team** <Auditor>
may perform **Action** <AuditorActions>
on **Targets** in **Scope** <AuditorTargets>
if **<THIS USER NEVERDID** <TellerActions> **TO THIS TARGET>** (5)

History-based Separation of Duty. In History-based Separation of Duty the same person may act in restricted roles as long as that person does not perform all the actions in an entire business task on the same target or collection.

History-based Separation of Duty may be order-dependent or order-independent. If the exclusion is order-dependent, then mutually excluded actions will have to be performed in a particular order. If the exclusion is order-independent, then the actions may be performed in any order.

Order-dependent history-based Separation of Duty. An example of order-dependent History-based Separation of Duty occurs often in the creation and approval of purchase orders. These two duties are often separated so that two people are required to create and approve a purchase order. Strong exclusion could be used to accomplish this, forcing order creators and order approvers to be disjoint sets of principles, but this is too restrictive and does not reflect real-world operation. Usually the same people may both create and approve orders, but may not approve an order that they themselves created. Both Simple Dynamic Separation of Duty and Operational Separation of Duty fail to provide the desired policy. Simple Dynamic Separation of Duty does not prevent the same person from creating an order and approving the same order at a different time. Operational Separation of Duty does not allow the same person to perform all the actions of an order creator and order approver since it does not allow these roles to share members because they span a business task. Only History-based Separation of Duty allows the full flexibility desired while preventing the creation of fraudulent orders by a single person.

If C is a team of order creators and A is a team of order approvers, then order-dependent exclusion can be expressed using history relations:

Users in **Team** <C>
may perform **Action** <CreateOrder>
on **Targets** in **Scope** <PurchaseOrders>
if <void> (6)

Users in **Team** <A>
may perform **Action** <ApproveOrder>
on **Targets** in **Scope** <PurchaseOrders>
if <**OTHER**(C)**HASDONE** CreateOrder **TO TARGET**> (7)

These rules have three effects:

1. They allow anyone in the authorized team of creators C to create a purchase order.
2. They require that someone other than the creator of the purchase order approve the order
3. (order-dependence) They require that a purchase order be created by an authorized creator before it can be approved (eliminating the possibility that an authorized approver could circumvent policy by constructing bogus order and approving it).

Order-independent history-based Separation of Duty. Order-independent exclusion is a form of multi-person control in which the order of operations is not important, merely that more than one person has participated in the completion of the task. For example, the security policy presented in the last section could be augmented by requiring that two different order approvers approve each order. In this case, the order of the two approvals is not important, only that they both occur after the creation of the order.

Since an order now requires two approvals before it is valid, we must introduce some way to ensure that orders with a single approval are not treated as valid. To do this we introduce another action called ShipOrder. In the order-dependent example, ShipOrder was an implicit part of ApproveOrder; once an order was created and approved according to policy, it could be shipped. In this example, we distinguish this separate step so that orders with only partial approval will not be shipped. The rules for the new policy are:

Users in **Team** <C>
may perform **Action** <CreateOrder>
on **Targets** in **Scope** <PurchaseOrders>
if <void> (8)

Users in **Team** <A>
may perform **Action** <ApproveOrder>
on **Targets** in **Scope** <PurchaseOrders>
if <[**OTHER**(C) **HASDONE** CreateOrder] **AND** [**THIS USER NEVERDID** ApproveOrder **TO THIS TARGET**]> (9)

Users in **Team** <A **OR** C>
may perform **Action** <ShipOrder>
on **Targets** in **Scope** <PurchaseOrders>
if <[**ANY**(C) **HASDONE** CreateOrder] **AND** [**2FROM**(A) **HASDONE** ApproveOrder]> (10)

The first rule is the same as for the order-dependent case. The second rule now prevents the same approver from approving an order twice. The third rule ensures that a shipped order was validly created and approved by two different approvers. Note that any member of A or C may ship a valid order, including the creator or the approvers. This is allowed since the order has already passed through the required three person control policy.

8. Conclusions

Separation of Duty is a well-known principle with a long history in the computer security literature. As the sophistication of security mechanisms (both modeled and implemented) has increased, the number of variations on the theme of Separation of Duty has increased. No work that we are aware of has tried to bring these variations together and categorize them until now. The increasing flexibility embodied in the more recently defined types of Separation of Duty attempts to eliminate some of the unrealistic constraints implicit in the earlier, simpler definitions. These more flexible variations also get closer to supporting actual operating procedures. A notion of history is necessary for this more general approach. History must contain information about which user took what action on what target.

We have outlined the authorization mechanisms necessary to support the variety of Separation of Duty policies. These are groupings of principals, actions, and targets, controls limiting how the membership of different groupings may overlap, controls limiting what roles may be activated, history-based controls on how activated roles can be used, and labels marking the relationships between principals, actions, and targets. We then introduced Adage, a general purpose, rules-based authorization system for distributed applications. We have shown how Adage supports Separation of Duty, at both the user interface and architectural level, by implementing the mechanisms necessary for the variety of identified Separation of Duty policies.

We have discovered that supporting Separation of Duty policies demands a balancing of user-centeredness (usability) with security (particularly risk management). Achieving the right trade-off in this area requires clear goals and careful thought. Properly balancing user-centeredness and

security is necessary to achieving the goal of a secure system that is actually deployed and correctly used to provide needed protection.

Acknowledgments

The Adage work is funded by DARPA under contract F30602-95-C-0293 (This paper is Approved for Public Release Distribution Unlimited). This paper has benefited from comments by Marty Hurley and Greg Carpenter.

References

[1] Baldwin, R. W. "Naming and Grouping Privileges to Simplify Security Management in Large Database" *Proc. 1990 IEEE Symposium on Security and Privacy,* 116-132, May 1990.

[2] Bell, D.E., LaPadula, L.J. "Secure Computer Systems: Unified Exposition and Multics Interpretation" MTR-2997 Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[3] Brewer, D. F. C., Nash, M. J. "The Chinese Wall Security Policy" *Proc. 1989 IEEE Symposium on Security and Privacy,* 206-214, May 1989.

[4] Clark, D.D., Wilson, D.R. "A Comparison of Commercial and Military Computer Security Policies" *Proc. 1987 IEEE Symposium on Security and Privacy,* 184-194, April 1987.

[5] Department of Defense National Computer Security Center, *Department of Defense Trusted Computer Systems Evaluation Criteria,* DoD 5200.28-STD, 1985.

[6] Edwards, K. E. "Policies and Roles in Collaborative Applications" *Proc. CSCW 96,* November 1996.

[7] Ferraiolo, D., Cugini, J., Kuhn, D. R. "Role-Based Access Control (RBAC): Features and Motivations" *Proc. 1995 Computer Security Applications Conference,* 241-248, December 1995.

[8] Nash, M. J., Poland, K. R. "Some Conundrums Concerning Separation of Duty" *Proc. 1990 IEEE Symposium on Security and Privacy,* 201-207, May 1990.

[9] Saltzer, J.H., and Schroeder, M.D. "The Protection of Information in Computer Systems," Proceedings of IEEE, 63(9), 1278-1308, 1975.

[10]Sandhu, R. "Transaction Control Expressions for Separation of Duties" *Proc. 4th Aerospace Computer Security Conference,* 282-286, Dec. 1988.

[11]Sandhu, R. "Separation of Duties in Computerized Information Systems" *Proc. IFIP WG11.3 Workshop on Database Security,* September 1990.

[12]Sandhu, R. "A Lattice Interpretation of the Chinese Wall Policy" *Proc. of the 15th NIST-NCSC National Computer Security Conference,* 221-235, October 1992.

[13]Sandhu, R., Feinstein, H. "A Three Tier Architecture for Role-Based Access Control" *Proc. of the 17th NIST-NCSC National Computer Security Conference,* 138-149, October 1994.

[14]Sandhu, R., Coyne, E., Feinstein, H., Youman, C. "Role-Based Access Control: A Multi-Dimensional View" *Proc. 10th Annual Computer Security Applications Conference,* 54-62, December 1994.

[15]Simon, R., Zurko, M. E. "Adage: An Architecture for Distributed Authorization" http://www.osf.org/www/adage/adage-arch-draft/adage-arch-draft.ps.

[16]Thomas, R. K., Sandhu, R. S. "Conceptual Foundations for a Model of Task-based Authorizations" *Proc. of The Computer Security Foundations Workshop VII,* June 1994.

[17]Zurko, M. E., Simon, R. T. "User Centered Security" *Proc. New Security Paradigms Workshop,* September 1996.