Enabling Role-Based Delegation and Revocation on Security-Enhanced Linux

Gail-Joon Ahn † and Dhruv Gami
University of North Carolina at Charlotte

Abstract

An increasing number of attacks experienced in existing enterprise networks and applications have recently created a huge demand for security mechanisms of operating systems. As a consequence, Security-Enhanced Linux (SELinux) was proposed by NSA and the industries have adopted SELinux at a fast rate. More and more enterprises are planning to move their business operations to such a secure computing environment, requiring the features of delegation and revocation. In this paper we seek to address the issue of how to leverage a role-based delegation in SELinux while minimizing the modification of SELinux system modules. Our approach is to utilize the flexible policy system used in SELinux that allows for custom rules to be defined for supporting access control requirements. We also demonstrate the feasibility of our framework through a proof-of-concept implementation.

1 Introduction

An increasing number of attacks experienced in existing enterprise networks and applications created a huge demand for operating systems security mechanisms. As a consequence, SELinux has been proposed to provide flexible support for security policies, attempting to support various security requirements [1]. Using the flexibility of SELinux, it is possible to configure the system to support a wide variety of security policies such as separation policies, containment policies, integrity policies and invocation policies [2]. Also, SELinux provides a secure work environment by labelling all objects in the operating systems, and performing strict access control checks based on a flexible policy system [3].

SELinux is being adopted by the industry at a fast rate. More and more enterprises are planning to move their business operations to such a secure computing environment. There is, however, an unfortunate mismatch between the security requirements of enterprises and the functionalities offered by SELinux. The current implementations of SELinux do not support the concepts of delegation and revocation that are critical to support large-scale enterprises with dynamic collaborative environments. In this paper we seek to address the issue of how to advocate the features of delegation and revocation in SELinux while minimizing the modification of SELinux system modules. We integrate a role-based delegation framework in SELinux. Our approach is to leverage the flexible policy system used in SELinux that allows for custom rules to be defined to provide other access control requirements. The custom rules are developed to support both administration-based and self-managed delegations.

The paper is organized as follows. Section 2 briefly describes related works and background information. Section 3 overviews a role-based delegation model followed by an architecture of our solution and a proof-of-concept prototype implementation in Section 4. Section 5 concludes the paper with future directions.

2 Related Work

SELinux is built upon a robust architecture derived by seamlessly integrating the concepts of type enforcement, Flask architecture, and Linux security modules. In type enforcement [4], all objects are partitioned into equivalence classes based on the integrity properties on objects. Each equivalence class for objects is called an object type. The subject space is partitioned into equivalence classes based on their roles in the system and are called domains. The type enforcement access matrix provides a separation of the policy and enforcement mechanisms. Moreover, the relationship between a subject and its executable is tightly controlled, offering protection against execution of malicious code. The Flask architecture [5] requires some form of separation between security policies and their enforcement, enabling the enforcement of security polices to be transparent to applications. Flask architecture is achieved in SELinux by encapsulating security policy decision logic into a new kernel component. This component makes labelling, access and poly-instantiation decisions in response to policy-

† All correspondence should be addressed to: Dr. Gail-Joon Ahn, Software and Information Systems Department, College of Computing and Informatics, University of North Carolina at Charlotte, 9201 University City Blvd., Charlotte, NC 28223; email:gahn@uncc.edu. This work was supported, in part, by funds provided by National Science Foundation (NSF-IIS-0242393), Department of Energy CAREER Award (DE-FG02-03ER25565) and Department of Defense (H98230-05-1-0119).

1-4244-1521-7/07/$25.00 ©2007 IEEE

independent requests that have been placed throughout the kernel. This enables the kernel to enforce policy decisions without needing access to the details of the policy. Linux security modules (LSM) are kernel modules that mediate access to kernel objects by placing hooks in the kernel code [6]. A hook makes a call to a function that the LSM module must provide before an internal object is accessed by the kernel, transferring control from the kernel to the LSM Module. The SELinux policy is a highly flexible policy system [2] and supports role-based access control (RBAC) [7] to a certain extent. The SELinux policies are defined in plaintext and compiled into a kernel loadable binary format. If any modifications are needed at run-time, the policy has to be recompiled and re-loaded into the memory.

3 Role-based Delegation and Revocation

Ahn et al. [8] recently identified various factors that have to be considered for formulating the mechanisms for role-based delegation and revocation. In order to leverage those features in SELinux, we adopt the existing models [7, 9]. To illustrate each functional component in our approach, we use the role hierarchy example illustrated in Figure 1 and Table 1. To simplify the discussion of delegation, we assume a user cannot be delegated to a role if the user is already a member of that role. For example, project leader Deloris with role PL1 cannot be delegated to either PC1 or PLO since he has already been an implicit member of these roles.

Table 1. Role Membership

ROLES	DIR	PL1	PL2	PC1	PC2
USERS	John	Deloris	Cathy	Michael David	Mark Lewis

3.1 Role Delegation

We first define a new relation called delegation relation (DLGT). It includes sets of three elements: original user assignments UAO, delegated user assignment UAD, and constraints. The motivation behind this relation is to address the relationships among different components involved in a delegation. In a user-to-user delegation, there are four components: a delegating user, a delegating role, a delegated user, and a delegated role. For example, ($Deloris$, PL1, $Cathy$, PL1) means $Deloris$ acting in role PL1 delegates role PL1 to $Cathy$. A delegation relation is one-to-many relationship on user assignments. The delegation relation supports role hierarchies: a user who is authorized to delegate a role r can also delegate a role r' that is junior to r. For example, ($Deloris$, PL1, $Lewis$, PC1) means $Deloris$ acting in role PL1 delegates a junior role PC1 to $Lewis$. A delegation relation is one-to-many relationship on user assignments. It consists of original user delegation (ODLGT) and delegated user delegation (DDLGT). Figure 2 illustrates such components and their relations. We assume each delegation relation may have a duration constraint associated with it. If the duration is not explicitly specified, we consider the delegation as permanent unless another user revokes it. The function Duration returns the assigned duration-restriction constraint of a delegated user assignment. If there is no assigned duration, it returns a maximum value.

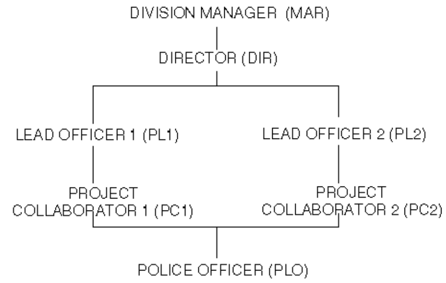

Figure 1. Role Hierarchy

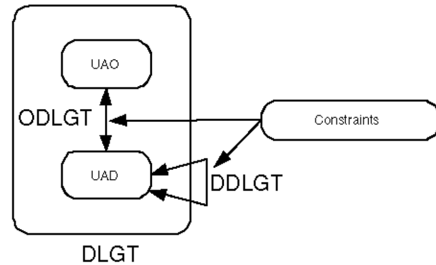

Figure 2. Delegation Relation

Our role-based delegation has the following components and theses components are formalized from the above discussions.

- T is a set of duration-restricted constraint.
- DLGT $\subseteq$ UA $\times$ UA is one to many delegation relation. A delegation relation can be represented by (u, r, u', r') $\in$ DLGT, which means the delegating user u with role r delegated role r' to user u'.
- ODLGT $\subseteq$ UAO $\times$ UAD is an original user delegation relation.
- DDLGT $\subseteq$ UAD $\times$ UAD is a delegated user delegation relation.

• DLGT = ODLGT ∪ DDLGT.

In some cases, we may need to define whether or not each delegation can be further delegated and for how many times, or up to the maximum delegation depth. We introduce two types of delegation: single-step delegation and multi-step delegation. Single-step delegation does not allow the delegated role to be further delegated; multi-step delegation allows multiple delegations until it reaches the maximum delegation depth. The maximum delegation depth is a natural number defined to impose restriction on the delegation. Single-step delegation is a special case of multi-step delegation with maximum delegation depth equal to one.

Also, we have an additional concept, delegation path (DP) that is an ordered list of user assignment relations generated through multi-step delegation. A delegation path always starts from an original user assignment. We use the following notation to represent a delegation path.

$uao_0 \rightarrow uad_1 \rightarrow uad_i \rightarrow uad_n$

Delegation paths starting with the same original user assignment can further construct a delegation tree. A delegation tree (DT) expresses the delegation paths in a hierarchical structure. Each node in the tree refers to a user assignment and each edge to a delegation relation. The layer of a user assignment in the tree is referred as the delegation depth. The function Prior maps one delegated user assignment to the delegating user assignment; function Path returns the path of a delegated user assignment; and function Depth returns the depth of the delegation path.

3.2 Role Revocation

Several different semantics are possible for user revocation. Hagstrom and others [10] categorized revocations into three dimensions in the context of owner-based approach : global and local (propagation), strong and weak (dominance), and deletion or negative (resilience). Barka and Sandhu [11] further identified user grant-dependent and grant-independent revocation (grant-dependency). We articulate user revocation in the following dimensions: grant-dependency, propagation, and dominance. Grant-dependency refers to the legitimacy of a user who can revoke a delegated role. Grant-dependent revocation means only the delegating user can revoke the delegated user from the delegated role membership. Grant-independent revocation means any original user of the delegating role can revoke the user from the delegated role. Dominance refers to the effect of a revocation on implicit/explicit role memberships of a user. A strong revocation of a user from a role requires that the user be removed not only from the explicit membership but also from the implicit memberships of the delegated role. A weak revocation only removes the user from the delegated role (explicit membership) and leaves other roles intact. Strong revocation is theoretically equivalent to a series of weak revocations. To perform strong revocation, the implied weak revocations are authorized based on revocation policies. However, a strong revocation may have no effect if any upward weak revocation in the role hierarchy fails. Propagation refers to the extent of the revocation to other delegated users. A cascading revocation directly revokes a delegated user assignment in a delegation relation and also indirectly revokes a set of subsequent propagated user assignments. A non-cascading revocation only revokes a delegated user assignment.

Suppose the revocation in Figure 3 needs a weak non-cascading approach, for $John$ to revoke $Cathy$ from role PL1, it is important to note that only $Cathy$'s membership of role PL1 is changed; other role memberships of $Cathy$ and all the delegated user assignments propagated by $Cathy$ are still valid. If the revoked node is not a leaf node, non-cascading revocation may leave a "hole" in the delegation tree. A solution might be the revoking user takes over the delegating user's responsibility. In this example, $John$ takes over the delegating user's responsibility from $Cathy$, and changes all delegation relations: ($Cathy$, PL1, u, r) ∈ DLGT to ($John$, DIR, u, r) ∈ DLGT. In this case, $John$ takes over $Cathy$'s delegating responsibility for $Mark$ and $Lewis$.

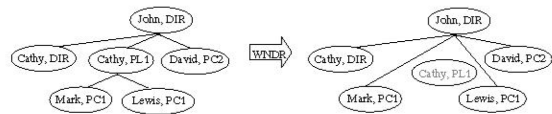

Figure 3. Weak Non-cascading Revocation

4 Enabling Role-based Delegation in SELinux

4.1 RoMan Framework

In this section, we propose a system framework called RoMan (Role Manger) to enhance SELinux's capabilities supporting the delegation functionalities outlined in Section 3. RoMan includes a set of system commands and a rule translator with a web based user interface. It takes a request from a user and converts the request to SELinux rules to grant an appropriate access. It takes care of role administration, role hierarchy, delegation & revocation, logging, and auditing. As shown in Figure 4, RoMan is a system component built upon SELinux. The shaded part in SELinux Policy represents the policy modifications conducted by RoMan. Normally RoMan generates and updates the policies to achieve the delegation rulesets.

The proposed system framework consists of four subsystems as follows [1]:

- **User Interface subsystem:** is a web-based user interface, which provides an easy way to take user input for managing a role based environment. This subsystem provides validation of data input by the user before handing it over to the Information Repository subsystem.
- **System Commands subsystem:** is a set of system level programs written to interact with the Information Repository. These commands include `addrole`, `adduser`, `getroles`, `getjuniors`, `delegate` and `revoke`. By passing proper parameters to these commands, the information repository is updated accordingly.
- **Information Repository subsystem:** is a repository specifically to maintain information that is needed to manage roles, users, delegations and revocations. Current SELinux does not provide a mechanism to store and utilize such information.
- **Translation subsystem:** is a subsystem that forms the bridge between the above-mentioned subsystems and the existing SELinux subsystems. The translator picks up information from the repository and generates a SELinux policy needed for the expected functionality.

The interactions among the subsystems are shown in Figure 5. The user (admin or system user) specifies delegation, revocation or role management requests. System commands are invoked with this information (1-2). The user interface is a simple web application that provides basic validation on data input. The user then invokes the translator to convert information into SELinux policy (3-4). The translation subsystem follows a systematic translation of the crude information stored in the repository into a set of SELinux rules that together perform the expected task. After the translation is complete, the policy is recompiled and reloaded into the kernel, thereby effectively enforcing the policy in the system instantaneously (5-6). Our implementation of RoMan provides a mechanism for a user to delegate or revoke roles (access level determined by a simple password based authentication). The back-end module is a set of system commands that update the information repository. Once the repository is updated, the translator is invoked to build necessary rules.

4.2 Implementation Details

There are six system commands that we developed in RoMan as shown in Table 2. These commands interacts with

[1] In addition, we need two existing subsystems in SELinux such as policy subsystem and enforcement subsystem.

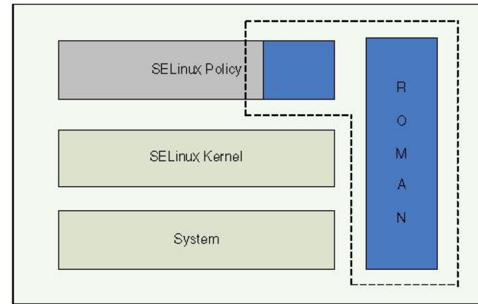

Figure 4. Role of RoMan

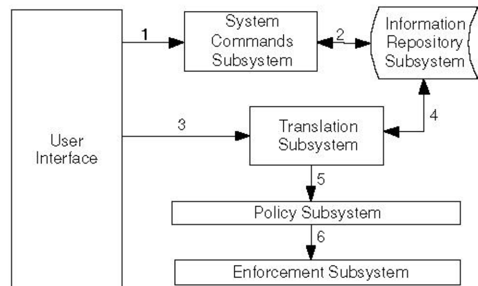

Figure 5. System Flow

six system files in the information repository as illustrated in Table 3. The `/etc/roles` and `/etc/users` files are used to maintain a list of roles and users in the system, respectively. The roles and users contained herein are the ones added via the RoMan interface and do not include the built-in system roles and users in SELinux. The `/etc/rolehr` file contains information about the hierarchical relationship of roles. By specifying the immediate seniors and immediate juniors (comma separated lists), a hierarchy tree can be constructed for each role. This information is a core component of RoMan because role hierarchy is needed for proper permission inheritance and delegation of roles [2]. Using `/etc/rolehr`, we can find all seniors and juniors for a role by respectively chasing the parents and children. For example, for the DIR role of Figure 1 we can construct the seniors and juniors list as follows.

Role	Seniors	Juniors
DIR	MAR	PL1, PL2

We say a user is an explicit member of a role if the user is explicitly designated–or through a delegation–as a member of the role. A user is an implicit member of a role if the user is an explicit member of some senior role. A

[2] Permissions of a junior role are inherited to its senior roles. Also, a role cannot be delegated to a user whose original role is senior to the delegating role.

Table 2. System Commands

Command	Purpose
addrole	Create a new role to the system `usage:  addrole <rolename> <seniors> <juniors>`
getroles	List roles currently present in the system `usage:  getroles //no parameters needed`
adduser	Create a new user to the system and assign an original role to the user `usage:  adduser <username> <password> <original_role>`
getjuniors	List all juniors to a specific role from the role hierarchy `usage:  getjuniors <rolename>`
delegate	Delegate a role from a user being a member of a specific role to another user `usage:  delegate <dlg'ing user> <dlg'ing role> <dlg'ed user> <dlg'ed role>`
revoke	Revoke an existing delegated role from a delegated user `usage:  revoke <dlg'ing user> <dlg'ing role> <dlg'ed user> <dlg'ed role>`

Table 3. Information Repository

Object	File Name	Purpose
ROLES	`/etc/roles`	Contain a list of roles in the system
HIERARCHY	`/etc/rolehr`	Maintain the hierarchical structure of roles
USERS	`/etc/users`	Contain a list of users
EXPLICIT	`/etc/explicit`	Contain a list of roles and users assigned explicitly to those roles
DELEGATIONS	`/etc/delegations`	Maintain all delegation relations
LOG	`/var/log/roman_log`	Contain a log of all actions, delegations, and revocations

user can simultaneously be an explicit and implicit member of the same role. For example, Alice can be an explicit member of PLO and PL1, in which case she is also an implicit member of PLO (by virtue of membership in PL1). To simulate this, we maintain information about explicit membership in `/etc/explicit` including original roles and delegated roles that the user is member of. Similarly, the `/etc/delegations` maintains all delegation activities based on the delegation relation. For instance, the delegation examples addressed in Section 3.1 would look like:

dlg'ing_user	dlg'ing_role	dlg'ed_user	dlg'ed_role
Deloris	PL1	Cathy	PL1
Deloris	PL1	Lewis	PC1

This file is used for logging and auditing purposes as well. Additional information such as duration of delegation and allowable delegation depth can be stored in this file to further extend our system.

One of important modules is the translation subsystem that picks up the current state of the information repository and translates the information therein into SELinux policy rules. Instead of comparing previous states and current states to validate changes of the policy, it was found to be more efficient if a new rule is developed for being inserted to the policy modules whenever needed. A base policy would not be changed but the role management rules then need to be inserted in the relevant places by translating each file of the repository. The translation has five steps as follows:

1. *Initialization*: The translation is initialized by getting rid of the existing policy files and creating a new workspace for the policy to be generated based on the new data in the information repository. The base policy is then loaded into the workspace and made available for rules pertaining to role based access control managed by RoMan.

2. *Creating New Roles*: In RoMan's information repository, role names are stored in ROLES. Since the base policy does not include a notion of role definition, we first need to create role definition rules into the policy. The rules are:

```
/src/policy/domains/user.te:
   full_user_role(newrole) allow
   system_r newrole_r;
   allow sysadm_r newrole_r;
```

```
/src/policy/macros/user_macros.te
   define('in_user_role','
   role newrole_r types  $1; ')
```

3. *Constructing Role Hierarchy*: Role hierarchy is stored in HIERARCHY which is an element of the information repository. The rules for defining role hierarchy in SELinux policy look like:

```
/src/policy/rbac
   dominance {role senior_r
              {role junior_r;}}
   allow senior_r junior_r;
```

For example, a role hierarchy of Figure 1 can be constructed by the following policy rules.

```
/src/policy/rbac
   dominance {role MAR_r{role DIR_r;}}
   allow MAR_r DIR_r;

   dominance {role DIR_r{role PL1_r;}}
   dominance {role DIR_r{role PL2_r;}}
   allow DIR_r PL1_r;
   allow DIR_r PL2_r;
```

4. *Assigning Users to Roles*: In the information repository, the user-role assignment is stored in EXPLICIT. The first field is a user that the roles are associated with. Separated by a colon, the next field is the original role that the user is member of. The last field contains a list of roles that are delegated to the user. The syntax for assigning roles to users in information repository is `username : original_role : delegated_roles`. From this information, the following policy is used to translate all entries in EXPLICIT.

```
src/policy/users
   user username roles {original_role};
```

5. *Performing Delegation*: DELEGATIONS object in the repository contains delegating user, delegating role, delegated user, and delegated role. Based on this format, we need to define rules to allow the delegated user to be member of a delegating user's role and to inherit its permissions. In our system, SELinux policy for this relation is:

```
user delegated_user
     roles {delegated_role};
```

5 Conclusions

We have addressed the need for an implementation of role-based delegation in SELinux. We also articulated a way to further enhance the flexibility of SELinux supporting delegation features addressed in [8]. We also demonstrated feasibility of the proposed approach through a proof-of-concept prototype implementation of a system component called RoMan. In the future, more functionalities of delegation and revocation for RoMan will be investigated. A future experiment also includes the work of porting the current RoMan code into m4 macros.

References

[1] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[2] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, 2001.

[3] Bill McCarty. SELinux O'Reilly, October 2004, ISBN: 0-596-00716-7

[4] W.E. Boebert and W.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, p.18, 1985.

[5] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of The Eighth USENIX Security Symposium*, 123-139, August 1999.

[6] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, Aug 2002.

[7] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[8] Gail-J. Ahn, Longhua Zhang, Dongwan Shin and Bill Chu. Authorization Management for Role-based Collaboration. In *IEEE International Conference on System, Man and Cybernetic (SMC2003)*, pages 4128-4214, Washington, DC, October 2003.

[9] L. Zhang, Gail-J. Ahn and B. Chu. A Rule-Based Framework for Role-Based Delegation and Revocation. *ACM Transactions on Information and System Security*, Vol.6, No.3, August 2003.

[10] A. Hagstrom, S. Jajodia, F. P. Presicce, and D. Wijesekera. Revocations - a classification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 44–58, Nova Scotia, Canada, June 2001.

[11] E. Barka and R. Sandhu. Framework for role-based delegation model. In *Proceedings of 23rd National Information Systems Security Conference*, pages 101–114, Baltimore, MD, October 16-19 2000.