

Chapter 1

Introduction

1.1 Context and Motivation

Information is very sensitive in healthcare sector. A lot of documentation is done to record all encounter and treatment which a patient undergoes. Data is collected through many types of equipments and persons, and then shared with and analysed by healthcare professionals. A lot of time is being spent by doctors, researchers in collecting and then analyzing this data.

There is lot of effort worldwide to automate the healthcare so that it is reached out to remote corners of the world where healthcare is not available. This would require standardization of healthcare standards and online decision support for healthcare. This would be cost effective that anyone with its health parameters and basic health test results can search for right treatment without human intervention. But to realize such system, medical information of a person have to be shared with many people ranging from a local nurse , to doctors and patients themselves.

Medical Information about a person must be secured so that confidentiality and correctness is maintained. Legally, laws and morals of healthcare protects the information. However as the system widespends by use of IT, the risk of malicious activities also increase. Due to heightened risks, IT should take responsibility in managing the security .

In this project, we are exploring use of role based access system with SELinux to manage the information stored in a healthcare information system and secure the access of sensitive information to required role only, instead of providing access to everyone.

1.2 Role based access control in Healthcare Information System

Since healthcare information is very sensitive and should be confidential, IT-based solutions should take care of this aspect that what information is shared with whom. IT system should control and monitor access for updating, viewing, transmitting and analyzing that sensitive information.

In healthcare information system, therefore it becomes important that access should be regulated with roles rather than person, organization or professionals. These roles should be used to monitor and access control of all information maintained and stored by the system.

1.3 Report outline

The first chapter gives background about access control methods and research work done in this area. The following chapters will talk about RBAC standards written down by NIST, SELinux MAC control system and Hierarchical RBAC based implementation of a healthcare system and how the sensitive information is protected by SELinux policy file even when the prototype healthcare application is rigged by unauthorized access. The last chapter will talk about conclusion and future work to be done in this area.

Chapter 2

Background

2.1 Introduction

Access control in computer systems is a fundamental security mechanism that protects the security and privacy of the information. Access control originated in the early 1960s, when investigations were carried out in order to control the accesses made in multi-user and resource-sharing computer environments. The aim was to create an access control model which could protect the confidentiality and integrity of information at different levels of sensitivity. . In the early 1990s RBAC and TE were introduced into the arena of access control models, providing flexibility and simplicity while implementing security policies.

Along with access control models, considerable effort has been spent in the creation of trusted OSs. One of the first results of these efforts was seen timesharing OS called Multics. Multics provided the basis for the development of current OSs such as UNIX-based OSs. In the last decade, the commercial availability of OSs which enforces a MAC security policy has made an important change in the arena of Trusted OSs. Systems like Trusted Solaris, Trusted BSD and SELinux were released to the public, to be implemented for commercial purposes. However, SELinux is one of the most promising options to provide MAC at the OS layer in future computer systems.

The chapter is composed of sections which define and describe the main terms, technologies and concepts related with access control.

2.2 Access Control

Access control, is the process which determines whether a user has permission to conduct a specific action over resources in a system.

Normally, in a computer system, users first have to log in using some type of authentication mechanism. Once the user has logged in, the access control mechanisms restrict the operations users are able to perform with the resources. Usually this is done comparing the user identifier against an access control database. The access control database reflects the company's security policies and the permission level assigned to users and groups.

Access control can be considered as one of the most fundamental security mechanisms used in computer systems today. In current OSs a type of access control mechanism always exists in order to restrict the way in which users have access to the resources. In a business perspective, almost all organizations implement a type of access control to restrict access to facilities or information.

2.2.1 Authentication and Authorization

Both, authentication and authorization are fundamental to access control. Authentication is about verifying the identity of a user, while authorization is concerned with verifying the authority of the user. Even if this seems to be obvious these two terms are commonly confused. As stated by Ferraiolo, Kuhn, and Chandramouli (2003), the confusion lies on the need of authentication in order to properly manage authorization. Authorization restricts the access permissions users have over the system resources, but if the user is not properly authenticated then the confidentiality or integrity of the information could be compromised.

Authentication is the process which determines whether a claimed identity is legitimate. Authentication is essential for access control since access control is based on the identity of the user which determines authorized access permissions on the resources. The token used to perform authentication is called an authenticator.

The most common form of authentication is the use of a password. Systems that make use of passwords have the belief that the knowledge of the password guarantees the authenticity of the user. The problem with passwords is that they can be stolen, accidentally revealed or forgotten.

Authentication is the basis for authorization, while verifying the identity of the user during the log in process. Once in the system, users have to be restricted to authorized operations in order to preserve the three security principles. Authorization is the process which determines the operations users are permitted to perform while in the system. In computer systems, the user is not the one who directly access to the resources, this is done through applications that work on behalf of the user. Since applications cannot be trusted, accesses have to be restricted to prevent damage occurring from compromised applications. For doing this, authorization is the preferred solution to determine access permissions that users have while in the system according to the security policy implemented in the system.

2.2.2 Access Control Models

Access control models can be defined as "those that decide on the ways in which the availability of resources in a system are managed and collective decisions of the nature of the environment are expressed" (Tolone, Ahn, Pai, and Hong, 2005). There are different models of access control which can be

implemented in a computer system. These can be categorized into two types (Tolone et al., 2005):

- **Passive Access Control Models.** These access controls models are those whose primary function is to maintain permission assignment without regard to the context in which the permission is assigned. Examples of these types of models are DAC, MAC and RBAC.
- **Active Access Control Models.** These access control models are those which take into account the context in which access to the resources have to be granted. Examples of these models are: Dynamic, Context-Aware Access Control (Hu and Weaver, 2004) and Open Architecture for Secure Interworking Service (OASIS) (Bacon, Moody, and Yao, 2003).

2.2.2.1 Discretionary Access Control

Currently DAC is the most commonly used access control model to enforce access control over resources in computer systems (Liu et al., 2007a). DAC is based on the principle that the access to the information is at the discretion of the creator of the information. DAC is defined by the Trusted Computer System Evaluation Criteria (TCSEC) (Department of Defense [DoD], 1985) as those controls in which a subject with certain access permissions is capable of passing those permissions to other users or applications acting on behalf of users. For example, an authorized user, *Bob*, can create a resource (file) and pass access permissions over that resource to other users, such as *Alice*, as a result *Alice* get full access permissions over that resource.

It has been recognized that DAC mechanisms are inadequate due to the discretion in which permissions are passed between users. In DAC systems, if the application running on behalf of the users, and consequently granted with all

the permission of the user, is compromised, the attacker will be able to modify resources far beyond the needs of the application and could compromise the entire system.

2.2.2.2 Mandatory Access Control

MAC is a preferred access control model to provide a truly secure scheme in which systems are guaranteed to remain secure (Ferraiolo et al., 2003). As defined by Loscocco et al. (2001), MAC systems are those in which "the policy logic and the security attributes are tightly controlled by a system security policy administrator". In the TCSEC (DoD, 1985) it is stated that in MAC access to the resources is not at the discretion of users, but restricted by the system according to the security policy. MAC takes access decisions based on the comparison of labels containing security relevant information which are assigned to subjects and objects in the system. Access permissions are not at the discretion of the owner of the information. Those who create access and maintain information shall follow rules set by the policy and administered by the organization.

The most common implementation of MAC is MLS which is based on assigning hierarchical clearances and classification to subjects and objects respectively and non hierarchical categories for both. MLS is based on a formal model called the Bell-LaPadula model. MLS models are designed to protect the confidentiality and integrity of the information in a very strict and inflexible manner, which is appropriate for military environments but not for commercial environments.

2.3 Access Control Lists

An Access Control List (ACL) is the most common access control mechanisms implemented in OSs to determine allowed access permissions users have over resources in the system (Daswani, Kern, and Kesavan, 2007). ACLs list the users with right to access and object together with the type of permitted access such as read, write, or execute. A common representation of the ACL is a set of users and corresponding set of resources to which they are allowed to access. This relationship can be seen as follows (Daswani et al., 2007):

User	Resource	Privilege
Alice	/home/Alice/*	read, write, execute
Bob	/home/Bob/*	read, write, execute

Table 1 - Access Control List.

An ACL can be seen as discretionary if the owner of the object can fully control the privileges and users in the ACL. On the other hand, an ACL can be seen as mandatory if the ACL is controlled by the system according to a system-wide security policy.

In a more sophisticated way, ACLs can also enforce RBAC in order to simplify the user-permission assignment in systems with a large number of users. In these systems users are assigned to roles instead of privileges enabling the users to access particular resources. The ACL in this type of systems could be seen as follows:

User	Role
Alice	Doctor, Nurse
Bob	Doctor

Table 2 - Roles in ACLs.

And the ACL assigning the permission could be seen as follows:

Role	Resources	Privileges
Doctor	/healthcare/doctors/*	read, write, execute
Nurse	/healthcare/nurses/*	read, write, execute

Table 3 - Role-Permission assignment in ACLs.

In this case the user *Alice* is assigned to the roles Doctor and Nurse allowing *Alice* to read, write and execute the content in the /healthcare/doctors and /healthcare/nurse directories. On the other hand, *Bob* is only authorized to read, write and execute the contents of the directory /healthcare/doctor.

In OSs which makes use of ACLs to manage access rights, each objects ACL is identified by its security attribute. This list contains an entry of access privileges for each system user. When a subject performs an operation on an object, the system first checks the ACL corresponding to the object for an applicable entry in order to determine whether or not the operation proceeds. Within OSs which make use of ACLs are: Windows NT family systems, Novell NetWare, and Unix-based systems. Each of these systems has a different way of implementing ACLs, but its function remains the same.

2.4 Reference Monitor

The reference monitor concept was introduced in a report published in 1972 by the Computer Security Technology Planning Study, conducted by James P. Anderson. In this report, known as the "Anderson Report", the reference monitor was introduced as a module which "validates all references to programs or data according to the access authority of the user on whose behalf the program is executing" (Anderson, 1972).

The reference monitor mediates every reference made by programs in execution against the list of permissions authorized for the user (Anderson, 1972). In the reference monitor, the system resources are isolated in two main groups based on the distinction between passive and active entities. Active entities within the system such as running processes are grouped into subjects, and passive entities such as files are grouped into objects.

The reference validation mechanisms, also known as the reference monitor mechanisms, are responsible for the validation of accesses from subject to objects in the system. When a subject makes an access requests over an object in the system, the reference validation mechanisms authorize the request based on the comparison between the security attributes of the subject with that of the object, and the information contained in an access control database. The access control database represents the access control policy in terms of subjects and objects security attributes and access rights. Access decisions made by the reference validation mechanisms are based on the security attributes associated with each subject and object in the system.

2.5 Mandatory Access Control Models

The "Anderson Report" was the trigger which initiated an increasing research on formal security models to formally describe security policies. That is, formal security models are used to describe the entities to which the security policy applies and the rules to control its behavior. One of the pioneer models in computer security was the Bell and LaPadula model which focuses on the confidentiality of classified information. In later years, formal security models to protect the integrity of the information, such as Biba, were developed due to the importance of integrity. However, the Biba and Bell- LaPadula models, common examples of MLS, are very inflexible for commercial organizations. For this reason models like Clark-Wilson and RBAC models were created.

In this section, formal models of access control are introduced in order to provide a background on the protection of the security principles in computer systems.

2.5.1 Bell-LaPadula Model

The Bell-LaPadula model is a formal state-transition model focusing on the confidentiality of classified information (LaPadula, 1996). The model was introduced in 1973 by David Elliot Bell and Len LaPadula as part of a research to protect classified information in military environments.

A secure state can be defined as the condition in which no unauthorized access has occurred to confidential data according to the security policy. The Bell-LaPadula security theorem states that if the initial state of the system is secure and all the transitions in the system are secure, then all subsequent states are going to be secure regardless of any input occurred (McLean, 1985).

The Bell-LaPadula model is based on the isolation of entities in the computer system into subjects and objects. Objects are passive entities in the computer system that contains or receive information.

That is, repositories of information such as files, datasets, etc. Subjects are active entities in the computer systems which are responsible for changing the state of the systems and make the information flow between objects, such as processes.

Every subject and object in the system is labeled with a security attribute which is constituted by a hierarchical and a non-hierarchical component. Subjects are assigned with a security clearance and objects with a security classification. Security clearances and classifications are ordered in hierarchical levels, so that clearances and classifications higher in the hierarchy dominate those in lower levels. The aim is to prevent subjects to access objects which are higher in the hierarchy. Examples of sensitivity levels could be *Top Secret*, *Secret* and *Confidential*. Subjects and objects in the systems are also assigned with categories which are non-hierarchical components. The main purpose of the categories is to enforce the need-to-know principle by restricting subjects to access only those objects which are within its domains. For example, a user with *Top Secret* clearance would be able to access everything in the system even though the user belongs to the financial department and the information is *Confidential* in the IT department. Therefore, it would be desirable to give the user a *Financial* category so that the user is only allowed to access information within the financial department.

Accesses from subjects to objects in the systems are permitted by comparing the security attributes of the subject against the security attributes of the

object. The Bell-LaPadula model states that a security attribute of a subject dominates over the security attribute of an object if and only if:

- The classification of the object is lower or at the same level in the hierarchy than the clearance of the subject; and
- The category set of the object is a subset of the category set of the subject.

For example, assume that there is a subject with the clearance *Secret* and the category set *Finance, IT* and an object with classification *Confidential* and category set *IT*. In this case the subject security attributes dominate over the object security attributes.

2.5.2 Biba Model

In order to solve the problem of unauthorized modification or deletion of the information in the Bell-LaPadula model, the Biba model was designed to protect the integrity of the information. The Biba model was introduced in 1977 as a complement of the Bell-LaPadula model. The Biba model is based on the same characteristics as the Bell-LaPadula model in which every subject and object in the system is labelled with a security level. However, in the Biba model, subjects and objects are labelled with integrity levels instead of confidentiality levels. This prevents subjects in higher security levels to read object in lower security levels, preventing the subject to process data that could compromise the data integrity in a higher level. The integrity security level assigned to a subject indicates the level of trust set on the subject to modify sensitive information and the integrity security level in objects indicates the sensitivity of the information to be modified. Examples of integrity levels could be *Critical, Important, and Ordinary*.

The Biba model uses principles similar to those in the Bell-LaPadula model, but with two main differences: the principles work with integrity levels; and the dominance relations of the principles are reversed.

2.5.3 Role-Based Access Control Model

In 1992, Ferraiolo and Kuhn (1992) proposed the RBAC model which simplifies the complexity and cost of security administration in large scale systems. RBAC was introduced as a relatively simple model in which access to computer system objects is based on a user's role in the organization. In RBAC, permissions are assigned to roles rather than individual users. A role is a collection of permissions that may be assigned to users based on the corresponding organizational job function. All the operations performed by users are accomplished through transactions, except for identification and authentication operations. A transaction is defined as a transformation procedure (change objects from one state to another) and all required access permissions. The paper introduced by Ferraiolo and Kuhn specified three basic requirements in RBAC (Ferraiolo et al., 2003):

- **Role assignment.** A subject can execute a transaction only if the subject has selected or been selected a role.
- **Role authorization.** A subject's active role must be authorized for the subject.
- **Transaction authorization.** A subject can complete a transaction only if the transaction is authorized for the subject's active role.

In 1996, Sandhu, Cope, Feinstein, and Youman (1996) introduced a framework of RBAC models known as RBAC96 which specifies four different conceptual models of RBAC. RBACO is the base model including

the minimal requirements for a RBAC system. RBAC1 and RBAC2 include RBACO, but additionally RBAC1 includes roles hierarchies and RBAC2 includes constraints such as SoD. The fourth component, RBAC3, includes the characteristics of both RBAC1 and RBAC2. RBAC96 provided a modular RBAC which can be used according to the different requirements of organizations. A simplified implementation of RBAC in a commercial organization could use RBACO while a more complex implementation could make use of other features in advanced levels of RBAC96.

In 2000 the National Institute of Standard and Technology (NIST) initiated an effort to create a standard for RBAC (Ferraiolo et al., 2003). This standard was based on the RBAC96, but it adds features in regard of requirements from the commercial vendor community.

RBACO (core RBAC) embodies the essential aspects of RBAC, that is, users are assigned to roles, permissions are assigned to roles, and users acquire permissions when assigned to roles. The RBAC standard allows the user-role and role-permission relationships to be done many-to-many. That is, users can be assigned to more than one role and a role can be assigned to many users. In a similar manner, roles can be assigned to one or more permissions and permission to many roles.

RBAC1 (hierarchical RBAC) adds the concept of role hierarchies. Hierarchies are defined as partially ordered senior relationships between roles. RBAC2 (constrained RBAC) add SoD relations to the RBAC model. SoD is used to enforce conflict of interest policies in order to ensure that fraud and errors cannot occur without deliberate malicious agreement between multiple users. The RBAC standard allows for both static and dynamic SoD. Static SoD

enforces constraints on the assignment of users to roles to prevent conflict of interest in a role-based system.

2.5.4 Domain and Type Enforcement Model

The domain and type enforcement (DTE) model was introduced in 1995 as an enhanced version of the traditional TE model, designed to address some existing issues in the model (Badger, Sterne, Sherman, Walker, and Haight, 1995). As with many other access control models, TE splits a system into two sets of logical entities: subjects and objects. Access control attributes are assigned to subjects and objects in the system. Domains are associated with subjects and types are associated with objects. Access control permissions are associated with both domains and types. Accesses are allowed to be made between domains and from domains to types. Permissions therefore, can be seen in two groups which consist of the domain-domain group and the domain-type group.

The DTE model has been compared with the RBAC model because of the way in which the model restricts the operations on objects based on domains. RBAC restricts the operations users are allowed to commit over objects in the system through the use of roles. In a similar way, DTE limits the operation that subjects can do over objects through the use of domains. This similarity allows DTE to implement policies that represent a RBAC model by associating users with subjects and roles with domains (Ferraiolo et al., 2003).

2.6 Layers of Security

In order to create trusted systems, that is, systems which are reliable in regard to the enforcement of a security policy, security mechanisms have to be

properly implemented in different layers in a computer system. These layers are:

- **Application Layer Security.** Application software can provide support and granularity in a complex security policy.
- **Operating System Layer.** The OS is responsible for the overall management of the hardware resources in the system, consequently is the one responsible for providing ways to control access to these resources.
- **Hardware Layer.** Hardware can provide the means to protect the integrity for the OS boot process, audit trails and/or logs.
- **Network Layer.** Network layer security has to be provided in order to ensure that only valid data packets are received in web servers and malicious traffic is not allowed to interact with applications and the OS.

2.7 Operating Systems Mandatory Access Control

Security at the application layer has been improved, but there are still daily reports about security breaches in computer systems due to compromised applications.

Kernel-enforced access controls at the OS level can be used to mitigate the risks from a compromised application. This idea is based on the fact that the kernel access controls cannot be overridden or subverted by any application at the user space level. The OS is able to limit the damage that a compromised application can do to the system and other applications running on the system. This is done through setting strict controls over the resources in the system thus restricting the operations that the applications are allowed to do. Once an

application has been compromised, it can be used to compromise the entire system.

In order to mitigate the risks to these exploits support from the OS has to be implemented in the computer systems. MAC has to be provided by the OS so that applications are contained to specific resources according to a security policy. If DAC mechanisms are used by the OS to protect the resources of the system, an attacker can bypass the mechanisms by acquiring the privileges of the compromised application. Once the OS is configured to enforce a specific security policy applications are restricted to spaces (sandboxes) in which they have limited privileges to specific resources. If compromised, the application cannot damage the system by affecting resources outside its space.

There are some OS that have implemented MAC to create what is called a Trusted System. These systems are created so that they can enforce a specific security policy through the use of access control mechanisms in the kernel level. The following sections provide an overview of this type of OSs.

2.7.1 Security Enhanced Linux

SELinux was initially developed by the NSA based on the implementation of MAC in microkernel systems. In 1999, as an outcome from this project the Flux Advanced Security Kernel (Flask) architecture was created to provide better support for dynamic security policies.

SELinux was released in December 2000 by the NSA and developed with cooperation from highly recognized security institutions such as NAI Labs, SCC, and MITRE. In 2001, the Linux Security Module (LSM) project was

commenced in order to allow modular addition of different security extensions into the Linux kernel. Once the LSM framework was completed, the NSA began to adapt SELinux to use the LSM framework.

Currently, SELinux is shipped as part of Fedora Core, and is supported by Red Hat as part of Red Hat Enterprise Linux. Due to the effort from the NSA and Red Hat to integrate SELinux as part of the mainline Fedora Core Linux distribution, Fedora Core was released with SELinux enabled by default since Fedora Core 4.

SELinux is a Linux variant that makes use of the LSM to implement MAC in the Linux kernel. SELinux implements a type of MAC called TE which is based on the assignment of security attributes (labels) to every object and subject in the system. SELinux also provides a form of RBAC built upon TE in which roles are used to group domain types and relate these domains with users, but decisions are based on TE rules instead of RBAC permission assignments. SELinux manages orthogonal user's identifiers mapped to traditional Linux UID. In this way, the mandatory accesses controls introduced by SELinux are kept orthogonal to traditional DAC mechanisms in Linux. This improves the level of accountability of the actions made by a user. SELinux provides support for policy changes and is independent of policy, policy languages, and labeling format.

The use of SELinux restricts activities of an attacker by distributing authority to users and removing excessive privileges from processes. If an application is limited to only necessary privileges, an attacker who takes control of the application cannot damage the whole system. SELinux creates sandboxes defined by domains assigned to each process in which activities of processes are limited by the sandbox boundaries.

2.8 Summary

In order to protect the confidentiality, integrity and availability of resources in the systems, access control mechanisms have to be properly implemented. Access control models have been created over the years to provide ways to control the access of resources.

RBAC is one of the most widespread adopted models in commercial organizations, including healthcare organizations.

In order to create systems that can be trusted in regard to their enforcement of a security policy, security mechanisms have to be implemented at different layers of security. Systems have to be constructed with the support from mechanisms at the OS layer. In the last decade Sun Microsystems and Red Hat made a significant change in the arena of trusted OS by introducing Solaris Trusted Extensions and SELinux for commercial organizations.

Chapter 3

Introduction to RBAC

3.1 Introduction

Traditional computer systems provide Discretionary Access Control (DAC) and Mandatory Access Control (MAC) mechanisms to provide access control mechanism for various objects in the system. In DAC, Access control is decided by the owner of an object. Owner can delegate access rights to other users. In MAC, Security labels are associated to subjects and objects. Access control is decided by the system.

But as organizations are growing and changing so there is a need of access control system which can be easily adaptable to changes in organization structure. System should be adaptable to change in role of user, higher role more privileges in the system.

Access control is decided on the role played by different users in the organization. It is similar to concept of groups in linux. It categorizes the groups of users and group of permissions as compared to user groups which define user sets.

RBAC provides the capability to add relation between users-roles and between roles- permissions.

3.2 RBAC Concept of Role and Permission

Permission is set of action which a user can have while accessing an object, an object can be a file, directory or some executable. Permission can be defined as a pair of object-access method in an object oriented environment. We can also assign roles to users.

In RBAC method, we can define permission as an entity to access an object as in Figure 1. Therefore RBAC is access control mechanism which help administration in overcoming frequent organization changes in term of number of users and also organization structural changes by defining complex control policies.

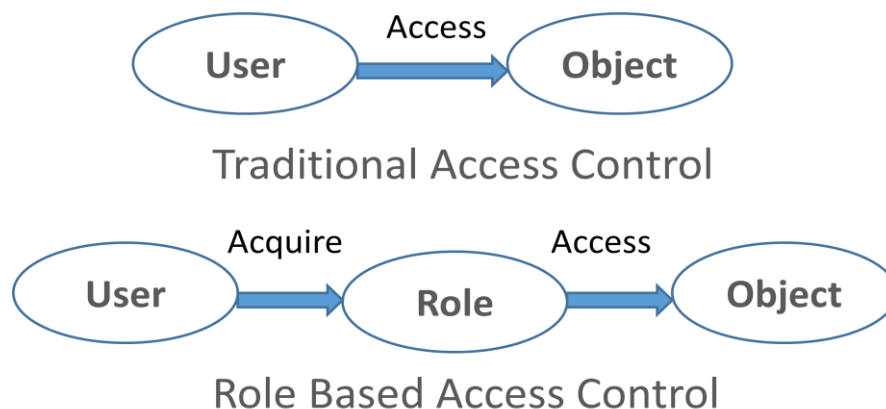


Figure 1 - Traditional Access and RBAC(Ravi S. Sandhu “ Role-Based Access Control “).

Following relationships define RBAC access control policy:

- Role-Permission Relationships
- User-Role Relationships
- Role-Role Relationships

The above relationships define what kind of access user can have for an object in system.

Following diagram, defines how relationships flow from users to roles and then from roles to permissions to individual components in system.

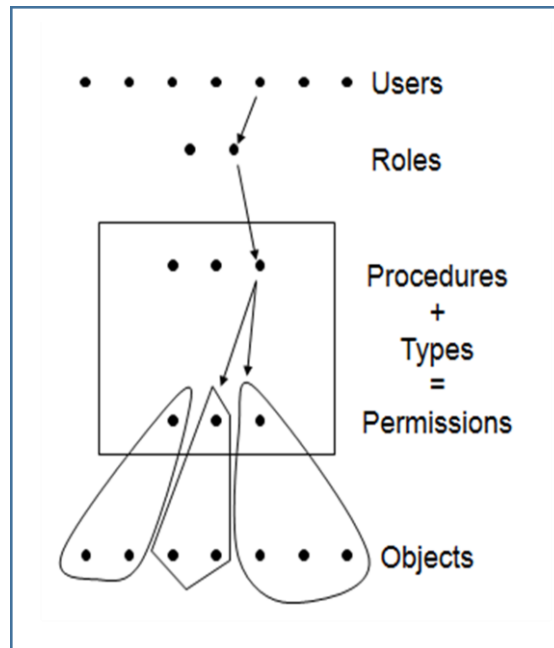


Figure 2 - Relationship Flow (Ravi S. Sandhu “ Role-Based Access Control “)

3.3 Principles of RBAC

RBAC works on following three principles:

- Least Privilege
- Separation of duties
- Data Abstraction

Least Privilege means assigning limiting the privileges/permissions to the users which are required to perform a particular role. It requires understanding users roles and responsibilities and identifying minimum permissions which are required to be assigned to a role.

Separation of duties means defining separate mutual exclusive roles. If a clerk is required to check the account status and then issue a cheque, two different roles should be created for a issuing clerk and account manager instead of single role of a clerk. Since the activity requires two different operations.

Data Abstraction is realised by defining abstract permissions like debit and credit for an account. The implementation details define extent to which data abstraction is supported.

3.4 Applications

Following is the list of applications which uses some form RBAC.

- Microsoft Active Directory
- Microsoft SQL Server
- SELinux
- FreeBSD
- Solaris
- Oracle DBMS
- PostgreSQL 8.1

3.5 RBAC Standards

RBAC cannot be treated as a single model. As a single model may either include or exclude many things, and would represent only a part of technology and choices.

NIST(National Institute of Standards and Technology) proposed RBAC standardization , so that issues of different definitions and scope of similar concepts can be addresses.

The standard begins by defining basic RBAC elements: user, roles, permissions,

operations, and objects) and relations as types and functions that are included in this standard. This helps in defining scope of RBAC features and meaning of role hierarchies, static constraint relation and dynamic constraint relations.

The NIST RBAC model(Sandhu R., Ferraiolo D. and Kuhn R) is defined in terms of four model components.

- Core RBAC
- Hierarchical RBAC
- Static Separation of Duty Relations
- Dynamic Separation of Duty Relations

Each of above components defines basic element sets and basic relationships between these elements. And a set of mapping function to define mapping between various element sets.

3.5.1 Core RBAC

It defines the basic concept of RBAC. The basic concept of RBAC is that users are assigned roles and user acquires permissions by owning up a role.

It defines many to many relations between a user and roles, and also roles and permissions; User can be assigned multiple roles simultaneously

Following diagram(,efines Core RBAC.

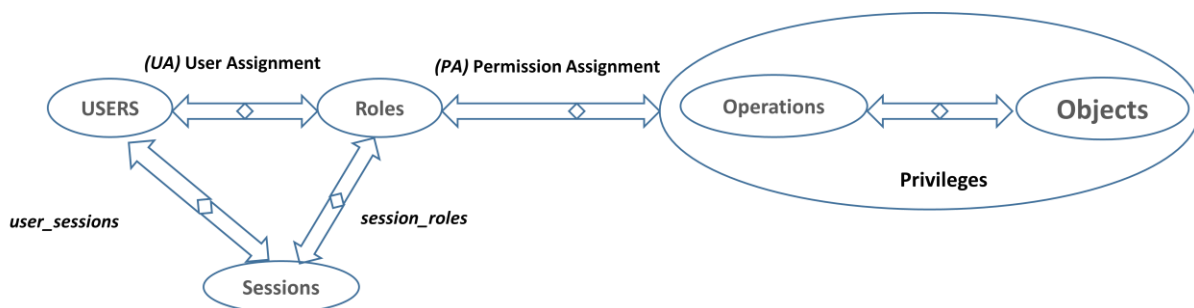


Figure 3 - Core RBAC(Sandhu R. et. al.)

A session is a mapping between user and current active set of roles for a user. Privileges are domain/system dependent.

Core RBAC defines basic concepts of **USERS**, **ROLES**, **PERMISSIONS** and **SESSIONS**.

User: A human being or intelligent autonomous agent

Role: Job functions within the context of organization and defines the authority and responsibility

Permission: an operation that can be exercised on objects. Objects and operations are domain dependent

Example: DBs

objects are tables, columns, and rows

operations are insert, delete, and update

Session

It is an instance of a connection of a user to the system. It defines the subset of activated roles. Different sessions for the same user can be active at each time

For Example:

Given the following User-role assignment and Permission-role assignment matrix:

User	Role
Alice	Radiologist
Alice	GP
Bob	GP
Charlie	Radiologist
David	Nurse

Table 4 - User-role Matrix

Role	Permission
Nurse	(read, prescription)
GP	(read, prescription)
GP	(write, prescription)
GP	(read, history)
Radiologist	(read, history)
Radiologist	(insert, image scan)

Table 5 - Permission-role Matrix

Seeing above user-role and role-permission tables, we can easily make out access control table as following:

User	Prescription	History	Image Scan
Alice	Read Write	Read	Insert
Bob	Read Write	Read	
Charlie		Read	Insert
David	Read		

Table 6 - Access Control Table

3.5.2 Hierarchical RBAC

It adds support of role Hierarchies. A hierarchy is an order of defining seniority of roles between various roles, it includes concept of multiple inheritance to roles and permissions assigned to the roles.

- **General Hierarchical RBAC:** It provides support of any arbitrary order to define role hierarchy. So that the concept of multiple inheritances of permissions and user membership among roles can be supported.
- **Limited Hierarchical RBAC:** Some systems may impose restrictions on the role hierarchy. Most commonly, hierarchies are limited to simple structures such as trees and inverted trees.

As shown in figure below, it adds support of role hierarchies while defining ad

deducing permissions for various roles.

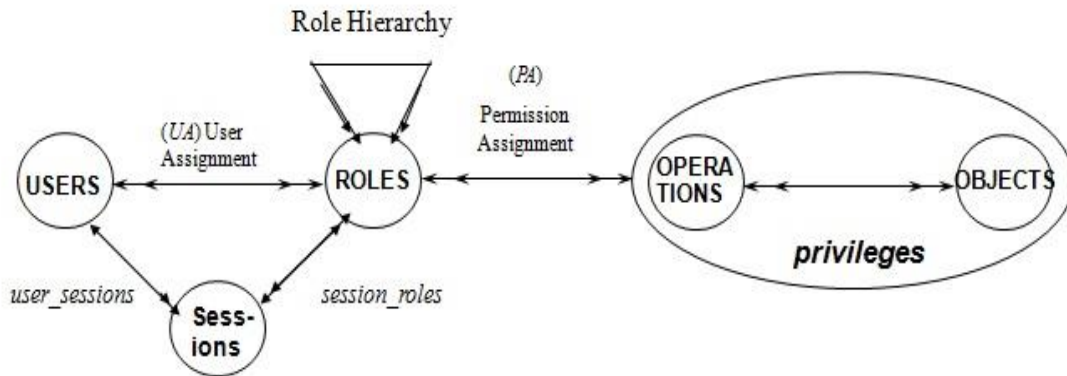


Figure 4 - Hierarchical RBAC(Sandhu R. et. al.)

Following figure, shows roles hierarchy in a hospital:

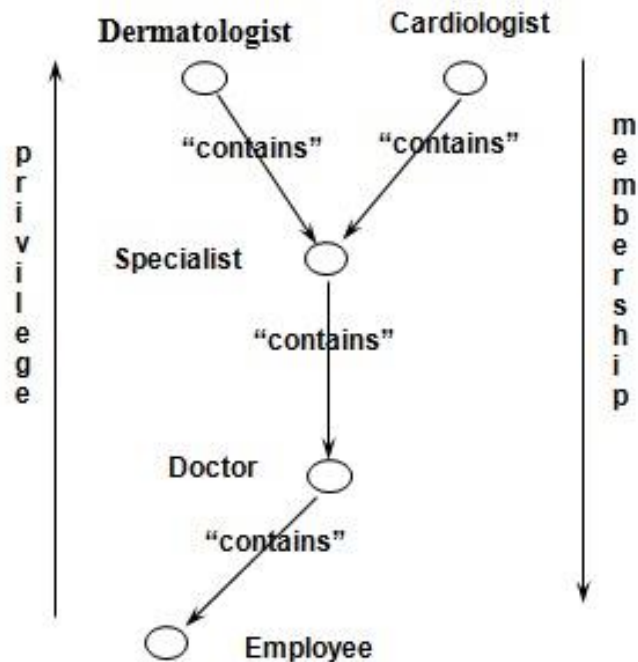


Figure 5 - Hierarchy in a Hospital(Nicola Zinnanon)

So as the membership grows over the hierarchy tree, the privileges and set of permissions increases. And the node up in hierarchy contains rights of roles which are below in hierarchy.

As shown in figure below, Director Role will have all permission of a project leader 1 and project lead 2 roles. Similarly Project lead role will automatically have permissions of both quality engineer and Production engineer role down in hierarchy.

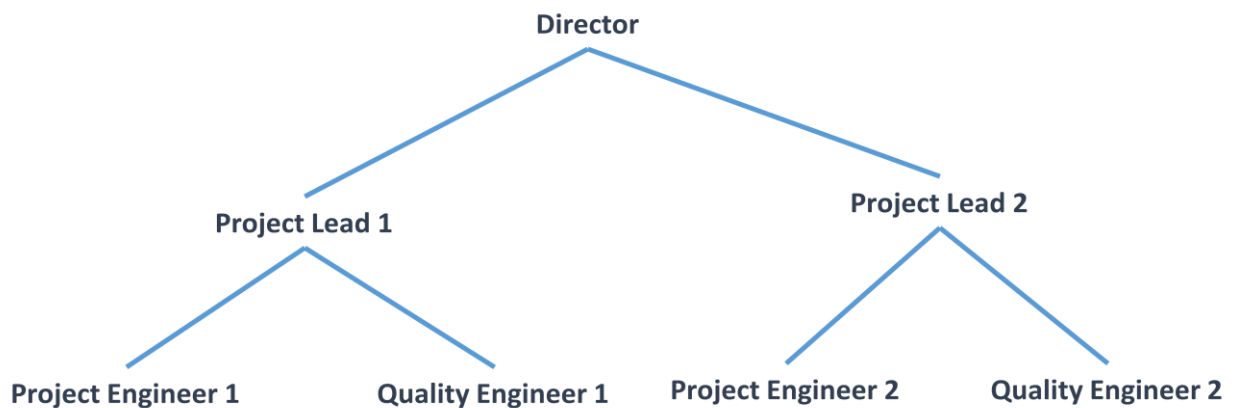


Figure 6 - Hierarchy in a Corporate(Nicola Zinnanon)

General Role Hierarchy, it supports multiple inheritances, so which mean ability to inherit permission of two or more roles down the hierarchy.

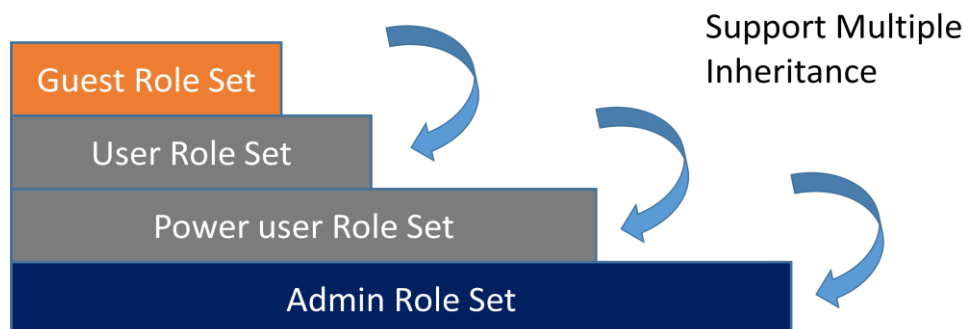


Figure 7 - General Role Hierarchy(Sandhu R. et. al.)

Limited role hierarchy, it limits inheritance of role hierarchy to a immediate descendant only, as shown in figure below:

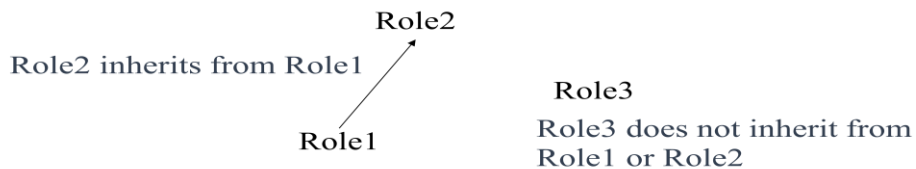


Figure 8 - Limited Role Hierarchy(Sandhu R. et. Al)

3.5.3 Static Separation of Duty Relations

Separation of duty relations can enforce interest policies conflict. A user gaining certain permissions with conflicting roles can result in conflicts of interest. *Static separation of duty* (SSD) can prevent this conflict by enforcing constraints on the defining the users-roles.

One example of static constraint is the by defining two roles to be mutually exclusive; for example, if one role does some changes and another reviews them, the same user would be prohibited from being assigned to both roles.

The SSD policy can be specified centrally and then imposed on overall system roles. Due to likelihood for inconsistencies with respect to static separation of duty relations and inheritance relations of a role hierarchy, we define SSD requirements are defined in both cases in the presence or absence of role hierarchies.

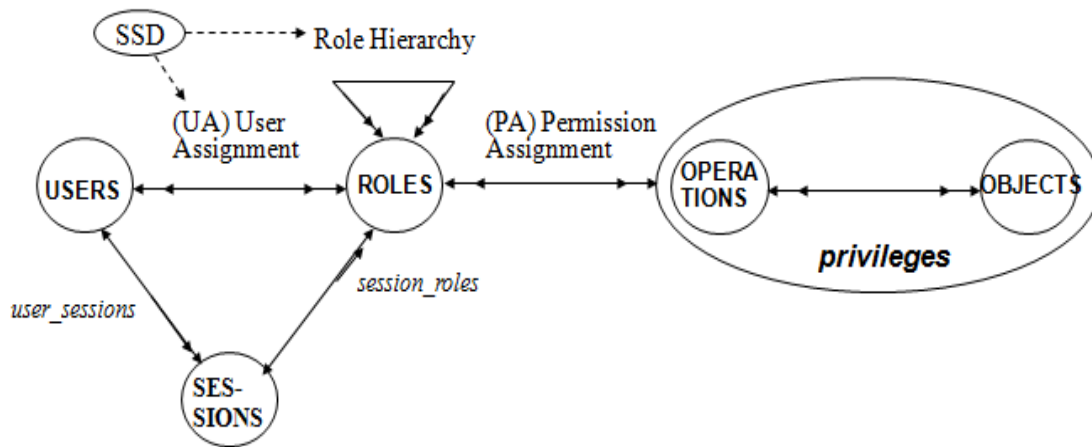


Figure 9 - Static Separation of Duty (Sandhu R. et. Al)

So SSD policies deter fraud by placing constraints on administrative actions and there by restricting combinations of privileges that are available to users.

E.g., No user can be a member of both Cashier and AR Clerk roles in Accounts Receivable Department

Static Separation of Duty SSD relations monitors and controls the user and role relations. SD rules prevents one user to be assigned a role if it is already assigned some other role.

Static Separation of Duty in the Presence of a Hierarchy. It is similar to SSD with addition that inherited roles are also considered while enforcing the constraints.

3.5.4 Dynamic Separation of Duty Relations

Dynamic separation of duty (DSD) relations, like SSD relations, limits the permissions that are available to a user. However the context in which limitations are imposed makes DSD relations differ from SSD relations.

The constraints are placed on the roles that can be activated within or across a user's sessions, this limits the availability of permissions.

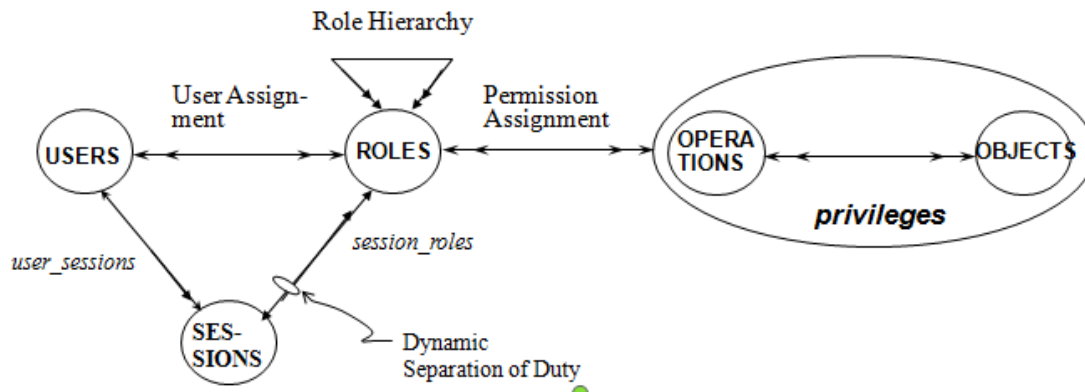


Figure 10 - Dynamic Separation of Duty Relations (Sandhu R. et. Al)

3.6 Methodology to create an RBAC Package

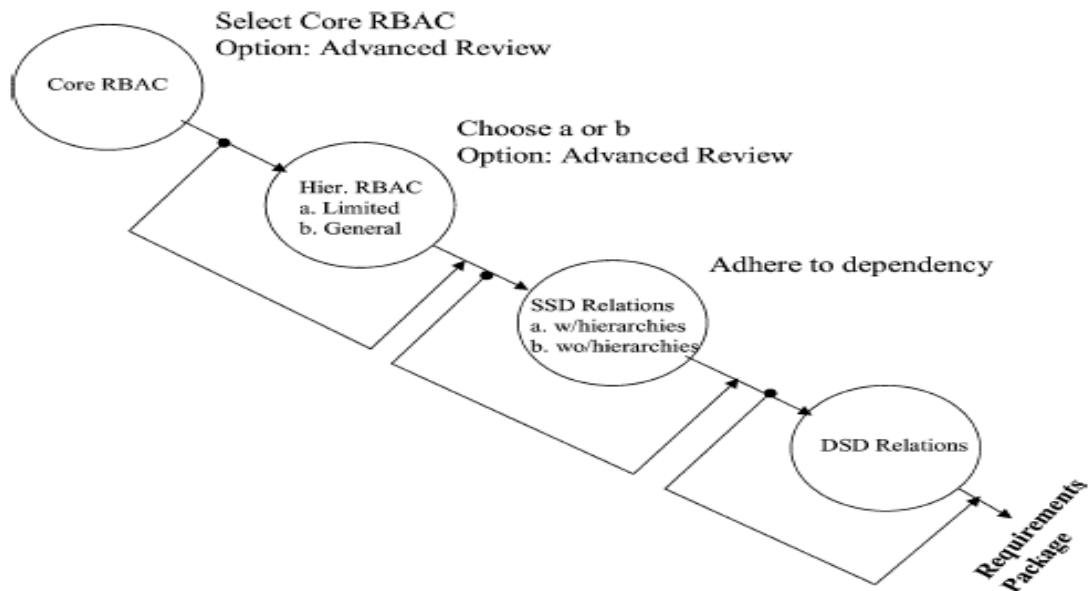


Figure 11 - Methodology to Create RBAC Package (Nicola Zinnanon)

The figure, above sums up the steps to create a RBAC package. From figure one can see, we start with basic component of RBAC model and keep on upgrading the package by applying tenets of other RBAC models.

Chapter 4

SELinux to Enforce Mandatory Access Control in Health Information Systems

4.1 Introduction

SELinux provides a flexible policy configuration allowing a high level of granularity when protecting system resources. Flexibility in the security mechanisms is an important feature while supporting security policies in healthcare organizations with different security requirements and resources. The provided access control mechanisms need to be reliable in the enforcement of security policies while allowing flexibility to support variations in the security policies.

In Access control is a fundamental mechanism in multi-user systems such as HIS. Access control has to be enforced at the OS layer reflecting the security policy of the healthcare organization. OSs that implement MAC mechanisms are a preferred solution in which access to the system resources do not rely on the owner of the resources, but in the security policies implemented in the OS.

SELinux implements a flexible and fine-grained MAC in the Linux kernel. This is done through the use of a flexible architecture called Flask and the LSM framework. SELinux use a type of MAC called TE and a type of RBAC which is built upon TE. In addition, SELinux provides security features and tools to simplify the implementation and management of SELinux policies.

4.2 SELinux Architecture

In order to introduce a flexible implementation of MAC in the Linux Kernel, SELinux was implemented using the LSM framework. The LSM framework allows loading different access control models in the Linux Kernel as loadable kernel modules

4.2.1 Linux Security Module Framework

It is a framework which uses hooks to call modules which enforce the security policy in the system. These hooks mediate accesses to different kernel objects, such as files, sockets, processes, and so on.

LSM hooks are responsible to call the modules which enforce the security policy configured in the Linux system. The hooks only ask a simple question: is this access allowed or not? In this way, the modules can enforce any different access control module without the whole architecture needing to be modified. However, a drawback of this model is that in case the request is rejected there is no way to identify the main reason of the rejection. In case of a functional error within the access control module, the system will reject the request without knowing if it was due to policy enforcement or functional error.

SELinux is implemented as a set of hooks that are located throughout the Linux Kernel for policy enforcement and a LSM module which is called for access control decision making.

4.2.2 Flask Architecture

SELinux was designed with the idea to create a MAC solution flexible enough to support a wide variety of security policies in real-world environments.

In the Flask security architecture there is a clear distinction between mechanism and policy to enable a variety of policies to be supported with less policy-specific customization. The Flask security architecture provides three main components for object management, which are (Spencer, 1999):

- **The Security Server.** Its main role is the policy decision making. The security server provides interfaces for retrieving access, labeling and policy instantiation decisions.
- **The Access Vector Cache (AVC).** The AVC allows the object manager to cache decisions made by the security server to minimize the performance overhead.
- **The Object Managers.** These are the responsible for defining a control policy which enforces the decisions made by the security server over the objects they manage. The Object Managers also have to provide the mechanisms to label the objects they manage according to the specifications of the security server.

According to the Flask security architecture every object controlled by the security policy is labeled with a set of attributes known as the security context. The Flask architecture provides two data types for object labeling which is (Spencer, 1999):

- **A Security Context.** The security context can be considered to be an opaque string which might consist of different attributes depending on the security policy.
- **The Security Identifier (SID).** This data type is a 32-bit integer which is only interpreted by the security server and is mapped to a security context. The SID is interpreted by the AVC as an opaque that uniquely references a security context.

When an object is created within the objects managed by the object manager, it is assigned a SID and it defines the security context under which the object is created. The security server is the one responsible for choosing a unique identifier as the SID for the object which is computerized from the security context. The security context assigned by the security server to the object typically depends on the client requesting the object creation and the environment in which the object is created (i.e. the object class and/or security context of the directory).

The AVC is a common security decision library shared between object managers. The AVC is able to coordinate the policy between the security server and the object manager.

The use of the SID allows more flexibility with the content and the format of the security context. The object manager assigns a SID to every objects and subject that they manage without concern of the way in which the security policy works with the security contexts. This allows a strong distinction between security policy decision making and enforcement functions. Hence, the Flask security architecture allows completely replacing the security server with a new access control policy without changing the object managers.

As shown in Figure 12, the Flask architecture in SELinux is reflected in the SELinux LSM module. In SELinux the security server and the AVC are contained in the SELinux LSM module and the object managers are represented by the LSM hooks.

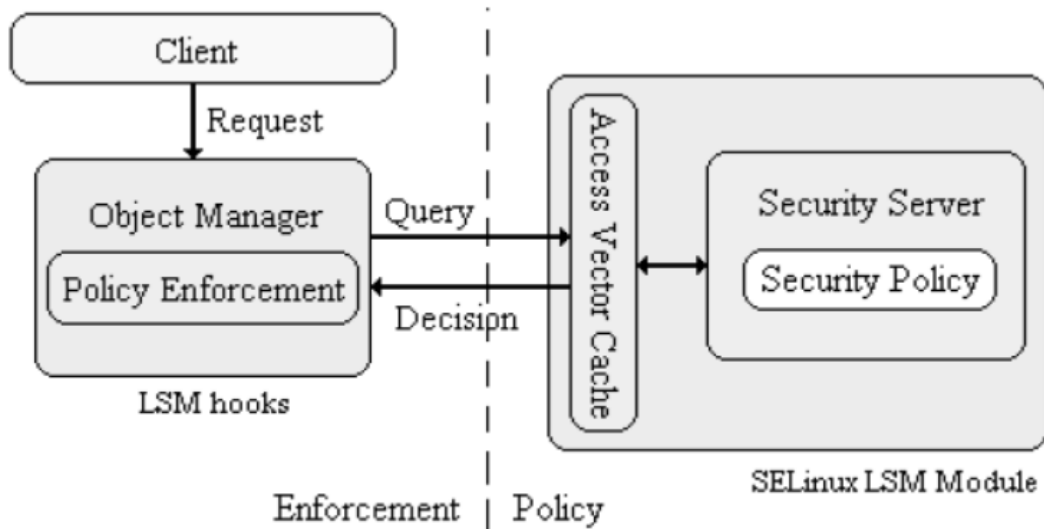


Figure 12 - SELinux LSM Module and the Flask Architecture

The flask architecture organizes the operating system components and data into subjects and objects. Subjects are processes: applications, drivers, system tasks that are currently running. Objects are fixed components such as files, directories, sockets, network interfaces, and devices. For each subject and object, a security context is defined. A security context is a set of security attributes that determines how a subject or object can be used.

SELinux uses a combination of the Type Enforcement (TE), Role Based Access Control (RBAC), and Multi-Level Security (MLS) security models. Type Enforcement focuses on objects and processes like directories and applications, whereas Role Based Access Enforcement controls user access. For the Type Enforcement model, the security attributes assigned to an object are known as either domains or types. Types are used for fixed objects such as files, and domains are used for processes such as running applications.

For user access to processes and objects, SELinux makes use of the Role Based Access Control model. When new processes or objects are created, transition rules specify the type or domain they belong to in their security contexts. With the RBAC model, users are assigned roles for which permissions are defined. The roles restrict what objects and processes a user can access.

4.2.2.1 Terminology

SELinux uses several terms that have different meanings in other contexts. The terminology can be confusing because some of the terms, such as domain, have different meanings in other, related, areas. For example, a domain in SELinux is a process as opposed to an object, whereas in networking the term refers to network DNS addresses.

4.2.2.2 Identity

SELinux creates identities with which to control access. Identities are not the same as traditional user IDs. At the same time, each user normally has an SELinux identity, though the two are not linked. Affecting a user does not affect the corresponding SELinux identity.

SELinux can set up a separate corresponding identity for each user.. A general user identity is used for all normal users, restricting users to user-level access, whereas administrators are given administrative identities.

Should a user change user IDs, that user's security identity will not change. A user will always have the same security identity. In traditional Linux systems, a user can use commands like `su` to change his or her user ID, becoming a different user. On SELinux, even though a user can still change his or her Linux user ID, the user still retains the same original security ID.

The security identity can have limited access. So, even though a user may use the Linux `su` command to become the root user, the user's security identity could prevent him or her from performing any root user administrative commands.

Use `id -Z` to see what the security context for a security identity is, what roles the identities have, and what kind of objects it can access. This will list the user security context that starts with the security ID, followed by a colon, and then the roles a user has and the objects the user can control.

The following example shows a standard user with the general security identity:

```
$ id -Z
```

```
user_u:user_r:user_t
```

In this example the user has a security identity called `george`:

```
$ id -Z
```

```
george:user_r:user_t
```

The `new role` command can be used to change the role a user is allowed. Changing to a system administrative role, the user can then have equivalent root access.

```
$ id -Z
```

```
george:sysadm_r:sysadm_t
```


4.2.2.3 Domains

Domains control processes, a process executed by different domains acquire different domain. A process executed under a domain, is restricted by its own set of permissions. A domain, on the other hand, can be tailored to access some areas but not others. For example, the administrative domain is `sysadm_t`, whereas the DNS server uses only `named_t`, and users have a `user_t` domain.

4.2.2.4 Types

Types control objects like files and directories. Files and directories are grouped into types that can be used to control who can have access to them. The type names have the same format as the domain names, ending with a `_t` suffix.

4.2.2.5 Roles

Users (security identities) with a given role can access types and domains assigned to that role. For example, most users can access `user_t` type objects but not `sysadm_t` objects. The types and domains a user can access are set by the role entry in configuration files. The following example allows users to access objects with the `user password` type:

```
role user_r types user_passwd_t
```

4.2.2.6 Security Context

Each object has a security context that set its security attributes. These include identity, role, domain or type. A file will have a security context listing the kind of identity that can access it, the role under which it can be accessed, and the security type it belongs to. Each component adds its own refined level of

security. Passive objects are usually assigned a generic role, `object_r`, which has no effect; as such objects cannot initiate actions.

A normal file created by users in their own directories will have the following identity, role, and type. The identity is a user and the role is that of an object. The type is the user's home directory. This type is used for all subdirectories and their files created within a user's home directory.

```
user_u:object_r:user_home_t
```

A file or directory created by that same user in a different part of the file system will have a different type. For example, the type for files created in the `/tmp` directory will be `tmp`

```
user_u:object_r:tmp_t
```

4.2.2.7 Transition: Labeling

A transition, also known as labeling, assigns a security context to a process or file. For a file, the security context is assigned when it is created, whereas for a process the security context is determined when the process is run.

Making sure every file has an appropriate security context is called labeling. Adding another file system requires that labels are added (add security contexts) to the directories and files on it. Labeling varies, depending on the policy is used.

Each policy may have different security contexts for objects and processes. Relabeling is carried out using the `fixfiles` command in the policy source directory.

```
$fixfiles relabel
```

4.2.2.8 Policies

A policy is a set of rules to determine the relationships between users, roles, and types or domains. These rules state what types a role can access and what roles a user can have.

4.2.2.9 SELinux Policy Rules

Policy rules can be made up of either type (Type Enforcement, or TE) or RBAC (Role Based Access Control) statements. A type statement can be a type or attribute declaration or a transition, change, or assertion rule. The RBAC statements can be role declarations or dominance, or they can allow roles. Policy configuration can be difficult, using extensive and complicated rules. For this reason, many rules are implemented using M4 macros in fi files that will in turn generate the appropriate rules.

4.2.2.10 Type and Role Declarations

A type declaration starts with the keyword `type`, followed by the type name (identifier) and any optional attributes or aliases. The type name will have a `_t` suffix. Standard type definitions are included for objects such as files. The following is a default type for any file, with attributes `file_type` and `sysadmfile`:

```
type file_t, file_type, sysadmfile;
```

The root will have its own type declaration:

```
type root_t, file_type, sysadmfile;
```

Specialized directories such as the boot directory will also have their own type:

```
type boot_t, file_type, sysadmfile;
```

A role declaration determines the roles that can access objects of a certain type. These rules begin with the keyword `role` followed by the role and the objects associated with that role. In this example, the amanda objects (`amanda_t`) can be accessed by a user or process with the system role (`system_r`):

```
role system_r
types amanda_t;
```

Types are also set up for the files created in the user home directory:

```
type user_home_t, file_type, sysadmfile, home_type;
type user_home_dir_t, file_type, sysadmfile, home_dir_type;
```

4.2.2.11 File Contexts

File contexts associate specific files with security contexts. The file or files are listed first, with multiple files represented with regular expressions. Then the role, type, and security level are specified. The following creates a security context for all files in the `/etc` directory (configuration files). These are accessible from the system user (`system_u`) and are objects of the `etc_t` type with a security level of 0, `s0`.

```
/etc(/.*)? system_u:object_r:etc_t:s0
```

Certain files can belong to other types; for instance, the `resolve.conf` configuration file belongs to the `net_conf` type:

```
/etc/resolv\conf.* -- system_u:object_r:net_conf_t:s0
```

Certain services will have their own security contexts for their configuration files:

```
/etc/amanda(/.*)? system_u:object_r:amanda_config_t:s0
```

File contexts are located in the `file_contexts` file in the policy's contexts directory, such as `/etc/selinux/targeted/contexts/files/file_contexts`.

4.2.2.12 User Roles

User roles define what roles a user can take on. Such a role begins with the keyword `user` followed by the username, then the keyword `roles`, and finally the roles it can use. These rules can be found in the SELinux reference policy source code files. The following example is a definition of the `system_u` user:

```
user system_u roles system_r;
```

If a user can have several roles, then they are listed in brackets. The following is the definition of the standard user role in the targeted policy, which allows users to take on system administrative roles:

```
user user_u roles { user_r sysadm_r system_r };
```

4.2.2.13 Access Vector Rules: allow

Access vector rules are used to define permissions for objects and processes. The `allow` keyword is followed by the object or process type and then the types it can access or be accessed by and the permissions used. The following allows processes in the `amanda_t` domain to search the Amanda configuration directories (any directories of type `amanda_config_t`):

```
allow amanda_t amanda_config_t:dir search;
```

The following example allows Amanda to read the files in a user home directory:

```
allow amanda_t user_home_type:file { getattr read };
```

The next example allows Amanda to read, search, and write files in the Amanda data directories:

```
allow amanda_t amanda_data_t:dir { read search write };
```

4.2.2.14 Role Allow Rules

Roles can also have allowed rules. Though they can be used for domains and objects, they are usually used to control role transitions, specifying whether a role can transition to another role. These rules are listed in the RBAC configuration file. The following entry allows the user to transition to a system administrator role:

```
allow user_r sysadm_r;
```

4.2.2.15 Transition and Vector Rule Macros

The type transition rules set the type used for rules to create objects. Transition rules also require corresponding access vector rules to enable permissions for the objects or processes. Instead of creating separate rules, macros are used that will generate the needed rules. The following example sets the transition and access rules for user files in the home directory, using the `file_type_auto_trans` macro:

```
file_type_auto_trans(privhome, user_home_dir_t, user_home_t)
```

The next example sets the Amanda process transition and access rules for creating processes:

```
domain_auto_trans(inetd_t, amanda_inetd_exec_t, amanda_t)
```

4.2.2.16 SELinux Policy Configuration Files

Configuration files are normally changed using .te and .fc files. These are missing from the module headers in `/usr/share/selinux`. To add a module .te and .fc files are needed to be created for it.

4.2.2.17 Compiling SELinux Modules

Instead of compiling the entire source each time a change is required, a module can be compiled for the changed area only. The modules directory holds the different modules. Each module is built from a corresponding .te file. The `checkmodule` command is used to create a .mod module file from the .te file, and then the `semodule_package` command is used to create the loadable .pp module file as well as a .fc file context file.

For example if developer needs to change the configuration for `syslogd`, first use the following to create a `syslogd.mod` file using `syslogd.te`. The `-M` option specifies support for MLS security levels.

```
checkmodule -M -m syslogd.te -o syslogd.mod
```

Then use the `semodule_package` command to create a `syslogd.pp` file from the `syslogd.mod` file. The `-f` option specifies the file context file.

```
semodule_package -m syslogd.mod -o syslogd.pp -f syslogd.fc
```

To add the module use `semodule` and the `-i` option. `semodule -i syslogd.pp`

4.2.2.18 Interface Files

File interface files allow management tools to generate policy modules. They define interface macros for current policy. The `refpolicy` SELinux source file

will hold .if files for each module, along with .te and .fc files. Also, the .if files in the /usr/share/selinux/devel directory can be used to generate modules.

4.2.2.19 Types Files

In the targeted policy, the modules directory that defines types holds a range of files, including nfs.te and network.te configuration files. The .te files are no longer included with standard SELinux installation. Instead, download and install the sere policy source package. This is the original source and allows user to completely reconfigure SELinux policy, instead of managing modules with management tools like semanage. The modules directory will hold .te files for each module, listing their TE rules.

4.2.2.20 Module Files

Module is located among several directories in the policy/modules directory. There will be three corresponding files for each application or service. There will be a .te file that contains the actual Type Enforcement rules, an .if, for interface (a file that allows other applications to interact with the module), and the .fc files that define the file contexts.

4.2.2.21 Security Context Files

Security contexts for different files are detailed in security context files. The file_contexts file holds security context configurations for different groups, directories, and files. Each configuration file has an .fc extension. The types.fc file holds security contexts for various system files and directories, particularly access to configuration files in the /etc directory. In the SELinux source, each module will have its own .fc file, along with corresponding .te and .if files.

4.2.2.22 User Configuration: Roles

Global user configuration is defined in the policy directory's users file. It contains the user definitions and the roles they have for standard users (user_u) and administrators (admin_u). To add new users, use the **local.users** file.

If a new user needs no special access, the generic SELinux user_u identity is used. If, however user can take on roles that would otherwise be restricted, such as a system administrator role in the strict policy, and configure the user accordingly. To do this, add the user to the **local.users** file in the policy user's directory, as in `/etc/selinux/targeted/policy/users/local.users`. Note that this is different from the local.users file in the src directory, which is compiled directly into the policy. The user rules have the syntax:

```
user username roles { rolelist };
```

The following example adds the sysadm role to the george user:user george roles { user_r sysadm_r }; Once the role is added, reload the policy.
make reload

Developer can also manage users with the semanage command with the user option. To see what users are currently active, and list them with the semanage user command and the -l option.

```
# semanage user -l

system_u: system_r

user_u: user_r sysadm_r system_r

root: user_r sysadm_r system_r
```

The semanage user command has a, d, m, options for adding, removing, or changing users, respectively. The a and m options let specify roles to add to a user, whereas the d option will remove the user.

4.2.2.23 Policy Module Tools

To create a policy module and load it, several policy module tools are used. First the check module command is used to create .mod file from a .te file. Then the semodule_package tool takes the .mod file and any supporting .fc file, and generates a module policy package file, .pp. Finally, the semodule tool can take the policy package file and install it as part of SELinux policy.

4.2.2.24 Sample Files

Sample type file: cash_register.te

```
policy_module(cash_register,1.0.1)

#####

#

# Declarations

#

type cash_register_exec_t;

files_type(cash_register_exec_t);

cashier_role_domain(cashier)
cashier_role_domain(mgr)
```

```
type cashier_topdir_t;
files_type(cashier_topdir_t);

type final_file_t;
files_type(final_file_t);

type final_dir_t;
files_type(final_dir_t);

role cashier_r types { cashier_t cashier_register_t };

role mgr_r types { mgr_t mgr_register_t cashier_register_t };

allow system_r mgr_r;

allow system_r cashier_r;

allow mgr_r cashier_r;

allow mgr_register_t cashier_dir_t:dir search_dir_perms;

allow mgr_register_t cashier_file_t:file r_file_perms;

allow mgr_register_t final_dir_t:dir { add_entry_dir_perms create_dir_perms };

allow mgr_register_t final_file_t:file { append_file_perms create_file_perms };

type_transition mgr_register_t final_dir_t:file final_file_t;
```

Sample file context definition file : cash_register.fc

```
/data -d gen_context(system_u:object_r:cashier_topdir_t,s0)

/data/cashier_r      -d    gen_context(system_u:object_r:cashier_dir_t,s0)

/data/mgr_r         -d    gen_context(system_u:object_r:mgr_dir_t,s0)

/data/final        -d    gen_context(system_u:object_r:final_dir_t,s0)

/data/cashier_r/*. * -d    gen_context(system_u:object_r:cashier_dir_t,s0)

/data/mgr_r/*. *   -d    gen_context(system_u:object_r:mgr_dir_t,s0)

/data/final/*. *   -d    gen_context(system_u:object_r:final_dir_t,s0)

/data/cashier_r/*. * --   gen_context(system_u:object_r:cashier_file_t,s0)

/data/mgr_r/*. *   --   gen_context(system_u:object_r:mgr_file_t,s0)

/data/final/*. *   --   gen_context(system_u:object_r:final_file_t,s0)
```

Sample interface file – cash_register.if

```
interface(`cashier_role_domain',`

    type $1_register_t; # cashier_t running /bin/register.py

    domain_type($1_register_t)

    userdom_unpriv_user_template($1);

    corecmd_shell_entry_type($1_t);

    corecmd_exec_shell($1_t);

    domain_entry_file($1_t, shell_exec_t)

    auth_domtrans_pam_console($1_t);

    domain_transition_pattern($1_t, cash_register_exec_t, $1_register_t);

    domain_entry_file($1_register_t, cash_register_exec_t);

    domain_auto_trans($1_t, cash_register_exec_t, $1_register_t);

    allow $1_register_t $1_t:process sigchld;

    allow $1_register_t $1_tty_device_t:chr_file { rw_term_perms append };

    allow $1_register_t $1_devpts_t:chr_file { rw_term_perms append };

    type $1_file_t;

    files_type($1_file_t);

    type $1_dir_t;

    files_type($1_dir_t);
```

4.3 Conclusion

This chapter introduced SELinux as a feasible approach to protect resources at the OS layer in HIS. The flexibility of mechanisms in SELinux allows implementing SELinux in different organisations with different security requirements. SELinux implements a flexible and fine-grained MAC called TE and a type of RBAC built upon TE. These two mechanisms satisfy requirements such as domain separation and enforcement of the least privilege principle. Also, the use of RBAC helps to simplify user management tasks. Additional technologies such as loadable policy modules and conditional policies are also desirable characteristics while managing SELinux Policies. Conditional policies allow making changes to the policy on the spot and loadable policy modules simplify the creation and implementation of SELinux Policies. Therefore, SELinux is a recommended viable approach to aid in the protection of resources in HIS.

Chapter 5

Hierarchal Role-Based Access Control and Type Enforcement for Health Information Systems

5.1 Introduction

This chapter introduces a framework, based on SELinux Profiles, to implement and manage SELinux in HIS. SELinux Profiles are introduced as the way in which processes running on behalf of users are restricted to specific resources in the system. In this way, damage from compromised applications can be controlled. SELinux Profiles use TE and RBAC to restrict authorized access permissions over the system resources. SELinux Profiles are created by loadable policy modules, which help to simplify the creation and implementation of SELinux Profiles. In addition, conditional policies allow the simplification of the management of SELinux Profiles when changes to the SELinux Policy have to be made on the spot.

In the following section, SELinux Profiles are introduced to demonstrate how TE and RBAC can be used to protect the resources in the system. SELinux profiles make use of TE and RBAC in order to restrict the operations that users are allowed to perform while working in the Linux system.

5.2 SELinux Profiling

SELinux profile is defined as:

The authorized environment for subjects which determine the way authorized users interact with subjects and objects in the system.

Figure 14, shows the process followed while assigning a SELinux profile to an authorized user. Once the user is properly authenticated in the system (authentication mechanisms are out of the scope of this research), the "log in" process will assign the user with a Linux UID. The Linux UID is mapped to a SELinux UID which remains orthogonal to the Linux UID.

The SELinux UID is the one used to determine the SELinux profile that corresponds to the user according to the SELinux policy. The SELinux policy determines the authorized roles, domains and types that are associated with the SELinux UID. These roles, domains and types along with corresponding TE rules constitute the SELinux Profiles. For example, a user named *Alice* can be assigned to a SELinux profile which authorizes *Alice* to access domains (applications) and types (files, executable files) authorized for physicians. SELinux profiles also determine the way in which the domains authorized for a user interact with subjects and objects in the system. That is, when *Alice* runs an application, the application running on behalf of *Alice* is restricted to certain access permissions over system resources.



Figure 13 - SELinux Profiles Assigning Process.

SELinux profiles are based on SELinux TE and RBAC along with SELinux technologies such as the conditional policies and loadable policy modules. Types, domains, TE rules, RBAC rules, booleans and conditional statements constitute the SELinux Profiles. These components are coded in loadable policy modules to create SELinux Profiles that can be implemented in different systems. Basically, SELinux Profiles are created through the implementation of one or more policy modules. These modules are loaded into the Linux kernel using command line tools (i.e. *semodule*) existing in current Linux distributions (that support SELinux) such as RHEL and Fedora Core. These modules can be

loaded into the kernel and interact with predefined base modules such as the Targeted Policy which is provided by default since Fedora Core 5. If no predefined based modules such as the Targeted or the strict policies are to be used, the modules can be loaded as part of a new Reference Policy. In this research, loadable policy modules were loaded into the kernel as non-based modules, to interact with the base modules provided by the Targeted Policy.

Loadable policy modules are created in such a way that they can be loaded in systems with similar characteristics. Meaning, systems with similar applications and a similar directory tree structure. Once the policy modules are loaded into the kernel, types are then assigned to objects which are similar in the systems. This is important, as once types are assigned, the system is able to restrict the access permissions that the subjects have over objects.

The use of loadable policy modules simplifies the management of SELinux policies, helping to easily deploy, modify and update SELinux policies. The SELinux policy administrator does not have to re-compile the entire policy every time a change is required to the modules. With loadable policy modules, a change would require the modification of only the related module. Only the modified module would then be reloaded into the kernel, not the whole policy. Furthermore, once the module is loaded, the system does not require to be rebooted for the changes to take effect. Changes made to the module are enforced in the system as soon as the module is successfully loaded. However, because objects are labeled when the system starts, if changes to the module require modification to the security contexts of any object, the objects need to be relabeled. In order to do this, the SELinux policy administrator, after successfully loading the modified module, has to restore the security contexts of

those objects affected by the module. Figure 15 shows the steps that a SELinux policy administrator has to follow when managing loadable policy modules.

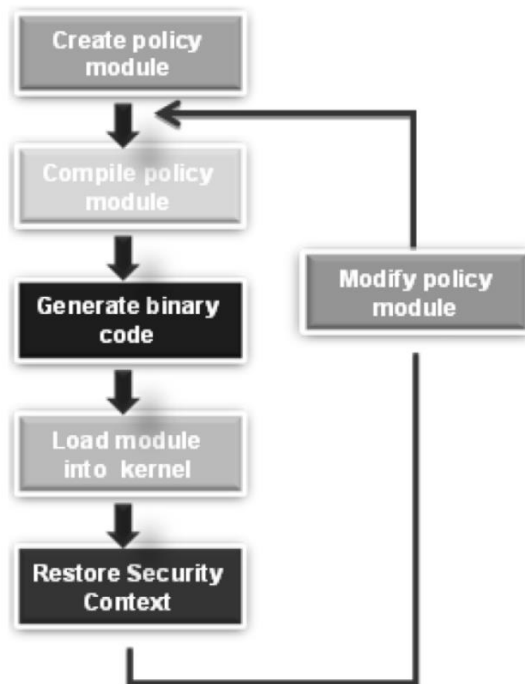


Figure 14 - Loadable Policy Modules Management Process

5.3 Healthcare Scenario

This section is going to describe the implementation and use of SELinux Profiles through the description of a practical healthcare scenario. The scenario demonstrates the circumstances in which SELinux Profiles can contain attacks from malicious code. This scenario is one of the different scenarios used to test the prototype that was developed as proof of concept for this research. Two HIS applications , *appointment* and *hospital_sys* and related files and directories were created as part of this scenario. The policy module for the *appointment* and *hospital_sys* can be found in Appendix. This module contains the access permissions part of the SELinux Profile assigned to users so that they can interact with these applications..

When a new module is developed, this module has to be placed in one of this layers (physically represented by directories) based on the function of the module. For example, modules of the *appointment* and *hospital_sys* applications are placed on the Apps layer since it has its own domain types. That is, these are not directly related with the kernel or any other system services.

Policy modules in the Reference Policy are constituted by three files which are:

- **The type enforcement file.** This file contains all the Type Enforcement logic which is private to the policy. The file also contains the private types and domains of the policy.
- **The interfaces file.** This file contains the interfaces and templates that determine the way to access or create private types and domains in the module.
- **The file contexts file.** This file contains the default security contexts to be assigned to files and directories in the system.

5.3.1 The HIS Applications

The Appointment Application has the following requirements.

The appointment application allows creation of appointment for patients in the HIS. The doctors are authorized to create, read and delete appointments for patients, whereas nurses can only create appointments for patients. Patients can check their appointments and similarly for pathologists can check appointments in the system. Pharmacists wont have any access to appointment application.

The *hospital_sys* application works as application to connect between various users in a HIS. The information related to treatment details, opd records, prescribed tests, test results , allergies and personal information of patients are stored in different directories in the system. Files stored under treatment directory contains date-wise records of treatment given to users, similarly files stored tests directory will contain tests prescribed to patient on different dates. Each patient will have a unique id and files are stored with patient-id as names under various directories.

Doctors are authorized to read patients personal record data, medication history, allergy details, treatment history. The can also update patients medications, allergy details, treatment details. All information will be stored under various directories in separate files for different patients.

Nurses are authorized only to update and read opd parameters.

Patients are authorized to read their personal records, treatment history, allergy details, medication history and test results.

Pathologists can read tests prescribed for patient and can update results.

Pharmacists can read personal records of patients to verify insurance detail and can read medication prescribed to patients.

5.3.2 Working with Roles

The first step is to determine the roles that are going to be allowed to interact with the domains that correspond to the *appointment* and *hospital_sys* applications. The identified roles are described in Table 7.

Role	SELinux Role	Description
Administrator	hosp_admin_r	Can access and update all sections.
Doctor	hosp_doc_r	Read – patient personal info, past medications, past surgery, past OPD visits, past allergies, update OPD visit ,medications, tests.Create , read and delete appointment for patient
Nurse	hosp_nur_r	Read – update basic health parameters for the OPD session.Only create appointment for patient.
Patient	hosp_pnt_r	Read all sections. Read appointment detail.
Pathologist	hosp_path_r	Read and Validate patient has valid insurance. Read medications prescribed. Update section of medicines prescribed. Read and check appointment.
Pharmacist	hosp_pharma_r	Read and Validate patient personal details, tests and update test results. No access to appointment details

Table 7- HIS Roles

These are the roles to be authorized to certain SELinux UIDs. Once the user is authenticated and assigned to his/her corresponding SELinux UID (mapped to the Linux UID), the user is authorized to access certain roles

according to the specifications in the SELinux Policy. The role to which the user is authorized determines the domains that the user can access. That is, the role determines the subjects (processes) that can be accessed by the user.

The following HIS security policy will see following hierarchy of roles:

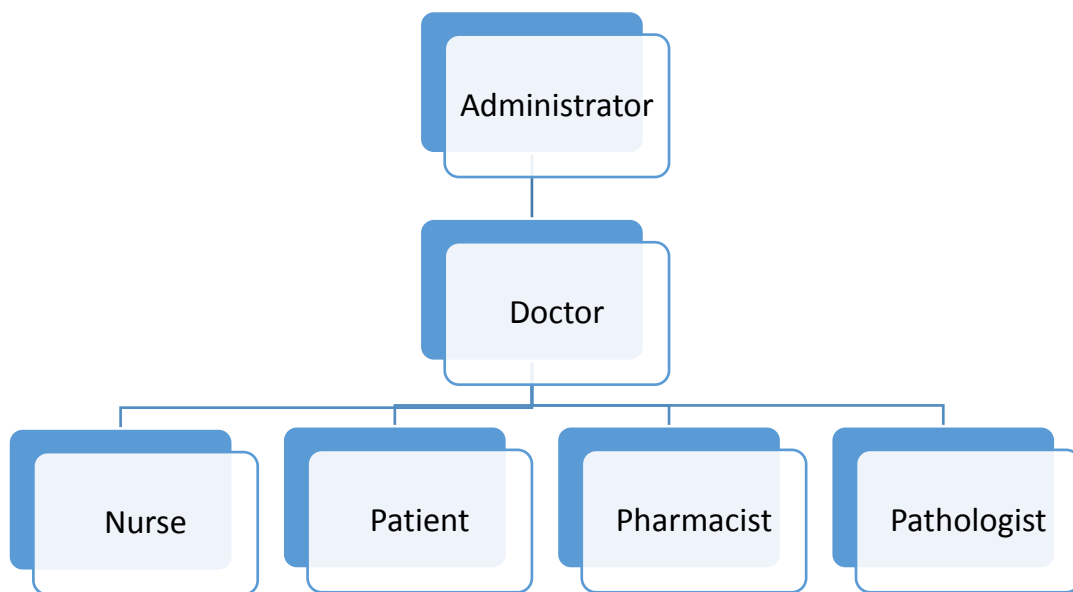


Figure 15: Role Hierarchy in HIS

5.3.3 Role Based Access Control and Type Enforcement

Once the roles are defined, the next step is to determine the domains authorized for each role and the access permissions that each domain has over specific types. In this scenario the types for the objects used by the *appointment and hospital_sys Application* are shown in Table 8.

Object (Resource)	Object	Type
/hosp_sys/appointment	dir	hosp_appntmnt_dir_t
/hosp_sys/personal	dir	hosp_pers_dbdir_t
/hosp_sys/allergys	dir	hosp_allergy_dbdir_t
/hosp_sys/medics	dir	hosp_medic_dbdir_t
/hosp_sys/general	dir	hosp_general_dbdir_t
/hosp_sys/tests	dir	hosp_tests_dbdir_t
/hosp_sys/testresults	dir	hosp_testresults_dbdir_t
/hosp_sys/opd	dir	hosp_opd_dbdir_t
/hosp_sys/appointment/<patient_id>	file	hosp_appntmnt_file_t
/hosp_sys/personal/<patient_id>	file	hosp_pers_dbfile_t
/hosp_sys/allergys/<patient_id>	file	hosp_allergy_dbfile_t
/hosp_sys/medics/<patient_id>	file	hosp_medic_dbfile_t
/hosp_sys/general/<patient_id>	file	hosp_general_dbfile_t
/hosp_sys/tests/<patient_id>	file	hosp_tests_dbfile_t
/hosp_sys/testresults/<patient_id>	file	hosp_testresults_dbfile_t
/hosp_sys/opd/<patient_id>	file	hosp_opd_dbfile_t
/bin/appointment	file	hosp_appntmnt_exec_t
/bin/hospital_sys	file	hosp_main_sys_exec_t

Table 8 - HIS Application Types

The first eight types shown in Table 8 are the directories where different information of patients are stored. The following eight files contain the data corresponding to different types. The last two types are executable files for appointment and hospital_system. Appendix has the code required to set these types as part of the security contexts in the directories and files of the HIS. This code is found in the *hospital_sys.fc* file as part of the policy module created for this scenario.

In this scenario, there could be the possibility that one domain is authorized to all roles. This domain could be the one that corresponds to the *appointment and hospital_sys* process. However, this is not a recommended approach. If only one domain is used, all the roles authorized to access this domain would have the same privileges while using these applications. In this way the least principle privilege is not enforced. If a nurse is able to compromise the code, the nurse would be able to access objects that only doctors, patients and pathologists can access.

In order to improve the granularity and restrict roles to have the least privileges over objects, more than one domain needs to be assigned to the *appointment and hospital_sys* process. Domains have to be created for each role. In this way, the roles are restricted to only specific access permissions over the resources. Even if the code is modified, a nurse would not be able to access resources that are not authorized for that role while using the *appointment and hospital_sys Application*. Table 9 shows the domains created for this scenario along with the role-domain relationship and the access permissions over the object types.

Role	Domain	Type	Permissions		
hosp_doc_r	hosp_doc_main_t	Permissions on files/directories as owned by nurse, patient, pathologist and pharmacist role and in addition to following permissions on other files/directories			
		hosp_medic_dbdir_t hosp_allergy_dbdir_t hosp_general_dbdir_t hosp_tests_dbdir_t	create_dir_perms add_entry_dir_perms		
		hosp_medic_dbfile_t hosp_allergy_dbfile_t hosp_general_dbfile_t hosp_tests_dbfile_t	append_file_perms create_file_perms rw_file_perms		
		hosp_nur_r	hc_nur_main_t	hosp_opd_dbdir_t	create_dir_perms add_entry_dir_perms
		hosp_opd_dbfile_t		append_file_perms create_file_perms rw_file_perms	

hosp_pnt_r	hosp_pnt_main_t	hosp_pers_dbfile_t hosp_allergy_dbfile_t hosp_medic_dbfile_t hosp_general_dbfile_t hosp_tests_dbfile_t hosp_testresults_dbfile_t hosp_opd_dbfile_t	read_file_perms
hosp_path_r	hosp_path_main_t	hosp_tests_dbfile_t hosp_pers_dbfile_t	read_file_perms
		hosp_testresults_dbdir_t	create_dir_perms add_entry_dir_perms
		hosp_testresults_dbfile_t	append_file_perms create_file_perms rw_file_perms
hosp_pharma_r	hosp_pharma_main_t	hosp_pers_dbfile_t hosp_medic_dbfile_t	read_file_perms
hosp_admin_r	hosp_admin_main_t	hosp_pers_dbfile_t	append_file_perms create_file_perms rw_file_perms del_entry_dir_perms search_dir_perms

		hosp_pers_dbdir_t	create_dir_perms add_entry_dir_perms del_entry_dir_perms search_dir_perms
		Permissions on files/directories as owned by Doctor in addition to permission mentioned above.	
Appointment Application domain			
hosp_doc_r	hosp_doc_appnt_t	hosp_appntmnt_dir_t	del_entry_dir_perms
hosp_admin_r	hosp_admin_appnt_t		search_dir_perms
		hosp_appntmnt_file_t	delete_file_perms
		Permissions to appointment files/directory as owned by patients/nurse role below in addition to permissions defined above	
hosp_nur_r	hosp_nur_appnt_t	hosp_appntmnt_dir_t	create_dir_perms add_entry_dir_perms
		hosp_appntmnt_file_t	append_file_perms create_file_perms rw_file_perms
hosp_pnt_r	hosp_pnt_appnt_t	hosp_appntmnt_file_t	read_file_perms
hosp_path_r	hosp_path_appnt_t		

Table 9 - HIS Application Roles, Domains and Types

A user that is authorized to access the role *hosp_doc_r* (doctor) is subsequently authorized to access the domain *hosp_doc_main_t* which is assigned to the *hospital_sys* process. Consequently, this user is authorized only to rights as mentioned in table above. Information about access permissions can be found in Appendix.

These domains and its authorized access permissions over the object types comprise the sandboxes in which the *appointment* and *hospital_sys* runs on behalf of the users. For example, the appointment application when run by patients enters into the sandbox whose boundaries are defined by the authorized permissions of the *hosp_pnt_appnt_t* domain. In this way, any damage from the compromised application is restricted to those resources authorized to the researchers, thus preventing any unauthorized disclosure of the information.

Nevertheless, the first domain which the users' role has to be authorized is the shell process domain. Once the user logs in, the user has to access the domain of the shell process in order to do anything in the system. Table 10 shows the shell process domains associated to the roles used in the *appointments and hospital_sys Application* scenario.

Role	Shell Domain
hosp_admin_r	hosp_admin_t
hosp_doc_r	hosp_doc_t
hosp_nur_r	hosp_nur_t
hosp_pnt_r	hosp_pnt_t
hosp_pharma_r	hosp_pharma_t
hosp_path_r	hosp_path_t

Table 10 - Linux Shell Types

These types have to be authorized to transit into the domains specified in Table 9. This has to be done in order to authorize the shell process running under the user login, to enter the corresponding sandbox. That is, the domain *hosp_doc_t* has to be authorized to transit into the domain *hosp_doc_main_t*, and the domain *hosp_pnt_t* has to be authorized to transit into the domain *hosp_pnt_main_t* and so on.

5.3.4 Creating and Implementing the Policy Module

Once the roles, domains and types along with their corresponding access permissions are defined, the next step is to code them in the policy module. This is done through the use of TE rules and RBAC rules. AVRs, transition rules, and RBAC rules are used to authorize access permissions to the domains and types as specified in Table 6 and Table 7. For example, the TE rules and RBAC rules for the researchers are as follows:

- **Transition Rule.** The following transition rule defines the default domain to transit when the shell process, running on behalf of a doctor

executes the *hospital_sys* executable file. That is, the domain *hosp_doc_t* by default transits to the *hosp_doc_main_t* domain when executing a file with type *hosp_main_sys_exec_t*

```
type transition hosp_doc_t hosp_main_sys_exec_t:file hosp_doc_main_t
```

- **AVRs.** The following AVRs authorize the *hospital_sys* process running on behalf of a pathologist to *search_dir_perms* in the *tests* directory and to read files inside the directory. That is, the domain *hosp_path_main_t* is authorized to search directories with the *hosp_tests_dbdir_t* and read files with type *hosp_tests_dbfile_t*.

```
allow hosp_path_main_t hosp_tests_dbdir_t:dir { search_dir_perms }
```

```
allow hosp_path_main_t hosp_tests_dbfile_t:file { read_file_perms };
```

- **RBAC rules.** The following role statement authorizes researchers to access the shell process and the *hospital_sys/appointment* process. it is required to specify the AVRs for domain transitions as follows:

```
allow hosp_doc_t hosp_main_sys_exec_t:file { getattr read execute };
```

```
allow hosp_doc_main_t hosp_main_sys_exec_t:file entrypoint;
```

```
allow hosp_doc_t hosp_doc_main_t :process transition;
```

These are the rules that allow the shell process domain to transit into the *hospital_sys* domain.

When the policy module is completed, it has to be compiled using the *Makefile* tool, part of the SELinux Policy Development Package, and loaded into the kernel using the *semodule* tool. Once the policy module is loaded in the kernel, the security context of files and directories affected by the policy module have to be restored in case any change is required.

To compile the module, the *Makefile* tool, part of the *selinux-policy-devel* package, is run in the directory in which the policy module is located, as follows:

```
[ user@<path to the policy module> ]# make -f  
/usr/share/selinux/ubuntu/include/Makefile
```

This statement will create the module package to be loaded in the kernel to interact with the base module (provided by the Targeted Policy). The created packet is loaded into the kernel using the *semodule* tool, as follows:

```
[ root@<path to the policy module> ]# semodule -i hospital_sys.pp
```

After the module is successfully loaded into the kernel, it will provide the roles, types, domains, TE rules, and RBAC rules that form part of the SELinux Profiles. For example, once the policy module is loaded into the kernel, it will provide the SELinux Profiles for users of the *hospital_sys* and *appointment* application.

However, before the SELinux Profiles can take effect over the resources used by the *hospital_sys* and *appointment*, the security contexts of files and directories have to be restored. For this reason, once the policy module is successfully loaded into the kernel, it is important to restore the security contexts using the *restorecon* tools. This command uses the security contexts in

the *file context* file of the policy module to define the default security contexts of the files and directories in the system. This command is used as follows:

```
[ root@home ]# >fixfiles -f relabel /hosp_sys /bin/appointment /bin/hospital_sys /home
```

5.3.5 Creating Users and Assigning Roles

Now that the SELinux Profiles for the users have been created, the only thing missing is to create Linux UIDs and map them to SELinux UIDs and corresponding roles. To do this, the system administrator has to first create the SELinux UIDs and associate them to roles. Then system administrator can create Linux UIDs which can be mapped to the SELinux UIDs by using the *semanage* tool. This is done with the *semanage* command as follows:

```
[root@host ~]# semanage user -a -R hosp_admin_r -P hosp_admin hosp_admin_u
```

```
[root@host ~]# semanage login -a -s hosp_admin_u hosp_admin
```

```
[root@host ~]# semanage user -a -R hosp_doc_r -P hosp_doc hosp_doc_u
```

```
[root@host ~]# semanage login -a -s hosp_doc_u hosp_doc
```

```
[root@host ~]# semanage user -a -R hosp_nur_r -P hosp_nur hosp_nur_u
```

```
[root@host ~]# semanage login -a -s hosp_nur_u hosp_nur
```

```
[root@host ~]# semanage user -a -R hosp_pnt_r -P hosp_pnt hosp_pnt_u
```

```
[root@host ~]# semanage login -a -s hosp_pnt_u hosp_pnt
```

```
[root@host ~]# semanage user -a -R hosp_pharma_r -P hosp_pharma  
hosp_pharma_u
```

```
[root@host ~]# semanage login -a -s hosp_pharma_u hosp_pharma
```

```
[root@host ~]# semanage user -a -R hosp_pharma_r -P hosp_pharma  
hosp_pharma_u
```

```
[root@host ~]# semanage login -a -s hosp_pharma_u hosp_pharma
```

This command creates a SELinux UID which is authorized to access the roles.

5.3.6 A closer look at SELinux Profiles

Let's assume that a pathologist called *Alice* is authorized to access the *hospital_sys*. The physician *Alice* will have the following identifiers:

Linux UID: dralice

SELinux UID: hosp_path_u

Authorized access permissions are as stated in Table 9 and Table 10, and roles are those stated in Table 7. In addition, the SELinux UID *hosp_path_u* is authorized to access the role *hosp_path_r*

Following the process shown in Figure 14, once the user logs in, the user is assigned to the Linux UID *dralice*, which is mapped to the SELinux UID *hosp_path_u*. The SELinux UID is used to determine the SELinux Profile that corresponds to *Alice*. In regards to the *hospital_sys*, the SELinux Profile of *Alice* is constituted by the roles *hosp_path_r*, the domain *hosp_path_main_t* and the types *hosp_tests_dbdir_t* and *hosp_testresults_dbfile_t*. Also, the SELinux profile is constituted by the access permissions that the domain *hosp_path_main_t* has over each of the types. Similarly, In regards to the *appointment* application, the SELinux Profile of *Alice* is constituted by the roles *hosp_path_r*, the domain *hosp_path_appnt_t* and the types *hosp_appntmnt_file_t*. Also, the SELinux profile is constituted by the access permissions that the domain *hosp_path_appnt_t* has over each of the types.

The SELinux Profile of *Alice* authorizes the creation of test results through the use of the *hospital_sys* Application. This is authorized because the role *hosp_path_r* can access the domain *hosp_path_main_t*. This domain has the access permissions, create and write over files with type *hosp_testresults_dbfile_t* (this type corresponds to the files containing the test results reports of the patient) and create permission over the directories with type *hosp_testresults_dbdir_t* (which corresponds to the test results directory in HIS).

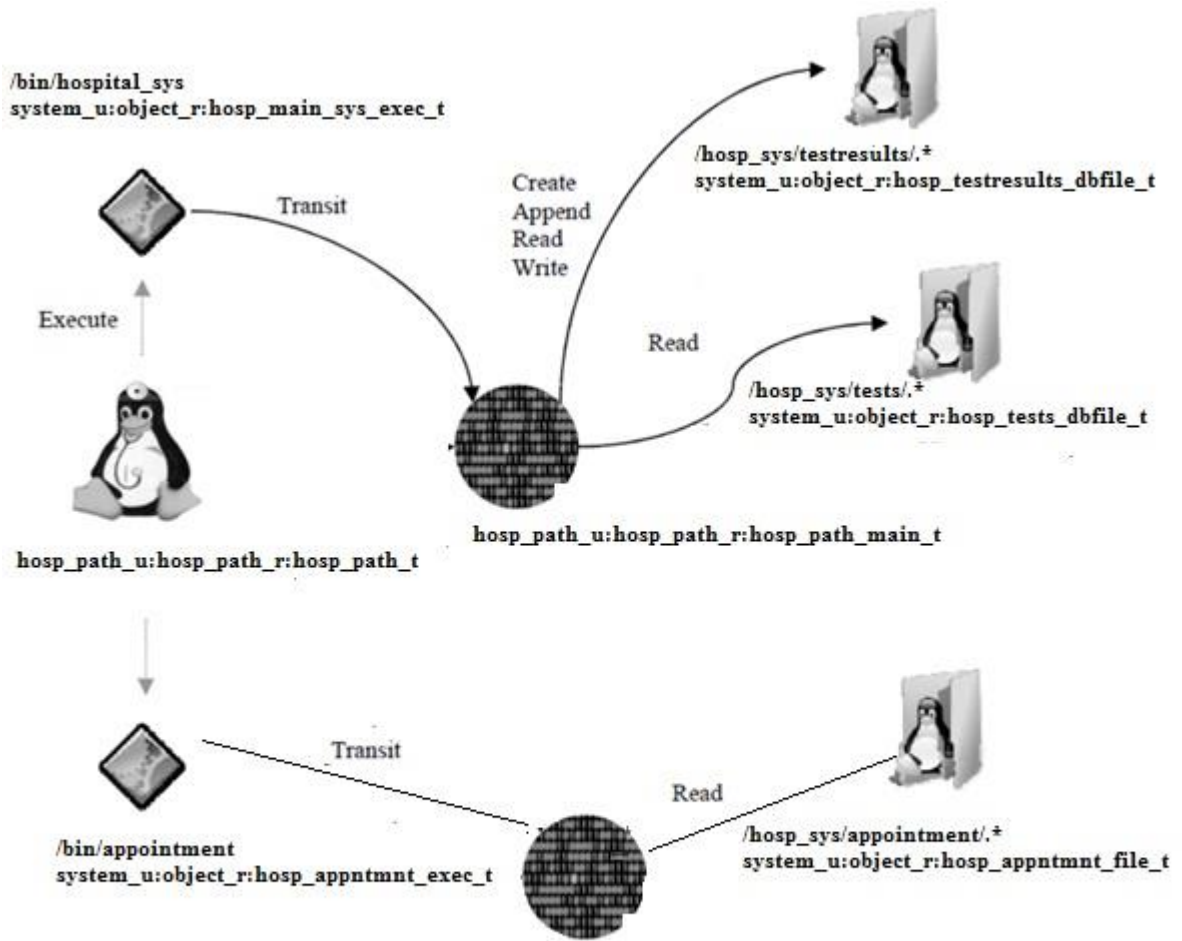


Figure 16 - SELinux Profiles `hospital_sys` and `appointment` applications for Pathologists

5.4 Healthcare Attack Scenario

The attack scenario explained in this section is based on the *hospital_sys* introduced in this chapter. However, this time the code has been attached with a backdoor allowing an attacker to access patients' sensitive healthcare information. SELinux Profiles are going to be used to create sandboxes in order to contain the damage caused by the compromised *hospital_sys*.

In the normal *hospital_sys* Application, only the doctors, nurses and patients are authorized to access the files in the patients' directories. These files contain diagnostic information that can be related to a specific patient. Doctors, nurses and patients are authorized only to access files in the *opd* parameters directory. However, the backdoor created by the attacker allows a pathologist to have access to *opd* parameters files in patients' directories, compromising information of the patients. The *hospital_sys* Application was modified so that when the attacker specifies the *backdoor* parameter, the pathologist is able to access *opd* parameters , which is only allowed for doctors, nurses and patients only

The following section describes how RBAC mechanisms at the OS layer can contain this attack.

5.4.1 RBAC Context

In contrast with traditional Linux systems, SELinux access to the resources is based on a pair of security attributes which are the subject's (source) security context and object's (target) security context. In SELinux, access control attributes for subjects and objects are the security context attributes constituted by the SELinux user identifier, the role and the object's type. These security attributes are assigned to subjects and objects based on the SELinux policy enforced by the Linux kernel.

Let us see the following security attributes of executable *hospital_sys*.

Executable File	Permission bits	Owner ID	Group ID	Executable File Security Context
/bin/hospital_sys	rwX rwX ---	root	Healthcare (51001)	system u:object r: hosp_main_sys_exec_t

Table 11 - Security Attributes for the HIS Application Executable File

Even if the *hospital_sys* executable file has the *rwX* permission bits activated for the owner and group, access to this file depends on the specification in the SELinux Policy. Users would need to be authorized to execute the file in the SELinux Policy. Also, the appropriate transition rules would have to be specified in order to authorize the user to access the domains of the *hospital_sys* application. For example, the TE rules needed to authorize the domain transition of the shell process domain *hosp_doc_t* to the *hospital_sys* Application process domain *hosp_doc_main_t* are as follows:

```

type transition hosp_doc_t hosp_main_sys_exec_t : file
hosp_doc_main_t

allow hosp_doc_t hosp_main_sys_exec_t: file { execute }

allow hosp_doc_main_t hosp_main_sys_exec_t: file entrypoint;

allow hosp_doc_t hosp_doc_main_t : process { transition };
    
```

These AVR will authorize the shell process running on behalf of the user *drpaul* to transit into the *hosp_doc_main_t* corresponding to the *hospital_sys* Application process running on behalf of a doctor. The domain type *hosp_doc_main_t* could be considered to be the sandbox which is going to

restrict the operations of the *hospital_sys application* running in behalf of any doctors.

In a similar way, the pathologists shell processes have to be authorized to transit into the domain of the *hospital_sys Application* process running on behalf of pathologists. In this case, instead of transiting into the domain *hosp_doc_main_t* the pathologists are authorized by the SELinux Profile to access the domain *hosp_path_main_t*. The *hosp_path_main_t* domain represents the sandbox that restricts the operations of the *hospital_sys Application* running on behalf of any pathologist.

The authorized types, domains and access permissions defined by the SELinux Profiles (for doctor and pathologist only) are shown in Table 12. For example, the domain *hosp_doc_t* belonging to a shell process running on behalf of a doctor, is able to transit to the domain *hosp_doc_main_t* and read files with the type *hosp_opd_dbfile_t*.

Role	Domain	Type	Permissions
hosp_doc_r	hosp_doc_main_t	hosp_pers_dbfile_t hosp_testresults_dbfile_t hosp_opd_dbfile_t	read_file_perms
		hosp_medic_dbdir_t hosp_allergy_dbdir_t hosp_general_dbdir_t hosp_tests_dbdir_t	create_dir_perms add_entry_dir_perms

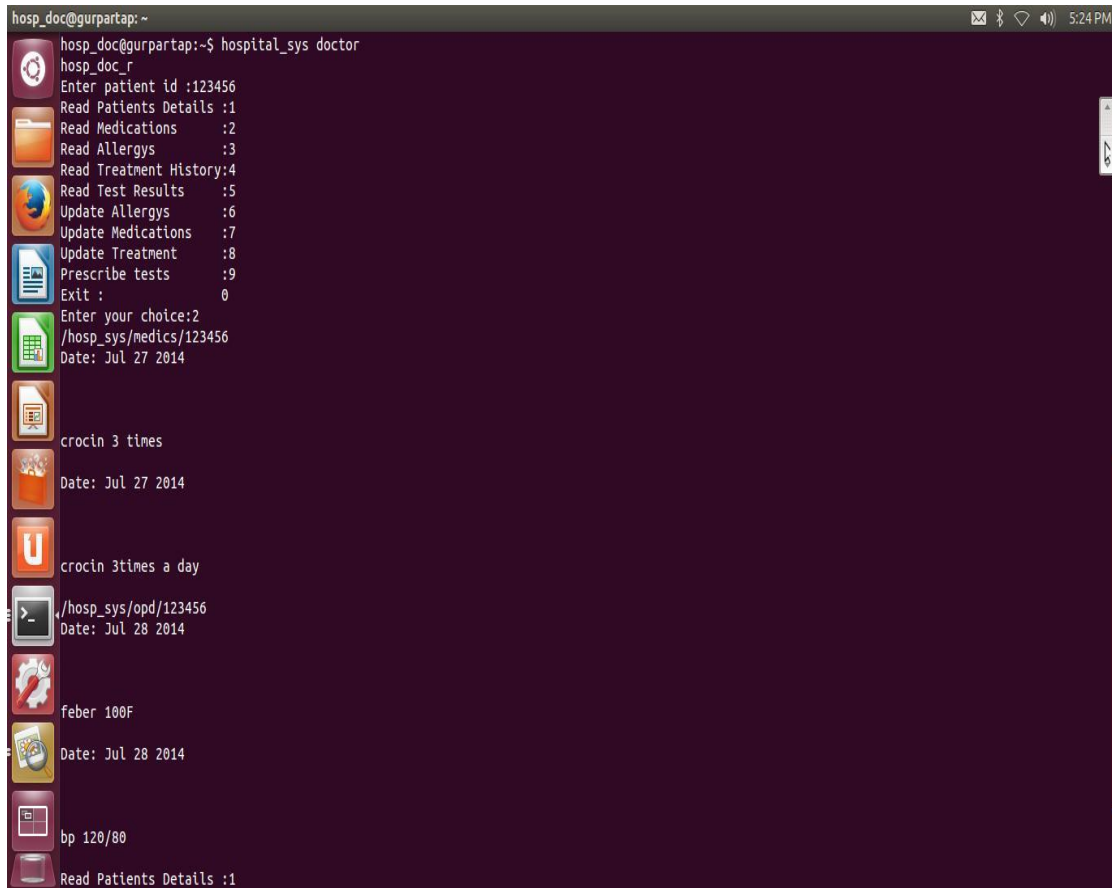
		hosp_medic_dbfile_t hosp_allergy_dbfile_t hosp_general_dbfile_t hosp_tests_dbfile_t	append_file_perms create_file_perms rw_file_perms
hosp_path_r	hosp_path_main_t	hosp_tests_dbfile_t	read_file_perms
		hosp_testresults_dbdir_t	create_dir_perms add_entry_dir_perms
		hosp_testresults_dbfile_t	append_file_perms create_file_perms rw_file_perms

Table 12. SELinux Profiles for doctors and pathologists.

In Table 12 can be seen which sandbox is assigned to each user and how these sandboxes restrict the access permissions over the objects. For example, when the *hospital_sys Application* is executed by a pathologist, the shell process with domain type *hosp_path_t* will transit into the domain *hosp_path_main_t*. This domain is only authorized to read files with type *hosp_tests_dbfile_t*, *hosp_testresults_dbfile_t* and access directories with type *hosp_testresults_dbdir_t*. If the pathologist try to access a directory or file with types not specified in this table, the access is denied.

Doctors are allowed to read files(allergies, treatments, medications, opd, tests, testresults) in the directories containing information that can be related to a specific patient. On the other hand, pathlogists can only access information

about prescribed tests in the “hosp_sys/tests” directory. In this example the doctor *hosp_doc* check the medication records for patient with id “123456” .



```
hosp_doc@gurpartap: ~  
hosp_doc@gurpartap:~$ hospital_sys doctor  
hosp_doc_r  
Enter patient id :123456  
Read Patients Details :1  
Read Medications :2  
Read Allergys :3  
Read Treatment History:4  
Read Test Results :5  
Update Allergys :6  
Update Medications :7  
Update Treatment :8  
Prescribe tests :9  
Exit : 0  
Enter your choice:2  
/hosp_sys/medics/123456  
Date: Jul 27 2014  
crocin 3 times  
Date: Jul 27 2014  
crocin 3times a day  
/hosp_sys/opd/123456  
Date: Jul 28 2014  
feber 100F  
Date: Jul 28 2014  
bp 120/80  
Read Patients Details :1
```

Sceenshot 1. Doctor checking patient medication records

If the pathologist tries to access the medical record of a particular patient or trick the application masquerading as a doctor, the application displays the message

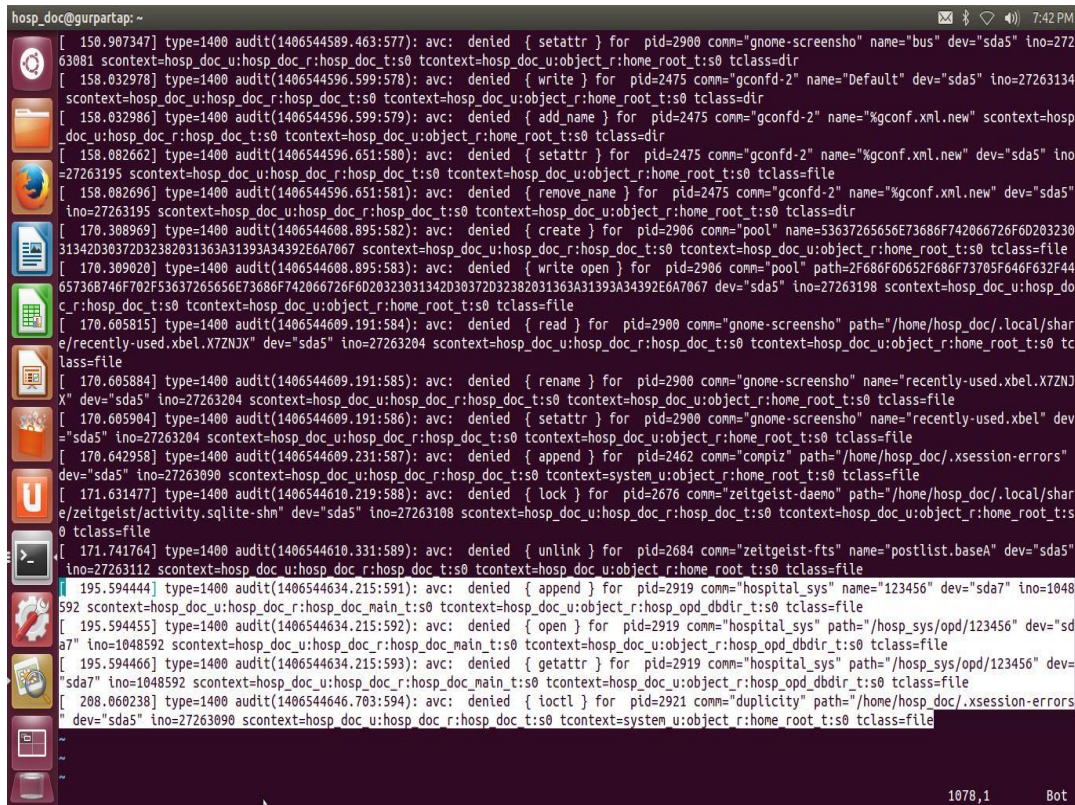


```
hosp_nur@gurpartap: ~  
hosp_nur@gurpartap:~$ id -Z  
hosp_nur_u:hosp_nur_r:hosp_nur_t:s0  
hosp_nur@gurpartap:~$ hospital_sys doctor  
hosp_nur_r  
No permission to execute doctor role  
hosp_nur@gurpartap:~$
```

Screenshot 2 . Nurse trying to access application with doctor role

Let us assume that the *hospital_sys* was modified and a backdoor was attached to application in order to allow a pathologist to specify the parameter “*backdoor*” and access patients information.

The *hospital_sys* while executed by pathologist will run under boundaries of domain “*hosp_path_main_t*”. The domain is authorized to access files of types *hosp_tests_dbfile_t*, *hosp_testresults_dbfile_t* only. Consequently if pathologist runs the application *hospital_sys* with back door as parameter and tries to access opd parameters of patient. SELinux will deny the access and creates an AVC message in the audit log as shown below:



Screenshot 3. AVC message logs showing denied access

5.5 Conclusion

This chapter described the proposed framework to implement and manage SELinux in HIS based on the use of SELinux Profiles. Also, in this chapter TE and RBAC mechanisms in SELinux were proposed as the preferred MAC mechanisms to prevent unauthorized disclosure and modification of information. To demonstrate the use of TE and RBAC mechanisms in SELinux, the concept of SELinux Profiles was introduced. SELinux Profiles restrict the authorized environment of subjects while accessing resources in the system. SELinux Profiles are constituted by roles, types, domains, TE rules, RBAC rules, conditional statements and Booleans. These components are coded into loadable policy modules for the easy management of the SELinux Policies. To demonstrate the functionality of SELinux Profiles, one of the scenarios used to

test the prototype developed for this research was illustrated. This scenario demonstrated that TE and RBAC mechanisms can effectively prevent intentional or unintentional attempts to access restricted resources.

In OSs that implements DAC mechanisms, the damage from compromised applications cannot be contained. SELinux is a preferred solution in order to minimize the effect of compromised applications using SELinux Profiles. SELinux Profiles are used in order to create sandboxes. Applications run inside the sandboxes, which restrict the access permissions to resources in the system.

Chapter 6

Conclusions

6.1 Research Findings

There are four main conclusions that can be inferred from this thesis and that were demonstrated by this research. These conclusions are listed as follows:

- **DAC mechanisms at the OS layer are not enough to satisfy security requirements in HIS.**

Healthcare organizations have security and privacy requirements from laws, regulations and ethical standards while storing, processing and transmitting their customers' healthcare information. In order to simplify the complexity while protecting the security and privacy of the information, appropriate information security services and mechanisms have to be implemented. To appropriately protect resources in a system from unauthorized accesses, access control mechanisms have to be implemented at the OS layer. Currently, most systems are constructed using OSs which implements DAC mechanisms. There are several issues in this type of OSs such as: access permissions are at the discretion of the users; the lack of domain separation; and the lack of enforcement of the least privileges principle.

In a DAC system, those authorized to access the resources can grant access permissions over these resources to others at their discretion. Access to resources in the system should not be at the discretion of the user. DAC systems manage only two levels of privileges which are the system administrator and the normal user. The existence of the system administrator layer is a clear example of how DAC systems do not enforce the least privilege principle.

Those who have access to the system administrator level, have full control over all resources in the system. Users in HIS, have to be restricted to the least privileges required to achieve their job functions. The lack of domain separation in DAC systems, allows applications to run without boundaries. If the application is compromised the damage cannot be contained. Therefore, in order to provide support from the OS layer and satisfy security requirements in HIS, it is necessary to use an OS which implements MAC mechanisms.

- **SELinux is a viable approach to provide MAC at the OS layer and aid to protect the security and privacy in HIS.**

SELinux is a recommended viable approach to aid in the protection of resources in HIS due to: its flexible and high granular MAC mechanisms; the increasing use of Linux; the importance of Open Solutions in future HIS, and the increasing number of tools and technologies to simplify the management of SELinux policies.

SELinux provides a flexible architecture that allows the enforcement of different security policies for different purposes at the OS layer. This is possible through a clear separation of policy enforcement and policy decision making which can be achieved with the use of the Flask architecture. TE and RBAC in SELinux are also important access control features that provide great benefits to the information system. The use of TE in SELinux provides fine-grained access controls and the possibility to enforce domain separation. The type of RBAC provided by SELinux can be used to simplify the user management task in large scale systems. Another important feature in SELinux is that rules are not hard-coded hence the system can be configured according to specific security requirements of organizations.

SELinux also provides other features such as orthogonal user identifiers, conditional policies and loadable policy modules. The use of orthogonal UIDs helps to provide better accountability of actions. Conditional Policies and loadable policy modules help to improve the implementation and management of SELinux Policies.

SELinux is part of the Open Solutions specified by Goldstein et al. (2007), which are needed to provide better HIS in the future. Linux is the main representative OSS solution. Linux has certain characteristics that make it desirable for organizations over proprietary software. Some of these characteristics are: its free availability, mainstream applications, user-friendly GUI, reliability of code and it is secure against most viruses and worms. Linux is also becoming one of the first freely available OSs which enforces MAC mechanisms due to the introduction of SELinux.

SELinux has already been proposed as a solution to provide MAC at the OS layer in HIS. However, because of the speed in which SELinux is changing, those researches are now out of date. Also, those researches did not provide a modern way in which SELinux can be implemented and managed. Consequently, this research proposes a modern framework to implement and manage SELinux based on the use of SELinux Profiles.

- **TE and RBAC in SELinux are a preferred solution to satisfy security requirements in HIS.**

In the past OSs that provides MAC at the OS layer were based on MLS. OSs that implements MLS are based on the Bell-LaPadula model, which is mainly dedicated to protect the confidentiality of classified information. These systems are considered to be very inflexible and unsuitable for commercial

organizations and also for HIS. Therefore, TE and RBAC in SELinux are a preferred solution over those systems that provide MLS.

TE in SELinux provides a high level of granularity which can aid in the protection of resources in HIS. In TE no access is allowed by default thus every access has to be explicitly authorized using TE rules. In this way, applications can be restricted to resources in a very granular way. This characteristic allows the implementation of domain separation and the creation of sandboxes. If subjects operate outside of their normal behavior and try to access unauthorized resources, SELinux denies the access since it is outside of the boundaries of the sandbox, that is, out of the authorized permissions for the domain.

RBAC in SELinux has been recognized as able to simplify the user management tasks. If the system administrator wants to revoke a domain which a group of users can access, the system administrator would only have to revoke access permissions to the role and not to individual users.

Enforcement of the least privilege principle is also achieved through the use of RBAC and TE. Roles are restricted to a specific set of domains. These domains are restricted to the least privileges required to achieve their tasks. Consequently, users are restricted to those privileges of the domains authorized to his/her role.

To demonstrate the use of TE and RBAC mechanisms in SELinux, the concept of SELinux Profiles was introduced. SELinux Profiles restrict the authorized environment of subjects while accessing resources in the system. SELinux Profiles are constituted by roles, types, domains, TE rules, RBAC rules, conditional statements and Booleans. These components are coded into loadable policy modules for the easy management of the SELinux Policies.

To demonstrate the functionality of SELinux Profiles, one of the scenarios used to test the prototype developed for this research was illustrated. The *hospital_sys* scenario was described and its security requirements satisfied. This scenario demonstrated that TE and RBAC mechanisms can effectively prevent intentional or unintentional attempts to access restricted resources. Also, the use of conditional policies was demonstrated to be a useful feature during emergency situations.

- **Application layer security alone is not enough to satisfy security requirements in HIS.**

Computer systems have to be constructed with the support from the underlying OS. MAC mechanisms in the OS layer can help to prevent attacks or minimize the damage from compromised applications. Viruses, Worms and Trojans are attacks commonly used to compromise applications and to access restricted resources in the system. In OSs that implements DAC mechanisms, the damage from compromised applications cannot be contained. Therefore, MAC mechanisms at the OS layer are a preferred solution in containing the damage from compromised applications.

SELinux is a preferred solution in order to minimize the effect of compromised applications using SELinux Profiles. SELinux Profiles are used to create sandboxes for applications running on behalf of specific users. Applications run inside the sandboxes, which restrict the access permissions to resources in the system. This behavior was demonstrated describing an attack scenario using the *hospital_sys Application*. This attack scenario was part of the test to which the prototype of this research was submitted. The scenario demonstrated that SELinux can effectively contain the damage from compromised applications.

This was achieved through the creation of sandboxes for the *hospital_sys Application* while running on behalf of specific users.

6.2 Future Work

The work done in this research was to explore use of SELinux to provide Hierarchical Role based access control for a Healthcare system.

In future research, work needs to be done for delegation of duties between roles based on some conditional Booleans in SELinux. The current research focused on security of data stored locally on a filesystem. The future research should extend to secure data lying on a distributed file system like Hadoop file system.

References

1. Anderson, J. P. (1972). Computer Security Technology Planning Study, Volume II. Retrieved 16 April, 2008, from <http://csrc.nist.gov/publications/history/ande72.pdf>
2. Bacon, J., Moody, K., & Yao, W. (2003). Access Control and Trust in the Use of Widely Distributed Services. *Software: Practice and Experience*, 33(4), 375394.
3. Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghghat, S. A. (1995). Practical domain and type enforcement for UNIX. *Proceedings of the 1995 IEEE Symposium on Security and Privacy*. Oakland, CA, USA.
4. Daswani, N., Kern, C., and Kesavan, A. (2007). *Foundations Programmer Needs to Know*. Berkeley, CA: Apress.
5. Ferraiolo, D.F., and Kuhn, R. (1992). Role-Based Access Control. *Proceedings of the 15th National Computer Security Conference*. Gaithersburg, Maryland, USA.
6. Ferraiolo, D. F., Kuhn, D. R., and Chandramouli, R. (2003). *Role-Based Access Control*. Norwood: Artech House.
7. Hu, J., & Weaver, A. C. (2004). Dynamic Context-Aware Access Control for Distributed Healthcare Applications. *Proceedings of the First Workshop on Pervasive Security, Privacy and Trust*. Boston, MA, USA.
8. Liu, V., Caelli, W., May, L., Croll, P., and Henricksen, M. (2007a). Current Approaches to Secure Health Information Systems are Not Sustainable: an Analysis. *Proceedings of MEDINFO 2007*. Brisbane, QLD, Australia.
9. Liu, V., May, L., Caelli, W., and Croll, P. (2007b). A sustainable approach to security and privacy in Health Information Systems. *Proceedings of the 18th*

Australasian Conference on Information Systems. Toowoomba, QLD, Australia.

10. Loscocco, P. C., Smalley, S. D. (2001). Meeting Critical Security Objectives with Security-Enhanced Linux. Proceedings of the Linux Symposium 2001. Ottawa, Canada

11. Nicola Zinnonan tutorial.

https://svn.win.tue.nl/viewvc/security_public/teaching/dtm/Slides/07-RBAC.pdf

12. Ravi S. Sandhu “ Role-Based Access Control “

13. Sandhu R. “Issues in RBAC”, 1st Workshop on Role-based Access Control, p. 21-24, 1995.

14. Sandhu R. et. al. “Role-based Access Control Models”. IEEE Computer, 29(2):38-47 February 1996

15. Sandhu R., Ferraiolo D. and Kuhn R. “The NIST Model for Role-Based Access Control.

16. Tolone, W., Ahn, G. J., Pai, T., and Hong, S. P. (2005). Access Control in Collaborative Systems. ACM Computing Surveys, 37(1), 29-41.