On the Security of Delegation in Access Control Systems

Qihua Wang, Ninghui Li, and Hong Chen
Department of Computer Science, Purdue University
{wangq, ninghui, chen131}@cs.purdue.edu

Abstract. Delegation is a mechanism that allows a user A to act on another user B's behalf by making B's access rights available to A. It is well recognized as an important mechanism to provide resiliency and flexibility in access control systems, and has gained popularity in the research community. However, most existing literature focuses on modeling and managing delegations. Little work has been done on understanding the impact of delegation on the security of existing access control systems. In particular, no formal notion of security with respect to delegation has been proposed. Many existing access control systems are designed without having delegation in mind. Simply incorporating a delegation module into those systems may cause security breaches.
This paper focuses on the security aspect of delegation in access control systems. We first give examples on how colluding users may abuse the delegation support of access control systems to circumvent security policies, such as separation of duty. As a major contribution, we propose a formal notion of security with respect to delegation in access control systems. After that, we discuss potential mechanisms to enforce security. In particular, we design a novel source-based enforcement mechanism for workflow authorization systems so as to achieve both security and efficiency.

1 Introduction

User-to-user delegation, or delegation for short, is a mechanism that allows a user A to act on another user B's behalf by making B's access rights available to A. It is well recognized as an important mechanism to provide resiliency and flexibility in access control systems. For example, when a user is unable to perform a task due to sickness, he/she may delegate the privileges to another user so that the latter user can use the privileges to complete the task on time.

Delegation has received significant attention from the research community. A number of delegation models have been proposed [2, 3, 8, 16, 15, 11, 1, 7, 6] and most of them are for Role-Based Access Control (RBAC). In contrast to normal access right administration operations, which are performed centrally, delegation operations are usually performed in a distributed manner. That is to say, users have certain control on the delegation of their own rights. In order to prevent abuse, some delegation models support specification of authorization rules, which control who can delegate what privileges to other users as well as who can receive what privileges from others.

Essentially, a delegation operation temporarily changes the access control state so as to allow a user to use another user's access privileges. Due to its effect on access control states, delegation may lead to violation of security policies, especially static separation of duty policies. For instance, if roles r_1 and r_2 are mutually exclusive, a user who is a member of r_1 should not be allowed to receive r_2 from others through delegation. Such a security requirement can be enforced using delegation authorization rules.

Delegation may be viewed as a module that introduces additional functionalities into access control systems. An important class of access control systems that greatly benefit from delegation is workflow authorization system. A workflow divides a task into a set of well-defined sub-tasks (called *steps* in this paper). Security policies in workflow authorization systems are specified using authorization constraints. Example authorization constraints are "Steps 1 and 2 must be performed by the same user" and "Steps 3 and 4 must be performed by two users without conflicts of interests". The modeling of workflow authorization systems has been studied in [4, 5, 10, 14, 12]. But only [14] considers the support of delegation. In other words, many existing workflow authorization systems are designed without delegation in mind.

To enhance existing access control systems with delegation, one needs to incorporate a delegation module into those systems. A naive approach is to place the delegation module on top of the access control module, and let the delegation module handle delegation operations and manipulate access control configuration. For example, when $Alice$ delegates the role r to Bob, the access control configuration is modified so that Bob is authorized for r in the new configuration. The underlying access control module consults the access control configuration without concerning delegation. Even though such a naive approach is simple and allows reusing existing implementation of access control modules, it introduces security breaches into the system. As we point out in Section 3.1, colluding users could exploit such breaches to circumvent security policies in the access control system. Due to the decentralized nature of delegation and the fact that not all the users in the system are trusted, collusion is a threat that must not be overlooked.

Since the naive approach could be insecure, more sophisticated methods are needed to create a secure system with delegation support. Surprisingly, even though delegation is well recognized as a very useful component of access control systems, to our knowledge, no work has performed in-depth study on how to incorporate a delegation module into access control systems in a secure manner.

This paper focuses on the security aspect of delegation in access control systems. We formally define the notion of security with respect to delegation. Intuitively, if an access control system is secure, then any group of users cannot "enhance the power" (i.e. become capable to complete more tasks than before) of the group through mutual delegation within the group. To justify this intuition, by delegating her privileges to user A, user B allows A to work on her behalf. This indicates that A gains no more than what B has, and thus, A should not be able to do more than A and B together can do before the delegation operation. This further implies that, after the delegation operation, A and B as a group cannot do more than before. If a system does not have such a property, when A and B collude, they may gain extra power by delegating privileges to each other. In that case, a group of colluding users can do more than they are supposed to do with the "help" of delegation, and the system is thus considered to be insecure with respect to delegation.

The rest of the paper is organized as follows. In Section 2, we provide definitions used in this paper. In Section 3, we give examples on how colluding users may bypass security policies using delegation, and then we provide a formal definition of security with respect to delegation. After that, we study enforcement mechanisms for delegation

security in Section 4 and design a secure workflow system in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Definitions

In this paper, we focus on role-based access control systems. Delegation models could be complicated. To create a delegation model, one needs to decide on a number of features, such as whether to allow partial delegation (i.e. delegating a portion of the permissions of a role), whether users can further delegate the privileges they received, how revocations are performed and so on. In order to describe the problem in a precise manner, we focus on a specific model rather than considering all possible options. However, our ideas and arguments will apply to delegation models with different features from ours.

In this section, we formalize delegation operations as access control state transition operations. We provide precise definitions on access control states, state transition rules and access control systems.

States (γ)**:** We assume that there are three countable sets: $\mathcal{U}$ (the set of all possible users), $\mathcal{R}$ (the set of all possible roles), $\mathcal{P}$ (the set of all possible permissions).

Definition 1 (Access Control State). An access control state γ is given as a 4-tuple $\langle UR, PA, DR, B\rangle$, where $UR \subseteq \mathcal{U} \times \mathcal{R}$ is user-role membership, $PA \subseteq \mathcal{P} \times \mathcal{R}$ is permission-role assignment, $DR \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{R} \times \{\text{“}g\text{”}, \text{“}t\text{”}\}$ is delegation relation, and B is a set of binary relations between users.

The user-role membership UR should not be confused with the user-role assignment relation UA in RBAC. When an RBAC system has both UA and a role hierarchy RH, the two relations UA and RH together determine UR. In other words, our notion of state abstracts away the details about how users gain role memberships.

In the delegation relation DR, $(u_1, u_2, r, \text{“}g\text{”})$ indicates that u_1 has delegated the role r to u_2 via a *grant* operation, while $(u_1, u_2, r, \text{“}t\text{”})$ indicates that u_1 has delegated the role r to u_2 via a *transfer* operation. The difference between grant and transfer will be discussed later in this section.

The binary relations defined in B will be useful in constraint specification in workflows. Examples on binary relations are “be a supervisor of” and “have conflicts of interests”.

Given a state γ, each user has a set of roles for which the user is authorized. A user is authorized for a role r if and only if he/she is a member of r or he/she received r from another user through delegation. We formalize this by defining a function $authR : \mathcal{U} \times \Gamma \rightarrow 2^{\mathcal{R}}$, where Γ is the set of all states.

$$authR(u, \langle UR, PA, DR, B\rangle) = \{r \mid (u, r) \in UR \vee \exists_{u'}((u', u, r, \text{“}g\text{”}) \in DR \vee (u', u, r, \text{“}t\text{”}) \in DR)\}$$

When a user u is authorized for the role r, he/she is authorized for the permissions assigned to r.

Delegation and State Transition: First of all, we introduce the notations related to delegation. Assume that $Alice$ delegates the role `Accountant` to Bob. In such an operation, $Alice$, who is the granter of privilege, is called *delegator*; Bob, who is the receiver of privilege, is called *delegatee*; the role `Accountant` is the *delegated privilege*.

We assume that each delegation operation has only one delegated privilege. If a user wants to delegate multiple privileges to the same receiver, he/she can perform multiple delegation operations.

A delegation operation is essentially an access control state transition operation, which takes one of the following three forms:

- $grant(u_1, u_2, r)$: user u_1 *grants* role r to user u_2. After the delegation operation, u_2 gains r and u_1 still keeps r.
- $trans(u_1, u_2, r)$: user u_1 *transfers* role r to user u_2. After the delegation operation, u_2 gains r and u_1 (temporarily) loses r.
- $revoke(u_1, u_2, r)$: user u_1 *revokes* the delegated privilege, role r, from u_2.

Note that a user can grant or transfer only the roles he/she is a member of to others. To simplify delegation relation, we assume that a delegatee cannot further delegate the delegated privilege to other users, and only the corresponding delegator can revoke the delegated privilege from the delegatee.

Since delegation is performed in a distributed manner, in the sense that everyone may perform delegation operations, it is undesirable to allow a user to delegate his/her roles in a completely unrestricted way. Delegation operations are thus subject to the control of authorization rules, which takes one of the following three forms:

- $can\_grant(cond, r)$: a user who satisfies condition $cond$ can grant r to other users, where $cond$ is an expression formed using roles, the binary operators $\wedge$ and $\vee$, the unary operator $\neg$, and parentheses.
- $can\_transfer(cond, r)$: a user who satisfies condition $cond$ can transfer r to other users.
- $can\_receive(cond, r)$: a user who satisfies condition $cond$ can receive r from other users.

  For example, the rule $can\_receive(\texttt{Clerk} \wedge \neg \texttt{Treasurer}, \texttt{Accountant})$ states that anyone who is a member of `Clerk` but not a member of `Treasurer` can receive the role `Accountant`.

Definition 2 (Administrative State). An *administrative state* consists of a set RL of authorization rules. Given RL, a delegation operation $grant(u_1, u_2, r)$ (or similarly, $trans(u_1, u_2, r)$) succeeds in the state $\langle UR, PA, DR, B\rangle$ if and only if

$$(u_1, r) \in UR \wedge can\_grant(c_1, r) \in RL \wedge (u_1 \text{ satisfies } c_1)$$
$$\wedge\ can\_receive(c_2, r) \in RL \wedge (u_2 \text{ satisfies } c_2)$$

Otherwise, the delegation operation fails.

To simplify management, we assume that if a user u_1 granted or transferred a role r to u_2 and has not revoked r from u_2 yet, then u_1 can neither grant nor transfer r to u_2 again. That is to say, at any moment, a user may receive a role from the same user at most once. But a user may receive the same role from different users.

We use $\gamma \rightarrow_{op}^{RL} \gamma'$ to denote the state transition from γ to γ' after applying the delegation operation op under administrative state RL. Let $\gamma = \langle UR, PA, DR, B\rangle$. The state transition rules are described as follows:

- $op = grant(u_1, u_2, r)$: If op fails, then $\gamma' = \gamma$. Otherwise, $\gamma' = \langle UR, PA, DR', B\rangle$, where $DR' = DR \cup \{(u_1, u_2, r, \text{“}g\text{”})\}$.

– If $op = trans(u_1, u_2, r)$: If op fails, then $\gamma' = \gamma$. Otherwise, $\gamma' = \langle UR', PA, DR', B\rangle$, where $UR' = UR/\{u_1, r\}$ and $DR' = DR \cup \{(u_1, u_2, r, \text{“}t\text{”})\}$.
– If $op = revoke(u_1, u_2, r)$: There are three cases. Let $\gamma' = \langle UR', PA, DR', B\rangle$.
  • If $(u_1, u_2, r, \text{“}g\text{”}) \in DR$, then $UR' = UR$ and $DR' = DR/\{(u_1, u_2, r, \text{“}g\text{”})\}$.
  • If $(u_1, u_2, r, \text{“}t\text{”}) \in DR$, then $UR' = UR \cup \{(u_1, r)\}$ and $DR' = DR/\{(u_1, u_2, r, \text{“}t\text{”})\}$.
  • Otherwise, $\gamma' = \gamma$. It indicates that u_2 did not receive r from u_1 in γ, and thus the revocation fails.

Note that PA and B are not affected by state transition rules.

With the above state transition rules, we may apply a sequence Q of delegation operations one by one to γ and acquire γ'. We say that γ' is *reachable* from γ under administrative state RL, which is denoted as $\gamma \rightsquigarrow_Q^{RL} \gamma'$.

Workflow and Access Control Systems: In this paper, a task is modeled as a workflow, which divides the task into a number of well-defined steps.

Definition 3 (Workflow and Constraints). A *workflow* is represented as a tuple $\langle S, \prec, C\rangle$, where S is a set of steps, $\prec \subseteq S \times S$ defines a partial order among steps in S, and C is a set of constraints. $s_1 \prec s_2$ indicates that s_1 must be performed before s_2.

A *constraint* takes the form of $ct\langle s_1, s_2, \rho\rangle$, where s_1 and s_2 are two steps and ρ is a binary relation between users. Let u_1 and u_2 be the users who perform s_1 and s_2, respectively. $ct\langle s_1, s_2, \rho\rangle$ is satisfied if and only if $(u_1, u_2) \in \rho$.

Binary relations between users play an important role in constraint specification in existing workflow models [4, 5, 10, 12]. Equality (=) and inequality ($\neq$) relations are most common ones, and they are supported by almost all existing models. Besides “=” and “$\neq$”, user-defined binary relations, such as “have conflicts of interests”, are supported by the workflow defined in Definition 3.

We call $c = ct\langle s_1, s_2, \rho\rangle$ a constraint on s_1 and s_2. If s_1 is executed later than s_2, then c is checked upon the execution of s_1; otherwise, c is checked upon the execution of s_2.

In an access control state, the permissions to perform steps in workflows are assigned to roles. Given an access control state $\gamma = \langle UR, PA, DR, B\rangle$, we say that a user u is *authorized* to perform a step s (or u is an *authorized user* for s), if and only if there exists a role r such that $r \in authR(u, \gamma)$ and $(p_s, r) \in PA$, where p_s is the permission to perform s.

When a task is performed, an instance of the corresponding workflow is created. In order to complete the workflow instance, every step of the workflow instance must be assigned to an authorized user and such assignments must not violate any constraint specified in the workflow. Note that during the execution of the workflow instance, the access control state may change due to delegation. We only need to ensure that a user is authorized to perform a step at the moment the step is performed. Constraint evaluation, which depends on user relations, is not affected by state changes, because the set B of user relations will not be modified by delegation operations.

An access control system with delegation support is defined in below.

Definition 4 (Access Control System). An *access control system* is represented as a 3-tuple $\langle \gamma, W, RL\rangle$, where γ is the initial access control state, W is a set of workflows and RL is the administrative state.

We assume that in the initial state $\gamma = \langle UR, PA, DR, B\rangle$ of an access control system, we always have $DR = \emptyset$. That is to say, no delegation operations have been performed in the initial state.

3 The Security of Delegation

We have provided precise definitions related to delegation and access control systems. In this section, we study the impact of delegation on the security of access control systems. First, we give examples on delegation-based attacks on access control systems. Second, we formally define the notion of security with respect to delegation in access control systems.

3.1 Circumventing Security Policies Using Delegation

In this section, we consider how malicious users may collude to circumvent security policies in access control systems. We present two examples describing two scenarios, in which colluding users successfully complete those tasks that they would not be able to complete without the "help" of delegation. After each example, we summarize the characteristic of the attack in the scenario.

Example 1. In an institution, a sensitive task t must be completed by a *single* user who is a member of both roles r_1 and r_2. Task t is modeled as workflow $w_1 = \langle S, \prec, C\rangle$, where $S = \{s_1, s_2\}, s_1 \prec s_2$ and $C = \{ct\langle s_1, s_2, =\rangle\}$. Permissions to perform s_1 and s_2 are assigned to r_1 and r_2, respectively. The constraint in C requires that the two steps must be performed by the same user, which enforces that an instance of w_1 can be completed only by a user who is a member of both r_1 and r_2.

$Alice$ and Bob are employees of the institution. $Alice$ is a member of r_1 but not r_2, while Bob is a member of r_2 but not r_1. Clearly, neither $Alice$ nor Bob is qualified to complete an instance of w_1. However, if $Alice$ delegates (either by grant or transfer) r_1 to Bob, then Bob is authorized to perform both s_1 and s_2 and he is thus able to complete an instance of w_1. In other words, if $Alice$ and Bob collude, they can complete a task which they should not be able to complete.

In Example 1, $Alice$ "lends" her role membership of r_1 to Bob to make him more "powerful" than before. The example demonstrates that, using delegation, a group of colluding users may create a "more powerful" user by aggregating role memberships of different individuals in the group. In that case, security policies that require a single user (rather than multiple users) with multiple role memberships to complete a task could be circumvented.

Example 2. In a company, the task of issuing checks is modeled as a workflow consisting of two steps s_{pre} and s_{app}, which stand for "check preparation" and "approval", respectively. In order to prevent fraudulent transactions, s_{pre} and s_{app} must be performed by two *different* members of the role `Treasurer` (or two `Treasurers` for short). The workflow can be represented as $w_2 = \langle S, \prec, C\rangle$, where $S = \{s_{pre}, s_{app}\}, s_{pre} \prec s_{app}$ and $C = \{ct\langle s_{pre}, s_{app}, \neq\rangle\}$. Also, for the sake of resiliency, the company allows a `Treasurer` to transfer his/her role to a `Clerk` in case he/she is not able to work due to sickness or some other reasons. In other words, $can\_transfer(\texttt{Treasurer}, \texttt{Treasurer}) \in RL$ and $can\_receive(\texttt{Clerk}, \texttt{Treasurer}) \in RL$.

$Alice$ and Bob are employees of the company and they decided to collude to issue checks for themselves. $Alice$ is a `Treasurer`, while Bob is a `Clerk` and is thus not qualified to perform any step in w_2. To achieve the goal, $Alice$ and Bob do the followings:

1. $Alice$ performs $trans(Alice, Bob, \texttt{Treasurer})$, which makes Bob a member of the role `Treasurer`.
2. Bob performs s_{pre} to prepare a check for $Alice$.
3. $Alice$ performs $revoke(Alice, Bob, \texttt{Treasurer})$ to revoke `Treasurer` from Bob and regains the role.
4. $Alice$ performs s_{app} to approve the check prepared by Bob.
   What the workflow system sees is that s_{pre} and s_{app} are performed by two different users. Thus, the constraint $ct\langle s_{pre}, s_{app}, \neq\rangle$ is satisfied and the operation succeeds.

After all of the above being done, a check is issued and $Alice$ and Bob may share the money.

In Example 2, $Alice$'s role membership of `Treasurer` is used twice by two different users in the same workflow instance. This example demonstrates that colluding users can make "copies" of their access privileges using delegation to bypass security constraints that enforce separation of duty.

3.2 Formal Definition of Security

We have seen examples on how colluding users may circumvent security policies in access control systems with the help of delegation. It is clear that if an access control system allows colluding users to bypass security policies, then the system is insecure. But, how can we tell whether a security policy has been circumvented by delegation operations? What should a "secure" system look like? We answer these fundamental questions by formally defining the notion of security with respect to delegation.

First of all, we present a general definition of security, which is independent of the concrete design of access control systems. Given an access control system, we define the predicate $can\_complete$, such that $can\_complete(t, U_1, U_2, \gamma)$ is "true" if and only if users in U_1 together can complete task t when the initial access control state is γ and only users in U_2 can perform delegation operations. The concrete definition of $can\_complete$ depends on how tasks are modeled and the concrete design of access control systems. We say that a group of users becomes more powerful (or gain power enhancement) when they eventually complete a task that they are not able to complete in the initial state (delegation is needed to change the state in this case). Intuitively, if an access control system is secure with respect to delegation, then a group of users cannot enhance the power of the group by performing delegation operations within the group. The following definition formally states such an intuition.

Definition 5 (Security). An access control system with initial access control state γ is *secure with respect to delegation* if and only if the following is true:

$$\forall_{t\in \mathrm{T}} \forall_{U\subseteq \mathcal{U}}\ can\_complete(t, U, U, \gamma) \Rightarrow can\_complete(t, U, \emptyset, \gamma)$$

where T is the set of all tasks and $\mathcal{U}$ is the set of all users in the system.

In the above definition, $can\_complete(t, U, U, \gamma)$ is "true" if and only if users in U together can complete t when the initial state is γ and delegation is available in such a way: the users may perform delegation operations to change the access control state,

Round 0:

The adversary selects a workflow $w \in W$ and a set U of users, such that U cannot complete w in γ without delegation. If such a combination of w and U does not exist, then the adversary loses (in this case, the system is trivially secure as everyone is able to complete every task). $PP \leftarrow \emptyset$ and $SS \leftarrow S$, where PP records past user-step assignments and SS records the remaining steps. $i \leftarrow 1$ and $\gamma_0 \leftarrow \gamma$.

Round i:

1. The adversary designs a sequence Q_i of delegation operations such that every delegation operation in Q_i involves only users in U [1]. The adversary applies Q_i to γ_{i-1} and acquires a new state γ_i.
2. The adversary selects a step s from SS such that $\forall_{s' \in S}(s' \prec s \Rightarrow s' \notin SS)$. The adversary selects a user u from U as well.
   If u is not authorized for s in γ_i, then the adversary loses.
   Otherwise, $PP \leftarrow PP \cup \{(u, s)\}$ and $SS \leftarrow SS/\{s\}$.
3. If $SS = \emptyset$, then
   If no constraint in C is violated by PP, then the adversary wins;
   Otherwise, the adversary loses.
   Otherwise, $i \leftarrow i + 1$ and the game continues to the next round.

Fig. 1. Description of the game in Definition 6

but no user outside of U is allowed to perform delegation operations. That is to say, users in U cannot get "help" from outsiders. In contrast, $can\_complete(t, U, \emptyset, \gamma)$ is "true" if and only if users in U together can complete t in state γ and no delegation operation is allowed. In general, Definition 5 essentially states that, in a secure access control system, if a set of users can complete a task without receiving any privilege from outsiders, then they must be able to compete the task without delegation at all. That is to say, delegation does not enable a set of users to enhance their own power by themselves.

The notion of security introduced in Definition 5 respects the definition of delegation. Delegation is defined as a mechanism that allows a user A to act on another user B's behalf by making B's access rights available to A. Let γ and γ' be the states before and after a delegation operation from B to A, respectively. The fact that A is working on B's behalf in γ' indicates that A should not be able to do more than A and B together (i.e. $\{A, B\}$) can do in γ. Furthermore, since B does not gain anything by delegating his/her privileges to A, $\{A, B\}$ in γ' cannot be more powerful than $\{A, B\}$ in γ. By generalizing such an argument to groups with arbitrary number of users, we acquire the notion of security in Definition 5.

We now illustrate the effect of delegation in a secure access control system by giving an example. Assume that $Alice$ grants (or transfers) a role r to Bob. Then, Bob may become more powerful by acquiring r. Furthermore, every group G such that $Bob \in G$ and $Alice \notin G$ may become more powerful as well, because one of its member (Bob) received a privilege from an outsider ($Alice$). However, every group G' such that $Alice, Bob \in G'$ should not gain power enhancement. Otherwise, G' enhances its own power after a delegation operation between its members and the access control system is insecure by Definition 5. In general, in a secure access control system, a group of users may gain power enhancement only if they receive privileges from outsiders.

Definition 5 is general and independent of concrete access control systems. In this paper, tasks are modeled as workflows. Using the definitions in Section 2, we provide a more concrete definition of security in below.

Definition 6 (Secure Workflow System). An access control system $\langle \gamma, W, RL \rangle$ is *secure with respect to delegation* if and only if an adversary can never win the one-person game described in Figure 1.

Note that in the above game, the effect of delegation operations is subject to RL. The adversary can perform a sequence of delegation operations to change the access control state at the beginning of each round. The game allows delegation operations between the execution of two steps (i.e. between two rounds) so that users can perform revocation to regain the roles that were transferred to other users in previous rounds. This gives the adversary more advantages than allowing the adversary to perform delegation operations only at the beginning of the game. In Example 2, delegation operations are performed between the execution of two steps.

The adversary winning the game indicates that there exist a group of users that can enhance themselves with the help of delegation. In that case, the access control system is vulnerable to collusion and is thus insecure with respect to delegation.

4 Enforcing the Security of Delegation

We have defined the formal notion of security with respect to delegation. A natural next step is to study mechanisms to enforce security. In this section, we study two approaches, static enforcement and dynamic enforcement. In static enforcement, security is ensured by careful design of administrative state. In dynamic enforcement, a verification procedure is performed by the end of the execution of each workflow instance to ensure that the participants have not enhanced their own power through delegation. In Section 5, we propose a third approach, the source-based enforcement mechanism, which employs a novel security policy evaluation method that is customized for delegation.

4.1 Static Enforcement

Given a set of workflows and an initial access control state, a straightforward approach to enforce security is to carefully design the administrative state RL so that no "dangerous" delegation operation would succeed. For instance, in Example 1, if RL does not allow members of r_2 to receive r_1 and vice versa, the collusion between $Alice$ and Bob could not succeed. Such an enforcement mechanism is called *static enforcement*, as the security of the system relies on (administrative) state configuration and can be verified in an off-line manner. An access control system that enforces security via a static enforcement mechanism is called a *statically secure system*.

The advantage of static enforcement is that, if we have already implemented an access control system with delegation support, we just need to modify the administrative state to enforce security. There is no need to change the existing implementation. However, static enforcement could make the administrative state more restrictive than necessary. For instance, assume that there are two workflows w_1 and w_2 in the system. $Alice$ and Bob are two users who are not supposed to complete w_1. But the system setting is such that if $Alice$ can successfully grant or transfer role r to Bob, then $Alice$ and Bob together can complete w_1. In order to prevent the potential collusion between $Alice$ and Bob, the administrative state must prevent $Alice$ from delegating r to Bob. But this is too restrictive as Bob may only intend to perform w_2 (instead of w_1) after receiving r, which could be allowed. But static enforcement mechanism does not take the actual usage of delegated privileges into account. Finally, the design of the administrative state is usually subject to administrative policies as well as practical considerations. It may be undesirable to dramatically alter the administrative state due to security concerns, for security should not significantly affect the usability of the system.

Let γ be the initial state of the access control system. For every workflow instance, the system does the followings. Let X be an instance of workflow w.

- When X is created: $U_X \leftarrow \emptyset$
- When a step s is performed by a user u: Let p_s be the permission to perform s and $\gamma' = \langle UR, PA, DR, B \rangle$ be the current state.
  - If there exists a role r such that $((u, r) \in UR \wedge (p_s, r) \in PA)$, then $U_X \leftarrow U_X \cup \{u\}$. This indicates that u can use his/her own privilege to perform the step.
  - Otherwise, u specifies a user u' such that $((u', u, r) \in DR \wedge (p_s, r) \in PA)$. $U_X \leftarrow U_X \cup \{u, u'\}$. This indicates that u is using a delegated privilege r received from u' to perform the step. When the choice of u' and r is unique, the system may do the selection itself rather than asking the user to specify the choice.
- After X is finished: The system solves $\mathsf{wsp}(U_X, w, \gamma)$. If the answer to $\mathsf{wsp}(U_X, w, \gamma)$ is "yes", then the result of X is confirmed; otherwise, the result of X is voided and necessary roll-back is performed.

Fig. 2. Description of dynamic enforcement

4.2 Dynamic Enforcement

Static enforcement is too restrictive as it does not take into account how delegatees use the delegated privileges. This motivates the proposal of dynamic enforcement for delegation security.

To begin with, we describe the high-level idea of dynamic enforcement. In dynamic enforcement, the initial state γ of the access control system is recorded. For every workflow instance X, the system maintains a list U_X of the participants for the instance. Every user who executed a step of X is added to U_X. When a user u requests to execute a step s, the system checks whether he/she needs to use a delegated privilege. If a delegated privilege r should be used by u to perform s, then both u and the delegator of the privilege are added to U_X. Note that if u has received r from multiple delegators, u has to specify the delegator of r for the execution of s. At the end of the instance, the system checks whether the users in U_X can complete the workflow in γ without delegation. If they can, then the execution of X is confirmed. Otherwise, the system gives warning that users in U_X have enhanced their own power through delegation. The execution of X is rejected.

The problem of checking whether a set of users can complete a workflow in an access control state without delegation is called the *Workflow Satisfaction Problem (*WSP*)*.

Definition 7 (Workflow Satisfaction Problem). Given a set U of users, a workflow $w = \langle S, \prec, C \rangle$ and an access control state γ, the *Workflow Satisfaction Problem (*WSP*)* asks whether we can assign a user $u \in U$ to every step $s \in S$ such that u is authorized for s in γ and no constraint in C is violated by the overall assignments. An instance of WSP is denoted as $\mathsf{wsp}(U, w, \gamma)$.

Detailed description of dynamic enforcement is given in Figure 2. Dynamic enforcement ensures that a workflow instance may be successfully completed only if the participants (including those users who perform a step and those delegators who contribute necessary privileges through delegation operations) can complete the same workflow instance in the initial state. Hence, the correctness of dynamic enforcement follows directly from Definition 5.

Dynamic enforcement monitors the usage of delegated privileges rather than placing restrictions on administrative states. It is thus less restrictive and more practical than static enforcement. However, dynamic enforcement introduces a performance overhead as the system needs to solve a WSP instance by the end of every workflow instance. It has been proved in [12] that WSP is **NP**-complete, which indicates that the runtime

overhead of dynamic enforcement for each workflow instance could be exponential in the size of the workflow.

In real-world, the number of steps in a workflow is normally small. Hence, it is possible that the performance of dynamic enforcement is acceptable in practice. Also, dynamic enforcement does not require changing existing implementation of workflow modules. All we need to do is to add a module to the system to perform recording and the closing verification procedure for workflow instances.

5 A Secure Workflow System

We have discussed two mechanisms to enforce delegation security in access control systems. Even though both approaches have the advantage of allowing the reuse of existing workflow implementation, they have major drawbacks: static enforcement is too restrictive and dynamic enforcement may introduce large performance overhead. A natural question is, if we are willing to redo the workflow module, can we have a better mechanism to enforce delegation security?

In this section, we propose the source-based enforcement mechanism, which employs a novel method to evaluate constraints in workflow systems. We describe the idea of source-based enforcement mechanism by presenting a design of a secure workflow system. Our workflow system is secure with respect to Definition 6 and introduces almost no performance overhead.

The high-level idea of source-based enforcement is that, when a user $Alice$ requests to perform a step s of a workflow instance, he/she must specify the privilege to be used and the source of the privilege. For instance, assume that $Alice$ requests to perform a step s with role r. If $Alice$ is a member of r, then $Alice$ may specify herself as the source of r. If $Alice$ received r from others, then $Alice$ may pick a delegator of r and specify the delegator as the source. Note that, even if $Alice$ is a member of r herself, she may still specify another user as the source of r as long as she has received r from that user.

Given the privilege r and its source u_o specified by $Alice$, the system checks the constraints on s as if it is u_o rather than $Alice$ who is performing s. For example, assume that workflow w consists of two steps s_1 and s_2, both of which can be performed by members of role `Accountant`. There is a constraint in w, which states that the users who perform s_1 and s_2 must not have conflicts of interests. Assume that $Alice$ has executed s_1 using her own membership of `Accountant`. Now, $Carl$ tries to use the delegated privilege `Accountant` received from Bob to perform s_2. Instead of checking conflicts of interests between $Carl$ and $Alice$ as what traditional workflow systems do, our system checks conflicts of interests between Bob and $Alice$. The intuition is that, since $Carl$ is using a delegated privilege from Bob, he is working on Bob's behalf. Hence, Bob and $Alice$ must not have conflicts of interests. By evaluating constraints in this way, we can ensure that the system is secure with respect to delegation.

Sometimes, in addition to sources of privileges, we want to take the actual performers into account while evaluating constraints. To achieve this, our system supports two types of constraints. Type-1 constraint only ensures that the sources of privileges satisfy the constraint; Type-2 constraint is more restrictive: if either the actual performer or the source violates the constraint, then the constraint is violated. For instance, if the

constraint in the example in the previous paragraph is a Type-2 constraint, then $Alice$ must not have conflicts of interests with either Bob (source) or $Carl$ (actual performer).

Next, we describe the design of a secure workflow system, which employs the source-based enforcement mechanism.

System Description: The system adopts the representations of access control state and the state transition rules introduced in Section 2. The only major change in this system is the way workflow constraints are evaluated.

A workflow is represented as $\langle S, \prec, C\rangle$, where S is a set of steps, $\prec \subseteq S \times S$ defines a partial order among steps in S, and C is a set of constraints. $s_1 \prec s_2$ indicates that s_1 must be performed before s_2.

A *constraint* takes the form of $ct\langle s_1, s_2, \rho, i\rangle$ where s_1 and s_2 are two steps, ρ is a binary relation between users and $i = 1$ or 2. When $i = 1$, the constraint is of *Type-1*, while when $i = 2$, the constraint is of *Type-2*.

Let $w = \langle S, \prec, C\rangle$. $\gamma = \langle UR, PA, DR, B\rangle$ is the current access control state. When a user u requests to perform a step s of an instance X of w, u presents a pair $\langle u_o, r\rangle$, where u_o is a user identity and r is a role. u_o is called the *source* of r. The pair $\langle u_o, r\rangle$ is valid if and only if one of the followings is true:

- $u = u_o \wedge (u, r) \in UR$. In other words, u is using his own role membership to perform s.
- $u \neq u_o \wedge ((u_o, u, r, \text{“}g\text{”}) \in DR \vee (u_o, u, r, \text{“}t\text{”}) \in DR)$. That is to say, u_o has granted or transferred r to u and u requests to perform s on u_o's behalf.

With the pair $\langle u_o, r\rangle$, u can successfully execute s if and only if both of the followings hold:

1. u is authorized to perform s with role r. That is, $(p_s, r) \in PA$, where p_s is the permission to perform s.
2. No constraint is violated. That is, for every constraint c on s:
   - Case $c = ct\langle s, s', \rho, 1\rangle$: $(u_o, u'_o) \in \rho$, where u'_o is the source of the privilege used to perform s'. The case $c = ct\langle s', s, \rho, 1\rangle$ is similar.
   - Case $c = ct\langle s, s', \rho, 2\rangle$: $(u, u') \in \rho \wedge (u_o, u') \in \rho \wedge (u, u'_o) \in \rho \wedge (u_o, u'_o) \in \rho$ where u' is the user who actually performed s' and u'_o is the source of the privilege used to perform s'. The case $c = ct\langle s', s, \rho, 2\rangle$ is similar.

   Note that in the first case, c is a Type-1 constraint and only the sources must satisfy the constraint. In the latter case, c is a Type-2 constraint, and both the sources and the actual performers are taken into account.

After a step is executed, the system records the identities of both the actual performer and the source of privilege for future reference.

The following example illustrates how the system works.

Example 3. In a bank, task t is modeled as a workflow $w = \langle S, \prec, C\rangle$, where $S = \{s_1, s_2\}, s_1 \prec s_2$ and $C = \{ct\langle s_1, s_2, \neq, 1\rangle\}$. The permissions to perform s_1 and s_2 are assigned to r_1 and r_2, respectively. $Alice$ is a member of r_1 and Bob is a member of r_2.

$Alice$ becomes too busy to work on t and would like to balance the workload with Bob by delegating r_1 to Bob. Let X be an instance of w. Bob performs s_1 in X by presenting $\langle Alice, r_1\rangle$ to the system. The system records that Bob is the actual performer of s_1 in X and $Alice$ is the source of privilege. Next, Bob requests to perform s_2 in X

by presenting $\langle Bob, r_2 \rangle$, which indicates that himself is the source of r_2. The system found that the constraint $ct\langle s_1, s_2, \neq, 1 \rangle$ needs to be checked. Since the constraint is of Type-1, the system only considers the sources of privilege for s_1 and s_2, which are $Alice$ and Bob respectively. Because $Alice \neq Bob$, the constraint is satisfied, and Bob completes X. Note that this does not violate the notion of security, because $Alice$ is involved in X by allowing Bob to work on her behalf, and $Alice$ and Bob together can complete w before the delegation operation.

Now, assume that the constraint in C is of Type-2 (i.e. $ct\langle s_1, s_2, \neq, 2 \rangle$). In this case, Bob cannot complete w. When $ct\langle s_1, s_2, \neq, 2 \rangle$ is checked, the system takes both the actual performers and the sources into account. When the system compares the actual performer of s_1 with the source of privilege (or the actual performer) of s_2, it has $Bob = Bob$, which indicates that $Bob \neq Bob$ does not hold. Hence, the constraint is violated and Bob is rejected from performing s_2.

It is clear that Type-2 constraints provide stronger security than Type-1 constraints. People may wonder why we support the seemingly less secure Type-1 constraints in our system. First of all, as we will prove later in this section, Type-1 constraints are sufficient to enforce the notion of security defined in Definition 6. Secondly, in certain situations, we may gain flexibility by using Type-1 constraints. For instance, a workflow may have a constraint c stating that s_1 and s_2 must be performed by the same user. Assume that $Alice$ has performed s_1 in an instance X of the workflow but she has to leave before performing s_2. If c is a Type-1 constraint (i.e. $c = ct\langle s_1, s_2, =, 1 \rangle$), then $Alice$ may delegate her privilege r to another user Bob who may complete s_2 in X by presenting the pair $\langle Alice, r \rangle$ to the system; but if c is a Type-2 constraint, then s_2 of X cannot be completed until $Alice$ comes back. In situations where it is more beneficial to complete the task, we should declare c as Type-1. In contrast, in situations where security is given high priority and we would rather have the task unfinished than allow another user to involve, we should declare c as Type-2. The choice between Type-1 and Type-2 constraints can be viewed as a flexibility-security trade-off. Our system provides the options and leaves the decisions to security policy designers.

Next, we prove that our workflow system is secure with respect to delegation. The general idea of the proof is that, for every workflow instance that is completed, we modify its user-step assignment by replacing the actual performer of each step with the corresponding source of privilege. Since our constraint evaluation procedure always takes sources into account, the modified user-step assignment must be valid for the workflow in the initial state of the system. This implies that the set of sources can complete the workflow in the initial state.

Theorem 1. The workflow system employing source-based enforcement mechanism is secure with respect to delegation.

Due to page limit, the proof of Theorem 1 is given in a technical report [13].

6 Related Work

Delegation has received considerable attention from the research community. In [2, 3], Barka and Sandhu proposed a framework for role-based delegation models (RBDM), which identifies a number of characteristics related to delegation. Example characteristics are monotonicity, totality, and levels of delegation.

There exist a wealth of delegation models in literature [8, 16, 15, 11, 1, 7, 6]. L. Zhang et al. [15] presented a role-based delegation model called RDM2000. Their model supports the specification of delegation authorization rules to impose restrictions on which roles can be delegated to whom. X. Zhang et al. [16] proposed a role-based delegation model called PBDM, which supports both role and permission level delegation. Their model controls delegation operations through the notion of delegatable roles such that only permissions assigned to these roles can be delegated to others. In [6], Crampton and Khambhammettu proposed a delegation model that supports both grant and transfer. Atluri and Warner [1] studied how to support delegation in workflow systems. They extended the notion of delegation to allow conditional delegation, where conditions can be based on time, workload and task attributes. One may specify rules to determine under what condition a delegation operation should be performed.

All the above work focus on the modeling and management of delegation, while our paper focuses on the security impact of delegation on access control systems. None of the above work proposes a formal notion of security regarding delegation or studies mechanisms to enforce security in access control systems with delegation support.

In [9], Shaad observed that delegation and revocation features of a system may be used to circumvent separation of duty properties. He gave an example to illustrate an attack conducted by a single user. In his example, there is a separation of duty policy which requires that no single user may first access an object o using privilege $auth_1$ and then access o again with privilege $auth_2$. The system he designed enforces such a policy by allowing a user to access o only if the user does not have both $auth_1$ and $auth_2$ at the time of access. Let $Alice$ be a malicious user having both $auth_1$ and $auth_2$. $Alice$ first transfers $auth_2$ to another user Bob so as to temporarily lose $auth_2$. Next, she accesses o with $auth_1$ and then revokes $auth_2$ from Bob to regain the privilege. Finally, $Alice$ transfers $auth_1$ to Bob and then accesses o again using $auth_2$. In this case, the separation of duty policy is circumvented. This example differs from our examples in Section 3.1 in a couple of ways:

1. The attack in [9] is conducted by a single user ($Alice$), as the delegatee (Bob) is not actively involved. In contrast, our examples are on multi-user collusion, where all principles are actively involved in the attack.
2. The attack in [9] relies on a specific way in which separation of duty is implemented. In particular, it is assumed that the system does not maintain any historical record. But this is not the case in most of the existing workflow authorization systems [4, 5, 10, 14], as these systems keep track of which users have performed which steps so as to enforce constraints. In contrast, our examples apply to workflow authorization systems in existing literature.

In general, the example in [9] has a very different nature from our examples in Section 3.1. Shaad's paper [9] is about an access control framework and the interaction between delegation and security policies is not the main focus of the paper. Problems such as collusion and enforcement mechanisms for security, which are studied in our paper, are not discussed in [9].

7 Conclusion

We have studied the impact of delegation on the security of access control systems. Collusion is a potential threat in those access control systems that support delegation.

We have formally defined the notion of security with respect to delegation. A system that is secure regarding delegation is resistent to collusion. We have also studied different mechanisms to enforce security. In particular, we have designed a workflow system that implements the source-based enforcement mechanism through a novel constraint evaluation approach. Our design is secure and introduces little performance overhead.

References

1. V. Atluri and J. Warner. Supporting conditional delegation in secure workflow management systems. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 49–58, New York, NY, USA, 2005. ACM Press.
2. E. Barka and R. Sandhu. Framework for role-based delegation models. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 168, Washington, DC, USA, 2000. IEEE Computer Society.
3. E. Barka and R. Sandhu. A role-based delegation model and some extensions, 2000.
4. E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, Feb. 1999.
5. J. Crampton. A reference monitor for workflow systems with constrained task execution. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies (SACMAT 2005)*, pages 38–47, Stockholm, Sweden, June 2005.
6. J. Crampton and H. Khambhammettu. Delegation in role-based access control. In *Proceedings of 11th European Symposium on Research in Computer Security*, 2006.
7. J. B. D. Joshi and E. Bertino. Fine-grained role-based delegation in presence of the hybrid role hierarchy. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 81–90, New York, NY, USA, 2006. ACM Press.
8. S. Na and S. Cheon. Role delegation in role-based access control. In *RBAC '00: Proceedings of the fifth ACM workshop on Role-based access control*, pages 39–44, New York, NY, USA, 2000. ACM Press.
9. A. Schaad. A framework for organisational control principles. In *PhD Thesis, University of York*, 2003.
10. K. Tan, J. Crampton, and C. Gunter. The consistency of task-based authorization constraints in workflow systems. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169, 2004.
11. J. Wainer and A. Kumar. A fine-grained, controllable, user-to-user delegation method in rbac. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 59–66, New York, NY, USA, 2005. ACM Press.
12. Q. Wang and N. Li. Satisfiability and resiliency in workflow systems. In *Proc. European Symp. on Research in Computer Security*, Sept. 2007.
13. Q. Wang and N. Li. On the security of delegation in access control systems. CERIAS Technical Report, http://www.cs.purdue.edu/homes/wangq/papers/delegation.pdf, Jul. 2008.
14. J. Warner and V. Atluri. Inter-instance authorization constraints for secure workflow management. In *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 190–199, 2006.
15. L. Zhang, G.-J. Ahn, and B.-T. Chu. A rule-based framework for role-based delegation and revocation. *ACM Trans. Inf. Syst. Secur.*, 6(3):404–441, 2003.
16. X. Zhang, S. Oh, and R. Sandhu. Pbdm: a flexible delegation model in rbac. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 149–157, New York, NY, USA, 2003. ACM Press.